LB59xx Programming Guide

Programming Reference Guide for LB59xxA/L

LadyBug Technologies LLC

2/4/2020

Revised 7/21/21

This document is to be used as a reference in programming the LB59xx series of USB power sensors including but not limited to the LB5908A/L, LB5912A/L, LB5918A/L, LB5926A/L and LB5940A/L.

Contents

Overview

This manual serves two purposes.

1. It provides a short basic tutorial on programming the LB59xx series of power sensors.
2. It is a programming and SCPI[4] command reference for the LB59xx instruments.

Sample command sequences are included for many commands throughout the manual. In general, the sequences were created using the LadyBug Interactive IO application.

To replicate and test these sequences we recommend using the LadyBug Interactive IO. The Ladybug Interactive IO application is part of the default installation process. This application provides additional features such as macros and repeating commands that are particularly useful in demonstrating various features.

Another benefit to using the LadyBug Interactive IO is that the application installation bypasses the need to download and install large libraries and applications. That said, the LB59xx sensors are command compatible with the Keysight[1] U2000[5] series of sensors[2]. Furthermore, they conform to the USBTMC standard and can be controlled using the applications and libraries provided by Keysight or National instruments[3].

Introduction

In automated test environments power sensors are typically controlled or programmed using SCPI[4] commands. The LB59xx sensors are no different. The LB59xx sensors use normal or standard SCPI commands such as "*IDN?", "FETCH?", "FREQ", etc.

And, as previously stated the LB59xx sensors are command compatible[2] with the Keysight[1] U2000[5] sensors. So, if you know how to program a Keysight[1] U2000[5] you know how to program the LadyBug LB59xx.

Communication and control of the LB59xx sensors can be accomplished using the industry standard USBTMC[6] protocol. The LB59xx sensors support this USB protocol. These sensors also support SCPI over HID with no performance penalty. The additional protocol support is provided in order to minimize the installation load for the user at startup.

The LB59xx sensors natively support SCPI over USB. It uses two protocols to do this. First is USBTMC. Secondly is a USB HID protocol. The commands in both cases are identical. A review of all the interfaces and protocols supported by the LB59xx sensors follows. The sensor supports two

[1] Keysight is a trademark of Keysight Technologies. Keysight is not associated with LadyBug in any way
[2] U2000 compatibility claims are made solely by Ladybug Technologies
[3] National Instruments and NI are trademarks of the National Instruments company
[4] SCPI is an acronym for Standard Commands for Programmable Instruments
[5] U2000A is a power sensor product manufactured by Keysight Technologies
[6] USBTMC is an acronym for USB Test and Measurement Class

physical interfaces. One is USB. The USB interface is present on all sensors. The second physical interface is optional. Its availability is easily noted by the additional ribbon cable. If you don't see a ribbon cable coming out the back of the sensor (next to the USB cable) then the optional communication path is not present in this sensor.

- Physical USB interface connector supports two protocols:
    - USBTMC protocol –The standard protocol used to communicate with USB instruments. This protocol requires the installation of either KeySight I/O libraries or NI (National Instruments) Max.
    - HID (or **H**uman **I**nterface **D**evice) protocol – HID is fundamental to Windows, Mac IOS and LINUX and even some embedded solutions. HID is the most ubiquitous of all USB protocols. The ubiquity of HID mitigates driver support issues.
- Physical 10 Wire Interface (two protocols). This interface is an optional 10 wire ribbon cable with a 2x5, 0.05" spaced plug that is normally connected to the user's controller board. This connector exits out the back of the unit adjacent to the USB connector. This ribbon cable supports:
    - SPI or Serial peripheral interface
    - I2C or Inter-integrated circuit interface

The LB59xx sensors can be programmed using a single interface during any given session. A session as defined here, is the time between issuing a reset (*RST) commands or cycling the power. You are free to choose any interface with each new session. So you can communicate using USBTMC, USB-HID and SPI or I2C via the ribbon cable assuming the option was purchased.

Having said this, it is possible, within a session to communicate via USBTMC then HID SICL or SPI/I2C and back. Although possible, it is not generally advised. In other words, you will find anomalies. If you choose to switch between interfaces in a single session it should be done with care and understanding. *However, this is neither recommended nor supported.*

An Introduction to Programming LB59xx Sensors

This section is intended to provide a brief introduction to SCPI programming. This is not a comprehensive review of SCPI. There are downloadable standards for that purpose. Instead, the aim here is to provide information relevant to control and programming the LB59xx sensors. This assumes some level of expertise.

Sample code and Following Along

In this manual there are several examples of commands and command sequences. You can follow along in one of two ways:

1. Use the Interactive LadyBug Interactive IO application shipped with the sensor. This should be installed with the software. This interface uses a USB HID protocol but in all cases uses the same commands in the same form.

2. Using the Keysight Connection Expert or NI-Max application. This uses USBTMC protocol.

In any case the process is the same. Select the instrument you wish to communicate with and then begin sending commands by typing them in. For commands that return values, ensure "Send Command…" and "Auto Query" are checked. After you've entered a command press "Return". If the command contains a "?" then "Get Buffer" will be clicked automatically. Alternately, you can click the "Get Buffer" button.

States of Operation

Normal SCPI (USBTMC or USB HID)

This is the default state. This is what most users expect. In this state the LB59xx power sensors make average power measurements as a direct consequence of SPCI commands. The normal sequence is to send a measurement command then query the sensor. This is the default state and is the primary focus of this manual.

RO or Recorder Out

In this state the recorder output port provides a voltage proportional to the measured power in Watts. The output voltage is available at the trigger out (or TI/RO) SMB connector on the adjacent to the USB connector on the back of sensors rear bulkhead.

UOP

The LB59xx has a third optional state. This state is called UOP or unattended operation. If present, UOP is available through all interfaces and protocols and ports previously mentioned.

In this mode the LB59xx is first setup to make measurements. Then a SCPI command is issued causing the sensor to enter UOP state. Upon entering UOP state the sensor begins making and recording time-stamped measurements in on-board non-volatile memory. This continues for as long as power is applied, free memory is available and UOP is active.

UOP may be setup to resume recording measurements if power is lost and later restored.

To access the UOP results you must exit UOP. This is done by issuing a SCPI command. Then, UOP specific SCPI commands can be used to access the data. Alternatively you may find the PMA-12 application a more convenient means of accessing this data.

LB59xx Persona

The Persona feature allows the user to setup the LB59xx to report ID information of the user's choosing. This allows the sensor to respond appropriately to *IDN? by returning any firmware, version, model string, manufacturer and serial number the you desire

In short, the information reported in response to *IDN can be modified to suit the user's preference. Again, this allows the LB59xx to masquerade as any sensor (including another LB59xx sensor). This can be useful in testing software (for purposes of compatibility) or delaying or completely avoiding software driver maintenance.

The follow outlines the procedure for enabling the Persona feature:

1. Connect the sensor you wish to emulate (e.g. U2000A)
2. Issue a *IDN? command to the sensor
3. Faithfully and completely record the sensors response. Pay special attention to lower and upper case letters and spaces.
4. Disconnect the sensor you wish to emulate
5. Connect the LB59xx sensor
6. Start the Persona application that is installed during the default installation.
7. Set LB59xx sensor up using the persona application and the strings you
8. Disconnect and reconnect LB59xx sensor (to reboot the firmware)
9. Issue a *IDN? command to the LB59xx
10. Faithfully and completely record the LB59xx response. Compare the LB59xx response to the sensor to emulate response in step 3.
11. If the ID information in step 3 and step 9 do not match exactly, repeat the process

If they match, you can connect the LB59xx sensor to the system and run a test using the software and driver you've been using. In this way you can readily determine if the LB59xx is an effective drop in replacement. In summary, the LB59xx sensors can be setup to report alternative ID information for any purpose you desire.

An Introduction to SCPI

The acronym, SCPI, stands for Standard Commands for Programmable Instruments. There are a number of online documents detailing the SCPI standard. This document is not intended to be a general purpose, broad based description of SCPI. Instead it is a very brief, purpose built summary of SCPI and the LB59xx:

- The allowable SCPI characters are:
    - Special characters
        - *
        - ?
        - .
        - ,
        - +
        - -
        - :
    - " " or space
    - A-Z
    - a-z
    - 0-9

    - SCPI is case insensitive so that A-Z = a-z
- All communication (or commands) sent to the sensor are composed of one or two parts. These parts are the command and the parameters.
    - Commands are composed of one or more headers. A header is 3-12 characters in length. Headers can be concatenated using a colon.
        - A single header – `FREQ?`
        - Concatenated headers – `SENS:FREQ:CW`
    - Parameters are limited to floating point numbers, integers, Boolean and text. The number and types of parameters are specific to each command. Parameters are concatenated by commas
        - A single parameter 10
        - Multiple parameters 10, 3
        - Another example of multiple parameters 10.0e6, 3.0
    - Commands are separated from parameters by a single space. The following command will set the frequency to 1.02GHz. Note that the command is FREQ and the parameter is 1.02E+9 and they are separated by a space.

      FREQ 1.02E+9

      Note: Multiple spaces between the command and parameters are treated as a single space.

- SCPI commands are case insensitive. So a LB59xx power sensor sees all of the following commands as equivalent:
    - `FREQ?`
    - `freq?`
    - `fReQ?`
    - `freq?`
- This manual uses the most common conventions for expressing SPCI commands: The conventions are:
    - Brackets [] identify optional headers of a command. Brackets may be nested. Any header designated as optional may be omitted. Consider the following definition of a command:

      `[SENSe[1]:]FREQuency[:CW|:FIXed] <numeric_value>`

      Given this definition the following commands are equivalent:

`FREQUENCY 100MHZ`	omitting all optional headers
`SENSE1:FREQUENCY 100MHZ`	including the **`SENSE[1]`** header
`SENSE:FREQUENCY 100MHZ`	including the **`SENSE`** , but omitting **`[1]`**
`SENSE:FREQUENCY:CW 100MHZ`	including and omitting headers
`SENSE:FREQUENCY:FIXED 100MHZ`	

- A vertical line | is used in the definitions to delineate mutually exclusive portions. All of the following are acceptable and equivalent. In these examples the focus is on the [:CW|:FIXed] portion of the command:

  `[SENSe[1]:]FREQuency[:CW|:FIXed] <numeric_value>`

▪ **`FREQ:CW`**	selecting the **`[:CW]`** option
▪ **`FREQ:FIXED`**	selecting the **`[:FIXed]`** option
▪ **`FREQ`**	omitting the optional selection inside brackets

- Upper and lower case letters in a ***definition*** delineates the short form (or abbreviation) and the long form of a header. The upper case letters indicate the short or abbreviated form of a header. The entire header (upper and lower case) represent the long form of the header. Consider the following command definition:

  `[SENSe[1]:]FREQuency[:CW|:FIXed] <numeric_value>`

  Given the previous command definition, the following are equivalent:

`FREQUENCY 100.0E+6`	uses a long header, excludes all options
`SENSE1:FREQUENCY 100MHZ`	includes the optional **`[1]`**
`SENSE:FREQUENCY 100.0e+6`	omits optional **`[1]`** and **`[:CW|:FIXed]`**
`SENSE:FREQUENCY:CW 100MHZ`	selects the optional **`[:CW]`**
`SENSE:FREQUENCY:FIXED 100MHZ`	selects the optional **`[:FIXed]`**
`FREQ 100MHZ`	uses short form headers, excludes all options
`SENS:FREQ 100MHZ`	uses short headers, omits **`[:CW|:FIXed]`**
`SENS:FREQ:CW 100MHZ`	uses short headers, includes **`[:CW]`**
`SENS:FREQ:FIX 100MHZ`	uses short headers, includes **`[:FIX]`**

  Remember, SCPI is case insensitive, so the following are equivalent …

`FREQ 100MHZ`	uses short form headers, excludes all options
`freq 100MHZ`	very common form of this command
`SENS:frEQ 100MHZ`	short form with upper and lower case
`SEns:FREQ:CW 100MHZ`	
`sens:freq:cw 100MHZ`	
`sens:FREQ:cw 100mHZ`	
`SENS:FREQ:fix 100MHZ`	
`sense:frequency:cw 100mhz`	

  …and so on.

- In some cases units may be appended to a numeric value. However, this is always specific to the command. For instance:

▪ `FREQ` **`1.3MHZ`**	includes the units
▪ `FREQ` **`1.3E+6`**	does not include the units
▪ `CONF` **`-20DBM,`** `2`	includes units where appropriate

- `CONF -20, 2` does not includes units

The Basics of Making Power Measurements

This section is a brief tutorial. It provides an overview on how to make basic power measurements using the LB59xx sensors. The examples use the Ladybug Interactive IO application. Finer points on command syntax have been omitted. This section is followed by the main command reference section.

The primary purpose of the LB59xx USB sensors is to make average power measurements using the average detector. And that is the focus of this tutorial. Three commands are used to make average power measurements. They are:

- **`FETCH?`** – affords the user the greatest control and is the most complex.
- **`READ?`** –is equivalent to an **`INIT`** command followed by **`FETCH?`**. It offers less control but is easier to use than **`FETCH?`** At the same time **`READ?`** provides more control than **`MEASURE?`** and is more complex than **`MEASURE?`**
- **`MEASURE?`** –is equivalent to a **`CONFIGURE`** command followed by a **`READ?`** Also, **`MEASURE?`** provides the user with least control over the measurement process. But acquiring a valid measurement requires the least knowledge.

The short form of these commands will generally be used in this section. The short forms are **`FETC? READ?`** and **`MEAS?`** respectively. And, it is true that FETC? can do anything that READ can do. And READ can do anything that MEAS? can do. The reverse is not the case. In other words, MEAS inherently and deliberately lacks the flexibility of READ?. And READ? Inherently and deliberately lacks the flexibility of FETC? Each command can produce measurements. The reason for using one command instead of another has to do with degree of control desired by the user.

MEAS?

We'll start by demonstrating the **`MEASure?`** or **`MEAS?`** command. The following command sequences are copied directly from the LadyBug Interactive IO application. This application is installed by default. Proceed as follows:

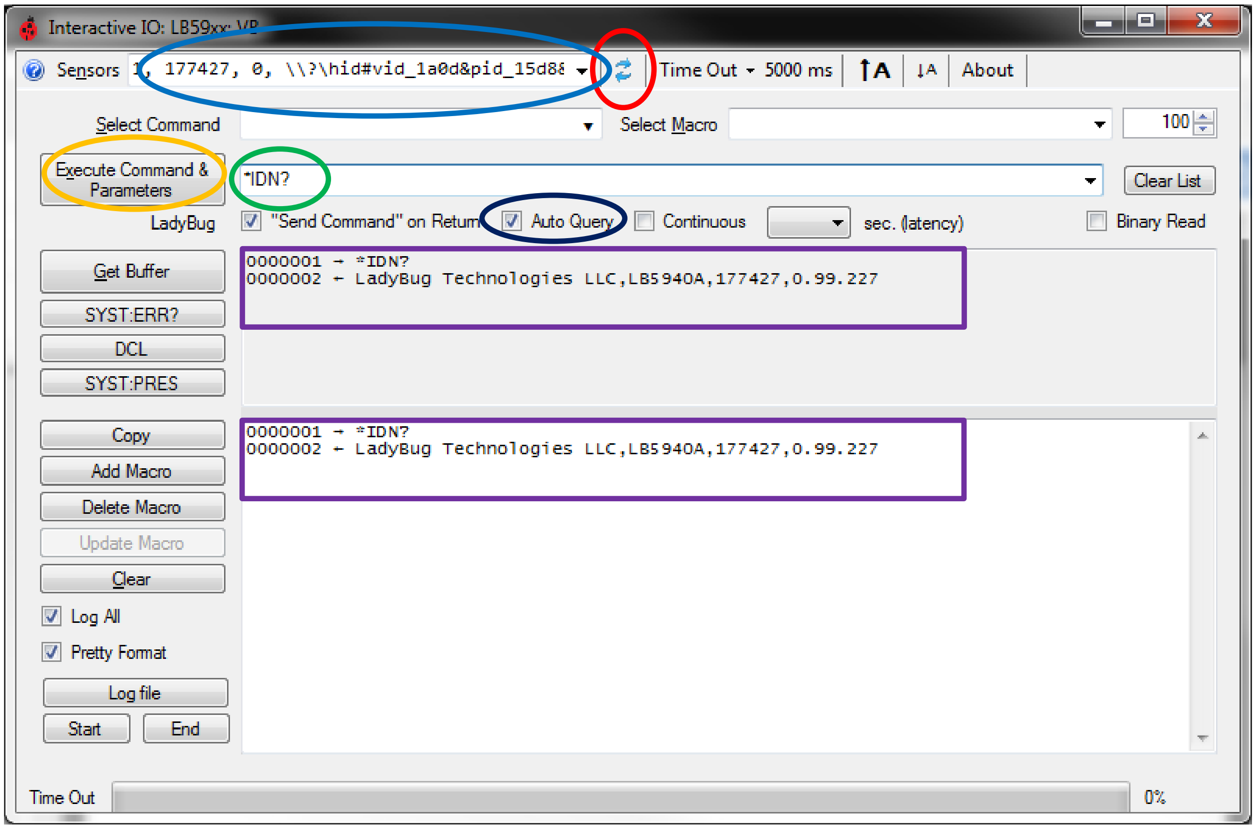
1. Connect the sensor to your PC and wait a few seconds for the green LED to go to a steady dim green color
2. Open the LadyBug Interactive IO application. To start the interactive IO application:

   Windows button-> All Programs->LadyBug Technologies LLC->Tools->Interactive IO

3. Connect the sensor to a CW source. Set the source frequency to 1GHz and the source amplitude (level) to -20dBm. Enable the RF output.

(Refer to the graphic below for additional information)

4. Click the instrument list refresh button (circled in *red below*)
5. Find and select your sensor from the sensor list using the serial number (blue circle)
6. Enter the **`*IDN`**? command in the command text box as shown (green circle)
7. To execute the **`*IDN?`** command simply press return after the command - or click the Execute Command & Parameters button (orange or yellow)



```
1, 177427, 0, \\?\hid#vid_1a0d&pid_15d8
```

```
*IDN?
```

```
0000001 → *IDN?
0000002 ← LadyBug Technologies LLC,LB5940A,177427,0.99.227
```

```
0000001 → *IDN?
0000002 ← LadyBug Technologies LLC,LB5940A,177427,0.99.227
```

8. Examine the results text box (upper purple rectangle) and the command log text box (lower purple rectangle). Text can be copied from the log text box and pasted into other applications. The sequences displayed in this manual are copied in this fashion.

Note the string returned. This string was returned is because the "Auto Query" (dark blue) was checked. With Auto Query enabled, the "Get Buffer" button is clicked anytime an executed command contains a question mark.

If "Auto Query" was not checked you would have to click the "Get Buffer" button to see the return (after clicking the Execute Command & Parameters button). It is generally very convenient to leave "Auto Query" checked and this is the default state.

The returned string contains the identification information of the sensor including the manufacturer, model number, serial number and version of firmware. This command is often the first command issued in test applications. It is used to confirm the configuration information of the test station and to associate subsequent measurement values with a particular instrument.

Now enter a *RST command in the command text box and press return. This resets the sensor to a known state. Note that *RST and power on states can and are somewhat different.

You should have the sensor connected to a CW source (RF output enabled) with the frequency set to 1GHz and power level to -20dBm. You can now enter the **MEAS**? command in the command text box. Press return. The log text box should now contain the following text as a result of these commands (clearly, your measured value will differ somewhat):

```
0000001 → *IDN?
0000002 ← LadyBug Technologies LLC,LB5940A,177427,0.99.227
0000003 → *RST
0000004 → MEAS?
0000005 ← -2.02798295E+01
```

Now enter a READ? command and press return. The log should now contain the following text:

```
0000001 → *IDN?
0000002 ← LadyBug Technologies LLC,LB5940A,177427,0.99.227
0000003 → *RST
0000004 → MEAS?
0000005 ← -2.02798295E+01
0000006 → READ?
0000007 ← -2.02804834E+0
```

Notice that MEAS? and READ? returned close to the same value. Now we'll use FETCH? But given the current settings you'll need to enter a command to tell the sensor to start a measurement. First execute and INIT. Then execute FETCH? This should result in the following:

```
0000001 → *IDN?
```

```
0000002 ← LadyBug Technologies LLC,LB5940A,177427,0.99.227
0000003 → *RST
0000004 → MEAS?
0000005 ← -2.02798295E+01
0000006 → READ?
0000007 ← -2.02804834E+01
0000008 → INIT
0000009 → FETCH?
0000010 ← -2.02833832E+01
```

Again, another measurement that is very close in value to previous measurements. This demonstrates that the three commands can measure the same signals and get the same results. So, how are these commands different?

Let's start by examining MEAS? This command automates much of the measurement. It starts by setting up certain parameters. It then samples the incoming measurement and, for a given resolution, it determines how long to measure. It also makes sure that the measurement starts when the command is sent. To demonstrate this, execute the following command sequence. But start by setting your source power to -10dBm.

```
*RST
INIT:CONT?                check the state of INIT:CONT
INIT:CONT 1               enable INIT:CONT
INIT:CONT?                ensure we've enabled INIT:CONT
AVER:COUN:AUTO?           check the state of AVER:COUN:AUTO (auto averaging)
AVER:COUN:AUTO 0          disable AVER:COUN:AUTO
AVER:COUN:AUTO?           ensure we've disabled AVER:COUN:AUTO
MEAS?                     make a measurement
INIT:CONT?                check the state of INIT:CONT
AVER:COUN:AUTO?           check the state of AVER:COUN:AUTO (auto averaging)
```

If you did this correctly (and set the power level to -10dBm) you should get a log that is similar to the following (again, copied from the log text box, ignore line numbers):

```
0000019 → *RST
0000020 → INIT:CONT?
0000021 ← 0
0000022 → INIT:CONT 1              enable init:cont
0000023 → INIT:CONT?               check to see if did we succeeded
0000024 ← 1                        yes, INIT:CONT is enabled
0000025 → AVER:COUN:AUTO?
0000026 ← 1
0000027 → AVER:COUN:AUTO 0         disable auto averaging
0000028 → AVER:COUN:AUTO?          check to see if did we succeeded
0000029 ← 0                        yes, auto averaging is disabled
```

`0000030 → MEAS?`	
`0000031 ← -1.02138776E+01`	
`0000032 → INIT:CONT?`	check INIT:CONT, should still be enabled
`0000033 ← 0`	???
`0000034 → AVER:COUN:AUTO?`	check auto averaging, should still be disabled
`0000035 ← 1`	???

So what is going on here? Well, MEAS? does more for you. Two of the various things it does is it disables INIT:CONT and enables AVER:COUN:AUTO. This ensures a good measurement result. If you want more control consider READ? or FETCH? However, you can "tune" up the MEAS? command to some degree.

The primary control you have with MEAS? is trading off resolution for measurement speed. In the examples above we issued a simple "MEAS?" command using the default configuration. It turns out that you can tell MEAS? to speed things up. But this speed up comes with some loss in settling time. The complete definition of the MEAS? command is:

```
MEASure[1][:SCALar][:POWer:AC]? [<expected_value>[,<resolution>[,<source list>]]]
```

Using short headers and omitting some optional headers yields a command like:

```
MEAS? <expected_value>,<resolution>
```

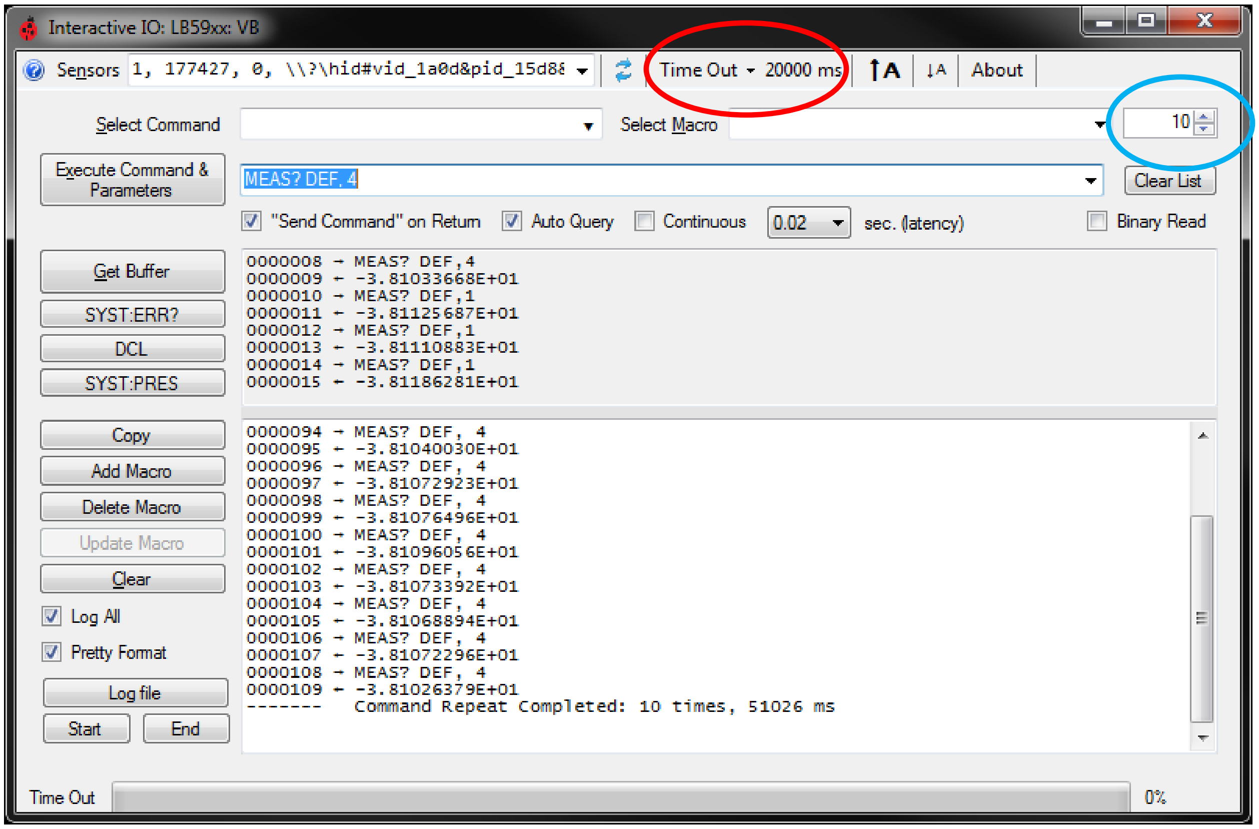
To demonstrate the impact of resolution on measurement time...start by ensuring the time out is set to 20,000ms (red oval in following graphic). Connect the sensor to a 1GHz, -38dBm signal. Then execute the following sequence.

```
*RST
MEAS? DEF, 2
MEAS? DEF, 3
MEAS? DEF, 4
```

Notice that the value DEF is used for <expected value>. DEF is short for default. In this case it means don't change the expected value. Expected value is used by the U2000 but it is not used the LB59xx.

Command	Time
MEAS? DEF, 2	0.2s
MEAS? DEF, 3	0.8s
MEAS? DEF, 4	5.1s

These times were times were determined using the repeat function in the LadyBug Interactive IO application. To use this feature first set the repeat count to 10 (blue oval below). Then enter the command and then press Ctrl-T instead of return. The command should repeat 10 times. At completion, the count and elapsed time are displayed. If you set repeat count to 1 you get (about) the same answers.



```
1, 177427, 0, \\?\hid#vid_1a0d&pid_15d8
MEAS? DEF, 4
```

```
0000008 → MEAS? DEF,4
0000009 ← -3.81033668E+01
0000010 → MEAS? DEF,1
0000011 ← -3.81125687E+01
0000012 → MEAS? DEF,1
0000013 ← -3.81110883E+01
0000014 → MEAS? DEF,1
0000015 ← -3.81186281E+01
```

```
0000094 → MEAS? DEF, 4
0000095 ← -3.81040030E+01
0000096 → MEAS? DEF, 4
0000097 ← -3.81072923E+01
0000098 → MEAS? DEF, 4
0000099 ← -3.81076496E+01
0000100 → MEAS? DEF, 4
0000101 ← -3.81096056E+01
0000102 → MEAS? DEF, 4
0000103 ← -3.81073392E+01
0000104 → MEAS? DEF, 4
0000105 ← -3.81068894E+01
0000106 → MEAS? DEF, 4
0000107 ← -3.81072296E+01
0000108 → MEAS? DEF, 4
0000109 ← -3.81026379E+01
-------   Command Repeat Completed: 10 times, 51026 ms
```

The following log the command meas? def, 4 (note the use of lower case) was executed 10 times. You can see that the resulting total time is consistent with the statement above. This concludes the overview of MEAS?

```
-------   Command Repeat = 10
0000041 → meas? def, 4
0000042 ← -3.83283989E+01
0000043 → meas? def, 4
0000044 ← -3.83283533E+01
0000045 → meas? def, 4
0000046 ← -3.83248385E+01
0000047 → meas? def, 4
0000048 ← -3.83234973E+01
0000049 → meas? def, 4
0000050 ← -3.83262600E+01
0000051 → meas? def, 4
0000052 ← -3.83264578E+01
0000053 → meas? def, 4
0000054 ← -3.83249416E+01
0000055 → meas? def, 4
0000056 ← -3.83229953E+01
0000057 → meas? def, 4
0000058 ← -3.83262596E+01
0000059 → meas? def, 4
0000060 ← -3.83269416E+01
-------   Command Repeat Completed: 10 times, 51073 ms
```

READ?

If you require additional control over the measurement process then READ? may provide an answer. The most important additional control provided by READ? is direct control over the measurement averaging time or more commonly referred to as averaging. The following sequence demonstrates one possible way of exercising this additional control.

```
0000485 → *RST                  place the sensor in a known state
0000486 → AVER:COUN:AUTO?       request the state of auto averaging
0000487 ← 1                     auto averaging is enabled
0000489 → AVER:COUN:AUTO 0      disable auto averaging
0000490 → AVER:COUN:AUTO?       were we successful in disabling auto averaging?
0000491 ← 0                     yes, auto averaging is disabled
0000492 → AVER:COUN?            what is the current averaging count?
0000493 ← +4                    it's 4…your system may show a different number
0000494 → AVER:COUN 10          set it to 10
0000495 → AVER:COUN?            make sure the change took…
0000496 ← +10                   …and it did
0000497 → READ?                 see if we can measure
0000498 ← -3.81234644E+01       yes, we can.

-------   Command Repeat = 10   repeat the measurement (10 averages) 10 times
0000499 → READ?
0000500 ← -3.81332656E+01
0000501 → READ?
0000502 ← -3.81370566E+01
0000503 → READ?
0000504 ← -3.81419867E+01
0000505 → READ?
0000506 ← -3.81278593E+01
0000507 → READ?
0000508 ← -3.81283168E+01
0000509 → READ?
0000510 ← -3.81333663E+01
0000511 → READ?
0000512 ← -3.81288760E+01
0000513 → READ?
0000514 ← -3.81322877E+01
0000515 → READ?
0000516 ← -3.81239859E+01
0000517 → READ?
0000518 ← -3.81386944E+01
-------   Command Repeat Completed: 10 times, 4022 ms
                                4.022 seconds for 10 measurements with average count = 10

0000519 → AVER:COUN?            make sure it is still…
0000520 ← +10                   …10 averages
0000521 → AVER:COUN 250         set averaging to 250 (25 times as much averaging)
0000522 → READ?                 make a measurement?
```

```
0000523 ← -3.81281075E+01          yes

-------   Command Repeat = 10       repeat the measurement (250 averages) 10 times
0000524 → READ?
0000525 ← -3.81298139E+01
0000526 → READ?
0000527 ← -3.81285339E+01
0000528 → READ?
0000529 ← -3.81259109E+01
0000530 → READ?
0000531 ← -3.81237059E+01
0000532 → READ?
0000533 ← -3.81208487E+01
0000534 → READ?
0000535 ← -3.81277904E+01
0000536 → READ?
0000537 ← -3.81318585E+01
0000538 → READ?
0000539 ← -3.81304820E+01
0000540 → READ?
0000541 ← -3.81308281E+01
0000542 → READ?
0000543 ← -3.81277778E+01
-------   Command Repeat Completed: 10 times, 96233 ms
                                    10 measurements in 96.233 seconds, average count = 250
0000544 → AVER::COUN:AUTO?          and auto averaging is still disabled
0000545 ← 0
```

In the previous sequences a measurement with 10 averages took about 0.4 seconds. And 250 averages took 9.6 seconds each. If you take the ratio you see that it took 24 times as long to make a measurement using 250 averages vs 10 averages. With a little experimentation you can estimate the time to make a single measurement using READ? assuming that auto averaging is disabled.

In comparing the measurement times of this sequence to the sequences using MEAS? it is apparent that the setting of auto averaging is overridden (and enabled) anytime MEAS? is executed while READ? does not override this setting.

Finally, should you use READ? to manage the measurement time? That will be a function of your measurement requirements. Another source of control over the measurement time is the state of the Step Detection feature. Control over this feature is also available with MEAS? but its effect are more easily demonstrated with READ?

If step detection is enabled, the sensor monitors the measured value during the measurement process. And if the measured value changes more that 12% the measurement is restarted. To understand the impact, examine the following sequence:

```
0000580 → AVER:SDET?                check the state of the step detection feature
0000581 ← 1                         it is enabled
```

```
0000583 → AVER:COUN:AUTO?             how about auto averaging?
0000584 ← 0                           it is disabled
0000585 → AVER:COUN?                  average count?
0000586 ← +250                        250

-------   Command Repeat = 1          do a measurement with power set to -28dBm
0000587 → READ?
0000588 ← -2.81266043E+01
-------   Command Repeat Completed: 1 times, 9615 ms
                                      took a 9.6 seconds…about what we expected


-------   Command Repeat = 1          do another measurement, start a -38dBm. Then 5 seconds after
                                      starting the measurement change the value to -28dBm
0000589 → READ?
0000590 ← -2.81240338E+01
-------   Command Repeat Completed: 1 times, 14712 ms
                                      took a 14.7 seconds…not what we had before

0000591 → AVER:SDET?                  step detection is still on?
0000592 ← 1                           yes
0000594 → AVER:SDET 0                 turn it off
0000595 → AVER:SDET?                  is it off?
0000596 ← 0                           yes

-------   Command Repeat = 1          do another measurement, start a -38dBm. Then 5 seconds after
                                      starting the measurement change the value to -28dBm
0000601 → READ?
0000602 ← -3.13556068E+01
-------   Command Repeat Completed: 1 times, 9622 ms
                                      time is 9.6 (ok) but now the level is incorrect?

-------   Command Repeat = 1          do another measurement, don't change level this time
0000599 → READ?
0000600 ← -2.81280784E+01
-------   Command Repeat Completed: 1 times, 9618 ms
                                      Both time and level are correct
```

This sequence demonstrates that the state of step detection can affect both the reported value and/or the time it takes to return to measurement. In the second measurement in this sequence while the measurement step detection is enabled. And the level is changed while the measurement was underway.

While measuring, the sensor detected a change in level and automatically restarted the measurement process.

The purpose of Step Detection is to detect a change in power level during the measurement. It is there specifically for this purpose. If a change in the measured value occurs then the measurement process can be restarted – this results in a more settled value. So the first time it took about 9.6

seconds to return a value. The second time it took 14.7 seconds. The extra time was a direct result of step detection restarting the measurement process. Both times the same value was returned.

Subsequently, the two measurement sequence is repeated with step detection disabled. In this case you can see the time for both measurements was the same. But the measured value, in the case where power level changed, returned a value that was very different. This is because with step detection disabled, the measurement continued through the change in power. And this was done precisely because step detection was disabled. The question is, should you enable or disable step detection?

FETCH?

This is the last command we'll consider here. This command (FETCH? or FETC?) gives the user additional control over the process. As stated in the beginning of this tutorial, READ? is equivalent to an INIT command followed by a FETCH? command. If this is true, we should be able to issue an INIT followed by a FETCH? to yield a measurement. Connect the sensor to an RF source (1GHz and -15dBm). Then execute the following sequence of commands.

```
0000390 → *RST                      place the instrument in a known state
0000391 → INIT:CONT?                ensure INIT:CONT is disabled…
0000392 ← 0                         …it is
0000393 → INIT                      start a measurement
0000394 → FETCH?                    retrieve the result
0000395 ← -1.54978680E+01           Ok
```

This sequence starts with a reset. Again, this places the instrument in a known state for this sequence. Then the state of INIT:CONT (continuous triggering) is checked. It is turned off. An INIT command is then issued (starting the measurement process) followed by a FETCH? command. You can experiment with this yourself. So an INIT followed by FETCH? is equivalent to a READ? So, why have FETCH? Well demonstrate just one of the reasons.

FETCH? has a unique feature we haven't demonstrated as of yet. Remember, we said READ? is equivalent to an INIT plus a FETCH? But what if the sensor issued a new INIT as soon as the current measurement completed? It seems then it should be possible to continuously FETCH? To tell the sensor to issue and INIT all you have to do is send the command INIT:CONT 1. In essence, the sensor begins to behave like a voltmeter – except it is measuring RF power. This is also referred to as the free run mode.

To demonstrate examine the following sequence:

```
0000497 → *RST                      place the instrument in a known state
0000498 → INIT:CONT?                check the state of INIT:CONT…
0000499 ← 0                         …it's disabled
0000501 → INIT:CONT 1               enable INIT:CONT …
0000502 → INIT:CONT?                check the state of INIT:CONT…
0000503 ← 1                         …now it's enabled
0000504 → AVER:COUN 256             set averaging to 256 (about 10 seconds)
0000505 → FETCH?                    make a measurement…
0000506 ← -1.54586278E+01           comes right back
0000507 → FETCH?                    again…
0000508 ← -1.54586877E+01           comes right back
0000509 → FETCH?                    …
0000510 ← -1.54586709E+01           …
0000511 → FETCH?                    …
0000512 ← -1.54587870E+01           …

-------   Command Repeat = 5        now we'll make 5 measurements
```

```
0000513 → FETCH?
0000514 ← -1.54594488E+01
0000515 → FETCH?
0000516 ← -1.54594554E+01
0000517 → FETCH?
0000518 ← -1.54594590E+01
0000519 → FETCH?
0000520 ← -1.54594573E+01
0000521 → FETCH?
0000522 ← -1.54594561E+01
-------   Command Repeat Completed: 5 times, 135 ms
```

…that's fast. What's going on?

If you perform the sequence you can see the measurements return quickly with INIT:CONT enabled. This is because the sensor is constantly filling up a circular buffer and (when requested) returns a value based on the data in the circular buffer and then posts it. If you execute FETCH? command as demonstrated here you get the most recent value.

Unlike READ? which starts a measurement upon receipt of the command, FETCH? with INIT:CONT enabled, causes the sensor to generate a stream of INITs and their resulting measured values regardless of whether or not the measured values are retrieved. Essentially the sensor measurement mode is now in free run if TRIG:SOUR is also is set to IMM. Neither MEAS? or READ? can be used in this way.

To be clear, this is not the only difference between READ? and FETCH? However, it is an important difference. Hopefully these examples have given you basic information for making average power measurements using the LB59xx line of power sensors.

LB59xx Programming Reference

This is the beginning of the programming reference. Generally all commands sharing the same first header (e.g. SENSE or TRIGGER) are related and grouped together. Each individual command is detailed as to its syntax and usage. Interactions are often noted as well as some of the more common usage errors associated with the various commands.

The explanation of most commands is accompanied by sequences of commands and their return values. These sequences (and return values) were executed using the Ladybug Interactive IO. The results were copied directly from the Ladybug Interactive IO application into this document. The commands sequences can be repeated in other vendor's interactive applications that provide USBTMC support (such as National Instrument or Keysight IO libraries).

Measurement Commands

The following parameters are common to most measure commands.

Measurement Command Optional Parameters

The following applies to measurement commands. Some commands use these optional parameters to configure the instrument while other commands use them for comparative purposes. In these comparative cases, if the parameters passed in do not match current settings one or more errors are generated. When this comparative process generates errors the measurement process is halted.

Name	Description	Acceptable Values
<expected value> or Expected value	Indicates the expected power level. Not used by the LB59xx sensors. Numeric values are dBm or W depending on the unit setting.	-60dBm to +26dBm 1.000001E-9W to +3.981071E-01W DEF[8] [8]
<function>	Measurement function	POW:AC
<resolution> or Resolution	Sets the resolution. This is used to determine averaging time if AVER:COUN:AUTO = TRUE	1, 2, 3, 4[7] DEF[8]
<source list> or Source list	Measurement channel	(@1)[9]

[7] The allowable values are 1, 2, 3 and 4. These values are interpreted to mean 1dB, 0.1dB, 0.01dB and 0.001dB of resolution respectively.
[8] DEF means default. The sensor interprets DEF this mean "use this parameter's current value". So that CONF DEF,3 would not change <expected value>. However, <resolution> would be set to 3.
[9] This parameter may only be (@1). It is normally not supplied. If omitted, the value is assumed to be (@1)

CONFigure[1]?

Syntax:

Most common form:

```
CONF?
```

Long form:

```
CONFIGURE?
```

Description:

This command queries the sensor for its current configuration. A single string containing four parameters is returned. The parameters returned in the string are: <function>, <expected value>, <resolution> and <source list> respectively.

Example:

In the following example various forms of the command are exercised.

```
0000115 → CONF?
0000116 ← "POW:AC +2.000000E+01,+4,(@1)"
0000117 → CONFIGURE?
0000118 ← "POW:AC +2.000000E+01,+4,(@1)"
0000119 → CONF1?
0000120 ← "POW:AC +2.000000E+01,+4,(@1)"
0000121 → CONFIGURE1?
0000122 ← "POW:AC +2.000000E+01,+4,(@1)"
```

Explanation of returned values in line 0000116:

- The various parameters are:
  ```
  <function> = POW:AC
  <expected value> = +2.000000E+01
  <resolution> = +4
  <source list> =(@1)
  ```
- `POW:AC is` the command or measurement function (average power)
- `+2.000000E+01` is the expected value. This is 20dBm and assumes UNIT:POW = DBM. This value is not used by LB59xx sensors. However, it is tracked and reported as if it is being used for compatibility reasons.
- `+4` indicates a resolution of 0.001dB. The value can range from 1 to 4 inclusive. This parameter is used by AVER:COUN:AUTO (auto averaging) in conjunction with the measured value to determine the number of averages or averaging time.
- `(@1)` indicates the measurement channel. The LB59xx supports one channel so this will always be (@1)

Reset Condition:

The parameters are sets as shown below upon *RST

- Command function is set to POW:AC
- Expected value is set to +20dBm
- Resolution is set to 3
- Source list is set to (@1)

CONFigure[1] or CONFigure[1][:SCALar][:POWer:AC]

Syntax:

Most common form:

```
CONF <expected value>, <resolution>
```

Long form:

```
CONFIGURE:SCALAR:POWER:AC <expected value>, <resolution>,(@1)
```

Description:

This command configures the sensor for measurements.

- Expected value – This parameter tells the sensor the power level the user intends to measure. The value often passed into the LB59xx is DEF (default) since this parameter has no effect with the LB59xx. However, the value is retained and tracked as if it is used for compatibility reasons.
- Resolution – This parameter sets the number of settled digits for the measurement. The permissible values are 1, 2, 3 and 4. Where 1 indicates a resolution of 1dB and 4 indicates a resolution of 0.001dB. If DEF is passed instead of a value then the current value is used.
- Source List - With the LB59xx sensors this is normally omitted in that it has no affect. However, Source list is (if included) must be (@1).

Example:

In the following example the configuration is set using the most common form and the full form of the command. After setting the configuration the configuration is queried to verify its effect.

```
0000127 → CONF 10,2
0000128 → CONF?
0000129 ← "POW:AC +1.000000E+01,+2,(@1)"
0000130 → CONFIGURE:SCALAR:POWER:AC 15, 1, (@1)
0000131 → CONF?
0000132 ← "POW:AC +1.500000E+01,+1,(@1)"
```

Additional Information:

After this command is executed, measurements can be made by executing a MEAS? or READ? or an INIT command followed by a FETCH? command. Executing CONF has side effects in that changes are made to other settings. These changes are as follows:

INIT:CONT OFF	This may affect subsequent FETCH? commands
TRIG:SOUR IMM	The sensor to starts a measurement upon receipt of the command.
TRIG:DEL:AUTO ON	Enables automatic delay before making a measurement
AVER:COUN:AUTO ON	Enables auto averaging.
AVER:STAT ON	Enables averaging

FETCh[1]?or FETCh[1][:SCALar][:POWer:AC]?

Syntax:

Most common form:

```
FETC?
```

Long form:

```
FETCH1:SCALAR:POWER:AC? <expected value>, <resolution>,(@1)
```

Description:

Executing this command causes a measurement to be calculated and returns the value to the host (PC) when queried if the value is valid. The value can be invalidated when:

- *RST is executed
- A measurement is initiated
- Any time a parameter or setting is changed that affects the value is changed.

The returned value is normally text but it can be in a binary form (see the FORMAT commands).

Example:

In this sequence INIT:CONT is set to zero. This means that you must send the INIT command to initiate a new measurement. If you don't send an INIT before fetching (INIT:CONT disabled) then you'll get the most recent measurement. In lines 53 and 55 the values are repeated. This indicates that you're getting the same measurement. This is normal operation for FETCH? with INIT:CONT disabled.

```
0000045 → INIT:CONT 0
0000046 → INIT:CONT?
0000047 ← 0
0000048 → INIT
0000049 → FETCH?
0000050 ← -3.82458461E+01
0000051 → INIT
0000052 → FETCH?
0000053 ← -3.82517515E+01
0000054 → FETCH?
0000055 ← -3.82517515E+01
```

If INIT:CONT is enabled you get a new measured value each time. This is because the sensor is supplying a continuous stream of INITs. This is sometimes referred to as free run mode. This is shown in the following sequence. You can see that the measured value changes for each FETCH? The sensor is supplying the INIT commands so that there is no need for the INIT.

```
0000056 → INIT:CONT 1
0000057 → INIT:CONT?
0000058 ← 1
0000059 → FETCH?
```

```
0000060 ← -3.82422265E+01
0000061 → FETCH?
0000062 ← -3.82466078E+01
0000063 → FETCH?
0000064 ← -3.82478426E+01
```

An example of the the long form of FETCH? is shown in the following sequence.

```
0000078 → CONF?
0000079 ← "POW:AC -3.000000E+01,+4,(@1)"
0000080 → FETCH? -30,4,(@1)
0000081 ← -3.82494301E+01
0000082 → FETCH? -30,4,(@1)
0000083 ← -3.82418777E+01
0000084 → FETCH? -30,4,(@1)
0000085 ← -3.82471240E+01
0000086 → FETCH? -30,3,(@1)
0000087 ← timed out
0000088 → SYST:ERR?
0000089 ← -221,"Settings conflict"
0000090 → FETCH? DEF,DEF,(@1)
0000091 ← -3.82537947E+01
0000092 → FETCH? DEF,DEF,(@1)
0000093 ← -3.82523698E+01
0000094 → FETCH? DEF,DEF,(@1)
0000095 ← -3.82494074E+01
```

Note that, unlike MEAS?, the parameter information provided in a FETCH? (in lines 80, 82 and 84) does not change the configuration. Instead the parameters are used to confirm that the current configuration matches the parameters sent as part of the FETCH? command.

A mismatch between the parameters and the configuration results in “Settings conflict” error as seen in line 89. This occurred because the resolution in the FETCH? command was set to 3. And as previously demonstrated it is set to 4. This generated the timeout. Finally, if there is a need to use the full form of the command and you want to avoid the error message you can use DEF instead of passing explicit values for the parameters.

Common Error Messages:

Error -230 “Data corrupt or stale”: This can happen after a *RST, a measurement is initiated, or as a result of changing certain parameters (frequency, averaging, trigger source etc.). These are parameters that can potentially affect the measurement result.

MEASure[1]? or MEASure[1] [:SCALar][:POWer:AC]?

Syntax:

Most common form:

```
MEAS?
```

Long form:

```
MEASURE1:SCALAR:POWER:AC? <expected value>, <resolution>,(@1)
```

Description:

The measure command starts by configuring the sensor using the parameters passed in the command. After the configuration is complete it continues with the measurement and ends by placing the result in the output buffer. The process commences when the command is received.

MEASure? = CONFigure + READ?

It is important to remember that MEAS? forces a CONF command to be executed. While the executing CONF command has no measureable effect on the total measurement time it does carry all of the attendant side effect. These side effects noted in the CONF command description includes disabling INIT:CONT, setting the trigger source to immediate and so on.

This command configures the sensor for measurements then makes the measurement.

- Expected value – This parameter tells the sensor the power level the user intends to measure. The value often passed into the LB59xx is DEF (default) since this parameter has no effect with the LB59xx. However, the value is retained and tracked as if it is used for compatibility reasons.
- Resolution – This parameter sets the number of settled digits for the measurement. The permissible values are 1, 2, 3 and 4. Where 1 indicates a resolution of 1dB and 4 indicates a resolution of 0.001dB. If DEF is passed instead of a value then the current value is used.
- Source List - With the LB59xx sensors this is normally omitted in that it has no affect. However, Source list is (if included) must be (@1).

Example:

The following demonstrates how to make a measurement using the most common and the long form or MEAS?

```
0000001 → *RST
0000002 → MEAS?
0000003 ← -3.03355249E+01
0000004 → MEASURE1:SCALAR:POWER:AC? -30, 4,(@1)
0000005 ← -3.03415129E+01
```

READ[1]? or READ[[1] [:SCALar][:POWer:AC]?

Syntax:

Most common form:

```
READ?
```

Long form:

```
READ1:SCALAR:POWER:AC? <expected value>, <resolution>,(@1)
```

Additional Forms

```
READ? DEF, <resolution>
```

Description:

This query aborts any measurement in process. This is followed by an INIT command (internally) and potentially continues with the measurement concluding by placing the measurement result in the output buffer. Unlike MEAS? this command does not use the optional configuration information to configure the measurement.

Instead, the configuration parameters passed via READ? command are used for comparison. And if there is a mismatch between the parameters passed in the READ? command and the current configuration an error is generated. No measurement is placed in the output buffer. If the parameters do match the process continues with the measurement. And the subsequent result is placed in the output buffer.

Example:

In the sequence below, several forms of READ? are exercised. Included is the deliberate generation of a configuration mismatch error. READ? is often used with the CONF command. In this case you must be sure to mitigate unwanted side effects of using CONF,

Commands	Comments
0000001 → *RST	
0000002 → READ?	common form of READ?
0000003 ← -2.04422867E+01	
0000004 → READ:SCALAR:POWER:AC?	long form of READ? without parameters
0000005 ← -2.04437082E+01	
0000006 → READ:SCALAR:POWER:AC? DEF, DEF, (@1)	long form of READ? with default parameters
0000007 ← -2.04398956E+01	
0000008 → CONF?	get current configuration
0000009 ← "POW:AC +2.000000E+01,+3,(@1)"	

```
0000010 → READ:SCALAR:POWER:AC? DEF, 4, (@1)   resolution mismatch!
0000011 ← timed out                            the mismatch should generate an error or two,
                                               and it did!
0000012 → SYST:ERR?                            get…
0000013 ← -221,"Settings conflict"             …the first error
0000014 → SYST:ERR?                            get…
0000015 ← -420,"Query UNTERMINATED"            …the second error
0000016 → READ:SCALAR:POWER:AC? DEF, 3, (@1)   resolution matches this time…
0000017 ← -2.04420077E+01                      …no error, returns a measurement
0000018 → READ:SCALAR:POWER:AC? DEF, 3         same command without source list
0000019 ← -2.04415148E+01
```

Common Error Messages:

Again, it is important to refer to the CONF command to understand the side effects associated with CONF. Some settings can generate errors when using READ? and prevent READ? from delivering a result.

- INIT:CONT must be disabled or OFF before trying to make a measurement with READ? If a READ? command is issued with INIT:CON enabled or ON the following errors are generated:
    - 213 “Init ignored”
    - 420 “Query UNTERMINATED”
- If the trigger source is either BUS or HOLD the following errors are generated:
    - 214,"Trigger deadlock"
    - 420,"Query UNTERMINATED"
- Finally, if parameters are passed as part of a READ? command they must either match the current settings or be set to default (“DEF”). Otherwise the following errors will be generated:
    - 221,"Settings conflict"
    - 420,"Query UNTERMINATED"

Calculate Commands

In power sensors supporting SPCI, the calculate commands use measurement data in post-processing. The primary function is setting limits and reporting how a sequence of measurements performed relative to these limits.

To this purpose the LB59xx power sensors allow the user to set upper and lower power limits. These limits are then used for "failure" counting. A failure is any measured outside the defined limits. The type of counting is dependent on the settings. It can keep track of failures in one of two ways:

1. It can report the accumulated number of failures since the failure count was reset
2. In can report that a failure did or did not occur with the most recent measurement
3. Combines the first two options

In the second case the count is continuously reset. This limits the count to 0 and 1 indicating that the most recent measurement has failed. Otherwise, the count is generally allowed to accumulate until it is cleared. The upper limit of counting is 65535. An error is issued anytime the count exceeds this value.

CALC:FEED[?] or CALCulate[1]:FEED[?]

CALC:MATH[?]or CALCulate[1]:MATH:EXPRession[?]

CALC:MATH:CAT? or CALCulate:MATH:EXPRession:CATalog?

Syntax:

Most common forms:

```
CALC:FEED?
CALC:FEED "POW:AVER"
CALC:MATH?
CALC:MATH "(SENS1)"
CALC:MATH:CAT?
```

Long forms:

```
CALCulate1:FEED?
CALCulate1:FEED "POW:AVER"
CALCulate1:MATH:EXPRession?
CALCulate1:MATH:EXPRession "(SENS1)"
CALCulate:MATH:EXPRession:CATalog?
```

Description:

These commands set or return the current function (“POW:AVER”) or the math expression catalog. These values are singular and fixed. In other words there is one, and only one, permissible value for each of these commands. These commands have greater value in instruments possessing multiple detectors and sensors. Since this sensor is dedicated to single channel average power measurements, these commands are of limited use beyond command and driver compatibility.

Examples:

Commands	Comments
0000124 → CALC1:FEED?	what is the measurement mode…
0000125 ← "POW:AVER"	…measuring average power
0000126 → CALC1:FEED "POW:AVER"	set the feed to the only permissible value
0000127 → CALC:MATH:EXPR?	what is the current math expression…
0000128 ← "(SENS1)"	…we have one channel
0000129 → CALC:MATH "(SENS1)"	set the math expression to the only allowable value…
0000130 → CALC:MATH:EXPR:CAT?	get a list of math expressions…
0000131 ← "(SENS1)"	…there is only one

CALC:LIM:CLEar:AUTO[?] or CALCulate[1]:LIMit:CLEar:AUTO{?}

Syntax:

Most common forms:

```
CALC:CLE:AUTO?
CALC:CLE:AUTO    <0|1|ONCE>
```

Long forms:

```
CALCulate1:LIMit:CLEar:AUTO?
CALCulate1:LIMit:CLEar:AUTO     <0|1|ONCE>
```

Description:

The parameter provided with this command controls how and under what conditions the failure count is cleared. Permissible values are 0, 1 or ONCE. Each meaning is below:

- If the value is 0 or OFF the failure count is NOT automatically cleared. Rather the count is cleared ONLY when the sensor receives a CALC:LIM:CLE command.
- If the value is 1 or ON the failure count is cleared immediately before each measurement. This setting indicates pass/fail of the most recent measurement. These command cause the count to be cleared if:
  - INIT or INIT:IMM command is issued
  - A measurement commences with INIT:CONT set to 1 or ON
  - A MEAS? command is executed
  - A READ? command is executed
- If the feature is set to ONCE, the count is cleared starting upon starting the next measurement. Thereafter the sensor behaves as if this feature was set to 0 (never automatically cleared)
- **The count is cleared or reset anytime the state of CALC:LIM:AUTO state changes?????**

Example

This sequence demonstrates the use of most of the CALC commands. This is done because of inadequacy of demonstrating the commands in isolation of each other.

The sequence starts by setting limits and enabling limit checking. Initially source power is set to a level between the upper and lower limits. Measurements then proceed. Under these circumstances no failure should occur.

The sequence continues by setting the source power below the lower limit. Measurements recommence and failures occur. Source settings remain unchanged thereafter. However, settings that manage failure tracking are varied and the resulting effects are demonstrated.

Commands	Comments
`0000714 → *RST`	Set sensor to a known state
`0000715 → AVER:COUN:AUTO 0`	Disable auto averaging (convenience)
`0000716 → AVER:COUN 10`	Set average count to 10 (faster)

```
0000717 → CALC:LIM:LOW -10          Set lower…
0000718 → CALC:LIM:UPP 10           …and upper limits
0000719 → CALC:LIM:STAT 1           Turn on limit checking
0000720 → CALC:LIM:FAIL?            Any failures yet?
0000721 ← 0                         Nope
0000725 → CALC:LIM:FCO?             Failure count?
0000726 ← +0                        None
Set source power to 0dBm            Should not cause failures
0000727 → READ?                     Start making some measurements
0000728 ← -3.83508613E-01
0000729 → READ?
0000730 ← -3.84266024E-01
0000731 → READ?
0000732 ← -3.84728069E-01
0000733 → CALC:LIM:FAIL?            Any failures?
0000734 ← 0                         Nope!
0000735 → CALC:LIM:FCO?             And the count…
0000736 ← +0                        …is zero
Set source power to -20dBm          This change in power level should cause failures..
0000737 → READ?                     Start making more measurements
0000738 ← -2.04607346E+01
0000739 → READ?
0000740 ← -2.04601199E+01
0000741 → READ?
0000742 ← -2.04593472E+01
0000743 → READ?
0000744 ← -2.04591619E+01
0000745 → CALC:LIM:FAIL?            Any failures?
0000746 ← 1                         Yes
0000747 → CALC:LIM:FCO?             How many…(default value for CALC:LIM:CLE = 1)
0000748 ← +1                        … shows a count of one. Resets on start of measurement
0000749 → CALC:LIM:CLE              …clear the failures
0000750 → CALC:LIM:CLE:AUTO 0       …disable failure auto clear
0000751 → CALC:LIM:FAIL?            Any failures yet
0000752 ← 0                         No. There shouldn't be since ewe just cleared failures
0000753 → CALC:LIM:FCO?             And the count is…
0000754 ← +0                        …0, this is also correct
0000755 → READ?                     Make some measurements…source is still at -20dBm
0000756 ← -2.04572978E+01
0000757 → READ?
0000758 ← -2.04567314E+01
0000759 → READ?
```

```
0000760 ← -2.04565999E+01
0000761 → CALC:LIM:FAIL?                Any failures?
0000762 ← 1                             Yes…and there should be
0000763 → CALC:LIM:FCO?                 How many?
0000764 ← +3                            correct… count is not clearing for each measurement
0000765 → CALC:LIM:CLE:AUTO?            and auto failure clear is…
0000766 ← 0                             …still disabled
0000767 → CALC:LIM:CLE:AUTO ONCE        <<<<<---- Setup to clear once then count
0000771 → READ?                         Start making measurements…
0000772 ← -2.04522242E+01
0000773 → READ?
0000774 ← -2.04518298E+01
0000775 → CALC:LIM:FAIL?                Any failures…
0000776 ← 1                             Yes
0000777 → CALC:LIM:FCO?                 Count is..
0000778 ← +2                            2
0000779 → CALC:LIM:CLE:AUTO ONCE        <<<<<---- Set it to clear once and then count again
0000780 → READ?                         Make measurements
0000781 ← -2.04521913E+01
0000782 → READ?
0000783 ← -2.04511674E+01
0000784 → READ?
0000785 ← -2.04518987E+01
0000786 → READ?
0000787 ← -2.04508453E+01
0000788 → CALC:LIM:FAIL?                Any failures?
0000789 ← 1                             Yes
0000790 → CALC:LIM:FCO?                 And we counted only those failures that occurred…
0000791 ← +4                            … after clearing once
0000792 → READ?                         So keep making measurements
0000793 ← -2.04508859E+01
0000794 → READ?
0000795 ← -2.04509669E+01
0000796 → READ?
0000797 ← -2.04509931E+01
0000798 → CALC:LIM:FAIL?                Any failures
0000799 ← 1                             Yes
0000800 → CALC:LIM:FCO?                 How many?
0000801 ← +7                            Continued counting since clearing once on line 779
0000802 → CALC:LIM:CLE:AUTO ONCE        Auto clear to once a final time
0000803 → READ?                         Make some measurements
0000804 ← -2.04511196E+01
```

`0000805 → CALC:LIM:FAIL?`	Any failures?
`0000806 ← 1`	
`0000807 → CALC:LIM:FCO?`	Yes but…
`0000808 ← +1`	…just one so it must have cleared as it should

CALC:LIM:CLE[?] or CALCulate[1]:LIMit:CLEar[:IMMediate][?]

Syntax:

Most common form:
```
CALC:LIM:CLE
```

Long form:
```
CALCULATE1:LIMIT:CLEAR:IMMEDIATE
```

Description:

Clears the failure indicator (CALC:LIM:FAIL?) and failure counter (CALC:LIM:FCO).

Example:

See an extensive example of the CALC commands in CALC:LIM:CLEar:AUTO[?]

CALC:LIM:FAIL? or CALCulate[1]:LIMit:FAIL?

Syntax:

Most common form:
```
CALC:LIM:FAIL?
```

Long form:
```
CALCULATE:LIMIT:FAIL?
```

Description:

This command ANDs the bits of failure count. As a result, this query returns:

- 0 if no failures have occurred
- 1 if any failures have occurred.

This command is useful in determining if any failure has occurred. For this command to be function, CALC:LIM:STAT must be enabled.

Example:

See an extensive example of the CALC commands in CALC:LIM:CLEar:AUTO[?]

On RESET

This value is set to zero upon reset.

CALC:LIM:FCO? or CALCulate[1]:LIMit:FCOunt?

Syntax:

Most common form:

```
CALC:LIM:FCO?
```

Long form:

```
CALCULATE1:LIMIT:FCOUNT?
```

Description:

This query returns the number of failures counted so far in concert with other CALC settings. The count if dependent upon CALC:LIM:STAT begin enabled (equal to 1) and the value of CALC:LIM:CLE:AUTO.

Example:

See an extensive example of the CALC commands in CALC:LIM:CLEar:AUTO[?]

Reset:

On reset or power up the counter is set to zero.

CALC:LIM:LOW[?] or :CALCulate[1]:LIMit:LOWer[:DATA][?]

Syntax:

Most common form:

```
CALC:LIM:LOW <value>
CALC:LIM:LOW?
```

Long form:

```
CALCULTATE1:LIMIT:LOWER:DATA <value>
CALCULTATE1:LIMIT:LOWER:DATA?
```

Description:

The command sets the lower test limit to which the measured value is compared. If the measured value is lower than this limit, the failure count may be incremented. The values are:

- Minimum value is -150dBm
- Maximum value is +230dBm
- Reset or Default value is -90dBm

Example:

See an extensive example of the CALC commands in CALC:LIM:CLEar:AUTO[?]

CALC:LIM:STAT[?] or CALCulate[1]:LIMit:STATe[?]

Syntax:

Most common form:

```
CALC:LIM:STAT?
CALC:LIM:STATE < 0|1 >
```

Long form:

```
CALCULATE1:LIMIT:STATE?
CALCULATE1:LIMIT:STATE < 0|1 >
```

Description:

This setting enables or disables failure counting.

Example:

See an extensive example of the CALC commands in CALC:LIM:CLEar:AUTO[?]

Reset:

Upon reset or power on this is disabled.

CALC:LIM:UPP[?] or CALCulate[1]:LIMit:UPPer:DATA[?]

Syntax:

Most common form:

```
CALC:LIM:UPP?
CALC:LIM:UPP <value>
```

Long form:

```
CALCULATE1:LIMIT:UPPER:DATA?
CALCULATE1:LIMIT:UPPER:DATA <value>
```

Description:

The command sets the upper test limit to which the measured value is compared. If the measured value is above this limit, the failure count may be incremented. The values are:

- Minimum value is -150dBm
- Maximum value is +230dBm
- Reset or Default value is +90dBm

Example:

See an extensive example of the CALC commands in CALC:LIM:CLEar:AUTO[?]

Calibration

These commands are required by other sensors made by other vendors in order to properly zero a power sensor. However, zeroing is not required with theLB59xx line of power sensors. So that these commands are included for compatibility purposes only and have no effect on performance of the sensor. However, in some cases a value is set. In these cases the values are tracked.

CAL:ZERO:AUTO or CALibration1:ZERO:AUTO

Syntax:

Most common forms:

```
CAL:ZERO:AUTO?
CAL:ZERO:AUTO < 0|1 >
```

Long forms:

```
CALIBRATION1:ZERO:AUTO?
CALIBRATION1:ZERO:AUTO < 0|1 >
```

Description:

This command is non-functioning. It is included for compatibility purposes only. The value is tracked and returned but otherwise has no affect.

Examples:

```
0000013 → CAL:ZERO:AUTO?
0000014 ← 1
0000015 → CAL:ZERO:AUTO 0
0000016 → CAL:ZERO:AUTO?
0000017 ← 0
0000018 → CAL:ZERO:AUTO 1
0000019 → CAL:ZERO:AUTO?
0000020 ← 1
0000021 → *RST
0000022 → CAL:ZERO:AUTO?
0000023 ← 1
0000024 → CAL:ZERO:AUTO ONCE
0000025 → CAL:ZERO:AUTO?
0000026 ← 0
0000027 → CAL:ZERO:AUTO 1
0000028 → CAL:ZERO:AUTO?
0000029 ← 1
0000030 → CAL:ZERO:AUTO ONCE
0000031 → CAL:ZERO:AUTO?
0000032 ← 0
```

CAL:ZERO:TYPE or CALibration1:ZERO:TYPE

Syntax:

Most common forms:

```
CAL:ZERO:TYPE?
CAL:ZERO:TYPE < EXT|INT >
```

Long forms:

```
CALIBRATION1:ZERO:TYPE?
CALIBRATION1:ZERO:TYPE < EXTERNAL|INTERNAL >
```

Descriptions:

This command is non-functioning. It is included for compatibility purposes only. The value is tracked and returned but otherwise has no affect. The default value is INT or INTERNAL.

Example:

```
0000035 → CAL:ZERO:TYPE?
0000036 ← INT
0000037 → CAL:ZERO:TYPE EXT
0000038 → CAL:ZERO:TYPE?
0000039 ← EXT
0000040 → CAL:ZERO:TYPE INT
0000041 → CAL:ZERO:TYPE?
0000042 ← INT
0000043 → *RST
0000044 → CAL:ZERO:TYPE?
0000045 ← INT
```

CAL or CALibration1[:ALL]

Syntax:

Most common forms:

```
CAL?
CAL
```

Long forms:

```
CALIBRATION1:ALL?
CALIBRATION1:ALL
```

Descriptions:

This command is non-functioning. It is included for compatibility purposes only. When queried it always returns a zero.

Examples:

```
0000046 → CAL
0000047 → CAL?
0000048 ← 0
0000049 → CAL:ALL?
0000050 ← 0
0000051 → CAL:ALL
0000052 → CAL:ALL?
0000053 ← 0
```

Format

The format commands control two items. The format (binary or text) and the byte order in the event that the format of the data is set to binary.

FORMat[:READings]:BORDer

Syntax:

Most common forms:

```
FORM:BORD < NORM|SWAP >
FORM:BORD?
```

Long forms:

FORMAT:READINGS:BORDER < NORMAL|SWAPPED >
FORMAT:READINGS:BORDER?

Description:

This setting determines the byte ordering of data transferred in binary format.

Example:

The following example demonstrates exercising the command to change the byte ordering.

```
0000054 → FORM:BORD?
0000055 ← NORM
0000056 → FORM:BORD SWAP
0000057 → FORM:BORD?
0000058 ← SWAP
0000059 → FORM:BORD NORM
0000060 → FORM:BORD?
0000061 ← NORM
0000062 → FORMAT:READINGS:BORDER?
0000063 ← NORM
0000064 → FORMAT:READINGS:BORDER SWAPPED
0000065 → FORMAT:READINGS:BORDER?
0000066 ← SWAP
```

On Reset:

On reset or power up byte ordering (BORD) is set to NORMAL

Other Notes:

This has no effect if FORM? is set to ASC or ASCII

FORMat[:READings][:DATA]

Syntax:

Most common forms:

```
FORM < ASC|REAL >
FORM?
```

Long forms:

```
FORMAT:READINGS:DATA < ASCII|REAL >
FORMAT:READINGS:DATA?
```

Description:

This command determines whether the transfer of numeric data is either ASCII or REAL.

- If set to ASCII the text format is a number an exponent such as 1.000E+3 (representing 1000) called NR3 in SCPI. This is sometimes referred to as scientific notation.
- If set to REAL then the data is in a binary form consistent with IEEE 754. This is sometimes referred to as a 64 bit real or a double in some programming languages. Each value is terminated by a line feed (ASCII value of 10).

Examples:

This sequence demonstrates setting the format and retrieving the format.

```
0000088 → FORMAT:READINGS:DATA?
0000089 ← ASC
0000090 → FORMAT:READINGS:DATA REAL
0000091 → FORMAT:READINGS:DATA?
0000092 ← REAL
0000093 → FORM?
0000094 ← REAL
0000095 → FORM ASCII
0000096 → FORM?
0000097 ← ASC
```

On Reset

The format is set so ASCII on power on and reset.

Initiate

In general, the purpose of the initiate commands is to manage the sensors response to triggering. There are three possible states:

- Idle state: In this state the sensor will ignore all incoming triggers. The sensor will remain in an idle state until the measurement process is initiated.
- Active state: Ready to respond to the active trigger. The most common trigger sources are immediate triggers issued by the sensor (MEAS? or READ? commands), external triggers or internal triggers.

If the sensor is active, when a trigger arrives the sensor will make the measurement and place resultant measurement in the outgoing buffer. The sensor then returns to the idle state. It remains in the idle state until:

- The user issues an explicit INIT command
- The user issues an implicit INIT command by sending a MEAS? or READ? command
- The sensor provides a INITIATE command upon completion of a measurement (INIT:CONT = 1)

An important idea to keep in mind is that the initiate commands are considered to be “overlapped”.

INITiate:CONTinuous

INITiate1:CONTinuous

INITiate:CONTinuous:ALL

INITiate1:CONTinuous:ALL

INITiate:CONTinuous:SEQuence

INITiate1:CONTinuous:SEQuence

INITiate:CONTinuous:SEQuence1

INITiate1:CONTinuous:SEQuence1

Syntax:

Most common forms:

```
INIT:CONT < 0|1 >
INIT:CONT
```

Long forms:

```
INITIATE:CONTINUOUS < 0|1 >
INITIATE1:CONTINUOUS:SEQUENCE1 < 0|1 >
INITIATE1:CONTINUOUS:ALL < 0|1 >
INITIATE:CONTINUOUS?
INITIATE1:CONTINUOUS:SEQUENCE1?
INITIATE1:CONTINUOUS:ALL?
```

Description:

For this product line, all of these commands are identical. These commands cause the sensor to either wait for an explicate command to initiate a measurement cycle or generate the command internally. When enabled (INIT:CONT = 1) the sensor generates a continuous stream of INITIATE commands sometimes referred to as free run. This property can be queried. If INIT:CONT is set to:

1. 0 causes the sensor to remain in an idle state (doing nothing). It waits for the INIT command. When it receives and INIT command it exits the idle state and starts waits for a trigger.
2. 1 causes the sensor issues an INIT command internally upon completion of a measurement. This causes the sensor to continuously restart the measurement process. If the trigger source is set to immediate, the sensor continuously makes measurements.

Examples:

In the following examples INIT:CONT is set (to 1 or 0) and queried using of the various forms of the command.

```
0000041 → INIT:CONT?
```

```
0000042 ← 0
0000043 → INIT:CONT 1
0000044 → INIT:CONT?
0000045 ← 1
0000046 → INITIATE:CONTINUOUS?
0000047 ← 1
0000048 → INITIATE1:CONTINUOUS?
0000049 ← 1
0000050 → INITIATE1:CONTINUOUS 0
0000051 → INITIATE1:CONTINUOUS?
0000052 ← 0
0000053 → INITIATE1:CONTINUOUS 1
0000054 → INITIATE1:CONTINUOUS?
0000055 ← 1
```

On Power Up

INIT:CONT is set to 1 upon power up.

On Reset

INIT:CONT is set to 0 on reset (*RST)

Common Error Messages:

If INIT:CONT = 1, FETCH? must be used to make a measurement. In the first part of this example below the user attempts to make a measurement using the READ? (with INIT:CONT = 1). The READ? fails and error messages are generated. Folowing this, the FETCH? command is used to make a measurement and works quite nicely. Finally, and INIT is issued (INIT:CONT stil is set to 1). This causes an "INIT ignored" error to be generated by the sensor.

```
0000092 → INIT:CONT 1
0000093 → READ?
0000094 ← timed out
0000095 → SYST:ERR?
0000096 ← -213,"Init ignored"
0000097 → SYST:ERR?
0000098 ← -420,"Query UNTERMINATED"
0000099 → FETCH?
0000100 ← -1.03488405E+01
0000109 → INIT
0000110 → SYST:ERR?
0000111 ← -213,"Init ignored"
```

Other Notes:

Setting INIT:CONT = 1 and TRIG:SOUR = IMMEDITE is the same as free run.

If INIT:CONT = 1 then the INIT command should not be issued. To make a measurement with INIT:CONT = 1 you need only issue a FETCH? command. If INIT:CONT = 0 you must issue an INIT command to start the measurement process. Having issued an INIT command, any of the measurement commands (MEAS?, READ? or FETCH?) can be used. Finally, issuing a MEAS? command causes INIT:CONT to set to 0.

INITiate[:IMMediate]/nquery/

INITiate1[:IMMediate]/nquery/

INITiate[:IMMediate]:ALL/nquery/

INITiate1[:IMMediate]:ALL/nquery/

INITiate[:IMMediate]:SEQuence/nquery/

INITiate1[:IMMediate]:SEQuence/nquery/

INITiate[:IMMediate]:SEQuence1/nquery/

INITiate1[:IMMediate]:SEQuence1/nquery/

Syntax:

Most common forms:

```
INIT
```

Long forms:

```
INITIATE1:IMMEDIATE:SEQUENCE1
```

Description:

This command is issued when INIT:CONT is 0. The command causes the sensor to exit the idle state and begin waiting for a trigger. If the trigger source is set to immediate a measurement commences upon receipt of this command.

Examples:

In the example below we see the relationship between INIT:CONT and INIT demonstrated. It also demonstrates the incorrect way to use INIT. INIT assumes that INIT:CONT = 0. If that is not the case then an “Init ignored” error is generated.

```
0000114 → INIT:CONT?
0000115 ← 1
0000116 → INIT
0000117 → SYST:ERR?
0000118 ← -213,"Init ignored"
0000119 → SYST:ERR?
0000120 ← +0,"No error"
0000121 → INIT:CONT 0
0000122 → INIT
0000123 → FETCH?
0000124 ← -1.03615134E+01
0000125 → SYST:ERR?
0000126 ← +0,"No error"
0000127 → INIT
0000128 → READ?
```

```
0000129 ← -1.03532959E+01
```

Common Error Messages:

As shown in the example, INIT should not be issued when INIT:CONT = 1. In this case the sensor is generating the INIT commands internally. If an INIT is issued when INIT:CONT = 1 then an "Init ignored" error message is generated.

Other Notes:

The INIT command can be used with any of the measurement commands (MEAS? READ? or FETCH?). To use this command INIT:CONT must be set to 0 or OFF. If INIT:CONT = 1 and an INIT is issued an **`213,"Init ignored"`** error is generated.

This command is the same as TRIG:IMM or TRIGGER:IMMEDIATE

Input

There is a single input command. This command controls the input impedance of the trigger in port.

INPut:TRIGger:IMPedance

Syntax:

Most common forms:

```
INP:TRIG:IMP < LOW:HIGH >
INP:TRIG:IMP?
```

Long forms:

```
INPUT:TRIGGER:IMPEDANCE < LOW|HIGH >
INPUT:TRIGGER:IMPEDANCE?
```

Description:

This command controls the impedance seen at the trigger in port. It has two values and they are LOW and HIGH. A setting of LOW causes the trigger input impedance to be 50Ω. And a setting of HIGH causes the trigger input impedance to be 100kΩ.

Examples:

In the sequence below we are setting the input impedance low and high.

```
0000025 → INP:TRIG:IMP?
0000026 ← HIGH
0000027 → INP:TRIG:IMP LOW
0000028 → INP:TRIG:IMP?
0000029 ← LOW
0000030 → INP:TRIG:IMP HIGH
```

On Reset

After a power up or reset (*RST) the input impedance is set to HIGH by default.

Memory

The memory subsystem is used to store, edit and manage:

3. Frequency dependent offset tables (sometimes referred to as correction tables)
4. Save/Recall registers

The LB59xx sensors have 10 save/recall registers and 10 frequency dependent offset tables. Each table may contain up to 80 points of correction. Each point in the table consists of a frequency and a power level value.

The frequency dependent offset tables and registers (or states) are held in non-volatile memory. So, a loss of power will not cause the sensor to loose save/recall states or correction data settings.

NOTE: The MEM commands use numbers ranging from 0 to 9 for both the tables and registers. The *SAV and *RCL commands use register numbers ranging from 1 to 10. In other words, MEM commands use a zero based numbering system and the IEEE 488.2 *SAV and *RCL commands use a one based numbering system.

Fortunately most of the MEM commands do not use register numbers instead they use register names. The exception is the MEM:STAT:DEF or MEMORY:STATE:DEFINE command. This command allows the user to change the name of a SAVE/RECALL register. These names are reported in the catalog functions. For this command you must use a zero based numbering scheme. For example:

5. Save the current state using the command *SAV 5
6. Recall this same state using *RCL 5
7. However, to rename this same register you must issue the following command MEM:STAT:DEF “REG_5_NAME”, 4 (note the 4 in the command instead of 5)

While this may seem awkward it is necessary for command and software driver compatability.

MEMory:CATalog:STATe?/qonly/

Syntax:

Most common forms:

```
MEM:CAT:STAT?
```

Long forms:

```
MEMORY:CATALOG:STATE?
```

Description:

This query returns a catalog of registers. The format of the return string is:

```
<numeric>,<numeric>,"<string0>,<type>,<size>","<string1>,<type>,<size>", …
                                                       "<string9>,<type>,<size>",
```

The first numeric value is the number of bytes used. The second is number of bytes available. This data is followed by ten sets of save/recall register information. Each register has three pieces of information. The first is the name of the state, the second is the type of memory (always be STAT in this case) third is the size of allocated memory used by the register.

This data is held in non-volatile memory so that resets or power up/down do not affect the save recall registers. Instead, these registers must be cleared explicitly using SCPI commands such as MEM:CLE < name > where name is the name of either a register or table.

Example:

In this case we set the command requests a catalog of save/recall registers.

```
0000053 → MEM:CAT:STAT?
0000054 ←
0,2880,"State0,STAT,0","State1,STAT,0","State2,STAT,0","State3,STAT,0","State4,STAT,0","State
5,STAT,0","State6,STAT,0","State7,STAT,0","State8,STAT,0","State9,STAT,0"
```

Other Notes:

The numbering scheme between *SAV and *RCL are at variance with the memory subsystem's numbering system. The memory subsystem uses a zero based (0…9) sequence as shown in the return value above and the *SAV and *RCL use a 1 based (1…10) sequence.

MEMory:CATalog:TABLe?/qonly/

Syntax:

Command form:
MEM:CAT:TABL?

Long form:
MEMORY:CATALOG:TABLE?

Description:

This query only command returns a catalog of the saved frequency correction tables. The format of the return string is:

```
<numeric>,<numeric>,"<string0>,<type>,<size>","<string1>,<type>,<size>", …
                                                    "<string9>,<type>,<size>",
```

The first numeric value is the number of bytes used. The second is the number of bytes available. Each subsequent string contains three pieces of information. The first is the name of the table, the second is the type of memory (always be TABL in this case) third is the size of allocated memory used by the table.

This data is held in non-volatile memory so that resets or power up/down do not affect the tables. Instead, these tables must be cleared explicitly using SCPI commands such as MEM:CLE < name > where name is the name of either a register or table.

Example:

```
0000005 → MEM:CAT:TABL?

0000006 ←
0,4800,"CUSTOM_A,TABL,0","CUSTOM_B,TABL,0","CUSTOM_C,TABL,0","CUSTOM_D,TABL,0","CUSTOM_E,TABL
,0","CUSTOM_F,TABL,0","CUSTOM_G,TABL,0","CUSTOM_H,TABL,0","CUSTOM_I,TABL,0","CUSTOM_J,TABL,0"
```

MEMory:CATalog[:ALL]?/qonly/

Syntax:

Most common form:

```
MEM:CAT:ALL?
```

Long forms:

```
MEMORY:CATALOG:ALL?
```

Description:

This query only command returns a catalog of the saved registers and frequency correction tables. The format of the return string is:

```
<numeric>,<numeric>,"<string0>,<type>,<size>","<string1>,<type>,<size>", …
                                                    "<string9>,<type>,<size>",
```

The first numeric value is the number of bytes used. The second is total number of bytes available. This is followed by twenty sets of definitions. In each case the string contains the name of the register or table, the type shows the type of memory (should be STAT or TABL) and the number of bytes currently used by each state or table.

This data is held in non-volatile memory so that neither resets nor power up/down affects the state or tables. Instead, these states and tables must be cleared explicitly by a command such as MEM:CLE < name > where name is the name of either a register or table

Examples:

```
0000016 → MEM:CAT:ALL?
0000017 ←
288,7680,"CUSTOM_A,TABL,0","CUSTOM_B,TABL,0","CUSTOM_C,TABL,0","CUSTOM_D,TABL,0","CUSTOM_E,TA
BL,0","CUSTOM_F,TABL,0","CUSTOM_G,TABL,0","CUSTOM_H,TABL,0","CUSTOM_I,TABL,0","CUSTOM_J,TABL,
0","State0,STAT,288","State1,STAT,0","State2,STAT,0","State3,STAT,0","State4,STAT,0","State5,
STAT,0","State6,STAT,0","State7,STAT,0","State8,STAT,0","State9,STAT,0"
```

MEMory:CLEar:TABLe/nquery/

Syntax:

Most common form:

```
MEM:CLE:TABL
```

Long form:

```
MEMORY:CLEAR:TABLE
```

Description:

This command clears the currently selected table. If a table isn't selected a 221 "Settings conflict" error is generated.

Examples:

In this sequence we first catalog the tables. Note that CUSTOM_B contains information. We ask the sensor which table is selected. Initially no table is selected (hence a null string is returned). Then we issue the command to clear the currently selected table. This of course generates an error. Then we select CUSTOM_B and issue the command to clear the selected table. This works correctly as shown the size of CUSTOM_B is reduced from 30 to 0.

```
0000003 → MEM:CAT:TABL?
0000004 ←
30,4800,"JON,TABL,0","CUSTOM_B,TABL,30","CUSTOM_C,TABL,0","CUSTOM_D,TABL,0","CUSTOM_E,TABL,0"
,"CUSTOM_F,TABL,0","CUSTOM_G,TABL,0","CUSTOM_H,TABL,0","CUSTOM_I,TABL,0","CUSTOM_J,TABL,0"
0000005 → MEM:TABL:SEL?
0000006 ←
0000007 → MEM:CLE:TABL
0000008 → SYST:ERR?
0000009 ← -221,"Settings conflict"
0000010 → SYST:ERR?
0000011 ← +0,"No error"
0000012 → MEM:TABL:SEL "CUSTOM_B"
0000013 → MEM:CLE:TABL
0000014 → SYST:ERR?
0000015 ← +0,"No error"
0000016 → MEM:CAT:TABL?
0000017 ←
0,4800,"JON,TABL,0","CUSTOM_B,TABL,0","CUSTOM_C,TABL,0","CUSTOM_D,TABL,0","CUSTOM_E,TABL,0","
CUSTOM_F,TABL,0","CUSTOM_G,TABL,0","CUSTOM_H,TABL,0","CUSTOM_I,TABL,0","CUSTOM_J,TABL,0"
```

Common Error Messages:

If no table is selected error 221 "Settings conflict" is generated

Other Notes:

Cleared data is not recoverable.

MEMory:CLEar[:NAME]/nquery/

Syntax:

Most common forms:

```
MEM:CLE <name>
```

Long forms:

```
MEMORY:CLEAR:NAME <name>
```

Description:

This command clears the data associated with a named table or state. If a state is named then the state is cleared. If the table is named, the table is cleared.

Examples:

In this example a table is cleared then a state is cleared. The state and the table are cleared using the default names as shown in the MEM:CAT:ALL? command.

```
0000014 → MEM:CAT:ALL?
0000015 ←
0,7680,"CUSTOM_A,TABL,0","CUSTOM_B,TABL,0","CUSTOM_C,TABL,0","CUSTOM_D,TABL,0","CUSTOM_E,TABL
,0","CUSTOM_F,TABL,0","CUSTOM_G,TABL,0","CUSTOM_H,TABL,0","CUSTOM_I,TABL,0","CUSTOM_J,TABL,0"
,"State0,STAT,0","State1,STAT,0","State2,STAT,0","State3,STAT,0","State4,STAT,0","State5,STAT
,0","State6,STAT,0","State7,STAT,0","State8,STAT,0","State9,STAT,0"
0000016 → MEM:CLE "CUSTOM_B"
0000017 → MEM:CLE "State0"
```

Other Notes:

Once cleared, a table or state is not recoverable.

MEMory:FREE:STATe?/qonly/

Syntax:

Most common forms:

```
MEM:FREE:STAT?
```

Long forms:

```
MEMORY:FREE:STATE?
```

Description:

This query returns the total memory available and the memory used by the save/recall registers. Each register uses 288 bytes.

Examples:

```
0000210 → MEM:FREE:STAT?
0000211 ← 2880,288
0000212 → MEMORY:FREE:STATE?
0000213 ← 2880,288
```

MEMory:FREE:TABLe?/qonly/

Syntax:

Most common forms:

MEM:FREE:TABL?

Long forms:

MEMORY:FREE:TABLE?

Description:

This query returns the total memory available and memory used in bytes.

Examples:

The following sequence demonstrates this command.

```
0000205 → MEM:FREE:TABL?
0000206 ← 4800,0
0000207 → MEMORY:FREE:TABLE?
0000208 ← 4800,0
```

MEMory:FREE[:ALL]?/qonly/

Syntax:

Most common forms:

Long forms:

Description:

This query returns the total memory available for both the registers and tables and the total memory used by both registers and tables.

Examples:

In this sequence the command is exercised and the return values are shown.

```
0000022 → MEM:FREE:ALL?
0000023 ← 7680,576
0000024 ← timed out
0000025 → MEM:FREE:ALL?
0000026 ← 7680,576
0000027 → MEM:FREE:ALL?
0000028 ← 7680,576
0000029 → MEM:FREE?
0000030 ← 7680,576
0000031 → MEMORY:FREE?
0000032 ← 7680,576
```

MEMory:NSTates?/qonly/

Syntax:

Most common forms:

```
MEM:NST?
```

Long forms:

```
MEMORY:NSTATES?
```

Description:

This command returns the number of states available. Since the number of states available is always 10, this command always returns 10.

Examples:

```
0000077 → MEMORY:NSTATES?
0000078 ← +10
0000079 → MEM:NST?
0000080 ← +10
```

MEMory:STATe:CATalog?/qonly/

Syntax:

Most common forms:

```
MEM:STAT:CAT?
```

Long forms:

```
MEMORY:STATE:CATALOG?
```

Description:

This command lists the name of the save/recall states in order from the first state to the last state. Note that *SAV and *RCL use one based register numbers (1..10) while most other commands use 0 based (0..9) register numbers.

Examples:

```
0000083 → MEM:STAT:CAT?
0000084 ←
"State0","State1","State2","State3","State4","State5","State6","State7","State8","State9"
0000088 → MEMORY:STATE:CATALOG?
0000089 ←
"State0","State1","State2","State3","State4","State5","State6","State7","State8","State9"
```

MEMory:STATe:DEFine

Syntax:

Most common forms:

```
MEM:STAT:DEF <string>, <number>
MEM:STAT:DEF? <string>
```

Long forms:

```
MEMORY:STATE:DEFINE <string>, <number>
MEMORY:STATE:DEFINE? <string>
```

Description:

This command either sets the association between a name and a register or state number or recalls the numeric half of the association given the name. In essence, this command names a numbered state. Or it recalls the number of a named state. The state numbers for this and most other commands is a 0 based (0…9) numbering system. However, the *SAV and *RCL commands that use a 1 based numbering system (1…10).

Examples:

In this example the state catalog is first listed. Then we make a change to the name of the fifth state (number 4) and then catalog the states again.

```
0000200 → MEM:CAT:STAT?
0000201 ←
288,2880,"SETUP33,STAT,288","State1,STAT,0","STATE_1,STAT,0","State3,STAT,0","State4,STAT,0",
"State5,STAT,0","State6,STAT,0","State7,STAT,0","State8,STAT,0","State9,STAT,0"
0000202 → MEM:STAT:DEF "NEW_NAME_4", 4
0000203 → MEM:CAT:STAT?
0000204 ←
288,2880,"SETUP33,STAT,288","State1,STAT,0","STATE_1,STAT,0","State3,STAT,0","NEW_NAME_4,STAT
,0","State5,STAT,0","State6,STAT,0","State7,STAT,0","State8,STAT,0","State9,STAT,0""
```

Common Error Messages:

If you rename a state using a state number that is outside the values of 0 to 9 you will get a 222 “Data out of range” error message.

MEMory:TABLe:FREQuency

Syntax:

Most common forms:

```
MEM:TABL:FREQ <frequency>,<frequency>,<frequency>, … <frequency>
```

Long forms:

```
MEMORY:TABLE:FREQUENCY <frequency>,<frequency>,<frequency>, … <frequency>
```

Description:

This command allows the user to enter a sorted frequency list into the currently selected table. The previous values in the selected table are cleared. As noted, the list of frequencies must be entered in ascending (sorted) order.

If a signal is measured, and the frequency as set by the user is outside the range of points the sensor selects the closest point. So if the set frequency is below the lowest point in the table, then sensor uses the first point in the table. If the frequency set frequency is above the last (highest) point in the table, then the last point will be used.

When entering the frequencies, the frequency can be entered without any units. In this case the units are assumed to be Hz. However you can enter the values with the units shown below. Also note, as with commands, these entries are case insensitive.

8. Hz
9. kHz
10. MHz
11. GHz

In any case, frequencies are truncated (not rounded) to the closest kHz.

Examples:

In this example the number of points is queried. The number returned is "+NAN". This indicates a table has not been selected as shown in subsequent commands. We eventually select a table ("CUSTOM_B") and query the number of frequency points again. At this point the value of zero is returned. Then we proceed to add frequency points. These points are checked. Then the same number of gain points are added and then checked.

```
0000025 → MEM:TABL:FREQ:POIN?
0000026 ← +NAN
0000027 → MEM:TABL:SEL?
0000028 ←
0000029 → MEM:CAT:TABL?
0000030 ←
0,4800,"CUSTOM_A,TABL,0","CUSTOM_B,TABL,0","CUSTOM_C,TABL,0","CUSTOM_D,TABL,0","CUSTOM_E,TABL
,0","CUSTOM_F,TABL,0","CUSTOM_G,TABL,0","CUSTOM_H,TABL,0","CUSTOM_I,TABL,0","CUSTOM_J,TABL,0"
0000031 → MEM:TABL:SEL "CUSTOM_B"
0000032 → MEM:TABL:FREQ:POIN?
0000033 ← +0.000000E+00
```

```
0000034 → MEM:TABL:FREQ 500MHZ,1GHZ,2GHZ,3GHZ
0000035 → MEM:TABL:FREQ?
0000036 ← 5.000000E+08,1.000000E+09,2.000000E+09,3.000000E+09
0000037 → MEM:TABL:FREQ:POIN?
0000038 ← +4.000000E+00
0000039 → MEM:TABL:GAIN 50.0, 100.0,150.0, 100.0
0000040 → MEM:TABL:GAIN:POIN?
0000041 ← +4.000000E+00
0000042 → MEM:TABL:GAIN?
0000043 ← 5.000000e+01,1.000000e+02,1.500000e+02,1.000000e+02
```

Common Error Messages:

12. Attempting to add more than 80 points results in error -108, "Parameter not allowed"
13. If the frequencies are not entered in ascending, this results in error -220, Parameter error: Frequency list must be in ascending order"
14. If a table has not been selected (MEM:TABL:SEL) then error -221 "Settings conflict" results
15. Any attempt to enter frequencies outside the allowable range (1kHz to 1000GHz) results in a -222 "Data out of range" error.

MEMory:TABLe:FREQuency:POINts?/qonly/

Syntax:

Most common forms:

```
MEM:TABL:FREQ:POIN?
```

Long forms:

```
MEMORY:TABLE:FREQUENCY:POINTS?
```

Description:

This command returns the number of frequency points in the currently selected table. If a table is not selected it returns +NAN.

Examples:

In this example the number of points is queried. The number returned is “+NAN”. This indicates a table has not been selected as shown in subsequent commands. We eventually select a table (“CUSTOM_B”) and query the number of frequency points again. Now a value of zero is returned. Then we proceed to add frequency points, gain points and rechecked the count in each case.

```
0000025 → MEM:TABL:FREQ:POIN?
0000026 ← +NAN
0000027 → MEM:TABL:SEL?
0000028 ←
0000029 → MEM:CAT:TABL?
0000030 ←
0,4800,"CUSTOM_A,TABL,0","CUSTOM_B,TABL,0","CUSTOM_C,TABL,0","CUSTOM_D,TABL,0","CUSTOM_E,TABL
,0","CUSTOM_F,TABL,0","CUSTOM_G,TABL,0","CUSTOM_H,TABL,0","CUSTOM_I,TABL,0","CUSTOM_J,TABL,0"
0000031 → MEM:TABL:SEL "CUSTOM_B"
0000032 → MEM:TABL:FREQ:POIN?
0000033 ← +0.000000E+00
0000034 → MEM:TABL:FREQ 500MHZ,1GHZ,2GHZ,3GHZ
0000035 → MEM:TABL:FREQ?
0000036 ← 5.000000E+08,1.000000E+09,2.000000E+09,3.000000E+09
0000037 → MEM:TABL:FREQ:POIN?
0000038 ← +4.000000E+00
0000039 → MEM:TABL:GAIN 50.0, 100.0,150.0, 100.0
0000040 → MEM:TABL:GAIN:POIN?
0000041 ← +4.000000E+00
0000042 → MEM:TABL:GAIN?
0000043 ← 5.000000e+01,1.000000e+02,1.500000e+02,1.000000e+02
```

Common Error Messages:

16. Attempting to add more than 80 points result in a -108, “Parameter not allowed” error
17. If the frequencies are not entered in ascending results in error -220
18. Any attempt to enter frequencies outside the allowable range (1kHz to 1000GHz) results in a -222 “Data out of range” error.

MEMory:TABLe:GAIN[:MAGNitude]

Syntax:

Most common forms:

```
MEM:TABL:GAIN <gain>,<gain>,<gain> … <gain>
MEM:TABL:GAIN?
```

Long forms:

```
MEMORY:TABLE:GAIN:MAGNITUDE <gain>,<gain>,<gain> … <gain>
MEMORY:TABLE:GAIN:MAGNITUDE?
```

Description:

This command allows the user to enter or query a sequence of offsets. These offsets are associated with the corresponding frequency, by order, in currently selected table. Any previous magnitude values in the selected table are cleared. ***All gain values are in percent.***

Note that FDO offsets describe the ***system response***. As a result these offsets be removed or subtracted from the uncorrected measured value to arrive at the corrected value.

For example, a system with a gain of 50% is interpreted as a 3.01dB loss. More correctly, the system response is -3.01dB (note the sign). This means that to arrive at the corrected value we must subtract -3.01dB from the uncorrected value.

So, assume we measured an uncorrected value of +10dBm. This uncorrected measured value includes the system response. To correct this value the system response must be removed or **subtracted** from the uncorrected value. The system response (which is -3.01dB or a 3.01dB loss) must be subtracted from +10dBm. This correction yields a corrected value of 13.01dBm. The arithmetic is as follows (note signs):

+10dBm **– (-3.01dB)** =

+10dBm **+ 3dB** = 13.0dBm.

If a signal is measured, and the frequency selected by the user is outside the range of FDO frequency points, the sensor selects the closest point. So if selected frequency is below the lowest point in the table, the sensor uses the first point in the table. Conversely, if the selected frequency is above the last (highest) point in the table, then the last point will be used.

Simple, straight line interpolation (frequency and Watts) is used for signals whose selected frequency falls within the bounds of FDO frequency points.

Again, when entering the correction the value units are assumed to be in percent. And the values reflect the system response. The maximum range of correction is 1 per cent to 150 per cent. To calculate or convert between offset (percent or dB) use on of the following:

System response in dB = 10 * Log_{10} (Offset in percent/100.0)

System response offset in per cent = 100.0 * $10.0^{(\text{offset in dB}/10.0)}$

You may find the following table to be a useful crosscheck:

Percent	dB
1	-20.0dB
10	-10.0dB
50	-3.01dB
75	-1.25dB
100	+0.00dB
125	+0.97dB
150	+1.76dB

The correction is applied in the following manner (dB):

corrected value$_{dbm}$ = uncorrected value$_{dbm}$ - FDO$_{dbm}$

For example, assume a value of -20.0dBm was measured before FDO correction. If FDO value of 50% would cause the -3.01dB of correction to be subtracted from the measured value. So, that -20dBm would be reported as -16.99dBm. In the same way, an FDO of 150% would cause +1.76 to be subtracted from -20dBm resulting in a corrected value as -21.76dBm.

Examples:

```
0000025 → MEM:TABL:FREQ:POIN?
0000026 ← +NAN
0000027 → MEM:TABL:SEL?
0000028 ←
0000029 → MEM:CAT:TABL?
0000030 ←
0,4800,"CUSTOM_A,TABL,0","CUSTOM_B,TABL,0","CUSTOM_C,TABL,0","CUSTOM_D,TABL,0","CUSTOM_E,TABL
,0","CUSTOM_F,TABL,0","CUSTOM_G,TABL,0","CUSTOM_H,TABL,0","CUSTOM_I,TABL,0","CUSTOM_J,TABL,0"
0000031 → MEM:TABL:SEL "CUSTOM_B"
0000032 → MEM:TABL:FREQ:POIN?
0000033 ← +0.000000E+00
0000034 → MEM:TABL:FREQ 500MHZ,1GHZ,2GHZ,3GHZ
0000035 → MEM:TABL:FREQ?
0000036 ← 5.000000E+08,1.000000E+09,2.000000E+09,3.000000E+09
0000037 → MEM:TABL:FREQ:POIN?
0000038 ← +4.000000E+00
0000039 → MEM:TABL:GAIN 50.0, 100.0,150.0, 100.0
0000040 → MEM:TABL:GAIN:POIN?
0000041 ← +4.000000E+00
0000042 → MEM:TABL:GAIN?
0000043 ← 5.000000e+01,1.000000e+02,1.500000e+02,1.000000e+02
```

MEMory:TABLe:GAIN[:MAGNitude]:POINts?/qonly/

Syntax:

Most common forms:

MEM:TABL:GAIN:POIN?

Long forms:

MEMORY:TABLE:GAIN:MAGNITUE:POINTS?

Description:

This command returns the number of points in the table currently selected for editing (using the **MEM:TABL:SEL** command).

Examples:

```
0000025 → MEM:TABL:FREQ:POIN?
0000026 ← +NAN
0000027 → MEM:TABL:SEL?
0000028 ←
0000029 → MEM:CAT:TABL?
0000030 ←
0,4800,"CUSTOM_A,TABL,0","CUSTOM_B,TABL,0","CUSTOM_C,TABL,0","CUSTOM_D,TABL,0","CUSTOM_E,TABL
,0","CUSTOM_F,TABL,0","CUSTOM_G,TABL,0","CUSTOM_H,TABL,0","CUSTOM_I,TABL,0","CUSTOM_J,TABL,0"
0000031 → MEM:TABL:SEL "CUSTOM_B"
0000032 → MEM:TABL:FREQ:POIN?
0000033 ← +0.000000E+00
0000034 → MEM:TABL:FREQ 500MHZ,1GHZ,2GHZ,3GHZ
0000035 → MEM:TABL:FREQ?
0000036 ← 5.000000E+08,1.000000E+09,2.000000E+09,3.000000E+09
0000037 → MEM:TABL:FREQ:POIN?
0000038 ← +4.000000E+00
0000039 → MEM:TABL:GAIN 50.0, 100.0,150.0, 100.0
0000040 → MEM:TABL:GAIN:POIN?
0000041 ← +4.000000E+00
0000042 → MEM:TABL:GAIN?
0000043 ← 5.000000e+01,1.000000e+02,1.500000e+02,1.000000e+02
```

MEMory:TABLe:MOVE/nquery/

Syntax:

Most common form:

```
MEM:TABL:MOVE <existing table name>,<new table name>
```

Long form:

```
MEMORY:TABLE:MOVE <existing table name>,<new table name>
```

Description:

This command is used to rename a FDO (frequency dependent offset) table.

Examples:

```
0000065 → MEM::CAT:TABL?
0000066 ←
24,4800,"CUSTOM_A,TABL,0","CUSTOM_B,TABL,24","CUSTOM_C,TABL,0","CUSTOM_D,TABL,0","CUSTOM_E,TA
BL,0","CUSTOM_F,TABL,0","CUSTOM_G,TABL,0","CUSTOM_H,TABL,0","CUSTOM_I,TABL,0","CUSTOM_J,TABL,
0"
0000067 → MEM:TABL:MOVE  "CUSTOM_B","CUSTOM_Z"
0000068 → MEM::CAT:TABL?
0000069 ←
24,4800,"CUSTOM_A,TABL,0","CUSTOM_Z,TABL,24","CUSTOM_C,TABL,0","CUSTOM_D,TABL,0","CUSTOM_E,TA
BL,0","CUSTOM_F,TABL,0","CUSTOM_G,TABL,0","CUSTOM_H,TABL,0","CUSTOM_I,TABL,0","CUSTOM_J,TABL,
0"
0000070 → MEM:TABL:MOVE  "CUSTOM_Z","CUSTOM_B"
0000071 → MEM::CAT:TABL?
0000072 ←
24,4800,"CUSTOM_A,TABL,0","CUSTOM_B,TABL,24","CUSTOM_C,TABL,0","CUSTOM_D,TABL,0","CUSTOM_E,TA
BL,0","CUSTOM_F,TABL,0","CUSTOM_G,TABL,0","CUSTOM_H,TABL,0","CUSTOM_I,TABL,0","CUSTOM_J,TABL,
0"
```

Common Error Messages:

19. If either table name is invalid this results in error -224, “Illegal parameter value”
20. If the first parameter does not match and existing table the error -226, “File name not found is issued.
21. If the second parameter matches an existing table then error -257,”File name error” is issued.

Other Notes:

The first parameter must match and existing file exactly. File names must consist of only upper and lower case letters (A…Z, a…z), the numbers 0…9 and the underscore. No other characters are permitted.

MEMory:TABLe:SELect

Syntax:

Most common forms:

```
MEM:TABL:SEL <table name>
MEM:TABL:SEL?
```

Long forms:

```
MEMORY:TABLE:SELECT <table name>
MEMORY:TABLE:SELECT?
```

Description:

This command selects an FDO (frequency dependent offset) table for editing using the memory commands.

Examples:

```
0000077 → MEM::CAT:TABL?
0000078 ←
24,4800,"CUSTOM_A,TABL,0","CUSTOM_B,TABL,24","CUSTOM_C,TABL,0","CUSTOM_D,TABL,0","CUSTOM_E,TA
BL,0","CUSTOM_F,TABL,0","CUSTOM_G,TABL,0","CUSTOM_H,TABL,0","CUSTOM_I,TABL,0","CUSTOM_J,TABL,
0"
0000079 → MEM:TABL:SEL?
0000080 ←
0000081 → MEM:TABL:SEL "CUSTOM_B"
0000082 → MEM:TABL:SEL?
0000083 ← CUSTOM_B
```

Output

The output commands are used to control recorder and trigger outputs. When recorder out is enabled, the sensor places DC voltage on the trigger out (TO) port that is proportional to the power in Watts. Trigger out sends a trigger out the TO port any time a measurement is made and trigger out is enabled. Since recorder out and trigger out share the same physical port they are by definition mutually exclusive.

OUTPut:RECorder:FEED

OUTPut:RECorder1:FEED

Syntax:

Most common forms:

```
OUTP:REC:FEED?
OUTP:REC:FEED CALC
```

Long forms:

```
OUTPUT:RECORDER:FEED?
OUTPUT:RECORDER:FEED CACL
OUTPUT:RECORDER:FEED CACL1
```

Description:

This command is included to support command compatibility with other sensors that support recorder out. It serves no additional purpose. The command takes and single parameter and that parameter must be CALC or CALC1 which are equivalent.

Examples:

This example shows setting and getting the parameter.

```
0000003 → OUTP:REC:FEED?
0000004 ← CALC
0000005 → OUTP:REC:FEED CALC
0000006 → OUTP:REC:FEED CALC1
0000007 → OUTP:REC:FEED?
0000008 ← CALC1
```

On Reset

The parameter must always be CALC or CALC1. These values are equivalent.

OUTPut:RECorder:FILTer

OUTPut:RECorder1:FILTer

Syntax:

Most common forms:

```
OUTP:REC:FILT <value>
OUTP:REC:FILT?
```

Long forms:

```
OUTPUT:RECORDER:FILTER <value>
OUTPUT:RECORDER1:FILTER <value>
OUTPUT:RECORDER:FILTER?
OUTPUT:RECORDER1:FILTER?
```

Description:

This command sets bandwidth of recorder out. Valid values for the bandwidth are between 0.001HZ to 32Hz inclusive. The default value is 32Hz. The maximum output value is 1V into a 1kOhm load (2.0V into an open). The filter is used to affect the reported value. The voltage at the output follows the reported value.

Example:

This is a long example. The setup for this example consists of connecting the sensor to a 1GHz RF source with a power level of 10.0dBm. During the example the power level should be changed between +10dBm and 0dBm. A DC voltmeter should be connected to the SMB TO or trigger output on the back of the sensor. The output can be considered unloaded since the impedance of the voltmeter is about >1MOhm.

The sensor is setup accomplished using the SPCI commands shown below. Finally, power is decreased by 3dB and continuous measurements are initiated simultaneously. The resultant measurements are about 1 second apart. This can be done in the interactive IO by:

22. Setting the latency to 1 sec
23. Sending one FETC? command
24. Then simultaneously
    - Checking continuous
    - Decrease the source 1GHz power level by 3dB simultaneously

After about 10 measurements, uncheck continuous to halt measurements. Then change the filter to 32Hz and repeat the process. In the first case using 0.1Hz, the measured power and measured voltage (not shown) start at 10dBm and 2Volts (open circuit). When the power drops by 3dB the reported value (resulting from continuous FETCH?) slowly decreases as does the voltage on the meter.

In the second case where the filter is set to 32Hz the change occurs rapidly.

```
0000082 → *RST
0000083 → FREQ?
```

```
0000084 ← +5.00000000E+07
0000085 → FREQ 1GHZ
0000086 → FREQ?
0000087 ← +1.00000000E+09
0000088 → INIT:CONT?
0000089 ← 0
0000090 → INIT:CONT 1
0000091 → FETC?
0000092 ← +1.00393940E+01
0000093 → OUTP:REC:FILT?
0000094 ← 3.200000E+01
0000095 → OUTP:REC:FILT 0.1
0000096 → OUTP:REC:FILT?
0000097 ← 1.000000E-01
0000098 → OUTP:REC:LIM:LOW?
0000099 ← -3.300000E+01
0000100 → OUTP:REC:LIM:LOW 0.0
0000101 → OUTP:REC:LIM:LOW?
0000102 ← +0.000000E+00
0000103 → OUTP:REC:LIM:UPP?
0000104 ← +2.000000E+01
0000105 → OUTP:REC:LIM:UPP 10.0
0000106 → OUTP:REC:LIM:UPP?
0000107 ← +1.000000E+01
0000108 → OUTP:REC:STAT?
0000109 ← 0
0000110 → OUTP:REC:STAT 1
0000111 → FETC?
0000112 ← +9.96684595E+00
0000113 → FETC?
0000114 ← +9.97565169E+00
```

The source and sensor frequency are set to FREQ = 1GHz. Sensor INIT:CONT = 1 is on so we can produce a measurement with just a FETC?, We've ensured source power level is +10dBm. This should produce 2VDC unloaded or 1VDC loaded at the TO (RO) output on the back of the sensor.

Set power level to +10dBm and wait 10 seconds. The no-load voltage should be 2VDC with the power level at +10dBm.

```
0000115 → FETC?
0000116 ← +9.98380112E+00
```

Set the power level to 0dBm. Wait 10 seconds. The voltage should be close to 0.0VDC. And the power level should be about 0.0dBm.

```
0000117 → FETC?
0000118 ← -1.26547706E-01
```

Now raise and lower the power level 10dB. Watch the RO voltage as you do so. You should see the voltage take several seconds to settle at either 0.0VDC or 2.0VDC. I've set the latency on the Interactive IO to either 0.5 or 1.0 seconds. To get a series of measurements with time between them. I check "Continuous" quickly after changing the power level. With this filter setting, after about 10 seconds I uncheck continuous.

In this case I've changed the source power level from +10.0dBm to 0.0dBm and very quickly checked "Continuous". At the end I unchecked "Continuous". I observed the voltmeter change also.

```
0000411 → FETC?
0000412 ← -9.63773193E-02
0000413 → FETC?
0000414 ← -9.70746011E-02
0000415 → FETC?
0000416 ← -9.72791906E-02
0000417 → FETC?
0000418 ← +6.04672479E+00
0000419 → FETC?
0000420 ← +8.35610990E+00
0000421 → FETC?
0000422 ← +9.21085784E+00
0000423 → FETC?
0000424 ← +9.60174160E+00
0000425 → FETC?
0000426 ← +9.79398714E+00
0000427 → FETC?
0000428 ← +9.89131474E+00
0000429 → FETC?
0000430 ← +9.94044006E+00
0000431 → FETC?
0000432 ← +9.96530503E+00
0000433 → FETC?
0000434 ← +9.97743852E+00
```

You can see the power level settled after several seconds. Lowering the source power again...

```
0000435 → FETC?
0000436 ← +9.98264508E+00
0000437 → FETC?
0000438 ← +8.01688527E+00
0000439 → FETC?
0000440 ← +5.78197966E+00
0000441 → FETC?
0000442 ← +3.88227759E+00
0000443 → FETC?
0000444 ← +2.40653472E+00
0000445 → FETC?
0000446 ← +1.37443741E+00
0000447 → FETC?
0000448 ← +7.16433882E-01
0000449 → FETC?
0000450 ← +3.30276201E-01
0000451 → FETC?
0000452 ← +1.17082104E-01
0000453 → FETC?
0000454 ← +4.15967830E-03
0000455 → FETC?
0000456 ← -5.38437541E-02
0000457 → FETC?
0000458 ← -8.21761081E-02
```

```
0000459 → FETC?
0000460 ← -9.49067693E-02
0000461 → FETC?
0000462 ← -9.97843608E-02
```

Again, it took several seconds for the power level to settle. Now change the filter to 32Hz.

```
0000463 → OUTP:REC:FILT 32.0
0000464 → OUTP:REC:FILT?
0000465 ← 3.200000E+01

0000470 → FETC?
0000471 ← -1.01244000E-01
0000472 → FETC?
0000473 ← +1.00062548E+01
```

Notice, in this case (filter set to 32Hz) the measured power goes from 0.0dBm to 10.0dBm very quickly.

On Reset

The filter value is set to 32Hz after a *RST.

Other Notes:

The range for the filter is 0.001Hz to 32Hz

OUTPut:RECorder:LIMit:LOWer

OUTPut:RECorder1:LIMit:LOWer

OUTPut:RECorder:LIMit:UPPer

OUTPut:RECorder1:LIMit:UPPer

Lower and upper limits of Recorder output operate as a pair, so it is fitting that they are covered as a pair in this section.

Syntax:

Most common forms:

```
OUTP:REC:LIM:LOW <value>
OUTP:REC:LIM:UPP <value>
OUTP:REC:LIM:LOW?
OUTP:REC:LIM:UPP?
```

Long forms:

```
OUTPUT:RECORDER:LIMIT:LOWER <value>
OUTPUT:RECORDER:LIMIT:UPPER <value>
OUTPUT:RECORDER:LIMIT:LOWER?
OUTPUT:RECORDER:LIMIT:UPPER?
OUTPUT:RECORDER1:LIMIT:LOWER <value>
OUTPUT:RECORDER1:LIMIT:UPPER <value>
OUTPUT:RECORDER1:LIMIT:LOWER?
OUTPUT:RECORDER1:LIMIT:UPPER?
```

Description:

The commands set the power measurement boundaries associated with recorder output voltage. The voltage output is a straight line interpolation of the lower and upper limits in Watts. The voltage out is between 0VDC and 1VDC into a 1kOhm load. If no load is attached then the voltage is twice this value or between 0VDC and 2VDC. An open can be approximated nicely with most DC voltmeters (high impedance).

An example calculation is shown below. The voltages are reported assuming the recorder output is properly loaded (1kOhm load). Note that all calculations use Watts.

$\text{Lim}_{\text{lower}}$ = 1mW (or +0.0dBm)

$\text{Lim}_{\text{upper}}$ = 10mW (or +10.0dBm)

Conditions:

25. Measured power between 1mW and 10mW
    - $V_{\text{out}} = (\text{Pmeas} - \text{Lim}_{\text{lower}})/(\text{Lim}_{\text{upper}} - \text{Lim}_{\text{lower}})$
    - So that for 5mW (or +6.99dBm):

      $V_{\text{out}} = (5.0 – 1.0)/(10.0 – 1.0) = 0.444$VDC into 1kOhm

If the voltage was measured across an open (such as a voltmeter) the voltage will be twice the calculated value or 0.888VDC

26. Measured power below Lim_{lower} or 1mW: V_{out} = 0.0VDC
27. Measured power above Lim_{upper} or 10mW: V_{out} = 1.0VDC (or 2.0VDC into an open)

Example:

In this example we connect the sensor to an RF source and set the power levels as measured below. A DC voltmeter was connected to the TO or recorder output on the back of the sensor. The calculations for the voltages were done exactly has shown earlier in this section.

```
0000487 → *RST
0000488 → OUTP:REC:FILT?
0000489 ← 3.200000E+01
0000490 → OUTP:REC:LIM:LOW?
0000491 ← -3.300000E+01
0000492 → OUTP:REC:LIM:LOW 0.0
0000493 → OUTP:REC:LIM:UPP?
0000494 ← +2.000000E+01
0000495 → OUTP:REC:LIM:UPP 10.0
0000496 → OUTP:REC:LIM:LOW?
0000497 ← +0.000000E+00
0000498 → OUTP:REC:LIM:UPP?
0000499 ← +1.000000E+01
0000500 → INIT:CONT?
0000501 ← 0
0000502 → INIT:CONT 1
0000503 → FETCH?
0000504 ← +6.92778023E+00
0000505 → OUTP:REC:STAT?
0000506 ← 0
0000507 → OUTP:REC:STAT 1
0000508 → FETCH?
0000509 ← +6.96345236E+00

Voltage measured using voltmeter was 0.891VDC.

+6.96345236E+00 -> 4.969mW

Calculated voltage = (4.969 - 1)/(10.0 - 1) = 0.441VDC into 1kOhm or
                                              0.882VDC into an open (as measured)

0000510 → FETCH?
0000511 ← +4.03776174E+00

Voltage measured using voltmeter was 0.344VDC

+4.03776174E+00 -> 2.534mW

Calculated voltage = (2.534 - 1)/(10.0 - 1) = 0.170VDC into 1kOhm or
                                              0.340VDC into an open (as measured)
```

On Reset

The lower limit is set to -30.0dBm and the upper limit is set to +20.0dBm.

OUTPut:RECorder:STATe

OUTPut:RECorder1:STATe

Syntax:

Most common forms:

```
OUTP:REC:STAT?
OUTP:REC:STAT <0 or 1>
```

Long forms:

OUTPUT:RECORDER:STATE?

Description:

This command checks the state or recorder out or disables or enables recorder out.

Examples:

In this series of commands the recorder output is turned on and off.

```
0000513 → *RST

Check the state of recorder out
0000514 → OUTP:REC:STAT?
0000515 ← 0

Set the state of recorder out to enabled
0000516 → OUTP:REC:STAT 1

Recheck the state
0000517 → OUTP:REC:STAT?
0000518 ← 1
```

On Reset

The recorder output default state is off or 0. And it is place in this state on *RST or power on.

Other Notes:

If recorder out is enabled and then trigger out is enabled recorder out is then disabled.

OUTPut:TRIGger:SLOPe

Syntax:

Most common forms:

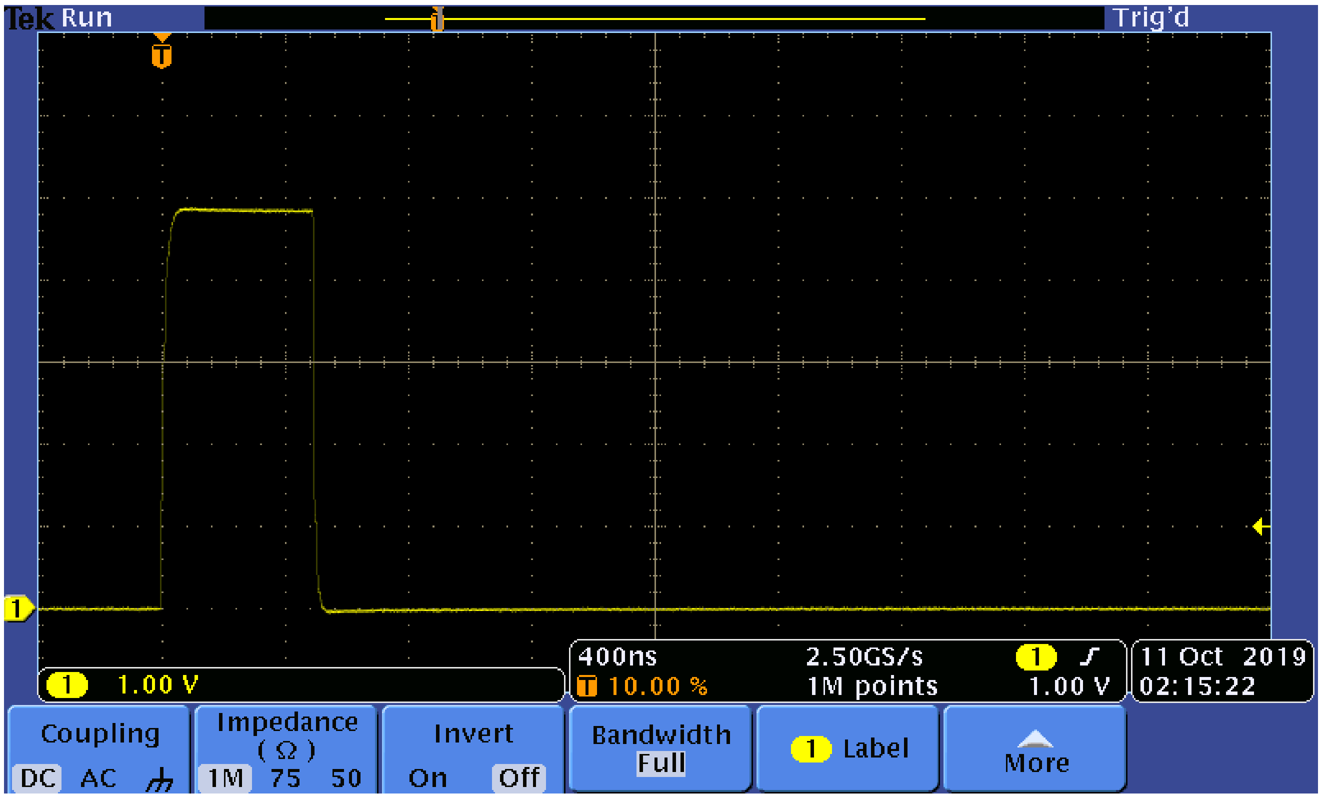
```
OUTP:TRIG:SLOP?
OUTP:TRIG:SLOP [NEG|POS]
```

Long forms:

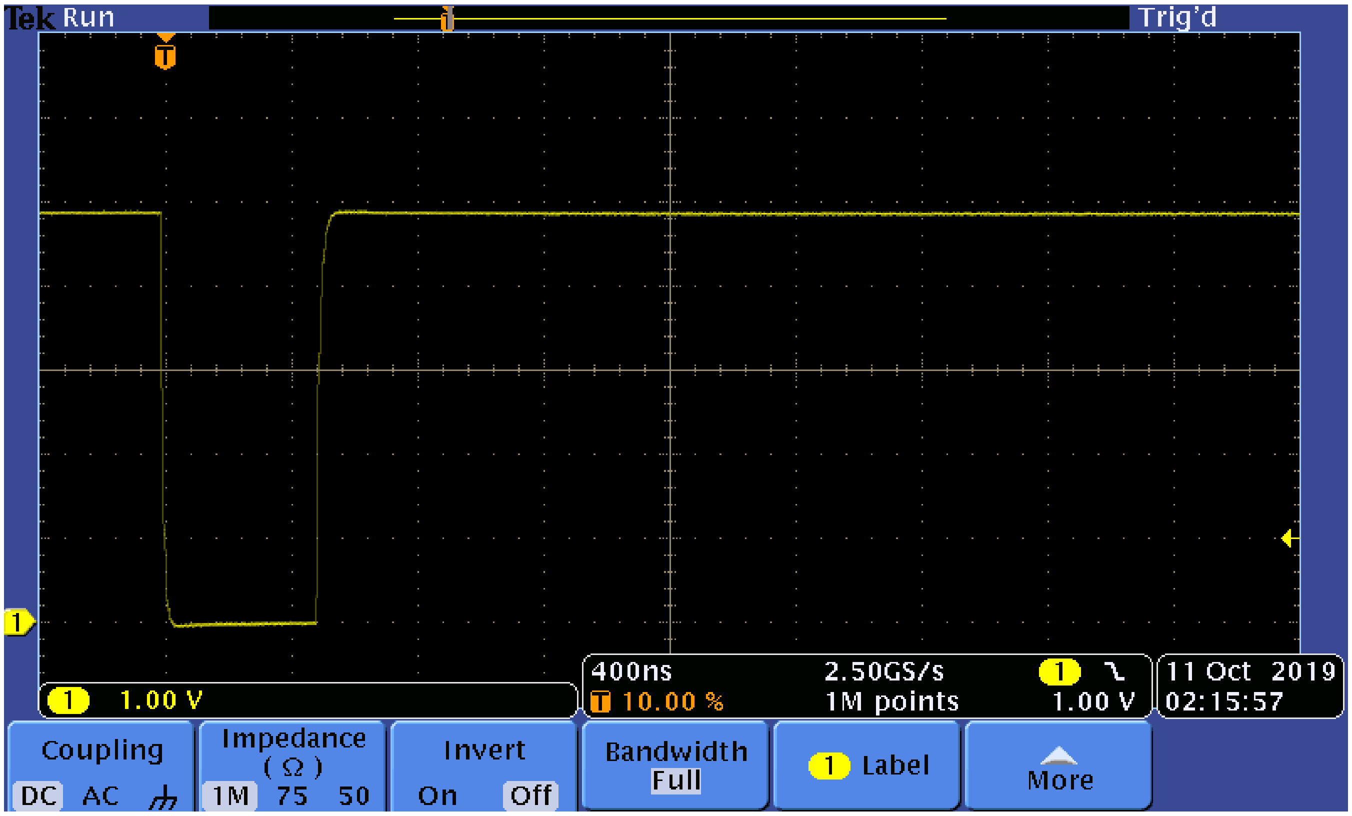
```
OUTPUT:TRIGGER:SLOPE?
OUTPUT:TRIGGER:SLOPE [NEG|POS]
```

Description:

This command determines whether the TTL compatible trigger out signal will present a negative or positive pulse when a measurement is made. If the pulse is positive then with no measurement the trigger out voltage will be 0V. When a measurement occurs (assuming trigger output is enabled) a positive going TTL compatible pulse is sent to the trigger out port. The pulse width is approximately 500ns as shown below.



If the trigger slope is negative, a negative going pulse will be placed on the output port as shown below.



Examples:

In this case we are checking and setting the trigger slope. Then doing a *RST and testing the default value after a reset.

```
0001111 → OUTP:TRIG:SLOP?
0001112 ← NEG
0001113 → OUTP:TRIG:SLOP POS
0001114 → OUTP:TRIG:SLOP?
0001115 ← POS
0001116 → OUTP:TRIG:SLOP NEG
0001117 → OUTP:TRIG:SLOP?
0001118 ← NEG
0001119 → *RST
0001120 → OUTP:TRIG:SLOP?
0001121 ← POS
```

OUTPut:TRIGger[:STATe]

Syntax:

Most common forms:

```
OUTP:TRIG:STAT?
OUTP:TRIG:STAT [0|1]
```

Long forms:

```
OUTPUT:TRIGGER:STATE?
OUTPUT:TRIGGER:STATE [0|1]
```

Description:

This command enables or disables the trigger out signal that is placed on the TO port. A trigger is generated each time a measurement is made.

Examples:

```
0001119 → *RST
0001120 → OUTP:TRIG:SLOP?
0001121 ← POS
0001122 → OUTP:TRIG:STAT?
0001123 ← 0
0001124 → OUTP:TRIG:STAT 1
0001125 → OUTP:TRIG:STAT?
0001126 ← 1
0001127 → OUTP:TRIG:STAT 0
0001128 → OUTP:TRIG:STAT?
0001129 ← 0
```

On Reset

On reset the trigger state is set to 0, OFF or disabled.

Other Notes:

Setting OUTP:TRIG:STAT to 1 or enabled will disable recorder out if it is disabled. In a like manner, enabling recorder out will disable OUTP:TRIG:STAT.

Sense

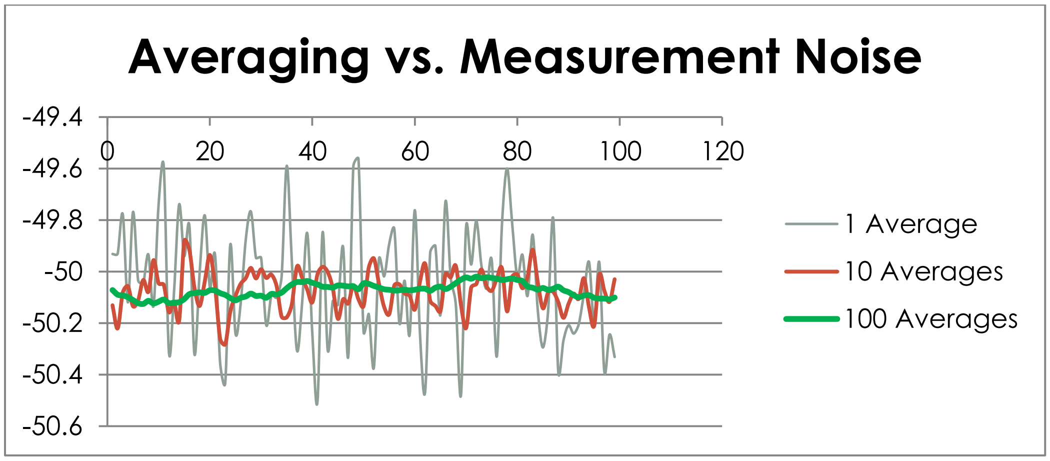
This group of commands controls the measurement parameters and processes. It includes control of the samples per average, many kinds of corrections, how much averaging is to be done, some aspects of triggering, setting up frequency and power sweeps, traces and many other aspects. Aside from the basic measurement commands (MEAS?, READ? and FETCH) this set of command are most central to the purpose of the power sensor.

It's important to understand that some of the commands place the sensor in a "mode". And that to return to default average power measurement mode you will need to explicitly exit the current mode of the sensor. It isn't difficult to do but it is very easy to overlook. The modes of the sensor and their relationships are as follows:

- Detector state = AVERAGE and its various modes (DET:FUNC AVER)
    - Default average power measurement mode – Returns a single average power measurement (BUFF:COUN = 1) that can be triggered:
        - When a command is received
        - Continuously (INIT:CONT = 1)
        - External TTL triggers
    - Buffered average power measurement mode using external trigger – Returns a multiple average power measurements (BUFF:COUN > 1). This is always triggered externally (TRIG:SOUR EXT).
    - Buffered average power measurements mode using immediate trigger – Returns multiple power measurements (TRIG:COUN >= 1). This is always triggered using IMMEDIATE trigger and MRAT = FAST. This mode is covered in the triggering section.
    - Frequency sweep mode – Returns a series of externally triggered average power measurements. Each of these measurements are assumed to occur at different frequencies. And so each measurement is frequency corrected using one of the frequency dependent offset tables. (FREQ:CW:START| STOP| STEP)
    - Power sweep mode – Returns a series of externally triggered average power measurements. (BUFF:COUN > 1). This is the same as buffered average power measurements. Measurements are triggered externally.
- Detector state = NORMAL and its various modes (DET:FUNC NORM)
    - Gated power measurements mode – Returns a single time gated power measurement using the wider bandwidth of the normal detector. This provides for internal or external triggering. The internal triggering is based on the level and slope of the incoming signal.
    - Trace mode – Returns a series of measurements using the wider bandwidth normal detector. The trace length and delay are settable over a wide range. The result is a series of power measurements that show a time domain representation of the incoming signal – or a trace. The traces come in a variety varying resolution.

Averaging Commands Overview

These commands control the measurement time. Measurement time and measurement noise are usually traded off against each other. As measurement time increases, measurement noise decreases. To give you an idea of how this might affect your measurement consider the following chart.



As averaging is increased from 1 average per point: to 10 averages per point: and finally to 100 averages the noise (point to point variation) of the measurement decreases. These measurements were made at about -50dBm. However, another consideration is time.

To understand the effects of time and averaging you may want to consider the following. I Using the InteractiveIO application by executing the following (RF source is set to -50dBm):

```
0000203 → *RST
0000204 → aver:coun:auto?
0000205 ← 1
0000206 → aver:coun:auto 0
0000207 → aver:coun:auto?
0000208 ← 0
0000209 → aver:coun?
0000210 ← +4
0000211 → aver:coun 1
0000212 → aver:coun?
0000213 ← +1
0000214 → SENS:AVER:SDET?
0000215 ← 1
0000216 → SENS:AVER:SDET 0
0000217 → read?
0000218 ← -5.01313042E+01
```

Then I created a simple read macro by highlighting the **read?** and adding a macro named “SimpleRead”. Then I repeated “SimpleRead” 20 times (Ctrl-T) for the following result.

```
-------   Start Macro [#SimpleRead#]
0000219 → read?
```

```
0000220 ← -5.03173258E+01
-------   End Macro
…
…
…
0000278 → read?
0000280 ← -5.00247631E+01
-------   End Macro [#SimpleRead#]
-------   #SimpleRead# was repeated 20 times in 1011 ms
```

I changed AVER:COUN to 10 then repeated "SimpleMacro" 20 time with the following result:

```
0000281 → aver:coun 10
-------   Start Macro [#SimpleRead#]
0000282 → read?
0000284 ← -5.00902647E+01
-------   End Macro
…
…
…
0000339 → read?
0000341 ← -5.00950042E+01
-------   End Macro [#SimpleRead#]
-------   #SimpleRead# was repeated 20 times in 7922 ms
```

Then I set AVER:COUN to 100 and repeated the process with the following result:

```
0000342 → aver:coun 100
-------   Start Macro [#SimpleRead#]
0000343 → read?
0000345 ← -5.01078571E+01
…
…
…
0000400 → read?
0000402 ← -5.01120167E+01
-------   End Macro [#SimpleRead#]
-------   #SimpleRead# was repeated 20 times in 77044 ms
```

The table below summarizes the impact of average count on measurement time:

AVER:COUN	Measurement Time (sec)
1	0.051
10	0.396
100	3.850

The increase in measurement time is proportional to the number of averages. AVER:COUN can be increased to 1000 or even to 4096. And, increasing it will decrease measurement noise. On the other hand, measurement time might become prohibitive. Still, for some situations, this increased measurement time is acceptable.

The following properties may affect total measurement time:

- `[SENSE:]AVERage:COUNt`
- `[SENSE:]AVERage:AUTO`
- `[SENSE:]AVERage:STATe`
- `[SENSE:]MRATe`
- `[SENSE:]AVERage:SDETect`
- `[SENSE:]BUFFer:COUNt`
- Triggering setup

All but the last one (triggering setup) will be covered in this, SENSE, section.

[SENSe]:AVERage:COUNt

SENSe1:AVERage:COUNt

Syntax:

Most common forms:

```
AVER:COUN?
AVER:COUN <NUM>
AVER:COUN? MIN
AVER:COUN? MAX
```

Long forms:

```
SENSE:AVERAGE:COUNT?
SENSE:AVERAGE:COUNT? MIN
SENSE:AVERAGE:COUNT? MAX
SENSE1:AVERAGE:COUNT?
SENSE1:AVERAGE:COUNT? MIN
SENSE1:AVERAGE:COUNT? MAX
SENSE:AVERAGE:COUNT <NUM>
SENSE1:AVERAGE:COUNT <NUM>
```

Description:

This sets or gets the number of averages per measurement. This command also accepts MIN and MAX as pass parameters. These values request the minimum and maximum values for AVER:COUN.

An average should not be confused with sample. Generally, an average is not equivalent to a sample. Generally, each average is the composed of several samples. The average property also interacts with the MRAT (measurement rate) property or command.

Examples:

```
0000203 → *RST
0000204 → aver:coun:auto?
0000205 ← 1
0000206 → aver:coun:auto 0
0000207 → aver:coun:auto?
0000208 ← 0
0000209 → aver:coun?
0000210 ← +4
0000211 → aver:coun 1
0000212 → aver:coun?
0000213 ← +1

0000445 → AVER:COUN? MIN
0000446 ← +1
0000447 → AVER:COUN? MAX
0000448 ← +4096
```

On Reset

On reset the average count is set to 4.

Common Error Messages:

If SENSE:MRATE is set to FAST and the user attempts to set averages, a -221 “Settings conflict” error message is issued. This is because MRAT FAST does not allow averaging. However, if SENSE:MRATE is set to SUPer then averaging parameter can be set without issue.

[SENSe]:AVERage:COUNt:AUTO

SENSe1:AVERage:COUNt:AUTO

Syntax:

Most common forms:

```
AVER:COUN:AUTO?
AVER:COUN:AUTO 0
AVER:COUN:AUTO 1
```

Long forms:

```
SENSE:AVERAGE:COUNT:AUTO?
SENSE:AVERAGE:COUNT:AUTO 0
SENSE:AVERAGE:COUNT:AUTO 1
SENSE1:AVERAGE:COUNT:AUTO?
SENSE1:AVERAGE:COUNT:AUTO 0
SENSE1:AVERAGE:COUNT:AUTO 1
```

Description:

This command allows the user to control the state of the automatic averaging (or auto-averaging) feature. This is also referred to as the auto-filter. This command also allows the user to query the state of auto-averaging. When enabled, the average count command rendered ineffective. So, the user is not required to set the average count explicitly. Instead, the sensor samples the incoming signal and adjusts the averaging based on the resolution specified by the user. The resolution is set as part of the MEASure? or CONFigure command . For a more thorough treatment of these commands refer to “The Basics of Making Power Measurements” in this document.

It is important to note that this command interacts with or is affected by the following parameters or commands:

- AVERAGE:STATE or AVER:STAT – enables or disables averaging and so that the state of AVER:COUN:AUTO is overridden but its value remains unchanged.
- AVER:STAT is enabled anytime this command or parameter to ON or 1
- Both MEAS? and CONF automatically enable AVER:STATE:AUTO
- AVERAGE:COUNT or AVER:COUN disables AVER:COUN:AUTO anytime AVER:COUN is set
- MRAT disallows AVER:COUN:AUTO to be enabled if MRAT = FAST or SUPER

The table below gives the averages for various power levels when this parameter is enabled:

Examples:

On Reset

A *RST command enables AVER:COUN:AUTO.

Common Error Messages:

Other Notes:

[SENSe]:AVERage:SDETect

SENSe1:AVERage:SDETect

Syntax:

Most common forms:

```
AVER:SDET?
AVER:SDET 1
AVER:SDET 0
```

Long forms:

```
SENSE:AVERAGE:SDETECT?
SENSE:AVERAGE:SDETECT 0
SENSE:AVERAGE:SDETECT 1
SENSE1:AVERAGE:SDETECT?
SENSE1:AVERAGE:SDETECT 0
SENSE1:AVERAGE:SDETECT 1
```

Description:

Step detection (SDET) is used to improve the chances of getting a more settled measurement. This is accomplished by monitoring the incoming signal. If the average power changes more than 12.5% (about 0.511dB) during the course of the measurement then the signal automatically is once re-measured. Note that this can increase the time required to return a value.

Note that this function works slightly differently in the LB59xx than it does in the U2000. The U2000 SDET function continues to re-measure as long as the signal appears unstable. The number of re-measures is unlimited. This situation can (easily) result in value is never being returned.

It can make the sensor seem to hang by never terminating the measurement process. This is easily be demonstrated with the U2000. can with the U2000 if the signal is noisy or low level (less than -55dBm). As noted, in these cases the measurement may never terminate and may appear to be hung.

To combat this prevent this hang, the LB59xx allows 1 re-measurement. And so the increase in measurement time is limited to doubling with the LB59xx. However, it is possible that the LB59xx will return an unsettled value. But it will return a value.

Examples:

In this example the source power is varied by 3dB during the course of the measurement. The average count is set long for purposes of demonstration. The measurement time is noted for each case with SDET set to 0 and 1. The increase in measurement time is easily detected with these settings. Notice that I've created a macro called INIT_READ. I set the count to 1. Then I selected the macro and pressed Ctrl-T so that the time to complete the macro is recorded.

```
This is just setup…

0000008 → *RST
0000009 → AVER:COUN:AUTO 0
0000010 → AVER:COUN 100
```

```
0000011 → MRAT NORM
Start with step detection disabled…
0000012 → AVER:SDET 0

During this measurement the power was left unchanged. And SDET was 0 or OFF
-------   Start Macro [#JUST_READ#]
0000013 → READ?
0000015 ← -1.03571152E+01
-------   End Macro [#JUST_READ#]
-------   #JUST_READ# was repeated 1 times in 3858 ms

I changed power during measurement by 3dB. Measurement time was unaffected. However, the
average power is incorrect.

-------   Start Macro [#JUST_READ#]
0000016 → READ?
0000018 ← -8.04517581E+00
-------   End Macro [#JUST_READ#]
-------   #JUST_READ# was repeated 1 times in 3852 ms

Now I'll enable step detection…
0000019 → AVER:SDET 1

And I changed power during the measurement by 3dB. Total time increased because SDET was
enabled but the average power reading is now correct.

-------   Start Macro [#INIT_READ#]
0000020 → READ?
0000022 ← -4.41788894E+00
-------   End Macro [#JUST_READ#]
-------   #JUST_READ# was repeated 1 times in 5070 ms
```

On Reset

Step detection is enabled by default on power up and reset.

[SENSe]:AVERage[:STATe]

SENSe1:AVERage[:STATe]

Syntax:

Most common forms:

```
AVER?
AVER 0
AVER 1
```

Long forms:

```
SENSE:AVERAGE:STATE?
SENSE:AVERAGE:STATE 0
SENSE:AVERAGE:STATE 1
SENSE1:AVERAGE:STATE?
SENSE1:AVERAGE:STATE 0
SENSE1:AVERAGE:STATE 1
```

Description:

This enables or disables averaging. This includes auto averaging, average count and step detection. This allows measurements to return very quickly so that measurements are often not settled. However, one common use of disabling averaging is to get a quick sense of the measured power level.

Examples:

In this example average count is set to 100 and averaging (AVER:STAT) is enabled then disabled. Note the dramatic change in measurement time.

```
0000023 → *RST
0000024 → AVER:COUN:AUTO 0
0000025 → AVER:COUN 100
0000026 → AVER:STAT?
0000027 ← 1
-------   Start Macro [#JUST_READ#]
0000028 → READ?
0000029 ← -4.43118460E+00
-------   End Macro [#JUST_READ#]
-------   Start Macro [#JUST_READ#]
0000030 → READ?
0000032 ← -4.43476595E+00
-------   End Macro [#JUST_READ#]
-------   #JUST_READ# was repeated 1 times in 3853 ms
0000033 → AVER:STAT 0
-------   Start Macro [#JUST_READ#]
0000034 → READ?
0000036 ← -4.43769685E+00
-------   End Macro [#JUST_READ#]
-------   #JUST_READ# was repeated 1 times in 51 ms
```

On Reset

Averaging is by default on at reset and power on.

Common Error Messages:

If AVER:STAT is set to 1 while MRAT is set to FAST. This message does not apply when MRAT is set to SUPER (or SUP).

[SENSe]:BUFFer:COUNt

SENSe1:BUFFer:COUNt

Syntax:

Most common forms:

```
BUFF:COUN?
BUFF:COUN <NUMBER)
```

Long forms:

```
SENSE:BUFFER:COUNT?
SENSE:BUFFER:COUNT? MIN
SENSE:BUFFER:COUNT? MAX
SENSE:BUFFER:COUNT <NUMBER>
```

Description:

Buffer count is used with external triggering. The range for buffer count is 1 to 250. Frequency sweep takes control of buffer count when it is enabled thereby causing buffer count to be overwritten.

Examples:

This example attempts to set buffer count with the trigger source set to INT (default). In this case buffer count does not change and an error occurs. Then I change the trigger source to external and find that buffer count can now be set without generating an error.

```
0000048 → *RST
0000049 → BUFF:COUN?
0000050 ← +1
0000051 → BUFF:COUN 100
0000052 → BUFF:COUN?
0000053 ← +1
0000054 → SYST:ERR?
0000055 ← -221,"Settings conflict"
0000056 → TRIG:SOUR EXT
0000057 → BUFF:COUN 100
0000058 → BUFF:COUN?
0000059 ← +100
```

On Reset

Buffer count is set to 1 by default.

Common Error Messages:

Settings conflict error messages can result from the trigger source being set incorrectly.

If FREQ:STEP is not equal to zero error message -221,"Settings conflict" will be generated

Other Notes:

This parameter is used by frequency sweep. So that the value of BUFF:COUN is read only unless FREQ:STEP = 0.

[SENSe]:CORRection:CSET2:STATe

SENSe1:CORRection:CSET2:STATe

[SENSe]:CORRection:CSET2[:SELect]

SENSe1:CORRection:CSET2[:SELect]

[SENSe]:CORRection:FDOFfset[:INPut][:MAGNitude]?/qonly/

SENSe1:CORRection:FDOFfset[:INPut][:MAGNitude]?/qonly/

[SENSe]:CORRection:GAIN4[:INPut][:MAGNitude]?/qonly/

SENSe1:CORRection:GAIN4[:INPut][:MAGNitude]?/qonly/

Syntax:

Most common forms:

```
CORR:CSET2 <TABLE NAME>
CORR:CSET2:STAT 0
CORR:CSET2:STAT 1
CORR:FDOF?
```

Long forms:

```
SENSE:CORRECTION:CSET2:SELECT <TABLE NAME>
SENSE:CORRECTION:CSET2:STATE 0
SENSE:CORRECTION:CSET2:STATE 1
SENSE:CORRECTION:FDOFFSET:INPUT:MAGNITUDE?
SENSE1:CORRECTION:CSET2:SELECT <TABLE NAME>
SENSE1:CORRECTION:CSET2:STATE 0
SENSE1:CORRECTION:CSET2:STATE 1
SENSE1:CORRECTION:FDOFFSET:INPUT:MAGNITUDE?
```

Description:

The CSET2 (FDOF) commands covered here apply to frequency dependent offset tables. They are covered as a set because they are used in concert. Other related commands are the MEM:TABL commands covered in considerable detail in the memory section of this manual. As with other STAT or STATE commands, the CORR:CSET2:STAT command enables and disables the frequency dependent offset table. While the CORR:CSET2 command selects one of 10 (0…9) tables.

Note that GAIN4 refers to FDOFFSET. And that CORR:FDOF? returns the FDO offset applied to the current measurement. This will be a value of 100.0 (meaning 100% or no offset) when GAIN2 is disabled. This function is handy for verifying your frequency dependent offset table.

Examples:

This example focuses on using CSET2 and FDO commands. See the MEM:TABL sections of this manual for additional information. In this example I've set up my source for about 0.0dBm. Note that CSET2 or FDO is the measurement system response. When FDO is enabled the reported value will be the measured value minus the system response. The table used in this example is "CUSTOM_A". When selecting a table enclose the name in double quotes as shown below.

```
0000282 → *RST

Which table is selected? We want CUSTOM_A but CUSTOM_B was previously selected
0000283 → CORR:CSET2?
0000284 ← CUSTOM_B

So we'll start by selecting CUSTOM_A
0000285 → CORR:CSET2 "CUSTOM_A"

For convenience, setup for quick measurements
0000286 → AVER:COUN:AUTO 0
0000287 → AVER:SDET 0
0000288 → AVER:COUN 10
0000289 → FREQ 1GHZ

What does the CUSTOM_A table look like?
0000290 → MEM:TABL:SEL?
0000291 ← CUSTOM_A
0000292 → MEM:TABL:FREQ?
0000293 ← 1.000000E+09,2.000000E+09,3.000000E+09,4.000000E+09
0000294 → MEM:TABL:GAIN?
0000295 ← 5.000000e+01,1.000000e+02,1.500000e+02,1.000000e+02

Check the state of FDO correction…it should be off and it is
0000296 → CORR:CSET2:STAT?
0000297 ← 0

Make a quick measurement and recheck our frequency…note that this is about -0.001dB
0000298 → READ?
0000299 ← -1.22661803E-03
0000300 → FREQ?
0000301 ← +1.00000000E+09

Check to see how much offset we are applying…should be none (100%) because FDO is disabled
0000302 → CORR:FDOF?
0000303 ← +1.00000000E+02

Now we'll enable FDO
0000304 → CORR:CSET2:STAT 1
0000305 → CORR:FDOF?

Recheck the current FDO offset (1GHZ). It should correspond to the table above and it does…
0000306 ← +5.00000000E+01

Now make a measurement…should be about 3.01dB and it is
0000307 → READ?
0000308 ← +3.00918434E+00
```

```
Change the frequency and make a new measurement…
0000309 → FREQ 2GHZ
0000310 → CORR:FDOF?
0000311 ← +1.00000000E+02
0000312 → READ?
0000313 ← +8.16211979E-02

Do it again (note 150% corresponds to about 1.76dB)…
0000314 → FREQ 3GHZ
0000315 → CORR:FDOF?
0000316 ← +1.50000000E+02
0000317 → READ?
0000318 ← -1.75841388E+00

…and again…
0000319 → FREQ 4GHZ
0000320 → CORR:FDOF?
0000321 ← +1.00000000E+02
0000322 → READ?
0000323 ← +1.68819594E-02

Return to 1GHz and recheck with FDO enabled
0000324 → FREQ 1GHZ
0000325 → CORR:FDOF?
0000326 ← +5.00000000E+01
0000327 → READ?
0000328 ← +3.01174029E+00

Disabled FDO and recheck…looks Ok
0000329 → CORR:CSET2:STAT 0
0000330 → READ?
0000331 ← +5.52826404E-04
```

On Reset

It is important to note that the state of these properties are unaffected by a *RST. In other words, if CSET2 is enabled before a *RST it will be enabled after a *RST.

Common Error Messages:

If you enable CSET2 (FDO) without a table being selected you'll get error -221 "Settings Conflict" and of course CSET2 will remain off. If you try to select a table that isn't present you'll get -256, "File name not found". If your name contains invalid characters (e.g. "#") you'll generate a -224, "Illegal parameter value".

Finally, the LB59xx error checks the table upon selection. It does this by comparing the number of frequency points to the number of gain/loss points. If the count differs this generates a -226, "Lists not the same length" error.

Other Notes:

The example code in the memory chapter can be helpful in understanding frequency dependent offset tables.

[SENSe]:CORRection:DCYCle:STATe

SENSe1:CORRection:DCYCle:STATe

[SENSe]:CORRection:DCYCle[:INPut][:MAGNitude]

SENSe1:CORRection:DCYCle[:INPut][:MAGNitude]

[SENSe]:CORRection:GAIN3:STATe

SENSe1:CORRection: GAIN3:STATe

[SENSe]:CORRection: GAIN3[:INPut][:MAGNitude]

SENSe1:CORRection: GAIN3 [:INPut][:MAGNitude]

Syntax:

Most common forms:

```
CORR:DCYC <NUMBER>
CORR:DCYC?
CORR:DCYC:STAT?
CORR:DCYC:STAT 0
CORR:DCYC:STAT 1
```

Long forms (a few):

```
SENSE:CORRECTION:DCYCLE:INPUT:MAGNITUDE <NUMBER>
SENSE:CORRECTION:DCYCLE:STATE?
SENSE:CORRECTION:DCYCLE:STATE 0
SENSE:CORRECTION:DCYCLE:STATE 1
SENSE:CORRECTION:GAIN3:INPUT:MAGNITUDE <NUMBER>
SENSE:CORRECTION:GAIN3:STATE?
SENSE:CORRECTION:GAIN3:STATE 0
SENSE:CORRECTION:GAIN3:STATE 1
```

Description:

In this command set, DCYCLE and GAIN3 are synonyms. This command is used to adjust the measured value by an assumed duty cycle. The duty cycle can take on a value of between 0.001 and 99.999 with PCT as optional units. So that both 10.01 and 10.01 PCT are acceptable and equivalent.

It is important to note that simply setting the value of duty cycle also enables duty cycle.

To calculate the offset in dB:

dB = 10.0 * log10(value in per cent/100.0)

The default value of 1% results in 20dB of correction. And 50% yields 3.01dB of correction.

Examples:

In this example the power level from the source is set to about 3dBm.

```
0000386 → *RST

Setup for quick measurements
0000387 → AVER:COUN:AUTO 0
0000388 → AVER:SDET 0
0000389 → AVER:COUN 10

Check the value of DCYCLE or duty cycle…it is 1 PCT.
0000390 → CORR:DCYC?
0000391 ← +1.000000E+00

Set it to 50.0 or 50 PCT
0000392 → CORR:DCYC 50 PCT
0000393 → CORR:DCYC?
0000394 ← +5.000000E+01
0000395 → CORR:DCYC 50.0
0000396 → CORR:DCYC?
0000397 ← +5.000000E+01

We didn't enable DCYCLE, yet it appears enabled. This was a result of setting the value
0000398 → CORR:DCYC:STAT?
0000399 ← 1

We'll turn it off and check the power level..3dB
0000400 → CORR:DCYC:STAT 0
0000401 → READ?
0000402 ← +3.01424083E+00

Turn it on (50% duty cycle) and we get a 3dB increase.
0000403 → CORR:DCYC:STAT 1
0000404 → READ?
0000405 ← +6.02054845E+00

Now we'll query the sensor for the minimum and maximum allowable values
0000406 → CORR:DCYC? MIN
0000407 ← +1.000000E-03
0000408 → CORR:DCYC? MAX
0000409 ← +9.999900E+01
```

On Reset

The value is set to 1% and the state is disabled upon *RST.

Common Error Messages:

If you enable the state while MRATE = FAST a -221, "Settings conflict" error is generated. If you set the value of DCYC with MRATE = FAST, the value will change but the state of duty cycle correction (CORR:DCYC:STAT) will not be enabled.

Other Notes:

Setting the value of GAIN3 or DCYCLE enables this feature.

[SENSe]:CORRection:GAIN2:STATe

SENSe1:CORRection:GAIN2:STATe

[SENSe]:CORRection:GAIN2[:INPut][:MAGNitude]

SENSe1:CORRection:GAIN2[:INPut][:MAGNitude]

Syntax:

Most common forms:

```
CORR:GAIN2?
CORR:GAIN2? MIN
CORR:GAIN2? MAX
CORR:GAIN2:STAT?
CORR:GAIN2 <VALUE>
CORR:GAIN2:STAT 0
CORR:GAIN2:STAT 1
```

Long forms (a few):

```
SENSE:CORRECTION:GAIN2:STATE?
SESNE:CORRECTION:GAIN2:STATE 0
SESNE:CORRECTION:GAIN2:STATE 1
SENSE:CORRECTION:GAIN2?
SESNE:CORRECTION:GAIN2 <VALUE>
```

Description:

As with many parameters you have the option of setting the value (CORR:GAIN2) and enabling or disabling the parameter (CORR:GAIN2:STAT). This parameter is allows the user to setup an "general" correction value. This can be used along with other forms of correction (e.g. FDO, MLP). This parameter is applied by addition. Unlike FDO where the values entered are the response of the system, this value is the actual correction that is to be applied to the measured value (by addition). CORR:GAIN2 allows a range -100dB to +100db.

Examples:

In this sequence, -5dB and +5dB of correction is applied and enabled and disabled

```
0000958 → *RST

Set up for quick measurements…
0000959 → AVER:COUN:AUTO 0
0000960 → AVER:SDET 0
0000961 → AVER:COUN 10

Make a measurement with no correction
0000962 → READ?
0000963 ← +2.99571307E+00

Verify the state of GAIN2 correction
0000964 → CORR:GAIN2:STAT?
```

```
0000965 ← 0
0000966 → CORR:GAIN2?
0000967 ← +0.000000E+00

Set GAIN2… also enables GAIN2:STAT. When read, the -5dB of correction is apparent.
0000968 → CORR:GAIN2 -5.0
0000969 → READ?
0000970 ← -2.00712283E+00
0000971 → READ?
0000972 ← -2.00735009E+00

You can see that GAIN:STAT has been enabled
0000973 → CORR:GAIN2:STAT?
0000974 ← 1

Change the CORR:GAIN to +5 dB of correction…you can see it works properly
0000975 → CORR:GAIN2 5.0
0000976 → READ?
0000977 ← +7.99374313E+00
0000978 → READ?
0000979 ← +7.99407945E+00
0000980 → READ?
0000981 ← +7.99338090E+00

Turn off or disable CORR:GAIN2 and the power reading shows the uncorrected value
0000982 → CORR:GAIN2:STAT 0
0000983 → READ?
0000984 ← +2.99642488E+00

These commands demonstrate the option of getting MIN and MAX at runtime
0000985 → CORR:GAIN2? MAX
0000986 ← +1.000000E+02
0000987 → CORR:GAIN2? MIN
0000988 ← -1.000000E+02
```

On Reset

GAIN2 is set to 0.0 and the STATE is disabled.

Common Error Messages:

A -221, “Settings conflict” message is generated when MRAT = FAST

SENSe:CORRection:MLPad[:INPut]:STATe

SENSe1:CORRection:MLPad[:INPut]:STATe

Syntax:

Most common forms:

```
SENS:CORR:MLP:STAT?
SENS:CORR:MLP:STAT 0
SENS:CORR:MLP:STAT 1
```

Long forms:

```
SENSE:CORRECTION:MLPAD:INPUT:STATE?
SENSE:CORRECTION:MLPAD:INPUT:STATE 0
SENSE:CORRECTION:MLPAD:INPUT:STATE 1
SENSE1:CORRECTION:MLPAD:INPUT:STATE?
SENSE1:CORRECTION:MLPAD:INPUT:STATE 0
SENSE1:CORRECTION:MLPAD:INPUT:STATE 1
```

Description:

This function is unique to the LB59xx. If enabled, it applies a correction for a 75Ohm to 50 Ohm minimum loss pad correction of 5.719dB. This is the power lost in the MLP impedance matching devices. The correction is additive.

Examples:

```
Start with a reset and then setup for fast measurements
0001051 → *RST
0001052 → AVER:COUN:AUTO 0
0001053 → AVER:SDET 0
0001054 → AVER:COUN 10

Make a measurement
0001055 → READ?
0001056 ← +2.98555903E+00
Check the MLP…it's disabled
0001057 → SENS:CORR:MLP:STATE?
0001058 ← 0

Enable MLP and re-measure. Note the change in the measured value
0001061 → SENS:CORR:MLP:STATE 1
0001062 → READ?
0001063 ← +8.70673521E+00

Reset and note that the MLP was not disabled with a *RST
0001064 → *RST
0001065 → READ?
0001066 ← +8.70707513E+00
0001067 → SENS:CORR:MLP:STATE 0
0001068 → READ?
0001069 ← +2.98864140E+00
```

On Reset

Be aware that the state of MPL is unchanged by a *RST.

[SENSe]:DETector:FUNCtion

SENSe1:DETector:FUNCtion

Syntax:

Most common forms:

```
DET AVER:NORM
```

Long forms:

```
SENSE:DETECTOR:FUNCTION AVERAGE|NORMAL
```

Description:

For most measurements the detector is set to average. However, for traces and sweep commands the detector is set to NORMAL. The normal detector has a wider bandwidth and so it can be useful in time domain measurements.

Examples:

```
0000255 → *RST
0000256 → DET:FUNC?
0000257 ← AVER
0000258 → DET:FUNC NORM
0000259 → DET:FUNC?
0000260 ← NORM
0000261 → DET:FUNC AVER
0000262 → DET:FUNC?
0000263 ← AVER
```

On Reset

The detector is set to AVERAGE.

Common Error Messages:

If this command is used with the LB59xxL models a -241,"Hardware missing" error message will be generated.

Other Notes:

The detector function has two states:

Average – when selected

- TRIG:SOUR is automatically set to IMMEDIATE if internal or external is selected
- INIT:CONT is automatically set to 1 or ON
- CALC:FEED is automatically set to "POW:AVER"

Normal – when selected

- Duty cycle state is set to off
- Trigger source is set to internal if was set to immediate
- CALC:FEED is automatically set to "POW:AVER ON SWEEP1"

[SENSe]:FREQuency[:CW]

SENSe1:FREQuency[:CW]

[SENSe]:FREQuency[:FIXed]

SENSe1:FREQuency[:FIXed]

Syntax:

Most common forms:

```
FREQ?
FREQ? MIN
FREQ? MAX
FREQ <NUMBER>
```

Long forms:

```
SENSE:FREQUENCY:CW?
SENSE:FREQUENCY:CW? MIN
SENSE:FREQUENCY:CW? MAX
SENSE:FREQUENCY:CW? DEF
SENSE:FREQUENCY:CW <NUMBER>
SENSE1:FREQUENCY:CW? MIN
SENSE1:FREQUENCY:CW? MAX
SENSE1:FREQUENCY:CW? DEF
SENSE1:FREQUENCY:CW <NUMBER>
```

Note that CW and FIXED are synonyms in this set of commands. So that SENSE:FREQUENCY:CW? And SENSE:FREQUENCY:FIXED? Are equivalent.

Description:

This is used to set the frequency. This is then used to correct the measured value for the frequency response of the sensor. Units can be appended to the value. The applicable units are Hz, kHz, MHz and GHz. When frequency is set the FDO is recalculated and applied to any measurement.

Examples:

```
What happens with *RST…
0001093 → *RST
0001094 → FREQ?
0001095 ← +5.00000000E+07

Lower case units…
0001096 → FREQ 10ghz
0001097 → FREQ?
0001098 ← +1.00000000E+10

Upper case units
0001099 → FREQ 11GHZ
0001100 → FREQ?
0001101 ← +1.10000000E+10

Get the minimum and maximum frequency supported by this sensor
0001102 → FREQ? MIN
```

```
0001103 ← +9.00000000E+03
0001104 → FREQ? MAX
0001105 ← +2.65000000E+10
```

On Reset

A *RST sets frequency to 50MHz.

[SENSe]:FREQuency[:CW|FIXED]:STARt

SENSe1:FREQuency[:CW|FIXED]:STARt

[SENSe]:FREQuency[:CW|FIXED]:STOP

SENSe1:FREQuency[:CW|FIXED]:STOP

[SENSe]:FREQuency[:CW|FIXED]:STEP

SENSe1:FREQuency[:CW|FIXED]:STEP

Note: These commands require an external trigger. This is often accomplished by using an RF source that frequency steps. These sources often output a trigger for each frequency step. Of course the trigger must be connected to the trigger input of the sensor. The sensor synchronizes its measurements to the incoming trigger by making a measurement at the next specified frequency when a trigger is received.

Syntax:

Most common forms:

```
FREQ:STAR?
FREQ:STAR? DEF|MIN|MAX
FREQ:STAR <number>
FREQ:STOP?
FREQ:STOP? DEF|MIN|MAX
FREQ:STOP <number>
FREQ:STEP?
FREQ:STEP? DEF|MIN|MAX
FREQ:STEP <number>
```

Long forms:

```
SENSE:FREQUENCY:FIXED:START?
SENSE:FREQUENCY:FIXED:STOP?
SENSE:FREQUENCY:FIXED:STEP?
```

Description:

These commands cause the sensor to make a series of measurements and then deliver this same series of measurements as a group or buffer. The number of measurements delivered in the buffer is determined by FREQ:STEP. Two other commands, FREQ:START and FREQ:STOP, set the end points of the sweep.

These commands are very specific in how they operate. It is essential to understand that setting FREQ:STEP to a positive value (between 1 and 250) will place the sensor in “frequency sweep mode”. To resume average power measurements you must set FREQ:STEP back to 0 (default value). Finally, each new sweep requires that *OPC followed by setting FREQ:STEP. The requirement to set FREQ:STEP must be met even if the current value is identical previous value (see the example below). The sensor firmware uses the setting of *OPC and FREQ:STEP as a signal that a new sweep should commence.

Normally, at implementation the start and stop frequencies are known. The variability is normally associated with the number of steps or interval. Given a specific start and stop and number of steps, an evenly spaced interval can be calculated as follows:

$f_{interval} = (f_{stop} - f_{start})/(step - 1)$

For example, if the start frequency was 540MHz and the stop frequency was 1GHz and 100 steps were desired the interval would be (frequencies in MHz):

$f_{interval} = (1000 - 540)/(100-1)$ **or** $460/99 \Rightarrow 4.646$**MHz**

Notice that the step frequency is rounded to the nearest kHz. This is what the sensor does. Given a specific start and stop and interval, the number of steps can be calculated as follows:

$step = (f_{stop} - f_{start} + f_{interval})/(f_{interval})$

Using the example above except this time we'll use the start, stop and interval we get:

$step = (f_{stop} - f_{start} + f_{interval})/(f_{interval})$ **or** $(1000 - 540 + 4.646)/4.646 \Rightarrow 100$

Where:

f_{start} = the start frequency

f_{stopt} = the stop frequency

$f_{intervalt}$ = the size or interval between steps

step = the number of steps

Examples:

In this example the sensor is swept twice. The first time a complete setup is done. In the next sweep *OPC and FREQ:STEP are used to repeat the sweep. After this, frequency sweep mode is exited and an average power measurement is made using immediate triggering. Note that the trigger source is returned to immediate and the frequency step must be set to 0.

```
Start from a known state…
0000335 → *RST

Start the setup for the first sweep…we want relatively quick measurement for the demo…
0000336 → AVER:COUN:AUTO 0
0000337 → AVER:SDET 0
0000338 → AVER:COUN 10

Setup for external triggering…
0000339 → TRIG:SOUR EXT
0000340 → TRIG:SLOP POS

…and then the start and stop freuqencies
0000341 → FREQ:STAR 500MHZ
```

```
0000342 → FREQ:STOP 1000MHZ

The sensor must be set to immediately watch for the next trigger…
0000343 → INIT:CONT 1

Set the operation complete bit so *ESR? tells us when the sweep is done…
0000344 → *OPC

…query *ESR until you get a 0…then *ESR? will be watching for us.
0000345 → *ESR?
0000346 ← +1
0000347 → *ESR?
0000348 ← +0

To start the sweep set FREQ:STEP…
0000349 → FREQ:STEP 10

Now that the sweep is started we will repeatedly query *ESR?
0000350 → *ESR?
0000351 ← +0
0000352 → *ESR?
0000353 ← +0
…
…
0000366 → *ESR?
0000367 ← +0

Finally *ESR? tells us the sweep is done by returning a 1
0000368 → *ESR?
0000369 ← +1

Now we'll get the data
0000370 → FETCH?
0000371 ←
+2.94186480E+00,+2.94203762E+00,+2.94395445E+00,+2.94310356E+00,+2.94242219E+00,+2.94087565E+
00,+2.94188366E+00,+2.94354162E+00,+2.94251383E+00,+2.94232040E+00
Ok, now get the next sweep by setting *OPC…
0000372 → *OPC
…then frequency step
0000373 → FREQ:STEP 10

Now we query *ESR? Until it returns 1
0000374 → *ESR?
0000375 ← +0
0000376 → *ESR?
0000377 ← +0
…
…
0000384 → *ESR?
0000385 ← +0

All done
0000386 → *ESR?
0000387 ← +1
```

```
Now we get the data
0000388 → FETCH?
0000389 ←
+2.94371970E+00,+2.94248834E+00,+2.94154475E+00,+2.94166740E+00,+2.94263360E+00,+2.94317034E+
00,+2.94361075E+00,+2.94070365E+00,+2.94215700E+00,+2.94503756E+00

Finally, we exit frequency sweep mode and make an average measurement
0000390 → FREQ:STEP 0
0000391 → TRIG:SOUR IMM
0000392 → FETCH?
0000393 ← +2.94474820E+00
```

On Reset

Common Error Messages:

Other Notes:

[SENSe]:MRATe

SENSe1:MRATe

[SENSe]:SPEed

SENSe1:SPEed

Syntax:

Most common forms:

```
MRAT NORM
MRAT DOUB
MRAT FAST
MRAT SUP
MRAT?
```

Long forms:

```
SENSE:MRATE NORMAL
SENSE:MRATE DOUBLE
SENSE:MRATE FAST
SENSE:MRATE SUPER
SENSE:MRATE?
```

Description:

The measurement rate or MRAT setting determines the rate of averaging or number of samples per average. As you move from NORM to DOUB to FAST the number of samples per average decreases. As a result, the number of completed measurements per second increases. So that if the number of averages is set to 1 (AVER:COUN 1) then the following applies:

MRATE	Maximum readings per second
NORMAL	20
DOUBLE	40
FAST	110
SUPER	110

Higher read rates can be achieved with trigger counts of 50 (using buffers).

Note that the U2000 does not allow the number of averages to be set when MRAT = FAST. Trying to set the number of averages with MRAT = FAST will generate a -221, “Settings confict” error message. To avoid this you can set MRAT = SUPER. Super is in every way identical to fast except it allows you to set the number of averages.

Note: The SENS:SPEED commands are included here for compatibility purposes only. MRATE is the preferred command. SENS:SPEED takes (or returns) a numeric parameter. The numeric parameters are 20, 40 and 110.

Examples:

These examples were executed in the Interactive IO. Any deviation from the read rates noted above is a result of software not the sensor. But reasonable approximations of this table can be achieved using the Interactive IO application.

```
0001884 → *RST
0001885 → AVER:COUN:AUTO 0
0001886 → AVER:SDET 0
0001887 → AVER:COUN 1
0001888 → MRAT NORM
-------   Start Macro [#JUST_READ#]
0001889 → READ?
0001891 ← +2.95001684E+00
-------   End Macro [#JUST_READ#]
-------   Start Macro [#JUST_READ#]
0001892 → READ?
0001894 ← +2.95477319E+00
-------   End Macro [#JUST_READ#]
…
…
…
…
0001939 ← +2.94952262E+00
-------   End Macro [#JUST_READ#]
-------   Start Macro [#JUST_READ#]
0001940 → READ?
0001942 ← +2.94993072E+00
-------   End Macro [#JUST_READ#]
-------   Start Macro [#JUST_READ#]
0001943 → READ?
0001945 ← +2.94988765E+00
-------   End Macro [#JUST_READ#]
-------   Start Macro [#JUST_READ#]
0001946 → READ?
0001948 ← +2.94975026E+00
-------   End Macro [#JUST_READ#]
-------   #JUST_READ# was repeated 20 times in 1027 ms
```

On Reset

MRATE is set to NORMAL on reset.

Common Error Messages:

As stated earlier, if you attempt to set the average count with MRATE = FAST you will get a -221, “Settings conflict” message. If the SPEED command is used and the value of the parameter is not 20, 40 or 110 a -224, ”Illegal parameter” message is generated.

[SENSe]:POWer:AC:RANGe:AUTO

SENSe1:POWer:AC:RANGe:AUTO

[SENSe]:POWer:AC:RANGe

SENSe1:POWer:AC:RANGe

Syntax:

Most common forms:

```
POW:AC:RANG:AUTO?
POW:AC:RANG:AUTO 1
```

Long forms:

```
SENSE:POWER:AC:RANGE:AUTO?
SENSE:POWER:AC:RANGE:AUTO 1
SENSE:POWER:AC:RANGE 0
SENSE:POWER:AC:RANGE 1
```

Description:

POW:AC:RANG is used to select the upper or lower range manually. Control over the selected path could be valuable when measuring very narrow pulsed signals. The value 0 selects the lower range (less then about -15dBm) and 1 selects the upper range (greater than about -15dBm). If you select either range POW:AC:RANGE:AUTO is automatically disabled. You will need to explicitly enable POW:AC:RANG:AUTO (or issue a *RST) to re-enable this feature.

Examples:

In this measurements are made using each range within its useable range, at its limit and beyond. Finally, automatic range selection is re-enabled and measurements are repeated.

```
Start from a known state…
0000105 → *RST
0000106 → POW:AC:RANG:AUTO?
0000107 ← 1
0000108 → POW:AC:RANG?
0000109 ← 1
0000110 → POW:AC:RANG 0
0000111 → POW:AC:RANG:AUTO?
0000112 ← 0

Set source power to 0dBm
0000113 → INIT:CONT?
0000114 ← 0
0000115 → INIT:CONT 1
0000116 → FETCH?
0000117 ← -9.72820365E+00
0000118 → POW:AC:RANG 1
0000119 → FETCH?
0000120 ← -4.56114304E-01

Set source power to -30dBm
```

```
0000121 → FETCH?
0000122 ← -3.04687767E+01
0000123 → POW:AC:RANG 0
0000124 → FETCH?
0000125 ← -3.04975394E+01
0000126 → POW:AC:RANG 1
0000127 → FETCH?
0000128 ← -3.03713741E+01

Set source power to -40dBm
0000129 → FETCH?
0000130 ← -3.21224785E+01
0000131 → POW:AC:RANG 0
0000132 → FETCH?
0000133 ← -4.05227654E+01

Re-enable AUTO RANGE selection
0000134 → POW:AC:RANG:AUTO 1

Source power is still set to -40dBm
0000135 → FETCH?
0000136 ← -4.05205412E+01
0000137 → FETCH?
0000138 ← -4.05280215E+01

Source power to 0 dBm
0000139 → FETCH?
0000140 ← -4.58481196E-01
0000141 → FETCH?
0000142 ← -4.58466587E-01
```

Sweep Commands

The sweep commands use the normal detector (10kHz to 40kHz bandwidth) instead of the average detector (<100Hz bandwidth). In this way, time gated power measurements are supported. The key commands are selecting the detector (NORMAL), setting the trigger source to INTERNAL or EXTERNAL, setting the trigger level, sweep time (measurement time) and offset time (offset from trigger. Finally the CALC:FEED should be set to "POW:AVER ON SWEEP". However this should be done automatically when the NORMAL detector is selected.

The measurement reports back the average power over the specified sweep time. The sweep time starts after a trigger plus the offset time. Again, this uses the NORMAL detector. So that measurements below about -45dBm become difficult.

Finally, these measurements are often referred to as gated power measurements.

[SENSe[1]]:SWEep[1]:TIME

[SENSe[1]]:SWEep[1]:OFFSet:TIME

Syntax:

Most common forms:

```
SWE:TIME?
SWE:TIME? MIN
SWE:TIME? MAX
SWE:TIME <number>
SWE:OFFS:TIME?
SWE:OFFS:TIME <number>
```

Long forms:

```
SENSE:SWEEP:TIME?
SENSE:SWEEP:TIME <number>
SENSE:SWEEP:OFFSET:TIME?
SENES:SWEEP:OFFSET:TIME <number>
```

Description:

Sweep time and sweep offset time define a measurement period offset from a trigger event. Sometimes this is referred to as a gate The trigger can be an internal trigger or external trigger. The sweep time can range from 100us to 0.15s in increments of 1us. The sweep offset time can range from -0.15 to +0.15 seconds with a resolution of 1us. In all cases the detector function must be NORMAL.

Example:

In this code a series of measurement are taken. The sweep is triggered on the waveform. The waveform is a 1kHz, 90% AM modulated signal. The trigger level is set to 0.0dBm. And the sweep is 100us and the sweep offset is walked along 1 whole cycle in 100us increments.

```
Start from a known state
0000472 → *RST

Setup for gated measurements
0000473 → DET:FUNC NORM
0000474 → TRIG:SLOP POS
0000475 → TRIG:SOUR INT

The signal is a 10dBm AM signal, 90% AM, 1kHz sine wave, so we'll use a trigger level of 3dBm
0000476 → TRIG:LEV 3.0
This is a 100us gate with an offset of 100us (from the trigger)
0000477 → SWE:TIME 0.0001
0000478 → SWE:OFFS:TIME 0.0001
0000479 → CALC:FEED?
0000480 ← "POW:AVER ON SWEEP1"

Set the sweep offset and execute another sweep
```

```
0000484 → SWE:OFFS:TIME 0.002
0000485 → TRIG:SOUR INT

This is 8.8dBm
0000486 → FETCH?
0000487 ← +8.82669904E+00

Increment the sweep offset by 100us and measure 14.4dBm
0000488 → SWE:OFFS:TIME 0.0002
0000489 → TRIG:SOUR INT
0000490 → FETCH?
0000491 ← +1.43773951E+01

Increment the sweep offset by 100us and measure 15.5dBm
0000492 → SWE:OFFS:TIME 0.0003
0000493 → TRIG:SOUR INT
0000494 → FETCH?
0000495 ← +1.54545783E+01

Increment the sweep offset by 100us and measure 14.6dBm
0000496 → SWE:OFFS:TIME 0.0004
0000497 → TRIG:SOUR INT
0000498 → FETCH?
0000499 ← +1.46409066E+01

Increment the sweep offset by 100us and measure 12.1dBm
0000500 → SWE:OFFS:TIME 0.0005
0000501 → TRIG:SOUR INT
0000502 → FETCH?
0000503 ← +1.20742272E+01

Increment the sweep offset by 100us and measure 7.4dBm
0000504 → SWE:OFFS:TIME 0.0006
0000505 → TRIG:SOUR INT
0000506 → FETCH?
0000507 ← +7.38070841E+00

Increment the sweep offset by 100us and measure -4.7dBm
0000508 → SWE:OFFS:TIME 0.0007
0000509 → TRIG:SOUR INT
0000510 → FETCH?
0000511 ← -4.64489528E-01

Increment the sweep offset by 100us and measure -9,66dBm
0000512 → SWE:OFFS:TIME 0.0008
0000513 → TRIG:SOUR INT
0000514 → FETCH?
0000515 ← -9.59349186E+00

Increment the sweep offset by 100us and measure -1.0dBm
0000516 → SWE:OFFS:TIME 0.0009
0000517 → TRIG:SOUR INT
0000518 → FETCH?
0000519 ← -1.00320165E+00
```

```
Increment the sweep offset by 100us and measure 7.2dBm
0000520 → SWE:OFFS:TIME 0.001
0000521 → TRIG:SOUR INT
0000522 → FETCH?
0000523 ← +7.21992165E+00
```

With this we've traced out one cycle of modulation. Now return to average measurements..

```
0000525 → DET:FUNC AVER
0000526 → CALC:FEED?
0000527 ← "POW:AVER"
0000528 → TRIG:SOUR?
0000529 ← IMM
0000530 → FETCH?
0000531 ← +1.10779875E+01
```

On Reset

Sweep time is 100us and sweep offset is 0.0us. Sweep time can go from 100us to 0.15s. And sweep offset has a range of +/-0.15seconds

Common Error Messages:

If the sensor is a LB59xxL then a -241,"Hardware missing" is generated when you use these commands.

If you fail to set the detector to NORMAL a -221,"Settings conflict" error message is generated.

Other Notes:

[SENSe]:TEMPerature?/qonly/

SENSe1:TEMPerature?/qonly/

Syntax:

Most common forms:

```
TEMP?
```

Long forms:

```
SENSE:TEMP?
SENSE1:TEMP?
```

Description:

Returns the temperature of the sensor in degrees Celsius.

Examples:

```
0000394 → TEMP?
0000395 ← +3.432300E+01
```

Trace Commands

Trace commands support the acquisition of time domain power measurements. These measurements are similar to oscilloscope traces except that:

1. The units of measurement are power (Watts or dBm) instead of volts
2. Instead of an x-y plot you get an ordered (by time) list of points in a binary format. Displaying the data is a user task.
3. The y-range of measurements covers two separate paths. For more information on the ranges refer to the SENSE:POWER:AC:RANGE commands. The data for these two paths are merged if both paths are active.
4. Finally, the merged data for these two paths can have different bandwidths.
    - The upper range has a bandwidth of about 40kHz
    - The lower range has a bandwidth of about 10kHz

[SENSe[1]]:TRACe[1]:STATe

[SENSe[1]]:TRACe[1]:UNIT

[SENSe[1]]:TRACe[1][:DATA]?/qonly/

[SENSe[1]]:TRACe:TIME

[SENSe[1]]:TRACe:OFFSet:TIME

Syntax:

Most common forms:

```
TRAC:STAT 0|1
TRAC:STAT?
TRAC:TIME?
TRAC:TIME <number>
TRAC:DATA? LRES|MRES|HRES
TRAC:OFFS:TIME?
TRAC:OFFS:TIME <number>
TRAC:UNIT DBM | W
```

(Some) Long forms:

```
SENSE:TRACE:STATE?
SENSE:TRACE:STATE 0|1
SENSE:TRACE:DATA? LRES|MRES|HRES
SENSE:TRACE:TIME?
SENSE:TRACE:TIME <number>
SENSE:TRACE:OFFSET:TIME?
SENSE:TRACE:OFFSET:TIME <number>
SENSE:TRACE:UNIT DBM | W
```

Description:

These functions are being covered together because they are used in together. Covering them separately would serve to transform the obvious into the arcane. Ultimately these commands require the detector to be set to NORMAL. When the detector is set to NORMAL, the sensor has a wider bandwidth (up to 40 kHz) so it is useful in the time domain.

When getting traces the setup is somewhat like setting up and oscilloscope. You have set the trigger source and sweep time and any additional triggering settings such as delay. In addition, you must enable the trace mode. And, when requesting the data you must specify the resolution (LRES, MRES, HRES).

Note that the trace data is always returned in a binary format. When using these commands in the LadyBug Interactive IO check “Binary Read” to transform the binary data into text. The code to transform the binary data into text is easily examined in the Interactive IO project.

The TRAC:DATA? command has a single pass parameter. That parameter specifies the resolution of the trace. The parameters available are:

Parameter Value	Meaning
LRES	This parameter value causes the command to return low resolution traces containing 230 points. This trace length is achieved through decimation.
MRES	This parameter value causes the command to return medium resolution traces containing 1000 points. This trace length is achieved through decimation.
HRES	This parameter value causes the command to return high resolution traces containing a number of points at the sample rate of 100kHz. So that the trace time determines the number of points

Trace time is limited to 100usec to 150msec. The default trace time is 500usec. The units for the NORMAL detector are the same as the units for the AVERAGE detector. But they are set and managed as a separate property.

Trace offset is limited to +/- 0.15s or 150ms. The default value is 0.0s

Don't forget to enable the trace by setting TRAC:STAT to 1 or ON. To exit the trace mode and return to default power measurements you must set TRAC:STAT to 0 or OFF and to set the detector to NORMAL.

An overview of the process for taking a trace is as follows:

- Select the NORMAL detector
- Setup the trigger source and any other triggering requirements
- Enable the trace state
- Set the trace time (and offset if desired)
- Take the trace

Subsequent traces (assuming everything is still setup) you:

- ***<u>Set the trace time!</u>***
- Take the trace

If you forget to set TRAC:TIME before executing TRAC:DATA? you'll retrieve the previous data. To acquire new data, trace time must be set before each TRAC:DATA? call. In this sense, executing TRAC:TIME acts as a the signal to the sensor to take new data.

Finally the trace data is returned in a binary format. This is standard IEEE 488 formatting with 32 bit floats being returned. In the LadyBug Interactive IO application the binary data can be displayed. The binary data can be displayed raw so that it looks like gibberish in ASCII. The first part of such a result is shown below. Notice on line 0000234 that it starts with #920. Also notice that an LRES trace was requested.

Since this is a LRES trace we expect 230, 4 byte floats or a total of 920 bytes. Examine the first 5 characters of the returned data. The hash mark ("#") indicates that the following data is in a binary format. The next character indicates the number of ASCII digits that follow. This character is a 3 so the next 3 characters indicate the size of the buffer that will follow. In this case the next three characters are 920. This corresponds nicely to what you might expect since 230 points of 4 byte each requires 920 bytes.

```
0000233 → TRAC:DATA? LRES
0000234 ← #3920??@?A??%A??BADCYAE?hArsAv2xA@?wAK]rAC?gA?1WA??@Ah?"A?l?@h??@?\????h?
```

If you requested an MRES trace (1000 points) the sensor would return 4000 bytes of data. And the number 4000 is represented by 4 characters in ASCII. So you would expect the binary data to begin with #44000. And as shown below, that is exactly what you get.

```
0000236 → TRAC:DATA? MRES
0000237 ← #44000?vA~?uAc3tA??rA??pA?ZnA??kA?hA??eA??bA??^A??ZA??VA??QA<MA??GA?......
```

If you use LadyBug interactive IO you can check “Binary Read” and the data will be transformed from binary to ASCII. The following shows the previous MRES trace decoded when “Binary Read” is checked.

```
0000238 → TRAC:DATA? MRES
0000239 ←
15.41213
15.34224
15.25727
15.15648
15.03098
14.89192
14.7383
…

…

…

…
15.53115
15.50525
15.48187
15.42457
```

Example:

This example shows taking a trace using the LadyBug Interactive IO. To do this in the LadyBug Interactive IO be sure to check “Binary Read”. As stated before, this feature is handy because it lets you ensure you’re getting back the trace data back you expect. The graphs were created in Excel. To return to making average power measurements you must set the TRAC:STAT to OFF and set the detector to AVERAGE. This is demonstrated at the end of the example.

```
0000240 → *RST
0000241 → DET:FUNC NORM
0000242 → TRIG:SOUR EXT
0000243 → TRAC:STAT ON
0000244 → TRAC:TIME 0.01
0000245 → TRAC:DATA? LRES
0000246 ←
2.22765
-2.190216
```

```
-7.256439
-11.12852
-9.961212
-4.240711
0.6899719
…
…
…
13.78869
12.47282
10.8202
8.648251
5.884064
2.342735
```

Now try to make a average power measurement…we at least have to select the average detector

```
0000247 → DET:FUNC AVER
0000248 → MEAS?
0000249 ← timed out
```

Didn't work because the trace state is still enabled…

```
0000250 → SYST:ERR?
0000251 ← -221,"Settings conflict;Trace mode active"
```

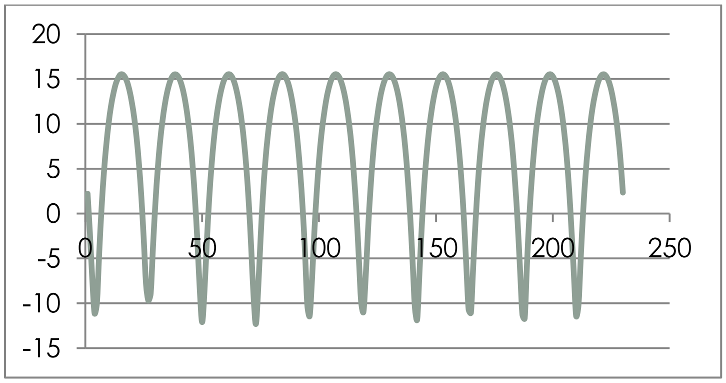
Ok, we'll turn the trace state to off

```
0000252 → TRAC:STAT OFF
```

Now it works

```
0000253 → MEAS?
0000254 ← +1.10655815E+01
```

The graph below shows this data on a simple scatter diagram (x-y plot). The signal was a 1kHz amplitude modulated waveform. The time span is 10ms and there are 230 points.



<u>On Reset</u>

TRACE:STATE is disabled

TRACE:TIME is set to 500ms

TRACE:OFFSET:TIME is set to 0.0

TRACE:UNIT is set to dB

Common Error Messages:

If you try to make an average power measurement with the trace state still on you'll get a -221,"Settings Conflict" message.

If the sensor is an L model (e.g. LB5918L, LB5926L…etc) and you try to make trace measurements you'll get a -241,"Hardware missing" error.

If the AVERAGE detector is selected and you attempt some of these commands you'll get a -221,"Settings conflict"

Service

Service is a collection of commands that don't have a direct bearing on measurements. Never the less, these functions are widely used. These commands are often used in systems where some relationship between the device under test and the equipment being used must be recorded.

These commands include setting or getting the last calibration date, the sensors serial number, firmware version etc. It also supports a number of functions related to the capabilities of the sensor including maximum power and frequency.

SERVice:BIST:TRIGger:LEVel:STATe?/qonly/

Syntax:

Most common forms:

```
SERV:BIST:TRIG:LEV:STAT?
```

Long forms:

```
SERVICE:BIST:TRIGGER:LEVEL:STATE?
```

Description:

Returns a 0 when external trigger in is low or 1 external trigger in is high. External trigger is an SMB connector labeled TI on the back of the sensor.

Examples:

In this example a 1Hz, 5V square wave has been connected to trigger in port. This causes the return value to switch between 0 and 1 depending on when the command was sent relative to the square wave.

```
0000090 → *RST
0000091 → SERV:BIST:TRIG:LEV:STAT?
0000092 ← 1
0000093 → SERV:BIST:TRIG:LEV:STAT?
0000094 ← 0
0000095 → SERV:BIST:TRIG:LEV:STAT?
0000096 ← 1
0000097 → SERV:BIST:TRIG:LEV:STAT?
0000098 ← 0
0000099 → SERV:BIST:TRIG:LEV:STAT?
0000100 ← 1
0000101 → SERV:BIST:TRIG:LEV:STAT?
0000102 ← 1
0000103 → SERV:BIST:TRIG:LEV:STAT?
0000104 ← 0
0000105 → SERV:BIST:TRIG:LEV:STAT?
0000106 ← 1
0000107 → SERV:BIST:TRIG:LEV:STAT?
0000108 ← 0
0000109 → SERV:BIST:TRIG:LEV:STAT?
0000110 ← 1
0000111 → SERV:BIST:TRIG:LEV:STAT?
0000112 ← 0
```

SERVice:OPTion/qonly/

Syntax:

Most common forms:

```
SERV:OPT?
```

Long forms:

```
SERVICE:OPTION?
```

Description:

This returns a list of options that are installed and enabled on the sensor in question.

Examples:

In this example the sensor has options 001 and 003 installed and enabled. Also, the return value indicates the connector which in this case is 3.5mm male.

```
0000113 → SERV:OPT?
0000114 ← "001,003,35M"
```

SERVice:SECure:ERASe/nquery/

Syntax:

Most common forms:

```
SERV:SEC:ERAS
SERV:SEC:ERAS FAST
```

Long forms:

```
SERVICE:SECURE:ERASE
SERVICE:SECURE:ERASE FAST
```

Description:

“Secure erase” allows the user to clear all relevant non-volatile memory. It includes save/recall registers, frequency dependent offset tables, user calibration correction, state information and a number of other items. The user may clear the data with or without the FAST parameter. In short, any parameter or value the user can set, directly or indirectly, is cleared in the following manner:

28. “FAST” mode
    - All bytes are set to 0x00
29. Normal mode
    - All bytes are set to 0xFF
    - All bytes are set to a random number between 0x00 and 0xFF inclusive
    - All bytes are set to 0x00

Other Notes:

There are a range of motivations for employing this command. One motivation is to place the sensor in a known, factory-like original state. Any single pass clearing of the data would meet the need. So, the command SERV:SEC:ERAS FAST would be sufficient. The FAST parameter serves to speed up this process considerably.

A second motivation might be to clear the sensors of sensitive data for securing reasons. In this case, clearly the one pass FAST erase is insufficient. To satisfy this need, simply execute the SERV:SEC:ERAS several times with no parameters. For example, if you are required to obliterate the data in 32 passes, simply repeat the command 32 times. If you need 64 passes then repeat the command 64 times. Each pass will take between 7-15 seconds.

One final note, the random numbers used are generated using an “analog entropy source” or, analog noise sources in the microprocessor.

SERVice:SENSor:CDATe?/qonly/

SERVice:SENSor1:CDATe?/qonly/

Syntax:

Most common forms:

```
SERV:SENS:CDAT?
```

Long forms:

```
SERIVCE:SENSOR:CDATE?
SERIVCE:SENSOR1:CDATE?
```

Description:

Returns the date of calibration in the form of Year, Month, Day

Examples:

In this example the date of calibration is August 8, 2016

```
0000117 → SERV:SENS:CDAT?
0000118 ← 2018,8,6
```

SERVice:SENSor:CDUEdate

SERVice:SENSor1:CDUEdate

Syntax:

Most common forms:

```
SERV:SENS:CDUE?
SERV:SENS:CDUE <YEAR>,<Month>,<DAY>
```

Long forms:

```
SERVICE:SENSE:CDUEDATE?
SERVICE:SENSE1:CDUEDATE?
SERVICE:SENSE:CDUEDATE  <YEAR>,<Month>,<DAY>
SERVICE:SENSE1:CDUEDATE <YEAR>,<Month>,<DAY>
```

Description:

This command either sets or returns the current calibration due date as stored in non-volatile memory. The year, month, day must be enclosed in quotes (“) as shown in the example. Note, the parameters are not range checked.

Examples:

In this example the date is queried, then set, queried again the cleared and queried once again.

```
0000161 → *RST
0000162 → SERV:SENS:CDUE?
0000163 ← NONE
0000164 → SERV:SENS:CDUE "2020,6,15"
0000165 → SERV:SENS:CDUE?
0000166 ← 2020,6,15
0000167 → SERV:SENS:CDUE ""
0000168 → SERV:SENS:CDUE?
0000169 ← NONE
```

Common Error Messages:

If the surrounding quotes are omitted then error -148, “Character data not allowed” is issued.

SERVice:SENSor:CPLace

SERVice:SENSor1:CPLace

Syntax:

Most common forms:

```
SERV:SENS:CPL?
SERV:SENS:CPL <string>
```

Long forms:

```
SERVICE:SENSE:CPLACE?
SERVICE:SENSE:CPLACE <string>
```

Description:

This returns the place of calibration. The calibration place must in in quotes as shown in the example. To clear the place of calibration supply the command is a quoted null string as shown in the example.

Examples:

In this example the calibration place is queried, then set to Boise, ID, queried again, cleared and queried once more. Note that the string must be in quotes as shown below.

```
0000178 → SERV:SENS:CPL?
0000179 ← NONE
0000180 → SERV:SENS:CPL "Boise, ID"
0000181 → SERV:SENS:CPL?
0000182 ← Boise, ID
0000183 → SERV:SENS:CPL ""
0000184 → SERV:SENS:CPL?
0000185 ← NONE
```

Common Error Messages:

If the surrounding quotes are omitted then error -148, “Character data not allowed” is issued.

SERVice:SENSor:FREQuency:MAXimum?/qonly/

SERVice:SENSor1:FREQuency:MAXimum?/qonly/

SERVice:SENSor:FREQuency:MINimum?/qonly/

SERVice:SENSor1:FREQuency:MINimum?/qonly/

Syntax:

Most common forms:

```
SERV:SENS:FREQ:MAX?
SERV:SENS:FREQ:MIN?
```

Long forms:

```
SERVICE:SENSOR:FREQUENCY:MAXIMUM?
SERVICE:SENSOR1:FREQUENCY:MAXIMUM?
SERVICE:SENSOR:FREQUENCY:MINUMUM?
SERVICE:SENSOR1:FREQUENCY: MINUMUM?
```

Description:

As you might expect, these commands return the maximum and minimum operating frequency of the sensor.

Examples:

In the example below, we are querying an LB5926L

```
I 0000189 → SERV:SENS:FREQ:MAX?
0000190 ← +2.65000000E+10
0000191 → SERV:SENS:FREQ:MIN?
0000192 ← +9.00000000E+03
```

SERVice:SENSor:POWer:AVERage:MAXimum?/qonly/

SERVice:SENSor1:POWer:AVERage:MAXimum?/qonly/

Syntax:

Most common forms:

```
SERV:SENS:POW:AVER:MAX?
```

Long forms:

```
SERVICE:SENSOR:POWER:AVERAGE:MAXIMUM?
```

Description:

This command returns the maximum calibrated power for the sensor.

Examples:

```
0000193 → SERV:SENS:POW:AVER:MAX?
0000194 ← +2.600000E+01
```

SERVice:SENSor:POWer:PEAK:MAXimum?/qonly/

SERVice:SENSor1:POWer:PEAK:MAXimum?/qonly/

Syntax:

Most common forms:

```
SERV:SENS:POW:PEAK:MAX?
```

Long forms:

```
SERVICE:SENSOR:POWER:PEAK:MAXIMUM?
SERVICE:SENSOR1:POWER:PEAK:MAXIMUM?
```

Description:

This command returns the maximum peak power. The peak power specification is both power and time limited. So, measuring peak power requires that you comply with both the peak power limitation and the time/duty cycle limits of this specification.

Examples:

```
0000201 → SERV:SENS:POW:PEAK:MAX?
0000202 ← +3.300000E+01
```

SERVice:SENSor:POWer:USABle:MAXimum?/qonly/

SERVice:SENSor1:POWer:USABle:MAXimum?/qonly/

SERVice:SENSor:POWer:USABle:MINimum?/qonly/

SERVice:SENSor1:POWer:USABle:MINimum?/qonly/

Syntax:

Most common forms:

```
SERV:SENS:POWER:USAB:MAX?
SERV:SENS:POWER:USAB:MIN?
```

Long forms:

```
SERVICE:SENSOR:POWER:USABLE:MAXIMUM?
SERVICE:SENSOR:POWER:USABLE:MINIMUM?
SERVICE:SENSOR1:POWER:USABLE:MAXIMUM?
SERVICE:SENSOR1:POWER:USABLE:MINIMUM?
```

Description:

This returns the maximum and minimum usable specified power.

Examples:

```
0000203 → SERV:SENS:POW:USAB:MAX?
0000204 ← +2.600000E+01
0000205 → SERV:SENS:POW:USAB:MIN?
0000206 ← -6.000000E+01
```

SERVice:SENSor:RADC?/qonly/

SERVice:SENSor1:RADC?/qonly/

Syntax:

Most common forms:

```
SERV:SENS:RADC?
```

Long forms:

```
SERVICE:SENSOR:RADC?
SERVICE:SENSOR1:RADC?
```

Description:

This returns the ADC values of the two paths. The first number is the value of the least sensitive path. The second number is the most sensitive path.

Examples:

```
0000207 → SERV:SENS:RADC?
0000208 ← 39214,59706
```

SERVice:SENSor:SNUMber?/qonly/

SERVice:SENSor1:SNUMber?/qonly/

Syntax:

Most common forms:

```
SERV:SENS:SNUM?
```

Long forms:

```
SERVICE:SENSOR:SNUMBER?
SERVICE:SENSOR1:SNUMBER?
```

Description:

This returns the factory serial number of the sensor.

Examples:

In this case the returned serial number is 177464. That you match the serial number on the rear bulkhead of the instrument.

```
0000209 → SERV:SENS:SNUM?
0000210 ← 177464
```

SERVice:SENSor:TNUMber

SERVice:SENSor1:TNUMber

Syntax:

Most common forms:

```
SERV:SENS:TNUM?
SERV:SENS:TNUM <string>
```

Long forms:

```
SERVICE:SENSOR:TNUMBER?
SERVICE:SENSOR:TNUMBER <string>
SERVICE:SENSOR1:TNUMBER?
SERVICE:SENSOR1:TNUMBER <string>
```

Description:

This command allows the user to set and recall the tracking number for their own purposes.

Examples:

In the example below the tracking number is first queried. However, it is not set so "NONE" is returned. Then we attempt to improperly set the tracking number. Notice error message -148. We omitted the quotes. Then we reset the command with the quotes added. Finally we checked it and then set it back to a null string.

```
0000211 → SERV:SENS:TNUM?
0000212 ← NONE
0000213 → SERV:SENS:TNUM 123456789
0000214 → SYST:ERR?
0000215 ← -148,"Character data not allowed"
0000216 → SERV:SENS:TNUM "123456789"
0000217 → SERV:SENS:TNUM?
0000218 ← 123456789
0000219 → SERV:SENS:TNUM ""
0000220 → SERV:SENS:TNUM?
0000221 ← NONE
```

Common Error Messages:

As shown in the example, the most common error that might occur is because the quote marks are omitted. The forces error -148,"Character data not allowed"

SERVice:SENSor:TYPE?/qonly/

SERVice:SENSor1:TYPE?/qonly/

Syntax:

Most common forms:

```
SERV:SENS:TYPE?
```

Long forms:

```
SERVICE:SENSOR:TYPE?
SERVICE:SENSOR1:TYPE?
```

Description:

This queries the sensors model number.

Examples:

```
0000222 → SERV:SENS:TYPE?
0000223 ← LB5926L
```

SERVice:VERSion:PROCessor?/qonly/

Syntax:

Most common forms:

```
SERV:VERS:PROC?
```

Long forms:

```
SERVICE:VERSION:PROCESSOR?
```

Description:

This returns information about the sensor and its construction. If you call for support you are likely to be asked for this return string.

Examples:

```
0000224 → SERV:VERS:PROC?
0000225 ←
CPU=20016419,RF=0008,USB=0027,NAND=0FFF,DC=0359,UPT=0273824446,INT=2108311039,2E=0000000000,2
0E=0000000000
```

SERVice:VERSion:SYSTem:DFU/nquery/

Syntax:

Most common forms:

```
SERV:VERS:SYST:DFU
```

Long forms:

```
SERVICE:VERS:SYSTEM:DFU
```

Description:

This places the sensor into a state that allows for firmware upgrades. When this command is issued the sensor will be seen as a new type of device and the LED on the rear panel will start to blink alternating between green and red.

You will lose the ability to communicate with it except through the upgrade application. To restore the sensor to normal operation without upgrading, simply unplug and then plug in the USB cable on the sensor. This will restore the sensor to its normal state.

Examples:

```
0000226 → SERV:VERS:SYST:DFU
```

SERVice:VERSion:SYSTem?/qonly/

Syntax:

Most common forms:

```
SERV:VERS:SYST?
```

Long forms:

```
SERVICE:VERSION:SYSTEM?
```

Description:

This returns the firmware version of the sensor.

Examples:

```
0000229 → SERV:VERS:SYST?
0000230 ← 0.99.242_20190611_1132_s_ma
```

Status

The status commands can be used to monitor the state of the sensor. However, the explanation of Status information may, for many, be overly complex. This is especially true when compared to actual use. Typically, simply read the status byte. Occasionally, systems use the lower limit and upper limit fail registers.

In any case, the arrangement or structure of the status registers is as follows:

Base status registers ➔Intermediate status registers ➔Status Byte

This means that each base register feeds an intermediate register. Each intermediate register feeds a bit in the status byte.

In the LB59xx sensors, each ***BASE*** register is 16 bits long but only one bit is used. The used bit used in the ***BASE*** registers is ***always*** bit 1. Sometimes it is said that the base register bits are ORed together before being used to set a bit in an intermediate registers. Again, there is only one active bit in base registers. Stating the base register bits are ORed together is a *bit* pretentious (pun intended).

Some Intermediate registers receive the ORed values from base registers. Other intermediate registers do not receive ORed values from base registers. Instead they have their own bits. In any case, the intermediate register bits are ORed together and the result is used to set specific bits in the Status Byte. In short, each intermediate register can set or clear 1 Status Byte bit.

The table below summarizes this arrangement.

Base Registers		**Intermediate Registers**		**Status Byte**
Name	Bit	Bit	Name	Bit
(no base registers)			(unused)	0
		3	Device Status	1
			Error Event Queue	2
Questionable POWer Summary	1	3	Questionable Status	3
Questionable CALibration Summary	1	8		
(no base registers)			Output Queue	4
		0..7	Standard Event	5
		1..5, 7	Status Byte	6
Operation CALibrating Summary	1	0	Operation Status	7
Operation MEASuing Summary	1	4		
Operation TRIGger Summary	1	5		
Operation SENSe Summary	1	10		
Operation Lower Limit Fail Summary	1	11		
Operation Upper Limit Fail Summary	1	12		

Table 1 Status Registers

The table is read left to right. But an example makes it easy to interpret.

Example #1: Base register **Questionable POWer Summary**, bit 1, and base register **Questionable CALibration Summary**, bit 1, feed intermediate register **Questionable Status**, bits 3 and 8 respectively. The ORed value of intermediate register **Questionable Status** bits 3 and 8, are in turn used to set bit 3 of the **Status Byte**. That's it.

Example #2: Intermediate register, **Device Status** is not fed by any base register. However, the ORed value of intermediate register **Device Status,** bit 3, sets **Status Byte**, bit 1.

Example #3: The ORed value of the intermediate register **Standard Event** bits 0...7 are used to set **Status Byte** bit 5.

Up to this point we've treated the registers as simple devices. But in fact the status registers are a bit more complex.

Note: Individual users will determine the value of this information. Many (if not most) programmers eschew this capability and simply monitor the status byte. This is the status byte is often sufficient for their purposes. One exception might be the LLF and ULF summary registers.

In reality, each status register is composed of multiple layers. Each layer feeds the next. These layers (in order) are as follows:

- The current value or condition register – This register is updated real time. Conceptually, this is what has been discussed in this section up to this point. The current value is passed to its associated transition filter.
- Transition filter – This layer controls which changes in the condition register are passed to its associated Event register.
  - Positive transition – if a particular bit in the condition register changes from 0 to 1 this is considered a positive transition
  - Negative transition – if a particular bit in the condition register changes from 1 to 0 this is considered a negative transition
- Event register - This layer latches any Transition filter changes clear to set. Once a bit is set it remains set until it is cleared by a *CLS command. The contents of the event register are then passed through the associated enable mask.
- Enable – This layer is simply a binary mask. The Enable mask passes bits from the Event register to the associated Summary.
- Summary – This register ORs of all of the bits sent to it by the enable register. This ORed value (a 0 or 1) is the output or value of the Summary. This value is in turn passed to a single status register bit. So, if any bit in the Event register is set: and the Enable mask has enabled these bits: the Summary will set its associated bit in the Status Byte.

The purpose of all this is to "capture" changes in sensor status and report information asynchronously. To the extent that a programmer is required to monitor, analyze and report this status information, these functions may be useful. To satisfy such a requirement without filters, event registers and so on, would require the programmer constant polling. This logic allows the user set up filters, and masks. And then monitor the Status Byte as time permits.

Once the status byte reports a value of interest the programmer can interrogate the remaining registers to determine the nature of the fault. And the programmer can be confident in the fact that all such events have been captured. *This asynchronous process is possible because all enabled Events are latched until cleared by a *CLS.*

The following status commands are covered as a group because of their association with this explanation and their similarity in form.

Base Register	Command Prefix
Questionable POWer Summary	`STATus:QUEStionable:POWer[:SUMMary]:`
Questionable CALibration Summary	`STATus:QUEStionable:CALibration[:SUMMary]:`
Operation CALibrating Summary	`STATus:OPERation:CALibrating[:SUMMary]:`
Operation MEASuing Summary	`STATus:OPERation:MEASuring[:SUMMary]:`
Operation TRIGger Summary	`STATus:OPERation:TRIGger[:SUMMary]:`
Operation SENSe Summary	`STATus:OPERation:SENSe[:SUMMary]:`
Operation Lower Limit Fail Summary	`STATus:OPERation:LLFail[:SUMMary]:`
Operation Upper Limit Fail Summary	`STATus:OPERation:ULFail[:SUMMary]:`

To complete a command append one of the following suffixes:

- **`CONDition?/qonly/ - queries the current state of the register`**
- **`ENABle – enables or disables selected bits`**
- **`NTRasition – sets up or queries the negative transition mask`**
- **`PTRansition – sets up or queries the positive transition mask`**
- **`EVENt?/qonly/- queries the event status`**

So to query the POWer summary registers current state (condition) you would append the associated command prefix with the condition query suffix:

```
STATus:QUEStionable:POWer[:SUMMary]: + Condition?/qonly/
```

…yielding

```
STAT:QUES:POW:SUMM:COND?
```

…or

```
STAT:QUES:POW:COND? (SUMM is optional)
```

From the intereactive IO application it would look like this:

```
0000001 → STAT:QUES:POW:COND?
0000002 ← +0
```

To determine if any events have occurred in the Operation Lower Limit Fail Summary you would send the following query (from InteractiveIO):

```
0000005 → STAT:OPER:LLF:EVEN?
0000006 ← +0
```

And to examine the negative transition (NTR) mask then examine and set the positive transition (PTR) mask of the same register:

```
0000011 → STAT:OPER:LLF:NTR?
0000012 ← +0
0000013 → STAT:OPER:LLF:PTR?
0000014 ← +32767
0000015 → STAT:OPER:LLF:PTR 0
```

```
0000016 → STAT:OPER:LLF:PTR?
0000017 ← +0
0000018 → STAT:OPER:LLF:PTR +32767
0000019 → STAT:OPER:LLF:PTR?
0000020 ← +32767
```

The intermediate and status registers are much the same in that they use a prefix and the same suffixes.

Intermediate and Status Registers	Command Prefix
Device Status	**STATus:DEVice:**
Questionable Status	**STATus:QUEStionable:**
Operation Status	**STATus:OPERation:**

And, just as with the base status registers, completing the command is accomplished by appending one of the following suffixes:

- **CONDition?/qonly/**
- **ENABle**
- **NTRasition**
- **PTRansition**
- **EVENt?/qonly/**

By way of example (again from the interactive IO application):

```
0000035 → STAT:DEV:COND?
0000036 ← +0
0000037 → STAT:DEV:NTR?
0000038 ← +0
0000039 → STAT:DEV:PTR?
0000040 ← +8
```

STATus:PRESet/nquery/

Syntax:

Most common forms:

```
STAT:PRES
```

Long forms:

```
STATUS:PRESET
```

Description:

This command sets certain register filters to their preset or power on values. This command should not be confused with *RST. For all status registers, the PTR (positive transition) filters are always set to all 1s. And all NTR (negative transition) filters are set to 0. Note that, regardless of the following information, bit 15 is always set to 0.

Finally, all of the ENABle masks for all status registers are set to all 1s except for the ENABle masks for the the OPERational and QUEStionalbe. In these cases they ENABle masks are set to 0. The default or preset state of these masks are why most programmers find the STATUS BYTE sufficient to determine the status of the sensor.

System

The system commands provide commands to query, control or configure the sensor in a general sense.

SYSTem:BLINk/nquery/

SYSTem:BLINk1/nquery/

Syntax:

Most common forms:
```
BLINK
```

Long forms:
```
SYSTEM:BLINK
SYSTEM:BLINK1
```

Description:

Issuing this command causes the LED on the rear bulkhead to blink bright green once. The bright green color can be hard to see of the LED is bright red.

Examples:

```
0000254 → BLINK
```

Other Notes:

The most common uses for this command, is to show that you're communicating with the sensor. If there are multiple sensors it can be useful in differentiating between the sensors.

SYSTem:COMMunicate:SPI:CLOCk

Description:

This command has to do with specifying the SPI clock signal. This command is specific to the SPI/I2C option. Please reference the option specific documents for additional information.

SYSTem:COMMunicate:USB:ADDRess

Syntax:

Most common forms:

```
SYST:COMM:USB:ADDR <0…127>
```

Long forms:

```
SYSTEM:COMMUNICATION:USB:ADDRESS <0..127>
```

Description:

Allows the user to set/get a value between 0…127 inclusive. It is not otherwise used by the sensor.

Examples:

This demonstrates setting and getting the variable. The highlighted area demonstrates the error condition.

```
0000309 → SYST:COMM:USB:ADDR?
0000310 ← +0
0000311 → SYST:COMM:USB:ADDR 10
0000312 → SYST:COMM:USB:ADDR?
0000313 ← +10
0000314 → SYST:COMM:USB:ADDR 127
0000315 → SYST:COMM:USB:ADDR?
0000316 ← +127
0000317 → SYST:COMM:USB:ADDR 128
0000318 → SYST:ERR?
0000319 ← -222,"Data out of range"
0000320 → SYST:COMM:USB:ADDR 0
0000321 → SYST:COMM:USB:ADDR?
0000322 ← +0
```

On Reset

This value is unaffected by *RST

Common Error Messages:

The most common error message will be the error message generate by going out of range as shown above.

SYSTem:DATE

Syntax:

Most common forms:

```
SYST:DAT?
SYST:DAT <string>
```

Long forms:

SYSTEM:DATE?

SYSTEM:DATE <string>

Description:

This command allows the user to set or get the date in the sensor. Many of the early sensors have the calendar/clock even if UOP was not enabled. In newer sensors the calendar/clock is only available with UOP. The calendar/clock is useful in sensors that have the UOP (unattended operation) option. In this event all measurements are time stamped when then are stored.

Examples:

Early sensor or newer sensor with the UOP option:

```
0000283 → SYST:DATE?
0000284 ← 2000,1,1
0000285 → SYST:DATE 2015,7,31
0000283 → SYST:DATE?
0000284 ← 2015,7,31
```

This is a newer sensor without the UOP option. Note the LED changed from green to red on the highlighted line:

```
0000283 → SYST:DATE?
0000284 ← 2000,1,1
0000285 → SYST:DATE 2015, 7,31
0000286 → SYST:ERR?
0000287 ← -241,"Hardware missing;UOP option"
```

On Reset

The calendar is unaffected by a *RST but it is cleared (to 2000,1,1) with a SERV:SEC:ERAS.

SYSTem:ERRor?/qonly/

Syntax:

Most common forms:

```
ERR?
SYST:ERR?
```

Long forms:

```
SYSTEM:ERROR?
```

Description:

This command returns the next error in the error queue. If there are no errors to return then a message indicating no error is returned. Anytime there are errors in the queue, the LED on the rear bulkhead is bright red.

Examples:

This sequence starts with an empty error queue and a series of queries showing what an empty error queue looks like. Finally, an error is induced. In this case READ? times out because INIT:CONT is enabled. Then we query the instrument and see an error message is present.

```
0000012 → SYST:ERR?
0000013 ← +0,"No error"
0000014 → SYSTEM:ERROR?
0000015 ← +0,"No error"
0000016 → ERR?
0000017 ← +0,"No error"
0000018 → INIT:CONT?
0000019 ← 1
0000020 → READ?
0000021 ← timed out
0000022 → SYST:ERR?
0000023 ← -213,"Init ignored"
```

On Reset

The error queue is NOT cleaned out after a *RST or SYST:PRES.

Other Notes:

Issue a *CLS command to clear the error queue.

SYSTem:HELP:HEADers?/qonly/

Syntax:

Most common forms:

```
SYST:HELP:HEAD?
```

Long forms:

```
SYSTEM:HELP:HEADERS?
```

Description:

This command returns a list of the SCPI commands. The commands are the same strings that have been used to create the headers in this document. The string returned is an arbitrary data block (as defined in IEEE 488.2). The format of the arbitrary data block is as follows:

30. In general the form is #Xyyy..dddddd<LF>
31. The block starts with a "#"
32. The "#" is followed by a ***single*** ASCII character. This will be referred to as ***X***. This ASCII character must be one of the following ASCII characters 1,2,3,4,5,6,7,8,9. In particular, it cannot be a letter, nor can it be a special character, nor can it be zero ("0").
33. ***#X*** is followed by a string of ASCII characters. The individual characters in this string are represented with a ***y***. Each of the ***y*** characters must be one of the following ASCII characters: 0,1,2,3,4,5,6,7,8,9. The number of ***y*** characters is ***X*** in length. So that if X=3 then there will be three ***y***s. If X=7, then there will be seven ***y***s as shown in this string:
    - `#72345689`
    - `#` = start of return string
    - `7` = ***X***
    - `2345689` = ***yyyyyyy*** (note that there are seven ***y***s)
34. The sequence of ***y***s are grouped together and interpreted as a single number. We'll refer to the numeric value represented by the ASCII collection of ***y***s as ***Y***. By way of example, the following uses this string:
    - `#32154892079...`

    In this case ***X***=3 (highlighted in green), ***yyy*** (highlighted in magenta) is 215. The ASCII value represented by ***yyy*** is 215. So that ***Y*** is 215. Y is the length of the data block that follows ***yyy***. So that in this case the returned data begins with "4892079" and continues on for a total of 214 characters followed by a <LF> or line feed. This makes the total character count it 215 which is what we expect. Note that #3215 is not included in the count of 215 but the trailing <LF> is counted in the 215.

    Assume a command returned this string:

```
#228This is just a silly string<LF>
#220Another demo string<LF>
#16Hello<LF>
```

In the first case ***X***=2, ***yy*** = 28. This means ***Y*** = 28. So ***Y*** tells us that the data block (highlighted in grey) is 28 characters long. Note that <LF> is a common representation for a single line feed character whose decimal value is 10. In the second case ***X***=2, ***yy*** = 20. This means ***Y*** = 20. So ***Y*** tells us that the data block (highlighted in grey) is 20 characters long. In the third case ***X***=1, ***y*** = 6. This means ***Y*** = 6. So ***Y*** tells us that the data block (highlighted in grey), in this case "Hello<LF>" is 6 characters long.

35. In our examples we have assumed the data block is composed of ASCII characters. But the data in the data block could just as easily be binary data (as in traces).

<u>Examples:</u>

In this example we demonstrate SYST:HELP:HEAD?

```
0000010 → SYST:HELP:HEAD?
0000011 ← #511121
:ABORt/nquery/
:ABORt1/nquery/
:CALCulate:FEED
:CALCulate:FEED1
:CALCu…..
…
…
…
…
…
…
*IDN?/qonly/
*OPC
*OPT?/qonly/
*RCL/nquery/
*RST/nquery/
*SAV/nquery/
*SRE
*STB?/qonly/
*TRG/nquery/
*TST?/qonly/
*WAI/nquery/
```

You can see that ***X***=5, ***yyyyy*** = 11121 so that ***Y*** is 11121 bytes long. If you copy and paste this string into Notepad++ (or other Word processor) you might see that the length of the reported string is longer that ***Y*** indicates. This is a result of several things:

36. Most word processors will try to format the text and in so doing will add a <CR> and or a <LF> at the end of each line
37. With some processors additional "hidden" characters are added
38. You've copied the sent command or other extraneous text
39. You're counting the "`#511121`" portion of the returned string

SYSTem:PERSona:MANufacturer:DEFault

SYSTem:PERSona:MANufacturer/nquery/

SYSTem:PERSona:MODel:DEFault

SYSTem:PERSona:MODel/nquery/

DIAG:BOOT:COLD/nquery/

Syntax:

Most common forms:

```
SYST:PERS:MAN:DEF
SYST:PERS:MAN:DEF?
SYST:PERS:MAN <string>
SYST:PERS:MOD:DEF
SYST:PERS:MOD:DEF?
SYST:PERS:MOD <string>,<string>,<string>
DIAG:BOOT:COLD
```

Long forms:

```
SYSTEM:PERSONA:MANUFACTURER:DEFAULT
SYSTEM:PERSONA:MANUFACTURER:DEFAULT?
SYSTEM:PERSONA:MANUFACTURER <string>
SYSTEM:PERSONA:MODEL:DEFAULT
SYSTEM:PERSONA:MODEL:DEFAULT?
SYSTEM:PERSONA:MODEL <string>,<string>,<string>
DIAG:BOOT:COLD
```

Description:

A summary of the commands are as follows:

SYST:PERS:MAN:DEF causes the sensor to revert to the original manufacturer (LadyBug)
SYST:PERS:MOD:DEF causes the sensor to revert to the original model number, serial number and firmware version.
SYST:PERS:MAN:DEF? returns the default (original) manufacturer even when the persona is enabled.
SYST:PERS:MOD:DEF? returns the default (original) model, serial and firmware version information even when persona is enabled.
SYST:PERS:MAN <string> sets the persona manufacturer information
SYST:PERS:MOD <string>,<string>,<string> sets the person model, serial number and firmware version information
DIAG:BOOT:COLD causes the sensor to reboot

These commands are being covered together because they are used as a set. Also, they are unique to the LB59xx sensors. This set of commands may prove useful in a number of situations where it is important to emulate a particular sensor, modify a sensors ID for software testing more thoroughly (i.e. does an LB59xx really look like a U2000 with my driver?) or to create a new identity for an existing sensor.

NOTE: When persona is active, only the USBTMC interface is available. So the LadyBug apps will not function. Ensure you have either Agilent/Keysight IO libraries installed or that you have NI Max installed. Otherwise you will not be able to communicate with the instrument.

These commands allow the end user to change the reported manufacturer, model number, serial number and firmware version. LadyBug has a Persona application (with source code) that allows a user to setup a sensor persona. Bear in mind that *RST, SYST:PRES and CLS do not clear the persona settings. In order to permanently clear the and active persona, you must use SYST:PERS:MAN DEF and SYST:PERS:MOD DEF. It's also true that SERV:SEC:ERASE will remove an active persona. However, if persona is active, these commands must be issued from USBTMC.

Again, if persona is active you must use USBTMC to communicate with the sensor. The power meter application and the interactive IO applications no longer work because they use USB-HID. So before you try to use persona, ensure you have either Agilent/Keysight IO libraries or NI Max installed. Otherwise you'll have a sensor you can't communicate with.

That said, setting up and using a persona is very simple.

1. Get the required information:
    - The manufacturers name…exactly
    - The model number…exactly
    - Serial number…exactly
    - Firmware version…exactly
2. Set the manufacturer's information (LadyBug interactive IO, Agilent IO libraries, NI Max)
3. Set the model information (LadyBug interactive IO, Agilent IO libraries, NI Max)
4. Reboot (DIAG:BOOT:COLD will do the same thing)
5. Now the sensor is a USBTMC only sensor
6. You now communicate via USBTMC only (Agilent IO libraries, NI Max)
7. To revert to normal mode (disable persona) issue the following commands (Agilent IO libraries, NI Max):
    a. SYST:PERS:MAN:DEF
    b. SYST:PERS:MOD:DEF
    c. DIAG:BOOT:COLD

Examples:

Acquire the data needed. I'm going to use an Agilent U2000. So that from Agilent/Keysight Interactive IO:

```
-> *IDN?
<- Agilent Technologies,U2000A,MY50000235,A1.04.00
```

I then disconnected my U2000A from the system. Then I removed it from the IO libraries list. Please don't forget to do this.

The highlighted portions are going to be copied and pasted into the persona commands. Now connect to LB59xx sensor and open the LadyBug interactive IO. Enter the following commands:

```
0000045 → *IDN?
0000046 ← LadyBug Technologies LLC,LB5926L,177464,0.99.242
0000047 → SYST:PERS:MAN "Agilent Technologies"
0000048 → SYST:PERS:MOD "U2000A","MY50000235","A1.04.00"
0000049 → DIAG:BOOT:COLD
```

At this point the LadyBug sensor changes its persona. It no longer can be seen in LadyBug Interactive IO. So we move back to the Agilent/Keysight Interactive IO.

To test the persona, execute the *IDN? command. You should get:

```
-> *IDN?
<- Agilent Technologies,U2000A,MY50000235,A1.04.00
```

Now it should appear to be a U2000A. If you just wanted to force the LadyBug sensor to be a USBTMC only sensor, just make a small change and set the persona. From that point on it will be a USTMC only sensor.

To revert back to a LadyBug sensor, continue as follows:

```
-> SYST:PERS:MAN:DEF
-> SYST:PERS:MOD:DEF
-> DIAG:BOOT:COLD
```

SYSTem:PRESet/nquery/

<u>Syntax:</u>

<u>Most common forms:</u>

```
SYST:PRES
```

<u>Long forms:</u>

```
SYSTEM:PRESET
```

<u>Description:</u>

System preset is equivalent to the IEEE 488.2 command *RST. The same reset values and behavior apply.

<u>Examples:</u>

```
0000005 → SYST:PRES
```

SYSTem:TIME

Syntax:

Most common forms:

```
SYST:TIME?
SYST:TIME <hh,mm,ss>
```

Long forms:

```
SYSTEM:TIME?
SYSTEM:TIME <hh,mm,ss>
```

Description:

This command allows the user to set or get the time in the sensor clock. Many of the early sensors have the calendar/clock even if UOP was not enabled. In newer sensors the calendar/clock is only available with UOP. The calendar/clock is useful in sensors that have the UOP (unattended operation) option. If UOP is present and in use, all measurements are time and date stamped when then are stored.

Examples:

```
0000003 → SYST:TIME?
0000004 ← 0,0,0
```

Common Error Messages:

As with the SYST:DATE command, if the UOP option is not present then then a **`241,"Hardware missing;UOP option`**" error is issued.

Other Notes:

The calendar/clock is unaffected by a *RST but it is cleared with a SERV:SEC:ERAS.

SYSTem:VERSion?/qonly/

Syntax:

Most common forms:

```
SYST:VERS?
```

Long forms:

```
SYSTEM:VERSION?
```

Description:

This command returns the version of SCPI used in the LB59xx sensors. The version is fixed at "2006.1" so as to be compatible with the U2000x.

Examples:

```
0000001 → SYST:VERS?
0000002 ← "2006.1"
```

Trigger

TRIGger:DELay:AUTO

TRIGger1:DELay:AUTO

TRIGger[:SEQuence]:DELay:AUTO

TRIGger:SEQuence1:DELay:AUTO

Syntax:

Most common forms:

```
TRIG:DEL:AUTO?
TRIG:DEL:AUTO [0|1]
```

Long forms:

```
TRIGGER:DELAY:AUTO?
TRIGGER:DELAY:AUTO [0|1]
TRIGGER1:SEQUENCE1:DELAY:AUTO?
```

Description:

This enables or disables the automatic settling time algorithm in the sensor. In some cases (large power level changes) additional time may be required. Even with this feature enabled, if you wish to ensure the measurement is settled to your liking take and compare two back to back readings. If this feature is disabled the sensor begins measurements as soon as a trigger occurs.

Examples:

In this example the automatic trigger delay is enabled and disabled.

```
0000053 → *RST
0000054 → TRIG:DEL:AUTO?
0000055 ← 1
0000056 → TRIG:DEL:AUTO 0
0000057 → TRIG:DEL:AUTO?
0000058 ← 0
0000059 → TRIG:DEL:AUTO 1
0000060 → TRIG:DEL:AUTO?
0000061 ← 1
```

On Reset:

As shown in the example, TRIG:DEL:AUTO is enabled with the sensor is *RST

TRIGger[:IMMediate]

TRIGger1[:IMMediate]/nquery/

TRIGger[:SEQuence]:IMMediate/nquery/

TRIGger:SEQuence1:IMMediate/nquery/

This command is the same as INIT, INIT:IMM or INITIATE:IMMEDIATE. Please consult the INIT:IMMEDIATE command elsewhere in the manual.

TRIGger[:SEQuence]:COUNt

TRIGger:SEQuence1:COUNt

Syntax:

Most common forms:

```
TRIG:COUN?
TRIG:COUN <number>
```

Long forms:

```
TRIGGER:SEQUENCE:COUNT?
TRIGGER:SEQUENCY:COUNT <number>
```

Description:

This is used to get a series of measurements (1-50) in a very short time period. To use trigger count the sensor must be in free run (INIT:CONT = 1) and MRAT = FAST or SUPer. With MRAT in fast you are restricted to 1 average. If MRAT = SUP you can set the averaging. However, a group of measurements will take longer (because averaging will be applied to each measurement). Trigger count can be set from 1 to 50. The default value of TRIG:COUN = 1. If TRIG:COUN is greater than 1 and you set MRAT to anything other than FAST or SUP you'll generate a -221,"Settings conflict" error message. Finally, in this mode you'll use FETCH? to retrieve the measure values. Each time FETCH? is called new values are returned.

Examples:

In this example we use trigger count with MRAT = FAST and SUP. Setting TRIG:COUN to 1 when MRAT is set to NORM or DOUB is also demonstrated.

```
0000065 → *RST
0000066 → INIT:CONT 1
0000067 → MRAT FAST

Take 5 measurements
0000068 → TRIG:COUN 5
0000069 → FETCH?
0000070 ← -7.35443947E+01,-7.48911388E+01,-7.47946415E+01,-7.54379048E+01,-7.42985568E+01
0000071 → FETCH?
0000072 ← -7.24080734E+01,-6.70442087E+01,-6.39921714E+01,-6.49239257E+01,-7.25393731E+01
0000073 → FETCH?
0000074 ← -6.03682284E+01,-6.17204047E+01,-6.90643411E+01,-7.33756753E+01,-7.27013575E+01

Take 25 measurements
0000075 → TRIG:COUN 25
0000076 → FETCH?
0000077 ← -7.29191367E+01,-7.44701710E+01,-7.52893055E+01,-7.45513153E+01,-7.22251496E+01,-
7.20036427E+01,-7.31956406E+01,-6.83442754E+01,-6.40323521E+01,-6.86150444E+01,-
7.19510663E+01,-7.17784388E+01,-7.14157007E+01,-7.01503679E+01,-7.20639699E+01,-
6.83238684E+01,-6.92899976E+01,-6.64262832E+01,-6.82780003E+01,-6.65846408E+01,-
6.51137256E+01,-6.53830441E+01,-7.14434742E+01,-6.71367839E+01,-6.41970447E+01
```

```
0000078 → FETCH?
0000079 ← -6.27615498E+01,-6.32740443E+01,-7.14917086E+01,-7.40471500E+01,-7.35028169E+01,-
7.43737523E+01,-7.37897553E+01,-7.01428597E+01,-7.30247388E+01,-7.49765855E+01,-
7.56000669E+01,-7.29210676E+01,-6.93758311E+01,-6.91521323E+01,-6.32603145E+01,-
6.03508988E+01,-5.99164785E+01,-6.21363357E+01,-6.88567995E+01,-7.25752253E+01,-
6.46710806E+01,-6.14748912E+01,-6.07040792E+01,-6.25539551E+01,-7.08493321E+01

Demonstrate the automatic setting of TRIG:COUN to 1 when entering MRAT = NORM, DOUB
0000080 → TRIG:COUN?
0000081 ← +25
0000082 → MRAT NORM
0000083 → FETCH?
0000084 ← -6.60257398E+01
0000085 → TRIG:COUN?
0000086 ← +1
0000087 → TRIG:COUN?
0000088 ← +1
0000089 → MRAT FAST
0000090 → TRIG:COUN 25
0000091 → TRIG:COUN?
0000092 ← +25
0000093 → MRAT DOUB
0000094 → TRIG:COUN?
0000095 ← +1

Demonstrate using MRAT SUPER with averaging enabled and set
0000096 → MRAT SUP
0000097 → TRIG:COUN 25

0000098 → AVER:COUN 1
0000099 → FETCH?
0000100 ← -6.74649564E+01,-7.04888926E+01,-7.18685634E+01,-7.30780375E+01,-7.38256239E+01,-
7.32865064E+01,-7.14980760E+01,-7.19042443E+01,-7.29505223E+01,-7.24571208E+01,-
7.28574115E+01,-7.33785127E+01,-7.30421127E+01,-6.69614439E+01,-6.33718800E+01,-
6.55794058E+01,-6.95728846E+01,-7.11227133E+01,-7.26417679E+01,-7.35419244E+01,-
7.22315382E+01,-7.00012812E+01,-6.97223095E+01,-7.05828003E+01,-6.99339710E+01

Increase AVERL:COUN… measurement took longer because of the increased averaging.
0000101 → AVER:COUN 20
0000102 → FETCH?
0000103 ← -6.08579985E+01,-6.30138628E+01,-6.03772131E+01,-7.22413889E+01,-6.26971233E+01,-
6.42394105E+01,-7.00969250E+01,-6.15620482E+01,-6.04158772E+01,-6.11781545E+01,-
6.30169957E+01,-6.50615501E+01,-6.62549224E+01,-6.46924124E+01,-6.86386605E+01,-
7.10452634E+01,-7.24810440E+01,-7.28060714E+01,-6.85563579E+01,-6.14815571E+01,-
6.01117589E+01,-5.98852302E+01,-6.53155626E+01,-6.53447041E+01,-6.40082301E+01
```

<u>On Reset:</u>

TRIG:COUN = 1

<u>Error Messages:</u>

When MRAT is NORM or DOUB and you attempt to set the trigger count to any value other than 1 a -221,"Settings conflict" error message will be generated.

<u>Notes:</u>

When MRAT is set to NORMAL or DOUBLE while TRIG:COUN > 1, TRIG:COUN is set to 1.

TRIGger[:SEQuence]:DELay

TRIGger:SEQuence1:DELay

Syntax:

Most common forms:

```
TRIG:DEL? [DEF|MIN|MAX]
TRIG:DEL <number>
```

Long forms:

```
TRIGGER:SEQUENCE:DELAY? [DEF|MIN|MAX]
TRIGGER:SEQUENCE:DELAY <number>
```

Description:

This sets the trigger delay or the time between the trigger event and the beginning of a measurement. This time may be between -1.00 and 1.00 seconds (+/-150ms). Note that the U2000 allows -0.15 to +0.15 seconds. The default value is 0 seconds. Delay time is resolved to 1usec.

Examples:

In this example the delay time is queried and set:

```
0000041 → *RST
0000042 → TRIG:DEL? DEF
0000043 ← +0.000000E+00
0000044 → TRIG:DEL?
0000045 ← +0.000000E+00
0000046 → TRIG:DEL? MIN
0000047 ← -1.000000E+01
0000048 → TRIG:DEL? MAX
0000049 ← +1.000000E+01
0000050 → TRIG:DEL 0.2
0000051 → TRIG:DEL?
0000052 ← +2.000000E-01
```

On Reset:

The trigger delay is set to 0 upon receiving a *RST command.

Notes:

TRIGger[:SEQuence]:HOLDoff

TRIGger:SEQuence1:HOLDoff

Syntax:

Most common forms:

```
TRIG:HOLD? [MIN,MAX,DEF]
TRIG:HOLD <numeric>
```

Long forms:

```
TRIGGER:SEQUENCE:HOLDOFF? [MIN|MAX|DEF]
TRIGGER:SEQUENCE:HOLDOFF <numeric>
```

Description:

This command prevents another trigger from occurring for a set period of time (between 1us to 400ms inclusive). One use of this command is to prevent unwanted triggers to occur. This can be very helpful with noisy or non-repeating signals.

Examples:

```
0000104 → TRIG:HOLD? MIN
0000105 ← +1.000000E-06
0000106 → TRIG:HOLD? MAX
0000107 ← +4.000000E-01
0000108 → TRIG:HOLD? DEF
0000109 ← +1.000000E-06
0000110 → TRIG:HOLD?
0000111 ← +1.000000E-06
0000112 → TRIG:HOLD 0.1
0000113 → TRIG:HOLD?
0000114 ← +1.000000E-01
```

On Reset:

TRIG:HOLD = 1usec (minimum time).

TRIGger[:SEQuence]:HYSTeresis

TRIGger:SEQuence1:HYSTeresis

Syntax:

Most common forms:

```
TRIG:HYST? [MIN|MAX|DEF}
TRIG:HYST <number>
```

Long forms:

```
TRIGGER:SEQUENCE:HYSTERESIS? [MIN|MAX|DEF]
TRIGGER:SEQUENCE:HYSTERESIS? <number>
```

Description:

This command can only be used when DET:FUNC = NORM. When DET:FUNC = AVER and you try to set TRIG:HYST you'll get a -221,"Conflict settings". This value is relative to TRIG:LEV and TRIG:SLOP. The hysteresis specifies how far the signal level must rise (negative slope) or fall (positive slope) relative to the trigger level before another trigger is detected.

So that if the trigger level is +10dBm, trigger slope is positive and hysteresis is 1dB then the signal level must fall below +9dBm before another trigger is detected. If the slope was negative then the trigger level must rise above +11dBm before another trigger will be acknowledged.

Examples:

This queries and sets hysteresis under various circumstances.

```
0000186 → *RST
0000187 → TRIG:HYST?
0000188 ← +0.000000E+00
0000189 → TRIG:HYST? DEF
0000190 ← +0.000000E+00
0000191 → TRIG:HYST? MIN
0000192 ← +0.000000E+00
0000193 → TRIG:HYST? MAX
0000194 ← +3.000000E+00
0000195 → DET:FUNC?
0000196 ← AVER
0000197 → TRIG:HYST 1.0
0000198 → SYST:ERR?
0000199 ← -221,"Settings conflict"
0000200 → DET:FUNC NORM
0000201 → TRIG:HYST 1.0
0000202 → SYST:ERR?
0000203 ← +0,"No error"
```

On Reset:

TRIG:HYST = 0.0

Common Error Messages:

If DEF:FUNC is not NORM and you attempt to set TRIG:HYST a -221,"Settings conflict" error message is generated.

TRIGger[:SEQuence]:LEVel

TRIGger:SEQuence1:LEVel

Syntax:

Most common forms:

```
TRIG:LEV?[MIN|MAX|DEF]
TRIG:LEV <number>
```

Long forms:

```
TRIGGER:SEQUENCE:LEVEL?[MIN|MAX|DEF]
TRIGGER:SEQUENCE:LEVEL  <number>
```

Description:

This command can only be used when DET:FUNC = NORM. When DET:FUNC = AVER and you try to set TRIG:LEV you'll get a -221,"Conflict settings".

This set the level at which a trigger is generated and a measurement begins. The level refers to the level of the incoming signal. So that this feature is only useful if, in addition to DET:FUNC, if the TRIG:SOUR = INT. If you do set the trigger level, TRIG:SOUR will be set to INT. Finally, if the signal applied (incoming signal) never crosses the threshold set by TRIG:LEV a trigger will never occur.

Examples:

```
0000218 → TRIG:LEV?
0000219 ← +0.000000E+00
0000220 → TRIG:LEV? MIN
0000221 ← -5.000000E+01
0000222 → TRIG:LEV? MAX
0000223 ← +2.000000E+01
0000224 → TRIG:LEV? DEF
0000225 ← +0.000000E+00
0000226 → TRIG:LEV -10
0000227 → SYST:ERR?
0000228 ← -221,"Settings conflict"
0000229 → DET:FUNC NORM
0000230 → TRIG:LEV -10
0000231 → TRIG:LEV?
0000232 ← -1.000000E+01
0000233 → TRIG:SOUR?
0000234 ← INT
```

On Reset:

TRIG:LEV = 0

TRIGger[:SEQuence]:SLOPe

TRIGger:SEQuence1:SLOPe

Syntax:

Most common forms:

TRIG:SLOP?
TRIG:SLOP [NEG|POS]

Long forms:

TRIGGER:SEQUENCE:SLOPE?
TRIGGER:SEQUENCE:SLOPE [NEG|POS]

Description:

Trigger slope is used only when the TRIG:SOUR = INT or EXT. And if the source is external the detector can be AVER or NORM. However, you can set its value at any time. If the value is POS then the trigger will occur on the rising edge. If the value is NEG then the trigger will occur on a falling edge.

Examples:

```
0000254 → *RST
0000255 → DET:FUNC?
0000256 ← AVER
0000257 → TRIG:SLOP?
0000258 ← POS
0000259 → TRIG:SLOP NEG
0000260 → TRIG:SLOP POS
```

On Reset:

On *RST the value is set to POS

TRIGger:SOURce

TRIGger[1]:SOURce

TRIGger[:SEQuence]:SOURce

TRIGger:SEQuence1:SOURce

Syntax:

Most common forms:

```
TRIG:SOUR?
TRIG:SOUR [IMM|INT|EXT|HOLD|BUS]
TRIG:SOUR
```

Long forms:

```
TRIGGER:SEQUENCE:SOURCE?
TRIGGER:SEQUENCY:SOURCE [IMM|INT|EXT|HOLD|BUS]
```

Description:

This command sets (or queries) the current trigger source. While this command determines the source of the trigger it does not necessarily place the sensor in a state to respond to the trigger. The INIT command will do this unless INIT:CONT is true.

Trigger Source	**Notes**
IMMediate	This causes the trigger system to always be enabled. If INIT:CONT = True then the sensor continuously generates INIT command or triggers causing the sensor to be in free run. The sensor returns to the idle state upon completing a measurement (of course if INIT:CONT = True then another measurement is initiated.
INTernal	Uses the measured signal as a signal source. The detector must in the NORMAL mode (not AVERAGE mode. See the example below.
EXTernal	The TTL compatible trigger and the Trigger In port is used to generate the INIT. If no trigger appears then no measurement occurs. This could be the source of timeouts.
HOLD	Causes triggering to be disabled or suspended unless TRIG:IMM is used
BUS	Waits for a *TRG SCPI command

Examples:

In this example the trigger source is set to IMM, EXT and finally to INT. The trigger states IMM and EXT work just fine but INT causes and error message. The detector function is changed to NORMAL and then setting the trigger source is accepted without error:

```
0000015 → *RST
0000016 → TRIG:SOUR?
0000017 ← IMM
```

```
0000018 → TRIG:SOUR EXT
0000019 → TRIG:SOUR?
0000020 ← EXT
0000021 → TRIG:SOUR IMM
0000022 → TRIG:SOUR INT
0000023 → SYST:ERR?
0000024 ← -221,"Settings conflict"
0000025 → DET:FUNC?
0000026 ← AVER
0000027 → DET:FUNC NORM
0000028 → TRIG:SOUR INT
0000029 → SYST:ERR?
0000030 ← +0,"No error"
```

On Reset:

The trigger source is set to IMM upon *RST.

Common Error Messages:

- If the trigger source is set to INT or EXT with MRAT = FAST a -221,"Setting conflict" error will be generated.
- If the source is set to INT and DET:FUN = AVER then a -221,"Settings conflict" error is generated.
- If the source is set to INT with using LB59xxL sensor then a -221,"Settings conflict" error is generated.

Unit

The units command sets the units of the returned value.

UNIT:POWer

UNIT1:POWer

Syntax:

Most common forms:

```
UNIT:POW < DBM|W >
UNIT:POW?
```

Long forms:

```
UNIT:POWER < DBM|W>
UNIT:POWER?
```

Description:

These commands sets or gets the measurement units to DBM or W (Watts).

Example:

This sequence sets and gets the units to Watts and DBM. It also demonstrates the effect on measurements. The power level from the source was set to -10dBm. Note that when units are set to W a value of 9.26… E-05 is returned. This is about 92.6microWatts or about -10.33dBm. Just as shown in the sequence.

```
0000054 → *RST
0000055 → UNIT:POW?
0000056 ← DBM
0000057 → UNIT:POWER?
0000058 ← DBM
0000059 → UNIT:POWER W
0000060 → UNIT:POWER?
0000061 ← W
0000062 → READ?
0000063 ← +9.26046559E-05
0000064 → UNIT:POW DBM
0000065 → READ?
0000066 ← -1.03342328E+01
```

On Reset

The units are set to DBM on reset

Standard SCPI commands

These commands are the IEEE 488.2 commands that are supported by the LB59xx. These commands are supported by most USBTMC compatible instruments.

***CLS/nquery/**

Syntax:

Most common forms:

```
*CLS
```

Long forms:

```
*CLS
```

Description:

This command clears all status information including:

- SCPI registers
- Standard event register
- Status byte
- Error message queue

Examples:

This example first tests to ensure that no error is present and the status byte is 0. Then a deliberate error is made. The status byte is read showing that, in this case, an error is present in the error queue. Then *CLS is executed and the status byte and error messages are read. Of course the *CLS should clear the status byte and error message queue.

```
0000158 → *STB?
0000159 ← +0
0000160 → SYST:ERR?
0000161 ← +0,"No error"
0000162 → FREQ QERQWER                          force an error state
0000163 → *STB?
0000164 ← +4                                    we've got a non-zero status byte
0000165 → SYST:ERR?
0000166 ← -224,"Illegal parameter value"        we've got an error message
0000167 → FREQ QERQWER                          force the error again
0000168 → *STB?
0000169 ← +4                                    we've got a non-zero status byte
0000170 → *STB?
0000171 ← +4
0000172 → *STB?
0000173 ← +4
0000174 → *CLS                                  now we'll clear the status byte…
0000175 → *STB?                                 …sure enough it is cleared
0000176 ← +0
0000177 → SYST:ERR?                             and the message queue is cleared
0000178 ← +0,"No error"
```

*ESE

Syntax:

Most common forms:

```
*ESE <0..7>
*ESE?
```

Long forms:

*ESE <0..7>

*ESE?

Description:

This command sets or gets the Standard Event Status Enable Register. This register is used to mask the output of Standard Event Status Register bits that are, in turn, logically or'ed together. The result the logical or is used to set or clear bit 5 of the Status Register byte (*STB?).

Bit	Weight	Meaning
0	1	Operation complete
1	2	Request control (not used)
2	4	Query error
3	8	Device dependent error
4	16	Execution error
5	32	Command error
6	64	Not used
7	128	Power on

Examples:

In the following sequence we repeatedly check, change and check the value of the event status register enable.

```
0000680 → *ESE?
0000681 ← +0
0000682 → *ESE 1
0000683 → *ESE?
0000684 ← +1
0000685 → *ESE 255
0000686 → *ESE?
0000687 ← +255
0000688 → *ESE 0
0000689 → *ESE?
0000690 ← +0
```

On Reset

The ESE register is cleared on a power on. It is unaffected by *RST.

*ESR?/qonly/

Syntax:

Most common forms:

```
*ESR?
```

Long forms:

```
*ESR?
```

Description:

This command returns the contents of the Standard Event Status Register. Once it is read, the register is cleared. The meaning of the individual bits are as follows:

Bit	Weight	Meaning
0	1	Operation complete
1	2	Request control (not used)
2	4	Query error
3	8	Device dependent error
4	16	Execution error
5	32	Command error
6	64	Not used
7	128	Power on

Examples:

This is an example of reading the Standard Event Status Register shortly after powered up. Note that the first returned value indicates “Power On + Operation Complete.” After being read the register is cleared.

```
0000703 → *ESR?
0000704 ← +129
0000705 → *ESR?
0000706 ← +0
```

*IDN?/qonly/

Syntax:

Most common forms:

`*IDN?`

Long forms:

`*IDN?`

Description:

This command requests the sensors identity. Specifically it requests the manufacturer, model number, serial number and firmware revision.

Examples:

In this example the sensor responds to a *IDN? command.

```
0000214 → *IDN?
0000215 ← LadyBug Technologies LLC,LB5926L,177464,0.99.242
```

The manufacturer is: **`LadyBug Technologies LLC`**

The model is: **`LB5926L`**

The serial number is: **`177464`**

The firmware version is: **`0.99.242`**

*OPC

Syntax:

Most common forms:

```
*OPC
*OPC?
```

Long forms:

```
*OPC
*OPC?
```

Description:

*OPC causes the sensor to set the operation complete bit in the Standard Event Status register when all pending operations are complete. The query returns a 1 when all pending operations are complete.

*OPT?/qonly/

Syntax:

Most common forms:

```
*OPT?
```

Long forms:

```
*OPT?
```

Description:

This command returns the option information for the sensor. See the example below.

Examples:

In the following example the sensor indicates that that it has options 001, 003 and a 3.5mm connector installed.

```
0000290 → *OPT?
0000291 ← "001,003,35M"
```

***RCL/nquery/**

Syntax:

Command form:

```
*RCL <NRf>
```

Description:

The command recalls a previously saved sensor state from the specified register. The recalled state then becomes the current sensor state. A state must have been previously saved to the specified register otherwise an error will result. Note that the registers are 1 based (1…10) for this command and the *SAV command.

Examples:

```
0000085 → *RCL 1
```

*RST/nquery/

Syntax:

Most common forms:

```
*RST
```

Long forms:

```
*RST
```

Description:

This command causes the sensor to reset itself. Note that this changes the state of the sensor to its default state. However, errors are note cleared.

Examples:

```
0000280 → *RST
```

*SAV/nquery/

Syntax:

Command form:

```
*SAV <NRf>
```

Description:

This command saves the current state of the sensor in the specified register. Note that the registers are 1 based (1…10) for this command and the *RCL command.

Example:

The following sequence saves the current state in register 1.

```
0000072 → *SAV 1
```

*SRE

Syntax:

Most common forms:

```
*SRE <0…255>
*SRE?
```

Long forms:

```
*SRE <0…255>
*SRE?
```

Description:

The values passed in the *SRE command are floats or integer. The value (if required) is rounded to an integer value. This command reports or controls the enable mask for the Service Request Register bits. This command either sets or gets the Service Request Enable register. The Service Request Enable register bits are as follows:

Bit	Weight	Meaning
0	1	Not used
1	2	Not used
2	4	Device dependent
3	8	QUEStionable Status Summary
4	16	Message available
5	32	Event Status But
6	64	Not used
7	128	OPERation Status Summary

The Status Register Enable may take on any value of between 0..255 inclusive.

The value is the sum of the enabled bits. If a 1 occupies any position in the Service Request Enable register then that bit is enabled in the Status Byte Register. If a 0 occupies any position in the Service Request Enable register then that bit is disabled in the Status Byte Register.

For instance, if the Status Request Enable register value is set to a value of 20 (only bits 2 and 4 are set) then Device dependent and Message available events will be made available to the status register as they occur.

Examples:

In this simple example the value is set to 20 and then checked.

```
0000404 → *SRE?
0000405 ← +0
0000406 → *SRE 20
0000407 → *SRE?
0000408 ← +20
```

<u>On Reset</u>

The value is set to 0.

<u>Common Error Messages:</u>

<u>Other Notes:</u>

*STB?/qonly/

Syntax:

Most common forms:

```
*STB?
```

Long forms:

```
*STB?
```

Description:

This command returns a single byte summarizing the status information of the sensor. Each bit in the eight byte summary reports a particular status (or is unused). The meaning of each bit is shown in the following table:

Bit	Weight	Interpretation
0	1	Not used
1	2	Device dependent or specific status
		0 – A device specific status condition has occurred
		1 – A device specific status condition has occurred
2	4	Error/Event queue
		0 – Queue is empty
		1 – Queue is not empty
3	8	Questionable status
		0 – A questionable status condition has not occurred
		1 – A questionable status condition has occurred
4	16	Message available
		0 – A message is not available
		1 – A message is available
5	32	Event status bit
		0 – a status event or condition has not occurred
		1 – a status event or condition has occurred
6	64	Master summary status
		0 – LB59xx is not requesting service
		1 – LB59xx is requesting service
7	128	Operation status summary
		0 – No operation status conditions have occurred
		1 – One or more operation status conditions have occurred

Examples:

```
0000003 → *STB?
0000004 ← +0
```

*TRG/nquery/

Syntax:

Most common forms:
`*TRG`

Long forms:
`*TRG`

Description:

This command triggers the LB59xx sensor if it is waiting for a trigger.

Common Error Messages:

- If TRIGger[1]:SOURce is not set to BUS error -211 “Trigger ignored” error is generated
- If the sensor is not waiting for a trigger then -211 “Trigger ignored” is generated

*TST?/qonly/

Syntax:

Most common forms:

`*TST?`

Long forms:

`*TST?`

Description:

This command causes the LB59xx to run a self-test. The result of the self-test is returned. If the return value = 0 then no fault was found. If the return value ≠ 0 or it is 1 then a fault was found. This command requires more than 20 seconds to complete.

Examples:

```
0000277 → *TST?
0000278 ← 0
```

*WAI/nquery/

Syntax:

Most common forms:

`*WAI`

Long forms:

`*WAI`

Description:

This command causes the sensor to wait for one of the following:

- All pending operations complete
- Device clear is received
- Power is cycled

DCL

This is the device clear command. When a DCL is issued to these sensors:

- All pending operations are halted and the instrument is placed in an idle mode
- The parser is reset
- The return/measurement output buffer is cleared

Unattended Operation (notes)

UOPeration:ACTive

UOPeration[:BASic]:INTerval

UOPeration[:BASic]:RBACk?/qonly/

UOPeration[:BASic]:RECord

UOPeration[:BASic]:ROEN

These commands are used with the unattended operation option. For more information please consult the appropriate User's Guide.