SA API

Generated by Doxygen 1.9.4

Chapter 1

SA44/124 API Reference

This manual is a reference for the Signal Hound SA Series application programming interface (API). This API supports the SA44B, SA44, and SA124B/A Signal Hound devices. The API provides a set of C functions for making measurements with the SA series devices. The API is C ABI compatible making is possible to be interfaced from most programming languages.

1.1 Examples

All code examples are located in the *examples/* folder in the SDK which can be downloaded at `www.signalhound.com/software-development-kit`.

1.2 Measurements

This section covers the main measurements available through the API.

- Swept Spectrum Analysis
- Real-Time Spectrum Analysis
- I/Q Streaming
- Audio Demodulation
- Scalar Network Analysis

Also see Theory of Operation for more information.

1.3 Build/Version Notes

1.3.1 Builds

Windows builds are available for both x86 and x64. They are compiled with Visual Studio 2012 and any application using this library will require distributing the VS2012 redistributable libraries.

1.3.2 Versions

Versions are of the form *major.minor.revision*.

A *major* change signifies a significant change in functionality relating to one or more measurements, or the addition of significant functionality. Function prototypes have likely changed.

A *minor* change signifies additions that may improve existing functionality or fix major bugs but make no changes that might affect existing user's measurements. Function prototypes can change but do not change existing parameters meanings.

A *revision* change signifies minor changes or bug fixes. Function prototypes will not change. Users should be able to update by simply replacing the .DLL.

See the change log in the SDK for detailed API version history.

1.3.2.1 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.

1.4 Requirements

1.4.1 Software Requirements

- Windows 7/8/10/11 (64-bit Windows 10 recommended)
- Windows C/C++ development tools and environment.
  - API was compiled using VS2012 and VS2019.
    - VS2012/VS2019 C++ redistributables are required.
- **sa_api.h**: API header file.
- **sa_api.lib/sa_api.dll**: These are the main API library files.
- **CDM v2.12.00 WHQL Certified.exe**: USB driver from FTDI.
- **ftd2xx.dll**: Must be in the same directory as the sa_api.dll.

1.4.1.1 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.

1.4.2 PC Requirements

- Dual-core Intel processor.
- USB 2.0 connectivity (2x for SA124B).

1.5 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.

1.6 Error Handling

All API functions return the type saStatus. saStatus is an enumerated type representing the success of a given function call.

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.

1.7 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.

1.8 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

1.9 Setting Gain and Attenuation

When automatic atten or automatic gain is 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 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 midrange. 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.

1.10 Code Examples

The code examples are located in the examples/ folder found in the API download.

1.11 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 pointers. Supplying these parameters is supported by the languages and tools mentioned above.

One such primitive type without a directly 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++.

1.12 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.

1.13 Thread Safety

The SA API is not thread safe. A multi-threaded application is free to call the API from any number of threads if the function calls are synchronized (i.e. using a mutex). Not synchronizing your function calls will lead to undefined behavior.

1.14 Multiple Devices and Multiple Processes

The API can manage multiple devices within one process. In each process the API manages a list of open devices to prevent a process from opening a device more than once. You may open multiple devices by specifying the serial number or allowing the API to discover them automatically.

If you wish to use the API in multiple processes it is the user's responsibility to manage a list of devices to prevent the possibility of opening a device twice from two different processes. Two processes communicating to the same device will result in undefined behavior. One possible way to manage inter-process information like this is to use a named mutex on a Windows system.

1.15 Contact Information

For technical support, email `support@signalhound.com`.

For sales, email `sales@signalhound.com`.

Chapter 2

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 I/Q sample rate.
3. Initiate the device for a particular mode of operation, whether it be frequency domain sweeps or I/Q 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.

2.1 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 functions saOpenDevice and saOpenDeviceBySerialNumber provide this functionality. These functions return an integer ID to the device that was opened. Up to SA_MAX_DEVICES 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.

2.1.1 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 Signal Hound 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 that 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 `Spike`, then the calibration files will already exist on the host PC.

2.2 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 I/Q sampling rate. The configurations do not take effect until the device is initiated.

2.3 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.

2.4 Retrieving 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.

2.5 Aborting 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.

2.6 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.

Chapter 3

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 `Spike User Manual`.

Please also see the C++ programming examples for an example of interfacing the device for each measurement type.

3.1 Swept Spectrum 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.

3.1.1 Example

For a list of all examples, please see the *examples/* folder in the SDK.

```
#include "sa_api.h"
#pragma comment(lib, "sa_api.lib")
#include <iostream>
// This example is the minimal code needed to configure the device
// to perform a single sweep.  The device will block for several seconds
// while opening the device, and the call to saGetSweep will block
void simpleSweep1()
{
    int handle = -1;
    saStatus openStatus = saOpenDevice(&handle);
    if(openStatus != saNoError) {
        // Handle unable to open/find device error here
        std::cout « saGetErrorString(openStatus) « std::endl;
        return;
    }
    // Configure the device to sweep a 1MHz span centered on 900MHz
    // Min/Max detector, with RBW/VBW equal to 1kHz
    saConfigCenterSpan(handle, 900.0e6, 1.0e6);
    saConfigAcquisition(handle, SA_MIN_MAX, SA_LOG_SCALE);
    saConfigLevel(handle, -10.0);
    saConfigSweepCoupling(handle, 1.0e3, 1.0e3, true);
    // Initialize the device with the configuration just set
    saStatus initiateStatus = saInitiate(handle, SA_SWEEPING, 0);
    if(initiateStatus != saNoError) {
        // Handle unable to initialize
        std::cout « saGetErrorString(initiateStatus) « std::endl;
```

```
        return;
    }
    // Get sweep characteristics
    int sweepLen;
    double startFreq, binSize;
    saQuerySweepInfo(handle, &sweepLen, &startFreq, &binSize);
    // Allocate memory for the sweep
    float *min = new float[sweepLen];
    float *max = new float[sweepLen];
    // Get 1 or more sweeps with this configuration
    // This function can be called several times and will return a
    //  sweep measured directly after the function is called.
    // The function blocks until the sweep is returned.
    saStatus sweepStatus = saGetSweep_32f(handle, min, max);
    delete [] min;
    delete [] max;
    saAbort(handle);
    saCloseDevice(handle);
}
```

3.1.2 Configuration

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 – Configuring VBW processing.

Once you have configured the device, call saInitiate using the SA_SWEEPING flag.

3.1.3 Usage

This mode is driven by the programmer, causing a sweep to be collected only when the program requests one through the saGetSweep_32f/saGetSweep_64f 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_32f/saGetSweep_64f and saGetPartialSweep_32f/saGetPartialSweep_64f functions.

You can determine the frequency of any bin in the resulting sweep by the following function, where `n` is the zero-based index into the sweep array:

```
Frequency of nth sample point in returned sweep = startFreq + n * binSize
```

3.2 Real-Time Spectrum 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%.

3.2.1 Example

For a list of all examples, please see the *examples/* folder in the SDK.

```
#include "sa_api.h"
#pragma comment(lib, "sa_api.lib")
#include <iostream>
void realTimeSweep()
{
    int handle = -1;
    saStatus openStatus = saOpenDevice(&handle);
    if(openStatus != saNoError) {
        // Handle unable to open/find device error here
        std::cout « saGetErrorString(openStatus) « std::endl;
        return;
    }
    // Configure real-time analysis to be centered on a local
    // FM broadcast frequency, with a 1kHz RBW.
    // Set a frame rate of 30fps, and 100dB height on persistence frames.
    saConfigCenterSpan(handle, 97.1e6, 200.0e3);
    saConfigAcquisition(handle, SA_MIN_MAX, SA_LOG_SCALE);
    saConfigLevel(handle, -10.0);
    saConfigSweepCoupling(handle, 1.0e3, 1.0e3, true);
    saConfigRealTime(handle, 100.0, 30);
    // Initialize the device with the configuration just set
    saStatus initiateStatus = saInitiate(handle, SA_REAL_TIME, 0);
    if(initiateStatus != saNoError) {
        // Unable to initialize
        std::cout « saGetErrorString(initiateStatus) « std::endl;
        return;
    }
    // Get sweep and frame characteristics
    int sweepLen;
    double startFreq, binSize;
    saQuerySweepInfo(handle, &sweepLen, &startFreq, &binSize);
    int frameWidth, frameHeight;
    saQueryRealTimeFrameInfo(handle, &frameWidth, &frameHeight);
    // Allocate memory for the sweep and frame
    float *max = new float[sweepLen];
    float *frame = new float[frameWidth * frameHeight];
    // Get 30 frames and sweeps, representing 1 second of real-time analysis
    int frames = 0;
    while(frames < 30) {
        saStatus sweepStatus = saGetRealTimeFrame(handle, nullptr, max, frame, nullptr);
        frames++;
        // Update your application
    }
    saAbort(handle);
    delete [] max;
    delete [] frame;

    saCloseDevice(handle);
}
```

3.2.2 Configuration

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

Once you have configured the device, call saInitiate using the SA_REAL_TIME flag.

3.2.3 Usage

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 min/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. Both the sweep and density map are returned at rate specified by the function saConfigRealTime.

An alpha frame is also provided by the API. The alphaFrame is the same size as the frame and each index correlates to the same index in the frame. The alphaFrame values represent activity in the frame. When activity occurs in the frame, the index correlating to that activity is set to 1. As time passes and no further activity occurs in that bin, the alphaFrame exponentially decays from 1 to 0. The alphaFrame is useful to determine how recent the activity in the frame is and useful for plotting the frames.

For a full example of using real-time see Real-Time Spectrum Analysis.

3.3 I/Q Streaming

The API can provide programmers with a continuous stream of digital I/Q samples from the device.

3.3.1 Example

For a list of all examples, please see the *examples/* folder in the SDK.

```
#include "sa_api.h"
#pragma comment(lib, "sa_api.lib")
#include <iostream>
// This example demonstrates configuring the device to stream continuous
// IQ data to your application.
void iqStreaming()
{
    int handle = -1;
    saStatus openStatus = saOpenDevice(&handle);
    if(openStatus != saNoError) {
        // Handle unable to open/find device error here
        std::cout « saGetErrorString(openStatus) « std::endl;
        return;
    }
    // Set center freq, span is ignored
    saConfigCenterSpan(handle, 97.1e3, 1.0e3);
    // Set expected input level
    saConfigLevel(handle, -10.0);
    // Configure sample rate and bandwidth
    // Sample rate of 486111.11 / 1 and bandwidth of 250kHz
    saConfigIQ(handle, 1, 250.0e3);
    saInitiate(handle, SA_IQ, 0);
    // Verify the sample rate and bandwidth of the IQ stream
    double bandwidth, sampleRate;
    saQueryStreamInfo(handle, 0, &bandwidth, &sampleRate);
    // How many IQ samples to collect per call
    const int BUF_SIZE = 4096;
    saIQPacket pkt;
    pkt.iqData = new float[BUF_SIZE * 2];
    pkt.iqCount = BUF_SIZE;
    // Setting purge to false ensures each call to getIQPacket()
    // returns contiguous IQ data to the last time the function was called.
    // This also means IQ data must be queried at the rate of the
    // device sample rate.  In this case, the sample rate is 486.111k,
    // so the saGetIQData function must be called
    // 486111 / 4096 = ~118 times per second.
    pkt.purge = false;
    // Retreive about 1 second worth of contiguous IQ data or
    // 120 * 4096 IQ data values.
    int pktCount = 0;
    while(pktCount++ < 120) {
        // Get next contiguous block of IQ data
        saStatus iqStatus = saGetIQData(handle, &pkt);
        std::cout « pkt.sec « " " « pkt.milli « std::endl;
```

```
        // Store/process data before getting another buffer
        // Check any errors or status updates
    }
    // Clean up
    delete [] pkt.iqData;
    saAbort(handle);
    saCloseDevice(handle);
}
```

3.3.2 Configuration

Configuration routines used to prepare streaming are:

- saConfigCenterSpan – Set the center frequency of the I/Q data stream. Span is ignored.
- saConfigLevel – Set the expected input level.
- saConfigGainAtten – See Setting Gain and Attenuation.
- saConfigIQ – Specify the decimation and bandwidth of the I/Q data stream.

Once configured, initialize the device with the SA_IQ mode.

3.3.3 Usage

The digital I/Q stream consists of interleaved 32-bit floating point I/Q pairs scaled to mW. The digital samples are amplitude corrected providing accurate measurements. The I/Q 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 I/Q samples, so the API also provides a configurable bandpass filter to control the overall passband of a given I/Q data stream. The I/Q data stream can also be tuned to an arbitrary center frequency.

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.

3.4 Audio Demodulation

3.4.1 Configuration

Configure audio demodulation with:

- saConfigAudio - Specify the type of demodulation, the center frequency, and the characteristics of the filters.
  - See saConfigAudio to see which types of audio demodulation can be performed.
- saConfigGainAtten - Audio demodulation does not have auto ranging as do other operational modes, so saConfigGainAtten must be called to configure the internal gain, attenuator, and preamplifier.

Once configured, initialize the device with the SA_AUDIO mode.

3.4.2 Usage

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.

3.5 Scalar Network Analysis

When a Signal Hound tracking generator is paired together with a spectrum analyzer, the product 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.

3.5.1 Example

For a list of all examples, please see the *examples/* folder in the SDK.

```
#include "sa_api.h"
#pragma comment(lib, "sa_api.lib")
#include <iostream>
// This example demonstrates how to use the API to perform a single TG sweep.
// See the manual for a full description of each step of the process in the
// Scalar Network Analysis section.
void trackingGeneratorSweep()
{
    int handle = -1;
    saStatus openStatus = saOpenDevice(&handle);
    if(openStatus != saNoError) {
        // Handle unable to open/find device error here
        std::cout « saGetErrorString(openStatus) « std::endl;
        return;
    }
    if(saAttachTg(handle) != saNoError) {
        // Unable to find tracking generator
        return;
    }
    // Sweep some device at 900MHz center with 1MHz span
    saConfigCenterSpan(handle, 900.0e6, 1.0e6);
    saConfigAcquisition(handle, SA_MIN_MAX, SA_LOG_SCALE);
    saConfigLevel(handle, -10.0);
    saConfigSweepCoupling(handle, 1.0e3, 1.0e3, true);
    // Additional configuration routine
    // Configure a 100 point sweep
    // The size of the sweep is a suggestion to the API, it will attempt to
    // get near the requested size.
    // Optimized for high dynamic range and passive devices
    saConfigTgSweep(handle, 100, true, true);
    // Initialize the device with the configuration just set
    if(saInitiate(handle, SA_TG_SWEEP, 0) != saNoError) {
        // Handle unable to initialize
        return;
    }
    // Get sweep characteristics
    int sweepLen;
    double startFreq, binSize;
    saQuerySweepInfo(handle, &sweepLen, &startFreq, &binSize);
    // Allocate memory for the sweep
    float *min = new float[sweepLen];
    float *max = new float[sweepLen];
    // Create test set-up without DUT present
    // Get one sweep
    saGetSweep_32f(handle, min, max);
    // Store baseline
    saStoreTgThru(handle, TG_THRU_0DB);
    // Should pause here, and insert DUT into test set-up
    saGetSweep_32f(handle, min, max);
    // From here, you can sweep several times without needing to restore the thru,
    // once you change your setup, you should reconfigure the device and
    // store the thru again without the DUT inline.
    delete [] min;
    delete [] max;
    saAbort(handle);
    saCloseDevice(handle);
}
```

3.5.2 Configuration and Usage

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_32f/saGetSweep_64f or saGetPartialSweep_32f/saGetPartialSweep_64f. 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 thru must be performed again.

Chapter 4

Data Structure Index

4.1 Data Structures

Here are the data structures with brief descriptions:

Chapter 5

File Index

5.1 File List

Here is a list of all documented files with brief descriptions:

Chapter 6

Data Structure Documentation

6.1 saIQPacket Struct Reference

```
#include <sa_api.h>
```

Data Fields

- float * iqData
- int iqCount
- int purge
- int dataRemaining
- int sampleLoss
- int sec
- int milli

6.1.1 Detailed Description

Used to encapsulate I/Q data and metadata. See saGetIQData.

6.1.2 Field Documentation

6.1.2.1 iqData

```
float* iqData
```

Pointer to an array of 32-bit complex floating-point values. Complex values are interleaved real-imaginary pairs. This must point to a contiguous block of *iqCount* complex pairs.

6.1.2.2 iqCount

```
int iqCount
```

Number of I/Q data pairs to return.

6.1.2.3 purge

```
int purge
```

Specifies whether to discard any samples acquired by the API since the last time and 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 I/Q values from a previous call to this function.

6.1.2.4 dataRemaining

```
int dataRemaining
```

How many I/Q samples are still left buffered in the API. Set by API.

6.1.2.5 sampleLoss

```
int sampleLoss
```

Returns SA_TRUE or SA_FALSE. Will return SA_TRUE when the API is required to drop data due to internal buffers wrapping. This can be caused by I/Q samples not being polled fast enough, or in instances where the processing is not able to keep up (underpowered systems, or other programs utilizing the CPU) Will return SA_TRUE on the capture in which the sample break occurs. Does not indicate which sample the break occurs on. Will always return SA_FALSE if purge is true. Set by API.

6.1.2.6 sec

```
int sec
```

Seconds since epoch representing the timestamp of the first sample in the returned array. Set by API.

6.1.2.7 milli

```
int milli
```

Milliseconds representing the timestamp of the first sample in the returned array. Set by API.

The documentation for this struct was generated from the following file:

- sa_api.h

6.2 saSelfTestResults Struct Reference

```
#include <sa_api.h>
```

Data Fields

- bool highBandMixer
- bool attenuator
- double highBandMixerValue
- double attenuatorValue

6.2.1 Detailed Description

Results of running a self test. See saSelfTest.

6.2.2 Field Documentation

6.2.2.1 highBandMixer

```
bool highBandMixer
```

Band mixers pass/fail results.

6.2.2.2 attenuator

```
bool attenuator
```

Attenuator, second IF, and preamplifier pass/fail results.

6.2.2.3 highBandMixerValue

```
double highBandMixerValue
```

Band mixer readings.

6.2.2.4 attenuatorValue

```
double attenuatorValue
```

Attenuator, second IF, and preamplifier readings.

The documentation for this struct was generated from the following file:

- sa_api.h

Chapter 7

File Documentation

7.1 sa_api.h File Reference

API functions for the SA44/124 spectrum analyzers.

Data Structures

- struct saSelfTestResults
- struct saIQPacket

Macros

- #define SA_TRUE (1)
- #define SA_FALSE (0)
- #define SA_MAX_DEVICES 8
- #define SA44_MIN_FREQ (1.0)
- #define SA124_MIN_FREQ (100.0e3)
- #define SA44_MAX_FREQ (4.4e9)
- #define SA124_MAX_FREQ (13.0e9)
- #define SA_MIN_SPAN (1.0)
- #define SA_MAX_REF (20)
- #define SA_MAX_ATTEN (3)
- #define SA_MAX_GAIN (2)
- #define SA_MIN_RBW (0.1)
- #define SA_MAX_RBW (6.0e6)
- #define SA_MIN_RT_RBW (100.0)
- #define SA_MAX_RT_RBW (10000.0)
- #define SA_MIN_IQ_BANDWIDTH (100.0)
- #define SA_MAX_IQ_DECIMATION (128)
- #define SA_IQ_SAMPLE_RATE (486111.111)
- #define SA_IDLE (-1)
- #define SA_SWEEPING (0x0)
- #define SA_REAL_TIME (0x1)
- #define SA_IQ (0x2)
- #define SA_AUDIO (0x3)
- #define SA_TG_SWEEP (0x4)

- #define SA_RBW_SHAPE_FLATTOP (0x1)
- #define SA_RBW_SHAPE_CISPR (0x2)
- #define SA_MIN_MAX (0x0)
- #define SA_AVERAGE (0x1)
- #define SA_LOG_SCALE (0x0)
- #define SA_LIN_SCALE (0x1)
- #define SA_LOG_FULL_SCALE (0x2)
- #define SA_LIN_FULL_SCALE (0x3)
- #define SA_AUTO_ATTEN (-1)
- #define SA_AUTO_GAIN (-1)
- #define SA_LOG_UNITS (0x0)
- #define SA_VOLT_UNITS (0x1)
- #define SA_POWER_UNITS (0x2)
- #define SA_BYPASS (0x3)
- #define SA_AUDIO_AM (0x0)
- #define SA_AUDIO_FM (0x1)
- #define SA_AUDIO_USB (0x2)
- #define SA_AUDIO_LSB (0x3)
- #define SA_AUDIO_CW (0x4)
- #define TG_THRU_0DB (0x1)
- #define TG_THRU_20DB (0x2)
- #define SA_REF_UNUSED (0)
- #define SA_REF_INTERNAL_OUT (1)
- #define SA_REF_EXTERNAL_IN (2)

Enumerations

- enum saDeviceType {
  saDeviceTypeNone = 0 , saDeviceTypeSA44 = 1 , saDeviceTypeSA44B = 2 , saDeviceTypeSA124A = 3 ,
  saDeviceTypeSA124B = 4 }
- enum saStatus {
  saUnknownErr = -666 , saFrequencyRangeErr = -99 , saInvalidDetectorErr = -95 , saInvalidScaleErr = -94 ,
  saBandwidthErr = -91 , saExternalReferenceNotFound = -89 , saLNAErr = -21 , saOvenColdErr = -20 ,
  saInternetErr = -12 , saUSBCommErr = -11 , saTrackingGeneratorNotFound = -10 , saDeviceNotIdleErr = -9
  ,
  saDeviceNotFoundErr = -8 , saInvalidModeErr = -7 , saNotConfiguredErr = -6 , saTooManyDevicesErr = -5 ,
  saInvalidParameterErr = -4 , saDeviceNotOpenErr = -3 , saInvalidDeviceErr = -2 , saNullPtrErr = -1 ,
  saNoError = 0 , saNoCorrections = 1 , saCompressionWarning = 2 , saParameterClamped = 3 ,
  saBandwidthClamped = 4 , saCalFilePermissions = 5 }

Functions

- SA_API saStatus saGetSerialNumberList (int serialNumbers[8], int ∗deviceCount)
- SA_API saStatus saOpenDeviceBySerialNumber (int ∗device, int serialNumber)
- SA_API saStatus saOpenDevice (int ∗device)
- SA_API saStatus saCloseDevice (int device)
- SA_API saStatus saPreset (int device)
- SA_API saStatus saSetCalFilePath (const char ∗path)
- SA_API saStatus saGetSerialNumber (int device, int ∗serial)
- SA_API saStatus saGetFirmwareString (int device, char firmwareString[16])
- SA_API saStatus saGetDeviceType (int device, saDeviceType ∗device_type)
- SA_API saStatus saConfigAcquisition (int device, int detector, int scale)
- SA_API saStatus saConfigCenterSpan (int device, double center, double span)

- SA_API saStatus saConfigLevel (int device, double ref)
- SA_API saStatus saConfigGainAtten (int device, int atten, int gain, bool preAmp)
- SA_API saStatus saConfigSweepCoupling (int device, double rbw, double vbw, bool reject)
- SA_API saStatus saConfigRBWShape (int device, int rbwShape)
- SA_API saStatus saConfigProcUnits (int device, int units)
- SA_API saStatus saConfigIQ (int device, int decimation, double bandwidth)
- SA_API saStatus saConfigAudio (int device, int audioType, double centerFreq, double bandwidth, double audioLowPassFreq, double audioHighPassFreq, double fmDeemphasis)
- SA_API saStatus saConfigRealTime (int device, double frameScale, int frameRate)
- SA_API saStatus saConfigRealTimeOverlap (int device, double advanceRate)
- SA_API saStatus saSetTimebase (int device, int timebase)
- SA_API saStatus saInitiate (int device, int mode, int flag)
- SA_API saStatus saAbort (int device)
- SA_API saStatus saQuerySweepInfo (int device, int ∗sweepLength, double ∗startFreq, double ∗binSize)
- SA_API saStatus saQueryStreamInfo (int device, int ∗returnLen, double ∗bandwidth, double ∗samplesPer↩ Second)
- SA_API saStatus saQueryRealTimeFrameInfo (int device, int ∗frameWidth, int ∗frameHeight)
- SA_API saStatus saQueryRealTimePoi (int device, double ∗poi)
- SA_API saStatus saGetSweep_32f (int device, float ∗min, float ∗max)
- SA_API saStatus saGetSweep_64f (int device, double ∗min, double ∗max)
- SA_API saStatus saGetPartialSweep_32f (int device, float ∗min, float ∗max, int ∗start, int ∗stop)
- SA_API saStatus saGetPartialSweep_64f (int device, double ∗min, double ∗max, int ∗start, int ∗stop)
- SA_API saStatus saGetRealTimeFrame (int device, float ∗minSweep, float ∗maxSweep, float ∗colorFrame, float ∗alphaFrame)
- SA_API saStatus saGetIQ_32f (int device, float ∗iq)
- SA_API saStatus saGetIQ_64f (int device, double ∗iq)
- SA_API saStatus saGetIQData (int device, saIQPacket ∗pkt)
- SA_API saStatus saGetIQDataUnpacked (int device, float ∗iqData, int iqCount, int purge, int ∗dataRemaining, int ∗sampleLoss, int ∗sec, int ∗milli)
- SA_API saStatus saGetAudio (int device, float ∗audio)
- SA_API saStatus saQueryTemperature (int device, float ∗temp)
- SA_API saStatus saQueryDiagnostics (int device, float ∗voltage)
- SA_API saStatus saAttachTg (int device)
- SA_API saStatus saIsTgAttached (int device, bool ∗attached)
- SA_API saStatus saConfigTgSweep (int device, int sweepSize, bool highDynamicRange, bool passive↩ Device)
- SA_API saStatus saStoreTgThru (int device, int flag)
- SA_API saStatus saSetTg (int device, double frequency, double amplitude)
- SA_API saStatus saSetTgReference (int device, int reference)
- SA_API saStatus saGetTgFreqAmpl (int device, double ∗frequency, double ∗amplitude)
- SA_API saStatus saConfigIFOutput (int device, double inputFreq, double outputFreq, int inputAtten, int outputGain)
- SA_API saStatus saSelfTest (int device, saSelfTestResults ∗results)
- SA_API const char ∗ saGetAPIVersion ()
- SA_API const char ∗ saGetProductID ()
- SA_API const char ∗ saGetErrorString (saStatus code)

7.1.1 Detailed Description

API functions for the SA44/124 spectrum analyzers.

This is the main file for user accessible functions for controlling the SA44 and SA124 spectrum analyzers.

7.1.2 Macro Definition Documentation

7.1.2.1 SA_TRUE

```
#define SA_TRUE (1)
```

Used for boolean true when integer parameters are being used.

7.1.2.2 SA_FALSE

```
#define SA_FALSE (0)
```

Used for boolean false when integer parameters are being used.

7.1.2.3 SA_MAX_DEVICES

```
#define SA_MAX_DEVICES 8
```

Maximum number of devices that can be interfaced in the API. See saGetSerialNumberList.

7.1.2.4 SA44_MIN_FREQ

```
#define SA44_MIN_FREQ (1.0)
```

Minimum frequency (Hz) for sweeps, and minimum center frequency for I/Q measurements for SA44 devices. See saConfigCenterSpan.

7.1.2.5 SA124_MIN_FREQ

```
#define SA124_MIN_FREQ (100.0e3)
```

Minimum frequency (Hz) for sweeps, and minimum center frequency for I/Q measurements for SA124 devices. See saConfigCenterSpan.

7.1.2.6 SA44_MAX_FREQ

```
#define SA44_MAX_FREQ (4.4e9)
```

Maximum frequency (Hz) for sweeps, and maximum center frequency for I/Q measurements for SA44 devices. See saConfigCenterSpan.

7.1.2.7 SA124_MAX_FREQ

```
#define SA124_MAX_FREQ (13.0e9)
```

Maximum frequency (Hz) for sweeps, and maximum center frequency for I/Q measurements for SA124 devices. See saConfigCenterSpan.

7.1.2.8 SA_MIN_SPAN

```
#define SA_MIN_SPAN (1.0)
```

Minimum span (Hz) for sweeps. See saConfigCenterSpan.

7.1.2.9 SA_MAX_REF

```
#define SA_MAX_REF (20)
```

Maximum reference level in dBm. See saConfigLevel.

7.1.2.10 SA_MAX_ATTEN

```
#define SA_MAX_ATTEN (3)
```

Maximum attentuation. Valid values [0,3] or -1 for auto. See saConfigGainAtten.

7.1.2.11 SA_MAX_GAIN

```
#define SA_MAX_GAIN (2)
```

Maximum gain. Valid values [0,2] or -1 for auto. See saConfigGainAtten.

7.1.2.12 SA_MIN_RBW

```
#define SA_MIN_RBW (0.1)
```

Minimum RBW (Hz) for sweeps. See saConfigSweepCoupling.

7.1.2.13 SA_MAX_RBW

```
#define SA_MAX_RBW (6.0e6)
```

Maximum RBW (Hz) for sweeps. See saConfigSweepCoupling.

7.1.2.14 SA_MIN_RT_RBW

```
#define SA_MIN_RT_RBW (100.0)
```

Minimum RBW (Hz) for device configured in real-time measurement mode. See saConfigSweepCoupling.

7.1.2.15 SA_MAX_RT_RBW

```
#define SA_MAX_RT_RBW (10000.0)
```

Maximum RBW (Hz) for device configured in real-time measurement mode. See saConfigSweepCoupling.

7.1.2.16 SA_MIN_IQ_BANDWIDTH

```
#define SA_MIN_IQ_BANDWIDTH (100.0)
```

Minimum I/Q bandwidth (Hz). See saConfigIQ.

7.1.2.17 SA_MAX_IQ_DECIMATION

```
#define SA_MAX_IQ_DECIMATION (128)
```

Maximum I/Q bandwidth (Hz). See saConfigIQ.

7.1.2.18 SA_IQ_SAMPLE_RATE

```
#define SA_IQ_SAMPLE_RATE (486111.111)
```

Base I/Q sample rate in Hz. See saConfigIQ.

7.1.2.19 SA_IDLE

```
#define SA_IDLE (-1)
```

Measurement mode: Idle, no measurement. See saInitiate.

7.1.2.20 SA_SWEEPING

```
#define SA_SWEEPING (0x0)
```

Measurement mode: Swept spectrum analysis. See saInitiate.

7.1.2.21 SA_REAL_TIME

```
#define SA_REAL_TIME (0x1)
```

Measurement mode: Real-time spectrum analysis. See saInitiate.

7.1.2.22 SA_IQ

```
#define SA_IQ (0x2)
```

Measurement mode: I/Q streaming. See saInitiate.

7.1.2.23 SA_AUDIO

```
#define SA_AUDIO (0x3)
```

Measurement mode: Audio demod. See saInitiate.

7.1.2.24 SA_TG_SWEEP

```
#define SA_TG_SWEEP (0x4)
```

Measurement mode: Tracking generator sweeps for scalar network analysis. See saInitiate.

7.1.2.25 SA_RBW_SHAPE_FLATTOP

```
#define SA_RBW_SHAPE_FLATTOP (0x1)
```

Specifies the Stanford flattop window used for sweep and real-time analysis. See saConfigRBWShape.

7.1.2.26 SA_RBW_SHAPE_CISPR

```
#define SA_RBW_SHAPE_CISPR (0x2)
```

Specifies a Gaussian window with 6dB cutoff used for sweep and real-time analysis. See saConfigRBWShape.

7.1.2.27 SA_MIN_MAX

```
#define SA_MIN_MAX (0x0)
```

Use min/max detector for sweep and real-time spectrum analysis. See saConfigAcquisition.

7.1.2.28 SA_AVERAGE

```
#define SA_AVERAGE (0x1)
```

Use average detector for sweep and real-time spectrum analysis. See saConfigAcquisition.

7.1.2.29 SA_LOG_SCALE

```
#define SA_LOG_SCALE (0x0)
```

Specifies dBm units of sweep and real-time spectrum analysis measurements. See saConfigAcquisition.

7.1.2.30 SA_LIN_SCALE

```
#define SA_LIN_SCALE (0x1)
```

Specifies mV units of sweep and real-time spectrum analysis measurements. See saConfigAcquisition.

7.1.2.31 SA_LOG_FULL_SCALE

```
#define SA_LOG_FULL_SCALE (0x2)
```

Specifies dBm units, with no corrections, of sweep and real-time spectrum analysis measurements. See saConfigAcquisition.

7.1.2.32 SA_LIN_FULL_SCALE

```
#define SA_LIN_FULL_SCALE (0x3)
```

Specifies mV units, with no corrections, of sweep and real-time spectrum analysis measurements. See saConfigAcquisition.

7.1.2.33 SA_AUTO_ATTEN

```
#define SA_AUTO_ATTEN (-1)
```

Automatically choose attenuation based on reference level. See saConfigGainAtten.

7.1.2.34 SA_AUTO_GAIN

```
#define SA_AUTO_GAIN (-1)
```

Automatically choose gain based on reference level. See saConfigGainAtten.

7.1.2.35 SA_LOG_UNITS

```
#define SA_LOG_UNITS (0x0)
```

VBW processing occurs in dBm. See saConfigProcUnits.

7.1.2.36 SA_VOLT_UNITS

```
#define SA_VOLT_UNITS (0x1)
```

VBW processing occurs in linear voltage units (mV). See saConfigProcUnits.

7.1.2.37 SA_POWER_UNITS

```
#define SA_POWER_UNITS (0x2)
```

VBW processing occurs in linear power units (mW). See saConfigProcUnits.

7.1.2.38 SA_BYPASS

```
#define SA_BYPASS (0x3)
```

No VBW processing. See saConfigProcUnits.

7.1.2.39 SA_AUDIO_AM

```
#define SA_AUDIO_AM (0x0)
```

Audio demodulation type: AM. See saConfigAudio.

7.1.2.40 SA_AUDIO_FM

```
#define SA_AUDIO_FM (0x1)
```

Audio demodulation type: FM. See saConfigAudio.

7.1.2.41 SA_AUDIO_USB

```
#define SA_AUDIO_USB (0x2)
```

Audio demodulation type: Upper side band. See saConfigAudio.

7.1.2.42 SA_AUDIO_LSB

```
#define SA_AUDIO_LSB (0x3)
```

Audio demodulation type: Lower side band. See saConfigAudio.

7.1.2.43 SA_AUDIO_CW

```
#define SA_AUDIO_CW (0x4)
```

Audio demodulation type: CW. See saConfigAudio.

7.1.2.44 TG_THRU_0DB

```
#define TG_THRU_0DB (0x1)
```

In scalar network analysis, use the next trace as a thru. See saStoreTgThru.

7.1.2.45 TG_THRU_20DB

```
#define TG_THRU_20DB (0x2)
```

In scalar network analysis, improve accuracy with a second thru step. See saStoreTgThru.

7.1.2.46 SA_REF_UNUSED

```
#define SA_REF_UNUSED (0)
```

Additional corrections are applied to tracking generator timebase. See saSetTgReference.

7.1.2.47 SA_REF_INTERNAL_OUT

```
#define SA_REF_INTERNAL_OUT (1)
```

Use tracking generator timebase as frequency standard for system, and do not apply additional corrections. See saSetTgReference.

7.1.2.48 SA_REF_EXTERNAL_IN

```
#define SA_REF_EXTERNAL_IN (2)
```

Use an external reference for TG124A, and do not apply additional corrections to timebase. See saSetTgReference.

7.1.3 Enumeration Type Documentation

7.1.3.1 saDeviceType

```
enum saDeviceType
```

Device type

Enumerator

saDeviceTypeNone	None
saDeviceTypeSA44	SA44
saDeviceTypeSA44B	SA44B
saDeviceTypeSA124A	SA124A
saDeviceTypeSA124B	SA124B

7.1.3.2 saStatus

```
enum saStatus
```

Status code returned from all SA API functions. Errors are negative and suffixed with 'Err'. Errors stop the flow of execution, warnings do not.

Enumerator

saUnknownErr	Unknown/unexpected error.
saFrequencyRangeErr	Span outside frequency range.
saInvalidDetectorErr	Invalid detector value provided.
saInvalidScaleErr	Invalid scale provided.
saBandwidthErr	Invalid resolution bandwidth provided: must be between 0.1 Hz and 4.4 GHz.
saExternalReferenceNotFound	External Reference Not Found. Internal reference will be used.
saLNAErr	LNA error.
saOvenColdErr	10 MHz OCXO cold. Try again after 60 second warm-up.
saInternetErr	Unable to connect to the internet or unable to find the necessary calibration file on the internet during initialization. Ensure your PC has an internet connection and try again. If it is connected to the internet and you are still receiving this message, please contact Signal Hound.
saUSBCommErr	USB communications error.
saTrackingGeneratorNotFound	Unable to find a connected and available Signal Hound tracking generator.
saDeviceNotIdleErr	Cannot perform requested operation while the device is active.
saDeviceNotFoundErr	Device not found.
saInvalidModeErr	Cannot perform the requested operation in this mode of operation.
saNotConfiguredErr	The device is not properly configured.
saTooManyDevicesErr	Unable to open any more devices.
saInvalidParameterErr	Invalid parameter provided.
saDeviceNotOpenErr	Device specified is not open.
saInvalidDeviceErr	Invalid device number provided.
saNullPtrErr	One or more parameters are NULL.
saNoError	Function returned successfully. No warnings or errors.
saNoCorrections	No corrections found on device, data will be uncalibrated.
saCompressionWarning	Device in compression. IF Overload.
saParameterClamped	Supplied parameter clamped/limited to a set of known values or to within an accepted range of operating values.
saBandwidthClamped	Supplied bandwidth limited to within an accepted range of operating values.
saCalFilePermissions	Unable to store correction data locally.

7.1.4 Function Documentation

7.1.4.1 saGetSerialNumberList()

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

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

Parameters

out	*serialNumbers*	A pointer to an array of at minimum SA_MAX_DEVICES contiguous 32-bit integers. It is undefined behavior if the array pointed to by *serialNumbers* is not SA_MAX_DEVICES integers in length.
out	*deviceCount*	Pointer to an integer. Will be set to the number of devices found on the system.

Returns

7.1.4.2 saOpenDeviceBySerialNumber()

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

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

Parameters

out	*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.
in	*serialNumber*	User-provided serial number.

Returns

7.1.4.3 saOpenDevice()

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

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.

When successful, this function 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.

Parameters

out	*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.

Returns

7.1.4.4 saCloseDevice()

```
SA_API saStatus saCloseDevice (
              int device )
```

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.

Parameters

in	*device*	Device handle.

Returns

7.1.4.5 saPreset()

```
SA_API saStatus saPreset (
              int device )
```

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 to 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.

7.1.4.6 Example

```
 {c++}
// 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);
}
```

Parameters

in	*device*	Device handle.

Returns

7.1.4.7 saSetCalFilePath()

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

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/`.

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

Parameters

in	*path*	

Returns

7.1.4.9 saGetSerialNumber()

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

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

Parameters

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

Returns

7.1.4.10 saGetFirmwareString()

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

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

Parameters

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

Returns

7.1.4.11 saGetDeviceType()

```
SA_API saStatus saGetDeviceType (
            int device,
            saDeviceType * device_type )
```

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 the handle. *type* is an enumerated value of type saDeviceType.

Parameters

in	*device*	Device handle.
out	*device_type*	Pointer to an integer to receive the model type.

Returns

7.1.4.12 saConfigAcquisition()

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

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, 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

Parameters

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

Returns

7.1.4.13 saConfigCenterSpan()

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

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 saQuerySweepInfo for more information.

Each device has a specified operational frequency range between some minimum and maximum frequency. The limits are defined by SA44_MIN_FREQ, SA124_MIN_FREQ, SA44_MAX_FREQ, and SA124_MAX_FREQ. 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.

Parameters

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

Returns

7.1.4.14 saConfigLevel()

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

This function is best utilized when the device attenuation and gain are set to automatic (default). When both attenuation and gain are set to SA_AUTO_ATTEN and SA_AUTO_GAIN, respectively, 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, use auto gain and atten, and set your reference level at or slightly about your expected input power for best sensitivity. Reference level is specified in dBm units.

Parameters

in	*device*	Device handle.
in	*ref*	Reference level in dBm.

Returns

7.1.4.15 saConfigGainAtten()

```
SA_API saStatus saConfigGainAtten (
            int device,
            int atten,
            int gain,
            bool preAmp )
```

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.

7.1.4.16 atten parameter

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

7.1.4.17 gain parameter

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 Setting Gain and Attenuation.

Parameters

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

Returns

7.1.4.18 saConfigSweepCoupling()

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

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 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 `SA44B User Manual` or `SA124B User Manual` 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`, `BB60D`, `SM200B`, `SM200C`, or `SM435B` spectrum analyzers.

Parameters

in	*device*	Device handle.
in	*rbw*	Resolution bandwidth in Hz. RBW can be arbitrary.
in	*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.
in	*reject*	Indicates whether to enable image rejection.

Returns

7.1.4.19 saConfigRBWShape()

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

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.

Parameters

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

Returns

7.1.4.20 saConfigProcUnits()

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

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.

Macro	Unit
SA_LOG_UNITS	dBm
SA_VOLT_UNITS	mV
SA_POWER_UNITS	mW
SA_BYPASS	No video processing

Parameters

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

Returns

7.1.4.21 saConfigIQ()

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

This function is used to configure the digital I/Q data stream. A decimation factor and filter bandwidth are able to be specified. The decimation rate divides the I/Q 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 = 486111.11111~ / 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

Parameters

in	*device*	Device handle.
in	*decimation*	Specify a decimation rate for the I/Q data stream.
in	*bandwidth*	Specify the band pass filter width on the I/Q digital stream.

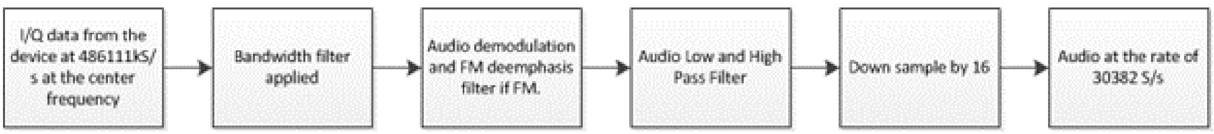
Returns

7.1.4.22 saConfigAudio()

```
SA_API saStatus saConfigAudio (
            int device,
            int audioType,
            double centerFreq,
            double bandwidth,
            double audioLowPassFreq,
            double audioHighPassFreq,
            double fmDeemphasis )
```

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:



Parameters

in	*device*	Device handle.
	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 microseconds. Should be between 1 and 100. This value is ignored if *audioType* is not equal to SA_AUDIO_FM.

Returns

7.1.4.23 saConfigRealTime()

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

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 Real-Time Spectrum Analysis.

Parameters

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

Returns

7.1.4.24 saConfigRealTimeOverlap()

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

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 is between [0.125, 10]. Increasing the advance rate reduces processing considerably but also increases the 100% probability of intercept of the device.

Parameters

in	*device*	Device handle.
in	*advanceRate*	FFT advance rate. See description.

Returns

7.1.4.25 saSetTimebase()

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

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.

Parameters

in	*device*	Device handle.
in	*timebase*	Time base setting value. Acceptable inputs are SA_REF_INTERNAL_OUT and SA_REF_EXTERNAL_IN.

Returns

7.1.4.26 saInitiate()

```
SA_API saStatus saInitiate (
            int device,
            int mode,
            int flag )
```

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.

Parameters

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

Returns

7.1.4.27 saAbort()

```
SA_API saStatus saAbort (
            int device )
```

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.

Parameters

in	*device*	Device handle.

Returns

7.1.4.28 saQuerySweepInfo()

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

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

Parameters

in	*device*	Device handle.
out	*sweepLength*	A pointer to a 32-bit integer. If the function returns successfully, the integer pointed to will contain the size of the *min* and *max* arrays returned by saGetSweep_32f and saGetSweep_64f.
out	*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.
out	*binSize*	A pointer to a 64-bit floating point variable. If the function returns successfully, the variable *binSize* points to will contain the frequency difference between each bin in the configured sweep.

Returns

7.1.4.29 saQueryStreamInfo()

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

Use this function to get the parameters of the I/Q data stream.

Parameters

in	*device*	Device handle.
out	*returnLen*	Pointer to a 32-bit integer. If the function returns successfully, the variable *returnLen* points to will contain the number of I/Q sample pairs that will be returned by calling saGetIQ_32f/saGetIQ_64f.
out	*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.
out	*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 I/Q data stream.

Returns

7.1.4.30 saQueryRealTimeFrameInfo()

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

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.

Parameters

in	*device*	Device handle.
out	*frameWidth*	Pointer to a 32-bit signed integer representing the width of the real-time frame.
out	*frameHeight*	Pointer to a 32-bit signed integer representing the height of the real-time frame.

Returns

7.1.4.31 saQueryRealTimePoi()

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

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.

Parameters

in	*device*	Device handle.
out	*poi*	Pointer to double. See description.

Returns

7.1.4.32 saGetSweep_32f()

```
SA_API saStatus saGetSweep_32f (
            int device,
            float * min,
            float * max )
```

Upon returning successfully, this function returns the *minimum* and *maximum* floating point 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.

Parameters

in	*device*	Device handle.
out	*min*	Pointer to the beginning of an array of 32-bit floating point values, whose length is equal to or greater than *sweepLength* returned from saQuerySweepInfo.
out	*max*	Pointer to the beginning of an array of 32-bit floating point values, whose length is equal to or greater than *sweepLength* returned from saQuerySweepInfo.

Returns

7.1.4.33 saGetSweep_64f()

```
SA_API saStatus saGetSweep_64f (
            int device,
            double * min,
            double * max )
```

See saGetSweep_32f.

Parameters

in	*device*	Device handle.
out	*min*	Pointer to the beginning of an array of 64-bit floating point values, whose length is equal to or greater than *sweepLength* returned from saQuerySweepInfo.
out	*max*	Pointer to the beginning of an array of 64-bit floating point values, whose length is equal to or greater than *sweepLength* returned from saQuerySweepInfo.

Returns

7.1.4.34 saGetPartialSweep_32f()

```
SA_API saStatus saGetPartialSweep_32f (
            int device,
            float * min,
            float * max,
            int * start,
            int * stop )
```

This function is similar to the saGetSweep_32f/saGetSweep_64f 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_32f/saGetPartialSweep_32f, 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.

Parameters

in	*device*	Device handle.
	min	Pointer to an array of 32-bit floating point values, whose length is equal to or greater than *sweepLength* returned from saQuerySweepInfo.
	max	Pointer to an array of 32-bit 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.

Returns

7.1.4.35 saGetPartialSweep_64f()

```
SA_API saStatus saGetPartialSweep_64f (
            int device,
            double * min,
            double * max,
            int * start,
            int * stop )
```

See saGetPartialSweep_32f.

Parameters

in	*device*	Device handle.
	min	Pointer to an array of 64-bit floating point values, whose length is equal to or greater than *sweepLength* returned from saQuerySweepInfo.
	max	Pointer to an array of 64-bit 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.

Returns

7.1.4.36 saGetRealTimeFrame()

```
SA_API saStatus saGetRealTimeFrame (
            int device,
            float * minSweep,
            float * maxSweep,
            float * colorFrame,
            float * alphaFrame )
```

This function is used to retrieve the real-time sweeps, frame, and alpha frame for a measurement interval. This function should be used instead of saGetSweep_32f/saGetSweep_64f and saGetPartialSweep_32f/saGetPartialSweep_64f for real-time mode. The sweep array should be 'N' contiguous floats, where N is the sweep length returned from saQuerySweepInfo. The *frame* and *alphaFrame* should be WxH values long where W and H are the values returned from saQueryRealTimeFrameInfo. For more information see Real-Time Spectrum Analysis.

Parameters

in	*device*	Device handle.
out	*minSweep*	If this pointer is non-null, the min held sweep will be returned to the user. If the detector is set to average, this array will be identical to the *maxSweep* array returned.
out	*maxSweep*	If this pointer is non-null, the max held sweep will be returned to the user. If the detector is set to average, this array contains the averaged results over the measurement interval.
out	*colorFrame*	Pointer to a 32-bit floating point array. If the function returns successfully, the contents of the array will contain a single real-time frame.
out	*alphaFrame*	Pointer to a 32-bit floating point array. If the function returns successfully, the contents of the array will contain a single real-time alpha frame.

Returns

7.1.4.37 saGetIQ_32f()

```
SA_API saStatus saGetIQ_32f (
            int device,
            float * iq )
```

Retrieve the next array of I/Q 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 I/Q 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 I/Q sample pairs scaled to mW. The time difference between each sample can be determined from the sample rate of the configured device.

Parameters

in	*device*	Device handle.
out	*iq*	A pointer to a 32-bit floating point array. The contents of this buffer will be updated with interleaved I/Q digital samples.

Returns

7.1.4.38 saGetIQ_64f()

```
SA_API saStatus saGetIQ_64f (
            int device,
            double * iq )
```

See saGetIQ_32f.

Parameters

in	*device*	Device handle.
out	*iq*	A pointer to a 64-bit floating point array. The contents of this buffer will be updated with interleaved I/Q digital samples.

Returns

7.1.4.39 saGetIQData()

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

This function retrieves one block of I/Q data as specified by the saIQPacket struct.

Parameters

in	*device*	Device handle.
out	*pkt*	Pointer to a saIQPacket struct.

Returns

7.1.4.40 saGetIQDataUnpacked()

```
SA_API saStatus saGetIQDataUnpacked (
            int device,
            float * iqData,
            int iqCount,
            int purge,
            int * dataRemaining,
            int * sampleLoss,
            int * sec,
            int * milli )
```

This function provides an alternate interface to the saGetIQData function. Using this function, you can eliminate the need for the saIQPacket struct, which eases development of non-C programming language bindings (languages and environments such as LabVIEW, MATLAB, Python, C#, etc).

The parameters to this function have a one-to-one mapping to the members of the saIQPacket struct.

This function is implemented by populating the saIQPacket struct with the provided parameters and calling saGetIQData.

Parameters

in	*device*	Device handle.
out	*iqData*	See saIQPacket.
out	*iqCount*	See saIQPacket.
out	*purge*	See saIQPacket.
out	*dataRemaining*	See saIQPacket.
out	*sampleLoss*	See saIQPacket.
out	*sec*	See saIQPacket.
out	*milli*	See saIQPacket.

Returns

7.1.4.41 saGetAudio()

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

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.

Parameters

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

Returns

7.1.4.42 saQueryTemperature()

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

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.

Parameters

in	*device*	Device handle.
out	*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.

Returns

7.1.4.43 saQueryDiagnostics()

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

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.

Parameters

in	*device*	Device handle.
out	*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.

Returns

7.1.4.44 saAttachTg()

```
SA_API saStatus saAttachTg (
            int device )
```

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

Parameters

in	*device*	Device handle.

Returns

7.1.4.45 saIsTgAttached()

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

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

Parameters

in	*device*	Device handle.
out	*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.

Returns

7.1.4.46 saConfigTgSweep()

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

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).

Parameters

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

Returns

7.1.4.47 saStoreTgThru()

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

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.

Parameters

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

Returns

7.1.4.48 saSetTg()

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

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

Parameters

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

Returns

7.1.4.49 saSetTgReference()

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

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.

Parameters

in	*device*	Device handle.
in	*reference*	A valid time base setting value. Possible values are SA_REF_UNUSED, SA_REF_INTERNAL_OUT, and SA_REF_EXTERNAL_IN.

Returns

7.1.4.50 saGetTgFreqAmpl()

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

Retrieve the last set TG output parameters the user set through the saSetTg function. The saSetTg 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 saSetTg function is called again.

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

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

Parameters

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

Returns

7.1.4.51 saConfigIFOutput()

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

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 I/Q 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.

Parameters

in	*device*	Device handle.
in	*inputFreq*	The input center frequency on the SMA connector specified in Hz. Must be between 125MHz and 13GHz.
in	*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.
in	*inputAtten*	Attenuation of the input signal specified in dB. Must be between 0 and 30 dB.
in	*outputGain*	Amplification of the output signal specified in dB. Must be between 0 and 60 dB.

Returns

7.1.4.52 saSelfTest()

```
SA_API saStatus saSelfTest (
            int device,
            saSelfTestResults * results )
```

Performs a self-test and returns the results in an saSelfTestResults struct.

Parameters

in	*device*	Device handle.
out	*results*	A pointer to an saSelfTestResults struct.

Returns

7.1.4.53 saGetAPIVersion()

```
SA_API const char * saGetAPIVersion ( )
```

Get the current API version.

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"
```

Returns

7.1.4.54 saGetProductID()

```
SA_API const char * saGetProductID ( )
```

The product ID is 4302-1103.

Returns

7.1.4.55 saGetErrorString()

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

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

Parameters

in	*code*	A saStatus value returned from an API call.

Returns

7.2 sa_api.h

Go to the documentation of this file.

```
// Copyright (c) 2022 Signal Hound
// For licensing information, please see the API license in the software_licenses folder

#ifndef __SA_API_H__
#define __SA_API_H__

#if defined(_WIN32)
#ifdef SA_EXPORTS
#define SA_API __declspec(dllexport)
#else
#define SA_API __declspec(dllimport)
#endif
#else // Linux
#define SA_API
#endif

#if defined(_WIN32)
#define SA_DEPRECATED(msg) __declspec(deprecated(msg))
#elif  defined(__GNUC__)
#define SA_DEPRECATED(msg) __attribute__((deprecated))
#else
#define SA_DEPRECATED(msg) msg
#endif

#define SA_TRUE (1)
#define SA_FALSE (0)

#define SA_MAX_DEVICES 8

typedef enum saDeviceType {
    saDeviceTypeNone = 0,
    saDeviceTypeSA44 = 1,
    saDeviceTypeSA44B = 2,
    saDeviceTypeSA124A = 3,
    saDeviceTypeSA124B = 4
} saDeviceType;

#define SA44_MIN_FREQ (1.0)
#define SA124_MIN_FREQ (100.0e3)
#define SA44_MAX_FREQ (4.4e9)
#define SA124_MAX_FREQ (13.0e9)
#define SA_MIN_SPAN (1.0)
#define SA_MAX_REF (20)
#define SA_MAX_ATTEN (3)
#define SA_MAX_GAIN (2)
#define SA_MIN_RBW (0.1)
#define SA_MAX_RBW (6.0e6)
#define SA_MIN_RT_RBW (100.0)
#define SA_MAX_RT_RBW (10000.0)
#define SA_MIN_IQ_BANDWIDTH (100.0)
#define SA_MAX_IQ_DECIMATION (128)

#define SA_IQ_SAMPLE_RATE (486111.111)

#define SA_IDLE      (-1)
#define SA_SWEEPING  (0x0)
#define SA_REAL_TIME (0x1)
#define SA_IQ        (0x2)
#define SA_AUDIO     (0x3)
#define SA_TG_SWEEP  (0x4)

#define SA_RBW_SHAPE_FLATTOP (0x1)
#define SA_RBW_SHAPE_CISPR (0x2)

#define SA_MIN_MAX (0x0)
#define SA_AVERAGE (0x1)

#define SA_LOG_SCALE      (0x0)
#define SA_LIN_SCALE      (0x1)
#define SA_LOG_FULL_SCALE (0x2) // N/A
#define SA_LIN_FULL_SCALE (0x3) // N/A

#define SA_AUTO_ATTEN (-1)
#define SA_AUTO_GAIN  (-1)
```

```
#define SA_LOG_UNITS   (0x0)
#define SA_VOLT_UNITS  (0x1)
#define SA_POWER_UNITS (0x2)
#define SA_BYPASS      (0x3)

#define SA_AUDIO_AM  (0x0)
#define SA_AUDIO_FM  (0x1)
#define SA_AUDIO_USB (0x2)
#define SA_AUDIO_LSB (0x3)
#define SA_AUDIO_CW  (0x4)

#define TG_THRU_0DB  (0x1)
#define TG_THRU_20DB  (0x2)

#define SA_REF_UNUSED (0)
#define SA_REF_INTERNAL_OUT (1)
#define SA_REF_EXTERNAL_IN (2)

typedef struct saSelfTestResults {
    bool highBandMixer, lowBandMixer;
    bool attenuator, secondIF, preamplifier;
    double highBandMixerValue, lowBandMixerValue;
    double attenuatorValue, secondIFValue, preamplifierValue;
} saSelfTestResults;

typedef struct saIQPacket {
    float *iqData;
    int iqCount;
    int purge;
    int dataRemaining;
    int sampleLoss;
    int sec;
    int milli;
} saIQPacket;

typedef enum saStatus {
    saUnknownErr = -666,

    // Setting specific error codes

    saFrequencyRangeErr = -99,
    saInvalidDetectorErr = -95,
    saInvalidScaleErr = -94,
    saBandwidthErr = -91,
    saExternalReferenceNotFound = -89,

    // Device-specific errors

    saLNAErr = -21,
    saOvenColdErr = -20,

    // Data errors

    saInternetErr = -12,
    saUSBCommErr = -11,

    // General configuration errors

    saTrackingGeneratorNotFound = -10,
    saDeviceNotIdleErr = -9,
    saDeviceNotFoundErr = -8,
    saInvalidModeErr = -7,
    saNotConfiguredErr = -6,
    saTooManyDevicesErr = -5,
    saInvalidParameterErr = -4,
    saDeviceNotOpenErr = -3,
    saInvalidDeviceErr = -2,
    saNullPtrErr = -1,

    saNoError = 0,

    // Warnings

    saNoCorrections = 1,
    saCompressionWarning = 2,
    saParameterClamped = 3,
    saBandwidthClamped = 4,
    saCalFilePermissions = 5,
} saStatus;

#ifdef __cplusplus
extern "C" {
#endif

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

```

```
SA_API saStatus saOpenDeviceBySerialNumber(int *device, int serialNumber);
SA_API saStatus saOpenDevice(int *device);
SA_API saStatus saCloseDevice(int device);
SA_API saStatus saPreset(int device);
SA_API saStatus saSetCalFilePath(const char *path);
SA_API saStatus saGetSerialNumber(int device, int *serial);
SA_API saStatus saGetFirmwareString(int device, char firmwareString[16]);
SA_API saStatus saGetDeviceType(int device, saDeviceType *device_type);
SA_API saStatus saConfigAcquisition(int device, int detector, int scale);
SA_API saStatus saConfigCenterSpan(int device, double center, double span);
SA_API saStatus saConfigLevel(int device, double ref);
SA_API saStatus saConfigGainAtten(int device, int atten, int gain, bool preAmp);
SA_API saStatus saConfigSweepCoupling(int device, double rbw, double vbw, bool reject);
SA_API saStatus saConfigRBWShape(int device, int rbwShape);
SA_API saStatus saConfigProcUnits(int device, int units);
SA_API saStatus saConfigIQ(int device, int decimation, double bandwidth);
SA_API saStatus saConfigAudio(int device, int audioType, double centerFreq,
                              double bandwidth, double audioLowPassFreq,
                              double audioHighPassFreq, double fmDeemphasis);
SA_API saStatus saConfigRealTime(int device, double frameScale, int frameRate);
SA_API saStatus saConfigRealTimeOverlap(int device, double advanceRate);
SA_API saStatus saSetTimebase(int device, int timebase);
SA_API saStatus saInitiate(int device, int mode, int flag);
SA_API saStatus saAbort(int device);
SA_API saStatus saQuerySweepInfo(int device, int *sweepLength, double *startFreq, double *binSize);
SA_API saStatus saQueryStreamInfo(int device, int *returnLen, double *bandwidth, double
  *samplesPerSecond);
SA_API saStatus saQueryRealTimeFrameInfo(int device, int *frameWidth, int *frameHeight);
SA_API saStatus saQueryRealTimePoi(int device, double *poi);
SA_API saStatus saGetSweep_32f(int device, float *min, float *max);
SA_API saStatus saGetSweep_64f(int device, double *min, double *max);
SA_API saStatus saGetPartialSweep_32f(int device, float *min, float *max, int *start, int *stop);
SA_API saStatus saGetPartialSweep_64f(int device, double *min, double *max, int *start, int *stop);
SA_API saStatus saGetRealTimeFrame(int device, float *minSweep, float *maxSweep, float *colorFrame,
  float *alphaFrame);
SA_API saStatus saGetIQ_32f(int device, float *iq);
SA_API saStatus saGetIQ_64f(int device, double *iq);
SA_API saStatus saGetIQData(int device, saIQPacket *pkt);
SA_API saStatus saGetIQDataUnpacked(int device, float *iqData, int iqCount, int purge,
                                    int *dataRemaining, int *sampleLoss, int *sec, int *milli);
SA_API saStatus saGetAudio(int device, float *audio);
SA_API saStatus saQueryTemperature(int device, float *temp);
SA_API saStatus saQueryDiagnostics(int device, float *voltage);
SA_API saStatus saAttachTg(int device);
SA_API saStatus saIsTgAttached(int device, bool *attached);
SA_API saStatus saConfigTgSweep(int device, int sweepSize, bool highDynamicRange, bool passiveDevice);
```

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

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

SA_API saStatus saSetTgReference(int device, int reference);

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

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

SA_API saStatus saSelfTest(int device, saSelfTestResults *results);

SA_API const char* saGetAPIVersion();

SA_API const char* saGetProductID();

SA_API const char* saGetErrorString(saStatus code);

#ifdef __cplusplus
} // extern "C"
#endif

#endif // SA_API_H
```

Index