TEST EQUIPMENT PLUS

API for Signal Hound USBSA-44

Signal Hound Application Programming Interface

TEST EQUIPMENT PLUS

Signal Hound USBSA-44 Application Programming Interface (API)

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

Table of Contents

Chapter

1

Introduction

About the 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™.

A simple application will send a series of commands to the API. The **first command to the API must be the initialize command**. This takes twenty seconds to execute because it must download a calibration table from the Signal Hound device. While the GUI can store this as a local file and only load it once, the API has no such luxury.

The **second command to the API must be a configuration command**. This selects the attenuator settings, mixer, intermediate frequency, and clock settings.

After these first two commands, the Signal Hound is ready to begin collecting data. A set of functions are available to you, each tailored to get the most out of a particular aspect of the Signal Hound.

A sample application is available to you. A drop-down menu allows you to initialize, configure, and collect data from the Signal Hound through the API. It is written in Visual C++ 6, but may be readily ported to any number of languages.

The API consists of SH_API.dll, SH_API.h, SH_API.lib, and this document. To use, put the DLL you're your application's working directory, insert the header file into your application's source code, include the library file in your project's settings, then build your application. Typically the DLL is installed with your application. The library and header files are used to build your application. If you are familiar with using DLLs this should be a straightforward process. If not, please review your compiler's documentation before proceeding.

Chapter

2

Function Listing

A Complete List of Functions for the Signal Hound API

Initialization

Functions: int SHAPI_Initialize()
int SHAPI_InitializeNext()
int SHAPI_SelectDevice(int deviceToSelect)

Arguments: deviceToSelect, a number from 0 to 7, where 0 is the first device to initialize, 1 is the second, and so on.

Execution Time: 20 seconds approx
Return values:
0 for success, otherwise returns error code (see appendix)

Remarks:
You MUST call this function before any others and wait for it to complete!
SHAPI_Initialize initializes a single Signal Hound USB Interface.
SHAPI_InitializeNext initializes the next Signal Hound for a multiple Signal Hound application. A maximum of 8 Signal Hounds may be initialized.

The above functions also load correction constants from the Signal Hound's flash memory into the DLL.

SHAPI_SelectDevice is used to switch between multiple Signal hounds when more than one is initialized. For multi-device applications, many DLL functions use the "selected device." A handful of functions accept a device number as an argument for simultaneous operation of Signal Hounds.

Configuration

Function: int **SHAPI_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 must be called before data is captured.

For multi-device applications, the currently selected device is used.

Slow Sweep

Function: int **SHAPI_GetSlowSweep**(double * dBArray, double startFreq, double stopFreq, int &returnCount, int FFTSize=1024, int avgCount=16, int imageHandling=0)

Arguments:
dBArray —Pointer to array of double precision floating point numbers. This is where your data will get stored.

StartFreq—Frequency of first amplitude value returned

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

returnCount —Count of amplitude values 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:
0 for success, otherwise returns error code (see Appendix A)

Remarks:
This function captures an array of data. Data points are amplitude, in dBm. The first data point is equal to the starting frequency. Subsequent data points are spaced by 486.1111 KHz / FFT size / decimation. You may call SHAPI_GetSlowSweepCount to get the size of this array.

For multi-device applications, the currently selected device is used.

Fast Sweep

Function: int SHAPI_GetFastSweep(double * dBArray, double startFreq, double stopFreq, int &returnCount, int FFTSize=16, int imageHandling=0)

Arguments:
dBArray —Pointer to array of double precision floating point numbers. This is where your data will get stored.

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.

returnCount —Count of amplitude values returned.

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

Remarks:
This function captures an array of data. Data points are amplitude, in dBm. The first data point is equal to the starting frequency. 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 is equal to 1.

For multi-device applications, the currently selected device is used.

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

Function: double SHAPI_GetRBW(int FFTSize, int decimation)

Arguments:

FFTSize —Size of FFT. 8-65536, in powers of 2.

Decimation—Signal Hound Decimation setting

Return values:
RBW in Hz. Equal to 1.6384e6 / decimation / FFTSize;

Remarks:
Returns an approximation of the RBW. For an FFT size of 1024 and decimation of 16, an RBW of 100 Hz is returned.

Get Sweep Count

Functions:
int SHAPI_GetSlowSweepCount(double startFreq, double stopFreq, int FFTSize);
int SHAPI_GetFastSweepCount(double startFreq, double stopFreq, int FFTSize);

Arguments:

Start & Stop Frequencies, in Hz.
FFTSize —Size of FFT. 16-65536, in powers of 2.

Return values:
Sample count. May be used to allocate memory.

Remarks:
Returns the count of double precision floating point values to expect from GetSlowSweep or GetFastSweep.

Cycle the Device (Preset)

Function:
int SHAPI_CyclePort();

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

Remarks:
Takes about 2.5 seconds. MUST INITIALIZE FIRST. Presets the Signal Hound hardware. Useful to restore Signal Hound to a known state.

For multi-device applications, only call this before initializing Signal Hounds, or when preparing to exit the application. Otherwise, behavior is unknown.

Function:
SHAPI_CyclePowerOnExit()

Power cycles all Signal Hounds. Use prior to closing your software to restore the Signal Hound's state to a known condition.

Select External Reference

Function:
int SHAPI_SelectExt10MHz()

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.

For multi-device applications, the currently selected device is used.

Get I / Q Data Packet

Function:
int SHAPI_GetIQDataPacket (int * pIData, int * pQData, double &centerFreq, int size)

Arguments:

pIData—Pointer to the buffer where In-phase channel data will be stored. Values –32768 to +32767. Raw DAC output.

pQData—Pointer to the buffer where Quadrature-phase channel data will be stored. Values –32768 to +32767. Raw DAC output.

CenterFreq—Passed by reference. The API will modify this to the nearest available actual center frequency. You may correct the I/Q data to a specific frequency and phase alignment by applying a linear phase offset.

size —Number of I/Q data pairs to store. Must be multiple of 512, up to 128,000.

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

Remarks:
Changes to selected center frequency, high-side LO injection, no image rejection. Reports SIZE data points at current decimation / clock rates.

For multi-device applications, the currently selected device is used.

Operating Over Full Temperature Range (USB-SA44B)

Functions:
float SHAPI_GetTemperature()
int SHAPI_LoadTemperatureCorrections(LPCSTR filename)

Arguments:

filename —Pointer to the string with the temperature correction data, typically formatted as "D01234567.bin".

Return values:
SHAPI_LoadTemperatureCorrections: "true" for success, "false" for failure.
SHAPI_GetTemperature: internal temperature in °C (32-bit floating point value)

Remarks:

USB-SA44B only!!!

Call LoadTemperatureCorrections to maintain amplitude accuracy when operating at cold or hot temperatures.
Call SHAPI_GetTemperature to read the current temperature and use it for amplitude corrections.

For multi-device applications, the currently selected device is used.

Using the RF Preamplifier (USB-SA44B)

Function:
void SHAPI_SetPreamp(int value)

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

Remarks:

USB-SA44B only!!!

For multi-device applications, the currently selected device is used. 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.

Function:
int SHAPI_IsPreampAvailable()

Return value:
1 if a preamplifier is available, , e.g. a USB-SA44B
0 if a preamplifier is not available, e.g. a USB-SA44

Remarks:

Tests to see if a preamplifier is available, e.g. is this device a USB-SA44B?

Using Multiple Signal Hounds

Function:
SHAPI_SelectDevice(int deviceToSelect)

Arguments: deviceToSelect, a number from 0 to 7, where 0 is the first device to initialize, 1 is the second, and so on.

Remarks:
Selects device, 0-7, in the order they initialized.

Function:
unsigned int SHAPI_GetSerNum()

Return Value: a 32-bit unsigned integer representing the serial number of the currently selected device.

Remarks:
Use this to determine the serial number for each initialized device in multi-device applications.

Notes:
The multiple Signal Hound features were designed such that one primary device could be used to scan the spectrum, then another device or devices could simultaneously receive and examine any signals that were identified

Functions which can use any device at any time, not just the selected device:

Function:
int SHAPI_SetupLO(double &centerFreq, int mixMode=1, int deviceNum=-1)

Arguments: centerFreq, passed by reference. The LO is selected to get as close as possible to the desired center freq. The function sets this parameter to the actual center freq.

mixMode: High side or low side injection. High side (mixMode =1) is default.
deviceNum: A negative value uses the currently selected device. Otherwise, pass the device number 0-7 you wish to set up.

Remarks:
Sets up the LO to downconvert at a specific frequency, for receiving streaming I/Q data.

Function:
int SHAPI_StartStreamingData(int deviceNum=-1)
int SHAPI_StopStreamingData(int deviceNum=-1)

Arguments: deviceNum. A negative value uses the currently selected device. Otherwise, pass the device number 0-7 you wish to start or stop.

Remarks:
Starts or stops the streaming of I/Q data at the selected frequency. Once you start streaming data, you must stop it before using any functions **except** SHAPI_GetStreamingPacket.

Function:
int SHAPI_GetStreamingPacket(int *bufI, int *bufQ, int deviceNum=-1)

Arguments: deviceNum. A negative value uses the currently selected device. Otherwise, pass the device number 0-7 you wish to get data from.
bufI, bufQ: 32-bit integer buffers of size 4096 samples, to receive the unprocessed I/Q data. Values will be -32768 to 32767.

Remarks:
Call this function to get the next chunk of 4096 samples from the receive buffer. Must be called in a timely fashion or data will be lost. Returns when data is received. The decimation rate in SHAPI_Configure controls the sample rate, and can be used to reduce the IF bandwidth and amount of data received.

The sequence for receiving streaming data should be:

1. Initialize
2. Configure
3. Setup LO
4. Start Streaming
5. Repeatedly Get Streaming Packet
6. Stop Streaming

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 SHAPI_RunMeasurementReceiver (void * pMeasRcvrStruct)

Arguments:

pMeasRcvrStruct —Pointer to the measurement receiver structure, with the RF frequency and filter settings previously set.

Return values:
Ignore the int return value. Your `MEAS_RCVR_STRUCT will be fully populated upon return.`

Remarks:
To use: You must call **SHAPI_Initialize** followed by **SHAPI_Configure** before you call **SHAPI_RunMeasurementReceiver**. It is strongly recommended that you use the 2.9 MHz IF in your **SHAPI_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 **`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 change **sensitivity** and **attenuator** settings **SHAPI_Configure** to change ranges, increasing 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.

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