SA44 and SA124 Application Programming Interface (API) Programmers Reference Manual

SA44 and SA124 Application Programming Interface (API) Programmers Reference Manual

Version 3
May 31, 2016

Table of Contents

Overview

This manual is the programmer's reference for the Signal Hound SA44 and SA124 series of spectrum analyzers. The document details the programming interface to these two devices, which will be referred to throughout this document as the API. The API is a set of C language functions targeting the SA44/SA124 spectrum analyzers, used to control the devices for frequency domain sweeps and streaming IQ data. The API is C ABI compatible, so it can be called from a number of other languages and environments such as Java, C#, Python, C++, Matlab, and Labview.

This manual will describe the requirements and knowledge needed to program to the API. If you are new to the API you should read the sections in this order: *Build/Version Notes*, *Requirements, Theory of Operation*, and *Modes of Operation*.

If you want to start programming immediately, the appendix contains a number of code examples, and you can reference the rest of the manual when you need to.

The Build/Version Notes details the available builds for the API and notes major changes to API versions.
The Requirements section details the physical and operational needs to use the API.
The Theory of Operation section details how to interface the device and covers every major component a program will implement when interfacing a Signal Hound spectrum analyzer.
The Modes of Operation section attempts to teach you how to use the device in each of its operational modes, from the required functions, to interpreting the data the device returns.
The API Functions section covers every function in depth. The knowledge learned in the *Theory and Modes of Operation* sections will help you navigate the API functions.
The Appendix provides various code examples and tips.

Contact Information

We are interested in your feedback and questions. Please report any issues/bugs as soon as possible. We will maintain the most up to date API on our website. We encourage any and all criticisms or ideas. We would love to hear how you might improve the API.

All programming and API related questions should be directed to aj@signalhound.com
All hardware/specification related questions should be directed to justin@teplus.com

You can also contact us via phone, 1-800-260-8378.

Build/Version Notes

Two Windows builds are available for 32 and 64 bit operating systems. The builds are compiled with Visual Studio 2012. Distributing an application using this library will require distributing the VS2012 redistributable libraries. The libraries are distributed as Windows DLL files.

A Linux ARMv7-A build is available, tested on the Raspberry Pi 2. This limited version requires a factory firmware upgrade for firmware versions 2.10, 3.10, and 3.11. See Appendix:ARMv7-A Support for more information.

The SA-series drivers should automatically install when you install the Spike software. To manually install the SA-series drivers (e.g. USB-SA44B), navigate to the application folder (where you installed the

Spike software) and find the **CDM v2.12.00 WHQL Certified.exe** file. Right click it and Run as administrator. Follow the installation instructions

Version 3.0.0

Initial release. Version numbering will begin at 3.0.0 to signify this is the 3rd major iteration on the SA44/SA124 programming interface. While not a direct descendant of previous versions, there is a large amount of work that is borrowed from earlier versions of the APIs that it is to be considered this to be a derivative work.

The programming interface has drastically changed from previous versions. It is now similar to our BB60 product programming interfaces. The interface is modeled roughly after the IviSpecAn specification.

Requirements

Windows Development Requirements

Below is a list of requirements needed to begin.

- Windows 7/8. The API is untested on Windows XP.
- Windows C/C++ development tools/environment. Preferably Visual Studio 2008 or later. If Visual Studio 2012 is not used, then the VS2012 redistributables will need to be installed.
- The *sa_api.h* API header file.
- The API library (sa_api.lib) and dynamic library (sa_api.dll) files.
- USB Driver: CDM v2.12.00 WHQL Certified.exe from FTDI

ARM v7-A Development Requirements

See the included README in the ARM SDK for full development requirements.

General Requirements

- A basic understanding of RF Spectrum Analysis.
- A Signal Hound SA44 or SA124 spectrum analyzer.

Setup and Initialization

The API requires two calibration files to exist on the host PC before the API will work correctly. To understand how this is performed, read the sections Opening a Device and First Time Opening a New Device.

Theory of Operation

The flow of any program interfacing a Signal Hound device will be as follows.

1) Open a USB 2.0 connected SA44/SA124 spectrum analyzer and obtain a unique handle to the device. The handle will be used for all subsequent function calls.
2) Configure the device, such as setting the frequency sweep ranges or the IQ sample rate.
3) Initiate the device for a particular mode of operation, whether it be frequency domain sweeps or IQ streaming.
4) Get data from the device. Which functions are called and what data is returned depends on the mode of operation.
5) Abort the current mode of operation

6) Close the device

The API provides functions for each step in this process. We have strived to mimic the functionality and naming conventions of SCPIs *IviSpecAn* Class Specification where possible. It is not necessary to be familiar with this specification but those who are should feel comfortable with our API immediately. The following sections further detail each of the six steps listed above.

Opening a Device

Before attempting to open a device programmatically, it must be physically connected to a USB 2.0 port with the provided cable. Ensure the power light is lit on the device and is solid green. Once the device is connected it can be opened. The function `saOpenDevice()` and `saOpenDeviceBySerialNumber()` provides this functionality. This function returns an integer ID to the device which was opened. Up to eight devices may be connected and interfaced through our API using the IDs. The integer ID returned is required for every function call in the API, as it uniquely identifies which device you are interfacing.

First Time Opening a New Device

All Signal Hound SA series spectrum analyzers need two calibration files to operate. These files reside on internal flash memory for newer models, and on older models, on the SignalHound.com web server. The files are copied to the host machine the first time a device is opened on a particular machine. The files are saved at C:/ProgramData/SignalHound/cal_files. For models which need the calibration data from the Signal Hound server, the first time the device is opened the host PC needs to have an active network connection. Downloading this file from our server, or pulling it from the device internal flash can cause the first invocation of saOpenDevice() to block for up to 10 seconds. Once these files are on a host PC this process does not need to be performed again. If the devices have been used in the standard Signal Hound application, then the calibration files will already exist on the host PC.

Configuring the Device

Once the device is opened, it may be configured. The API provides a number of configuration routines for its many operating states. Configuration routines modify the device global state, such as the sweep frequency or IQ sampling rate. The configurations do not take effect until the device is initiated.

Initiating the Device

Each device has two states.

1) A global state set through the API configure routines.
2) An operational/running state.

All configurations functions modify the global state which does not immediately affect the operation of the device. Once you have configured the global state to your liking, you may re-initiate the device into a mode of operation, in which the global state is copied into the running state. At this point, the running state is separate and not affected by future configuration function calls.

The `saInitiate()` function is used to initialize the device and enter one of the operational modes. The device can only be in one operational mode at a time. If `saInitiate()` is called on a device that is already initialized, the current mode is aborted before entering the new specified mode.

Retrieve Data from the Device

Once a device has been successfully initiated you can begin retrieving data from the device. Every mode of operation returns different types, and different amounts of data. The Modes of Operation section will help you determine how to collect data from the API for any given mode. Helper routines are also used for certain modes to determine how much data to expect from the device.

Abort the Current Mode

Aborting the operation of the device is achieved through the `saAbort()` function. This causes the device to cancel any pending operations and return to an idle state. Calling `saAbort()` explicitly is never required. If you attempt to initiate an already active device, `saAbort()` will be called for you. Also if you attempt to close an active device, `saAbort()` will be called. There are a few reasons you may wish to call `saAbort()` manually though.

- Certain modes combined with certain settings consume large amounts of resources such as memory and the spawning of many threads. Calling `saAbort()` will free those resources.
- Certain modes such as Real-Time Spectrum Analysis consume many CPU cycles, and they are always running in the background whether or not you are collecting and using the results they produce.
- Aborting an operational mode and spending more time in an idle state may help to reduce power consumption.

Closing the Device

When you are finished, you must call `saCloseDevice()`. This function attempts to safely close the USB 2.0 connection to the device and clean up any resources which may be allocated. A device may also be closed and opened multiple times during the execution of a program. This may be necessary if you want to change USB ports, or swap a device.

Modes of Operation

Now that we have seen how a typical application interfaces with the API, let’s examine the different modes of operation the API provides. Each mode will accept different configurations and have different boundary conditions. Each mode will also provide data formatted to match the mode selected. In the next sections you will see how to interact with each mode.

For a more in-depth examination of each mode of operation (read: *theory*) refer to the Signal Hound spectrum analyzer user manual.

Swept Analysis

Swept analysis represents the most traditional form of spectrum analysis. This mode offers the largest amount of configuration options, and returns traditional frequency domain sweeps. A frequency domain sweep displays amplitude on the vertical axis and frequency on the horizontal axis.

The configuration routines which affect the sweep results are

- `saConfigAcquisition() – Configuring the detector and linear/log scaling`
- `saConfigCenterSpan() – Configuring the sweep frequency range`
- `saConfigLevel() – Configuring reference level for automatic gain and attenuation`
- `saConfigGainAtten() – Configuring internal amplifiers and attenuators`

- `saConfigSweepCoupling() – Configuring RBW and VBW`
- `saConfigProcUnits() – Configure VBW processing`

Once you have configured the device, you will call `saInitiate()` using the `SA_SWEEPING` flag.

This mode is driven by the programmer, causing a sweep to be collected only when the program requests one through the `saGetSweep()` functions. The length of the sweep is determined by a combination of resolution bandwidth, video bandwidth, and span.

Once the device is initialized you can determine the characteristics of the sweep you will be collecting with `saQuerySweepInfo()`. This function returns the length of the sweep, the frequency of the first bin, and the bin size. You will need to allocate two arrays of memory, representing the minimum and maximum values for each frequency bin.

Now you are ready to call the `saGetSweep()` and `saGetPartialSweepFunctions()`.

You can determine the frequency of any bin in the resulting sweep by the function below where 'n' is the zero based index into the sweep array.

$$Frequency\ of\ n'th\ sample\ point\ in\ returned\ sweep = startFreq + n * binSize$$

Real-Time Analysis

The API provides the functionality of an online real-time spectrum analyzer for the full instantaneous bandwidth of the device (250kHz). In real-time FFTs are applied at an overlapping rate of 87.5%.

The configuration routines which affect the spectrum results are the same for swept analysis, but span is restricted to 250kHz.

The number of sweep results far exceeds a program's capability to acquire, view, and process, therefore the API combines sweeps results for a user specified amount of time. It does this in two ways. One, is the API either max holds or averages the sweep results into a standard sweep. Also the API creates an image frame which acts as a density map for every sweep result processed during a period of time. Both the sweep and density map are returned at rate specified by the function *saConfigRealTime.* For a full example of using real-time see Appendix: Code Examles: Real-Time Mode.

IQ Streaming

The API is capable of providing programmers with a continuous stream of digital IQ samples from the device. The digital IQ stream consists of interleaved 32-bit floating point IQ pairs scaled to mW. The digital samples are amplitude corrected providing accurate measurements. The IQ data rate at its highest is 486.111111~ kS/s and can be decimated down by a factor of up to 128 (in powers of two). Each decimation value further reduces the overall bandwidth of the IQ samples, so the API also provides a configurable bandpass filter to control the overall passband of a given IQ data stream. The IQ data stream can also be tuned to an arbitrary center frequency.

Configuration routines used to prepare streaming are

- `saConfigCenterSpan() – Set the center frequency of the IQ data stream. Span is ignored.`
- `saConfigLevel() – Set the expected input level.`

- `saConfigGainAtten() – See Appendix: Setting Gain and Attenuation.`
- `saConfigIQ() – Specify the decimation and bandwidth of the IQ data stream.`

Once configured, initialize the device with the SA_IQ mode. Data acquisition begins immediately. The API buffers ~3/4 second worth of digital samples in circular buffers. It is the responsibility of the user application to poll the samples via `saGetIQData()` fast enough to prevent the circular buffers from wrapping. We suggest a separate polling thread and synchronized data structure(buffer) for retrieving the samples and using them in your application.

NOTE: Decimation / Filtering / Calibration occur on the PC and can be processor intensive on certain hardware. Please characterize the processor load if you think this might be an issue for your application.

Audio Demodulation

Audio demodulation can be achieved using saConfigAudio, saInitiate, and saGetAudio. See saConfigAudio to see which types of audio demodulation can be performed.

saConfigAudio is used to specify the type of demodulation and the characteristics of the filters. The center frequency is specified in saConfigAudio. Audio demodulation does not have auto ranging as does other operational modes, so saConfigGainAtten must be called to configure the internal gain, attenuator, and preamplifier. Once desired settings are chosen, call saInitiate with the SA_AUDIO mode flag to begin streaming audio.

Once the device is streaming, use saGetAudio to retrieve 4096 audio samples for an audio sample rate of 30382. The API buffers many seconds worth of audio.

Scalar Network Analysis

When a Signal Hound tracking generator is paired together with a spectrum analyzer, the products can function as a scalar network analyzer to perform insertion loss measurements, or return loss measurements by adding a directional coupler. Throughout this document, this functionality will be referred to as tracking generator (or TG) sweeps.

Scalar Network Analysis can be realized by following these steps

1. Ensure a Signal Hound spectrum analyzer and tracking generator is connected to your PC.
2. Open the spectrum analyzer through normal means.
3. Associate a tracking generator to a spectrum analyzer by calling saAttachTg. At this point, if a TG is present, it is claimed by the API and cannot be discovered again until saCloseDevice is called.
4. Configure the device as normal, setting sweep frequencies and reference level (or manually setting gain and attenuation).
5. Configure the TG sweep with the saConfigTgSweep function. This function configures TG sweep specific parameters.
6. Call saInitiate with the SA_TG_SWEEP mode flag.
7. Get the sweep characteristics with saQuerySweepInfo.
8. Connect the SA and TG device into the final test state without the DUT and perform one sweep with saGetSweep or saGetPartialSweep. After one full sweep has returned, call saStoreTgThru with the TG_THRU_0DB flag.

9. (Optional) Configure the setup again still without the DUT but with a 20dB pad inserted into the system. Perform an additional full sweep and call saStoreTgThru with the TG_THRU_20DB.
10. Once store through has been called, insert your DUT into the system and then you can freely call the get sweep functions until you modify the configuration or settings.

If you modify the test setup or want to re-initialize the device with a new configuration, the store through must be performed again.

For a full code example, see the Appendix : Scalar Network Analysis Sweep.

API Functions

saGetSerialNumberList

Get a list of available devices connected to the PC

```
saStatus saGetSerialNumberList(int serialNumbers[8], int *deviceCount);
```

Parameters

serialNumbers	A pointer to an array of at minimum 8 contiguous 32-bit integers. It is undefined behavior if this array pointed to by *serialNumbers* is not 8 integers in length.
deviceCount	Pointer to a 32-bit integer.

Description

This function returns the devices that are unopened in the current process. Up to 8 devices max will be returned. The serial numbers of the unopened devices are returned. The array will be populated starting at index 0 of the provided array. The integer pointed to by *deviceCount* will equal the number of devices reported by this function upon returning.

Return Values

saNoError	The function returned successfully.
saNullPtrErr	One or more pointer parameters were NULL.

saOpenDevice

Open one Signal Hound device

```
saStatus saOpenDevice(int *device);
```

Parameters

device	Pointer to 32-bit integer variable. If this function returns successfully, the value *device* points to will contain a unique handle value to the device opened. This number is used for all successive API function calls.

Description

This function attempts to open the first SA44/SA124 it detects. If a device is opened successfully , a handle to the device will be returned through the *device* pointer which can be used to target that device for other API calls.

This function when successful, takes about 5 seconds to perform. If it is the first time a device is opened on a host PC, this function can potentially take much longer. See First Time Opening a New Device.

Return Values

saNoError	The function returned successfully.
saNullPtrErr	The *device* parameter is NULL.
saDeviceNotFoundErr	A valid Signal Hound device was not found.
saInternetErr	This error is returned if the API is unable to acquire the calibration files from the Signal Hound server. Read First Time Opening a New Device for more information.

saOpenDeviceBySerialNumber

Open one Signal Hound device

```
saStatus saOpenDeviceBySerialNumber(int *device, int serialNumber);
```

Parameters

device	Pointer to 32-bit integer variable. If this function returns successfully, the value *device* points to will contain a unique handle value to the device opened. This number is used for all successive API function calls.
serialNumber	User provided serial number.

Description

This function is similar to `saOpenDevice()` except you can specify the serial number of the device you wish to open. Everything else is identical.

Return Values

See `saOpenDevice()` for return values.

saCloseDevice

Close one Signal Hound device

```
saStatus saCloseDevice(int device);
```

Parameters

device	Device handle.

Description

This function is called when you wish to terminate a connection with a device. Any resources the device has allocated will be freed and the USB 2.0 connection to the device is terminated. The device closed will be released and will become available to be opened again. Any activity the device is performing is aborted automatically before closing.

Return Values

saNoError	The device closed successfully.
saDeviceNotOpenErr	The device specified is not open.

saPreset

Trigger a device preset

```
saStatus saPreset(int device);
```

Parameters

device	Device handle.

Description

This function exists to invoke a hard reset of the device. This will function similarly to a power cycle(unplug/re-connect the device). This might be useful if the device has entered an undesirable or unrecoverable state. This function might allow the software to perform the reset rather than ask the user perform a power cycle.

Functionally, in addition to a hard reset, this function closes the device as if *saCloseDevice* was called. This means the device handle becomes invalid and the device must be reopened for use.

This function is a blocking call and takes about 2.5 seconds to return.

Return Values

saNoError	Function completed successfully, the device will be reset.
saDeviceNotOpen	The device specified is not currently open.

Example

```
// This short function shows how to preset a device in the
//   quickest way possible. Both saPreset and saOpenDevice
//   are blocking calls. It may be preferred to perform this
//   function in a separate thread.
saStatus PresetDevice(int *device_id)
{
      saPreset(*device_id);
      return saOpenDevice(device_id);
}
```

saSetCalFilePath

Specify the directory where the calibration files will reside.
Only call this function if you wish to change the default calibration file directory.

```
saStatus saSetCalFilePath(const char *path);
```

Parameters

path	Relative or absolute path specifying the directory in which the API will look for and store the SA44 and SA124 calibration files, specifically the .bin and .tep files. Set to NULL (zero) to revert to the default path.

Description

This function should be called before opening the device. Ideally it should be the first function called in a program interfacing the device. The path specified should be suffixed with the '/' or '\\' character denoting that it is a directory. Failure to append a slash character will result in undesired behavior.

See examples of usage below.

When a device is opened, the API will look in the supplied path for the required calibration files. If the folder does not exist, it will be created. If the files do not exist in the folder, they will be acquired from the device flash memory where applicable, and for all other devices downloaded from the Signal Hound web server. The files will be placed in the supplied directory for future use. If the files exist in the directory, they will simply be loaded into memory.

This function does not need to be called, as it will use a default system path, typically C:/ProgramData/SignalHound/cal_files/

Examples

To set the working directory as the calibration file path call the function as

```
saSetCalFilePath(".\\");
```

To set an absolute path, you might call the function like so

```
saSetCalFilePath("C:\\ProgramData\\SignalHound\\CalFiles\\");
```

Return Values

This function always returns saNoError. Providing a bad directory path will result in undefined behavior.

saGetSerialNumber

Retrieve the serial number of a specified device

```
saStatus saGetSerialNumber(int device, int *serial);
```

Parameters

device	Device handle.
serial	Pointer to a 32-bit integer which will be assigned the serial number of the device specified.

Description

This function may be called only after the device has been opened. The serial number returned should match the number on the case.

Return Values

saNoError	The function returned successfully.
saDeviceNotOpenErr	The device specified is not open.
saNullPtrErr	The parameter *serial* is NULL.

saGetFirmwareString

Get a firmware version string of a device

```
saStatus saGetFirmwareString(int device, char firmwareString[16]);
```

Parameters

device	Device handle.
firmwareString	Pointer to a char array. The array should be at minimum 16 chars in length.

Description

Use this function to determine the firmware version of a specified device.

Return Values

saNoError	The function returned successfully.
saDeviceNotOpenErr	The device specified is not open.
saNullPtrErr	The parameter *firmwareString* is NULL.

saGetDeviceType

Retrieve the model type of a specified device

```
saStatus saGetDeviceType(int device, saDeviceType *type);
```

Parameters

device	Device handle.
type	Pointer to an integer to receive the model type.

Description

This function may be called only after the device has been opened. If the device handle is valid, *type* will contain the model type of the device pointed to by *handle.*

type is an enumerated value of type *saDeviceType*. *saDeviceType* is defined in sa_api.h.

Return Values

saNoError	The function returned successfully.
saDeviceNotOpenErr	The device specified is not open.
saNullPtrErr	The parameter *type* is NULL.

saConfigAcquisition

Change the detector type and choose between linear or log scaled returned sweeps

```
saStatus saConfigAcquisition(int device, int detector, int scale);
```

Parameters

device	Device handle.
detector	Specifies the video detector. The two possible values for *detector* are SA_MIN_MAX and SA_AVERAGE.
scale	Specifies the scale in which sweep results are returned int. The four possible values for *scale* are SA_LOG_SCALE, SA_LIN_SCALE, SA_LOG_FULL_SCALE, and SA_LIN_FULL_SCALE.

Description

detector specifies how to produce the results of the signal processing for the final sweep. Depending on settings, potentially many overlapping FFTs will be performed on the input time domain data to retrieve a more consistent and accurate final result. When the results overlap *detector* chooses whether to average the results together, or maintain the minimum and maximum values. If averaging is chosen, the *min* and *max* sweep arrays will contain the same averaged data.

The *scale* parameter will change the units of returned sweeps. If SA_LOG_SCALE is provided sweeps will be returned in amplitude unit dBm. If SA_LIN_SCALE is return, the returned units will be in millivolts. If the full scale units are specified, no corrections are applied to the data and amplitudes are taken directly from the full scale input.

Return Values

saNoError	The function returned successfully.
saDeviceNotOpenErr	The device specified is not open.
saInvalidDetectorErr	The *detector* value provided does not match the list of accepted values.
saInvalidScaleErr	The *scale* provided does not match the list of accepted values.

saConfigCenterSpan

Change the center and span frequencies

```
saStatus saConfigCenterSpan(int device, double center, double span);
```

Parameters

device	Device handle.
center	Center frequency in hertz.
span	Span in hertz.

Description

This function configures the operating frequency band of the device. Start and stop frequencies can be determined from the center and span.

- start = center – (span / 2)
- stop = center + (span / 2)

The values provided are used by the device during initialization and a more precise start frequency is returned after initiation. Refer to `saQueryTraceInfo()` for more information.

Each device has a specified operational frequency range between some minimum and maximum frequency. The limits are defined in sa_api.h. The *center* and *span* provided cannot specify a sweep outside of this range.

Certain modes of operation have specific frequency range limits. Those mode dependent limits are tested against during `saInitiate()` and not here.

Return Values

saNoError	The function returned successfully.
saDeviceNotOpenErr	The device specified is not open.
saFrequencyRangeErr	The calculated start or stop frequencies fall outside of the operational frequency range of the specified device. This error will also be returned when span is smaller than 1Hz.

saConfigLevel

Change the reference level of the device

```
saStatus saConfigLevel(int device, double ref);
```

Parameters

device	Device handle.
ref	Reference level in dBm.

Description

This function is best utilized when the device attenuation and gain is set to automatic(default). When both attenuation and gain are set to AUTO, the API uses the reference level to best choose the gain and attenuation for maximum dynamic range. The API chooses attenuation and gain values best for analyzing signal at or below the reference level. For this reason, to achieve the best results, ensure gain and attenuation are set to AUTO and your reference level is set at or slightly about your expected input power for best sensitivity. Reference level is specified in dBm units.

Return Values

saNoError	The function returned successfully.
saDeviceNotOpenErr	The device specified is not open.
saParameterClamped	The provided reference level exceeded and was clamped to 20 dBm.

saConfigGainAtten

Change the RF/IF gain and attenuation of the device

```
saStatus saConfigGain(int device, int atten, int gain, bool preamp);
```

Parameters

device	Device handle.
atten	Attenuator setting.
gain	Gain setting.
preamp	Specify whether to enable the internal device pre-amplifier.

Description

To set attenuation or gain to automatic, pass SA_AUTO_GAIN and SA_AUTO_ATTEN as parameters. The *preamp* parameter is ignored when gain and attenuation are automatic and is chosen automatically.

atten paremeter

Supplied Parameter	SA44 Attenuation	SA124 Attenuation
SA_AUTO_ATTEN (-1)	Auto	Auto
0	0 dB	0 dB
1	5 dB	10 dB
2	10 dB	20 dB
3	15 dB	30 dB

gain paremeter

Supplied Parameter	Gain
SA_AUTO_GAIN (-1)	Auto
0	16 dB Attenuation
1	Mid-Range
2	12 dB Digital Gain

By default, if this function is not called, gain and attenuation are set to automatic. It is suggested to leave these values as automatic as it will greatly increase the consistency of your results. If you choose to manually control gain and attenuation please read Appendix: Setting Gain and Attenuation.

Return Values

saNoError	The function returned successfully.
saDeviceNotOpenErr	The device specified is not open.
saParameterClamped	The provided gain or attenuation values were clamped to normal operating ranges.

saConfigSweepCoupling

Configure sweep processing characteristics

```
saStatus saConfigSweepCoupling(int device, double rbw, double vbw, bool reject);
```

Parameters

device	Handle to the device being configured.

rbw	Resolution bandwidth in Hz. RBW can be arbitrary.
vbw	Video bandwidth in Hz. VBW must be less than or equal to RBW. VBW can be arbitrary. For best performance use RBW as the VBW.
reject	Indicates whether to enable image rejection.

Description

The resolution bandwidth, or RBW, represents the bandwidth of spectral energy represented in each frequency bin. For example, with an RBW of 10 kHz, the amplitude value for each bin would represent the total energy from 5 kHz below to 5 kHz above the bin's center.

The video bandwidth, or VBW, is applied after the signal has been converted to frequency domain as power, voltage, or log units. It is implemented as a simple rectangular window, averaging the amplitude readings for each frequency bin over several overlapping FFTs. A signal whose amplitude is modulated at a much higher frequency than the VBW will be shown as an average, whereas amplitude modulation at a lower frequency will be shown as a minimum and maximum value.

Available RBWs are [0.1Hz – 100kHz] and 250kHz. For the SA124 devices, a 6MHz RBW is available as well. Not all RBWs will be available depending on span, for example the API may restrict RBW when a sweep size exceeds a certain amount. Also there are many hardware limitations that restrict certain RBWs, for a full list of these restrictions, see the Appendix:Setting RBW and VBW.

The parameter **reject** determines whether software image reject will be performed. The SA-series spectrum analyzers do not have hardware-based image rejection, instead relying on a software algorithm to reject image responses. See the USB-SA44B or USB-SA124B manuals for additional details. Generally, set **reject** to true for continuous signals, and false to catch short duration signals at a known frequency. To capture short duration signals with an unknown frequency, consider the Signal Hound BB60C.

Return Values

saNoError	The function returned successfully.
saDeviceNotOpenErr	The device specified is not open.
saBandwidthErr	*rbw* or *vbw* falls outside possible input range.
	vbw is greater than resolution bandwidth.

saConfigRBWShape

Specify the RBW filter shape

```
saStatus saConfigRBWShape(int device, int rbwShape);
```

Parameters

device	Device handle.
rbwShape	An acceptable RBW filter shape value, either SA_RBW_SHAPE_FLATTOP or SA_RBW_SHAPE_CISPR.

Description

Specify the RBW filter shape, which is achieved by changing the window function. When specifying SA_RBW_SHAPE_FLATTOP, a custom bandwidth flat-top window is used measured at the 3dB cutoff point. When specifying SA_RBW_SHAPE_CISPR, a Gaussian window with zero-padding is used to achieve the specified RBW. The Gaussian window is measured at the 6dB cutoff point.

Return Values

saNoError	The function returned successfully.
saInvalidParameterErr	The *rbwShape* parameter does not match an acceptable input value.

saConfigProcUnits

Configure video processing unit type

```
saStatus saConfigProcUnits(int device, int units);
```

Parameters

device	Device handle.
units	The possible values are `SA_POWER_UNITS`, `SA_LOG_UNITS`, `SA_VOLT_UNITS`, and `SA_BYPASS`.

Description

The *units* provided determines what unit type video processing occurs in. The chart below shows which unit types are used for each *units* selection.

For "average power" measurements, `SA_POWER_UNITS` should be selected. For cleaning up an amplitude modulated signal, `SA_VOLT_UNITS` would be a good choice. To emulate a traditional spectrum analyzer, select `SA_LOG_UNITS`. To minimize processing power and bypass video bandwidth processing, select `SA_BYPASS`.

SA_LOG_UNITS	dBm
SA_VOLT_UNITS	mV
SA_POWER_UNITS	mW
SA_BYPASS	No video processing

Return Values

saNoError	The function returned successfully.
saDeviceNotOpen	The device specified is not open.
saInvalidParameterErr	The value for *units* did not match any known value.

saConfigureIQ

Configure the digital IQ data stream

```
saStatus saConfigIQ(int device, int decimation, double bandwidth);
```

Parameters

device	Device handle.
decimation	Specify a decimation rate for the IQ data stream.
bandwidth	Specify the band pass filter width on the IQ digital stream.

Description

This function is used to configure the digital IQ data stream. A decimation factor and filter bandwidth are able to be specified. The decimation rate divides the IQ sample rate directly while the *bandwidth* parameter further filters the digital stream.

For any given decimation rate, a minimum filter bandwidth must be applied to account for sufficient filter roll off. If a bandwidth value is supplied above the maximum for a given decimation, the bandwidth will be clamped to the maximum value. For a list of possible decimation values and associated bandwidth values, see the table below.

The base sample rate of the SA44 and SA124 spectrum analyzers is 486.111111 (repeating) kS/s. To get a precise sample rate given a decimation value, use this equation.

$$Sample\ rate = \frac{486111.11111\sim}{decimation}$$

Decimation Rate	Maximum Bandwidth
1	250.0 kHz
2	225.0 kHz
4	100.0 kHz
8	50.0 kHz
16	20 kHz
32	12.0 kHz
64	5.0 kHz
128	3.0 kHz

Return Values

saNoError	The function returned successfully.
sDeviceNotOpenErr	The device specified is not open.
saInvalidParameterErr	The decimation rate is outside the acceptable input range. The decimation rate is not a power of two.

saConfigAudio

Configure audio demodulation settings

```
saStatus saConfigAudio(int device, int audioType, double centerfreq, double
bandwidth, double audioLowPassFreq, double audioHighPassFreq, double fmDeemphasis);
```

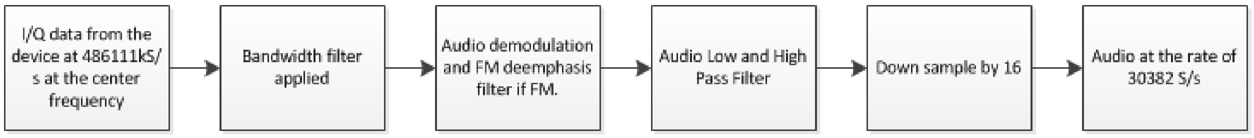
Parameters

device	Handle to the device being configured.
audioType	Specifies the demodulation scheme, possible values are SA_AUDIO_AM, SA_AUDIO_FM, SA_AUDIO_USB, SA_AUDIO_LSB, and SA_AUDIO_CW.
centerFreq	Center frequency in Hz of audio signal to demodulate.
bandwidth	Intermediate frequency bandwidth centered on freq. Filter takes place before demodulation. Specified in Hz. Should be between 500Hz and 500kHz.
audioLowPassFreq	Post demodulation filter in Hz. Should be between 1kHz and 12kHz.
audioHighPassFreq	Post demodulation filter in Hz. Should be between 20 and 1000Hz.
fmDeemphasis	Specified in micro-seconds. Should be between 1 and 100. This value is ignored if audioType is not equal to SA_AUDIO_FM.

Description

This function is used to configure the majority of the audio stream settings. A number of audio modulation types are supported, and a number of filter parameters can be set.

Below is the overall flow of data through our audio processing algorithm.


I/Q data from the device at 486111kS/s at the center frequency
Bandwidth filter applied
Audio demodulation and FM deemphasis filter if FM.
Audio Low and High Pass Filter
Down sample by 16
Audio at the rate of 30382 S/s

Return Values

saNoError	The function returned successfully.
saDeviceNotOpenErr	The device specified is not open.
saInvalidParameterErr	One or more parameters did not match known accepted values.

saConfigRealTime

Configure the real-time frame parameters

```
saStatus saConfigRealTime(int device, double frameScale, int frameRate);
```

Parameters

device	Device handle.
frameScale	Specify the real-time frame height in dB. Values can be between [10 - 200].
frameRate	Specify the real-time frame rate in frames per seconds. Values can be between [4 – 30]

Description

The function allows you to configure additional parameters of the real-time frames returned from the API. If this function is not called a scale of 100dB is used and a frame rate of 30fps is used. For more information regarding real-time mode see Modes of Operation : Real-Time Analysis.

Return Values

saNoError	The function returned successfully.
saDeviceNotOpenErr	The device specified is not open.
saParameterClamped	One of the parameters supplied was outside the range of acceptable inputs and was clamped to the range.

saConfigRealTimeOverlap

Configure the real-time processing overlap rate

```
saStatus saConfigRealTimeOverlap(int device, double advanceRate);
```

Parameters

advanceRate	FFT advance rate. See description.

Description

By setting the advance rate users can control the overlap rate of the FFT processing in real-time spectrum analysis. The *advanceRate* parameter specifies how far the FFT window slides through the data for each FFT as a function of FFT size. An *advanceRate* of 0.5 specifies that the FFT window will advance 50% the FFT length for each FFT for a 50% overlap rate. Specifying a value of 1.0 would mean the FFT window advances the full FFT length meaning there is no overlap in real-time processing. The default value is 0.125 and the range of acceptable values are between [0.125, 10]. Increasing the advance rate reduces processing considerably but also increases the 100% probability of intercept of the device.

Return Values

saParameterClamped	Value provided outside the range of acceptable inputs. A value within the acceptable range was used.

saSetTimebase

Set the timebase reference state of the device

```
saStatus saSetTimebase(int device, int timebase);
```

Parameters

device	Device handle.

timebase	Time base setting value. Acceptable inputs are SA_REF_INTERNAL_OUT and SA_REF_EXTERNAL_IN.

Description

Configure the time base reference port for the device. By passing a value of SA_REF_INTERNAL_OUT you can output the internal 10MHz time base of the device out on the reference port. By passing a value of SA_REF_EXTERNAL_IN the API attempts to enable a 10MHz reference on the reference BNC port. If no reference is found, the device continues to use the internal reference clock. Once a device has successfully switched to an external reference it must remain using it until the device is closed, and it is undefined behavior to disconnect the reference input from the reference BNC port.

Return Values

saNoError	The function returned successfully. This status will also be returned if the reference port has been previously successfully configured.
saInvalidDeviceErr	The setting provided is not supported on the device specified. See the markings next to the timebase port to determine supported functionality.
saExternalReferenceNotFound	Unable to find an external reference on the reference BNC port. The device will continue using the internal reference after this function is returned.

saInitiate

Change the operating state of the device

```
saStatus saInitiate(int device, int mode, unsigned int flag);
```

Parameters

device	Device handle.
mode	The possible values for *mode* are SA_IDLE, SA_SWEEPING, SA_REAL_TIME, SA_IQ, SA_AUDIO, and SA_TG_SWEEP.
flag	This value is currently unused. Pass 0 as a parameter.

Description

This function configures the device into a state determined by the *mode* parameter. For more information regarding operating states, refer to the Theory of Operation and Modes of Operation sections. This function calls `saAbort()` before attempting to reconfigure. It should be noted, if an error occurs attempting to configure the device, any past operating state will no longer be active and the device will become idle.

Return Values

saNoError	The device returned successfully.
saDeviceNotOpenErr	The device specified is not open.

saInvalidParameterErr	The value for *mode* did not match any known value. In real-time mode, this value may be returned if the span limits defined in the API header are broken. Also in real-time mode, this error will be returned if the resolution bandwidth is outside the limits defined in the API header.
saParameterClamp	One or more configuration values were clamped in order to configure the device into the *mode* specified.
saBandwidthClamped	The resolution or video bandwidth was limited based on supplied span and/or hardware limitations. The device limited the bandwidth and continued operation. For a full list of bandwidth restrictions, please see Appendix: Setting RBW and VBW.

saAbort

Stop the current mode of operation

```
saStatus saAbort(int device);
```

Parameters

device	Device handle.

Description

Stops the device operation and places the device into an idle state. If the device is currently idle, then the function returns normally and returns *saNoError.*

Return Values

saNoError	The function returned successfully.
saDeviceNotOpenErr	The device specified is not open.

saQuerySweepInfo

Returns values needed to query and analyze traces

```
saStatus saQuerySweepInfo(int device, int *sweepLength, double *startFreq, double
*binSize);
```

Parameters

device	Device handle.
sweepLength	A pointer to a 32-bit integer. If the function returns successfully, the integer *sweepLength* points to will contain the size of arrays returned by the fetch trace functions.
startFreq	A pointer to a 64-bit floating point variable. If the function returns successfully, the variable *startFreq* points to will equal the frequency of the first bin in the configured sweep.

binSize	A pointer to a 64-bit floating point variable. If the function returns successfully, the variable *start* points to will contain the frequency difference between each bin in the configured sweep.

Description

This function should be called to determine sweep characteristics after a device has been configured and initiated.

Return Values

saNoError	The function returned successfully.
saDeviceNotOpenErr	The device specified is not open.
saNullPtrErr	Indicates one or more pointer parameters were equal to NULL.
saNotConfiguredErr	The device is not configured for sweeps.

saQueryStreamInfo

Retrieve values needed to query and analyze an IQ data stream

```
saQueryStreamInfo(int device, int *returnLen, double *bandwidth, int
*samplesPerSecond);
```

Parameters

device	Device handle.
returnLen	Pointer to a 32-bit integer. If the function returns successfully, the variable *returnLen* points to will contain the number of IQ sample pairs which will be returned by calling `saGetIQ()`.
bandwidth	Pointer to a 64-bit float. If the function returns successfully, the variable *bandwidth* points to will contain the bandpass filter bandwidth width in Hz. Width is specified by the 3dB roll-off points.
samplesPerSecond	Pointer to a 32-bit integer. If the function returns successfully, the variable *samplesPerSecond* points to will contain the sample rate of the configured IQ data stream.

Description

Use this function to get the parameters of the IQ data stream.

Return Values

saNoError	The function returned successfully.
saDeviceNotOpenErr	The device specified is not open.
saNotConfiguredErr	The device specified is not configured for IQ streaming.

saQueryRealTimeFrameInfo

Query the frame size of the real-time frame

```
saStatus saQueryRealTimeFrameInfo(int device, int *frameWidth, int *frameHeight);
```

Parameters

device	Handle of an initialized device.
frameWidth	Pointer to a 32-bit signed integer.
frameHeight	Pointer to a 32-bit signed integer.

Description

This function should be called after initializing the device for Real-Time mode. This device returns the frame size of the real-time frame configured.

Return Values

saNoError	The function returned successfully.
saDeviceNotOpenErr	The device specified is not open.
saNullPtrErr	A pointer parameter supplied is equal to NULL.
saNotConfiguredErr	The device specified is not configured for real-time mode.

saQueryRealTimePoi

Get the configured 100% probability of intercept of the device configured for real-time spectrum analysis

```
saStatus saQueryRealTimePoi(int device, double *poi);
```

Parameters

poi	Pointer to double. See description.

Description

When this function returns successfully, the value *poi* points to will contain the 100% probability of intercept duration in seconds of the device as currently configured in real-time spectrum analysis. The device must actively be configured and initialized in the real-time spectrum analysis mode.

Return Values

saNullPtrErr	*poi* parameter is NULL.
saNotConfiguredErr	The device specified is not initialized for real-time spectrum analysis.

saGetSweep

Get one full sweep from a configured and initiated device

```
saStatus saGetSweep_32f(int device, float *min, float *max);
saStatus saGetSweep_64f(int device, double *min, double *max);
```

Parameters

device	Device handle.
min	Pointer to the beginning of an array of floating point values, whose length is equal to or greater than *sweepLength* returned from `saQuerySweepInfo()`.
max	Pointer to the beginning of an array of floating point values, whose length is equal to or greater than *sweepLength* returned from `saQuerySweepInfo()`.

Description

Upon returning successfully, this function returns the minimum and maximum arrays of one full sweep. If the *detector* provided in `saConfigAcquisition()` is SA_AVERAGE, the arrays will be populated with the same values. Element zero of each array corresponds to the *startFreq* returned from `saQuerySweepInfo()`.

Return Values

saNoError	The function returned successfully.
saNullPtrErr	Indicates that one or both pointer parameters was NULL.
saDeviceNotOpenErr	The device specified is not open.
saNotConfiguredErr	Indicates the device is idle.
saInvalidModeErr	Indicates the device is not configured in one of the sweep mode.
saCompressionWarning	This warning is returned when the ADC detects clipping of the input signal. This occurs when the maximum input voltage has been reached. Signal analysis and reconstruction become issues on clipped signals. To prevent this, a combination of increasing attenuation, decreasing gain, or increasing reference level(when gain/atten is automatic) will allow for more headroom.
saUSBCommErr	Device connection issues were present in the acquisition of this sweep. See Error Handling : Device Connection Errors.

saGetPartialSweep

Get a partial sweep from a configured and initiated device

```
saGetPartialSweep_32f(int device, float *min, float *max, int *start, int *stop);
saGetPartialSweep_64f(int device, double *min, double *max, int *start, int *stop);
```

Parameters

device	Device handle.
min	Pointer to the beginning of an array of floating point values, whose length is equal to or greater than *sweepLength* returned from `saQuerySweepInfo()`.

max	Pointer to the beginning of an array of floating point values, whose length is equal to or greater than *sweepLength* returned from `saQuerySweepInfo()`.
start	Pointer to a 32-bit integer variable. If the function returns successfully, the variable start points to will contain the start index of the updated portion of the sweep
stop	Pointer to a 32-bit integer variable. If the function returns successfully, the variable stop points to will contain the index+1 of the last update value in the updated portion of the sweep.

Description

This function is similar to the `saGetSweep` functions except it can return in certain contexts before the full sweep is ready. This function might return for sweeps which are going to take multiple seconds, providing you with only a portion of the results. This might be useful if you want to perform some signal analysis on portions of the sweep as it is being received, or update other portions of your application during a long acquisition process. Subsequent calls will always provide the next contiguous portion of spectrum.

A buffer that can hold the full sweep must be provided. The pointers start and stop are used to determine which portion of the sweep was updated. The elements in the arrays from [start, stop] will be updated. start and [stop-1] can be used to index the updated portion of the arrays.

The updated portion of the sweep will always at maximum end at the final element of the sweep. For example, if only 20 frequency bins remain after the previous call to saGetPartialSweep, the next call will update at most 20 points. When the final portion of the sweep has been updated, stop will equal the sweep length. Calling this function again will request and begin the next sweep.

Return Values

See saGetSweep() for a list of return values.

saGetRealTimeFrame

Retrieve one real-time sweep and frame

```
saStatus saGetRealTimeFrame(int device, float *sweep, float *frame);
```

Parameters

device	Handle to an initialized device configured in real-time mode.
sweep	Pointer to a floating point array. If the function returns successfully, the contents of the array will be a frequency sweep.
frame	Pointer to a floating point array. If the function returns successfully, the contents of the array will contain a single real-time frame.

Description

This function is used to retrieve one real-time frame and sweep. This function should be used instead of *saGetSweep* and *saGetPartialSweep* for real-time mode. The sweep array should be 'N' number of value long, where N is the sweep length returned from *saQuerySweepInfo.* The frame should be WxH values long where W and H are the values returned from *saQueryRealTimeFrameInfo.* For more information see Modes of Operation : Real-Time Analysis.

Return Values

saNoError	The function returned successfully.
saDeviceNotOpenErr	The device specified is not open.
saNullPtrErr	One or more of the supplied pointers are NULL.
saNotConfiguredError	The device specified is currently idle.
saInvalidModeErr	The device specified is not currently configured for real-time mode.
saUSBCommErr	Device connection issues were present in the acquisition of this sweep. See Error Handling : Device Connection Errors.
saCompressionWarning	This warning is returned when the ADC detects clipping of the input signal. This occurs when the maximum input voltage has been reached. Signal analysis and reconstruction become issues on clipped signals. To prevent this, a combination of increasing attenuation, decreasing gain, or increasing reference level(when gain/atten is automatic) will allow for more headroom.

saGetIQ

This function has been deprecated. See saGetIQData() for a replacement.

Retrieve raw data from a streaming device

```
saStatus saGetIQ_32f(int device, float *iq);
saStatus saGetIQ_64f(int device, double *iq);
```

Parameters

device	Handle of a streaming device.
iq	A pointer to a floating point array. The contents of this buffer will be updated with interleaved IQ digital samples.

Description

Retrieve the next array of IQ samples in the stream. The length of the buffer provided to this function is the return length from `saQueryStreamInfo()` * 2. `saQueryStreamInfo ()` returns the length as IQ sample pairs. This function will need to be called ~30 times per second for any given decimation rate for the internal circular buffers not to fall behind. We recommend polling this function from a separate thread and not performing any other tasks on the polling thread to ensure the thread does not fall behind.

The buffer will be populated with alternating IQ sample pairs scaled to mW. The time difference between each sample can be determined from the sample rate of the configured device.

Return Values

saNoError	The function returned successfully.

saDeviceNotOpenErr	The device specified is not open.
saNullPtrErr	One or more supplied parameters is NULL.
saNotConfigureErr	The device specified is not configured.
saInvalidModeErr	The device specified is not configured for IQ streaming.
saCompressionWarning	This warning is returned when the ADC detects clipping of the input signal. This occurs when the maximum voltage has been reached. Signal analysis and reconstruction become issues on clipped signals. To prevent this, increase the reference level.
saUSBCommErr	Device connection issues were present in the acquisition of data. See Error Handling : Device Connection Errors.

saGetIQData

Retrieve one block of IQ data specified by the IQ packet struct

```
saStatus saGetIQData(int device, saIQPacket *pkt);
```

Parameters

device	Handle to the device.
pkt	Pointer to a saIQPacket struct.

Description

This function retrieves one block of IQ data as specified by the saIQPacket struct. The members of the saIQPacket struct are

- **iqData –** Set by user, Pointer to an array of 32-bit complex float point values. Complex values are interleaved real-imaginary pairs. This must point to a contiguous block of iqCount complex values.
- **iqCount –** Set by user, specify the number of IQ data pairs to return.
- **purge –** Set by user, specify whether to discard any samples acquired by the API since the last time the saGetIQData function was called. Set to SA_TRUE if you wish to discard all previously acquired data, and SA_FALSE if you wish to retrieve the contiguous IQ values from a previous call to this function.
- **dataRemaining –** Set by API, how much data is still left buffered in the API.
- **sampleLoss –** Set by API, returns SA_TRUE or SA_FALSE for whether the API had to drop data due to the internal circular buffer filling.
- **sec –** Set by API, the seconds since epoch representing the timestamp of the first sample in the returned array.
- **milli –** Set by API, the milliseconds representing the timestamp of the first sample in the returned array.

The timestamps returned are set by the system clock and should not be used to make absolute measurements, only relative ones. For large contiguous IQ captures, one should only use the first timestamp collected and use the index and sample rate to determine the relative time of an individual sample.

Return Values

saNoError	The function returned successfully.
saDeviceNotOpenErr	The device specified is not open.
saNullPtrErr	One or more supplied parameters is NULL.
saNotConfigureErr	The device specified is not configured.
saInvalidModeErr	The device specified is not configured for IQ streaming.
saCompressionWarning	This warning is returned when the ADC detects clipping of the input signal. This occurs when the maximum voltage has been reached. Signal analysis and reconstruction become issues on clipped signals. To prevent this, increase the reference level.
saUSBCommErr	Device connection issues were present in the acquisition of data. See Error Handling : Device Connection Errors.

saGetAudio

Retrieve 4096 audio samples

```
saStatus saGetAudio(int device, float *audio);
```

Parameters

device	Device handle.
audio	Pointer to a 32-bit floating point value array.

Description

If the device is initiated and running in the audio demodulation mode, the function is a blocking call which returns the next 4096 audio samples. The approximate blocking time for this function is 128 ms if called again immediately after returning. There is no internal buffering of audio, meaning the audio will be overwritten if this function is not called in a timely fashion. The audio values are typically -1.0 to 1.0, representing full-scale audio. In FM mode, the audio values will scale with a change in IF bandwidth.

Return Values

saNoError	The function returned successfully.
saDeviceNotOpenErr	The device specified is not open.
saNotConfiguredErr	The device is not configured for audio. Ensure the device has been configured and `saInitiate()` has been called with the SA_AUDIO mode flag.
saNullPtrErr	The *audio* pointer is NULL.
saCompressionWarning	This warning is returned when the ADC detects clipping of the input signal. This occurs when the maximum voltage has been reached. Signal analysis and reconstruction become issues on clipped signals. To prevent this, increase the reference level.

saUSBCommErr	Device connection issues were present in the acquisition of data. See [Error Handling : Device Connection Errors.]()

saQueryTemperature

Retrieve the current internal temperature of the device

```
saStatus saQueryTemperature(int device, float *temp);
```

Parameters

device	Device handle.
temp	Pointer to a 32-bit floating point value. If the function returns successfully, the value *temp* points to will contain the current device temperature.

Description

Requesting the internal temperature of the device cannot be performed while the device is currently active. To receive the absolute current internal device temperature, ensure the device is inactive by calling *saAbort* before calling this function. If the device is active, the temperature returned will be the last temperature returned from this function.

Return Values

saNoError	The function returned successfully.
saDeviceNotOpenErr	The device specified is not open.
saNullPtrErr	The pointer *temp* does not point to a valid memory location.

saQueryDiagnostics

Retrieve the current internal device characteristics

```
saStatus saQueryDiagnostics(int device, float *voltage);
```

Parameters

device	Device handle.
voltage	Pointer to a 32-bit floating point value. If the function returns successfully the variable voltage points to will contain the current voltage of the system.

Description

A USB voltage below 4.55V may cause readings to be out of spec. Check your cable for damage and USB connectors for damage or oxidation.

Return Values

saNoError	The function returned successfully.
saDeviceNotOpenErr	The device specified is not open.
saNullPtrErr	The parameter *voltage* is NULL.

saDeviceNotIdleErr	The device is currently active and the voltage cannot be queried. Call `saAbort()` before trying again.

saAttachTg

Pairs an open spectrum analyzer with a tracking generator

```
saStatus saAttachTg(int device);
```

Parameters

device	Device handle.

Description

This function attempts to pair an unclaimed Signal Hound tracking generator with an open Signal Hound spectrum analyzer.

Return Values

saNoError	The function returned successfully.
saDeviceNotOpenErr	The device specified is not open.
saTrackingGeneratorNotFound	An unclaimed tracking generator was not discovered.

saIsTgAttached

Returns whether a tracking generator has been paired with a spectrum analyzer

```
saStatus saIsTgAttached(int device, bool *attached);
```

Parameters

device	Device handle.
attached	Pointer to a boolean variable. If this function returns successfully, the variable attached points to will contain a true/false value as to whether a tracking generator is paired with the spectrum analyzer.

Description

This function is a helper function to determine if a tracking generator has been previously paired with the specified device.

Return Values

saNoError	The function returned successfully.
saDeviceNotOpenErr	The device specified is not open.
saNullPtrErr	The *attached* parameter is NULL.

saConfigTgSweep

Configure a tracking generator sweep

```
saStatus saConfigTgSweep(int device, int sweepSize, bool highDynamicRange, bool
passiveDevice);
```

Parameters

device	Device handle.
sweepSize	Suggested sweep size.
highDynamicRange	Request the ability to perform two store throughs for an increased dynamic range sweep.
passiveDevice	Specify whether the device under test is a passive device (no gain).

Description

This function configures the tracking generator sweeps. Through this function you can request a sweep size. The sweep size is the number of discrete points returned in the sweep over the configured span. The final value chosen by the API can be different than the requested size by a factor of 2 at most. The dynamic range of the sweep is determined by the choice of *highDynamicRange* and *passiveDevice.* A value of *true* for both provides the highest dynamic range sweeps. Choosing *false* for *passiveDevice* suggests to the API that the device under test is an active device (amplification).

Return Values

saNoError	The function returned successfully.
saDeviceNotOpenErr	The device specified is not open.
saParameterClamped	The parameter *sweepSize* was limited to the range of 10-1000.

saStoreTgThru

Perform a store thru

```
saStatus saStoreTgThru(int device, int flag);
```

Parameters

device	Device handle.
flag	Specify the type of store thru. Possible values are TG_THRU_0DB and TG_THRU_20DB.

Description

This function, with flag set to TG_THRU_0DB, notifies the API to use the last trace as a thru (your 0 dB reference). Connect your tracking generator RF output to your spectrum analyzer RF input. This can be accomplished using the included SMA to SMA adapter, or anything else you want the software to establish as the 0 dB reference (e.g. the 0 dB setting on a step attenuator, or a 20 dB attenuator you will be including in your amplifier test setup).

After you have established your 0 dB reference, a second step may be performed to improve the accuracy below -40 dB. With approximately 20-30 dB of insertion loss between the spectrum analyzer and tracking generator, call saStoreTgThru with flag set to TG_THRU_20DB. This corrects for slight variations between the high gain and low gain sweeps.

Return Values

saNoError	The function returned successfully.
saDeviceNotOpenErr	The device specified is not open.
saInvalidParameterErr	The *flag* parameter does not match any accepted value.
saNotConfiguredErr	The device is not configured for tracking generator sweeps.

saSetTg

Set the frequency and amplitude output of a paired tracking generator

```
saStatus saSetTg(int device, double frequency, double amplitude);
```

Parameters

device	Device handle.
frequency	Set the frequency, in Hz, of the TG output
amplitude	Set the amplitude, in dBm, of the TG output

Description

This function sets the output frequency and amplitude of the tracking generator. This can only be performed is a tracking generator is paired with a spectrum analyzer and is currently not configured and initiated for TG sweeps.

Return Values

saNoError	The function returned successfully.
saDeviceNotOpenErr	The device specified is not open.
saTrackingGeneratorNotFound	A tracking generator was not found to be paired with the device specified.
saNotConfiguredErr	The API is currently configured and initiated for tracking generator sweeps and the tracking generator cannot be controlled at this time.

saSetTgReference

Configure the timebase for the TG44 and TG124

```
saStatus saSetTgReference(int device, int reference);
```

Parameters

reference	A valid time base setting value. Possible values are SA_REF_UNUSED, SA_REF_INTERNAL_OUT, and SA_REF_EXTERNAL_IN.

Description

Configure the time base for the tracking generator attached to the device specified. When SA_REF_UNUSED is specified additional frequency corrections are applied. If using an external reference or you are using the TG time base frequency as the frequency standard in your system, you will want to specify SA_REF_INTERNAL_OUT or SA_REF_EXTERNAL_IN so the additional corrections are not applied.

Return Values

saTrackingGeneratorNotFound	A tracking generator is not attached to the device specified.
saNotConfiguredErr	The tracking generator is actively sweeping and cannot be configured.
saInvalidParameterErr	The *reference* parameter is not one of the acceptable input values, or the *reference* parameter is not supported on the attached tracking generator.

saGetTgFreqAmpl

Retrieve the last set TG configuration

```
saStatus saGetTgFreqAmpl(int device, double *frequency, double *amplitude);
```

Parameters

device	Device handle.
frequency	The double variable that frequency points to will contain the last set frequency of the TG output in Hz.
amplitude	The double variable that amplitude points to will contain the last set amplitude of the TG output in dBm.

Description

Retrieve the last set TG output parameters the user set through the saSetTg function. The setTg function must have been called for this function to return valid values. If the TG was used to perform scalar network analysis at any point, this function will not return valid values until the setTg function is called again.

If a previously set parameter was clamped in the setTg function, this function will return the final clamped value.

If any pointer parameter is null, that value is ignored and not returned.

Return Values

saNoError	The function returned successfully.
saTrackingGeneratorNotFound	No tracking generator has been attached to the SA device.
saNotConfiguredErr	The API is currently configured and initiated for tracking generator sweeps and the tracking generator cannot be controlled at this time.

saConfigIFOutput

Configure the SA124A/B IF output port

```
saStatus saConfigIFOutput(int device, double inputFreq, double outputFreq, int
inputAtten, int outputGain);
```

Parameters

device	Device handle.
inputFreq	The input center frequency on the SMA connector specified in Hz. Must be between 125MHz and 13GHz.
outputFreq	The desired output frequency on the BNC port specified in Hz. Positive for low-side LO injection, negative for high-side. Must be between 34 and 38MHz for the SA124A and between 61 and 65MHz for the SA124B.
inputAtten	Attenuation of the input signal specified in dB. Must be between 0 and 30 dB.
outputGain	Amplification of the output signal specified in dB. Must be between 0 and 60 dB.

Description

The SA124A/B allows a user to configure the device to route the 6 MHz bandwidth intermediate frequency directly to the IF output BNC port. While the IF is routed to the BNC port, the device is incapable of performing sweeps or IQ streaming. There is no image rejection in this mode.

Calling this function while the device is currently active in a different mode will cause the API to abort the current mode of operation and enable the IF output BNC port. To disable the IF output, simply call *saInitiate()* with the new desired configuration.

The local oscillator mixed with the RF must be 138 MHz or higher, so only high side injection is available below 201 MHz.

Return Values

saNoError	The function returned successfully.
saDeviceNotOpenErr	The device specified is not open.
saInvalidDeviceErr	The device specified does not have the IF output functionality. Only the SA124A and SA124B have IF output capabilities.
saParameterClamped	One or more input parameters fell outside the range of accepted values. The offending parameters were clamped to the acceptable range.

saGetAPIVersion

Get an API software version string

```
const char* saGetAPIVersion();
```

Return Values

const char* The returned string is of the form

major.minor.revision

Ascii periods (".") separate positive integers. Major/Minor/Revision are not gauranteed to be a single decimal digit. The string is null terminated. An example string is below ..

['1' | '.' | '2' | '.' | '1' | '1' | '\0'] = "1.2.11"

saGetErrorString

Retrieve an error string given an API status code

```
const char* saGetErrorString(saStatus code);
```

Parameters

code	A saStatus value returned from an API call.

Description

Produce an ASCII string representation of a given status code. Useful for debugging.

Return Values

const char*	A pointer to a non-modifiable null terminated string. The memory should not be freed/deallocated.

Error Handling

All API functions return the type *saStatus*. *saStatus* is an enumerated type representing the success of a given function call. The return values can be found in sa_api.h. There are three types of returned status codes.

1) No error: Represented with value saNoError, equal to zero.
2) Error: interrupts function execution, represented by a negative return value.

3) Warning: does not interrupt function execution, but may leave the system in an undesirable state. Represented as a positive value.

The best way to address issues is to check the return values of the API functions. The API function *saGetErrorString* is provided to retrieve a string representation of any given status code for easy debugging.

Device Connection Errors

The API issues errors when fatal connection issues are present during normal operation of the device. Only one major error is returned due to fatal connections issues, saUSBCommErr. This error can be returned from all major Get() routines. If at any time the API experiences USB communication errors the next Get() routine will return this error.

If you receive this error, your program should call `saCloseDevice()` to free any remaining resources before checking the connection of your device and trying to open the device through the API again.

Appendix

Setting RBW and VBW

The SA44 and SA124 models have many device restrictions which prevent the user from selecting all possible combinations of RBW and VBW for any given sweep. Many restrictions are not known until `saInitiate()` is called and all parameters of the sweep are considered. The API clamps RBW/VBW when `saInitiate()` is called if they break the restrictions which are listed below. Concern yourself with these restrictions only when it is imperative the API use exactly the RBW and VBW you requested.

SA44A/B, SA124A/B limitations:

- Available RBWs in the standard sweep mode
  - 0.1Hz to 100kHz, and 250kHz.
- Available RBWs in real-time
  - 100Hz to 10kHz
- For the SA44A, RBW/VBW must be greater than or equal to 6.5kHz when
  - span is greater than 200kHz
- For the SA44B, SA124A, SA124B, RBW/VBW must be greater than or equal to 6.5kHz when
  - span is greater than or equal to 100MHz
  - span is greater than 200kHz AND start frequency < 16MHz
- For SA124A/B 6MHz RBW available when
  - Start frequency >= 200MHz AND Span >= 200MHz

Setting Gain and Attenuation

When automatic *atten* or automatic *gain* are selected, the API uses the reference level provided to choose the best settings for gain, attenuator, and preamplifier (if applicable) settings for an input signal with amplitude equal to reference level. For almost all cases, automatic settings are best. However, if your application must override the automatic settings, set all gain control values (atten, gain, and preamp) to manual values.

The **atten** parameter controls the RF input attenuator, and is adjustable from 0 to 30 dB in 10 dB steps for the SA124A and SA124B, or 0 to 15 dB in 5 dB steps for the SA44 / SA44B. The RF attenuator is the first gain control device in the front end, before the preamplifier (if any).

The **preamp** parameter, only for the USB-SA44B, controls whether the preamplifier is in circuit or bypassed. The preamplifier increases sensitivity, decreases local oscillator feed-through, and significantly increases the amplitude of intermodulation products. It is located immediately after the attenuator. For the SA124A/B the preamplifier is always on; use a higher attenuation setting for high amplitude signals.

The **gain** parameter controls analog and digital intermediate frequency (IF) gain. A setting of 1 is mid-range. Setting gain to 0 adds a 16 dB attenuator to the IF input. Setting gain to 2 adds 12 dB of digital gain to the IF signal processing chain before the 24 bit ADC values are truncated to 16 bits.

Code Examples

The code examples have been moved to the *examples/* folder found in the API download.

Programming Languages Other Than C++ (and bools)

Even though the API is compiled with Microsofts C++ compiler (VC++), name decoration is explicitly disabled for public functions so that a variety of programming languages and external tools can utilize this API. Programming languages such as C#, Java, and Python can call this API as well as tools such as Matlab and LabView.

Most function parameters are standard primitive data types, such as floating point numbers and integers. Parameters are passed either by value or as a pointer. Supplying these parameters is supported by the languages and tools mentioned above.

One such primitive type without a direct analogous type is the bool type. VC++ defines the bool type as an 8-bit integer. Passing 0 for false and 1 for true in an 8-bit integer type will work when a bool type is needed by the API.

Enums are defined as 32-bit integers in VC++.

Internal Auto-Calibration

Every time saInitiate is called, the API reads the internal temperature of the device and generates corrections accordingly. If you open the device shortly after plugging it in, before it has had 15 minutes to reach ambient temperature, your readings may drift as it warms up. It is recommended to re-initiate your sweep every 2-3 minutes until a stable internal temperature is reached.

ARMv7-A Support

This API has been compiled for the Raspberry Pi 2, using Raspbian kernel version 3.18, release date 2015-05-05, compiled with gcc version 4.6.3. The purpose of this API is to provide a low power, robust platform for remote or battery-powered spectrum monitoring applications.

This branch of the API has several major differences from the Windows API:

1. Only a single connected device is supported. This device must be a USB-SA44B or USB-SA124B.

2. **A factory firmware update is required**. You must contact Signal Hound for an RMA to use this API if you have firmware version 2.10 for the USB-SA44B, or firmware version 3.10 or 3.11 for the USB-SA124B.
3. The USB-SA44, USB-SA124A, and USB-SA44B with serial numbers below 21000000 (firmware versions 2.0x) are not supported.
4. There is no tracking generator support.
5. There is limited video bandwidth support. Video bandwidth will be internally clamped to RBW/2, and is limited to a minimum of 3 kHz when span > 200 kHz.
6. Resolution bandwidths below 3 kHz are available only for spans ≤200 kHz, and may occasionally drop data, triggering an automatic re-sweep. It is recommended to not use RBW below 3 kHz unless it is required for your application.
7. IQ streaming
    a. Only decimation of 1, 2, and 4 are supported.
    b. Decimation of 2 and 4 tested stable and reliable on the Raspberry Pi 2. Decimation of 1 may not provide continuous data, and should be avoided if possible.
    c. IF bandwidth is fixed at 250 kHz / decimation.
8. Real-time mode is not supported.

Directions for installation can be found in the README file included in the ARM SDK.

This API should work on other ARM processors as long as they have USB 2.0 and “hard floats”. Results may vary.