TEST EQUIPMENT PLUS

LINUX API for Signal Hound USBSA-44 version 0.1

Signal Hound Application Programming Interface

TEST EQUIPMENT PLUS

Signal Hound USBSA-44 LINUX Application Programming Interface (API)

35707 NE 86th Ave
Phone (360) 263-5006 • Fax (360) 263-5007

Table of Contents

Chapter

1

Introduction

About the Linux Signal Hound API and building applications.

The Signal Hound Application Programming Interface is a tool for software engineers to design custom applications for the Signal Hound. Like the Signal Hound Graphical User Interface (GUI), the API is used to send commands to, and receive data from, the Signal Hound device. But unlike the Signal Hound GUI, you have the flexibility as a programmer to control the Signal Hound at a lower level, and process, log or store data in any format you choose.

DISCLAIMER—This API is provided free of charge, without warranty or support. Software developers may only use this API with a genuine Signal Hound®.

This API is in BETA testing. Anticipate a few hiccups, and please report problems encountered.

The Linux API consists of a static library, libSHLAPI.a, and the Class CUSBSA (with files CUSBSA.cpp, CUSBSA.h, and SHLAPI.h). The library is built in C and can be used with any C compiler. The class, which provides increased encapsulation and ease of use, is C++ and will typically be compiled with gcc or an equivalent compiler. It is assumed the user is familiar with the Linux environment and including libraries in custom projects, and that you have glanced at the windows-based API manual.

This manual will primarily discuss using the static library with the CUSBSA class.

CUSBSA has been designed to make acquiring sweep data as easy as possible. It includes constructors and destructors, and manages trace data allocation for you. It will automatically call default initialize() and configure() commands if you skip ahead to just grabbing a sweep, so for some applications only a few lines of code are required. The simplest possible block of code to acquire a sweep from e.g. 310 MHz to 390 MHz is:

```
#include <iostream>
#include "CUSBSA.h"
using namespace std;
int main()
{
   CUSBSA mySignalHound;
   int i,returnCount = mySignalHound.FastSweep(310.0e6, 390.0e6);
   for(i=0; i<returnCount; i++)
     cout << mySignalHound.dTraceFreq[i]<< " Hz " << mySignalHound.dTraceAmpl[i] <<" dBm"<< endl;
}
```

Chapter

2

Using the Signal Hound in Linux

Installing the Linux Drivers

The Linux drivers can be downloaded from ftdichip.com. Please review the Readme before proceeding and follow the instructions.

The Signal Hound uses the D2XX drivers. For Linux, this means that if your build already includes the Virtual COM port, or VCP, drivers (Ubuntu does by default), you must "rmmod" or "blacklist" ftdi_sio and usbserial.

You may require a USB "Y" cable, with two type-A plugs, like you get with an external USB hard drive. I do not fully understand why, but mine would not reliably work without it.

I developed the API and CUSBSA class using the Code::Blocks IDE.

Chapter

3

Function Listing

A Partial List of Functions for the class CUSBSA. This list will be updated as functions are added.

Initialization

Functions: int CUSBSA::Initialize()
int CUSBSA::Initialize(unsigned char * pCalData)

Arguments: pCalData (optional), a 4096 byte table of cal data

Execution Time: 1-8 seconds
Return values:
0 for success, otherwise returns error code (see appendix)

Remarks:
This function will automatically be called the first time you sweep.
You MUST call this function before setting preamplifiers, external references, etc.
Returns much faster when cal data is supplied.

Configuration

Function: int CUSBSA:: Configure(double attenVal=10.0, int mixerBand=1, int sensitivity=0, int decimation=1, int IF_Path=0, int ADC_clock=0)

Arguments:
attenVal—Attenuator setting. Must be **0.0, 5.0, 10.0, or 15.0** dB. 10 dB is default.

mixerBand—For RF input frequencies below 150 MHz this should always be set to **0**. For RF frequencies above 150 MHz this should always be set to **1**.

Sensitivity—For lowest sensitivity, set to **0**. For highest sensitivity set to **2**.

Decimation—Sample rate is equal to 486.1111 Ksps divided by this number. Must be **between 1 and 16, inclusive**. Part of resolution bandwidth (RBW) calculation.

IF Path—Set to **0** for default 10.7 MHz Intermediate Frequency (IF) path. This path has higher selectivity but lower sensitivity. Set to **1** for 2.9 MHz IF path.

ADC clock—Set to 0 to select the default 23 1/3 MHz ADC clock. Set to 1 to select the for 22 ½ MHz ADC clock, which is useful if your frequency is a multiple of 23 1/3 MHz.

Execution Time: 400 msec or less.

Return values:
0 for success, otherwise returns error code (see appendix)

Remarks:
This function configures the Signal Hound and prepares it to receive an RF signal.
This function will automatically be called before your first sweep if you omit it, but should be used to optimize Signal Hound configuration for best performance.

Slow Sweep

Function: int **CUSBSA::SlowSweep**(double startFreq, double stopFreq, int FFTSize=1024, int avgCount=16, int imageHandling=0)

Arguments:

StartFreq—Frequency of first amplitude value returned

StopFreq—Minimum frequency of last amplitude value returned. Due to rounding, several additional values may be returned

FFTSize —Size of FFT. This and the decimation setting are used to calculate RBW. May be 16-65536 in powers of 2.

avgCount —Number of FFTs that get averaged together to produce the output. The amount of data captured at each frequency is a product of FFTSize and avgCount. *This product must be a multiple of 512.*

imageHandling —Set to 0 for default, IMAGE REJECTION ON (mask together high side and low side injection). Set to 1 for HIGH SIDE INJECTION. Set to 2 for LOW SIDE INJECTION.

Execution Time: [40 + (FFTSize * avgCount * decimation) / 486] msec per slice. The number of slices is equal to decimation * (stop – start) / 201KHz, rounded up.

Return values:

Count of frequency and amplitude values returned.

Remarks:
This function captures an array of data into the object's frequency and amplitude arrays. Amplitude points are in dBm. Frequency points are in Hz. The first data point is equal to the starting frequency. Subsequent data points are spaced by 486.1111 KHz / FFT size / decimation.
Data is stored in **dTraceAmpl** and **dTraceFreq** arrays.

External Trigger: When the external trigger is enabled, this function is blocking until a logic high is received. Call this function with image handling set to 1, otherwise you will need a second trigger pulse for the image rejection sweep.

Fast Sweep

Function: int **CUSBSA::**FastSweep(double startFreq, double stopFreq, int FFTSize=16, int imageHandling=0)

Arguments:

StartFreq—Frequency of first amplitude value returned. This value is rounded to the nearest 200 KHz.

StopFreq—Frequency of last amplitude value returned. This value is rounded to the nearest 200 KHz.

FFTSize —Size of FFT. This is used to calculate RBW. May be 1 or 16-256, in powers of 2.

imageHandling —Set to 0 for default, IMAGE REJECTION ON (mask together high side and low side injection). Set to 1 for HIGH SIDE INJECTION. Set to 2 for LOW SIDE INJECTION.

Execution Time: [40 + 1.2 * slice count] msec for large sweeps, up to twice this for small sweeps. The number of slices is equal to (stop – start) / 200KHz, rounded up.

Return values:

Count of frequency and amplitude values returned.

Remarks:
Initialization and configuration are automatically called if they have not yet been called.
This function captures an array of data into the object's frequency and amplitude arrays. Amplitude points are in dBm. Frequency points are in Hz. For FFT size of 1 (raw power only), data points are spaced 200 KHz. Otherwise data points are spaced 400 KHz / FFT Size.
RBW is based on FFT size only, as decimation must be equal to 1.
Do not span the band break of 150 MHz.

Prior to calling Fast Sweep, configure as follows:
--DECIMATION MUST BE SET TO 1.
--IF PATH MUST BE SET TO 0.
--ADC CLOCK MUST BE SET TO 0.

Calculate RBW

Approximate RBW in Hz is automatically calculated for each sweep and is available in **m_dCalcRBW** immediately after the sweep.

Select External Reference

Function:
int CUSBSA::External10MHz()

Return values:
0 for success, otherwise returns error code (see appendix)

Remarks:
Takes about 50 msec. Checks for >0 dBm 10 MHz reference. If present, the external 10 MHz is selected.

Select External Trigger or Sync Out

Function:
void CUSBSA:: SetSyncTrig(int mode)

Remarks:
Mode may be set to:

SHAPI_EXTERNALTRIGGER	triggers on an external logic high
SHAPI_SYNCOUT	pulses high when data collection begins
SHAPI_TRIGGERNORMAL	triggers immediately

When using an external trigger (3.3V or 5V OK), some functions (slow sweep, measurement receiver) will wait for a logic high before beginning data collection. There is no timeout, so use the external trigger with caution as it will halt operations until a TTL high is received.

Using the RF Preamplifier (USB-SA44B)

Function:
int CUSBSA::SetPreamp(int value)

Arguments:
value = 0 for preamplifier **off,** 1 for preamplifier **on**

Remarks:

USB-SA44B only!!!

Turns on or turns off the RF preamplifier. The preamplifier can be used to improve the sensitivity and decrease LO feed-through for sensitive readings. Set the attenuator to ensure the preamplifier input sees less than -25 dBm of input power to avoid overdriving your mixer and distorting your signal. Turn off the preamplifier below 500 KHz.

Using Multiple Signal Hounds

Define multiple CUSBSA objects and use them independently.

Using the Measurement Receiver

MEAS_RCVR_STRUCT:

```
       // *** INPUTS ***
       double  RFFrequency;                 //RF carrier frequency (Hz)
       double  AudioLPFreq;                 //Audio LowPass Cutoff  (Hz)
       double  AudioBPFreq;                 //Audio BandPass Center (Hz)
       int     UseLPF;                      //Set to non-zero to use audio low-pass filter
       int     UseBPF;                      //Set to non-zero to use audio low-pass filter

       // *** OUTPUTS ***
       double  RFCounter;                   //RF frequency count out (Hz)
       double  AMAudioFreq;                 //AF frequency count out after AM demod (Hz)
       double  FMAudioFreq;                 //AF frequency count out after FM demod (Hz)
       double  RFAmplitude;                 //dB Full Scale.

       double  FMPeakPlus;                  //Peak Positive Modulation, in Hz
       double  FMPeakMinus;                 //Peak Negative Modulation, in Hz
       double  FMRMS;                       //RMS  Modulation, in Hz

       double  AMPeakPlus;                  // In percent
       double  AMPeakMinus;
       double  AMRMS;
```

Function:
int CUSBSA::RunMeasurementReceiver ()

Arguments:
CUSBSA public variable **m_MeasRcvr** must be populated with appropriate input values. Initialize and configure must be called before using this function.

Return values:
Ignore the int return value. **m_MeasRcvr** `will be fully populated upon return.`

Remarks:
It is strongly recommended that you use the 2.9 MHz IF in your **Configure** call. The incidental AM for the 2.9 MHz IF is much lower than the 10.7 MHz, and it is more sensitive.

Keep your **m_MeasRcvr.`RFAmplitude`** readings between -45 and -5 dB Full Scale (dBFS) for best accuracy. As you approach 0 dBFS, readings may become inaccurate. Above 0 dBFS readings are meaningless as you are overdriving the ADC.

You may **Configure()** different **sensitivity** and **attenuator** settings to change ranges, increasing system dynamic range. The practice of taking a reading immediately before changing range, then immediately after changing to calculate an offset works well, and is required for a large dynamic range.

The IF Bandwidth is controlled by the decimation setting in your **SHAPI_Configure** call.
IF Bandwidth = 240 KHz / decmation. Decimations of 1,2,4,8, or 16 are recommended. 64K samples are taken regardless of IF bandwidth, so with decimation set to 16 the function will take about 2 seconds to return.

External Trigger: When the external trigger is enabled, this function is blocking until a logic high is received.

Appendix

A

Error Codes

ERROR_HOUND_NOT_FOUND	100
ERROR_PACKET_HEADER_NOT_FOUND	101
ERROR_WRITE_FAILED	102
ERROR_WRONG_NUM_READ	103
ERROR_READ_TIMEOUT	104
ERROR_DEVICE_NOT_LOADED	105
ERROR_MISSING_DATA	106
ERROR_EXTRA_DATA	107
ERROR_OUT_OF_RANGE	200
ERROR_NO_EXT_REF	201