

SP API

Generated by Doxygen 1.9.4

1 SP145 API Reference	1
1.1 Examples	1
1.2 Measurements	1
1.3 Build/Version Notes	1
1.4 PC Requirements	2
1.4.1 Windows Development Requirements	2
1.4.2 Linux Development Requirements	2
1.4.3 Other Requirements	2
1.5 Reference Level and Sensitivity	2
1.6 GPS and Timestamps	3
1.6.1 Acquiring GPS Lock	3
1.6.2 GPS Time Stamping	3
1.6.3 GPS Disciplining	3
1.7 Power States	4
1.8 Thread Safety	4
1.9 Multiple Devices and Multiple Processes	5
1.10 Linux Notes	5
1.10.1 USB Throughput	5
1.10.2 Multiple USB Devices	5
1.11 Programming Languages Compatibility	6
1.11.1 Python	6
1.11.1.1 Prerequisites	6
1.11.1.2 Setup	6
1.11.1.3 Usage	7
1.12 I/Q Acquisition	7
1.12.1 I/Q Sample Rates	7
1.12.2 I/Q Data Types	7
1.12.2.1 Converting From Full Scale to Corrected I/Q	7
1.12.3 I/Q Filtering and Bandwidth	8
1.13 Estimating Sweep Size	8
1.14 Window Functions	9
1.15 Automatic GPS Timebase Discipline	9
1.16 Contact Information	9
2 Basic API Usage	11
2.1 Opening a Device	11
2.2 Configuring the Device	11
2.3 Acquiring Measurements	11
2.3.1 Stopping the Measurements	12
2.4 Closing the Device	12
2.5 Recalibration	12
3 Sweep Mode	13

3.1 Example	13
3.2 Basics	13
3.3 Sweep Format	13
3.4 Min and Max Sweep Arrays	14
4 Real-Time Spectrum Analysis	15
4.1 Example	15
4.2 Basics	15
4.3 Real-Time Sweep	15
4.4 Real-Time Frame	16
4.5 RBW Restrictions	17
5 I/Q Streaming	19
5.1 Example	19
5.2 Basics	19
5.3 Sample Rate, Decimation, and Bandwidth	19
5.4 Polling Interface (I/Q)	20
5.5 External Triggering	20
5.6 Additional Information	20
6 I/Q Sweep List / Frequency Hopping	21
6.1 Example	21
6.2 Basics	21
6.2.1 Sweep List Configuration Example	21
6.3 Notes on Performance	22
6.4 I/Q Streaming vs I/Q Sweep List	22
6.5 Additional Information	22
7 I/Q Full Band	23
8 File Index	25
8.1 File List	25
9 File Documentation	27
9.1 sp_api.h File Reference	27
9.1.1 Detailed Description	29
9.1.2 Macro Definition Documentation	29
9.1.2.1 SP_TRUE	30
9.1.2.2 SP_FALSE	30
9.1.2.3 SP_MAX_DEVICES	30
9.1.2.4 SP_MAX_REF_LEVEL	30
9.1.2.5 SP_AUTO_ATTEN	30
9.1.2.6 SP_MAX_ATTEN	30
9.1.2.7 SP_MIN_FREQ	30

9.1.2.8 SP_MAX_FREQ	30
9.1.2.9 SP_MIN_SWEEP_TIME	31
9.1.2.10 SP_MAX_SWEEP_TIME	31
9.1.2.11 SP_REAL_TIME_MIN_SPAN	31
9.1.2.12 SP_REAL_TIME_MAX_SPAN	31
9.1.2.13 SP_REAL_TIME_MIN_RBW	31
9.1.2.14 SP_REAL_TIME_MAX_RBW	31
9.1.2.15 SP_MAX_IQ_DECIMATION	31
9.1.2.16 SP_MAX_SWEEP_QUEUE_SZ	31
9.1.3 Enumeration Type Documentation	31
9.1.3.1 SpStatus	31
9.1.3.2 SpDeviceType	32
9.1.3.3 SpBool	32
9.1.3.4 SpPowerState	33
9.1.3.5 SpMode	33
9.1.3.6 SpDetector	33
9.1.3.7 SpScale	34
9.1.3.8 SpVideoUnits	34
9.1.3.9 SpWindowType	34
9.1.3.10 SpDataType	35
9.1.3.11 SpTriggerEdge	35
9.1.3.12 SpGPSSState	35
9.1.3.13 SpAudioType	35
9.1.4 Function Documentation	36
9.1.4.1 spGetDeviceList()	36
9.1.4.2 spOpenDevice()	36
9.1.4.3 spOpenDeviceBySerial()	37
9.1.4.4 spCloseDevice()	37
9.1.4.5 spPresetDevice()	38
9.1.4.6 spSetPowerState()	38
9.1.4.7 spGetPowerState()	38
9.1.4.8 spGetSerialNumber()	39
9.1.4.9 spGetFirmwareVersion()	39
9.1.4.10 spGetDeviceDiagnostics()	40
9.1.4.11 spGetCalDate()	40
9.1.4.12 spSetExtReference()	40
9.1.4.13 spGetExtReference()	41
9.1.4.14 spSetGPSTimebaseUpdate()	41
9.1.4.15 spGetGPSTimebaseUpdate()	42
9.1.4.16 spGetGPSHoldoverInfo()	42
9.1.4.17 spGetGPSSState()	42
9.1.4.18 spSetRefLevel()	43

9.1.4.19 spGetRefLevel()	43
9.1.4.20 spSetAttenuator()	44
9.1.4.21 spGetAttenuator()	44
9.1.4.22 spSetSweepCenterSpan()	44
9.1.4.23 spSetSweepStartStop()	45
9.1.4.24 spSetSweepCoupling()	45
9.1.4.25 spSetSweepDetector()	46
9.1.4.26 spSetSweepScale()	46
9.1.4.27 spSetSweepWindow()	46
9.1.4.28 spSetRealTimeCenterSpan()	47
9.1.4.29 spSetRealTimeRBW()	47
9.1.4.30 spSetRealTimeDetector()	48
9.1.4.31 spSetRealTimeScale()	48
9.1.4.32 spSetRealTimeWindow()	49
9.1.4.33 spSetIQDataType()	49
9.1.4.34 spSetIQCenterFreq()	49
9.1.4.35 spGetIQCenterFreq()	50
9.1.4.36 spSetIQSampleRate()	50
9.1.4.37 spSetIQSoftwareFilter()	50
9.1.4.38 spSetIQBandwidth()	51
9.1.4.39 spSetIQExtTriggerEdge()	51
9.1.4.40 spSetIQTriggerSentinel()	52
9.1.4.41 spSetIQQueueSize()	52
9.1.4.42 spSetIQSweepListDataType()	52
9.1.4.43 spSetIQSweepListCorrected()	53
9.1.4.44 spSetIQSweepListSteps()	53
9.1.4.45 spGetIQSweepListSteps()	54
9.1.4.46 spSetIQSweepListFreq()	54
9.1.4.47 spSetIQSweepListRef()	54
9.1.4.48 spSetIQSweepListAtten()	55
9.1.4.49 spSetIQSweepListSampleCount()	55
9.1.4.50 spSetAudioCenterFreq()	56
9.1.4.51 spSetAudioType()	56
9.1.4.52 spSetAudioFilters()	56
9.1.4.53 spSetAudioFMDeemphasis()	57
9.1.4.54 spConfigure()	57
9.1.4.55 spGetCurrentMode()	58
9.1.4.56 spAbort()	58
9.1.4.57 spGetSweepParameters()	59
9.1.4.58 spGetRealTimeParameters()	59
9.1.4.59 spGetIQParameters()	60
9.1.4.60 spGetIQCorrection()	60

9.1.4.61 splQSweepListGetCorrections()	61
9.1.4.62 spGetSweep()	61
9.1.4.63 spGetRealTimeFrame()	61
9.1.4.64 spGetIQ()	62
9.1.4.65 splQSweepListGetSweep()	63
9.1.4.66 splQSweepListStartSweep()	64
9.1.4.67 splQSweepListFinishSweep()	64
9.1.4.68 spGetAudio()	65
9.1.4.69 spGetGPSInfo()	65
9.1.4.70 spSetFanThreshold()	66
9.1.4.71 spGetFanSettings()	66
9.1.4.72 spGetErrorString()	67
9.1.4.73 spGetAPIVersion()	67
9.2 sp_api.h	68
Index	73

Chapter 1

SP145 API Reference

This documentation is a reference for the Signal Hound SP145 (SP) spectrum analyzer programming interface (API). The API provides a set of C functions for making measurements with the SP devices. The API is C ABI compatible making it 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.

- [Sweep Mode](#)
- [Real-Time Spectrum Analysis](#)
- [I/Q Streaming](#)
- [I/Q Sweep List / Frequency Hopping](#)
- [I/Q Full Band](#)

Also see [Basic API Usage](#) for more information.

1.3 Build/Version Notes

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/.so.

- Version 1.0.0 – Official release, support for SP145A

1.4 PC Requirements

1.4.1 Windows Development Requirements

- Windows 10/11 (Recommended)
- Windows 7/8 (Minimum)
- Windows C/C++ development tools and environment.
 - API was compiled using VS2012 and VS2019.
 - * VS2012/VS2019 C++ redistributables are required.
- Library files [sp_api.h](#), [sp_api.lib](#), and [sp_api.dll](#)

1.4.2 Linux Development Requirements

- Linux 64-bit
 - Ubuntu 18.04/20.04
 - CentOS 7
 - Red Hat 7
- libusb-1.0
- System GCC compiler
- SP library files, [sp_api.h](#) and [libsp_api.so](#)

1.4.3 Other Requirements

- SP145A
 - USB-C 3.0 connectivity provided through 4th generation or later Intel CPUs. 4th generation Intel CPU systems might require updating USB 3.0 drivers to operate properly.
 - (Recommended) Quad core Intel i5 or i7 processor, 4th generation or later.
 - (Minimum) Dual core Intel i5 or i7 processor, 4rd generation or later.

1.5 Reference Level and Sensitivity

There are two ways to set the sensitivity of the receiver, through the attenuator or the reference level. ([spSetAttenuator/spSetRefLevel](#)) The [spSetAttenuator](#) function allows direct control of the sensitivity. If the attenuator is set to auto, then the API chooses the best attenuator value based on the reference level selected. The attenuator is set to auto by default.

The reference level setting will automatically adjust the sensitivity to have the most dynamic range for signals at or near (~5dB) below the reference level. If you know the expected input signal level of your signal, setting the reference level to 5dB above your expected input will provide the most dynamic range. Using the reference level, you can also ensure the receiver does not experience an ADC overload by setting a reference level well above input signal level ranges.

The reference level parameter is the suggested method of controlling the receiver sensitivity.

1.6 GPS and Timestamps

The internal GPS communicates to the API on initialization, during all active measurements, and when requested through the [spGetGPSInfo](#) function. It does not perform active communication to the PC at any time other than these.

NMEA sentences are updated once per second and timestamps are updated every time the GPS has a chance to communicate with the PC. This means, several consecutive sweeps within a 1 second frame have the chance to update the NMEA information at most once, and provide a new timestamp for each sweep.

1.6.1 Acquiring GPS Lock

The GPS will automatically lock with no external assistance. You can query the state of the GPS lock with either the [spGetGPSSState](#) function, or by examining the return status of [spGetGPSInfo](#). From a cold start, expect a lock within the first few minutes. A warm or hot start should see a lock much quicker.

1.6.2 GPS Time Stamping

When the GPS is locked, I/Q data and sweep timestamping occurs using the internal GPS PPS signal and NMEA information. Once GPS lock is achieved, GPS timestamping occurs immediately and required no user intervention. Until the GPS is locked, timestamping occurs with the system clock, which has a typical accuracy of +/- 16ms.

If the GPS loses lock, the timestamps will advance at the nominal rate until the GPS achieves lock again. Off GPS lock timestamps will be coherent between measurement reconfigurations until the device is closed through the API or the device loses power.

1.6.3 GPS Disciplining

The system GPS can be in one of three states,

1. GPS unlocked – Either the GPS antenna is disconnected or is connected and hasn't achieved lock yet. After connecting the antenna expect several minutes for the lock. If you do not see a lock after several minutes, you might need to reposition the antenna.
2. GPS locked – The GPS has achieved lock. At this point measurement timestamps will have full accuracy and geolocation information can be queried.
3. GPS disciplined – The GPS has disciplined the timebase and is updating the holdover values. (See the Spike user manual for more information about GPS holdover values)

The current GPS state can be queried with [spGetGPSSState](#). If the device is actively making measurements the recommended way to wait for lock/discipline is by querying the GPS state after each measurement. If the device is idle (after an [spAbort](#)) the recommended method is to query the GPS state in a busy loop, preferably with a small wait between queries, something like 1 second is adequate. (careful! it may never break out of a loop if you break on lock detect and the SP cannot achieve it)

The GPS will lock automatically with a GPS antenna attached, but for the GPS to discipline the SP, it must first be enabled. To enable GPS disciplining, use the [spSetGPSTimebaseUpdate](#) function. Below is the state machine for GPS disciplining. To summarize, the timebase is adjusted by the newer of the two correction factors, either the last GPS holdover value or the last Signal Hound calibration value. Only after enabling the GPS disciplining will the SP utilize a GPS lock to discipline the SP and store holdover values.

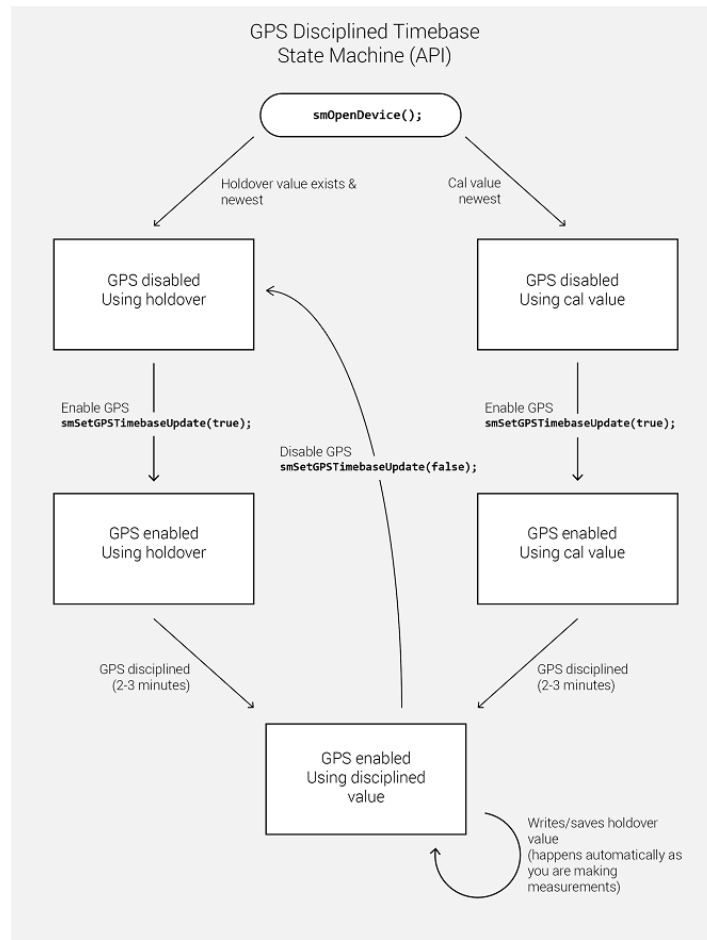


Figure 1.1 GPS Disciplining State Machine

1.7 Power States

The SP has 2 power states, on and standby. The device can be set to standby to save power either when the active measurement mode is idle or sweep mode (assuming no sweeps are currently active).

A short description of each power state is described below.

- [spPowerStateOn](#), Full power state. All circuitry is enabled. Power consumption is $\sim 30W$. The device is ready to make measurements.
- [spPowerStateStandby](#), Estimated power consumption $\sim 16W$. Some circuitry disabled. 100ms time to return to [spPowerStateOn](#).

1.8 Thread Safety

The SP 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.

1.9 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 of the device directly 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 is to use a named mutex on a Windows system.

If you wish to interface multiple devices on Linux, see the [Linux Notes](#).

1.10 Linux Notes

1.10.1 USB Throughput

By default, Linux applications cannot increase the priority of individual threads unless ran with elevated privilege (root). On Windows this issue does not exist, and the API will elevate the USB data acquisition threads to a higher priority to ensure USB data loss does not occur. On Linux, the user will need to run their application as root to ensure USB data acquisition is performed at a higher priority.

If this is not done, there is a higher risk of USB data loss for streaming modes such as I/Q, real-time, and fast sweep measurements on Linux.

In our testing, if little additional processing is occurring outside the API, 1 or 2 devices typically will not experience data loss due to this issue. Once the user application increases the processing load or starts performing I/O such as storing data to disk, the occurrence of USB data loss increases and the need to run the application as root increases.

1.10.2 Multiple USB Devices

There are system limitations when attempting to use multiple Signal Hound USB 3.0 devices* simultaneously on Linux operating systems. The default amount of memory allocated for USB transfers on Linux is 16MB. A single Signal Hound USB 3.0 device will stay within this allocation size, but two devices will exceed this limitation and can cause connection issues or will cause the software to crash.

The USB memory allocation size can be changed by writing to the file

```
/sys/module/usbcore/parameters/usbfs_memory_mb
```

A good value would be $N * 16$ where N is the number of devices you plan on interfacing simultaneously. One way to write to this file is with the command,

```
sudo sh -c `echo 32 > /sys/module/usbcore/parameters/usbfs_memory_mb`
```

where 32 can be replaced with any value you wish. You may need to restart the system for this change to take effect.

*Includes both Signal Hound USB 3.0 spectrum analyzers and signal generators.

1.11 Programming Languages Compatibility

The SP interface is C compatible which ensures it is possible to interface the API in most languages that can call C functions. These languages include C++, C#, Python, MATLAB, LabVIEW, Java, etc. Examples of calling the API in these other languages are included in the code examples folder.

The API consists of several enum types, which are often used as parameters. These values can be treated as 32-bit integers when calling the API functions from other programming languages. You will need to match the enumerated values defined in the API header file.

1.11.1 Python

This project enables the user to retrieve sampled I/Q data and frequency sweeps from the SP receiver through a number of convenience functions written in Python. These functions serve as a mostly one-to-one binding to the native API, and perform the majority of C shared library interfacing necessary to communicate with the receiver.

1.11.1.1 Prerequisites

Python 3 is required to use the Python interface functions. The API has been tested with Python 3.6.1 on Windows.

The Spike software must be fully installed before using the Python interface functions. Additionally, ensure the SP receiver is stable and functioning in Spike before using the Python interface.

Several SciPy and NumPy packages are required to use the Python interface functions, due to the need for fast array processing. The easiest way is to install the SciPy stack, available here: <https://www.scipy.org/install.html>. Alternatively, the necessary individual packages could be installed.

For our tests we used the 64-bit version of Anaconda for Windows.

1.11.1.2 Setup

1.11.1.2.1 Windows Place `sp_api.dll` into the `spdevice/` folder, along with `sp_api.py`.

To call the functions from outside the `spdevice/` directory you may need to add the `spdevice` folder to the Python search path and to the system path. This can be done by editing `PATH` and `PYTHONPATH` in: Control Panel > System and Security > System > Advanced system settings > Advanced > Environment Variables > System variables.

1.11.1.2.2 Linux Place `sp_api.so` into the `spdevice/` folder, along with `sp_api.py`.

Edit the line in `sp_api.py` from

```
splib = CDLL("spdevice/sp_api.dll")
to
splib = CDLL("spdevice/sp_api.so")
```

1.11.1.3 Usage

The functions under the "Public" heading are callable from external scripts. They are functionally equivalent to their C counterparts, except memory management is handled by the API instead of the user. Data is returned in Numpy arrays by the acquisition functions.

To run the example scripts, navigate to the folder containing the example .py files. Each example file is standalone and provides example code for calling the provided Python functions for the SP145.

1.12 I/Q Acquisition

This section describes several I/Q attributes common to many I/Q measurements.

1.12.1 I/Q Sample Rates

The table below outlines the available I/Q sample rates and corresponding decimations for SP devices. See the software filter limitations in the following section for more information about filtering and bandwidth.

Decimation	Native Rate (USB 3.0) MS/s	LTE Rate (USB 3.0) MS/s	Downsampling (All units)
1 (Minimum)	50	61.44	None
2	25	30.72	Hardware only
4	12.5	15.36	Hardware only
8	6.25	7.68	Hardware only
16	3.125	3.84	Hardware/Software
N = {32, 64, ...}	50 / N	61.44 / N	Hardware/Software
4096 (Maximum)	0.012207	0.015	Hardware/Software

1.12.2 I/Q Data Types

I/Q data can be returned either as 32-bit complex floats or 16-bit complex shorts depending on the data type set in [spSetIQDataType](#). 16-bit shorts are more memory efficient by a factor of 2 but require more effort to convert to absolute amplitudes and may be less convenient to work with.

When data is returned as 32-bit complex floats, the data is scaled to mW and the amplitude can be calculated by the following equation

$$\text{Sample Power (dBm)} = 10.0 * \log_{10}(\text{re}*\text{re} + \text{im}*\text{im});$$

Where **re** and **im** are the real and imaginary components of a single I/Q sample.

1.12.2.1 Converting From Full Scale to Corrected I/Q

When data is returned as 16-bit complex shorts, the data is full scale and a correction must be applied before you can measure mW or dBm. Values range from [-32768 to +32767]. To measure the power of a sample using the complex short data type, three steps are required.

1. Convert from short to float.

- `float re32f = ((float)re16s / 32768.0);`
- `float im32f = ((float)im16s / 32768.0);`
 - This converts the short to a float in the range of [-1.0 to +1.0]

2. Scale the floats by the correction value returned from [spGetIQCorrection](#).

- `re32f *= correction;`
- `im32f *= correction;`

3. Calculate power

- `Sample Power (dBm) = 10.0 * log10(re32f*re32f + im32f*im32f);`

1.12.3 I/Q Filtering and Bandwidth

The user can enable a baseband software filter on the I/Q data with a selectable bandwidth. If the software filter is disabled, the signal will only have been filtered by the hardware as described below.

The hardware uses several half-band filters to accomplish decimations 2, 4, and 8 and there is non-negligible aliasing between 0.8 and 1.0 of the sample rates. Software filtering will eliminate this aliasing at the cost of a slightly smaller cutoff frequency.

Most users will want to enable the software IF filter for better rejection in the stop band, as well as the convenience of a selectable IF bandwidth. Users may forgo the software filter to reduce CPU load on the PC or if custom signal conditioning is performed.

Software filtering is enabled by default for decimations greater than 8.

The table below shows the maximum available bandwidth with the filter disabled and the maximum bandwidth allowed with the filter enabled. These numbers apply for both base samples rates.

Decimation	Usable Bandwidth (MHz) Filter Disabled	Max Bandwidth (MHz) Filter Enabled
1	41.5	41.5
2	20	19.2
4	10	9.6
8	5	4.8
16	2.5	2.4
32	1.25	1.2
64	0.625	0.6
128	0.3125	0.3
256	0.15625	0.15
512	0.078125	0.075
1024	0.039063	0.0375
2048	0.019531	0.01875
4096	0.009766	0.009375

1.13 Estimating Sweep Size

It is useful to understand the relationship between sweep parameters and sweep size. It is not possible to directly calculate the sweep size of a given configuration beforehand, but it is possible to estimate the sweep size to within a power of 2.

The equation that can be used to estimate sweep size is

$$\text{Sweep Size (est.)} = (\text{Span} * \text{WindowBW}) / \text{RBW}$$

Where span and RBW are specified in Hz, and window bandwidth is specified in bins. Window bandwidth is the noise bandwidth of the FFT window function used. See the [Window Functions](#) section for more information.

1.14 Window Functions

Below are the window functions used in the API. The API uses zero-padding to achieve the requested RBW so the noise bandwidth in this table should not be directly used.

Window	NoiseBandwidth (bins)	Notes
Flat-Top	3.77	SRS flattop
Nuttall	2.02	None
Kaiser	1.79	$\alpha = 3$
Blackman	1.73	$\alpha = 0.16$
Chebyshev	1.94	$\alpha = 5$
Hamming	1.36	$\alpha = 0.54, \beta = 0.46$
Gaussian6dB	2.64	$\sigma = 0.1$

1.15 Automatic GPS Timebase Discipline

When enabled, the API will instruct the receiver to use the internal GPS PPS to discipline the 10MHz internal timebase. This disciplining process adjusts a tuning voltage which the API will then store on the PC filesystem. This stored tuning voltage will then be used by the API in the future to tune the timebase. This allows the receiver to reuse a good GPS frequency lock even when no GPS antenna is attached.

Note: The stored GPS tuning voltage will override the tuning voltage created during calibration, and in almost all cases this is preferred as the latest GPS discipline will be the best frequency tune.

The GPS tuning voltage is stored in the ProgramData/ folder at

C:\ProgramData\SignalHound\cal_files\sp#####gps.bin

where the # is the device serial number. Delete this file to have the API revert to using the internally stored frequency calibration.

Disable the automatic GPS timebase update to bypass this functionality with the [spSetGPSTimebaseUpdate](#) function.

1.16 Contact Information

For technical support, email support@signalhound.com.

For sales, email sales@signalhound.com.

Chapter 2

Basic API Usage

Any application using the SP API will follow these steps to interact and perform measurements on the device.

1. Open the device and receive a handle to the device resources.
2. Configure the device.
3. Acquire measurements.
4. Stop acquisitions, abort the current operation.
5. Close the device.
6. (Recalibration)

2.1 Opening a Device

Opening a device is done through the [spOpenDevice](#) or [spOpenDeviceBySerial](#) functions. These functions will perform the full initialization of the device and if successful, will return an integer handle which can be used to reference the device for the remainder of your program. See the list of all SP devices connected to the PC via the [spGetDeviceList](#) function.

2.2 Configuring the Device

Once the device is open, the next step is to configure the device for a measurement. The available measurement modes are listed on the [mainpage](#). Each mode has specific configurations routines, which set a temporary configuration state. Once all configuration routines have been called, calling the [spConfigure](#) function copies the temporary configuration state into the active measurement state and the device is ready for measurements. The provided code examples showcase how to configure the device for each measurement mode.

2.3 Acquiring Measurements

After the device has been successfully configured, the API provides several functions for acquiring measurements. Only certain measurements are available depending on the active measurement mode. For example, I/Q data acquisition is not available when the device is in a sweep measurement mode.

2.3.1 Stopping the Measurements

Stopping all measurements is achieved through the [spAbort](#) function. This causes the device to cancel or finish any pending operations and return to an idle state. Calling [spAbort](#) is never required, as it is called by default if you attempt to change the measurement mode or close the device, but it can be useful to do this.

- Certain measurement modes can consume large amounts of resources such as memory and CPU usage. Returning to an idle state will free those resources.
- Returning to an idle state will help reduce power consumption.

2.4 Closing the Device

When finished making measurements, you can close the device and free all resources related to the device with the [spCloseDevice](#) function. Once closed, the device will appear in the open device list again. It is possible to open and close a device multiple times during the execution of a program.

2.5 Recalibration

Recalibration is performed each time the device is reconfigured ([spConfigure](#)). For instance, when the device is configured for I/Q streaming, the instrument and measurement is calibrated for the current environment and will not be calibrated again until the device measurement is aborted and started again (read: the device will not recalibrate in the middle of measurements, as this would interrupt measurements such as I/Q streaming or real-time analysis).

Large temperature changes affect measurements the most, and it is recommended to reconfigure the device once a large temperature delta has been recorded.

It is recommended to use the temperature from the [spGetDeviceDiagnostics](#) function to detect a temperature drift and recalibrate again when you see a drift of 2-4 degrees Celsius.

Chapter 3

Sweep Mode

Sweep mode represents the common spectrum analyzer measurement of plotting amplitude over frequency. The API provides a simple interface through [spGetSweep](#) for acquiring single sweeps.

3.1 Example

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

3.2 Basics

Only 1 sweep configuration can be active at a time.

Changing a sweep setting requires reconfiguring the device with a new sweep configuration.

All sweeps must be finished to change sweep configuration.

To achieve a sustained 1THz/s sweep speed, use fast sweep speed and queued sweeps.

Only linear spaced sweeps can be performed.

3.3 Sweep Format

A sweep is returned from the API as a 1-dimensional array of measurement values. Each element in the array corresponds to a specific frequency. The frequency of any given element can be calculated as

$$\text{Frequency of N'th element in sweep} = \text{StartFreq} + N * \text{BinSize}$$

where `StartFreq` and `BinSize` are reported in the [spGetSweepParameters](#) function.

The measurement values can be returned in dBm or mV units.

3.4 Min and Max Sweep Arrays

All sweep functions in the API return 2 separate sweep arrays. The parameters are typically named `sweepMin` and `sweepMax`. To understand the purpose of these arrays, it is important to understand their relation to the analyzer's detector setting. Traditionally, spectrum analyzers offer several detector settings, the most common being peak-, peak+, and average. The API reduces this to either minmax or average. When the detector is set to minmax, the `sweepMin` array will contain the sweep as if a peak- detector is running, and the `sweepMax` array will contain the sweep of a peak+ detector. When average detector is enabled, `sweepMin` and `sweepMax` will be identical arrays and will be the result of an average detector.

If you are not interested in one of the sweeps, you can pass a NULL pointer for this parameter.

Most users will be interested in the `sweepMax` array as it will provide you either the peak+ and average detector results depending on detector setting. In this case, pass NULL for the `sweepMin` parameter.

Chapter 4

Real-Time Spectrum Analysis

Real-time spectrum analysis allows you to perform continuous, gap free spectrum analysis on bandwidths up to 160MHz. This provides you with the ability to detect short transient signals down to 3us in length.

4.1 Example

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

4.2 Basics

Real-time spectrum analysis is a frequency domain measurement. For time domain measurements see [I/Q Streaming](#).

RBW directly affects the 100% POI of signals in real-time mode.

Real-time spectrum analysis returns a sweep, frame, and alphaFrame from the [spGetRealTimeFrame](#) function. These are described in the sections below.

The real-time measurement is performed over short consecutive time periods and returned to the user as a sweep and frame representing spectrum activity over that time period. The duration of these time periods is ~33ms. This means you will receive ~30 sweep/frame pairings per second.

Once the measurement is initialized via [spConfigure](#), the API is continuously generating sweeps and frames for retrieval. The API can buffer ~1 second worth of past measurements. It is the responsibility of the user to request sweeps/frames at a rate that prevents the accumulation of measurements in the API.

Real-time spectrum analysis is accomplished using 50% overlapping FFTs with zero-padding to accomplish arbitrary RBWs. Spans above 40MHz utilize the FPGA to perform this processing which limits the RBW to 30kHz when using the Nuttall window. Spans 40MHz and below are processed on the PC and lower RBWs can be set.

4.3 Real-Time Sweep

The sweeps returned in real-time spectrum analysis are the result of applying the detector over all FFTs that occur during the measurement period. The min/max detector will return the peak-/peak+ sweeps. The average detector will return the averaged sweep over that time period.

When average detector is selected, both sweepMin and sweepMax return identical sweeps and one of them can be ignored.

4.4 Real-Time Frame

The frame is a 2-dimensional grid representing frequency on the x-axis and amplitude levels on the y-axis. Each index in the grid is the percentage of time the signal persisted at this frequency and amplitude. If a signal existed at this location for the full duration of the frame, the percentage will be close to 1.0. An index which contains the value 0.0 infers that no spectrum activity occurred at that location during the frame acquisition.

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 alpha frame is useful to determine how recent the activity in the frame is and useful for plotting the frames.

The sweep size is always an integer multiple of the frame width, which means the bin size of the frame is easily calculated. The vertical spacing can be calculated using the frame height, reference level, and frame scale (specified by the user in dB).

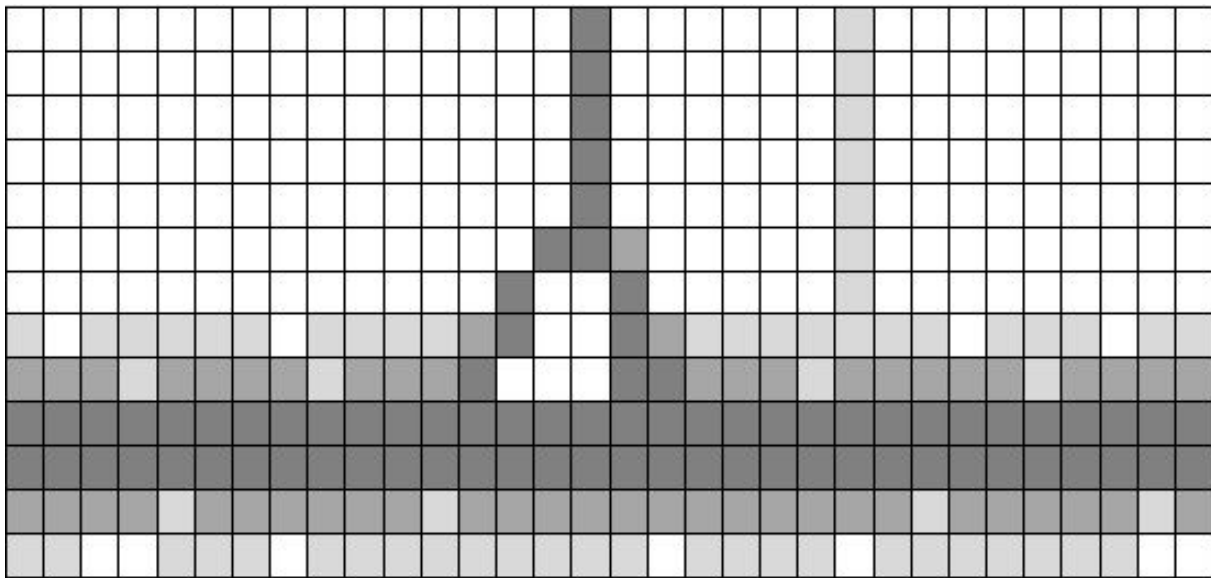


Figure 4.1 An example of a frame plotted as a gray scale image, mapping the density values between [0.0,1.0] to gray scale values between [0,255]. The frame shows a persistent CW signal near the center frequency and a short-lived CW signal.

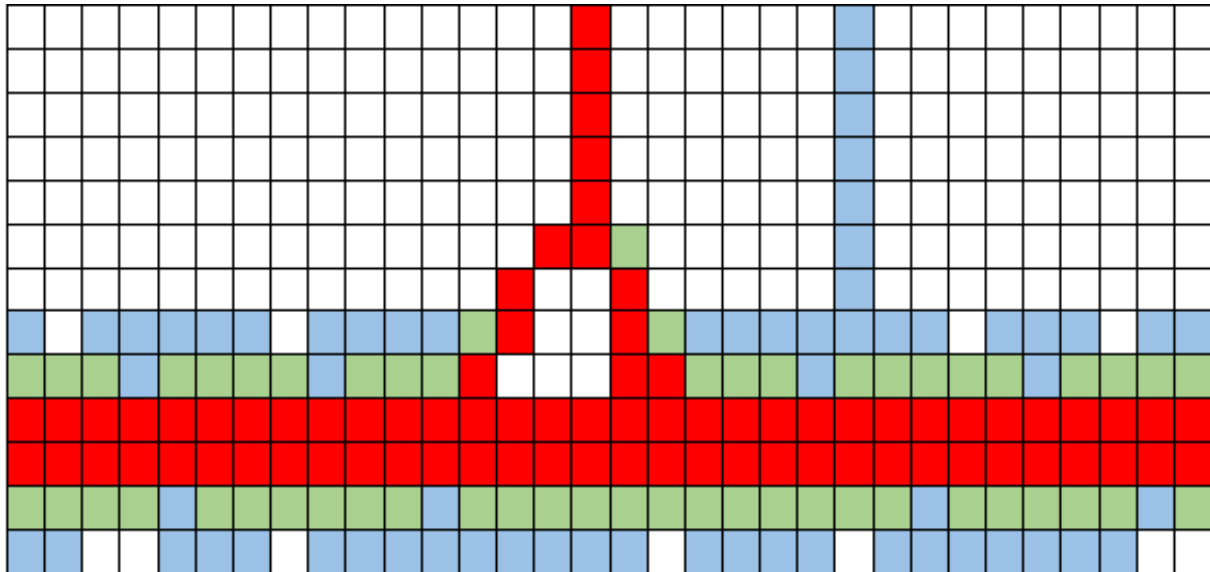


Figure 4.2 The same frame above as is plotted in Spike, where density values are mapped onto a color spectrum.

4.5 RBW Restrictions

The real-time span determines the minimum and maximum RBW.

Span	Minimum RBW (Nuttall window)	Maximum RBW (Nuttall window)
(> 40MHz)	30 kHz	1 MHz
(< 40MHz)	1.5 kHz	800 kHz

Chapter 5

I/Q Streaming

The I/Q streaming mode is used to stream continuous I/Q samples at a given center frequency. The sample rate, bandwidth, center frequency, and data type can be configured. If you need to capture I/Q data at many frequencies or don't need continuous streaming capabilities, consider using the [I/Q Sweep List / Frequency Hopping](#) measurements.

5.1 Example

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

5.2 Basics

The API provides the ability to stream I/Q samples up to the device's native sample rate or common LTE sample rates. See [I/Q Sample Rates](#) for more information.

I/Q data can be retrieved as 32-bit complex floats or 16-bit complex shorts. See [I/Q Data Types](#) for more information.

5.3 Sample Rate, Decimation, and Bandwidth

The I/Q data stream can be decimated by powers of 2 between 1 and 4096, starting at either the native sample rate or an LTE sample rate. Filtering is performed at each decimation stage. The final filter cutoff frequency is user selectable.

For decimations [1,2,4,8], custom cutoff frequencies are accomplished with a PC side lowpass filter. The PC software filter is optional for decimations between 1 and 8. If the software filter is disabled the FPGA half band filters are the only alias filter used for these decimation stages and there will be aliased signals in the roll off regions of the I/Q bandwidth. Disabling the software filter will reduce CPU load of the I/Q data stream at the cost of this aliasing.

Using decimations greater than 8, decimation and filtering occur entirely on the PC. The cutoff frequency of the filter must obey the Nyquist frequency for the selected sample rate. The downsample filter sizes cannot be changed and thus the roll off transition region is a fixed size for each decimation setting.

5.4 Polling Interface (I/Q)

The API for the I/Q data stream is a polling style interface, where the application must request I/Q data in blocks that will keep up with the device acquisition of data. The API's internal circular buffer can store up to 1/2 second worth of I/Q data before data loss occurs. It is the responsibility of the user's application to poll the I/Q data fast enough that data loss does not occur.

5.5 External Triggering

External trigger information can be retrieved when I/Q streaming. Trigger information is provided through the triggers buffer in the [spGetIQ](#) function.

If a trigger buffer is provided to [spGetIQ](#), any external trigger events seen during the acquisition of the returned I/Q data will be placed in the trigger buffer. External trigger events are returned as indices into the I/Q data at which the trigger event occurred. For example, if 1000 I/Q samples are requested and a trigger buffer of size 3 is provided, and the function returns with the trigger buffer set to [12,300,876], this indicates that an external trigger event occurred at I/Q sample index 12, 300, and 876 in the I/Q data returned from this function call.

If fewer external triggers were seen during the I/Q acquisition than the size of the trigger buffer provided, the remainder of the trigger buffer is set to the sentinel value. The default sentinel value is 0.0, so for example, if a trigger buffer of size 3 is provided, and only a single trigger event was seen, the trigger buffer will return [N, 0.0, 0.0] where N is the single trigger index returned.

If more trigger events were seen during the I/Q acquisition than the size of the trigger buffer, those trigger events that cannot fit in the buffer are discarded.

Triggers are provided as doubles and non-integer values can indicate the trigger occurred in between 2 samples. This can occur when performing decimation, as the triggers are recorded at a much higher resolution than the final sample rate.

A note on trigger sentinel values, the default sentinel value of 0.0 does not allow the detection of triggers occurring at the first sample point. If this is an issue, set the sentinel value to -1.0 or some other negative value which cannot be normally returned. The default value of 0.0 is the result of historical choices and will remain the default value.

5.6 Additional Information

See [I/Q Acquisition](#) for more information.

Chapter 6

I/Q Sweep List / Frequency Hopping

I/Q sweep list measurements perform frequency hopping I/Q captures at a list of preconfigured frequencies and capture sizes. Captures can be queued to sustain > 8000 frequency hops per second.

6.1 Example

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

6.2 Basics

I/Q sweep lists are finite length I/Q acquisitions across a series of frequencies. Lists of up to 1200 frequencies can be provided. At each frequency, the reference level and number of samples to be collected must be configured. One measurement/list is referred to as a “sweep” and iterates through all configured frequency steps. Several sweeps can be queued to maintain maximum throughput.

I/Q sweep lists are advantageous when needing to acquire a discrete number of I/Q samples at several different frequencies. I/Q samples are collected at the device's native sample rate, 50MS/s. The absolute fastest the SP device can switch frequencies is 120us. When I/Q capture amounts are small at each frequency, 120us frequency switch times can be achieved for a maximum of 8333.33 frequencies per second.

At each frequency, a timestamp is provided indicating the nanoseconds since epoch for the first I/Q sample at that frequency. If the internal GPS is locked, this time is GPS time, If GPS is not locked, system time is provided. Regardless of GPS lock, relative timings between timestamps are highly accurate through use of internal device counters.

6.2.1 Sweep List Configuration Example

A list of 3 frequencies is provided, 1GHz, 2GHz, and 3GHz. At each frequency 1000 I/Q samples are configured to be collected. Once configured a sweep can be performed which captures I/Q samples at the 3 frequencies, for a total of 3000 samples. If desired, N sweeps can be queued to be performed back-to-back, resulting in $N * 3000$ samples to be collected. By queuing the sweeps, blind time between sweeps is reduced or eliminated, improving probability of intercept and overall measurement speed.

6.3 Notes on Performance

While the user can specify an arbitrary number of samples at each frequency, the SP device is internally limited to multiples of 2048 samples. For this reason, it is optimum to round up to the next multiple of 2048, which will not affect acquisition speed and reduce the number samples discarded.

Maximum sweep speed occurs when at most N samples are requested at each frequency, where N is 2048 samples. When $\leq N$ samples are requested, the device will step at the maximum rate of 8333.33 frequencies per second. This equates to $\sim 333\text{GHz}$ of spectrum coverage per second.

6.4 I/Q Streaming vs I/Q Sweep List

One use case where I/Q sweep lists are preferred to I/Q streaming for single frequency measurements is when you know in advance how many I/Q samples you want to collect at that frequency. Using I/Q sweep lists to acquire these samples has less overhead than using I/Q streaming. Starting and stopping the I/Q stream can take $\sim 30\text{ms}$, where as the overhead associated with performing a single I/Q sweep list acquisition is 1-5ms.

6.5 Additional Information

See [I/Q Acquisition](#) for more information.

Chapter 7

I/Q Full Band

Full band I/Q captures are a special measurement mode that allow users to capture short acquisitions at the full base band rate. Up to 32k samples can be captured at the 500MS/s baseband rate, representing a $\sim 65\mu\text{s}$ capture length. Captures can be external or video triggered (with the right configuration). The measurement can also be swept (a sequence of captures at different frequencies). Swept captures cannot be triggered and are performed by stepping the LO and performing an I/Q acquisition at each frequency step. The frequency can be tuned in 39.0625MHz steps (the native hardware resolution). The capture is AC coupled, and will exhibit a notch in the center of the baseband capture. Sweeps can cover $> 1\text{THz}$ per second worth of spectrum.

Examples of full band I/Q captures can be found in the examples folder in the SDK.

Chapter 8

File Index

8.1 File List

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

sp_api.h	API functions for the SP145 spectrum analyzer	27
--------------------------	---	--------------------

Chapter 9

File Documentation

9.1 sp_api.h File Reference

API functions for the SP145 spectrum analyzer.

Macros

- `#define SP_TRUE (1)`
- `#define SP_FALSE (0)`
- `#define SP_MAX_DEVICES (9)`
- `#define SP_MAX_REF_LEVEL (20.0)`
- `#define SP_AUTO_ATTEN (-1)`
- `#define SP_MAX_ATTEN (6)`
- `#define SP_MIN_FREQ (9.0e3)`
- `#define SP_MAX_FREQ (15.0e9)`
- `#define SP_MIN_SWEEP_TIME (1.0e-6)`
- `#define SP_MAX_SWEEP_TIME (100.0)`
- `#define SP_REAL_TIME_MIN_SPAN (200.0e3)`
- `#define SP_REAL_TIME_MAX_SPAN (40.0e6)`
- `#define SP_REAL_TIME_MIN_RBW (2.0e3)`
- `#define SP_REAL_TIME_MAX_RBW (1.0e6)`
- `#define SP_MAX_IQ_DECIMATION (8192)`
- `#define SP_MAX_SWEEP_QUEUE_SZ (16)`

Enumerations

- `enum SpStatus {`
 `spDeviceNotFoundErr = -100 , spAllocationErr , spTransferErr , spMaxDevicesConnectedErr ,`
 `spInvalidParameterErr , spInvalidDeviceErr , spNullPtrErr , spConnectionLostErr ,`
 `spInvalidConfigurationErr , spSweepAlreadyActiveErr , spFileIOErr , spFx3RunErr ,`
 `spBootErr , spInvalidSweepPositionErr , spInvalidCalDataErr , spNoError = 0 ,`
 `spSettingClamped = 1 , spTempDrift = 2 , spADCOverflow = 3 , spUncalData = 4 ,`
 `spCPULimited = 5 , spInvalidCalData }`
- `enum SpDeviceType { spDeviceTypeSP145A = 0 }`
- `enum SpBool { spFalse = 0 , spTrue = 1 }`
- `enum SpPowerState { spPowerStateOn = 0 , spPowerStateStandby = 1 }`

- enum `SpMode` {
`spModeIdle` = 0 , `spModeSweeping` = 1 , `spModeRealTime` = 2 , `spModeIQStreaming` = 3 ,
`spModeIQSweepList` = 4 , `spModeAudio` = 5 }
- enum `SpDetector` { `spDetectorAverage` = 0 , `spDetectorMinMax` = 1 }
- enum `SpScale` { `spScaleLog` = 0 , `spScaleLin` = 1 , `spScaleFullScale` = 2 }
- enum `SpVideoUnits` { `spVideoLog` = 0 , `spVideoVoltage` = 1 , `spVideoPower` = 2 , `spVideoSample` = 3 }
- enum `SpWindowType` {
`spWindowFlatTop` = 0 , `spWindowNutall` = 1 , `spWindowBlackman` = 2 , `spWindowHamming` = 3 ,
`spWindowGaussian6dB` = 4 , `spWindowRect` = 5 }
- enum `SpDataType` { `spDataType32fc` = 0 , `spDataType16sc` = 1 }
- enum `SpTriggerEdge` { `spTriggerEdgeRising` = 0 , `spTriggerEdgeFalling` = 1 }
- enum `SpGPSSState` { `spGPSSStateNotPresent` = 0 , `spGPSSStateLocked` = 1 , `spGPSSStateDisciplined` = 2 }
- enum `SpAudioType` {
`spAudioTypeAM` = 0 , `spAudioTypeFM` = 1 , `spAudioTypeUSB` = 2 , `spAudioTypeLSB` = 3 ,
`spAudioTypeCW` = 4 }

Functions

- `SP_API SpStatus spGetDeviceList` (int *serials, int *deviceCount)
- `SP_API SpStatus spOpenDevice` (int *device)
- `SP_API SpStatus spOpenDeviceBySerial` (int *device, int serialNumber)
- `SP_API SpStatus spCloseDevice` (int device)
- `SP_API SpStatus spPresetDevice` (int device)
- `SP_API SpStatus spSetPowerState` (int device, `SpPowerState` powerState)
- `SP_API SpStatus spGetPowerState` (int device, `SpPowerState` *powerState)
- `SP_API SpStatus spGetSerialNumber` (int device, int *serialNumber)
- `SP_API SpStatus spGetFirmwareVersion` (int device, int *major, int *minor, int *rev)
- `SP_API SpStatus spGetDeviceDiagnostics` (int device, float *voltage, float *current, float *temperature)
- `SP_API SpStatus spGetCalDate` (int device, uint32_t *lastCalDate)
- `SP_API SpStatus spSetExtReference` (int device, `SpBool` enabled)
- `SP_API SpStatus spGetExtReference` (int device, `SpBool` *enabled)
- `SP_API SpStatus spSetGPSTimebaseUpdate` (int device, `SpBool` enabled)
- `SP_API SpStatus spGetGPSTimebaseUpdate` (int device, `SpBool` *enabled)
- `SP_API SpStatus spGetGPSHoldoverInfo` (int device, `SpBool` *usingGPSHoldover, uint32_t *lastHoldover←
Time)
- `SP_API SpStatus spGetGPSSState` (int device, `SpGPSSState` *GPSSState)
- `SP_API SpStatus spSetRefLevel` (int device, double refLevel)
- `SP_API SpStatus spGetRefLevel` (int device, double *refLevel)
- `SP_API SpStatus spSetAttenuator` (int device, int atten)
- `SP_API SpStatus spGetAttenuator` (int device, int *atten)
- `SP_API SpStatus spSetSweepCenterSpan` (int device, double centerFreqHz, double spanHz)
- `SP_API SpStatus spSetSweepStartStop` (int device, double startFreqHz, double stopFreqHz)
- `SP_API SpStatus spSetSweepCoupling` (int device, double rbw, double vbw, double sweepTime)
- `SP_API SpStatus spSetSweepDetector` (int device, `SpDetector` detector, `SpVideoUnits` videoUnits)
- `SP_API SpStatus spSetSweepScale` (int device, `SpScale` scale)
- `SP_API SpStatus spSetSweepWindow` (int device, `SpWindowType` window)
- `SP_API SpStatus spSetRealTimeCenterSpan` (int device, double centerFreqHz, double spanHz)
- `SP_API SpStatus spSetRealTimeRBW` (int device, double rbw)
- `SP_API SpStatus spSetRealTimeDetector` (int device, `SpDetector` detector)
- `SP_API SpStatus spSetRealTimeScale` (int device, `SpScale` scale, double frameRef, double frameScale)
- `SP_API SpStatus spSetRealTimeWindow` (int device, `SpWindowType` window)
- `SP_API SpStatus spSetIQDataType` (int device, `SpDataType` dataType)
- `SP_API SpStatus spSetIQCenterFreq` (int device, double centerFreqHz)
- `SP_API SpStatus spGetIQCenterFreq` (int device, double *centerFreqHz)

- SP_API [SpStatus spSetIQSampleRate](#) (int device, int decimation)
- SP_API [SpStatus spSetIQSoftwareFilter](#) (int device, [SpBool](#) enabled)
- SP_API [SpStatus spSetIQBandwidth](#) (int device, double bandwidth)
- SP_API [SpStatus spSetIQExtTriggerEdge](#) (int device, [SpTriggerEdge](#) edge)
- SP_API [SpStatus spSetIQTriggerSentinel](#) (double sentinelValue)
- SP_API [SpStatus spSetIQQueueSize](#) (int device, float ms)
- SP_API [SpStatus spSetIQSweepListDataType](#) (int device, [SpDataType](#) dataType)
- SP_API [SpStatus spSetIQSweepListCorrected](#) (int device, [SpBool](#) corrected)
- SP_API [SpStatus spSetIQSweepListSteps](#) (int device, int steps)
- SP_API [SpStatus spGetIQSweepListSteps](#) (int device, int *steps)
- SP_API [SpStatus spSetIQSweepListFreq](#) (int device, int step, double freq)
- SP_API [SpStatus spSetIQSweepListRef](#) (int device, int step, double level)
- SP_API [SpStatus spSetIQSweepListAtten](#) (int device, int step, int atten)
- SP_API [SpStatus spSetIQSweepListSampleCount](#) (int device, int step, uint32_t samples)
- SP_API [SpStatus spSetAudioCenterFreq](#) (int device, double centerFreqHz)
- SP_API [SpStatus spSetAudioType](#) (int device, [SpAudioType](#) audioType)
- SP_API [SpStatus spSetAudioFilters](#) (int device, double ifBandwidth, double audioLpf, double audioHpf)
- SP_API [SpStatus spSetAudioFMDeemphasis](#) (int device, double deemphasis)
- SP_API [SpStatus spConfigure](#) (int device, [SpMode](#) mode)
- SP_API [SpStatus spGetCurrentMode](#) (int device, [SpMode](#) *mode)
- SP_API [SpStatus spAbort](#) (int device)
- SP_API [SpStatus spGetSweepParameters](#) (int device, double *actualRBW, double *actualVBW, double *actualStartFreq, double *binSize, int *sweepSize)
- SP_API [SpStatus spGetRealTimeParameters](#) (int device, double *actualRBW, int *sweepSize, double *actualStartFreq, double *binSize, int *frameWidth, int *frameHeight, double *poi)
- SP_API [SpStatus spGetIQParameters](#) (int device, double *sampleRate, double *bandwidth)
- SP_API [SpStatus spGetIQCorrection](#) (int device, float *scale)
- SP_API [SpStatus spIQSweepListGetCorrections](#) (int device, float *corrections)
- SP_API [SpStatus spGetSweep](#) (int device, float *sweepMin, float *sweepMax, int64_t *nsSinceEpoch)
- SP_API [SpStatus spGetRealTimeFrame](#) (int device, float *colorFrame, float *alphaFrame, float *sweepMin, float *sweepMax, int *frameCount, int64_t *nsSinceEpoch)
- SP_API [SpStatus spGetIQ](#) (int device, void *iqBuf, int iqBufSize, double *triggers, int triggerBufSize, int64_t *nsSinceEpoch, [SpBool](#) purge, int *sampleLoss, int *samplesRemaining)
- SP_API [SpStatus spIQSweepListGetSweep](#) (int device, void *dst, int64_t *timestamps)
- SP_API [SpStatus spIQSweepListStartSweep](#) (int device, int pos, void *dst, int64_t *timestamps)
- SP_API [SpStatus spIQSweepListFinishSweep](#) (int device, int pos)
- SP_API [SpStatus spGetAudio](#) (int device, float *audio)
- SP_API [SpStatus spGetGPSInfo](#) (int device, [SpBool](#) refresh, [SpBool](#) *updated, int64_t *secSinceEpoch, double *latitude, double *longitude, double *altitude, char *nmea, int *nmeaLen)
- SP_API [SpStatus spSetFanThreshold](#) (int device, float temp)
- SP_API [SpStatus spGetFanSettings](#) (int device, float *threshold, float *voltage)
- SP_API const char * [spGetErrorString](#) ([SpStatus](#) status)
- SP_API const char * [spGetAPIVersion](#) ()

9.1.1 Detailed Description

API functions for the SP145 spectrum analyzer.

This is the main file for user-accessible functions for controlling the SP145 spectrum analyzer.

9.1.2 Macro Definition Documentation

9.1.2.1 SP_TRUE

```
#define SP_TRUE (1)
```

Used for boolean true when integer parameters are being used. Also see [SpBool](#).

9.1.2.2 SP_FALSE

```
#define SP_FALSE (0)
```

Used for boolean false when integer parameters are being used. Also see [SpBool](#).

9.1.2.3 SP_MAX_DEVICES

```
#define SP_MAX_DEVICES (9)
```

Max number of devices that can be interfaced in the API.

9.1.2.4 SP_MAX_REF_LEVEL

```
#define SP_MAX_REF_LEVEL (20.0)
```

Maximum reference level in dBm

9.1.2.5 SP_AUTO_ATTEN

```
#define SP_AUTO_ATTEN (-1)
```

Tells the API to automatically choose attenuation based on reference level.

9.1.2.6 SP_MAX_ATTEN

```
#define SP_MAX_ATTEN (6)
```

Valid atten values [0,6] or -1 for auto

9.1.2.7 SP_MIN_FREQ

```
#define SP_MIN_FREQ (9.0e3)
```

Min frequency for sweeps, and min center frequency for I/Q measurements.

9.1.2.8 SP_MAX_FREQ

```
#define SP_MAX_FREQ (15.0e9)
```

Max frequency for sweeps, and max center frequency for I/Q measurements.

9.1.2.9 SP_MIN_SWEEP_TIME

```
#define SP_MIN_SWEEP_TIME (1.0e-6)
```

Min sweep time in seconds. See [spSetSweepCoupling](#).

9.1.2.10 SP_MAX_SWEEP_TIME

```
#define SP_MAX_SWEEP_TIME (100.0)
```

Max sweep time in seconds. See [spSetSweepCoupling](#).

9.1.2.11 SP_REAL_TIME_MIN_SPAN

```
#define SP_REAL_TIME_MIN_SPAN (200.0e3)
```

Min span for device configured in real-time measurement mode

9.1.2.12 SP_REAL_TIME_MAX_SPAN

```
#define SP_REAL_TIME_MAX_SPAN (40.0e6)
```

Max span for device configured in real-time measurement mode

9.1.2.13 SP_REAL_TIME_MIN_RBW

```
#define SP_REAL_TIME_MIN_RBW (2.0e3)
```

Min RBW for device configured in real-time measurement mode

9.1.2.14 SP_REAL_TIME_MAX_RBW

```
#define SP_REAL_TIME_MAX_RBW (1.0e6)
```

Max RBW for device configured in real-time measurement mode

9.1.2.15 SP_MAX_IQ_DECIMATION

```
#define SP_MAX_IQ_DECIMATION (8192)
```

Max decimation for I/Q streaming.

9.1.2.16 SP_MAX_SWEEP_QUEUE_SZ

```
#define SP_MAX_SWEEP_QUEUE_SZ (16)
```

Maximum number of sweeps that can be queued up. Valid sweep indices between [0,15]

9.1.3 Enumeration Type Documentation

9.1.3.1 SpStatus

```
enum SpStatus
```

Status code returned from all SP API functions.

Enumerator

spDeviceNotFoundErr	Unable to open device
spAllocationErr	Unable to allocate resources needed to configure the measurement mode
spMaxDevicesConnectedErr	Only can connect up to SP_MAX_DEVICES receivers
spInvalidParameterErr	Required parameter found to have invalid value
spInvalidDeviceErr	User specified invalid device index
spNullPtrErr	One or more required pointer parameters were null
spConnectionLostErr	Device disconnected, likely USB error detected
spInvalidConfigurationErr	Attempting to perform an operation that cannot currently be performed. Often the result of trying to do something while the device is currently making measurements or not in an idle state.
spFx3RunErr	If the core FX3 program fails to run
spBootErr	Boot error
spInvalidSweepPositionErr	Invalid or already active sweep position
spNoError	Function returned successfully
spSettingClamped	One or more of the provided settings were adjusted
spTempDrift	Temperature drift occurred, measurements uncalibrated, reconfigure the device
spADCOverflow	Measurement includes data which caused an ADC overload (clipping/compression)
spUncalData	Measurement is uncalibrated, overrides ADC overflow
spCPULimited	Returned when the API was unable to keep up with the necessary processing
spInvalidCalData	Calibration data potentially corrupt

9.1.3.2 SpDeviceType

```
enum SpDeviceType
```

Device type

Enumerator

spDeviceTypeSP145A	SP145A
--------------------	--------

9.1.3.3 SpBool

```
enum SpBool
```

Boolean type. Used in public facing functions instead of `bool` to improve API use from different programming languages.

Enumerator

spFalse	False
spTrue	True

9.1.3.4 SpPowerState

enum [SpPowerState](#)

Specifies device power state. See [Power States](#) for more information.

Enumerator

spPowerStateOn	On
spPowerStateStandby	Standby

9.1.3.5 SpMode

enum [SpMode](#)

Measurement mode

Enumerator

spModelIdle	Idle, no measurement
spModeSweeping	Swept spectrum analysis
spModeRealTime	Real-time spectrum analysis
spModelQStreaming	I/Q streaming
spModelQSweepList	I/Q sweep list / frequency hopping
spModeAudio	Audio demod

9.1.3.6 SpDetector

enum [SpDetector](#)

Detector used for sweep and real-time spectrum analysis.

Enumerator

spDetectorAverage	Average
spDetectorMinMax	Min/Max

9.1.3.7 SpScale

enum [SpScale](#)

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

Enumerator

spScaleLog	dBm
spScaleLin	mV
spScaleFullScale	Log scale, no corrections

9.1.3.8 SpVideoUnits

enum [SpVideoUnits](#)

Specifies units in which VBW processing occurs.

Enumerator

spVideoLog	dBm
spVideoVoltage	Linear voltage
spVideoPower	Linear power
spVideoSample	No VBW processing

9.1.3.9 SpWindowType

enum [SpWindowType](#)

Specifies the window used for sweep and real-time analysis.

Enumerator

spWindowFlatTop	SRS flattop
spWindowNuttall	Nuttall
spWindowBlackman	Blackman
spWindowHamming	Hamming
spWindowGaussian6dB	Gaussian 6dB BW window for EMC measurements and CISPR compatibility
spWindowRect	Rectangular (no) window

9.1.3.10 SpDataType

enum [SpDataType](#)

Specifies a data type for data returned from the API

Enumerator

spDataType32fc	32-bit complex floats
spDataType16sc	16-bit complex shorts

9.1.3.11 SpTriggerEdge

enum [SpTriggerEdge](#)

Trigger edge for video and external triggers.

Enumerator

spTriggerEdgeRising	Rising edge
spTriggerEdgeFalling	Falling edge

9.1.3.12 SpGPSSState

enum [SpGPSSState](#)

Internal GPS state

Enumerator

spGPSSStateNotPresent	GPS is not locked
spGPSSStateLocked	GPS is locked, NMEA data is valid, but the timebase is not being disciplined by the GPS
spGPSSStateDisciplined	GPS is locked, NMEA data is valid, timebase is being disciplined by the GPS

9.1.3.13 SpAudioType

enum [SpAudioType](#)

Audio demodulation type.

Enumerator

spAudioTypeAM	AM
spAudioTypeFM	FM
spAudioTypeUSB	Upper side band
spAudioTypeLSB	Lower side band
spAudioTypeCW	CW

9.1.4 Function Documentation

9.1.4.1 spGetDeviceList()

```
SP_API SpStatus spGetDeviceList (
    int * serials,
    int * deviceCount )
```

This function is used to retrieve the serial numbers of all unopened SP devices connected to the PC. The maximum number of serial numbers that can be returned is [SP_MAX_DEVICES](#). The serial numbers returned can then be used to open specific devices with the [spOpenDeviceBySerial](#) function. When the function returns successfully, the *serials* array will contain *deviceCount* number of unique SP serial numbers. Only *deviceCount* values will be modified.

Parameters

out	<i>serials</i>	Pointer to an array of integers. The array must be larger than the number of SP devices connected to the PC.
out	<i>deviceCount</i>	If the function returns successfully <i>deviceCount</i> will be set to the number devices found on the system.

Returns

9.1.4.2 spOpenDevice()

```
SP_API SpStatus spOpenDevice (
    int * device )
```

Claim the first unopened SP device detected on the system. If the device is opened successfully, a handle to the function will be returned through the *device* pointer. This handle can then be used to refer to this device for all future API calls. This function has the same effect as calling [spGetDeviceList](#) and using the first device found to call [spOpenDeviceBySerial](#).

Parameters

out	<i>device</i>	Returns handle that can be used to interface the device.
-----	---------------	--

Returns

9.1.4.3 spOpenDeviceBySerial()

```
SP_API SpStatus spOpenDeviceBySerial (
    int * device,
    int serialNumber )
```

This function is similar to [spOpenDevice](#) except it allows you to specify the device you wish to open. This function is often used in conjunction with [spGetDeviceList](#) when managing several SP devices on one PC.

Parameters

out	<i>device</i>	Returns handle that can be used to interface the device.
in	<i>serialNumber</i>	Serial number of the device you wish to open.

Returns

9.1.4.4 spCloseDevice()

```
SP_API SpStatus spCloseDevice (
    int device )
```

This function should be called when you want to release the resources for a device. All resources (memory, etc.) will be released, and the device will become available again for use in the current process. The device handle specified will no longer point to a valid device and the device must be re-opened again to be used. This function should be called before the process exits, but it is not strictly required.

Parameters

in	<i>device</i>	Device handle.
----	---------------	----------------

Returns

9.1.4.5 spPresetDevice()

```
SP_API SpStatus spPresetDevice (
    int device )
```

Performs a full device preset. When this function returns, the hardware will have performed a full reset, the device handle will no longer be valid, the [spCloseDevice](#) function will have been called for the device handle, and the device will need to be re-opened again. This function can be used to recover from an undesirable device state.

Parameters

in	<i>device</i>	Device handle.
----	---------------	----------------

Returns

9.1.4.6 spSetPowerState()

```
SP_API SpStatus spSetPowerState (
    int device,
    SpPowerState powerState )
```

Change the power state of the device. The power state controls the power consumption of the device. See [Power States](#) for more information.

Parameters

in	<i>device</i>	Device handle.
in	<i>powerState</i>	New power state.

Returns

9.1.4.7 spGetPowerState()

```
SP_API SpStatus spGetPowerState (
    int device,
    SpPowerState * powerState )
```

Retrieves the current power state. See [Power States](#) for more information.

Parameters

in	<i>device</i>	Device handle.
out	<i>powerState</i>	Pointer to SpPowerState .

Returns

9.1.4.8 spGetSerialNumber()

```
SP_API SpStatus spGetSerialNumber (
    int device,
    int * serialNumber )
```

This function returns the serial number of a specific open device.

Parameters

in	<i>device</i>	Device handle.
out	<i>serialNumber</i>	Returns device serial number. Can be NULL.

Returns

9.1.4.9 spGetFirmwareVersion()

```
SP_API SpStatus spGetFirmwareVersion (
    int device,
    int * major,
    int * minor,
    int * rev )
```

Get the firmware version of the device. The firmware version is of the form `major.minor.revision`.

Parameters

in	<i>device</i>	Device handle.
out	<i>major</i>	Pointer to int. Can be NULL.
out	<i>minor</i>	Pointer to int. Can be NULL.
out	<i>revision</i>	Pointer to int. Can be NULL.

Returns

9.1.4.10 spGetDeviceDiagnostics()

```
SP_API SpStatus spGetDeviceDiagnostics (
    int device,
    float * voltage,
    float * current,
    float * temperature )
```

Return operational information about a device.

Parameters

in	<i>device</i>	Device handle.
out	<i>voltage</i>	Pointer to float, to contain device voltage. Can be NULL.
out	<i>current</i>	Pointer to float, to contain device current. Can be NULL.
out	<i>temperature</i>	Pointer to float, to contain device temperature. Can be NULL.

Returns

9.1.4.11 spGetCalDate()

```
SP_API SpStatus spGetCalDate (
    int device,
    uint32_t * lastCalDate )
```

Return the last device adjustment date.

Parameters

in	<i>device</i>	Device handle.
out	<i>lastCalDate</i>	Last adjustment data as seconds since epoch.

Returns

9.1.4.12 spSetExtReference()

```
SP_API SpStatus spSetExtReference (
    int device,
    SpBool enabled )
```

Enable or disable the 10MHz reference out port. If enabled, the current reference being used by the SP will be output on the 10MHz out port.

Parameters

in	<i>device</i>	Device handle.
in	<i>enabled</i>	Set to spTrue to enable the 10MHz reference out port.

Returns

9.1.4.13 spGetExtReference()

```
SP_API SpStatus spGetExtReference (
    int device,
    SpBool * enabled )
```

Return whether the 10MHz reference out port is enabled.

Parameters

in	<i>device</i>	Device handle.
out	<i>enabled</i>	Returns spTrue if the ref out port is enabled.

Returns

9.1.4.14 spSetGPSTimebaseUpdate()

```
SP_API SpStatus spSetGPSTimebaseUpdate (
    int device,
    SpBool enabled )
```

Enable whether or not the API auto updates the timebase calibration value when a valid GPS lock is acquired. This function must be called in an idle state. See [Automatic GPS Timebase Discipline](#) for more information.

Parameters

in	<i>device</i>	Device handle.
in	<i>enabled</i>	Send spTrue to enable.

Returns

9.1.4.15 spGetGPSTimebaseUpdate()

```
SP_API SpStatus spGetGPSTimebaseUpdate (
    int device,
    SpBool * enabled )
```

Get auto GPS timebase update status. See [Automatic GPS Timebase Discipline](#) for more information.

Parameters

in	<i>device</i>	Device handle.
out	<i>enabled</i>	Returns spTrue if auto GPS timebase update is enabled.

Returns

9.1.4.16 spGetGPSHoldoverInfo()

```
SP_API SpStatus spGetGPSHoldoverInfo (
    int device,
    SpBool * usingGPSHoldover,
    uint32_t * lastHoldoverTime )
```

Return information about the GPS holdover correction. Determine if a correction exists and when it was generated.

Parameters

in	<i>device</i>	Device handle.
out	<i>usingGPSHoldover</i>	Returns whether the GPS holdover value is newer than the factory calibration value. To determine whether the holdover value is actively in use, you will need to use this function in combination with spGetGPSSState . This parameter can be NULL.
out	<i>lastHoldoverTime</i>	If a GPS holdover value exists on the system, return the timestamp of the value. Value is seconds since epoch. This parameter can be NULL.

Returns

9.1.4.17 spGetGPSSState()

```
SP_API SpStatus spGetGPSSState (
    int device,
    SpGPSSState * GPSSState )
```

Determine the lock and discipline status of the GPS. See the [Acquiring GPS Lock](#) section for more information.

Parameters

in	<i>device</i>	Device handle.
out	<i>GPSSState</i>	Pointer to SpGPSSState .

Returns

9.1.4.18 spSetRefLevel()

```
SP_API SpStatus spSetRefLevel (
    int device,
    double refLevel )
```

The reference level controls the sensitivity of the receiver by setting the attenuation of the receiver to optimize measurements for signals at or below the reference level. See [Reference Level and Sensitivity](#) for more information. Attenuation must be set to automatic (-1) to set reference level.

Parameters

in	<i>device</i>	Device handle.
in	<i>refLevel</i>	Set the reference level of the receiver in dBm.

Returns

9.1.4.19 spGetRefLevel()

```
SP_API SpStatus spGetRefLevel (
    int device,
    double * refLevel )
```

Retrieve the current device reference level.

Parameters

in	<i>device</i>	Device handle.
out	<i>refLevel</i>	Reference level returned in dBm.

Returns

9.1.4.20 spSetAttenuator()

```
SP_API SpStatus spSetAttenuator (
    int device,
    int atten )
```

Set the device attenuation. See [Reference Level and Sensitivity](#) for more information. Valid values for attenuation are between [0,6] representing between [0,30] dB of attenuation (5dB steps). Setting the attenuation to -1 tells the receiver to automatically choose the best attenuation value for the specified reference level selected. Setting attenuation to a non-auto value overrides the reference level selection. The header file provides the [SP_AUTO_ATTEN](#) macro for -1.

Parameters

in	<i>device</i>	Device handle.
in	<i>atten</i>	Attenuation value between [-1,6].

Returns

9.1.4.21 spGetAttenuator()

```
SP_API SpStatus spGetAttenuator (
    int device,
    int * atten )
```

Get the device attenuation. See [Reference Level and Sensitivity](#) for more information.

Parameters

in	<i>device</i>	Device handle.
out	<i>atten</i>	Returns current attenuation value.

Returns

9.1.4.22 spSetSweepCenterSpan()

```
SP_API SpStatus spSetSweepCenterSpan (
    int device,
    double centerFreqHz,
    double spanHz )
```

Set sweep center/span.

Parameters

in	<i>device</i>	Device handle.
in	<i>centerFreqHz</i>	New center frequency in Hz.
in	<i>spanHz</i>	New span in Hz.

Returns

9.1.4.23 spSetSweepStartStop()

```
SP_API SpStatus spSetSweepStartStop (
    int device,
    double startFreqHz,
    double stopFreqHz )
```

Set sweep start/stop frequency.

Parameters

in	<i>device</i>	Device handle.
in	<i>startFreqHz</i>	Start frequency in Hz.
in	<i>stopFreqHz</i>	Stop frequency in Hz.

Returns

9.1.4.24 spSetSweepCoupling()

```
SP_API SpStatus spSetSweepCoupling (
    int device,
    double rbw,
    double vbw,
    double sweepTime )
```

Set sweep RBW/VBW parameters.

Parameters

in	<i>device</i>	Device handle.
in	<i>rbw</i>	Resolution bandwidth in Hz.
in	<i>vbw</i>	Video bandwidth in Hz. Cannot be greater than RBW.
in	<i>sweepTime</i>	Suggest the total acquisition time of the sweep. Specified in seconds. This parameter is a suggestion and will ensure RBW and VBW are first met before increasing sweep time.

Returns

9.1.4.25 spSetSweepDetector()

```
SP_API SpStatus spSetSweepDetector (
    int device,
    SpDetector detector,
    SpVideoUnits videoUnits )
```

Set sweep detector.

Parameters

in	<i>device</i>	Device handle.
in	<i>detector</i>	New sweep detector.
in	<i>videoUnits</i>	New video processing units.

Returns

9.1.4.26 spSetSweepScale()

```
SP_API SpStatus spSetSweepScale (
    int device,
    SpScale scale )
```

Set the sweep mode output unit type.

Parameters

in	<i>device</i>	Device handle.
in	<i>scale</i>	New sweep mode units.

Returns

9.1.4.27 spSetSweepWindow()

```
SP_API SpStatus spSetSweepWindow (
    int device,
    SpWindowType window )
```

Set sweep mode window function.

Parameters

in	<i>device</i>	Device handle.
in	<i>window</i>	New window function.

Returns

9.1.4.28 spSetRealTimeCenterSpan()

```
SP_API SpStatus spSetRealTimeCenterSpan (
    int device,
    double centerFreqHz,
    double spanHz )
```

Set the center frequency and span for real-time spectrum analysis.

Parameters

in	<i>device</i>	Device handle.
in	<i>centerFreqHz</i>	Center frequency in Hz.
in	<i>spanHz</i>	Span in Hz.

Returns

9.1.4.29 spSetRealTimeRBW()

```
SP_API SpStatus spSetRealTimeRBW (
    int device,
    double rbw )
```

Set the resolution bandwidth for real-time spectrum analysis.

Parameters

in	<i>device</i>	Device handle.
in	<i>rbw</i>	Resolution bandwidth in Hz.

Returns

9.1.4.30 spSetRealTimeDetector()

```
SP_API SpStatus spSetRealTimeDetector (
    int device,
    SpDetector detector )
```

Set the detector for real-time spectrum analysis.

Parameters

in	<i>device</i>	Device handle.
in	<i>detector</i>	New detector.

Returns

9.1.4.31 spSetRealTimeScale()

```
SP_API SpStatus spSetRealTimeScale (
    int device,
    SpScale scale,
    double frameRef,
    double frameScale )
```

Set the sweep and frame units used in real-time spectrum analysis.

Parameters

in	<i>device</i>	Device handle.
in	<i>scale</i>	Scale for the returned sweeps.
in	<i>frameRef</i>	Sets the reference level of the real-time frame, or, the amplitude of the highest pixel in the frame.
in	<i>frameScale</i>	Specify the height of the frame in dB. A common value is 100dB.

Returns

9.1.4.32 spSetRealTimeWindow()

```
SP_API SpStatus spSetRealTimeWindow (
    int device,
    SpWindowType window )
```

Specify the window function used for real-time spectrum analysis.

Parameters

in	<i>device</i>	Device handle.
in	<i>window</i>	New window function.

Returns

9.1.4.33 spSetIQDataType()

```
SP_API SpStatus spSetIQDataType (
    int device,
    SpDataType dataType )
```

Set the I/Q data type of the samples returned for I/Q streaming.

Parameters

in	<i>device</i>	Device handle.
in	<i>dataType</i>	Data type. See I/Q Data Types for more information.

Returns

9.1.4.34 spSetIQCenterFreq()

```
SP_API SpStatus spSetIQCenterFreq (
    int device,
    double centerFreqHz )
```

Set the center frequency for I/Q streaming.

Parameters

in	<i>device</i>	Device handle.
in	<i>centerFreqHz</i>	Center frequency in Hz.

Returns

9.1.4.35 spGetIQCenterFreq()

```
SP_API SpStatus spGetIQCenterFreq (
    int device,
    double * centerFreqHz )
```

Get the I/Q streaming center frequency.

Parameters

in	<i>device</i>	Device handle.
in	<i>centerFreqHz</i>	Pointer to double.

Returns

9.1.4.36 spSetIQSampleRate()

```
SP_API SpStatus spSetIQSampleRate (
    int device,
    int decimation )
```

Set sample rate for I/Q streaming.

Parameters

in	<i>device</i>	Device handle.
in	<i>decimation</i>	Decimation of the I/Q data as a power of 2. See I/Q Sample Rates for more information.

Returns

9.1.4.37 spSetIQSoftwareFilter()

```
SP_API SpStatus spSetIQSoftwareFilter (
    int device,
    SpBool enabled )
```

Enable/disable software filtering.

Parameters

in	<i>device</i>	Device handle.
in	<i>enableSoftwareFilter</i>	Set to spTrue to enable software filtering.

Returns

9.1.4.38 spSetIQBandwidth()

```
SP_API SpStatus spSetIQBandwidth (
    int device,
    double bandwidth )
```

Specify the software filter bandwidth in I/Q streaming. See [I/Q Sample Rates](#) for more information.

Parameters

in	<i>device</i>	Device handle.
in	<i>bandwidth</i>	The bandwidth in Hz.

Returns

9.1.4.39 spSetIQExtTriggerEdge()

```
SP_API SpStatus spSetIQExtTriggerEdge (
    int device,
    SpTriggerEdge edge )
```

Configure the external trigger edge detect in I/Q streaming.

Parameters

in	<i>device</i>	Device handle.
in	<i>edge</i>	Set the external trigger edge.

Returns

9.1.4.40 spSetIQTriggerSentinel()

```
SP_API SpStatus spSetIQTriggerSentinel (
    double sentinelValue )
```

Configure how external triggers are reported for I/Q streaming.

Parameters

in	<i>sentinelValue</i>	Value used to fill the remainder of the trigger buffer when the trigger buffer provided is larger than the number of triggers returned. The default sentinel value is zero. See the I/Q Streaming section for more information on triggering.
----	----------------------	---

Returns

9.1.4.41 spSetIQQueueSize()

```
SP_API SpStatus spSetIQQueueSize (
    int device,
    float ms )
```

Controls the size of the queue of data that is being actively requested by the API. For example, a queue size of 20ms means the API keeps up to 20ms of data requests active. A larger queue size means a greater tolerance to data loss in the event of an interruption. Because once data is requested, its transfer must be completed, a smaller queue size can give you faster reconfiguration times. For instance, if you wanted to change frequencies quickly, a smaller queue size would allow this. A default is chosen for the best resistance to data loss for both Linux and Windows. If you are on Linux and you are using multiple devices, please see [Linux Notes](#).

Parameters

in	<i>device</i>	Device handle
in	<i>ms</i>	Queue size in ms. Will be clamped to multiples of 2.62ms between 2 * 2.62ms and 16 * 2.62ms.

Returns

9.1.4.42 spSetIQSweepListDataType()

```
SP_API SpStatus spSetIQSweepListDataType (
    int device,
    SpDataType dataType )
```

Set the data type for data returned for I/Q sweep list measurements.

Parameters

in	<i>device</i>	Device handle.
in	<i>dataType</i>	See I/Q Data Types for more information.

Returns

9.1.4.43 spSetIQSweepListCorrected()

```
SP_API SpStatus spSetIQSweepListCorrected (
    int device,
    SpBool corrected )
```

Set whether the data returns for I/Q sweep list measurements is full-scale or corrected.

Parameters

in	<i>device</i>	Device handle.
in	<i>corrected</i>	Set to spFalse for the data to be returned as full scale, and spTrue to be returned amplitude corrected. See I/Q Data Types for more information on how to perform these conversions.

Returns

9.1.4.44 spSetIQSweepListSteps()

```
SP_API SpStatus spSetIQSweepListSteps (
    int device,
    int steps )
```

Set the number of frequency steps for I/Q sweep list measurements.

Parameters

in	<i>device</i>	Device handle.
in	<i>steps</i>	Number of frequency steps in I/Q sweep.

Returns

9.1.4.45 spGetIQSweepListSteps()

```
SP_API SpStatus spGetIQSweepListSteps (
    int device,
    int * steps )
```

Get the number steps in the I/Q sweep list measurement.

Parameters

in	<i>device</i>	Device handle.
out	<i>steps</i>	Pointer to int.

Returns

9.1.4.46 spSetIQSweepListFreq()

```
SP_API SpStatus spSetIQSweepListFreq (
    int device,
    int step,
    double freq )
```

Set the center frequency of the acquisition at a given step for the I/Q sweep list measurement.

Parameters

in	<i>device</i>	Device handle.
in	<i>step</i>	Step at which to configure the center frequency. Should be between [0, <i>steps</i> -1] where <i>steps</i> is set in the spSetIQSweepListSteps function.
in	<i>freq</i>	Center frequency in Hz.

Returns

9.1.4.47 spSetIQSweepListRef()

```
SP_API SpStatus spSetIQSweepListRef (
    int device,
    int step,
    double level )
```

Set the reference level for a step for the I/Q sweep list measurement.

Parameters

in	<i>device</i>	Device handle.
in	<i>step</i>	Step at which to configure the center frequency. Should be between [0, <i>steps</i> -1] where <i>steps</i> is set in the spSetIQSweepListSteps function.
in	<i>level</i>	Reference level in dBm. If this is set, attenuation is set to automatic for this step.

Returns

9.1.4.48 spSetIQSweepListAtten()

```
SP_API SpStatus spSetIQSweepListAtten (
    int device,
    int step,
    int atten )
```

Set the attenuation for a step for the I/Q sweep list measurement.

Parameters

in	<i>device</i>	Device handle.
in	<i>step</i>	Step at which to configure the center frequency. Should be between [0, <i>steps</i> -1] where <i>steps</i> is set in the spSetIQSweepListSteps function.
in	<i>atten</i>	Attenuation value between [0,6] representing [0,30] dB of attenuation (5dB steps). Setting the attenuation to -1 forces the attenuation to auto, at which time the reference level is used to control the attenuator instead.

Returns

9.1.4.49 spSetIQSweepListSampleCount()

```
SP_API SpStatus spSetIQSweepListSampleCount (
    int device,
    int step,
    uint32_t samples )
```

Set the number of I/Q samples to be collected at each step.

Parameters

in	<i>device</i>	Device handle.
in	<i>step</i>	Step at which to configure the center frequency. Should be between [0, <i>steps</i> -1] where <i>steps</i> is set in the spSetIQSweepListSteps function.
Generated by Doxygen	<i>samples</i>	Number of samples. Must be greater than 0. There is no upper limit, but keep in mind contiguous memory must be allocated for the capture. Memory allocation for the capture is the responsibility of the user program.

Returns

9.1.4.50 spSetAudioCenterFreq()

```
SP_API SpStatus spSetAudioCenterFreq (
    int device,
    double centerFreqHz )
```

Set the center frequency for audio demodulation.

Parameters

in	<i>device</i>	Device handle.
in	<i>centerFreqHz</i>	Center frequency in Hz.

Returns

9.1.4.51 spSetAudioType()

```
SP_API SpStatus spSetAudioType (
    int device,
    SpAudioType audioType )
```

Set the audio demodulator for audio demodulation.

Parameters

in	<i>device</i>	Device handle.
in	<i>audioType</i>	Demodulator.

Returns

9.1.4.52 spSetAudioFilters()

```
SP_API SpStatus spSetAudioFilters (
    int device,
```



```
double ifBandwidth,  
double audioLpf,  
double audioHpf )
```

Set the audio demodulation filters for audio demodulation.

Parameters

in	<i>device</i>	Device handle.
in	<i>ifBandwidth</i>	IF bandwidth (RBW) in Hz.
in	<i>audioLpf</i>	Audio low pass frequency in Hz.
in	<i>audioHpf</i>	Audio high pass frequency in Hz.

Returns

9.1.4.53 spSetAudioFMDeemphasis()

```
SP_API SpStatus spSetAudioFMDeemphasis (  
    int device,  
    double deemphasis )
```

Set the FM deemphasis for audio demodulation.

Parameters

in	<i>device</i>	Device handle.
in	<i>deemphasis</i>	Deemphasis in us.

Returns

9.1.4.54 spConfigure()

```
SP_API SpStatus spConfigure (  
    int device,  
    SpMode mode )
```

This function configures the receiver into a state determined by the mode parameter. All relevant configuration routines must have already been called. This function calls [spAbort](#) to end the previous measurement mode before attempting to configure the receiver. If any error occurs attempting to configure the new measurement state, the previous measurement mode will no longer be active.

Parameters

in	<i>device</i>	Device handle.
in	<i>mode</i>	New measurement mode.

Returns**9.1.4.55 spGetCurrentMode()**

```
SP_API SpStatus spGetCurrentMode (
    int device,
    SpMode * mode )
```

Retrieve the current device measurement mode.

Parameters

in	<i>device</i>	Device handle.
in	<i>mode</i>	Pointer to SpMode .

Returns**9.1.4.56 spAbort()**

```
SP_API SpStatus spAbort (
    int device )
```

This function ends the current measurement mode and puts the device into the idle state. Any current measurements are completed and discarded and will not be accessible after this function returns.

Parameters

in	<i>device</i>	Device handle.
----	---------------	----------------

Returns

9.1.4.57 spGetSweepParameters()

```
SP_API SpStatus spGetSweepParameters (
    int device,
    double * actualRBW,
    double * actualVBW,
    double * actualStartFreq,
    double * binSize,
    int * sweepSize )
```

Retrieves the sweep parameters for an active sweep measurement mode. This function should be called after a successful device configuration to retrieve the sweep characteristics.

Parameters

in	<i>device</i>	Device handle.
out	<i>actualRBW</i>	Returns the RBW being used in Hz. Can be NULL.
out	<i>actualVBW</i>	Returns the VBW being used in Hz. Can be NULL.
out	<i>actualStartFreq</i>	Returns the frequency of the first bin in Hz. Can be NULL.
out	<i>binSize</i>	Returns the frequency spacing between each frequency bin in the sweep in Hz.
out	<i>sweepSize</i>	Returns the length of the sweep (number of frequency bins). Can be NULL.

Returns

9.1.4.58 spGetRealTimeParameters()

```
SP_API SpStatus spGetRealTimeParameters (
    int device,
    double * actualRBW,
    int * sweepSize,
    double * actualStartFreq,
    double * binSize,
    int * frameWidth,
    int * frameHeight,
    double * poi )
```

Retrieve the real-time measurement mode parameters for an active real-time configuration. This function is typically called after a successful device configuration to retrieve the real-time sweep and frame characteristics.

Parameters

in	<i>device</i>	Device handle.
out	<i>actualRBW</i>	Returns the RBW used in Hz. Can be NULL.
out	<i>sweepSize</i>	Returns the number of frequency bins in the sweep. Can be NULL.
out	<i>actualStartFreq</i>	Returns the frequency of the first bin in the sweep in Hz. Can be NULL.
out	<i>binSize</i>	Frequency bin spacing in Hz. Can be NULL.
out	<i>frameWidth</i>	The width of the real-time frame. Can be NULL.
out	<i>frameHeight</i>	The height of the real-time frame. Can be NULL.
out	<i>poi</i>	100% probability of intercept of a signal given the current configuration. Can be NULL.

Returns

9.1.4.59 spGetIQParameters()

```
SP_API SpStatus spGetIQParameters (
    int device,
    double * sampleRate,
    double * bandwidth )
```

Retrieve the I/Q measurement mode parameters for an active I/Q stream. This function is called after a successful device configuration.

Parameters

in	<i>device</i>	Device handle.
out	<i>sampleRate</i>	The sample rate in Hz. Can be NULL.
	<i>bandwidth</i>	The bandwidth of the I/Q capture in Hz. Can be NULL.

Returns

9.1.4.60 spGetIQCorrection()

```
SP_API SpStatus spGetIQCorrection (
    int device,
    float * scale )
```

Retrieve the I/Q correction factor for an active I/Q stream. This function is called after a successful device configuration.

Parameters

in	<i>device</i>	Device handle.
out	<i>scale</i>	Amplitude correction used by the API to convert from full scale I/Q to amplitude corrected I/Q. The formulas for these conversions are in the I/Q Data Types section.

Returns

9.1.4.61 spIQSweepListGetCorrections()

```
SP_API SpStatus spIQSweepListGetCorrections (
    int device,
    float * corrections )
```

Retrieve the corrections used to convert full scale I/Q values to amplitude corrected I/Q values for the I/Q sweep list measurement. A correction is returned for each step configured. The device must be configured for I/Q sweep list measurements before calling this function.

Parameters

in	<i>device</i>	Device handle.
out	<i>corrections</i>	Pointer to an array. Array should have length \geq number of steps configured for the I/Q sweep list measurement. A correction value will be returned for each step configured.

Returns

9.1.4.62 spGetSweep()

```
SP_API SpStatus spGetSweep (
    int device,
    float * sweepMin,
    float * sweepMax,
    int64_t * nsSinceEpoch )
```

Perform a single sweep. Block until the sweep completes.

Parameters

in	<i>device</i>	Device handle.
out	<i>sweepMin</i>	Can be NULL.
out	<i>sweepMax</i>	Can be NULL.
out	<i>nsSinceEpoch</i>	Nanoseconds since epoch. Timestamp representing the end of the sweep. Can be NULL.

Returns

9.1.4.63 spGetRealTimeFrame()

```
SP_API SpStatus spGetRealTimeFrame (
    int device,
```

```

float * colorFrame,
float * alphaFrame,
float * sweepMin,
float * sweepMax,
int * frameCount,
int64_t * nsSinceEpoch )

```

Retrieve a single real-time frame. See [Real-Time Spectrum Analysis](#) for more information.

Parameters

in	<i>device</i>	Device handle.
out	<i>colorFrame</i>	Pointer to memory for the frame. Must be (<i>frameWidth</i> * <i>frameHeight</i>) floats in size. Can be NULL.
out	<i>alphaFrame</i>	Pointer to memory for the frame. Must be (<i>frameWidth</i> * <i>frameHeight</i>) floats in size. Can be NULL.
out	<i>sweepMin</i>	Can be NULL.
out	<i>sweepMax</i>	Can be NULL.
out	<i>frameCount</i>	Unique integer which refers to a real-time frame and sweep. The frame count starts at zero following a device reconfigure and increments by one for each frame.
out	<i>nsSinceEpoch</i>	Nanoseconds since epoch for the returned frame. For real-time mode, this value represents the time at the end of the real-time acquisition and processing of this given frame. It is approximate. Can be NULL.

Returns

9.1.4.64 spGetIQ()

```

SP_API SpStatus spGetIQ (
    int device,
    void * iqBuf,
    int iqBufSize,
    double * triggers,
    int triggerBufSize,
    int64_t * nsSinceEpoch,
    SpBool purge,
    int * sampleLoss,
    int * samplesRemaining )

```

Retrieve one block of I/Q data as specified by the user. This function blocks until the data requested is available.

Parameters

in	<i>device</i>	Device handle.
out	<i>iqBuf</i>	Pointer to user allocated buffer of complex values. The buffer size must be at least (<i>iqBufSize</i> * 2 * sizeof(dataTypeSelected)). Cannot be NULL. Data is returned as interleaved contiguous complex samples. For more information on the data returned and the selectable data types, see I/Q Data Types .
in	<i>iqBufSize</i>	Specifies the number of I/Q samples to be retrieved. Must be greater than zero.

Parameters

out	<i>triggers</i>	Pointer to user-allocated array of doubles. The buffer must be at least <i>triggerBufSize</i> contiguous doubles. The pointer can also be NULL to indicate you do not wish to receive external trigger information. See I/Q Streaming section for more information on triggers.
in	<i>triggerBufSize</i>	Specifies the size of the <i>triggers</i> array. If the <i>triggers</i> array is NULL, this value should be zero.
out	<i>nsSinceEpoch</i>	Nanoseconds since epoch. The time of the first I/Q sample returned. Can be NULL. See GPS and Timestamps for more information.
in	<i>purge</i>	When set to spTrue , any buffered I/Q data in the API is purged before returned beginning the I/Q block acquisition.
out	<i>sampleLoss</i>	Set by the API when a sample loss condition occurs. If enough I/Q data has accumulated in the internal API circular buffer, the buffer is cleared and the sample loss flag is set. If purge is set to true, the sample flag will always be set to SP_FALSE . Can be NULL.
out	<i>samplesRemaining</i>	Set by the API, returns the number of samples remaining in the I/Q circular buffer. Can be NULL.

Returns

9.1.4.65 spIQSweepListGetSweep()

```
SP_API SpStatus spIQSweepListGetSweep (
    int device,
    void * dst,
    int64_t * timestamps )
```

Perform an I/Q sweep. Blocks until the sweep is complete. Can only be called if no sweeps are in the queue.

Parameters

in	<i>device</i>	Device handle.
out	<i>dst</i>	Pointer to memory allocated for sweep. The user must allocate this memory before calling this function. Must be large enough to contain all samples for all steps in a sweep. The memory must be contiguous. The samples in the sweep are placed contiguously into the array (step 1 samples follow step 0, step 2 follows step 1, etc). Samples are tightly packed. It is the responsibility of the user to properly index the arrays when finished. The array will be cast to the user-selected data type internally in the API.
out	<i>timestamps</i>	Pointer to memory allocated for timestamps. The user must allocate this memory before calling these functions. Must be an array of <i>steps</i> int64_ts, where <i>steps</i> is the number of frequency steps in the sweep. When the sweep completes each timestamp in the array represents the time of the first sample at that frequency in the sweep. Can be NULL.

Returns

9.1.4.66 spIQSweepListStartSweep()

```
SP_API SpStatus spIQSweepListStartSweep (
    int device,
    int pos,
    void * dst,
    int64_t * timestamps )
```

Start an I/Q sweep at the given queue position. Up to 16 sweeps can be in queue.

Parameters

in	<i>device</i>	Device handle.
in	<i>pos</i>	Sweep queue position. Must be between [0,15].
out	<i>dst</i>	Pointer to memory allocated for sweep. The user must allocate this memory before calling this function. Must be large enough to contain all samples for all steps in a sweep. The memory must be contiguous. The samples in the sweep are placed contiguously into the array (step 1 samples follow step 0, step 2 follows step 1, etc). Samples are tightly packed. It is the responsibility of the user to properly index the arrays when finished. The array will be cast to the user-selected data type internally in the API.
out	<i>timestamps</i>	Pointer to memory allocated for timestamps. The user must allocate this memory before calling these functions. Must be an array of <i>steps</i> int64_ts, where <i>steps</i> is the number of frequency steps in the sweep. When the sweep completes each timestamp in the array represents the time of the first sample at that frequency in the sweep. Can be NULL.

Returns

9.1.4.67 spIQSweepListFinishSweep()

```
SP_API SpStatus spIQSweepListFinishSweep (
    int device,
    int pos )
```

Finish an I/Q sweep at the given queue position. Blocks until the sweep is finished.

Parameters

in	<i>device</i>	Device handle.
in	<i>pos</i>	Sweep queue position. Must be between [0,15].

Returns

9.1.4.68 spGetAudio()

```
SP_API SpStatus spGetAudio (
    int device,
    float * audio )
```

If the device is configured to audio demodulation, use this function to retrieve the next 1000 audio samples. This function will block until the data is ready. Minor buffering of audio data is performed in the API, so it is necessary this function is called repeatedly if contiguous audio data is required. The values returned range between [-1.0, 1.0] representing full-scale audio. In FM mode, the audio values will scale with a change in IF bandwidth.

Parameters

in	<i>device</i>	Device handle.
out	<i>audio</i>	Pointer to array of 1000 32-bit floats.

Returns

9.1.4.69 spGetGPSInfo()

```
SP_API SpStatus spGetGPSInfo (
    int device,
    SpBool refresh,
    SpBool * updated,
    int64_t * secSinceEpoch,
    double * latitude,
    double * longitude,
    double * altitude,
    char * nmea,
    int * nmeaLen )
```

Acquire the latest GPS information which includes a time stamp, location information, and NMEA sentences. The GPS info is updated once per second at the PPS interval. This function can be called while measurements are active. NMEA data can contain null values. When parsing, do not use the null delimiter to mark the end of the message, use the returned *nmeaLen*.

Parameters

in	<i>device</i>	Device handle.
in	<i>refresh</i>	When set to true and the device is not in a streaming mode, the API will request the latest GPS information. Otherwise the last retrieved data is returned.
out	<i>updated</i>	Will be set to true if the NMEA data has been updated since the last time the user called this function. Can be set to NULL.
out	<i>secSinceEpoch</i>	Number of seconds since epoch as reported by the GPS NMEA sentences. Last reported value by the GPS. If the GPS is not locked, this value will be set to zero. Can be NULL.
out	<i>latitude</i>	Latitude in decimal degrees. If the GPS is not locked, this value will be set to zero. Can be NULL.
out	<i>longitude</i>	Longitude in decimal degrees. If the GPS is not locked, this value will be set to zero. Can be NULL.

Parameters

out	<i>altitude</i>	Altitude in meters. If the GPS is not locked, this value will be set to zero. Can be NULL.
out	<i>nmea</i>	Pointer to user-allocated array of char. The length of this array is specified by the <i>nmeaLen</i> parameter. Can be set to NULL.
in, out	<i>nmeaLen</i>	Pointer to an integer. The integer will initially specify the length of the nmea buffer. If the nmea buffer is shorter than the NMEA sentences to be returned, the API will only copy over <i>nmeaLen</i> characters, including the null terminator. After the function returns, <i>nmeaLen</i> will be the length of the copied nmea data, including the null terminator. Can be set to NULL. If NULL, the nmea parameter is ignored.

Returns

9.1.4.70 spSetFanThreshold()

```
SP_API SpStatus spSetFanThreshold (
    int device,
    float temp )
```

Specify the temperature at which the fan should be enabled. This function has no effect if the optional fan assembly is not installed. The available temperature range is between [10-90] degrees. This function must be called when the device is idle (no measurement mode active).

Parameters

in	<i>device</i>	Device handle.
in	<i>temp</i>	Temperature in C.

Returns

9.1.4.71 spGetFanSettings()

```
SP_API SpStatus spGetFanSettings (
    int device,
    float * threshold,
    float * voltage )
```

Get current fan temperature threshold and voltage.

Parameters

in	<i>device</i>	Device handle.
out	<i>threshold</i>	Temperature in C.
out	<i>voltage</i>	Voltage in mV.

Returns

9.1.4.72 spGetErrorString()

```
SP_API const char * spGetErrorString (
    SpStatus status )
```

Retrieve a descriptive string of an [SpStatus](#) enumeration. Useful for debugging and diagnostic purposes.

Parameters

in	<i>status</i>	Status code returned from any API function.
----	---------------	---

Returns

9.1.4.73 spGetAPIVersion()

```
SP_API const char * spGetAPIVersion ( )
```

Get the API version.

Returns

The returned string is of the form

major.minor.revision

Ascii periods ('.') separate positive integers. Major/minor/revision are not guaranteed to be a single decimal digit. The string is null terminated. The string should not be modified or freed by the user. An example string is below?

```
['3' | ':' | '0' | ':' | '1' | '1' | '\0'] = "3.0.11"
```

9.2 sp_api.h

[Go to the documentation of this file.](#)

```

1 // Copyright (c).2023, Signal Hound, Inc.
2 // For licensing information, please see the API license in the software_licenses folder
3
13 #ifndef SP_API_H
14 #define SP_API_H
15
16 #if defined(_WIN32) // Windows
17 #ifdef SP_EXPORTS
18 #define SP_API __declspec(dllexport)
19 #else
20 #define SP_API
21 #endif
22
23 // bare minimum stdint typedef support
24 #if _MSC_VER < 1700 // For VS2010 or earlier
25     typedef signed char    int8_t;
26     typedef short          int16_t;
27     typedef int             int32_t;
28     typedef long long       int64_t;
29     typedef unsigned char   uint8_t;
30     typedef unsigned short  uint16_t;
31     typedef unsigned int    uint32_t;
32     typedef unsigned long long uint64_t;
33 #else
34 #include <stdint.h>
35 #endif
36
37 #define SP_DEPRECATED(comment) __declspec(deprecated(comment))
38 #else // Linux
39 #include <stdint.h>
40 #define SP_API __attribute__((visibility("default")))
41
42 #if defined(__GNUC__)
43 #define SP_DEPRECATED(comment) __attribute__((deprecated))
44 #else
45 #define SP_DEPRECATED(comment) comment
46 #endif
47 #endif
48
49 #define SP_INVALID_HANDLE (-1)
50
51 #define SP_TRUE (1)
52 #define SP_FALSE (0)
53
54 #define SP_MAX_DEVICES (9)
55
56 #define SP_MAX_REF_LEVEL (20.0)
57 #define SP_AUTO_ATTEN (-1)
58 #define SP_MAX_ATTEN (6)
59
60 #define SP_MIN_FREQ (9.0e3)
61 #define SP_MAX_FREQ (15.0e9)
62
63 #define SP_MIN_SWEEP_TIME (1.0e-6)
64 #define SP_MAX_SWEEP_TIME (100.0)
65
66 #define SP_REAL_TIME_MIN_SPAN (200.0e3)
67 #define SP_REAL_TIME_MAX_SPAN (40.0e6)
68 #define SP_REAL_TIME_MIN_RBW (2.0e3)
69 #define SP_REAL_TIME_MAX_RBW (1.0e6)
70
71 #define SP_MAX_IQ_DECIMATION (8192)
72
73 #define SP_MAX_SWEEP_QUEUE_SZ (16)
74
75 typedef enum SpStatus { //TODO: Assign values
76     spDeviceNotFoundErr = -100,
77     spAllocationErr,
78     spTransferErr,
79     spMaxDevicesConnectedErr,
80     spInvalidParameterErr,
81     spInvalidDeviceErr,
82     spNullPtrErr,
83     spConnectionLostErr,
84     spInvalidConfigurationErr,
85     spSweepAlreadyActiveErr,
86     spFileIOErr,
87     spFx3RunErr,
88     spBootErr,
89     spInvalidSweepPositionErr,
90     spInvalidCalDataErr,
91 }

```

```

127     spNoError = 0,
128
129     spSettingClamped = 1,
130     spTempDrift = 2,
131     spADCOverflow = 3,
132     spUncalData = 4,
133     spCPULimited = 5,
134     spInvalidCalData
135 } SpStatus;
136
137 typedef enum SpDeviceType {
138     spDeviceTypeSP145A = 0
139 } SpDeviceType;
140
141 typedef enum SpBool {
142     spFalse = 0,
143     spTrue = 1
144 } SpBool;
145
146 typedef enum SpPowerState {
147     spPowerStateOn = 0,
148     spPowerStateStandby = 1
149 } SpPowerState;
150
151 typedef enum SpMode {
152     spModeIdle = 0,
153     spModeSweeping = 1,
154     spModeRealTime = 2,
155     spModeIQStreaming = 3,
156     spModeIQSweepList = 4,
157     spModeAudio = 5,
158 } SpMode;
159
160 typedef enum SpDetector {
161     spDetectorAverage = 0,
162     spDetectorMinMax = 1
163 } SpDetector;
164
165 typedef enum SpScale {
166     spScaleLog = 0,
167     spScaleLin = 1,
168     spScaleFullScale = 2
169 } SpScale;
170
171 typedef enum SpVideoUnits {
172     spVideoLog = 0,
173     spVideoVoltage = 1,
174     spVideoPower = 2,
175     spVideoSample = 3
176 } SpVideoUnits;
177
178 typedef enum SpWindowType {
179     spWindowFlatTop = 0,
180     spWindowNutall = 1,
181     spWindowBlackman = 2,
182     spWindowHamming = 3,
183     spWindowGaussian6dB = 4,
184     spWindowRect = 5
185 } SpWindowType;
186
187 typedef enum SpDataType {
188     spDataType32fc = 0,
189     spDataType16sc = 1
190 } SpDataType;
191
192 typedef enum SpTriggerEdge {
193     spTriggerEdgeRising = 0,
194     spTriggerEdgeFalling = 1
195 } SpTriggerEdge;
196
197 typedef enum SpGPSState {
198     spGPSStateNotPresent = 0,
199     spGPSStateLocked = 1,
200     spGPSStateDisciplined = 2
201 } SpGPSState;
202
203 typedef enum SpAudioType {
204     spAudioTypeAM = 0,
205     spAudioTypeFM = 1,
206     spAudioTypeUSB = 2,
207     spAudioTypeLSB = 3,
208     spAudioTypeCW = 4
209 } SpAudioType;
210
211 #ifndef __cplusplus
212 extern "C" {
213 #endif

```

```

295
313 SP_API SpStatus spGetDeviceList(int *serials, int *deviceCount);
314
327 SP_API SpStatus spOpenDevice(int *device);
328
340 SP_API SpStatus spOpenDeviceBySerial(int *device, int serialNumber);
341
354 SP_API SpStatus spCloseDevice(int device);
355
367 SP_API SpStatus spPresetDevice(int device);
368
379 SP_API SpStatus spSetPowerState(int device, SpPowerState powerState);
380
390 SP_API SpStatus spGetPowerState(int device, SpPowerState *powerState);
391
401 SP_API SpStatus spGetSerialNumber(int device, int *serialNumber);
402
417 SP_API SpStatus spGetFirmwareVersion(int device, int *major, int *minor, int *rev);
418
435 SP_API SpStatus spGetDeviceDiagnostics(int device, float *voltage, float *current, float *temperature);
436
446 SP_API SpStatus spGetCalDate(int device, uint32_t *lastCalDate);
447
458 SP_API SpStatus spSetExtReference(int device, SpBool enabled);
459
469 SP_API SpStatus spGetExtReference(int device, SpBool *enabled);
470
482 SP_API SpStatus spSetGPSTimebaseUpdate(int device, SpBool enabled);
483
494 SP_API SpStatus spGetGPSTimebaseUpdate(int device, SpBool *enabled);
495
513 SP_API SpStatus spGetGPSHoldoverInfo(int device, SpBool *usingGPSHoldover, uint32_t *lastHoldoverTime);
514
525 SP_API SpStatus spGetGPSState(int device, SpGPSState *GPSState);
526
539 SP_API SpStatus spSetRefLevel(int device, double refLevel);
540
550 SP_API SpStatus spGetRefLevel(int device, double *refLevel);
551
567 SP_API SpStatus spSetAttenuator(int device, int atten);
568
579 SP_API SpStatus spGetAttenuator(int device, int *atten);
580
592 SP_API SpStatus spSetSweepCenterSpan(int device, double centerFreqHz, double spanHz);
593
605 SP_API SpStatus spSetSweepStartStop(int device, double startFreqHz, double stopFreqHz);
606
622 SP_API SpStatus spSetSweepCoupling(int device, double rbw, double vbw, double sweepTime);
623
635 SP_API SpStatus spSetSweepDetector(int device, SpDetector detector, SpVideoUnits videoUnits);
636
646 SP_API SpStatus spSetSweepScale(int device, SpScale scale);
647
657 SP_API SpStatus spSetSweepWindow(int device, SpWindowType window);
658
670 SP_API SpStatus spSetRealTimeCenterSpan(int device, double centerFreqHz, double spanHz);
671
681 SP_API SpStatus spSetRealTimeRBW(int device, double rbw);
682
692 SP_API SpStatus spSetRealTimeDetector(int device, SpDetector detector);
693
709 SP_API SpStatus spSetRealTimeScale(int device, SpScale scale, double frameRef, double frameScale);
710
720 SP_API SpStatus spSetRealTimeWindow(int device, SpWindowType window);
721
731 SP_API SpStatus spSetIQDataType(int device, SpDataType dataType);
732
742 SP_API SpStatus spSetIQCenterFreq(int device, double centerFreqHz);
743
753 SP_API SpStatus spGetIQCenterFreq(int device, double *centerFreqHz);
754
765 SP_API SpStatus spSetIQSampleRate(int device, int decimation);
766
776 SP_API SpStatus spSetIQSoftwareFilter(int device, SpBool enabled);
777
788 SP_API SpStatus spSetIQBandwidth(int device, double bandwidth);
789
799 SP_API SpStatus spSetIQExtTriggerEdge(int device, SpTriggerEdge edge);
800
811 SP_API SpStatus spSetIQTriggerSentinel(double sentinelValue);
812
831 SP_API SpStatus spSetIQQueueSize(int device, float ms);
832
842 SP_API SpStatus spSetIQSweepListDataType(int device, SpDataType dataType);
843
856 SP_API SpStatus spSetIQSweepListCorrected(int device, SpBool corrected);
857

```

```

867 SP_API SpStatus spSetIQSweepListSteps(int device, int steps);
868
878 SP_API SpStatus spGetIQSweepListSteps(int device, int *steps);
879
894 SP_API SpStatus spSetIQSweepListFreq(int device, int step, double freq);
895
910 SP_API SpStatus spSetIQSweepListRef(int device, int step, double level);
911
928 SP_API SpStatus spSetIQSweepListAtten(int device, int step, int atten);
929
946 SP_API SpStatus spSetIQSweepListSampleCount(int device, int step, uint32_t samples);
947
957 SP_API SpStatus spSetAudioCenterFreq(int device, double centerFreqHz);
958
968 SP_API SpStatus spSetAudioType(int device, SpAudioType audioType);
969
983 SP_API SpStatus spSetAudioFilters(int device,
984     double ifBandwidth,
985     double audioLpf,
986     double audioHpf);
987
997 SP_API SpStatus spSetAudioFMDeemphasis(int device, double deemphasis);
998
1013 SP_API SpStatus spConfigure(int device, SpMode mode);
1014
1024 SP_API SpStatus spGetCurrentMode(int device, SpMode *mode);
1025
1035 SP_API SpStatus spAbort(int device);
1036
1059 SP_API SpStatus spGetSweepParameters(int device, double *actualRBW, double *actualVBW,
1060     double *actualStartFreq, double *binSize, int *sweepSize);
1061
1088 SP_API SpStatus spGetRealTimeParameters(int device, double *actualRBW, int *sweepSize, double
1089     *actualStartFreq,
1090     double *binSize, int *frameWidth, int *frameHeight, double
1091     *poi);
1092
1103 SP_API SpStatus spGetIQParameters(int device, double *sampleRate, double *bandwidth);
1104
1105
1118 SP_API SpStatus spGetIQCorrection(int device, float *scale);
1119
1135 SP_API SpStatus spIQSweepListGetCorrections(int device, float *corrections);
1136
1151 SP_API SpStatus spGetSweep(int device, float *sweepMin, float *sweepMax, int64_t *nsSinceEpoch);
1152
1179 SP_API SpStatus spGetRealTimeFrame(int device, float *colorFrame, float *alphaFrame, float *sweepMin,
1180     float *sweepMax, int *frameCount, int64_t *nsSinceEpoch);
1181
1221 SP_API SpStatus spGetIQ(int device, void *iqBuf, int iqBufSize, double *triggers, int triggerBufSize,
1222     int64_t *nsSinceEpoch, SpBool purge, int *sampleLoss, int *samplesRemaining);
1223
1247 SP_API SpStatus spIQSweepListGetSweep(int device, void *dst, int64_t *timestamps);
1248
1274 SP_API SpStatus spIQSweepListStartSweep(int device, int pos, void *dst, int64_t *timestamps);
1275
1286 SP_API SpStatus spIQSweepListFinishSweep(int device, int pos);
1287
1303 SP_API SpStatus spGetAudio(int device, float *audio);
1304
1346 SP_API SpStatus spGetGPSInfo(int device, SpBool refresh, SpBool *updated, int64_t *secSinceEpoch,
1347     double *latitude, double *longitude, double *altitude, char *nmea, int
1348     *nmeaLen);
1349
1361 SP_API SpStatus spSetFanThreshold(int device, float temp);
1362
1374 SP_API SpStatus spGetFanSettings(int device, float *threshold, float *voltage);
1375
1384 SP_API const char* spGetErrorString(SpStatus status);
1385
1401 SP_API const char* spGetAPIVersion();
1402
1403 #ifdef __cplusplus
1404 } // extern "C"
1405 #endif
1406
1407 #endif // #ifndef SP_API_H

```


Index

[sp_api.h](#), 27

- [SP_AUTO_ATTEN](#), 30
- [SP_FALSE](#), 30
- [SP_MAX_ATTEN](#), 30
- [SP_MAX_DEVICES](#), 30
- [SP_MAX_FREQ](#), 30
- [SP_MAX_IQ_DECIMATION](#), 31
- [SP_MAX_REF_LEVEL](#), 30
- [SP_MAX_SWEEP_QUEUE_SZ](#), 31
- [SP_MAX_SWEEP_TIME](#), 31
- [SP_MIN_FREQ](#), 30
- [SP_MIN_SWEEP_TIME](#), 30
- [SP_REAL_TIME_MAX_RBW](#), 31
- [SP_REAL_TIME_MAX_SPAN](#), 31
- [SP_REAL_TIME_MIN_RBW](#), 31
- [SP_REAL_TIME_MIN_SPAN](#), 31
- [SP_TRUE](#), 29
- [spAbort](#), 58
- [spADCOverflow](#), 32
- [spAllocationErr](#), 32
- [SpAudioType](#), 35
- [spAudioTypeAM](#), 36
- [spAudioTypeCW](#), 36
- [spAudioTypeFM](#), 36
- [spAudioTypeLSB](#), 36
- [spAudioTypeUSB](#), 36
- [SpBool](#), 32
- [spBootErr](#), 32
- [spCloseDevice](#), 37
- [spConfigure](#), 57
- [spConnectionLostErr](#), 32
- [spCPULimited](#), 32
- [SpDataType](#), 34
- [spDataType16sc](#), 35
- [spDataType32fc](#), 35
- [SpDetector](#), 33
- [spDetectorAverage](#), 33
- [spDetectorMinMax](#), 33
- [spDeviceNotFoundErr](#), 32
- [SpDeviceType](#), 32
- [spDeviceTypeSP145A](#), 32
- [spFalse](#), 32
- [spFx3RunErr](#), 32
- [spGetAPIVersion](#), 67
- [spGetAttenuator](#), 44
- [spGetAudio](#), 64
- [spGetCalDate](#), 40
- [spGetCurrentMode](#), 58
- [spGetDeviceDiagnostics](#), 39
- [spGetDeviceList](#), 36
- [spGetErrorString](#), 67
- [spGetExtReference](#), 41
- [spGetFanSettings](#), 66
- [spGetFirmwareVersion](#), 39
- [spGetGPSHoldoverInfo](#), 42
- [spGetGPSInfo](#), 65
- [spGetGPSSState](#), 42
- [spGetGPSTimebaseUpdate](#), 41
- [spGetIQ](#), 62
- [spGetIQCenterFreq](#), 50
- [spGetIQCorrection](#), 60
- [spGetIQParameters](#), 60
- [spGetIQSweepListSteps](#), 53
- [spGetPowerState](#), 38
- [spGetRealTimeFrame](#), 61
- [spGetRealTimeParameters](#), 59
- [spGetRefLevel](#), 43
- [spGetSerialNumber](#), 39
- [spGetSweep](#), 61
- [spGetSweepParameters](#), 58
- [SpGPSSState](#), 35
- [spGPSSStateDisciplined](#), 35
- [spGPSSStateLocked](#), 35
- [spGPSSStateNotPresent](#), 35
- [splInvalidCalData](#), 32
- [splInvalidConfigurationErr](#), 32
- [splInvalidDeviceErr](#), 32
- [splInvalidParameterErr](#), 32
- [splInvalidSweepPositionErr](#), 32
- [splIQSweepListFinishSweep](#), 64
- [splIQSweepListGetCorrections](#), 60
- [splIQSweepListGetSweep](#), 63
- [splIQSweepListStartSweep](#), 64
- [spMaxDevicesConnectedErr](#), 32
- [SpMode](#), 33
- [spModeAudio](#), 33
- [spModeIdle](#), 33
- [spModeIQStreaming](#), 33
- [spModeIQSweepList](#), 33
- [spModeRealTime](#), 33
- [spModeSweeping](#), 33
- [spNoError](#), 32
- [spNullPtrErr](#), 32
- [spOpenDevice](#), 36
- [spOpenDeviceBySerial](#), 37
- [SpPowerState](#), 33
- [spPowerStateOn](#), 33
- [spPowerStateStandby](#), 33

- spPresetDevice, [37](#)
- SpScale, [33](#)
- spScaleFullScale, [34](#)
- spScaleLin, [34](#)
- spScaleLog, [34](#)
- spSetAttenuator, [43](#)
- spSetAudioCenterFreq, [56](#)
- spSetAudioFilters, [56](#)
- spSetAudioFMDeemphasis, [57](#)
- spSetAudioType, [56](#)
- spSetExtReference, [40](#)
- spSetFanThreshold, [66](#)
- spSetGPSTimebaseUpdate, [41](#)
- spSetIQBandwidth, [51](#)
- spSetIQCenterFreq, [49](#)
- spSetIQDataType, [49](#)
- spSetIQExtTriggerEdge, [51](#)
- spSetIQQueueSize, [52](#)
- spSetIQSampleRate, [50](#)
- spSetIQSoftwareFilter, [50](#)
- spSetIQSweepListAtten, [55](#)
- spSetIQSweepListCorrected, [53](#)
- spSetIQSweepListDataType, [52](#)
- spSetIQSweepListFreq, [54](#)
- spSetIQSweepListRef, [54](#)
- spSetIQSweepListSampleCount, [55](#)
- spSetIQSweepListSteps, [53](#)
- spSetIQTriggerSentinel, [51](#)
- spSetPowerState, [38](#)
- spSetRealTimeCenterSpan, [47](#)
- spSetRealTimeDetector, [48](#)
- spSetRealTimeRBW, [47](#)
- spSetRealTimeScale, [48](#)
- spSetRealTimeWindow, [48](#)
- spSetRefLevel, [43](#)
- spSetSweepCenterSpan, [44](#)
- spSetSweepCoupling, [45](#)
- spSetSweepDetector, [46](#)
- spSetSweepScale, [46](#)
- spSetSweepStartStop, [45](#)
- spSetSweepWindow, [46](#)
- spSettingClamped, [32](#)
- SpStatus, [31](#)
- spTempDrift, [32](#)
- SpTriggerEdge, [35](#)
- spTriggerEdgeFalling, [35](#)
- spTriggerEdgeRising, [35](#)
- spTrue, [32](#)
- spUncalData, [32](#)
- spVideoLog, [34](#)
- spVideoPower, [34](#)
- spVideoSample, [34](#)
- SpVideoUnits, [34](#)
- spVideoVoltage, [34](#)
- spWindowBlackman, [34](#)
- spWindowFlatTop, [34](#)
- spWindowGaussian6dB, [34](#)
- spWindowHamming, [34](#)
- spWindowNuttall, [34](#)
- spWindowRect, [34](#)
- SpWindowType, [34](#)
- SP_AUTO_ATTEN
 - sp_api.h, [30](#)
- SP_FALSE
 - sp_api.h, [30](#)
- SP_MAX_ATTEN
 - sp_api.h, [30](#)
- SP_MAX_DEVICES
 - sp_api.h, [30](#)
- SP_MAX_FREQ
 - sp_api.h, [30](#)
- SP_MAX_IQ_DECIMATION
 - sp_api.h, [31](#)
- SP_MAX_REF_LEVEL
 - sp_api.h, [30](#)
- SP_MAX_SWEEP_QUEUE_SZ
 - sp_api.h, [31](#)
- SP_MAX_SWEEP_TIME
 - sp_api.h, [31](#)
- SP_MIN_FREQ
 - sp_api.h, [30](#)
- SP_MIN_SWEEP_TIME
 - sp_api.h, [30](#)
- SP_REAL_TIME_MAX_RBW
 - sp_api.h, [31](#)
- SP_REAL_TIME_MAX_SPAN
 - sp_api.h, [31](#)
- SP_REAL_TIME_MIN_RBW
 - sp_api.h, [31](#)
- SP_REAL_TIME_MIN_SPAN
 - sp_api.h, [31](#)
- SP_TRUE
 - sp_api.h, [29](#)
- spAbort
 - sp_api.h, [58](#)
- spADCOverflow
 - sp_api.h, [32](#)
- spAllocationErr
 - sp_api.h, [32](#)
- SpAudioType
 - sp_api.h, [35](#)
- spAudioTypeAM
 - sp_api.h, [36](#)
- spAudioTypeCW
 - sp_api.h, [36](#)
- spAudioTypeFM
 - sp_api.h, [36](#)
- spAudioTypeLSB
 - sp_api.h, [36](#)
- spAudioTypeUSB
 - sp_api.h, [36](#)
- SpBool
 - sp_api.h, [32](#)
- spBootErr
 - sp_api.h, [32](#)
- spCloseDevice

sp_api.h, [37](#)
spConfigure
sp_api.h, [57](#)
spConnectionLostErr
sp_api.h, [32](#)
spCPULimited
sp_api.h, [32](#)
SpDataType
sp_api.h, [34](#)
spDataType16sc
sp_api.h, [35](#)
spDataType32fc
sp_api.h, [35](#)
SpDetector
sp_api.h, [33](#)
spDetectorAverage
sp_api.h, [33](#)
spDetectorMinMax
sp_api.h, [33](#)
spDeviceNotFoundErr
sp_api.h, [32](#)
SpDeviceType
sp_api.h, [32](#)
spDeviceTypeSP145A
sp_api.h, [32](#)
spFalse
sp_api.h, [32](#)
spFx3RunErr
sp_api.h, [32](#)
spGetAPIVersion
sp_api.h, [67](#)
spGetAttenuator
sp_api.h, [44](#)
spGetAudio
sp_api.h, [64](#)
spGetCalDate
sp_api.h, [40](#)
spGetCurrentMode
sp_api.h, [58](#)
spGetDeviceDiagnostics
sp_api.h, [39](#)
spGetDeviceList
sp_api.h, [36](#)
spGetErrorString
sp_api.h, [67](#)
spGetExtReference
sp_api.h, [41](#)
spGetFanSettings
sp_api.h, [66](#)
spGetFirmwareVersion
sp_api.h, [39](#)
spGetGPSHoldoverInfo
sp_api.h, [42](#)
spGetGPSInfo
sp_api.h, [65](#)
spGetGPSSState
sp_api.h, [42](#)
spGetGPSTimebaseUpdate
sp_api.h, [41](#)
spGetIQ
sp_api.h, [62](#)
spGetIQCenterFreq
sp_api.h, [50](#)
spGetIQCorrection
sp_api.h, [60](#)
spGetIQParameters
sp_api.h, [60](#)
spGetIQSweepListSteps
sp_api.h, [53](#)
spGetPowerState
sp_api.h, [38](#)
spGetRealTimeFrame
sp_api.h, [61](#)
spGetRealTimeParameters
sp_api.h, [59](#)
spGetRefLevel
sp_api.h, [43](#)
spGetSerialNumber
sp_api.h, [39](#)
spGetSweep
sp_api.h, [61](#)
spGetSweepParameters
sp_api.h, [58](#)
SpGPSSState
sp_api.h, [35](#)
spGPSSStateDisciplined
sp_api.h, [35](#)
spGPSSStateLocked
sp_api.h, [35](#)
spGPSSStateNotPresent
sp_api.h, [35](#)
spInvalidCalData
sp_api.h, [32](#)
spInvalidConfigurationErr
sp_api.h, [32](#)
spInvalidDeviceErr
sp_api.h, [32](#)
spInvalidParameterErr
sp_api.h, [32](#)
spInvalidSweepPositionErr
sp_api.h, [32](#)
spIQSweepListFinishSweep
sp_api.h, [64](#)
spIQSweepListGetCorrections
sp_api.h, [60](#)
spIQSweepListGetSweep
sp_api.h, [63](#)
spIQSweepListStartSweep
sp_api.h, [64](#)
spMaxDevicesConnectedErr
sp_api.h, [32](#)
SpMode
sp_api.h, [33](#)
spModeAudio
sp_api.h, [33](#)
spModeIdle

sp_api.h, 33
 spModelQStreaming
 sp_api.h, 33
 spModelQSweepList
 sp_api.h, 33
 spModeRealTime
 sp_api.h, 33
 spModeSweeping
 sp_api.h, 33
 spNoError
 sp_api.h, 32
 spNullPtrErr
 sp_api.h, 32
 spOpenDevice
 sp_api.h, 36
 spOpenDeviceBySerial
 sp_api.h, 37
 SpPowerState
 sp_api.h, 33
 spPowerStateOn
 sp_api.h, 33
 spPowerStateStandby
 sp_api.h, 33
 spPresetDevice
 sp_api.h, 37
 SpScale
 sp_api.h, 33
 spScaleFullScale
 sp_api.h, 34
 spScaleLin
 sp_api.h, 34
 spScaleLog
 sp_api.h, 34
 spSetAttenuator
 sp_api.h, 43
 spSetAudioCenterFreq
 sp_api.h, 56
 spSetAudioFilters
 sp_api.h, 56
 spSetAudioFMDeemphasis
 sp_api.h, 57
 spSetAudioType
 sp_api.h, 56
 spSetExtReference
 sp_api.h, 40
 spSetFanThreshold
 sp_api.h, 66
 spSetGPSTimebaseUpdate
 sp_api.h, 41
 spSetIQBandwidth
 sp_api.h, 51
 spSetIQCenterFreq
 sp_api.h, 49
 spSetIQDataType
 sp_api.h, 49
 spSetIQExtTriggerEdge
 sp_api.h, 51
 spSetIQQueueSize
 sp_api.h, 52
 spSetIQSampleRate
 sp_api.h, 50
 spSetIQSoftwareFilter
 sp_api.h, 50
 spSetIQSweepListAtten
 sp_api.h, 55
 spSetIQSweepListCorrected
 sp_api.h, 53
 spSetIQSweepListDataType
 sp_api.h, 52
 spSetIQSweepListFreq
 sp_api.h, 54
 spSetIQSweepListRef
 sp_api.h, 54
 spSetIQSweepListSampleCount
 sp_api.h, 55
 spSetIQSweepListSteps
 sp_api.h, 53
 spSetIQTriggerSentinel
 sp_api.h, 51
 spSetPowerState
 sp_api.h, 38
 spSetRealTimeCenterSpan
 sp_api.h, 47
 spSetRealTimeDetector
 sp_api.h, 48
 spSetRealTimeRBW
 sp_api.h, 47
 spSetRealTimeScale
 sp_api.h, 48
 spSetRealTimeWindow
 sp_api.h, 48
 spSetRefLevel
 sp_api.h, 43
 spSetSweepCenterSpan
 sp_api.h, 44
 spSetSweepCoupling
 sp_api.h, 45
 spSetSweepDetector
 sp_api.h, 46
 spSetSweepScale
 sp_api.h, 46
 spSetSweepStartStop
 sp_api.h, 45
 spSetSweepWindow
 sp_api.h, 46
 spSettingClamped
 sp_api.h, 32
 SpStatus
 sp_api.h, 31
 spTempDrift
 sp_api.h, 32
 SpTriggerEdge
 sp_api.h, 35
 spTriggerEdgeFalling
 sp_api.h, 35
 spTriggerEdgeRising

- [sp_api.h](#), [35](#)
- [spTrue](#)
 - [sp_api.h](#), [32](#)
- [spUncalData](#)
 - [sp_api.h](#), [32](#)
- [spVideoLog](#)
 - [sp_api.h](#), [34](#)
- [spVideoPower](#)
 - [sp_api.h](#), [34](#)
- [spVideoSample](#)
 - [sp_api.h](#), [34](#)
- [SpVideoUnits](#)
 - [sp_api.h](#), [34](#)
- [spVideoVoltage](#)
 - [sp_api.h](#), [34](#)
- [spWindowBlackman](#)
 - [sp_api.h](#), [34](#)
- [spWindowFlatTop](#)
 - [sp_api.h](#), [34](#)
- [spWindowGaussian6dB](#)
 - [sp_api.h](#), [34](#)
- [spWindowHamming](#)
 - [sp_api.h](#), [34](#)
- [spWindowNutall](#)
 - [sp_api.h](#), [34](#)
- [spWindowRect](#)
 - [sp_api.h](#), [34](#)
- [SpWindowType](#)
 - [sp_api.h](#), [34](#)