SM API

Generated by Doxygen 1.9.4

Chapter 1

SM200 and SM435 API Reference

This documentation is a reference for the Signal Hound SM200 and SM435 (SM) spectrum analyzer programming interface (API). The API provides a set of C functions for making measurements with the SM devices. The API is C ABI compatible making is possible to be interfaced from most programming languages.

1.1 Examples

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

1.2 Measurements

This section covers the main measurements available through the API.

- Sweep Mode
- Real-Time Spectrum Analysis
- I/Q Streaming
- I/Q Sweep List / Frequency Hopping
- I/Q Segmented Captures
- I/Q Full Band
- I/Q Streaming (VRT)

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 2.3.0 - Support for SM435C.
- Version 2.2.0 – Support for SM435B.
- Version 2.1.0 – Support for SM200C.
- Version 2.0.0 – Support for SM200B, LTE I/Q sample rates, and segmented I/Q captures.
- Version 1.1.2 – First release with support for Linux operating systems (libusb backend)
- Version 1.0.3 – Official release, support for SM200A

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 sm_api.h, sm_api.lib, and sm_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
- SM library files, sm_api.h and libsm_api.so

1.4.3 Other Requirements

See the 10GbE network configuration guide for setting up a 10GbE network for SM200C/SM435C operation.

- SM200A, SM200B, SM435B
  - USB 3.0 connectivity provided through 4th generator 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.
- SM200C, SM435C
  - 10GbE connectivity with SFP+ connector and fiber cable.
  - 10GbE connectivity provided through NIC adapter card or Thunderbolt 3 to SFP+ adapter.
  - (Recommended) Quad core Intel i7 processor, 8th 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. (smSetAttenuator/smSetRefLevel) The smSetAttenuator 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 smGetGPSInfo 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 a 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 smGetGPSState function, or by examining the return status of smGetGPSInfo. 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 smGetGPSState. 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 smAbort) 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 SM cannot achieve it)

The GPS will lock automatically with a GPS antenna attached, but for the GPS to discipline the SM, it must first be enabled. To enable GPS disciplining, use the smSetGPSTimebaseUpdate 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 SM utilize a GPS lock to discipline the SM and store holdover values.

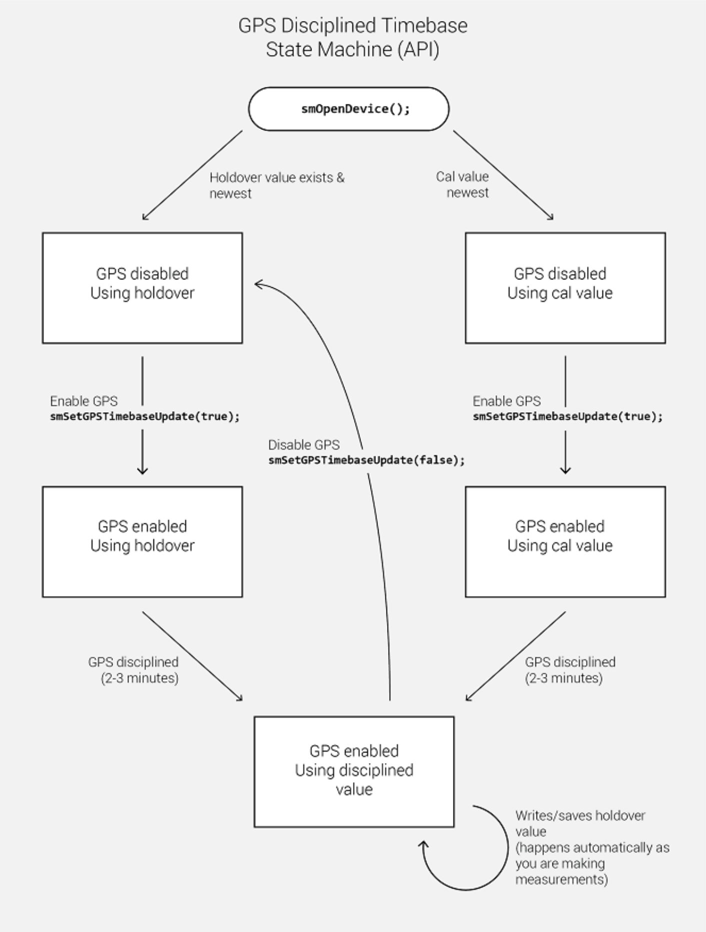

Figure 1.1 GPS Disciplining State Machine

1.6.4 Writing Messages to the GPS

Using the API, customers can write custom messages to the internal u-blox M8 GPS receiver. The user can also retrieve responses to these messages. The two functions that enable this are smWriteToGPS (writing) and smGetGPSInfo (reading). See these functions for more information.

This functionality is only available on receivers with the following firmware versions or newer.

```
SM200A: 4.5.10, SM200B: 4.5.13, SM200C: 6.6.4, SM435B: All, SM435C: All
```

Devices with this functionality will be referred to as devices with “GPS write” functionality in this document.

All messages sent to the GPS are sent over port 4 (SPI). This is the only port the customer has access to. UBX and NMEA messages can be sent. All messages are documented in the u-blox M8 GPS manual. Messages must match the frame structure documented in the u-blox manual. For example, to send a UBX message, the sync chars, class, ID, length, payload (if present), and 2-byte checksum must all be present and in the correct order in the provided message.

An example message for a “Get” UBX-CFG-NAV5 msg with empty payload is

```
msg[8] = {0xB5, 0x62, 0x06, 0x24, 0x0, 0x0, 0x2A, 0x84};
```

Responses are returned with the NMEA sentences through the smGetGPSInfo function. Responses must be parsed by the customer and can appear anywhere in the NMEA response buffer, including being split between buffers (rare).

To retrieve a response, call smGetGPSInfo with an adequately sized nmea buffer until the updated parameter is set to true, then parse the response for your message. The device does not have to have GPS lock to retrieve a response message.

See the SM C++ examples for a full example of sending and retrieving UBX messages.

A link to the u-blox M8 manual and protocol specification is below.

https://www.u-blox.com/sites/default/files/products/documents/u-blox8-M8_↩
ReceiverDescrProtSpec_%28UBX-13003221%29.pdf

1.7 GPIO

On the front panel of the SM there is a DB15 port which provides up to 8 digital logic lines available for immediate read inputs, or output lines as immediate write pins, or configurable through the API to be able to switch during sweeps and I/Q streaming.

Primary use cases for GPIO pins might be controlling an antenna assembly (switching between antennas) or interfacing attenuators.

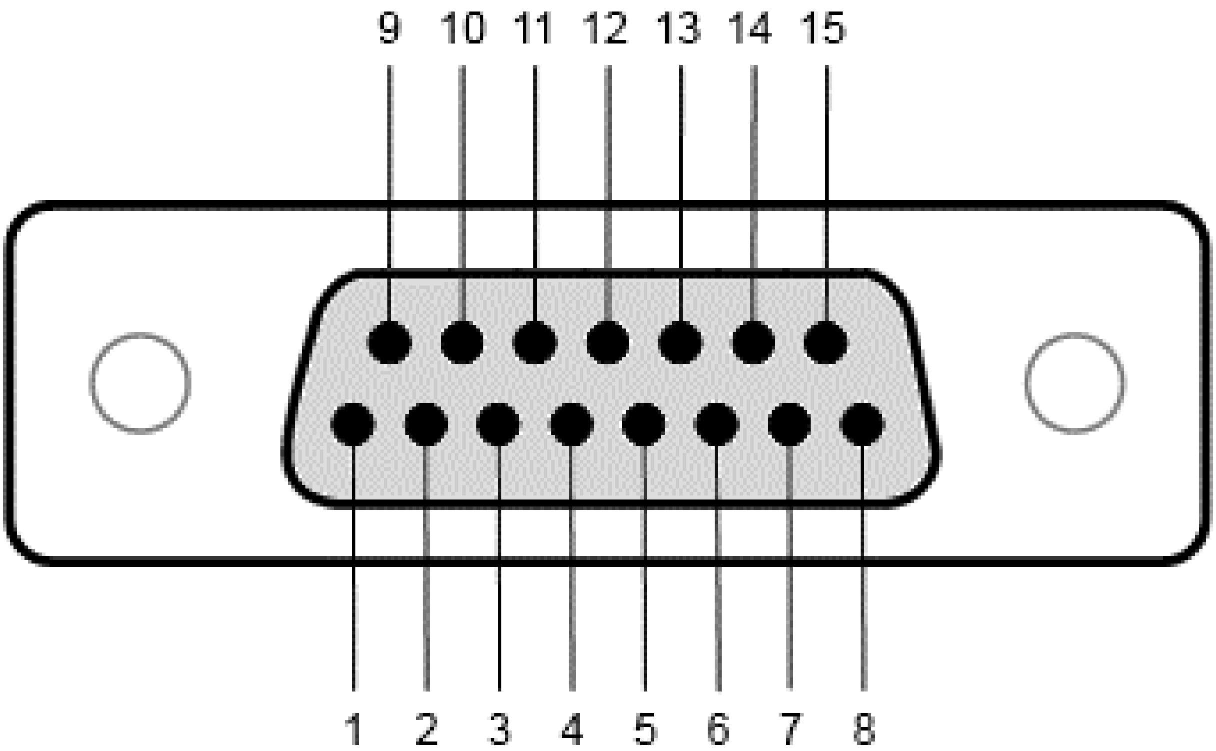

Figure 1.2 Front panel Female DB15 Port

Pinout

Pin	Description	Pin	Description
1	GPIO(0)	9	GPIO(1)
2	GPIO(2)	10	GPIO(3)
3	Vdd in (1.8 to 3.3V)	11	3.3V out (max 30 mA)
4	GND	12	SPI SCLK

Pin	Description	Pin	Description
5	SPI MOSI	13	SPI MISO
6	SPI Select	14	GPIO(4)
7	GPIO(5)	15	GPIO(6)
8	GPIO(7)	Shell	GND

GPIO pins are grouped into two nibbles (4-bits), GPIO pins [0,1,2,3] and GPIO pins [4,5,6,7]. Each nibble can be set to either read or write pins using the smSetGPIOState function. You can read or write pins using the smWriteGPIOImm or smReadGPIOImm functions. These functions can only be called when the device is in an idle state.

Additionally, there are two high speed pin switching modes that you can take advantage of. See the GPIO Sweeps and GPIO Switching (I/Q Streaming) sections for more information.

See the C++ code examples for using the set/get immediate functions.

1.7.1 GPIO Sweeps

GPIO can utilized in 2 ways with sweeps, inter-sweep, and intra-sweep. Inter-sweep GPIO changes occur in between sweeps. This can be used to change the GPIO up to a maximum of once per sweep. Intra-sweep GPIO changes occur during sweeps and is used to change the GPIO at fixed frequencies during a sweep.

1.7.1.1 Inter-sweep

Inter-sweep GPIO switching allows for rapidly sweeping a frequency range and modifying the GPIO pins for each sweep.

Inter-sweep GPIO changes are supported with the fast sweep speed and queued sweeps only. Using the smSetSweepGPIO function you can associate a GPIO setting with a given sweep. When the sweep is started the GPIO is changed just prior to the sweep occurring. There is ∼20 microseconds between the GPIO change and the sweep starting.

If inter-sweep GPIO changes are needed with normal sweep speed, avoid queued sweeps and write new GPIO settings using the smWriteGPIOImm function in between sweeps.

See the smSetSweepGPIO function description and code examples for more information.

1.7.1.2 Intra-sweep

The GPIO output pins can be configured to automatically update as the device sweeps across a specified frequency range. As the device sweeps across frequency and crosses user defined frequency boundaries, the GPIO can output specific values. The frequency boundaries and GPIO output settings are configured with smSetGPIOSweep.

This functionality is useful for controlling an antenna assembly while very quickly sweeping a large frequency range. For instance, using the GPIO to switch between different antennas to be used for different frequencies as the SM sweeps the configured span. This would be much faster than individually sweeping each antenna manually.

1.7.1.2.1 Switch Resolution Due to the digital nature of the SM device, there is a limit on the resolution on which the GPIO can change for GPIO sweeps. In normal sweep mode, the SM is processing spectrum in 39.0625MHz LO steps. This means the frequency resolution at which the GPIO can switch is 39.0625MHz. In fast sweep mode, the LO step size and GPIO switch resolution is increased to 156.25MHz.

This means the GPIO will not switch at the precise frequency you provide. The API is deterministic in choosing which frequencies to switch at, meaning, the same configuration will result in the same GPIO switch frequencies.

The API does not output or provide the actual frequencies the device switched the GPIO at. Signal Hound recommends experimenting with your setup until you find an adequate configuration.

1.7.2 GPIO Switching (I/Q Streaming)

The GPIO output pins can be configured to automatically switch at specific time intervals when the device is in I/Q streaming mode. In this mode, the user can configure a series of GPIO states to be output while the device is streaming I/Q data. This mode is useful for controlling antennas for DF and pseudo-doppler DF systems.

Up to 64 states with customizable dwell times can be configured. Dwell times can be set to a minimum of 40ns and incremented in 20ns steps. For example, a 4 antenna DF system might require a configuration like

State	GPIO Output	Dwell time (in 20ns ticks)
0	0x00	125,000
1	0x01	125,000
2	0x02	125,000
3	0x03	125,000

The configuration above configures the GPIO to switch between 4 states and dwell at each state for 2.5ms each. For a 4 antenna DF system, this configuration will cycle through all the antennas at 100Hz (10ms per revolution).

When I/Q GPIO switching is activated, the external trigger input port is disabled, and internal triggers are generated and provided in the I/Q data stream to indicate when the GPIO state has reached state zero. Triggers generated on the external trigger input port are discarded. Once GPIO switching is disable, the external trigger input is enabled again.

For more information about configuration see smSetGPIOSwitching and smSetGPIOSwitchingDisabled.

1.7.3 SPI

Through the front panel DB15 port the SM provides a SPI output interface. (SPI reads are not implemented, only SPI writes, See GPIO for the pinout) The SPI interface can be operated as output only, with a clock rate of 5.2Mbps. Between 1-4 bytes may be output through the SPI interface. Only immediate writes are available. (Direct writes while the device is idling)

The clock line idles high, and data transitions on the falling edge of the clock. It can be used to write to most SPI devices where data is latched on the rising edge of the clock.

See the C++ examples for an example of using the SPI interface.

1.8 SM435 IF Output Option

The SM435 can be configured with the IF output option. When this option is present, the SM435 device can function as a mmWave downconverter from 24-43.5GHz (configurable) and outputting it on the 10MHz output port at 1.5GHz center frequency (not configurable). Detailed specifications can be found in the SM product manual.

When the IF output option is present, the maximum frequency of the device is limited to 40.8GHz instead of 44GHz. The user is responsible for querying for the presence of the IF output option with the smHasIFOutput function and limiting the upper frequency accordingly. An IF output option device can be safely tuned above 40.8GHz without risk of damage but will not properly detect signals above this frequency.

The IF output can be enabled with the smSetIFOutput function. No other measurements can be active while the downconverter is active.

Both smSetAttenuator and smSetRefLevel can be used to control the leveling of the receiver.

1.9 Power States

The SM 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.

- smPowerStateOn, Full power state. All circuitry is enabled. Power consumption is ∼30W. The device is ready to make measurements.
- smPowerStateStandby, Estimated power consumption ∼16W. Some circuitry disabled. 100ms time to return to smPowerStateOn.

1.10 Thread Safety

The SM 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.

1.11 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.

1.12 Linux Notes

1.12.1 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.

1.12.2 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.

1.12.3 Network Devices

The SDK includes an example setup script which configures the parameters discussed below.

MTU size must be set to 9000 to enable jumbo packets.

Receive side socket buffers must be large enough to account for the amount of data each device can keep in flight. While I/Q streaming, a 10GbE SM device can keep up to ∼32MB of data in flight. We recommend setting the maximum receive buffer size to 50MB.

We recommend setting the ring buffer sizes for tx and rx to 4096. This helps reduces packet loss in certain scenarios.

1.13 Programming Languages Compatibility

The SM 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 callings the API functions from other programming languages. You will need to match the enumerated values defined in the API header file.

1.14 I/Q Acquisiton

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

1.14.1 I/Q Sample Rates

The table below outlines the available I/Q sample rates and corresponding decimations for both the USB and networked SM 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	Native Rate (10GbE) MS/s	LTE Rate (10GbE) MS/s	Downsampling (All units)
1 (Minimum)	50	61.44	200	122.88	None
2	25	30.72	100	61.44	Hardware only
4	12.5	15.36	50	30.72	Hardware only
8	6.25	7.68	25	15.36	Hardware only
16	3.125	3.84	12.5	7.68	Hardware/Software
N = {32, 64, . . . }	50 / N	61.44 / N	200 / N	122.88 / N	Hardware/Software
4096 (Maximum)	0.012207	0.015	0.048828	0.03	Hardware/Software

∗These sample rates are only available in SM200As with firmware >= 4.5.8, or with SM200Bs with firmware >= 4.5.11 combined with API version 2.0.2 or greater. All other SM devices have the LTE sample rates.

1.14.2 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 smSetIQDataType. 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

Sample Power (dBm) = 10.0 ∗ log10(re∗re + im∗im);

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

1.14.2.1 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 smGetIQCorrection.
   - `re32f *= correction;`
   - `im32f *= correction;`
3. Calculate power
   - `Sample Power (dBm) = 10.0 * log10(re32f*re32f + im32f*im32f);`

1.14.3 I/Q Filtering and Bandwidth (USB 3.0 devices)

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

1.15 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

```
Sweep Size (est.)  = (Span * WindowBW) / 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.

1.16 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	$\sigma = 0.1$

1.17 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\sm########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 smSetGPSTimebaseUpdate function.

1.18 Software Spur Rejection

Software spur rejection can be enabled only for sweep measurement modes with the smSetSweepSpurReject function.

When enabled, the SM device will sweep the frequency range twice using different LO and IF configurations. The two sweeps can be used to detect and eliminate spurious and mixer products generated by the hardware.

Software spur rejection is ideal for measuring slow moving or stationary signals of interest. It can make transient or fast-moving signals difficult to measure.

Software spur rejection is not as effective when sweeping the preselector frequency ranges when the preselector filters are enabled.

1.19 Contact Information

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

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

Chapter 2

Basic API Usage

Any application using the SM 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

How a device is opened depends on whether the device operates over USB 3.0 or 10GbE.

Opening a USB 3.0 device is done through the smOpenDevice or smOpenDeviceBySerial 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 USB SM devices connected to the PC via the smGetDeviceList function.

Opening a networked device is done through the smOpenNetworkedDevice function. All networked devices have a default network configuration that can be modified using the methods described below.

2.2 Configuring a Networked Device

There are two ways to change the network settings of a 10GbE based SM spectrum analyzer.

1. Through the smNetworkConfig∗∗∗ functions. This method allows you to set the network settings over USB. This method does not require the unit to have a 10GbE connection at the time of configuration. This method is ideal when the device may or may not be on a different subnet and cannot be addressed via the broadcast method. This method is also helpful when needing to interface several networked devices that might share a network and thus aren't individually addressable via the broadcast method.
2. Through the smBroadcastNetworkConfig function. The device must have a valid 10GbE connection. A broadcast UDP message is sent to the receiver to reconfigure its network settings. This method is ideal for single device and single use applications to quickly modify the network settings.

2.3 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 smConfigure 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.

2.4 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.

2.4.1 Stopping the Measurements

Stopping all measurements is achieved through the smAbort function. This causes the device to cancel or finish any pending operations and return to an idle state. Calling smAbort 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.

2.5 Closing the Device

When finished making measurements, you can close the device and free all resources related to the device with the smCloseDevice 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.

2.6 Recalibration

Recalibration is performed each time the device is reconfigured (smConfigure). 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 RFBoard temperature from the smGetFullDeviceDiagnostics function to detect a temperature drift and recalibrate again when you see a drift of 2-4 degrees Celsius. Using the temperature returned from smGetDeviceDiagnostics is also a valid approach but this function returns the FPGA temperature which has less correlation with the temperature corrections and tends to be more volatile.

Chapter 3

Sweep Mode

Sweep mode represents the common spectrum analyzer measurement of plotting amplitude over frequency. The API provides a simple interface through smGetSweep for acquiring single sweeps, or using smStartSweep and smFinishSweep, you can perform high throughput sweep measurements up to 1THz per second.

3.1 Example

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

```
// Configure the device for sweeps and perform a single sweep.
#include <cstdio>
#include <cstdlib>
#include <vector>
#include "sm_api.h"
void sm_example_sweep()
{
    int handle = -1;
    SmStatus status = smNoError;
    // Uncomment this to open a USB SM device
    status = smOpenDevice(&handle);
    // Uncomment this to open a networked SM device with a default network config
    //status = smOpenNetworkedDevice(&handle, SM_ADDR_ANY, SM_DEFAULT_ADDR, SM_DEFAULT_PORT);
    // Check open status
    if(status != smNoError) {
        printf("Unable to open device\n");
        exit(-1);
    }
    // Configure the sweep
    smSetRefLevel(handle, -20.0); // -20dBm reference level
    smSetSweepCenterSpan(handle, 2.45e9, 100.0e6); // ISM band
    smSetSweepCoupling(handle, 10.0e3, 10.0e3, 0.001); // 10kHz rbw/vbw, 1ms acquisition
    smSetSweepDetector(handle, smDetectorAverage, smVideoPower); // average power detector
    smSetSweepScale(handle, smScaleLog); // return sweep in dBm
    smSetSweepWindow(handle, smWindowFlatTop);
    smSetSweepSpurReject(handle, smFalse); // No software spur reject
    // Initialize the device for sweep measurement mode
    status = smConfigure(handle, smModeSweeping);
    if(status != smNoError) {
        printf("Unable to configure device\n");
        printf("%s\n", smGetErrorString(status));
        smCloseDevice(handle);
        exit(-1);
    }
    // Get the configured sweep parameters as reported by the receiver
    double actualRBW, actualVBW, actualStartFreq, binSize;
    int sweepSize;
    smGetSweepParameters(handle, &actualRBW, &actualVBW, &actualStartFreq, &binSize, &sweepSize);
    // Create memory for our sweep
    std::vector<float> sweep(sweepSize);
    // Get sweep, ignore the min sweep and the sweep time
    status = smGetSweep(handle, nullptr, sweep.data(), nullptr);
    if(status != smNoError) {
        printf("Sweep status:  %s\n", smGetErrorString(status));
    }
    // Done with the device
    smCloseDevice(handle);
}
```

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

```
Frequency of N'th element in sweep = StartFreq + N * BinSize
```

where `StartFreq` and `BinSize` are reported in the smGetSweepParameters 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.

3.5 Blocking vs. Queued Sweep Acquisition

The simple method of acquiring sweeps is to use the smGetSweep function. This function starts a sweep and blocks until the sweep is completed. This is adequate for many types of measurements but does not optimize for sweep speed. System latencies can be very large compared to total acquisition/processing time. To eliminate latencyies, you will need to take advantage of queued acquisitions.

The smStartSweep and smFinishSweep functions provide a way to eliminate latencies between sweeps which allows the device to sustain the full sweep speed throughput. Using these functions you can start up to several sweeps which ensures the receiver is continuously acquiring data for the next sweep. Using a circular buffer approach, you can ensure that there is no down time in sweep acquisition. See an example of this in the provided code examples.

Blocking and queued sweep acquisitions should not be mixed.

3.6 Sweep Speed

All SM devices have 3 sweeps speeds depending on the user's configuration. The sweep speed is determined from the sweep configuration, except in a few cases. The user can also configure the API to automatically choose the fastest sweep speed. The sweep speeds are described below.

- **Fast** – The SM sweeps $>$ 1THz per second in this mode. There are restrictions on settings which allow fast sweep. The max FFT size is 16K which limits RBW to $\sim$30-60kHz depending on the window function selected. Additionally, VBW must equal RBW, and sweep time is not selectable.
  - In fast sweep speed, the SM steps the LO in 156.25MHz steps across the desired frequency range.
- **Normal** – This mode offers better RF performance than fast sweep mode, with a sweep speed reduction of about 3X.
  - Maximum speed in this mode is $\sim$300GHz/s.
  - In normal sweep speed, the SM steps the LO in 39.0625MHz steps across the desired frequency range.
- **Slow/Narrow** – For spans below 5MHz, the API will perform sweeps in a way to achieve lower RBW/↩ VBWs. The sweep is accomplished by dwelling at a LO frequency. This is necessary for the low RBWs that accompany the narrow spans. The API will use this sweep speed below 5 MHz regardless of the users sweep speed selection.

This sweep speed can be partially controlled smSetSweepSpeed. Also see SmSweepSpeed.

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.

```
// Configure the device for real-time spectrum analysis, and retrieve the real-time sweeps and frames.
#include <cstdio>
#include <cstdlib>
#include "sm_api.h"
static void checkStatus(SmStatus status)
{
    if(status > 0) { // Warning
        printf("Warning: %s\n", smGetErrorString(status));
        return;
    } else if(status < 0) { // Error
        printf("Error: %s\n", smGetErrorString(status));
        exit(-1);
    }
}
void sm_example_real_time()
{
    int handle = -1;
    SmStatus status = smNoError;
    // Uncomment this to open a USB SM device
    status = smOpenDevice(&handle);
    // Uncomment this to open a networked SM device with a default network config
    //status = smOpenNetworkedDevice(&handle, SM_ADDR_ANY, SM_DEFAULT_ADDR, SM_DEFAULT_PORT);
    // Check open status
    checkStatus(status);
    // Configure the measurement
    smSetRefLevel(handle, -20.0); // -20dBm reference level
    smSetRealTimeCenterSpan(handle, 2.45e9, 160.0e6); // 160MHz span at 2.45GHz center freq
    smSetRealTimeRBW(handle, 30.0e3); // 30kHz min RBW with Nuttall window
    smSetRealTimeDetector(handle, smDetectorMinMax);
    smSetRealTimeScale(handle, smScaleLog, -20.0, 100.0); // On the frame, ref of -20, 100dB height
    smSetRealTimeWindow(handle, smWindowNutall);
    // Initialize the measurement
    status = smConfigure(handle, smModeRealTime);
    checkStatus(status);
    // Get the configured measurement parameters as reported by the receiver
    double actualRBW, actualStart, binSize, poi;
    int sweepSize, frameWidth, frameHeight;
    smGetRealTimeParameters(handle, &actualRBW, &sweepSize, &actualStart,
        &binSize, &frameWidth, &frameHeight, &poi);
    // Create memory for our sweep and frame
    float *sweep = new float[sweepSize];
    float *frame = new float[frameWidth * frameHeight];
    // Retrieve a series of sweeps/frames
    for(int i = 0; i < 100; i++) {
        // Retrieve just the color frame and max sweep.
        smGetRealTimeFrame(handle, frame, nullptr, nullptr, sweep, nullptr, nullptr);
        // Do something with data here
    }
```

```
    // Done with the device
    smAbort(handle);
    smCloseDevice(handle);
    // Clean up
    delete [] sweep;
    delete [] frame;
}
```

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 smGetRealTimeFrame 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 receiver ∼30 sweep/frame pairings per second.

Once the measurement is initialized via smConfigure, 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).

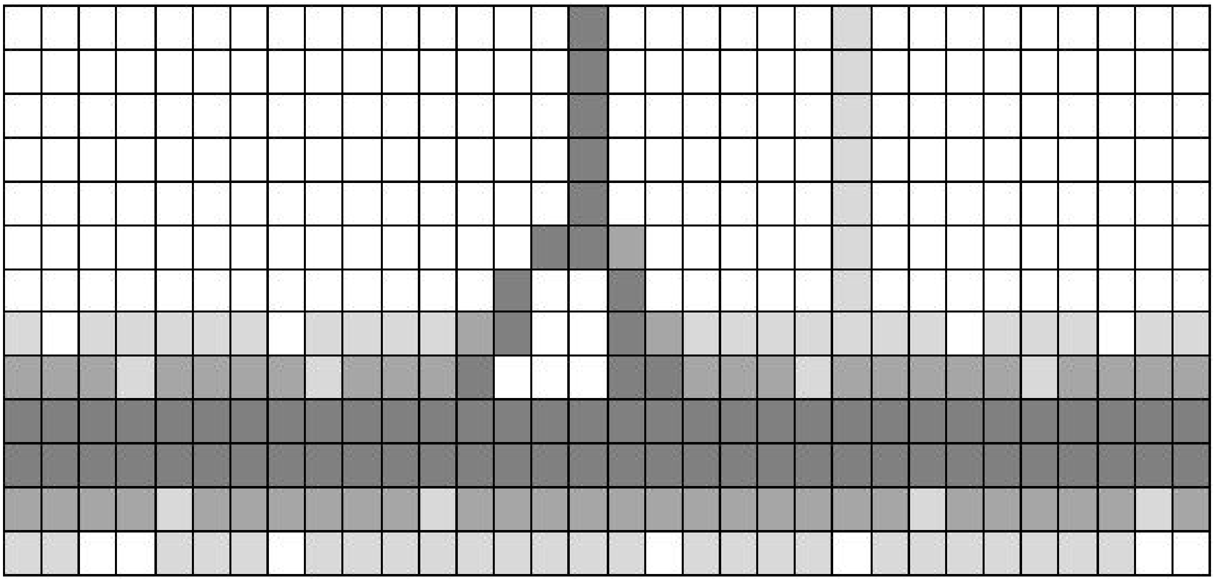

Figure 4.1 An example of a frame plotted as a gray scale image, mapping the density values between [0.0,1.0] to gray scale values between [0,255]. The frame shows a persistent CW signal near the center frequency and a short-lived CW signal.

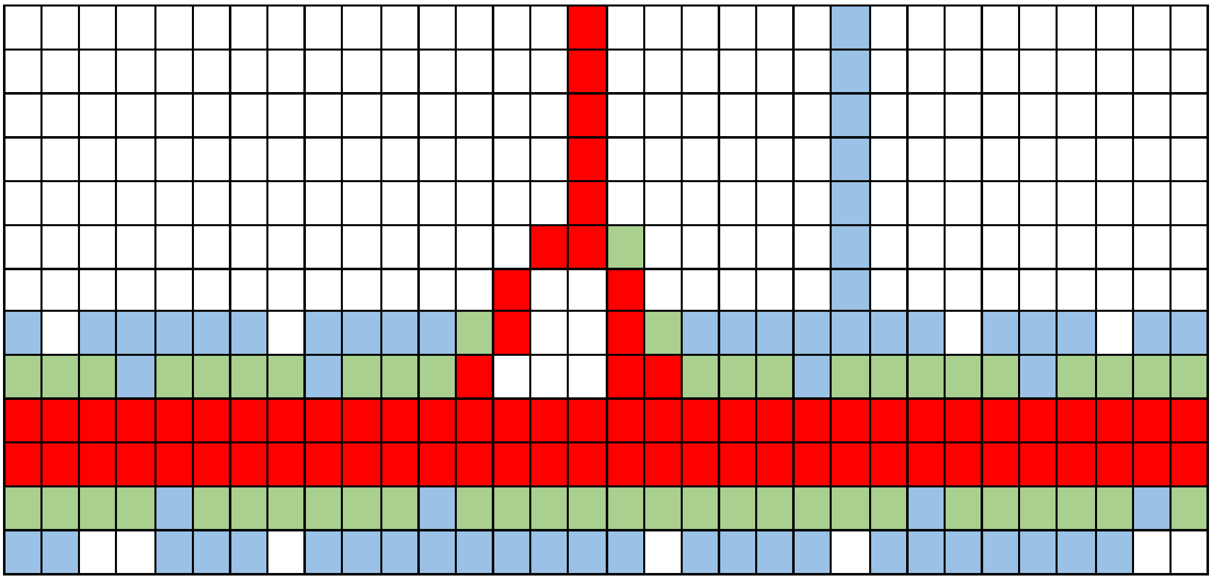

Figure 4.2 The same frame above as is plotted in Spike, where density values are mapped onto a color spectrum.

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.

```
// Configure the device for I/Q streaming and stream for a period of time
// This example assumes a USB 3.0 SM device.
#include <cstdio>
#include <cstdlib>
#include <vector>
#include "sm_api.h"
void sm_example_iq_stream()
{
    int handle = -1;
    SmStatus status = smNoError;
    // Uncomment this to open a USB SM device
    status = smOpenDevice(&handle);
    // Uncomment this to open a networked SM device with a default network config
    //status = smOpenNetworkedDevice(&handle, SM_ADDR_ANY, SM_DEFAULT_ADDR, SM_DEFAULT_PORT);
    // Check open status
    if(status != smNoError) {
        printf("Unable to open device\n");
        exit(-1);
    }
    // Configure the receiver for IQ acquisition
    smSetRefLevel(handle, -20.0); // -20 dBm reference level
    smSetIQCenterFreq(handle, 900.0e6); // 900MHz center frequency
    smSetIQBaseSampleRate(handle, smIQStreamSampleRateNative); // Use native 50MS/s base sample rate.
    smSetIQSampleRate(handle, 2); // 50 / 2 = 25MS/s IQ
    smSetIQBandwidth(handle, smTrue, 20.0e6); // 20MHz of bandwidth
    smSetIQDataType(handle, smDataType32fc);
    // Initialize the receiver with the above settings
    status = smConfigure(handle, smModeIQ);
    if(status != smNoError) {
        printf("Unable to configure device\n");
        printf("%s\n", smGetErrorString(status));
        smCloseDevice(handle);
        exit(-1);
    }
    // Query the receiver IQ stream characteristics
    // Should match what we set earlier
    double actualSampleRate, actualBandwidth;
    smGetIQParameters(handle, &actualSampleRate, &actualBandwidth);
    // Allocate memory for complex sample, IQ pairs interleaved
    int bufLen = 16384;
    std::vector<float> iqBuf(bufLen * 2);
    // Let's acquire 5 second worth of data
    int samplesNeeded = 5 * (int)actualSampleRate;
    while(samplesNeeded > 0) {
        // Notice the purge parameter is set to false, so that each time
```

```
        //  the get IQ function is called, the next contiguous block of data
        //  is returned.
        smGetIQ(handle, &iqBuf[0], bufLen, 0, 0, 0, smFalse, 0, 0);
        // Process/store data here
        // Data is interleaved 32-bit complex values
        // Need bufLen less samples
        samplesNeeded -= bufLen;
    }
    // Finished
    smCloseDevice(handle);
}
```

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.

(USB SM devices only) 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.

(10GbE SM devices only) For decimations [1,2,4,8], custom cutoff frequencies are performed on the device with no CPU penalty, and as such these filters are always active.

For all devices, 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 APIs 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 smGetIQ function.

If a trigger buffer is provided to smGetIQ, 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 Acquisiton 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.

```
// Configure and perform a single sweep in the I/Q sweep list mode.
// Configure a sweep with 3 different frequencies and different capture sizes at each frequency.
// Shows how to index the sweep.
// For a more basic example, see the 'simple' example.
// For an example which queues multiple sweeps, see the 'queue' example.
#include <complex>
#include <cstdio>
#include <cstdlib>
#include <vector>
#include "sm_api.h"
void sm_example_iq_sweep_list_single()
{
    int handle = -1;
    // Open a USB SM device
    SmStatus status = smOpenDevice(&handle);
    // Open a networked SM device
    //SmStatus status = smOpenNetworkedDevice(&handle, SM_ADDR_ANY, SM_DEFAULT_ADDR, SM_DEFAULT_PORT);
    if(status != smNoError) {
        printf("Unable to open device\n");
        exit(-1);
    }
    // The data returned should be corrected, scaled to sqrt(mW) instead of full scale.
    smSetIQSweepListCorrected(handle, smTrue);
    // Returne the data at 32-bit floating point complex values
    smSetIQSweepListDataType(handle, smDataType32fc);
    // 3 frequency steps
    smSetIQSweepListSteps(handle, 3);
    // If the GPS antenna is connected, this will instruct the device to
    // discipline to the internal GPS PPS. This will improve frequency
    // and timestamp accuracy.
    smSetGPSTimebaseUpdate(handle, smTrue);
    // Configure all three frequency steps
    // 1GHz, 1000 samples to be collected
    smSetIQSweepListFreq(handle, 0, 1.0e9);
    smSetIQSweepListRef(handle, 0, -20.0);
    smSetIQSweepListSampleCount(handle, 0, 1000);
    // 2GHz, 3000 samples to be collected
    smSetIQSweepListFreq(handle, 0, 2.0e9);
    smSetIQSweepListRef(handle, 0, -20.0);
    smSetIQSweepListSampleCount(handle, 0, 2000);
    // 3GHz, 3000 samples to be collected
    smSetIQSweepListFreq(handle, 0, 3.0e9);
    smSetIQSweepListRef(handle, 0, -20.0);
    smSetIQSweepListSampleCount(handle, 0, 3000);
    // Total samples between all 3 frequency steps
    const int totalSamples = 6000;
    // Configure the device
    smConfigure(handle, smModeIQSweepList);
```

```
    // Allocate memory for the capture
    std::vector<std::complex<float>> iq(totalSamples);
    // Memory for the timestamps
    int64_t timestamps[3];
    // Perform the sweep
    smIQSweepListGetSweep(handle, &iq[0], timestamps);
    // Example of how to index the data
    // Get pointers to the data for the 3 steps
    std::complex<float> *step1 = &iq[0];
    std::complex<float> *step2 = &iq[1000];
    std::complex<float> *step3 = &iq[3000];
    // Do something with the data here
    // The three timestamps will be the times of the samples at
    //   step1[0], step2[0], and step3[0]
    // GPS lock doesn't occur immediately upon opening.  If
    //   the GPS is cold it could take several minutes to acquire lock.  If warm, it
    //   might not lock for several seconds.  Generally the hardware will need
    //   to see at least 1 PPS after opening before lock can be determined.
    //   It will take multiple PPS after opening for disciplining to be achieved.
    // Call the smGetGPSState function to determine if the timestamps were returned
    //   under GPS lock.
    // Done with device
    smCloseDevice(handle);
}
```

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 devices native sample rate, 50MS/s for the USB SM devices and 200MS/s for the networked SM devices. The absolute fastest the SM 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 fist 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 SM 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. For the USB SM devices, N is 2048 samples, and for the networked devices, N is 6144 samples. When $<=$ N samples are requested, the device will step at the maximum rate of 8333.33 frequencies per second. This equates to $\sim$333GHz of spectrum coverage per second for the USB SM devices and $\sim$1.333THz of spectrum coverage per second for the networked SM devices.

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 ∼30ms, where as the overhead associated with performing a single I/Q sweep list acquisition is 1-5ms.

6.5 Additional Information

See I/Q Acquisiton for more information.

Chapter 7

I/Q Segmented Captures

Segmented I/Q captures allow USB 3.0 SM200B and SM435B devices to capture I/Q data with up to a 160MHz bandwidth. Complex triggering options allow 160MHz I/Q captures up to 2 seconds in length.

Segmented I/Q measurements are only available on the SM200B and SM435B devices.

10GbE devices support 160MHz I/Q bandwidth streaming and do not have segmented I/Q measurement capability.

7.1 Example

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

```
// SM200B/SM435B only.
// This example demonstrates setting up a single immediate triggered I/Q capture
//  using the 160MHz I/Q capture capabilities of the SM200B/SM435B. This examples uses the
//  convenience function for completing the capture.  See the "imm_manual" example for a full
//  example.
#include "sm_api.h"
#include <vector>
void sm_example_segmented_iq_imm_simple()
{
    // Number of I/Q samples, 50 million, 1/5th of a second
    const int CAPTURE_LEN = 50e6;
    int handle = -1;
    SmStatus status = smOpenDevice(&handle);
    if(status != smNoError) {
        // Unable to open device
        const char *errStr = smGetErrorString(status);
        return;
    }
    // Verify device has segmented I/Q capture capability
    SmDeviceType deviceType;
    smGetDeviceInfo(handle, &deviceType, 0);
    if(deviceType != smDeviceTypeSM200B && deviceType != smDeviceTypeSM435B) {
        // Invalid device type
        smCloseDevice(handle);
        return;
    }
    // Set device reference level, maximum expected input signal
    status = smSetRefLevel(handle, 0.0);
    // Configure the 160MHz capture
    status = smSetSegIQDataType(handle, smDataType32fc);
    status = smSetSegIQCenterFreq(handle, 2.45e9);
    // Setup a single segment capture
    status = smSetSegIQSegmentCount(handle, 1);
    status = smSetSegIQSegment(handle, 0, smTriggerTypeImm, 0, CAPTURE_LEN, 0.0);
    status = smConfigure(handle, smModeIQSegmentedCapture);
    if(status != smNoError) {
        // Unable to configure device
        const char *errStr = smGetErrorString(status);
        return;
    }
    // 2 floats for each I/Q sample
    std::vector<float> buf(CAPTURE_LEN * 2);
```

```
    int64_t nsSinceEpoch = 0;
    SmBool timedOut = smFalse;
    // This example uses the convenience function for completing the capture.
    // See the manual example for performing the full sequence of capture functions.
    // Immediate triggered acquisitions can't time out, so we ignore the timeout.
    status = smSegIQCaptureFull(handle, 0, buf.data(), 0, CAPTURE_LEN, &nsSinceEpoch, &timedOut);
    // Do something with data
    smAbort(handle);
    smCloseDevice(handle);
}
```

7.2 Basics

The SM200B and SM435B have an internal I/Q sample rate of 250MS/s with 160MHz of usable bandwidth. Due to the bandwidth limitations of USB 3.0 we cannot stream this full sample rate over USB to the PC. To accommodate these rates, these devices have 2GB of high-speed internal memory allowing customers to capture up to 2 seconds of I/Q data at the full 250MS/s rate.

With the API you can configure single triggered I/Q acquisitions up to 2 seconds or using the complex triggering capabilities, configure a sequence of trigger acquisitions to capture low duty cycle signals.

7.3 Acquisition Description

The sequence of a program performing segmented I/Q captures is,

1. Configure the segmented captures using the smSetSegIQ** functions.
2. Call smConfigure with the smModeIQSegmentedCapture parameter. This initializes the segmented captures with the settings set in step 1.
3. Retrieve measurement parameters with the smGetIQParameters and smSegIQGetMaxCaptures functions.
4. Retrieve measurement data using the smSegIQCapture** functions.
    - Start a trigger sequence.
    - Wait for it to finish.
    - Retrieve the measurement info and data.
    - Finish the capture. (Frees up resources)
    - Repeat (go to step a.)

7.4 Triggering

The API gives you the ability to configure a simple or complex trigger sequence. A trigger sequence is a sequence of triggers (imm/video/ext/FMT) that occur back to back, that allow re-arm times up to 25us (depending on parameters). A trigger sequence allows you to capture the signals you care about and ignore samples where signals are not present. A trigger sequence and the data it captures might look like this.

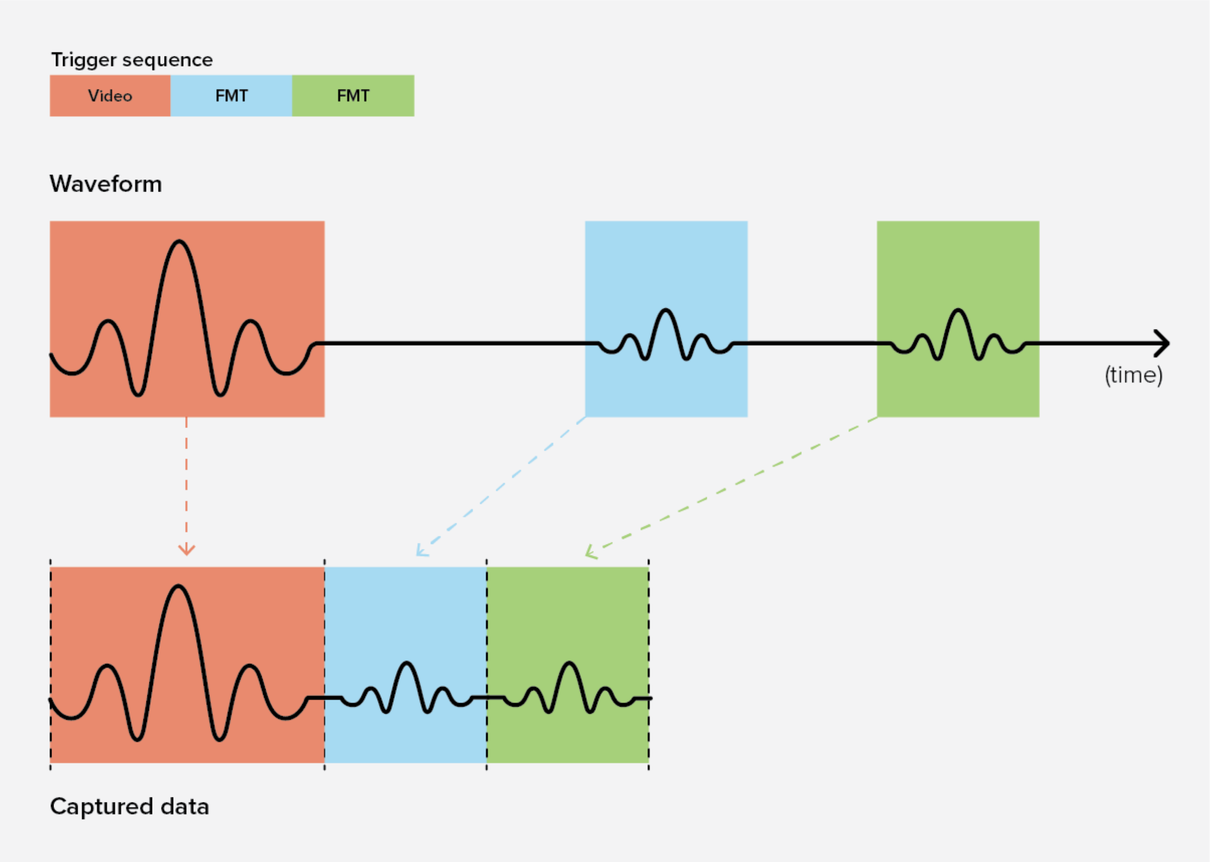

Figure 7.1 Trigger sequence captures 3 sparse signal events and ignores all other samples.

Trigger sequences can include up to 250 triggers. You are limited to one configuration of each trigger type, these types being,

- Video trigger (level/edge)
- External trigger (edge)
- Frequency mask trigger (size/mask).

For each trigger in a sequence you can configure

1. The trigger type
2. Pre-trigger length
3. Post-trigger length
4. Timeout length

Once you have configured your trigger sequence, you can queue many trigger sequences up simultaneously to increase capture throughput.

The maximum timeout length for a trigger sequence is the sum of the timeout lengths for all triggers in the sequence. The timeout period of one trigger does not start until the previous trigger has either been captured or timed-out.

Active trigger sequences must be finished before the device can be reconfigured for a different measurement, therefore it is important to avoid large timeout values if you need the device to remain responsive.

Chapter 8

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 ∼65us 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 > 1THz per second worth of spectrum.

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

8.1 Video Triggering

Video triggering full band I/Q captures is only available for the following devices.

- SM200C - FW version 7.7.5 or newer.
- SM435C - All

None of the USB SM models support full band I/Q captures with video triggering.

Chapter 9

I/Q Streaming (VRT)

The API provides the ability to stream I/Q samples contained in data packets that conform to the ANSI/VITA 49 Radio Transport (VRT) standard.

9.1 Examples

For a list of all examples, please see the *examples/cpp/vita49* folder in the SDK.

```
/*
*  Get a VRT Context packet from a Signal Hound SM Series device followed by a block of 1000 Signal Data
      packets
*
*/
#include "sm_api.h"
#include "sm_api_vrt.h"
#pragma comment(lib, "sm_api")
void getVRTPackets() {
    // Set up device
    int device = -1;
    //SmStatus status = smOpenDevice(&device); // USB
    SmStatus status = smOpenNetworkedDevice(&device, SM_ADDR_ANY, SM_DEFAULT_ADDR, SM_DEFAULT_PORT); //
      Networked
    if(status != smNoError) {
        // Could not open Sm Series device
    }
    // Set IQ parameters
    smSetIQCenterFreq(device, 3.0e9);
    smSetIQSampleRate(device, 2);
    smSetIQBandwidth(device, smTrue, 20.0e6);
    smSetRefLevel(device, -20);
    // Set VRt parameters
    smSetVrtStreamID(device, 1);
    smSetVrtPacketSize(device, 16384);
    // Configure
    status = smConfigure(device, smModeIQStreaming); // VRT mode
    if(status != smNoError) {
        // Could not configure SM Series device
    }
    // Allocate memory
    uint32_t contextWordCount;
    status = smGetVrtContextPktSize(device, &contextWordCount);
    if(status != smNoError) {
        // Could not get context packet size
    }
    const int dataPacketCount = 1000;
    uint16_t samplesPerPkt;
    uint32_t dataWordCount;
    status = smGetVrtPacketSize(device, &samplesPerPkt, &dataWordCount);
    if(status != smNoError) {
        // Could not get data packet size and word count
    }
    uint32_t wordCount = contextWordCount + dataWordCount * dataPacketCount;
    uint32_t *words = new uint32_t[wordCount];
    uint32_t *curr = words;
    // Get context packet
    uint32_t actualContextWordCount;
```

```
    status = smGetVrtContextPkt(device, curr, &actualContextWordCount);
    if(status != smNoError) {
        // Could not get context packet
    }
    if(actualContextWordCount != contextWordCount) {
        // Context packet is not the expected size
    }
    curr += contextWordCount;
    // Get data packets
    uint32_t actualDataWordCount;
    status = smGetVrtPackets(device, curr, &actualDataWordCount, dataPacketCount, smFalse);
    if(status != smNoError) {
        // Could not get data packets
    }
    if(actualDataWordCount != dataWordCount * dataPacketCount) {
        // Block of data packets is not the expected size
    }
    smCloseDevice(device);
    if(words) delete[] words;
}
```

9.1.1 Parsing Example

The parsing example demonstrates how to ingest VRT packets. The code can be used directly in projects that wish to use the VRT functionality.

It is located in *examples/cpp/vita49/gui*.

9.1.2 GUI Example

The GUI application provides a user-friendly graphical interface to easily experiment with the VRT functionality.

It is located in *examples/cpp/vita49/gui*, and uses the `Qt library`.

9.2 Basics

VRT is an open radio transport protocol used to transmit and receive sample data between devices. It is defined by the VITA 49 standard. Signal Hound uses the latest version of the standard, 49.2 (2017).

At a high level, blocks of I/Q samples and information about the receiver's state are wrapped/embedded in packets with standard formats.

9.2.1 Type and Function of Packets

VRT uses Signal Data packets and Context packets. Both types contain headers with metadata which includes a Stream Identifier and timestamp.

9.2.1.1 Signal Data Packets

Signal Data packets encapsulate variable-sized blocks of IQ data, along with a 32-bit trailer to convey additional critical information about the state of the receiver at the time the samples were obtained. For example, if the system was being overdriven this would be reported by an indicator in the trailer.

9.2.1.2 Context Packets

Context packets contain information about the receiver's settings. They are of variable size, depending on how many of the possible ∼25 fields are used. Which fields are used is communicated by the Context Indicator field, a 32-bit value which precedes the context fields. Signal Hound uses 10 of the possible context fields.

9.3 Specification

For a full, precise specification, please see the VRT Manual in the SDK.

Chapter 10

Data Structure Index

10.1 Data Structures

Here are the data structures with brief descriptions:

Chapter 11

File Index

11.1 File List

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

Chapter 12

Data Structure Documentation

12.1 SmDeviceDiagnostics Struct Reference

```
#include <sm_api.h>
```

Data Fields

- float voltage
- float currentInput
- float currentOCXO
- float current58
- float tempFPGAInternal
- float tempFPGANear
- float tempOCXO
- float tempVCO
- float tempRFBoardLO
- float tempPowerSupply

12.1.1 Detailed Description

For troubleshooting purposes. For standard diagnostics use smGetDeviceDiagnostics

12.1.2 Field Documentation

12.1.2.1 voltage

```
float voltage
```

Device voltage

12.1.2.2 currentInput

```
float currentInput
```

Input current

12.1.2.3 currentOCXO

```
float currentOCXO
```

OCXO current

12.1.2.4 current58

```
float current58
```

TODO

12.1.2.5 tempFPGAInternal

```
float tempFPGAInternal
```

FPGA core/internal temp

12.1.2.6 tempFPGANear

```
float tempFPGANear
```

Temp near FPGA

12.1.2.7 tempOCXO

```
float tempOCXO
```

OCXO temperature

12.1.2.8 tempVCO

```
float tempVCO
```

VCO temperature

12.1.2.9 tempRFBoardLO

```
float tempRFBoardLO
```

Temperature on RF board LO

12.1.2.10 tempPowerSupply

```
float tempPowerSupply
```

Power supply temperature

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

- sm_api.h

12.2 SmGPIOStep Struct Reference

```
#include <sm_api.h>
```

Data Fields

- double freq
- uint8_t mask

12.2.1 Detailed Description

Used to set the GPIO sweep. See GPIO for more information.

12.2.2 Field Documentation

12.2.2.1 freq

```
double freq
```

Frequency threshold

12.2.2.2 mask

```
uint8_t mask
```

GPIO setting for the given threshold

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

- sm_api.h

Chapter 13

File Documentation

13.1 sm_api.h File Reference

API functions for the SM435/SM200 spectrum analyzers.

Data Structures

- struct SmGPIOStep
- struct SmDeviceDiagnostics

Macros

- #define SM_TRUE (1)
- #define SM_FALSE (0)
- #define SM_MAX_DEVICES (9)
- #define SM_ADDR_ANY ("0.0.0.0")
- #define SM_DEFAULT_ADDR ("192.168.2.10")
- #define SM_DEFAULT_PORT (51665)
- #define SM_AUTO_ATTEN (-1)
- #define SM_MAX_ATTEN (6)
- #define SM_MAX_REF_LEVEL (20.0)
- #define SM_MAX_SWEEP_QUEUE_SZ (16)
- #define SM200_MIN_FREQ (100.0e3)
- #define SM200_MAX_FREQ (20.6e9)
- #define SM435_MIN_FREQ (100.0e3)
- #define SM435_MAX_FREQ (44.2e9)
- #define SM435_MAX_FREQ_IF_OPT (40.8e9)
- #define SM_MAX_IQ_DECIMATION (4096)
- #define SM_PRESELECTOR_MAX_FREQ (645.0e6)
- #define SM_FAST_SWEEP_MIN_RBW (30.0e3)
- #define SM_REAL_TIME_MIN_SPAN (200.0e3)
- #define SM_REAL_TIME_MAX_SPAN (160.0e6)
- #define SM_MIN_SWEEP_TIME (1.0e-6)
- #define SM_MAX_SWEEP_TIME (100.0)
- #define SM_SPI_MAX_BYTES (4)
- #define SM_GPIO_SWEEP_MAX_STEPS (64)

- #define SM_GPIO_SWITCH_MAX_STEPS (64)
- #define SM_GPIO_SWITCH_MIN_COUNT (2)
- #define SM_GPIO_SWITCH_MAX_COUNT (4194303 - 1)
- #define SM_TEMP_WARNING (95.0)
- #define SM_TEMP_MAX (102.0)
- #define SM_MAX_SEGMENTED_IQ_SEGMENTS (250)
- #define SM_MAX_SEGMENTED_IQ_SAMPLES (520e6)
- #define SM435_IF_OUTPUT_FREQ (1.5e9)
- #define SM435_IF_OUTPUT_MIN_FREQ (24.0e9)
- #define SM435_IF_OUTPUT_MAX_FREQ (43.5e9)

Enumerations

- enum SmStatus { }
- enum SmDataType { smDataType32fc , smDataType16sc }
- enum SmMode {
  smModeIdle = 0 , smModeSweeping = 1 , smModeRealTime = 2 , smModeIQStreaming = 3 ,
  smModeIQSegmentedCapture = 5 , smModeIQSweepList = 6 , smModeAudio = 4 , **smModeIQ** = 3 }
- enum SmSweepSpeed { smSweepSpeedAuto = 0 , smSweepSpeedNormal = 1 , smSweepSpeedFast = 2 }
- enum SmIQStreamSampleRate { smIQStreamSampleRateNative = 0 , smIQStreamSampleRateLTE = 1 }
- enum SmPowerState { smPowerStateOn = 0 , smPowerStateStandby = 1 }
- enum SmDetector { smDetectorAverage = 0 , smDetectorMinMax = 1 }
- enum SmScale { smScaleLog = 0 , smScaleLin = 1 , smScaleFullScale = 2 }
- enum SmVideoUnits { smVideoLog = 0 , smVideoVoltage = 1 , smVideoPower = 2 , smVideoSample = 3 }
- enum SmWindowType {
  smWindowFlatTop = 0 , smWindowNutall = 2 , smWindowBlackman = 3 , smWindowHamming = 4 ,
  smWindowGaussian6dB = 5 , smWindowRect = 6 }
- enum SmTriggerType { smTriggerTypeImm = 0 , smTriggerTypeVideo = 1 , smTriggerTypeExt = 2 ,
  smTriggerTypeFMT = 3 }
- enum SmTriggerEdge { smTriggerEdgeRising = 0 , smTriggerEdgeFalling = 1 }
- enum SmBool { smFalse = 0 , smTrue = 1 }
- enum SmGPIOState { smGPIOStateOutput = 0 , smGPIOStateInput = 1 }
- enum SmReference { smReferenceUseInternal = 0 , smReferenceUseExternal = 1 }
- enum SmDeviceType {
  smDeviceTypeSM200A = 0 , smDeviceTypeSM200B = 1 , smDeviceTypeSM200C = 2 , smDeviceTypeSM435B
  = 3 ,
  smDeviceTypeSM435C = 4 }
- enum SmAudioType {
  smAudioTypeAM = 0 , smAudioTypeFM = 1 , smAudioTypeUSB = 2 , smAudioTypeLSB = 3 ,
  smAudioTypeCW = 4 }
- enum SmGPSState { smGPSStateNotPresent = 0 , smGPSStateLocked = 1 , smGPSStateDisciplined = 2 }

Functions

- SM_API SmStatus smGetDeviceList (int ∗serials, int ∗deviceCount)
- SM_API SmStatus smGetDeviceList2 (int ∗serials, SmDeviceType ∗deviceTypes, int ∗deviceCount)
- SM_API SmStatus smOpenDevice (int ∗device)
- SM_API SmStatus smOpenDeviceBySerial (int ∗device, int serialNumber)
- SM_API SmStatus smOpenNetworkedDevice (int ∗device, const char ∗hostAddr, const char ∗deviceAddr, uint16_t port)
- SM_API SmStatus smCloseDevice (int device)
- SM_API SmStatus smPreset (int device)
- SM_API SmStatus smPresetSerial (int serialNumber)

- SM_API SmStatus smNetworkedSpeedTest (int device, double durationSeconds, double *bytesPerSecond)
- SM_API SmStatus smGetDeviceInfo (int device, SmDeviceType *deviceType, int *serialNumber)
- SM_API SmStatus smGetFirmwareVersion (int device, int *major, int *minor, int *revision)
- SM_API SmStatus smHasIFOutput (int device, SmBool *present)
- SM_API SmStatus smGetDeviceDiagnostics (int device, float *voltage, float *current, float *temperature)
- SM_API SmStatus smGetFullDeviceDiagnostics (int device, SmDeviceDiagnostics *diagnostics)
- SM_API SmStatus smGetSFPDiagnostics (int device, float *temp, float *voltage, float *txPower, float *rx←Power)
- SM_API SmStatus smSetPowerState (int device, SmPowerState powerState)
- SM_API SmStatus smGetPowerState (int device, SmPowerState *powerState)
- SM_API SmStatus smSetAttenuator (int device, int atten)
- SM_API SmStatus smGetAttenuator (int device, int *atten)
- SM_API SmStatus smSetRefLevel (int device, double refLevel)
- SM_API SmStatus smGetRefLevel (int device, double *refLevel)
- SM_API SmStatus smSetPreselector (int device, SmBool enabled)
- SM_API SmStatus smGetPreselector (int device, SmBool *enabled)
- SM_API SmStatus smSetGPIOState (int device, SmGPIOState lowerState, SmGPIOState upperState)
- SM_API SmStatus smGetGPIOState (int device, SmGPIOState *lowerState, SmGPIOState *upperState)
- SM_API SmStatus smWriteGPIOImm (int device, uint8_t data)
- SM_API SmStatus smReadGPIOImm (int device, uint8_t *data)
- SM_API SmStatus smWriteSPI (int device, uint32_t data, int byteCount)
- SM_API SmStatus smSetGPIOSweepDisabled (int device)
- SM_API SmStatus smSetGPIOSweep (int device, SmGPIOStep *steps, int stepCount)
- SM_API SmStatus smSetGPIOSwitchingDisabled (int device)
- SM_API SmStatus smSetGPIOSwitching (int device, uint8_t *gpio, uint32_t *counts, int gpioSteps)
- SM_API SmStatus smSetExternalReference (int device, SmBool enabled)
- SM_API SmStatus smGetExternalReference (int device, SmBool *enabled)
- SM_API SmStatus smSetReference (int device, SmReference reference)
- SM_API SmStatus smGetReference (int device, SmReference *reference)
- SM_API SmStatus smSetGPSTimebaseUpdate (int device, SmBool enabled)
- SM_API SmStatus smGetGPSTimebaseUpdate (int device, SmBool *enabled)
- SM_API SmStatus smGetGPSHoldoverInfo (int device, SmBool *usingGPSHoldover, uint64_t *last←HoldoverTime)
- SM_API SmStatus smGetGPSState (int device, SmGPSState *GPSState)
- SM_API SmStatus smSetSweepSpeed (int device, SmSweepSpeed sweepSpeed)
- SM_API SmStatus smSetSweepCenterSpan (int device, double centerFreqHz, double spanHz)
- SM_API SmStatus smSetSweepStartStop (int device, double startFreqHz, double stopFreqHz)
- SM_API SmStatus smSetSweepCoupling (int device, double rbw, double vbw, double sweepTime)
- SM_API SmStatus smSetSweepDetector (int device, SmDetector detector, SmVideoUnits videoUnits)
- SM_API SmStatus smSetSweepScale (int device, SmScale scale)
- SM_API SmStatus smSetSweepWindow (int device, SmWindowType window)
- SM_API SmStatus smSetSweepSpurReject (int device, SmBool spurRejectEnabled)
- SM_API SmStatus smSetRealTimeCenterSpan (int device, double centerFreqHz, double spanHz)
- SM_API SmStatus smSetRealTimeRBW (int device, double rbw)
- SM_API SmStatus smSetRealTimeDetector (int device, SmDetector detector)
- SM_API SmStatus smSetRealTimeScale (int device, SmScale scale, double frameRef, double frameScale)
- SM_API SmStatus smSetRealTimeWindow (int device, SmWindowType window)
- SM_API SmStatus smSetIQBaseSampleRate (int device, SmIQStreamSampleRate sampleRate)
- SM_API SmStatus smSetIQDataType (int device, SmDataType dataType)
- SM_API SmStatus smSetIQCenterFreq (int device, double centerFreqHz)
- SM_API SmStatus smGetIQCenterFreq (int device, double *centerFreqHz)
- SM_API SmStatus smSetIQSampleRate (int device, int decimation)
- SM_API SmStatus smSetIQBandwidth (int device, SmBool enableSoftwareFilter, double bandwidth)
- SM_API SmStatus smSetIQExtTriggerEdge (int device, SmTriggerEdge edge)
- SM_API SmStatus smSetIQTriggerSentinel (double sentinelValue)

- SM_API SmStatus smSetIQQueueSize (int device, float ms)
- SM_API SmStatus smSetIQSweepListDataType (int device, SmDataType dataType)
- SM_API SmStatus smSetIQSweepListCorrected (int device, SmBool corrected)
- SM_API SmStatus smSetIQSweepListSteps (int device, int steps)
- SM_API SmStatus smGetIQSweepListSteps (int device, int ∗steps)
- SM_API SmStatus smSetIQSweepListFreq (int device, int step, double freq)
- SM_API SmStatus smSetIQSweepListRef (int device, int step, double level)
- SM_API SmStatus smSetIQSweepListAtten (int device, int step, int atten)
- SM_API SmStatus smSetIQSweepListSampleCount (int device, int step, uint32_t samples)
- SM_API SmStatus smSetSegIQDataType (int device, SmDataType dataType)
- SM_API SmStatus smSetSegIQCenterFreq (int device, double centerFreqHz)
- SM_API SmStatus smSetSegIQVideoTrigger (int device, double triggerLevel, SmTriggerEdge triggerEdge)
- SM_API SmStatus smSetSegIQExtTrigger (int device, SmTriggerEdge extTriggerEdge)
- SM_API SmStatus smSetSegIQFMTParams (int device, int fftSize, const double ∗frequencies, const double ∗ampls, int count)
- SM_API SmStatus smSetSegIQSegmentCount (int device, int segmentCount)
- SM_API SmStatus smSetSegIQSegment (int device, int segment, SmTriggerType triggerType, int preTrigger, int captureSize, double timeoutSeconds)
- SM_API SmStatus smSetAudioCenterFreq (int device, double centerFreqHz)
- SM_API SmStatus smSetAudioType (int device, SmAudioType audioType)
- SM_API SmStatus smSetAudioFilters (int device, double ifBandwidth, double audioLpf, double audioHpf)
- SM_API SmStatus smSetAudioFMDeemphasis (int device, double deemphasis)
- SM_API SmStatus smConfigure (int device, SmMode mode)
- SM_API SmStatus smGetCurrentMode (int device, SmMode ∗mode)
- SM_API SmStatus smAbort (int device)
- SM_API SmStatus smGetSweepParameters (int device, double ∗actualRBW, double ∗actualVBW, double ∗actualStartFreq, double ∗binSize, int ∗sweepSize)
- SM_API SmStatus smGetRealTimeParameters (int device, double ∗actualRBW, int ∗sweepSize, double ∗actualStartFreq, double ∗binSize, int ∗frameWidth, int ∗frameHeight, double ∗poi)
- SM_API SmStatus smGetIQParameters (int device, double ∗sampleRate, double ∗bandwidth)
- SM_API SmStatus smGetIQCorrection (int device, float ∗scale)
- SM_API SmStatus smIQSweepListGetCorrections (int device, float ∗corrections)
- SM_API SmStatus smSegIQGetMaxCaptures (int device, int ∗maxCaptures)
- SM_API SmStatus smGetSweep (int device, float ∗sweepMin, float ∗sweepMax, int64_t ∗nsSinceEpoch)
- SM_API SmStatus smSetSweepGPIO (int device, int pos, uint8_t data)
- SM_API SmStatus smStartSweep (int device, int pos)
- SM_API SmStatus smFinishSweep (int device, int pos, float ∗sweepMin, float ∗sweepMax, int64_t ∗ns←SinceEpoch)
- SM_API SmStatus smGetRealTimeFrame (int device, float ∗colorFrame, float ∗alphaFrame, float ∗sweepMin, float ∗sweepMax, int ∗frameCount, int64_t ∗nsSinceEpoch)
- SM_API SmStatus smGetIQ (int device, void ∗iqBuf, int iqBufSize, double ∗triggers, int triggerBufSize, int64←_t ∗nsSinceEpoch, SmBool purge, int ∗sampleLoss, int ∗samplesRemaining)
- SM_API SmStatus smIQSweepListGetSweep (int device, void ∗dst, int64_t ∗timestamps)
- SM_API SmStatus smIQSweepListStartSweep (int device, int pos, void ∗dst, int64_t ∗timestamps)
- SM_API SmStatus smIQSweepListFinishSweep (int device, int pos)
- SM_API SmStatus smSegIQCaptureStart (int device, int capture)
- SM_API SmStatus smSegIQCaptureWait (int device, int capture)
- SM_API SmStatus smSegIQCaptureWaitAsync (int device, int capture, SmBool ∗completed)
- SM_API SmStatus smSegIQCaptureTimeout (int device, int capture, int segment, SmBool ∗timedOut)
- SM_API SmStatus smSegIQCaptureTime (int device, int capture, int segment, int64_t ∗nsSinceEpoch)
- SM_API SmStatus smSegIQCaptureRead (int device, int capture, int segment, void ∗iq, int offset, int len)
- SM_API SmStatus smSegIQCaptureFinish (int device, int capture)
- SM_API SmStatus smSegIQCaptureFull (int device, int capture, void ∗iq, int offset, int len, int64_t ∗nsSince←Epoch, SmBool ∗timedOut)

- SM_API SmStatus smSegIQLTEResample (float *input, int inputLen, float *output, int *outputLen, bool clearDelayLine)
- SM_API SmStatus smSetIQFullBandAtten (int device, int atten)
- SM_API SmStatus smSetIQFullBandCorrected (int device, SmBool corrected)
- SM_API SmStatus smSetIQFullBandSamples (int device, int samples)
- SM_API SmStatus smSetIQFullBandTriggerType (int device, SmTriggerType triggerType)
- SM_API SmStatus smSetIQFullBandVideoTrigger (int device, double triggerLevel)
- SM_API SmStatus smSetIQFullBandTriggerTimeout (int device, double triggerTimeout)
- SM_API SmStatus smGetIQFullBand (int device, float *iq, int freq)
- SM_API SmStatus smGetIQFullBandSweep (int device, float *iq, int startIndex, int stepSize, int steps)
- SM_API SmStatus smGetAudio (int device, float *audio)
- SM_API SmStatus smGetGPSInfo (int device, SmBool refresh, SmBool *updated, int64_t *secSinceEpoch, double *latitude, double *longitude, double *altitude, char *nmea, int *nmeaLen)
- SM_API SmStatus smWriteToGPS (int device, const uint8_t *mem, int len)
- SM_API SmStatus smSetFanThreshold (int device, int temp)
- SM_API SmStatus smGetFanThreshold (int device, int *temp)
- SM_API SmStatus smSetIFOutput (int device, double frequency)
- SM_API SmStatus smGetCalDate (int device, uint64_t *lastCalDate)
- SM_API SmStatus smBroadcastNetworkConfig (const char *hostAddr, const char *deviceAddr, uint16_t port, SmBool nonVolatile)
- SM_API SmStatus smNetworkConfigGetDeviceList (int *serials, int *deviceCount)
- SM_API SmStatus smNetworkConfigOpenDevice (int *device, int serialNumber)
- SM_API SmStatus smNetworkConfigCloseDevice (int device)
- SM_API SmStatus smNetworkConfigGetMAC (int device, char *mac)
- SM_API SmStatus smNetworkConfigSetIP (int device, const char *addr, SmBool nonVolatile)
- SM_API SmStatus smNetworkConfigGetIP (int device, char *addr)
- SM_API SmStatus smNetworkConfigSetPort (int device, int port, SmBool nonVolatile)
- SM_API SmStatus smNetworkConfigGetPort (int device, int *port)
- SM_API const char * smGetAPIVersion ()
- SM_API const char * smGetErrorString (SmStatus status)

13.1.1 Detailed Description

API functions for the SM435/SM200 spectrum analyzers.

This is the main file for user accessible functions for controlling the SM435/SM200 spectrum analyzers.

13.1.2 Macro Definition Documentation

13.1.2.1 SM_TRUE

```
#define SM_TRUE (1)
```

Used for boolean true when integer parameters are being used. Also see SmBool.

13.1.2.2 SM_FALSE

```
#define SM_FALSE (0)
```

Used for boolean false when integer parameters are being used. Also see SmBool.

13.1.2.3 SM_MAX_DEVICES

```
#define SM_MAX_DEVICES (9)
```

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

13.1.2.4 SM_ADDR_ANY

```
#define SM_ADDR_ANY ("0.0.0.0")
```

Convenience host address for connecting networked devices.

13.1.2.5 SM_DEFAULT_ADDR

```
#define SM_DEFAULT_ADDR ("192.168.2.10")
```

Default device IP address for networked devices.

13.1.2.6 SM_DEFAULT_PORT

```
#define SM_DEFAULT_PORT (51665)
```

Default port number for networked devices.

13.1.2.7 SM_AUTO_ATTEN

```
#define SM_AUTO_ATTEN (-1)
```

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

13.1.2.8 SM_MAX_ATTEN

```
#define SM_MAX_ATTEN (6)
```

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

13.1.2.9 SM_MAX_REF_LEVEL

```
#define SM_MAX_REF_LEVEL (20.0)
```

Maximum reference level in dBm

13.1.2.10 SM_MAX_SWEEP_QUEUE_SZ

```
#define SM_MAX_SWEEP_QUEUE_SZ (16)
```

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

13.1.2.11 SM200_MIN_FREQ

```
#define SM200_MIN_FREQ (100.0e3)
```

Min frequency for sweeps, and min center frequency for I/Q measurements for SM200 devices.

13.1.2.12 SM200_MAX_FREQ

```
#define SM200_MAX_FREQ (20.6e9)
```

Max frequency for sweeps, and max center frequency for I/Q measurements for SM200 devices.

13.1.2.13 SM435_MIN_FREQ

```
#define SM435_MIN_FREQ (100.0e3)
```

Min frequency for sweeps, and min center frequency for I/Q measurements for SM435 devices.

13.1.2.14 SM435_MAX_FREQ

```
#define SM435_MAX_FREQ (44.2e9)
```

Max frequency for sweeps, and max center frequency for I/Q measurements for SM435 devices.

13.1.2.15 SM435_MAX_FREQ_IF_OPT

```
#define SM435_MAX_FREQ_IF_OPT (40.8e9)
```

Max frequency for sweeps, and max center frequency for I/Q measurements for SM435 devices with the IF output option.

13.1.2.16 SM_MAX_IQ_DECIMATION

```
#define SM_MAX_IQ_DECIMATION (4096)
```

Max decimation for I/Q streaming.

13.1.2.17 SM_PRESELECTOR_MAX_FREQ

```
#define SM_PRESELECTOR_MAX_FREQ (645.0e6)
```

The frequency at which the manually controlled preselector filters end. Past this frequency, the preselector filters are always enabled.

13.1.2.18 SM_FAST_SWEEP_MIN_RBW

```
#define SM_FAST_SWEEP_MIN_RBW (30.0e3)
```

Minimum RBW in Hz for fast sweep with Nuttall window.

13.1.2.19 SM_REAL_TIME_MIN_SPAN

```
#define SM_REAL_TIME_MIN_SPAN (200.0e3)
```

Min span for device configured in real-time measurement mode

13.1.2.20 SM_REAL_TIME_MAX_SPAN

```
#define SM_REAL_TIME_MAX_SPAN (160.0e6)
```

Max span for device configured in real-time measurement mode

13.1.2.21 SM_MIN_SWEEP_TIME

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

Min sweep time in seconds. See smSetSweepCoupling.

13.1.2.22 SM_MAX_SWEEP_TIME

```
#define SM_MAX_SWEEP_TIME (100.0)
```

Max sweep time in seconds. See smSetSweepCoupling.

13.1.2.23 SM_SPI_MAX_BYTES

```
#define SM_SPI_MAX_BYTES (4)
```

Max number of bytes per SPI transfer.

13.1.2.24 SM_GPIO_SWEEP_MAX_STEPS

```
#define SM_GPIO_SWEEP_MAX_STEPS (64)
```

Max number of freq/state pairs for GPIO sweeps.

13.1.2.25 SM_GPIO_SWITCH_MAX_STEPS

```
#define SM_GPIO_SWITCH_MAX_STEPS (64)
```

Max number of GPIO states for I/Q streaming.

13.1.2.26 SM_GPIO_SWITCH_MIN_COUNT

```
#define SM_GPIO_SWITCH_MIN_COUNT (2)
```

Min length for GPIO state for I/Q streaming, in counts.

13.1.2.27 SM_GPIO_SWITCH_MAX_COUNT

```
#define SM_GPIO_SWITCH_MAX_COUNT (4194303 - 1)
```

Max length for GPIO state for I/Q streaming, in counts.

13.1.2.28 SM_TEMP_WARNING

```
#define SM_TEMP_WARNING (95.0)
```

FPGA core temp should not exceed this value, in C.

13.1.2.29 SM_TEMP_MAX

```
#define SM_TEMP_MAX (102.0)
```

FPGA shutdown temp, in C.

13.1.2.30 SM_MAX_SEGMENTED_IQ_SEGMENTS

```
#define SM_MAX_SEGMENTED_IQ_SEGMENTS (250)
```

Segmented I/Q captures, max segments.

13.1.2.31 SM_MAX_SEGMENTED_IQ_SAMPLES

```
#define SM_MAX_SEGMENTED_IQ_SAMPLES (520e6)
```

Segmented I/Q captures, max samples for all segments combined.

13.1.2.32 SM435_IF_OUTPUT_FREQ

```
#define SM435_IF_OUTPUT_FREQ (1.5e9)
```

IF output, output frequency. IF output option devices only.

13.1.2.33 SM435_IF_OUTPUT_MIN_FREQ

```
#define SM435_IF_OUTPUT_MIN_FREQ (24.0e9)
```

Min IF output, input frequency. IF output option devices only.

13.1.2.34 SM435_IF_OUTPUT_MAX_FREQ

```
#define SM435_IF_OUTPUT_MAX_FREQ (43.5e9)
```

Max IF output, input frequency. IF output option devices only.

13.1.3 Enumeration Type Documentation

13.1.3.1 SmStatus

```
enum SmStatus
```

Status code returned from all SM API functions.

Enumerator

smInvalidCalibrationFileErr	Calibration file unable to be used with the API
smInvalidCenterFreqErr	Invalid center frequency specified
smInvalidIQDecimationErr	I/Q decimation value provided not a valid value
smJESDErr	FPGA/initialization error
smNetworkErr	Socket/network error
smFx3RunErr	If the core FX3 program fails to run
smMaxDevicesConnectedErr	Only can connect up to SM_MAX_DEVICES receivers
smFPGABootErr	FPGA boot error
smBootErr	Boot error
smGpsNotLockedErr	Requesting GPS information when the GPS is not locked
smVersionMismatchErr	Invalid API version for target device, update API
smAllocationErr	Unable to allocate resources needed to configure the measurement mode
smSyncErr	Returned when the device detects framing issue on measurement data Measurement results are likely invalid. Device should be preset/power cycled
smInvalidSweepPosition	Invalid or already active sweep position
smInvalidConfigurationErr	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.

Enumerator

smConnectionLostErr	Device disconnected, likely USB error detected
smInvalidParameterErr	Required parameter found to have invalid value
smNullPtrErr	One or more required pointer parameters were null
smInvalidDeviceErr	User specified invalid device index
smDeviceNotFoundErr	Unable to open device
smNoError	Function returned successfully
smSettingClamped	One or more of the provided settings were adjusted
smAdcOverflow	Measurement includes data which caused an ADC overload (clipping/compression)
smUncalData	Measurement is uncalibrated, overrides ADC overflow
smTempDriftWarning	Temperature drift occured, measurements uncalibrated, reconfigure the device
smSpanExceedsPreselector	Warning when the preselector span is smaller than the user selected span
smTempHighWarning	Warning when the internal temperature gets too hot. The device is close to shutting down
smCpuLimited	Returned when the API was unable to keep up with the necessary processing
smUpdateAPI	Returned when the API detects a device with newer features than what was available when this version of the API was released. Suggested fix, update the API.
smInvalidCalData	Calibration data potentially corrupt

13.1.3.2 SmDataType

```
enum SmDataType
```

Specifies a data type for data returned from the API

Enumerator

smDataType32fc	32-bit complex floats
smDataType16sc	16-bit complex shorts

13.1.3.3 SmMode

```
enum SmMode
```

Measurement mode

Enumerator

smModeIdle	Idle, no measurement
smModeSweeping	Swept spectrum analysis
smModeRealTime	Real-time spectrum analysis

Enumerator

smModeIQStreaming	I/Q streaming
smModeIQSegmentedCapture	SM200B/SM435B wide band I/Q capture
smModeIQSweepList	I/Q sweep list / frequency hopping
smModeAudio	Audio demod

13.1.3.4 SmSweepSpeed

```
enum SmSweepSpeed
```

Sweep speed

Enumerator

smSweepSpeedAuto	Automatically choose the fastest sweep speed while maintaining customer requested settings
smSweepSpeedNormal	Use standard sweep speed, always available
smSweepSpeedFast	Choose fast sweep speed whenever possible, possibly ignoring some requested settings

13.1.3.5 SmIQStreamSampleRate

```
enum SmIQStreamSampleRate
```

Base sample rate used for I/Q streaming. See I/Q Acquisiton for more information.

Enumerator

smIQStreamSampleRateNative	Use device native sample rate
smIQStreamSampleRateLTE	Use LTE sample rates

13.1.3.6 SmPowerState

```
enum SmPowerState
```

Specifies device power state. See Power States for more information.

Enumerator

smPowerStateOn	On
smPowerStateStandby	Standby

13.1.3.7 SmDetector

```
enum SmDetector
```

Detector used for sweep and real-time spectrum analysis.

Enumerator

smDetectorAverage	Average
smDetectorMinMax	Min/Max

13.1.3.8 SmScale

```
enum SmScale
```

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

Enumerator

smScaleLog	dBm
smScaleLin	mV
smScaleFullScale	Log scale, no corrections

13.1.3.9 SmVideoUnits

```
enum SmVideoUnits
```

Specifies units in which VBW processing occurs.

Enumerator

smVideoLog	dBm
smVideoVoltage	Linear voltage
smVideoPower	Linear power
smVideoSample	No VBW processing

13.1.3.10 SmWindowType

```
enum SmWindowType
```

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

Enumerator

smWindowFlatTop	SRS flattop
smWindowNutall	Nutall
smWindowBlackman	Blackman
smWindowHamming	Hamming
smWindowGaussian6dB	Gaussian 6dB BW window for EMC measurements and CISPR compatibility
smWindowRect	Rectangular (no) window

13.1.3.11 SmTriggerType

`enum SmTriggerType`

Trigger type for specific I/Q capture modes.

Enumerator

smTriggerTypeImm	Immediate/no trigger
smTriggerTypeVideo	Video/level trigger
smTriggerTypeExt	External trigger
smTriggerTypeFMT	Frequency mask trigger

13.1.3.12 SmTriggerEdge

`enum SmTriggerEdge`

Trigger edge for video and external triggers.

Enumerator

smTriggerEdgeRising	Rising edge
smTriggerEdgeFalling	Falling edge

13.1.3.13 SmBool

`enum SmBool`

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

Enumerator

smFalse	False
smTrue	True

13.1.3.14 SmGPIOState

`enum SmGPIOState`

Used to set the 8 configurable GPIO pins to inputs/outputs.

Enumerator

smGPIOStateOutput	Output
smGPIOStateInput	Input

13.1.3.15 SmReference

`enum SmReference`

Used to indicate the source of the timebase reference for the device.

Enumerator

smReferenceUseInternal	Use the internal 10MHz timebase.
smReferenceUseExternal	Use an external 10MHz timebase on the `10 MHz In` port.

13.1.3.16 SmDeviceType

`enum SmDeviceType`

Device type

Enumerator

smDeviceTypeSM200A	SM200A
smDeviceTypeSM200B	SM200B
smDeviceTypeSM200C	SM200C
smDeviceTypeSM435B	SM435B
smDeviceTypeSM435C	SM435C

13.1.3.17 SmAudioType

```
enum SmAudioType
```

Audio demodulation type.

Enumerator

smAudioTypeAM	AM
smAudioTypeFM	FM
smAudioTypeUSB	Upper side band
smAudioTypeLSB	Lower side band
smAudioTypeCW	CW

13.1.3.18 SmGPSState

```
enum SmGPSState
```

Internal GPS state

Enumerator

smGPSStateNotPresent	GPS is not locked
smGPSStateLocked	GPS is locked, NMEA data is valid, but the timebase is not being disciplined by the GPS
smGPSStateDisciplined	GPS is locked, NMEA data is valid, timebase is being disciplined by the GPS

13.1.4 Function Documentation

13.1.4.1 smGetDeviceList()

```
SM_API SmStatus smGetDeviceList (
            int * serials,
            int * deviceCount )
```

This function is for USB SM devices only. This function is used to retrieve the serial numbers of all unopened USB SM devices connected to the PC. The maximum number of serial numbers that can be returned is 9. The serial numbers returned can then be used to open specific devices with the smOpenDeviceBySerial function. When the function returns successfully, the serials array will contain deviceCount number of unique SM serial numbers. Only deviceCount values will be modified. This function will not return the serial numbers of any connected networked devices.

Parameters

out	*serials*	Pointer to an array of integers. The array must be larger than the number of USB SM devices connected to the PC.
out	*deviceCount*	If the function returns successfully deviceCount will be set to the number devices found on the system.

Returns

13.1.4.2 smGetDeviceList2()

```
SM_API SmStatus smGetDeviceList2 (
            int * serials,
            SmDeviceType * deviceTypes,
            int * deviceCount )
```

This function is for USB SM devices only. This function is used to retrieve the serial numbers and device types of all unopened USB SM devices connected to the PC. The maximum number of serial numbers that can be returned is 9. The serial numbers returned can then be used to open specific devices with the smOpenDeviceBySerial function. When the function returns successfully, the serials and deviceCount array will contain deviceCount number of unique SM serial numbers and deviceTypes. Only deviceCount values will be modified. This function will not return the serial numbers of any connected networked devices.

Parameters

out	*serials*	Pointer to an array of integers. The array must be larger than the number of USB SM devices connected to the PC.
out	*deviceTypes*	Pointer to an array of SmDeviceType enums. The array must be larger than the number of USB SM devices connected to the PC.
out	*deviceCount*	Pointer to integer. If the function returns successfully deviceCount will be set to the number devices found on the system.

Returns

13.1.4.3 smOpenDevice()

```
SM_API SmStatus smOpenDevice (
            int * device )
```

This function is for USB SM devices only. Claim the first unopened USB SM 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 smGet↵ DeviceList and using the first device found to call smOpenDeviceBySerial.

Parameters

out	*device*	Returns handle that can be used to interface the device.

Returns

13.1.4.4 smOpenDeviceBySerial()

```
SM_API SmStatus smOpenDeviceBySerial (
            int * device,
            int serialNumber )
```

This function is similar to smOpenDevice except it allows you to specify the device you wish to open. This function is often used in conjunction with smGetDeviceList when managing several SM devices on on PC.

Parameters

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

Returns

13.1.4.5 smOpenNetworkedDevice()

```
SM_API SmStatus smOpenNetworkedDevice (
            int * device,
            const char * hostAddr,
            const char * deviceAddr,
            uint16_t port )
```

This function is for networked (10GbE) devices only. Attempts to connect to a networked device. 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. The device takes approximately 12 seconds to boot up after applying power. Until the device is booted, this function will return device not found. The SM API does not set the SO_REUSEADDR socket option. For customers connecting multiple networked devices, we recommend specifying the hostAddr explicitly instead of using “0.0.0.0”. Especially for configurations that involve multiple subnets. If not done, devices beyond the first will likely not be found and this function will return an error.

Parameters

out	*device*	Returns handle that can be used to interface the device.
in	*hostAddr*	Host interface IP on which the networked device is connected, provided as a string. Can be “0.0.0.0”. An example parameter is “192.168.2.2”.
in	*deviceAddr*	Target device IP provided as a string. If more than one device with this IP is connected to the host interface, the behavior is undefined.
in	*port*	Target device port.

Returns

13.1.4.6 smCloseDevice()

```
SM_API SmStatus smCloseDevice (
            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	*device*	Device handle.

Returns

13.1.4.7 smPreset()

```
SM_API SmStatus smPreset (
            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 smCloseDevice function will have been called for the device handle, and the device will need to be re-opened again. For USB devices, the full 20 seconds open cycle will occur when re-opening the device. For networked devices, this function blocks for an additional 15 seconds to ensure the device has fully power cycled and can be opened. This function can be used to recover from an undesirable device state.

Parameters

in	*device*	Device handle.

Returns

13.1.4.8 smPresetSerial()

```
SM_API SmStatus smPresetSerial (
            int serialNumber )
```

Performs a full device preset for a device that has not been opened with the smOpenDevice function. This function will open and then preset the device. This function does not check if the device is already opened. Calling this function on a device that is already open through the API is undefined behavior.

Parameters

in	*serialNumber*	Serial number of the device to preset.

Returns

13.1.4.9 smNetworkedSpeedTest()

```
SM_API SmStatus smNetworkedSpeedTest (
            int device,
            double durationSeconds,
            double * bytesPerSecond )
```

This function is for networked (10GbE) devices only. Measure the network throughput between the device and the PC. Useful for troubleshooting network throughput issues.

Parameters

in	*device*	Device handle.
in	*durationSeconds*	The duration of the test specified in seconds. Can be values between 16ms and 100s. Recommended value of 1 second minimum to produce good averaging and reduce startup overhead.
out	*bytesPerSecond*	Pointer to double which when finished, will contain the measured bytes per second throughput between the device and PC.

Returns

13.1.4.10 smGetDeviceInfo()

```
SM_API SmStatus smGetDeviceInfo (
            int device,
            SmDeviceType * deviceType,
            int * serialNumber )
```

This function returns basic information about a specific open device. Also see smGetDeviceDiagnostics.

Parameters

in	*device*	Device handle.
out	*deviceType*	Pointer to SmDeviceType to contain the device model number. Can be NULL.
out	*serialNumber*	Returns device serial number. Can be NULL.

Returns

13.1.4.11 smGetFirmwareVersion()

```
SM_API SmStatus smGetFirmwareVersion (
            int device,
            int * major,
            int * minor,
            int * revision )
```

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

Parameters

in	*device*	Device handle.
out	*major*	Pointer to int. Can be NULL.
out	*minor*	Pointer to int. Can be NULL.
out	*revision*	Pointer to int. Can be NULL.

Returns

13.1.4.12 smHasIFOutput()

```
SM_API SmStatus smHasIFOutput (
            int device,
            SmBool * present )
```

Returns whethe the SM435 device has the IF output option. See SM435 IF Output Option for more information.

Parameters

in	*device*	Device handle.
out	*present*	Set to smTrue if the device has the IF output option.

Returns

13.1.4.13 smGetDeviceDiagnostics()

```
SM_API SmStatus smGetDeviceDiagnostics (
            int device,
            float * voltage,
            float * current,
            float * temperature )
```

Returns operational information about a device.

Parameters

in	*device*	Device handle.
out	*voltage*	Pointer to float, to contain device voltage. Can be `NULL`.
out	*current*	Pointer to float, to contain device current. Can be `NULL`.
out	*temperature*	Pointer to float, to contain device temperature. Can be `NULL`.

Returns

13.1.4.14 smGetFullDeviceDiagnostics()

```
SM_API SmStatus smGetFullDeviceDiagnostics (
            int device,
            SmDeviceDiagnostics * diagnostics )
```

Returns operational information about a device. If any temperature sensors are unpopulated, the temperature returned for that sensor will be 240C. Should always be able to retrieve the FPGA core and RF board temperatures.

Parameters

in	*device*	Device handle.
out	*diagnostics*	Pointer to struct.

Returns

13.1.4.15 smGetSFPDiagnostics()

```
SM_API SmStatus smGetSFPDiagnostics (
            int device,
            float * temp,
            float * voltage,
            float * txPower,
            float * rxPower )
```

For networked (10GbE) devices only. Returns a number of diagnostic information for the SFP+ transceiver attached to the device. If either the device is not a networked device or the SFP+ does not communicate diagnostic information, the values returned will be zero.

Parameters

in	*device*	Device handle.
out	*temp*	Reported SFP+ temperature in C. Can be `NULL`.
out	*voltage*	Reported SFP+ voltage in V. Can be `NULL`.
out	*txPower*	Reported transmit power in mW. Can be `NULL`.
out	*rxPower*	Reported receive power in mW. Can be `NULL`.

Returns

13.1.4.16 smSetPowerState()

```
SM_API SmStatus smSetPowerState (
            int device,
            SmPowerState 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	*device*	Device handle.
in	*powerState*	New power state.

Returns

13.1.4.17 smGetPowerState()

```
SM_API SmStatus smGetPowerState (
            int device,
            SmPowerState * powerState )
```

Retrieves the current power state. See Power States for more information.

Parameters

in	*device*	Device handle.
out	*powerState*	Pointer to SmPowerState.

Returns

13.1.4.18 smSetAttenuator()

```
SM_API SmStatus smSetAttenuator (
            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 SM_AUTO_↩ATTEN macro for -1.

Parameters

in	*device*	Device handle.
in	*atten*	Attenuation value between [-1,6].

Returns

13.1.4.19 smGetAttenuator()

```
SM_API SmStatus smGetAttenuator (
            int device,
            int * atten )
```

Get the device attenuation. See Reference Level and Sensitivity for more information.

Parameters

in	*device*	Device handle.
out	*atten*	Returns current attenuation value.

Returns

13.1.4.20 smSetRefLevel()

```
SM_API SmStatus smSetRefLevel (
            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	*device*	Device handle.
in	*refLevel*	Set the reference level of the receiver in dBm.

Returns

13.1.4.21 smGetRefLevel()

```
SM_API SmStatus smGetRefLevel (
            int device,
            double * refLevel )
```

Retreive the current device reference level.

Parameters

in	*device*	Device handle.
out	*refLevel*	Reference level returned in dBm.

Returns

13.1.4.22 smSetPreselector()

```
SM_API SmStatus smSetPreselector (
            int device,
            SmBool enabled )
```

Enable/disable the RF preselector. This setting controls the preselector for all measurement modes.

Parameters

in	*device*	Device handle.
in	*enabled*	Set to smTrue to enable the preselector.

Returns

13.1.4.23 smGetPreselector()

```
SM_API SmStatus smGetPreselector (
            int device,
            SmBool * enabled )
```

Retrieve the current preselector setting.

Parameters

in	*device*	Device handle.
out	*enabled*	Returns smTrue if the preselector is enabled.

Returns

13.1.4.24 smSetGPIOState()

```
SM_API SmStatus smSetGPIOState (
            int device,
            SmGPIOState lowerState,
            SmGPIOState upperState )
```

Configure whether the GPIO pins are read/write. This affects the pins immediately. See the GPIO section for more information.

Parameters

in	*device*	Device handle.
in	*lowerState*	Sets the direction of the lower 4 GPIO pins.
in	*upperState*	Sets the direction of the upper 4 GPIO pins.

Returns

13.1.4.25 smGetGPIOState()

```
SM_API SmStatus smGetGPIOState (
            int device,
            SmGPIOState * lowerState,
            SmGPIOState * upperState )
```

Get the direction (read/write) of the GPIO pins. See the GPIO section for more information.

Parameters

in	*device*	Device handle.
out	*lowerState*	Returns the direction of the lower 4 GPIO pins.
out	*upperState*	Returns the direction of the upper 4 GPIO pins.

Returns

13.1.4.26 smWriteGPIOImm()

```
SM_API SmStatus smWriteGPIOImm (
            int device,
            uint8_t data )
```

Set the GPIO output levels. Will only affect GPIO pins configured as outputs. The bits in the data parameter that correspond with GPIO pins that have been set as inputs are ignored.

Parameters

in	*device*	Device handle.
in	*data*	Data used to set the GPIO. Each bit corresponds to the 8 GPIO pins.

Returns

13.1.4.27 smReadGPIOImm()

```
SM_API SmStatus smReadGPIOImm (
            int device,
            uint8_t * data )
```

Retrieve the values of the GPIO pins. GPIO pins that are configured as outputs will return the set output logic level. If the device is currently idle, the GPIO logic levels are sampled. If the device is configured in a measurement mode, the values returned are those reported from the last measurement taken. For example, if the device is configured for sweeping, each sweep performed will update the GPIO. To retrieve the most current values, either perform another sweep and re-request the GPIO state or put the device in an idle mode and query the GPIO.

Parameters

in	*device*	Device handle.
out	*data*	Pointer to byte. Each bit corresponds to the 8 GPIO pins.

Returns

13.1.4.28 smWriteSPI()

```
SM_API SmStatus smWriteSPI (
            int device,
            uint32_t data,
            int byteCount )
```

Output up to 4 bytes on the SPI data pins. See the SPI section for more information.

Parameters

in	*device*	Device handle.
in	*data*	Up to 4 bytes of data to transfer.
in	*byteCount*	Number of bytes to transfer.

Returns

13.1.4.29 smSetGPIOSweepDisabled()

```
SM_API SmStatus smSetGPIOSweepDisabled (
            int device )
```

Disables and clears the current GPIO sweep setup. The effect of this function will be seen the next time the device is configured. See the GPIO section for more information.

Parameters

in	*device*	Device handle.

Returns

13.1.4.30 smSetGPIOSweep()

```
SM_API SmStatus smSetGPIOSweep (
            int device,
            SmGPIOStep * steps,
            int stepCount )
```

This function is used to set the frequency cross over points for the GPIO sweep functionality and the associated GPIO output logic levels for each frequency. See GPIO for more information.

Parameters

in	*device*	Device handle.
in	*steps*	Array of SmGPIOStep structs. The array must be `stepCount` in length.
in	*stepCount*	The number of steps in the steps array.

Returns

13.1.4.31 smSetGPIOSwitchingDisabled()

```
SM_API SmStatus smSetGPIOSwitchingDisabled (
            int device )
```

Disables any GPIO switching setup. The effect of this function will be seen the next time the device is configured for I/Q streaming. If the device is actively in a GPIO switching loop (and I/Q streaming) the GPIO switching is not disabled until the device is reconfigured. This function can be called at any time. See GPIO for more information.

Parameters

in	*device*	Device handle.

Returns

13.1.4.32 smSetGPIOSwitching()

```
SM_API SmStatus smSetGPIOSwitching (
            int device,
            uint8_t * gpio,
            uint32_t * counts,
            int gpioSteps )
```

Configures the GPIO switching functionality. See GPIO for more information.

Parameters

in	*device*	Device handle.
in	*gpio*	Array of GPIO output settings.
in	*counts*	Array of dwell times (in 20ns counts). The maximum count value for a given state/step is (2^22 - 1).
in	*gpioSteps*	Number of GPIO steps.

Returns

13.1.4.33 smSetExternalReference()

```
SM_API SmStatus smSetExternalReference (
            int device,
            SmBool enabled )
```

Enable or disable the 10MHz reference out port. If enabled, the current reference being used by the SM (as specified by smSetReference) will be output on the 10MHz out port.

Parameters

in	*device*	Device handle.
in	*enabled*	Set to smTrue to enable the 10MHz reference out port.

Returns

13.1.4.34 smGetExternalReference()

```
SM_API SmStatus smGetExternalReference (
            int device,
            SmBool * enabled )
```

Return whether the 10MHz reference out port is enabled.

Parameters

in	*device*	Device handle.
out	*enabled*	Returns smTrue if the ref out port is enabled.

Returns

13.1.4.35 smSetReference()

```
SM_API SmStatus smSetReference (
            int device,
            SmReference reference )
```

Tell the receiver to use either the internal time base reference or use a 10MHz reference present on the 10MHz in port. The device must be in the idle state (call smAbort) for this function to take effect. If the function returns successfully, verify the new state with the smGetReference function.

Parameters

in	*device*	Device handle.
in	*reference*	New reference state.

Returns

13.1.4.36 smGetReference()

```
SM_API SmStatus smGetReference (
            int device,
            SmReference * reference )
```

Get the current reference state.

Parameters

in	*device*	Device handle.
out	*reference*	Returns current reference configuration.

Returns

13.1.4.37 smSetGPSTimebaseUpdate()

```
SM_API SmStatus smSetGPSTimebaseUpdate (
            int device,
            SmBool 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	*device*	Device handle.
in	*enabled*	Send smTrue to enable.

Returns

13.1.4.38 smGetGPSTimebaseUpdate()

```
SM_API SmStatus smGetGPSTimebaseUpdate (
            int device,
            SmBool * enabled )
```

Get auto GPS timebase update status. See Automatic GPS Timebase Discipline for more information.

Parameters

in	*device*	Device handle.
out	*enabled*	Returns smTrue if auto GPS timebase update is enabled.

Returns

13.1.4.39 smGetGPSHoldoverInfo()

```
SM_API SmStatus smGetGPSHoldoverInfo (
            int device,
            SmBool * usingGPSHoldover,
            uint64_t * lastHoldoverTime )
```

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

Parameters

in	*device*	Device handle.
out	*usingGPSHoldover*	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 smGetGPSState. This parameter can be NULL.
out	*lastHoldoverTime*	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

13.1.4.40 smGetGPSState()

```
SM_API SmStatus smGetGPSState (
            int device,
            SmGPSState * GPSState )
```

Determine the lock and discipline status of the GPS. See the Acquiring GPS Lock section for more information.

Parameters

in	*device*	Device handle.
out	*GPSState*	Pointer to SmGPSState.

Returns

13.1.4.41 smSetSweepSpeed()

```
SM_API SmStatus smSetSweepSpeed (
            int device,
            SmSweepSpeed sweepSpeed )
```

Set sweep speed.

Parameters

in	*device*	Device handle.
in	*sweepSpeed*	New sweep speed.

Returns

13.1.4.42 smSetSweepCenterSpan()

```
SM_API SmStatus smSetSweepCenterSpan (
            int device,
            double centerFreqHz,
            double spanHz )
```

Set sweep center/span.

Parameters

in	*device*	Device handle.
in	*centerFreqHz*	New center frequency in Hz.
in	*spanHz*	New span in Hz.

Returns

13.1.4.43 smSetSweepStartStop()

```
SM_API SmStatus smSetSweepStartStop (
            int device,
            double startFreqHz,
            double stopFreqHz )
```

Set sweep start/stop frequency.

Parameters

in	*device*	Device handle.
in	*startFreqHz*	Start frequency in Hz.
in	*stopFreqHz*	Stop frequency in Hz.

Returns

13.1.4.44 smSetSweepCoupling()

```
SM_API SmStatus smSetSweepCoupling (
            int device,
            double rbw,
            double vbw,
            double sweepTime )
```

Set sweep RBW/VBW parameters.

Parameters

in	*device*	Device handle.
in	*rbw*	Resolution bandwidth in Hz.
in	*vbw*	Video bandwidth in Hz. Cannot be greater than RBW.
in	*sweepTime*	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

13.1.4.45 smSetSweepDetector()

```
SM_API SmStatus smSetSweepDetector (
            int device,
            SmDetector detector,
            SmVideoUnits videoUnits )
```

Set sweep detector.

Parameters

in	*device*	Device handle.
in	*detector*	New sweep detector.
in	*videoUnits*	New video processing units.

Returns

13.1.4.46 smSetSweepScale()

```
SM_API SmStatus smSetSweepScale (
            int device,
            SmScale scale )
```

Set the sweep mode output unit type.

Parameters

in	*device*	Device handle.
in	*scale*	New sweep mode units.

Returns

13.1.4.47 smSetSweepWindow()

```
SM_API SmStatus smSetSweepWindow (
            int device,
            SmWindowType window )
```

Set sweep mode window function.

Parameters

in	*device*	Device handle.
in	*window*	New window function.

Returns

13.1.4.48 smSetSweepSpurReject()

```
SM_API SmStatus smSetSweepSpurReject (
            int device,
            SmBool spurRejectEnabled )
```

Set sweep mode spur rejection enable/disable.

Parameters

in	*device*	Device handle.
in	*spurRejectEnabled*	Enable/disable.

Returns

13.1.4.49 smSetRealTimeCenterSpan()

```
SM_API SmStatus smSetRealTimeCenterSpan (
            int device,
            double centerFreqHz,
            double spanHz )
```

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

Parameters

in	*device*	Device handle.
in	*centerFreqHz*	Center frequency in Hz.
in	*spanHz*	Span in Hz.

Returns

13.1.4.50 smSetRealTimeRBW()

```
SM_API SmStatus smSetRealTimeRBW (
            int device,
            double rbw )
```

Set the resolution bandwidth for real-time spectrum analysis.

Parameters

in	*device*	Device handle.
in	*rbw*	Resolution bandwidth in Hz.

Returns

13.1.4.51 smSetRealTimeDetector()

```
SM_API SmStatus smSetRealTimeDetector (
            int device,
            SmDetector detector )
```

Set the detector for real-time spectrum analysis.

Parameters

in	*device*	Device handle.
in	*detector*	New detector.

Returns

13.1.4.52 smSetRealTimeScale()

```
SM_API SmStatus smSetRealTimeScale (
            int device,
            SmScale scale,
            double frameRef,
            double frameScale )
```

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

Parameters

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

Returns

13.1.4.53 smSetRealTimeWindow()

```
SM_API SmStatus smSetRealTimeWindow (
            int device,
            SmWindowType window )
```

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

Parameters

in	*device*	Device handle.
in	*window*	New window function.

Returns

13.1.4.54 smSetIQBaseSampleRate()

```
SM_API SmStatus smSetIQBaseSampleRate (
            int device,
            SmIQStreamSampleRate sampleRate )
```

Set the base sample rate for I/Q streaming.

Parameters

in	*device*	Device handle.
in	*sampleRate*	Base sample rate. Any decimation selected occurs on this sample rate. See I/Q Sample Rates for more information.

Returns

13.1.4.55 smSetIQDataType()

```
SM_API SmStatus smSetIQDataType (
            int device,
            SmDataType dataType )
```

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

Parameters

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

Returns

13.1.4.56 smSetIQCenterFreq()

```
SM_API SmStatus smSetIQCenterFreq (
            int device,
            double centerFreqHz )
```

Set the center frequency for I/Q streaming.

Parameters

in	*device*	Device handle.
in	*centerFreqHz*	Center frequency in Hz.

Returns

13.1.4.57 smGetIQCenterFreq()

```
SM_API SmStatus smGetIQCenterFreq (
            int device,
            double * centerFreqHz )
```

Get the I/Q streaming center frequency.

Parameters

in	*device*	Device handle.
in	*centerFreqHz*	Pointer to double.

Returns

13.1.4.58 smSetIQSampleRate()

```
SM_API SmStatus smSetIQSampleRate (
            int device,
            int decimation )
```

Set sample rate for I/Q streaming.

Parameters

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

Returns

13.1.4.59 smSetIQBandwidth()

```
SM_API SmStatus smSetIQBandwidth (
            int device,
            SmBool enableSoftwareFilter,
            double bandwidth )
```

Specify the software filter bandwidth in I/Q streaming. See I/Q Sample Rates for more information.

Parameters

in	*device*	Device handle.
in	*enableSoftwareFilter*	Set to true to enable software filtering (USB devices only). This values is ignored for 10GbE devices.
in	*bandwidth*	The bandwidth in Hz.

Returns

13.1.4.60 smSetIQExtTriggerEdge()

```
SM_API SmStatus smSetIQExtTriggerEdge (
            int device,
            SmTriggerEdge edge )
```

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

Parameters

in	*device*	Device handle.
in	*edge*	Set the external trigger edge.

Returns

13.1.4.61 smSetIQTriggerSentinel()

```
SM_API SmStatus smSetIQTriggerSentinel (
            double sentinelValue )
```

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

Parameters

in	*sentinelValue*	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

13.1.4.62 smSetIQQueueSize()

```
SM_API SmStatus smSetIQQueueSize (
            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, it's 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	*device*	Device handle
in	*ms*	Queue size in ms. Will be clamped to multiples of 2.62ms between $2 * 2.62$ms and $16 * 2.62$ms.

Returns

13.1.4.63 smSetIQSweepListDataType()

```
SM_API SmStatus smSetIQSweepListDataType (
            int device,
            SmDataType dataType )
```

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

Parameters

in	*device*	Device handle.
in	*dataType*	See I/Q Data Types for more information.

Returns

13.1.4.64 smSetIQSweepListCorrected()

```
SM_API SmStatus smSetIQSweepListCorrected (
            int device,
            SmBool corrected )
```

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

Parameters

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

Returns

13.1.4.65 smSetIQSweepListSteps()

```
SM_API SmStatus smSetIQSweepListSteps (
            int device,
            int steps )
```

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

Parameters

in	*device*	Device handle.
in	*steps*	Number of frequency steps in I/Q sweep.

Returns

13.1.4.66 smGetIQSweepListSteps()

```
SM_API SmStatus smGetIQSweepListSteps (
            int device,
            int * steps )
```

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

Parameters

in	*device*	Device handle.
out	*steps*	Pointer to int.

Returns

13.1.4.67 smSetIQSweepListFreq()

```
SM_API SmStatus smSetIQSweepListFreq (
            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	*device*	Device handle.
in	*step*	Step at which to configure the center frequency. Should be between [0, steps-1] where steps is set in the smSetIQSweepListSteps function.
in	*freq*	Center frequency in Hz.

Returns

13.1.4.68 smSetIQSweepListRef()

```
SM_API SmStatus smSetIQSweepListRef (
            int device,
            int step,
            double level )
```

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

Parameters

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

Returns

13.1.4.69 smSetIQSweepListAtten()

```
SM_API SmStatus smSetIQSweepListAtten (
            int device,
            int step,
            int atten )
```

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

Parameters

in	*device*	Device handle.
in	*step*	Step at which to configure the center frequency. Should be between [0, steps-1] where steps is set in the smSetIQSweepListSteps function.
in	*atten*	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

13.1.4.70 smSetIQSweepListSampleCount()

```
SM_API SmStatus smSetIQSweepListSampleCount (
            int device,
            int step,
            uint32_t samples )
```

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

Parameters

in	*device*	Device handle.
in	*step*	Step at which to configure the center frequency. Should be between [0, steps-1] where steps is set in the smSetIQSweepListSteps function.
in	*samples*	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

13.1.4.71 smSetSegIQDataType()

```
SM_API SmStatus smSetSegIQDataType (
            int device,
            SmDataType dataType )
```

Set the data type for the data returned for segmented I/Q captures.

Parameters

in	*device*	Device handle.
in	*dataType*	New data type.

Returns

13.1.4.72 smSetSegIQCenterFreq()

```
SM_API SmStatus smSetSegIQCenterFreq (
            int device,
            double centerFreqHz )
```

Set the center frequency for segmeneted I/Q captures.

Parameters

in	*device*	Device handle.
in	*centerFreqHz*	Center frequency in Hz.

Returns

13.1.4.73 smSetSegIQVideoTrigger()

```
SM_API SmStatus smSetSegIQVideoTrigger (
            int device,
```

```
double triggerLevel,
SmTriggerEdge triggerEdge )
```

Configure the video trigger available in segmented I/Q captures. Only 1 video trigger configuration can be set.

Parameters

in	*device*	Device handle.
in	*triggerLevel*	Trigger level in dBm.
in	*triggerEdge*	Video trigger edge type.

Returns

13.1.4.74 smSetSegIQExtTrigger()

```
SM_API SmStatus smSetSegIQExtTrigger (
            int device,
            SmTriggerEdge extTriggerEdge )
```

Configure the external trigger available in segmented I/Q captures. Only 1 external trigger configuration can be set.

Parameters

in	*device*	Device handle.
in	*extTriggerEdge*	External trigger edge type.

Returns

13.1.4.75 smSetSegIQFMTParams()

```
SM_API SmStatus smSetSegIQFMTParams (
            int device,
            int fftSize,
            const double * frequencies,
            const double * ampls,
            int count )
```

Configure the frequency mask trigger available in segmented I/Q captures. Only 1 frequency mask trigger configuration can be set.

Parameters

in	*device*	Device handle.
in	*fftSize*	Size of the FFT used for FMT triggering. This value must be a power of two between 512 and 16384. The frequency/amplitude mask provided by the user is linearly interpolated and tested at each of the FFT result bins. Smaller FFT sizes provide more time resolution at the expense of frequency resolution, while larger FFT sizes improve frequency resolution at the expense of time resolution. The complex FFT is performed at the 250MS/s I/Q samples with a 50% overlap.

Parameters

in	*frequencies*	Array of count frequencies, specified as Hz, specifying the frequency points of the FMT mask.
in	*ampls*	Array of count amplitudes, specified as dBm, specifying the amplitude threshold limits of the FMT mask.
in	*count*	Number of FMT points in the frequencies and ampls arrays.

Returns

13.1.4.76 smSetSegIQSegmentCount()

```
SM_API SmStatus smSetSegIQSegmentCount (
            int device,
            int segmentCount )
```

Set the number of segments in the segmented I/Q captures.

Parameters

in	*device*	Device handle.
in	*segmentCount*	Number of segments. Must be set before configuring each segment.

Returns

13.1.4.77 smSetSegIQSegment()

```
SM_API SmStatus smSetSegIQSegment (
            int device,
            int segment,
            SmTriggerType triggerType,
            int preTrigger,
            int captureSize,
            double timeoutSeconds )
```

Configure a segment for segmented I/Q captures.

Parameters

in	*device*	Device handle.
in	*segment*	Segment to configure. Must be between [0,segmentCount-1] where segmentCount is set in smSetSegIQSegmentCount.

Parameters

in	*triggerType*	Specify the trigger used for this segment.
in	*preTrigger*	The number of samples to capture before the trigger event. This is in addition to the capture size. For immediate trigger, pretrigger is added to capture size and then set to zero.
in	*captureSize*	The number of sample to capture after the trigger event. For immediate triggers, pretrigger is added to this value and pretrigger is set to zero.
in	*timeoutSeconds*	The amount of time to wait for the trigger before returning. If a timeout occurs, a capture still occurs at the moment of the timeout and the API will report a timeout condition.

Returns

13.1.4.78 smSetAudioCenterFreq()

```
SM_API SmStatus smSetAudioCenterFreq (
            int device,
            double centerFreqHz )
```

Set the center frequency for audio demodulation.

Parameters

in	*device*	Device handle.
in	*centerFreqHz*	Center frequency in Hz.

Returns

13.1.4.79 smSetAudioType()

```
SM_API SmStatus smSetAudioType (
            int device,
            SmAudioType audioType )
```

Set the audio demodulator for audio demodulation.

Parameters

in	*device*	Device handle.
in	*audioType*	Demodulator.

Returns

13.1.4.80 smSetAudioFilters()

```
SM_API SmStatus smSetAudioFilters (
            int device,
            double ifBandwidth,
            double audioLpf,
            double audioHpf )
```

Set the audio demodulation filters for audio demodulation.

Parameters

in	*device*	Device handle.
in	*ifBandwidth*	IF bandwidth (RBW) in Hz.
in	*audioLpf*	Audio low pass frequency in Hz.
in	*audioHpf*	Audio high pass frequency in Hz.

Returns

13.1.4.81 smSetAudioFMDeemphasis()

```
SM_API SmStatus smSetAudioFMDeemphasis (
            int device,
            double deemphasis )
```

Set the FM deemphasis for audio demodulation.

Parameters

in	*device*	Device handle.
in	*deemphasis*	Deemphasis in us.

Returns

13.1.4.82 smConfigure()

```
SM_API SmStatus smConfigure (
            int device,
            SmMode 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 smAbort 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	*device*	Device handle.
in	*mode*	New measurement mode.

Returns

13.1.4.83 smGetCurrentMode()

```
SM_API SmStatus smGetCurrentMode (
            int device,
            SmMode * mode )
```

Retrieve the current device measurement mode.

Parameters

in	*device*	Device handle.
in	*mode*	Pointer to SmMode.

Returns

13.1.4.84 smAbort()

```
SM_API SmStatus smAbort (
            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	*device*	Device handle.

Returns

13.1.4.85 smGetSweepParameters()

```
SM_API SmStatus smGetSweepParameters (
            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	*device*	Device handle.
out	*actualRBW*	Returns the RBW being used in Hz. Can be NULL.
out	*actualVBW*	Returns the VBW being used in Hz. Can be NULL.
out	*actualStartFreq*	Returns the frequency of the first bin in Hz. Can be NULL.
out	*binSize*	Returns the frequency spacing between each frequency bin in the sweep in Hz.
out	*sweepSize*	Returns the length of the sweep (number of frequency bins). Can be NULL.

Returns

13.1.4.86 smGetRealTimeParameters()

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

Returns

13.1.4.87 smGetIQParameters()

```
SM_API SmStatus smGetIQParameters (
            int device,
            double * sampleRate,
            double * bandwidth )
```

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

Parameters

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

Returns

13.1.4.88 smGetIQCorrection()

```
SM_API SmStatus smGetIQCorrection (
            int device,
            float * scale )
```

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

Parameters

in	*device*	Device handle.
out	*scale*	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

13.1.4.89 smIQSweepListGetCorrections()

```
SM_API SmStatus smIQSweepListGetCorrections (
            int device,
            float * corrections )
```

Retrieve the correctsions 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	*device*	Device handle.
out	*corrections*	Pointer to an array. Array should length >= number of steps configured for the I/Q sweep list measurement. A correction value will be returned for each step configured.

Returns

13.1.4.90 smSegIQGetMaxCaptures()

```
SM_API SmStatus smSegIQGetMaxCaptures (
            int device,
            int * maxCaptures )
```

This function is called after the device is successfully configured for segmented I/Q acquisition. Returns the maximum number of queued captures that can be active. This is calculated with the formula (250 / # of segments in each capture). See I/Q Segmented Captures for more information.

Parameters

in	*device*	Device handle.
out	*maxCaptures*	The maximum number of queued segmented acquisitions that can be active at any time.

Returns

13.1.4.91 smGetSweep()

```
SM_API SmStatus smGetSweep (
            int device,
            float * sweepMin,
            float * sweepMax,
            int64_t * nsSinceEpoch )
```

Perform a single sweep. Block until the sweep completes. Internally, this function is implemented as calling smStartSweep followed by smFinishSweep with a sweep position of zero (0). This means that if you want to mix the blocking and queue sweep acquisitions, avoid using index zero for queued sweeps.

Parameters

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

Returns

13.1.4.92 smSetSweepGPIO()

```
SM_API SmStatus smSetSweepGPIO (
            int device,
            int pos,
            uint8_t data )
```

Set the GPIO setting to use for a queued sweep. The next time this sweep is started, the GPIO will change to this value just prior to the sweep starting.

Parameters

in	*device*	Device handle.
in	*pos*	Sweep queue position.
in	*data*	Data used to set the GPIO pins. Each bit represents a single GPIO pin.

Returns

13.1.4.93 smStartSweep()

```
SM_API SmStatus smStartSweep (
            int device,
            int pos )
```

Start a sweep at the queue position. If successful, this function returns immediately.

Parameters

in	*device*	Device handle.
in	*pos*	Sweep queue position.

Returns

13.1.4.94 smFinishSweep()

```
SM_API SmStatus smFinishSweep (
            int device,
            int pos,
            float * sweepMin,
            float * sweepMax,
            int64_t * nsSinceEpoch )
```

Finish a previously started queued sweep. Blocks until the sweep completes.

Parameters

in	*device*	Device handle.
in	*pos*	Sweep queue position.
out	*sweepMin*	Can be NULL.
out	*sweepMax*	Can be NULL.
out	*nsSinceEpoch*	Nanoseconds since epoch. Timestamp representing the end of the sweep. Can be NULL.

Returns

13.1.4.95 smGetRealTimeFrame()

```
SM_API SmStatus smGetRealTimeFrame (
            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	*device*	Device handle.
out	*colorFrame*	Pointer to memory for the frame. Must be (frameWidth * frameHeight) floats in size. Can be NULL.
out	*alphaFrame*	Pointer to memory for the frame. Must be (frameWidth * frameHeight) floats in size. Can be NULL.
out	*sweepMin*	Can be NULL.
out	*sweepMax*	Can be NULL.
out	*frameCount*	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	*nsSinceEpoch*	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

13.1.4.96 smGetIQ()

```
SM_API SmStatus smGetIQ (
            int device,
            void * iqBuf,
            int iqBufSize,
            double * triggers,
            int triggerBufSize,
            int64_t * nsSinceEpoch,
            SmBool 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	*device*	Device handle.

Parameters

out	*iqBuf*	Pointer to user allocated buffer of complex values. The buffer size must be at least (iqBufSize ∗ 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	*iqBufSize*	Specifies the number of I/Q samples to be retrieves. Must be greater than zero.
out	*triggers*	Pointer to user allocated array of doubles. The buffer must be at least triggerBufSize 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	*triggerBufSize*	Specifies the size of the triggersr array. If the triggers array is NULL, this value should be zero.
out	*nsSinceEpoch*	Nanoseconds since epoch. The time of the first I/Q sample returned. Can be NULL. See GPS and Timestamps for more information.
in	*purge*	When set to smTrue, any buffered I/Q data in the API is purged before returned beginning the I/Q block acquisition.
out	*sampleLoss*	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 SM_FALSE. Can be NULL.
out	*samplesRemaining*	Set by the API, returns the number of samples remaining in the I/Q circular buffer. Can be NULL.

Returns

13.1.4.97 smIQSweepListGetSweep()

```
SM_API SmStatus smIQSweepListGetSweep (
            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	*device*	Device handle.
out	*dst*	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	*timestamps*	Pointer to memory allocated for timestamps. The user must allocate this memory before calling these functions. Must be an array of steps int64_t's, where steps are 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

13.1.4.98 smIQSweepListStartSweep()

```
SM_API SmStatus smIQSweepListStartSweep (
            int device,
            int pos,
            void * dst,
            int64_t * timestamps )
```

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

Parameters

in	*device*	Device handle.
in	*pos*	Sweep queue position. Must be between [0,15].
out	*dst*	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	*timestamps*	Pointer to memory allocated for timestamps. The user must allocate this memory before calling these functions. Must be an array of steps int64_t's, where steps are 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

13.1.4.99 smIQSweepListFinishSweep()

```
SM_API SmStatus smIQSweepListFinishSweep (
            int device,
            int pos )
```

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

Parameters

in	*device*	Device handle.
in	*pos*	Sweep queue position. Must be betwee [0,15].

Returns

13.1.4.100 smSegIQCaptureStart()

```
SM_API SmStatus smSegIQCaptureStart (
            int device,
            int capture )
```

Initializes a segmented I/Q capture with the given capture index. If no other captures are active, this capture begins immediately.

Parameters

in	*device*	Device handle.
in	*capture*	Capture index, must be between [0, maxCaptures-1].

Returns

13.1.4.101 smSegIQCaptureWait()

```
SM_API SmStatus smSegIQCaptureWait (
            int device,
            int capture )
```

Waits for a capture to complete. This is a blocking function. To determine if a capture is complete without blocking, use the smSegIQCaptureWaitAsync function.

Parameters

in	*device*	Device index.
in	*capture*	Capture index, must be between [0, maxCaptures-1].

Returns

13.1.4.102 smSegIQCaptureWaitAsync()

```
SM_API SmStatus smSegIQCaptureWaitAsync (
            int device,
```

```
int capture,
SmBool * completed )
```

Queries whether the capture is completed. This is a non-blocking function.

Parameters

in	*device*	Device handle.
in	*capture*	Capture index, must be between [0, maxCaptures-1].
out	*completed*	Returns true if the capture is completed.

Returns

13.1.4.103 smSegIQCaptureTimeout()

```
SM_API SmStatus smSegIQCaptureTimeout (
            int device,
            int capture,
            int segment,
            SmBool * timedOut )
```

Determines if the capture timed out.

Parameters

in	*device*	Device handle.
in	*capture*	Capture index, must be between [0, maxCaptures-1].
in	*segment*	Segment index within capture. Must be between [0,segmentCount-1].
out	*timedOut*	Returns true if the segment was not triggered in time according to the configuration and a timeout occurred.

Returns

13.1.4.104 smSegIQCaptureTime()

```
SM_API SmStatus smSegIQCaptureTime (
            int device,
            int capture,
            int segment,
            int64_t * nsSinceEpoch )
```

Retrieve the timestamp of the capture. The capture should be completed before calling this function.

Parameters

in	*device*	Device handle.
in	*capture*	Capture index, must be between [0, maxCaptures-1].
in	*segment*	Segment index within capture. Must be between [0,segmentCount-1].
out	*nsSinceEpoch*	Nanoseconds since epoch of first sample in capture. If the GPS is locked, this time is synchronized to GPS, otherwise the time is synchronized to the system clock and the system sample rate. When using the system clock, the PC system clock is cached for the first time returned, and all subsequent timings are extrapolated from the first clock using the system clock. If over 16 seconds pass between segment acquisitions, a new CPU system clock is cached. This ensures very accurate relative timings for closely spaced acquisitions when a GPS is not present.

Returns

13.1.4.105 smSegIQCaptureRead()

```
SM_API SmStatus smSegIQCaptureRead (
            int device,
            int capture,
            int segment,
            void * iq,
            int offset,
            int len )
```

Retrieves the I/Q sampes of the capture. The capture should be completed before calling this function.

Parameters

in	*device*	Device handle.
in	*capture*	Capture index, must be between [0, maxCaptures-1].
in	*segment*	Segment index within capture. Must be between [0,segmentCount-1].
out	*iq*	User provided I/Q buffer of len complex samples. Should be large enough to accommodate 32-bit complex floats or 16-bit complex shorts depending on the data type selected by smSegIQSetDataType.
in	*offset*	I/Q sample offset into segment to retrieve.
in	*len*	Number of samples after the offset to retrive.

Returns

13.1.4.106 smSegIQCaptureFinish()

```
SM_API SmStatus smSegIQCaptureFinish (
            int device,
            int capture )
```

Frees the capture so that it can be started again.

Parameters

in	*device*	Device handle.
in	*capture*	Capture index, must be between [0, maxCaptures-1].

Returns

13.1.4.107 smSegIQCaptureFull()

```
SM_API SmStatus smSegIQCaptureFull (
            int device,
            int capture,
            void * iq,
            int offset,
            int len,
            int64_t * nsSinceEpoch,
            SmBool * timedOut )
```

Convenience function for captures that only have 1 segment. Performs the full start/wait/time/read/finish sequences.

Parameters

in	*device*	Device handle.
in	*capture*	Capture index, must be between [0, maxCaptures-1].
out	*iq*	User provided I/Q buffer of len complex samples. Should be large enough to accommodate 32-bit complex floats or 16-bit complex shorts depending on the data type selected by smSegIQSetDataType.
in	*offset*	I/Q sample offset into segment to retrieve.
in	*len*	Number of samples after the offset to retrive.
out	*nsSinceEpoch*	Nanoseconds since epoch of first sample in capture.
out	*timedOut*	Returns true if the segment was not triggered in time according to the configuration and a timeout occurred.

Returns

13.1.4.108 smSegIQLTEResample()

```
SM_API SmStatus smSegIQLTEResample (
            float * input,
            int inputLen,
            float * output,
            int * outputLen,
            bool clearDelayLine )
```

This function is a convenience function for resampling the 250MS/s I/Q output of the segmented I/Q captures to a 245.76MS/s rate required for LTE demodulation. This is a complex to complex resample using a polyphase resample filter with resample fraction 3072/3125. Filter performance is ~24M samples per second. For example, if you provided a 200M sample input, this function would take approximately 8.3 seconds to complete.

Parameters

in	*input*	Pointer to input array. Input array should be interleaved I/Q samples retrieved from the segmented I/Q capture functions.
in	*inputLen*	Number of complex I/Q samples in input.
out	*output*	Pointer to destination buffer. Should be large enough to accept a resampled input. To guarantee this, a simple approach would be to ensure the output buffer is the same size as the input buffer.
in,out	*outputLen*	The integer pointed to by outputLen should initially be the size of the output buffer. If the function returns successfully, the integer pointed to by outputLen will contain the number of I/Q samples in the output buffer.
in	*clearDelayLine*	Set to true to clear the filter delay line. Set to true when providing the first set of samples in a capture. If the samples provided are a continuation of a capture, set this to false.

Returns

13.1.4.109 smSetIQFullBandAtten()

```
SM_API SmStatus smSetIQFullBandAtten (
            int device,
            int atten )
```

Configure the attenuation for the full band I/Q measurement.

Parameters

in	*device*	Device handle.
in	*atten*	Value between [0,6]. Sets the attenuator in 5dB steps between [0,30]dB. Cannot be set to auto (-1).

Returns

13.1.4.110 smSetIQFullBandCorrected()

```
SM_API SmStatus smSetIQFullBandCorrected (
            int device,
            SmBool corrected )
```

Enable/disable the I/Q flatness and imbalance corrections for full band I/Q measurements.

Parameters

in	*device*	Device handle.
in	*corrected*	When set to smTrue, the IF image and flatness response is corrected. When set to smFalse, no corrections are applied. RF leveling corrections are not applied at any point. Data returned is full scale.

Returns

13.1.4.111 smSetIQFullBandSamples()

```
SM_API SmStatus smSetIQFullBandSamples (
            int device,
            int samples )
```

Set the number of samples to be collected in full band I/Q measurements.

Parameters

in	*device*	Device handle.
in	*samples*	Number of samples between [2048, 32768]. For full band I/Q sweeps, this is the number of samples to be collected at each freuqency.

Returns

13.1.4.112 smSetIQFullBandTriggerType()

```
SM_API SmStatus smSetIQFullBandTriggerType (
```

```
int device,
SmTriggerType triggerType )
```

Configure the I/Q full band trigger type.

Parameters

in	*device*	Device handle.
in	*triggerType*	Can be set to immediate, video, or external only. Video trigger is only available for certain devices. See I/Q Full Band for more information.

Returns

13.1.4.113 smSetIQFullBandVideoTrigger()

```
SM_API SmStatus smSetIQFullBandVideoTrigger (
            int device,
            double triggerLevel )
```

Configure the video trigger level for full band I/Q measurements.

Parameters

in	*device*	Device handle.
in	*triggerLevel*	Trigger level in dBFS.

Returns

13.1.4.114 smSetIQFullBandTriggerTimeout()

```
SM_API SmStatus smSetIQFullBandTriggerTimeout (
            int device,
            double triggerTimeout )
```

Specify the video/external trigger timeout for full band I/Q captures. This is how long the device will wait for a trigger. This setting can only be changed for devices that support video triggering. See I/Q Full Band for more information. If the device does not support video triggering, this value is fixed at 1 second.

Parameters

in	*device*	Device handle.
in	*triggerTimeout*	Timeout in seconds. Can be between [0,1].

Returns

13.1.4.115 smGetIQFullBand()

```
SM_API SmStatus smGetIQFullBand (
            int device,
            float * iq,
            int freq )
```

The device must be idle to call this function. When this function returns the device is left in the idle state. This function fully configures and performs this measurement before returning. Calling smConfigure is not required.

See smSetIQFullBand*** functions for all configuration parameters associated with this capture.

This function acquires I/Q samples at the baseband 500MS/s sample rate at a single frequency. While the IF flatness and image corrections can be applied, RF leveling corrections are not applied. The I/Q data is in full scale. This function can be useful for measuring short transients or fast rise times.

When external or video trigger is selected, the SM device will wait up to the configured timeout period before capturing. The capture will automatically trigger after this wait period and no trigger has occurred. If the trigger is detected, the capture will return immediately.

There is approximately 48ns of pre-trigger for any video or external trigger capture. This is ∼24 samples of pre-trigger.

There is no indication that the trigger occurred. The customer will need to inspect the data to verify if a trigger occurred. One possible way to detect whether a trigger occurred is to time the duration of this function call with a long trigger timeout (for example, 1 second). If the function returns much sooner than the timeout period, a trigger likely occurred.

See the SDK for an example of using this function.

Parameters

in	*device*	Device handle.
out	*iq*	Pointer to array of interleaved I/Q values. The size of the array should be equal to the number of samples set in the smSetIQFullBandSamples function. When the function returns successfully, this array will contain the captured data.
in	*freq*	Sets the frequency in 39.0625MHz steps. The center frequency of the capture is equal to (freq + 1) * 39.0625MHz.

Returns

13.1.4.116 smGetIQFullBandSweep()

```
SM_API SmStatus smGetIQFullBandSweep (
            int device,
            float * iq,
            int startIndex,
            int stepSize,
            int steps )
```

The device must be idle to call this function. When this function returns the device is left in the idle state. This function fully configures and performs this measurement before returning. Calling smConfigure is not required.

See smSetIQFullBand∗∗∗ functions for all configuration parameters associated with this capture.

This function acquires I/Q samples at the baseband 500MS/s sample rate at several different frequencies.

The frequency indices that data are collected at are equal to

startIndex + N ∗ stepSize

where N is in the range of [0, steps-1]. The center frequency of any given frequency index is,

(index + 1) ∗ 39.0625MHz.

For example, with a startIndex = 26, step = 4, and stepSize = 9, the frequency indices at which data is collected are

(26, 35, 44, 53)

which correspond to the center frequencies,

(1054.6875 MHz, 1406.25 MHz, 1757.8125 MHz, 2070.3125 MHz).

Center frequencies below 650MHz have reduced bandwidth. Above 650MHz, the maximum step size while still maintaining full frequency coverage is 9 (or 351.5625 MHz per step). Reasonable parameters for sweeping from 600MHz to 20GHz with full frequency coverage are startIndex = 16, stepSize = 9, and steps = 56.

While the IF flatness and image corrections can be applied, RF leveling corrections are not applied. The amplitude is in full scale. This function can be useful for measuring short transients or fast rise times.

Full band sweeps cannot be used in conjunction with triggering. For full band sweeps, immediate triggering is used.

See the SDK for an example of using this function.

Parameters

in	*device*	Device handle.
out	*iq*	Pointer to an array of steps ∗ samplesPerStep number of interleaved complex values, or 2 ∗ steps ∗ samplesPerStep floating point values. When this function returns data will be stored contiguously in this array. The data at step N is in the index range of [N∗samplesPerStep, (N+1)∗samplesPerStep-1]. (zero-based indexing) Samples per step is determined by the smSetIQFullBandSamples function.
in	*startIndex*	The frequency index of the first acquisition. See the description for how the frequencies are determined.
in	*stepSize*	Determines the frequency index step size between each acquisition. Can be zero or negative. See the description for more information.
in	*steps*	Determines the number of steps at which I/Q data is collected. Must be in the range of [1, 64]. See the description for more information.

Returns

13.1.4.117 smGetAudio()

```
SM_API SmStatus smGetAudio (
            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	*device*	Device handle.
out	*audio*	Pointer to array of 1000 32-bit floats.

Returns

13.1.4.118 smGetGPSInfo()

```
SM_API SmStatus smGetGPSInfo (
            int device,
            SmBool refresh,
            SmBool * 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. For devices with GPS write capability (see Writing Messages to the GPS) this function has slightly modified behavior. The nmea data will update once per second even when GPS lock is not present. This allows users to retrieve msg responses as a result of sending a message with the smWriteToGPS function. 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	*device*	Device handle.
in	*refresh*	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.

Parameters

out	*updated*	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	*secSinceEpoch*	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	*latitude*	Latitude in decimal degrees. If the GPS is not locked, this value will be set to zero. Can be NULL.
out	*longitude*	Longitude in decimal degrees. If the GPS is not locked, this value will be set to zero. Can be NULL.
out	*altitude*	Altitude in meters. If the GPS is not locked, this value will be set to zero. Can be NULL.
out	*nmea*	Pointer to user allocated array of char. The length of this array is specified by the nmeaLen parameter. Can be set to NULL.
in,out	*nmeaLen*	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 nmeaLen characters, including the null terminator. After the function returns, nmeaLen 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

13.1.4.119 smWriteToGPS()

```
SM_API SmStatus smWriteToGPS (
            int device,
            const uint8_t * mem,
            int len )
```

Receivers must have GPS write capability to use this function. See Writing Messages to the GPS. Use this function to send messages to the internal u-blox M8 GPS. Messages provided are rounded/padded up to the next multiple of 4 bytes. The padded bytes are set to zero.

Parameters

in	*device*	Device handle.
in	*mem*	The message to send to the GPS.
in	*len*	The length of the message in bytes.

Returns

13.1.4.120 smSetFanThreshold()

```
SM_API SmStatus smSetFanThreshold (
            int device,
            int 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	*device*	Device handle.
in	*temp*	Temperature in C.

Returns

13.1.4.121 smGetFanThreshold()

```
SM_API SmStatus smGetFanThreshold (
            int device,
            int * temp )
```

Get current fan temperature threshold.

Parameters

in	*device*	Device handle.
out	*temp*	Temperature in C.

Returns

13.1.4.122 smSetIFOutput()

```
SM_API SmStatus smSetIFOutput (
            int device,
            double frequency )
```

This command configures the down converted IF output. The device must be idle before calling this function. When this function returns successfully, the RF input will be down converted from the specified frequency to 1.5↵ GHz, output on the 10MHz output port. Use smSetAttenuator and smSetRefLevel to adjust device sensitivity. See SM435 IF Output Option for more information.

Parameters

in	*device*	Device handle.
in	*frequency*	Input frequency between [24GHz, 43.5GHz], as Hz.

Returns

13.1.4.123 smGetCalDate()

```
SM_API SmStatus smGetCalDate (
            int device,
            uint64_t * lastCalDate )
```

Returns the last device adjustment date.

Parameters

in	*device*	Device handle.
out	*lastCalDate*	Last adjustment data as seconds since epoch.

Returns

13.1.4.124 smBroadcastNetworkConfig()

```
SM_API SmStatus smBroadcastNetworkConfig (
            const char * hostAddr,
            const char * deviceAddr,
            uint16_t port,
            SmBool nonVolatile )
```

This function is for networked devices only. This function broadcasts a configuration UDP packet on the host network interface with the IP specified by hostAddr. The device will take on the IP address and port specified by deviceAddr and port.

Parameters

in	*hostAddr*	This is the host IP address the broadcast message will be sent on.
in	*deviceAddr*	This is the address the device will use if it receives the broadcast message.
in	*port*	This is the port the device will use if it receives the broadcast message.
in	*nonVolatile*	If set to true, the device will use the address and port on future power ups. As this requires a flash erase/write, setting this value to true reduces the life of the flash memory on the device. We recommend either setting this value to false and broadcasting the configuration before each connect, or only setting the device once up front with the nonvolatile flag set to true.

Returns

13.1.4.125 smNetworkConfigGetDeviceList()

```
SM_API SmStatus smNetworkConfigGetDeviceList (
            int * serials,
            int * deviceCount )
```

This function is part of a group of functions used to configure the network settings of a networked SM device over the USB 2.0 port. The handle used for these functions can only be used with the other network config functions.

Parameters

out	*serials*	Array of ints. Must be as big as deviceCount.
in,out	*deviceCount*	Point to int that contains the size of the serials array. When the function returns it will contain the number of devices returned.

Returns

13.1.4.126 smNetworkConfigOpenDevice()

```
SM_API SmStatus smNetworkConfigOpenDevice (
            int * device,
            int serialNumber )
```

This function is part of a group of functions used to configure the network settings of a networked SM device over the USB 2.0 port. The handle used for these functions can only be used with the other network config functions.

Parameters

out	*device*	If successful, device will point to an integer that can be used to
in	*serialNumber*	Serial number of the device to open. A list of connected serial numbers can be retrieved from the smNetworkConfigGetDeviceList function.

Returns

13.1.4.127 smNetworkConfigCloseDevice()

```
SM_API SmStatus smNetworkConfigCloseDevice (
            int device )
```

This function is part of a group of functions used to configure the network settings of a networked SM device over the USB 2.0 port. The handle used for these functions can only be used with the other network config functions.

Closes the device and frees any resources. The handle should be closed before interfacing the device through the main API interface.

Parameters

in	*device*	Device handle.

Returns

13.1.4.128 smNetworkConfigGetMAC()

```
SM_API SmStatus smNetworkConfigGetMAC (
            int device,
            char * mac )
```

This function is part of a group of functions used to configure the network settings of a networked SM device over the USB 2.0 port. The handle used for these functions can only be used with the other network config functions.

Retrieve the device MAC address.

Parameters

in	*device*	Device handle.
out	*mac*	Pointer to char buffer, to contain the null terminated MAC address string of the unit, with the following format “XX-XX-XX-XX-XX-XX”. Must be large enough to accommodate this string including null termination.

Returns

13.1.4.129 smNetworkConfigSetIP()

```
SM_API SmStatus smNetworkConfigSetIP (
            int device,
```

```
            const char * addr,
            SmBool nonVolatile )
```

This function is part of a group of functions used to configure the network settings of a networked SM device over the USB 2.0 port. The handle used for these functions can only be used with the other network config functions.

Set device IP address.

Parameters

in	*device*	Device handle.
in	*addr*	pointer to char buffer, to contain the null terminated IP address string of the form "xxx.xxx.xxx.xxx". For functions retrieving the IP address, the buffer must be large enough to hold this string including null termination.
in	*nonVolatile*	When set to smTrue, setting applied will be written to internal flash, which will persist through a device power cycle.

Returns

13.1.4.130 smNetworkConfigGetIP()

```
SM_API SmStatus smNetworkConfigGetIP (
            int device,
            char * addr )
```

This function is part of a group of functions used to configure the network settings of a networked SM device over the USB 2.0 port. The handle used for these functions can only be used with the other network config functions.

Get device IP address.

Parameters

in	*device*	Device handle.
out	*addr*	pointer to char buffer, to contain the null terminated IP address string of the form "xxx.xxx.xxx.xxx". For functions retrieving the IP address, the buffer must be large enough to hold this string including null termination.

Returns

13.1.4.131 smNetworkConfigSetPort()

```
SM_API SmStatus smNetworkConfigSetPort (
            int device,
```

```
int port,
SmBool nonVolatile )
```

This function is part of a group of functions used to configure the network settings of a networked SM device over the USB 2.0 port. The handle used for these functions can only be used with the other network config functions.

Set the device IP port.

Parameters

in	*device*	Device handle.
in	*port*	Port number.
in	*nonVolatile*	When set to smTrue, setting applied will be written to internal flash, which will persist through a device power cycle.

Returns

13.1.4.132 smNetworkConfigGetPort()

```
SM_API SmStatus smNetworkConfigGetPort (
    int device,
    int * port )
```

This function is part of a group of functions used to configure the network settings of a networked SM device over the USB 2.0 port. The handle used for these functions can only be used with the other network config functions.

Get the device IP port.

Parameters

in	*device*	Device handle.
out	*port*	Port number.

Returns

13.1.4.133 smGetAPIVersion()

```
SM_API const char * smGetAPIVersion ( )
```

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"

13.1.4.134 smGetErrorString()

```
SM_API const char * smGetErrorString (
            SmStatus status )
```

Retrieve a descriptive string of a SmStatus enumeration. Useful for debugging and diagnostic purposes.

Parameters

in	*status*	Status code returned from any API function.

Returns

13.2 sm_api.h

Go to the documentation of this file.

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

#ifndef SM_API_H
#define SM_API_H

#if defined(_WIN32) // Windows
#ifdef SM_EXPORTS
#define SM_API __declspec(dllexport)
#else
#define SM_API
#endif

    // bare minimum stdint typedef support
#if _MSC_VER < 1700 // For VS2010 or earlier
        typedef signed char        int8_t;
        typedef short              int16_t;
        typedef int                int32_t;
        typedef long long          int64_t;
        typedef unsigned char      uint8_t;
        typedef unsigned short     uint16_t;
        typedef unsigned int       uint32_t;
        typedef unsigned long long uint64_t;
#else
#include <stdint.h>
#endif

#define SM_DEPRECATED(comment) __declspec(deprecated(comment))
#else // Linux
#include <stdint.h>
#define SM_API __attribute__((visibility("default")))

#if defined(__GNUC__)
#define SM_DEPRECATED(comment) __attribute__((deprecated))
#else
```

```
#define SM_DEPRECATED(comment) comment
#endif
#endif

#define SM_INVALID_HANDLE (-1)

#define SM_TRUE (1)
#define SM_FALSE (0)

#define SM_MAX_DEVICES (9)

#define SM_ADDR_ANY ("0.0.0.0")
#define SM_DEFAULT_ADDR ("192.168.2.10")
#define SM_DEFAULT_PORT (51665)

#define SM_AUTO_ATTEN (-1)
#define SM_MAX_ATTEN (6)
#define SM_MAX_REF_LEVEL (20.0)

#define SM_MAX_SWEEP_QUEUE_SZ (16)

#define SM200_MIN_FREQ (100.0e3)
#define SM200_MAX_FREQ (20.6e9)
#define SM435_MIN_FREQ (100.0e3)
#define SM435_MAX_FREQ (44.2e9)
#define SM435_MAX_FREQ_IF_OPT (40.8e9)

#define SM_MAX_IQ_DECIMATION (4096)

#define SM_PRESELECTOR_MAX_FREQ (645.0e6)

#define SM_FAST_SWEEP_MIN_RBW (30.0e3)

#define SM_REAL_TIME_MIN_SPAN (200.0e3)
#define SM_REAL_TIME_MAX_SPAN (160.0e6)

#define SM_MIN_SWEEP_TIME (1.0e-6)
#define SM_MAX_SWEEP_TIME (100.0)

#define SM_SPI_MAX_BYTES (4)

#define SM_GPIO_SWEEP_MAX_STEPS (64)

#define SM_GPIO_SWITCH_MAX_STEPS (64)
#define SM_GPIO_SWITCH_MIN_COUNT (2)
#define SM_GPIO_SWITCH_MAX_COUNT (4194303 - 1)

#define SM_TEMP_WARNING (95.0)
#define SM_TEMP_MAX (102.0)

#define SM_MAX_SEGMENTED_IQ_SEGMENTS (250)
#define SM_MAX_SEGMENTED_IQ_SAMPLES (520e6)

#define SM435_IF_OUTPUT_FREQ (1.5e9)
#define SM435_IF_OUTPUT_MIN_FREQ (24.0e9)
#define SM435_IF_OUTPUT_MAX_FREQ (43.5e9)

typedef enum SmStatus {
    // Internal use
    smCalErr = -1003,
    // Internal use
    smMeasErr = -1002,
    // Internal use
    smErrorIOErr = -1001,

    smInvalidCalibrationFileErr = -200,
    smInvalidCenterFreqErr = -101,
    smInvalidIQDecimationErr = -100,

    smJESDErr = -54,
    smNetworkErr = -53,
    smFx3RunErr = -52,
    smMaxDevicesConnectedErr = -51,
    smFPGABootErr = -50,
    smBootErr = -49,

    smGpsNotLockedErr = -16,
    smVersionMismatchErr = -14,
    smAllocationErr = -13,

    smSyncErr = -11,
    smInvalidSweepPosition = -10,
    smInvalidConfigurationErr = -8,
    smConnectionLostErr = -6,
    smInvalidParameterErr = -5,
    smNullPtrErr = -4,
    smInvalidDeviceErr = -3,
```

```
    smDeviceNotFoundErr = -2,

    smNoError = 0,

    smSettingClamped = 1,
    smAdcOverflow = 2,
    smUncalData = 3,
    smTempDriftWarning = 4,
    smSpanExceedsPreselector = 5,
    smTempHighWarning = 6,
    smCpuLimited = 7,
    smUpdateAPI = 8,
    smInvalidCalData = 9,
} SmStatus;

typedef enum SmDataType {
    smDataType32fc,
    smDataType16sc
} SmDataType;

typedef enum SmMode {
    smModeIdle = 0,
    smModeSweeping = 1,
    smModeRealTime = 2,
    smModeIQStreaming = 3,
    smModeIQSegmentedCapture = 5,
    smModeIQSweepList = 6,
    smModeAudio = 4,

    // Deprecated, use smModeIQStreaming
    smModeIQ = 3,
} SmMode;

typedef enum SmSweepSpeed {
    smSweepSpeedAuto = 0,
    smSweepSpeedNormal = 1,
    smSweepSpeedFast = 2
} SmSweepSpeed;

typedef enum SmIQStreamSampleRate {
    smIQStreamSampleRateNative = 0,
    smIQStreamSampleRateLTE = 1,
} SmIQStreamSampleRate;

typedef enum SmPowerState {
    smPowerStateOn = 0,
    smPowerStateStandby = 1
} SmPowerState;

typedef enum SmDetector {
    smDetectorAverage = 0,
    smDetectorMinMax = 1
} SmDetector;

typedef enum SmScale {
    smScaleLog = 0,
    smScaleLin = 1,
    smScaleFullScale = 2
} SmScale;

typedef enum SmVideoUnits {
    smVideoLog = 0,
    smVideoVoltage = 1,
    smVideoPower = 2,
    smVideoSample = 3
} SmVideoUnits;

typedef enum SmWindowType {
    smWindowFlatTop = 0,
    smWindowNutall = 2,
    smWindowBlackman = 3,
    smWindowHamming = 4,
    smWindowGaussian6dB = 5,
    smWindowRect = 6
} SmWindowType;

typedef enum SmTriggerType {
    smTriggerTypeImm = 0,
    smTriggerTypeVideo = 1,
    smTriggerTypeExt = 2,
    smTriggerTypeFMT = 3
} SmTriggerType;

typedef enum SmTriggerEdge {
    smTriggerEdgeRising = 0,
    smTriggerEdgeFalling = 1
} SmTriggerEdge;
```

```
typedef enum SmBool {
    smFalse = 0,
    smTrue = 1
} SmBool;

typedef enum SmGPIOState {
    smGPIOStateOutput = 0,
    smGPIOStateInput = 1
} SmGPIOState;

typedef enum SmReference {
    smReferenceUseInternal = 0,
    smReferenceUseExternal = 1
} SmReference;

typedef enum SmDeviceType {
    smDeviceTypeSM200A = 0,
    smDeviceTypeSM200B = 1,
    smDeviceTypeSM200C = 2,
    smDeviceTypeSM435B = 3,
    smDeviceTypeSM435C = 4
} SmDeviceType;

typedef enum SmAudioType {
    smAudioTypeAM = 0,
    smAudioTypeFM = 1,
    smAudioTypeUSB = 2,
    smAudioTypeLSB = 3,
    smAudioTypeCW = 4
} SmAudioType;

typedef enum SmGPSState {
    smGPSStateNotPresent = 0,
    smGPSStateLocked = 1,
    smGPSStateDisciplined = 2
} SmGPSState;

typedef struct SmGPIOStep {
    double freq;
    uint8_t mask;
} SmGPIOStep;

typedef struct SmDeviceDiagnostics {
    float voltage;
    float currentInput;
    float currentOCXO;
    float current58;
    float tempFPGAInternal;
    float tempFPGANear;
    float tempOCXO;
    float tempVCO;
    float tempRFBoardLO;
    float tempPowerSupply;
} SmDeviceDiagnostics;

#ifdef __cplusplus
extern "C" {
#endif

SM_API SmStatus smGetDeviceList(int *serials, int *deviceCount);

SM_API SmStatus smGetDeviceList2(int *serials, SmDeviceType *deviceTypes, int *deviceCount);

SM_API SmStatus smOpenDevice(int *device);

SM_API SmStatus smOpenDeviceBySerial(int *device, int serialNumber);

SM_API SmStatus smOpenNetworkedDevice(int *device,
                                      const char *hostAddr,
                                      const char *deviceAddr,
                                      uint16_t port);

SM_API SmStatus smCloseDevice(int device);

SM_API SmStatus smPreset(int device);

SM_API SmStatus smPresetSerial(int serialNumber);

SM_API SmStatus smNetworkedSpeedTest(int device, double durationSeconds, double *bytesPerSecond);

SM_API SmStatus smGetDeviceInfo(int device, SmDeviceType *deviceType, int *serialNumber);

SM_API SmStatus smGetFirmwareVersion(int device, int *major, int *minor, int *revision);

SM_API SmStatus smHasIFOutput(int device, SmBool *present);

```

```
SM_API SmStatus smGetDeviceDiagnostics(int device, float *voltage, float *current, float *temperature);
SM_API SmStatus smGetFullDeviceDiagnostics(int device, SmDeviceDiagnostics *diagnostics);
SM_API SmStatus smGetSFPDiagnostics(int device,
                                    float *temp,
                                    float *voltage,
                                    float *txPower,
                                    float *rxPower);
SM_API SmStatus smSetPowerState(int device, SmPowerState powerState);
SM_API SmStatus smGetPowerState(int device, SmPowerState *powerState);
SM_API SmStatus smSetAttenuator(int device, int atten);
SM_API SmStatus smGetAttenuator(int device, int *atten);
SM_API SmStatus smSetRefLevel(int device, double refLevel);
SM_API SmStatus smGetRefLevel(int device, double *refLevel);
SM_API SmStatus smSetPreselector(int device, SmBool enabled);
SM_API SmStatus smGetPreselector(int device, SmBool *enabled);
SM_API SmStatus smSetGPIOState(int device, SmGPIOState lowerState, SmGPIOState upperState);
SM_API SmStatus smGetGPIOState(int device, SmGPIOState *lowerState, SmGPIOState *upperState);
SM_API SmStatus smWriteGPIOImm(int device, uint8_t data);
SM_API SmStatus smReadGPIOImm(int device, uint8_t *data);
SM_API SmStatus smWriteSPI(int device, uint32_t data, int byteCount);
SM_API SmStatus smSetGPIOSweepDisabled(int device);
SM_API SmStatus smSetGPIOSweep(int device, SmGPIOStep *steps, int stepCount);
SM_API SmStatus smSetGPIOSwitchingDisabled(int device);
SM_API SmStatus smSetGPIOSwitching(int device, uint8_t *gpio, uint32_t *counts, int gpioSteps);
SM_API SmStatus smSetExternalReference(int device, SmBool enabled);
SM_API SmStatus smGetExternalReference(int device, SmBool *enabled);
SM_API SmStatus smSetReference(int device, SmReference reference);
SM_API SmStatus smGetReference(int device, SmReference *reference);
SM_API SmStatus smSetGPSTimebaseUpdate(int device, SmBool enabled);
SM_API SmStatus smGetGPSTimebaseUpdate(int device, SmBool *enabled);
SM_API SmStatus smGetGPSHoldoverInfo(int device, SmBool *usingGPSHoldover, uint64_t *lastHoldoverTime);
SM_API SmStatus smGetGPSState(int device, SmGPSState *GPSState);
SM_API SmStatus smSetSweepSpeed(int device, SmSweepSpeed sweepSpeed);
SM_API SmStatus smSetSweepCenterSpan(int device, double centerFreqHz, double spanHz);
SM_API SmStatus smSetSweepStartStop(int device, double startFreqHz, double stopFreqHz);
SM_API SmStatus smSetSweepCoupling(int device, double rbw, double vbw, double sweepTime);
SM_API SmStatus smSetSweepDetector(int device, SmDetector detector, SmVideoUnits videoUnits);
SM_API SmStatus smSetSweepScale(int device, SmScale scale);
SM_API SmStatus smSetSweepWindow(int device, SmWindowType window);
SM_API SmStatus smSetSweepSpurReject(int device, SmBool spurRejectEnabled);
SM_API SmStatus smSetRealTimeCenterSpan(int device, double centerFreqHz, double spanHz);
SM_API SmStatus smSetRealTimeRBW(int device, double rbw);
SM_API SmStatus smSetRealTimeDetector(int device, SmDetector detector);
SM_API SmStatus smSetRealTimeScale(int device, SmScale scale, double frameRef, double frameScale);
SM_API SmStatus smSetRealTimeWindow(int device, SmWindowType window);
SM_API SmStatus smSetIQBaseSampleRate(int device, SmIQStreamSampleRate sampleRate);
```

```
SM_API SmStatus smSetIQDataType(int device, SmDataType dataType);

SM_API SmStatus smSetIQCenterFreq(int device, double centerFreqHz);

SM_API SmStatus smGetIQCenterFreq(int device, double *centerFreqHz);

SM_API SmStatus smSetIQSampleRate(int device, int decimation);

SM_API SmStatus smSetIQBandwidth(int device, SmBool enableSoftwareFilter, double bandwidth);

SM_API SmStatus smSetIQExtTriggerEdge(int device, SmTriggerEdge edge);

SM_API SmStatus smSetIQTriggerSentinel(double sentinelValue);

SM_API SmStatus smSetIQQueueSize(int device, float ms);

SM_API SmStatus smSetIQSweepListDataType(int device, SmDataType dataType);

SM_API SmStatus smSetIQSweepListCorrected(int device, SmBool corrected);

SM_API SmStatus smSetIQSweepListSteps(int device, int steps);

SM_API SmStatus smGetIQSweepListSteps(int device, int *steps);

SM_API SmStatus smSetIQSweepListFreq(int device, int step, double freq);

SM_API SmStatus smSetIQSweepListRef(int device, int step, double level);

SM_API SmStatus smSetIQSweepListAtten(int device, int step, int atten);

SM_API SmStatus smSetIQSweepListSampleCount(int device, int step, uint32_t samples);

SM_API SmStatus smSetSegIQDataType(int device, SmDataType dataType);

SM_API SmStatus smSetSegIQCenterFreq(int device, double centerFreqHz);

SM_API SmStatus smSetSegIQVideoTrigger(int device, double triggerLevel, SmTriggerEdge triggerEdge);

SM_API SmStatus smSetSegIQExtTrigger(int device, SmTriggerEdge extTriggerEdge);

SM_API SmStatus smSetSegIQFMTParams(int device,
                                    int fftSize,
                                    const double *frequencies,
                                    const double *ampls,
                                    int count);

SM_API SmStatus smSetSegIQSegmentCount(int device, int segmentCount);

SM_API SmStatus smSetSegIQSegment(int device,
                                  int segment,
                                  SmTriggerType triggerType,
                                  int preTrigger,
                                  int captureSize,
                                  double timeoutSeconds);

SM_API SmStatus smSetAudioCenterFreq(int device, double centerFreqHz);

SM_API SmStatus smSetAudioType(int device, SmAudioType audioType);

SM_API SmStatus smSetAudioFilters(int device,
                                  double ifBandwidth,
                                  double audioLpf,
                                  double audioHpf);

SM_API SmStatus smSetAudioFMDeemphasis(int device, double deemphasis);

SM_API SmStatus smConfigure(int device, SmMode mode);

SM_API SmStatus smGetCurrentMode(int device, SmMode *mode);

SM_API SmStatus smAbort(int device);

SM_API SmStatus smGetSweepParameters(int device,
                                     double *actualRBW,
                                     double *actualVBW,
                                     double *actualStartFreq,
                                     double *binSize,
                                     int *sweepSize);

SM_API SmStatus smGetRealTimeParameters(int device,
                                        double *actualRBW,
                                        int *sweepSize,
                                        double *actualStartFreq,
                                        double *binSize,
                                        int *frameWidth,
                                        int *frameHeight,
```

```
                                        double *poi);
SM_API SmStatus smGetIQParameters(int device, double *sampleRate, double *bandwidth);
SM_API SmStatus smGetIQCorrection(int device, float *scale);
SM_API SmStatus smIQSweepListGetCorrections(int device, float *corrections);
SM_API SmStatus smSegIQGetMaxCaptures(int device, int *maxCaptures);
SM_API SmStatus smGetSweep(int device, float *sweepMin, float *sweepMax, int64_t *nsSinceEpoch);
SM_API SmStatus smSetSweepGPIO(int device, int pos, uint8_t data);
SM_API SmStatus smStartSweep(int device, int pos);
SM_API SmStatus smFinishSweep(int device, int pos, float *sweepMin, float *sweepMax, int64_t *nsSinceEpoch);
SM_API SmStatus smGetRealTimeFrame(int device,
                                   float *colorFrame,
                                   float *alphaFrame,
                                   float *sweepMin,
                                   float *sweepMax,
                                   int *frameCount,
                                   int64_t *nsSinceEpoch);
SM_API SmStatus smGetIQ(int device,
                        void *iqBuf,
                        int iqBufSize,
                        double *triggers,
                        int triggerBufSize,
                        int64_t *nsSinceEpoch,
                        SmBool purge,
                        int *sampleLoss,
                        int *samplesRemaining);
SM_API SmStatus smIQSweepListGetSweep(int device, void *dst, int64_t *timestamps);
SM_API SmStatus smIQSweepListStartSweep(int device, int pos, void *dst, int64_t *timestamps);
SM_API SmStatus smIQSweepListFinishSweep(int device, int pos);
SM_API SmStatus smSegIQCaptureStart(int device, int capture);
SM_API SmStatus smSegIQCaptureWait(int device, int capture);
SM_API SmStatus smSegIQCaptureWaitAsync(int device, int capture, SmBool *completed);
SM_API SmStatus smSegIQCaptureTimeout(int device, int capture, int segment, SmBool *timedOut);
SM_API SmStatus smSegIQCaptureTime(int device, int capture, int segment, int64_t *nsSinceEpoch);
SM_API SmStatus smSegIQCaptureRead(int device, int capture, int segment, void *iq, int offset, int len);
SM_API SmStatus smSegIQCaptureFinish(int device, int capture);
SM_API SmStatus smSegIQCaptureFull(int device,
                                   int capture,
                                   void *iq,
                                   int offset,
                                   int len,
                                   int64_t *nsSinceEpoch,
                                   SmBool *timedOut);
SM_API SmStatus smSegIQLTEResample(float *input,
                                   int inputLen,
                                   float *output,
                                   int *outputLen,
                                   bool clearDelayLine);
SM_API SmStatus smSetIQFullBandAtten(int device, int atten);
SM_API SmStatus smSetIQFullBandCorrected(int device, SmBool corrected);
SM_API SmStatus smSetIQFullBandSamples(int device, int samples);
SM_API SmStatus smSetIQFullBandTriggerType(int device, SmTriggerType triggerType);
SM_API SmStatus smSetIQFullBandVideoTrigger(int device, double triggerLevel);
SM_API SmStatus smSetIQFullBandTriggerTimeout(int device, double triggerTimeout);
SM_API SmStatus smGetIQFullBand(int device, float *iq, int freq);
SM_API SmStatus smGetIQFullBandSweep(int device, float *iq, int startIndex, int stepSize, int steps);
```

```
SM_API SmStatus smGetAudio(int device, float *audio);

SM_API SmStatus smGetGPSInfo(int device,
                             SmBool refresh,
                             SmBool *updated,
                             int64_t *secSinceEpoch,
                             double *latitude,
                             double *longitude,
                             double *altitude,
                             char *nmea,
                             int *nmeaLen);

SM_API SmStatus smWriteToGPS(int device, const uint8_t *mem, int len);

SM_API SmStatus smSetFanThreshold(int device, int temp);

SM_API SmStatus smGetFanThreshold(int device, int *temp);

SM_API SmStatus smSetIFOutput(int device, double frequency);

SM_API SmStatus smGetCalDate(int device, uint64_t *lastCalDate);

SM_API SmStatus smBroadcastNetworkConfig(const char *hostAddr,
                                         const char *deviceAddr,
                                         uint16_t port,
                                         SmBool nonVolatile);

SM_API SmStatus smNetworkConfigGetDeviceList(int *serials, int *deviceCount);

SM_API SmStatus smNetworkConfigOpenDevice(int *device, int serialNumber);

SM_API SmStatus smNetworkConfigCloseDevice(int device);

SM_API SmStatus smNetworkConfigGetMAC(int device, char *mac);

SM_API SmStatus smNetworkConfigSetIP(int device, const char *addr, SmBool nonVolatile);

SM_API SmStatus smNetworkConfigGetIP(int device, char *addr);

SM_API SmStatus smNetworkConfigSetPort(int device, int port, SmBool nonVolatile);

SM_API SmStatus smNetworkConfigGetPort(int device, int *port);

SM_API const char* smGetAPIVersion();

SM_API const char* smGetErrorString(SmStatus status);

SM_DEPRECATED("smSetIQUSBQueueSize has been deprecated, use smSetIQQueueSize")
SM_API SmStatus smSetIQUSBQueueSize(int device, float ms);

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

// Deprecated macros
#define SM200A_AUTO_ATTEN (SM_AUTO_ATTEN)
#define SM200A_MAX_ATTEN (SM_MAX_ATTEN)
#define SM200A_MAX_REF_LEVEL (SM_MAX_REF_LEVEL)
#define SM200A_MAX_SWEEP_QUEUE_SZ (SM_MAX_SWEEP_QUEUE_SZ)
#define SM200A_MIN_FREQ (SM200_MIN_FREQ)
#define SM200A_MAX_FREQ (SM200_MAX_FREQ)
#define SM200A_MAX_IQ_DECIMATION (SM_MAX_IQ_DECIMATION)
#define SM200A_PRESELECTOR_MAX_FREQ (SM_PRESELECTOR_MAX_FREQ)
#define SM200A_FAST_SWEEP_MIN_RBW (SM_FAST_SWEEP_MIN_RBW)
#define SM200A_RTSA_MIN_SPAN (SM_REAL_TIME_MIN_SPAN)
#define SM200A_RTSA_MAX_SPAN (SM_REAL_TIME_MAX_SPAN)
#define SM200A_MIN_SWEEP_TIME (SM_MIN_SWEEP_TIME)
#define SM200A_MAX_SWEEP_TIME (SM_MAX_SWEEP_TIME)
#define SM200A_SPI_MAX_BYTES (SM_SPI_MAX_BYTES)
#define SM200A_GPIO_SWEEP_MAX_STEPS (SM_GPIO_SWEEP_MAX_STEPS)
#define SM200A_GPIO_SWITCH_MAX_STEPS (SM_GPIO_SWITCH_MAX_STEPS)
#define SM200A_GPIO_SWITCH_MIN_COUNT (SM_GPIO_SWITCH_MIN_COUNT)
#define SM200A_GPIO_SWITCH_MAX_COUNT (SM_GPIO_SWITCH_MAX_COUNT)
#define SM200A_TEMP_WARNING (SM_TEMP_WARNING)
#define SM200A_TEMP_MAX (SM_TEMP_MAX)
#define SM200B_MAX_SEGMENTED_IQ_SEGMENTS (SM_MAX_SEGMENTED_IQ_SEGMENTS)
#define SM200B_MAX_SEGMENTED_IQ_SAMPLES (SM_MAX_SEGMENTED_IQ_SAMPLES)
#define SM200_ADDR_ANY (SM_ADDR_ANY)
#define SM200_DEFAULT_ADDR (SM_DEFAULT_ADDR)
#define SM200_DEFAULT_PORT (SM_DEFAULT_PORT)

#endif // SM_API_H
```

13.3 sm_api_vrt.h File Reference

VITA 49 interface.

Functions

- SM_API SmStatus smSetVrtStreamID (int device, uint32_t sid)
- SM_API SmStatus smGetVrtContextPktSize (int device, uint32_t *wordCount)
- SM_API SmStatus smGetVrtContextPkt (int device, uint32_t *words, uint32_t *wordCount)
- SM_API SmStatus smSetVrtPacketSize (int device, uint16_t samplesPerPkt)
- SM_API SmStatus smGetVrtPacketSize (int device, uint16_t *samplesPerPkt, uint32_t *wordCount)
- SM_API SmStatus smGetVrtPackets (int device, uint32_t *words, uint32_t *wordCount, uint32_t packetCount, SmBool purgeBeforeAcquire)

13.3.1 Detailed Description

VITA 49 interface.

These functions are used to stream I/Q data in the VRT data format.

13.3.2 Function Documentation

13.3.2.1 smSetVrtStreamID()

```
SM_API SmStatus smSetVrtStreamID (
            int device,
            uint32_t sid )
```

Set the stream identifier, which is used to identify each VRT packet with the device.

Parameters

in	*device*	Device handle.
in	*sid*	New stream ID.

Returns

13.3.2.2 smGetVrtContextPktSize()

```
SM_API SmStatus smGetVrtContextPktSize (
            int device,
            uint32_t * wordCount )
```

Retrieve the number of words in a VRT context packet. Use this to allocate an appropriately sized buffer for smGetVrtContextPkt.

Parameters

in	*device*	Device handle.
out	*wordCount*	Returns the number of words in a VRT context packet.

Returns

13.3.2.3 smGetVrtContextPkt()

```
SM_API SmStatus smGetVrtContextPkt (
            int device,
            uint32_t * words,
            uint32_t * wordCount )
```

Retrieve one VRT context packet.

Parameters

in	*device*	Device handle.
out	*words*	User allocated buffer. Should be length wordCount number of words long, where wordCount was returned from smGetVrtContextPktSize.
out	*wordCount*	Number of words written to the words buffer.

Returns

13.3.2.4 smSetVrtPacketSize()

```
SM_API SmStatus smSetVrtPacketSize (
            int device,
            uint16_t samplesPerPkt )
```

This function specifies the number of I/Q samples in each VRT data packet.

Parameters

in	*device*	Device handle.
in	*samplesPerPkt*	The number of I/Q samples.

Returns

13.3.2.5 smGetVrtPacketSize()

```
SM_API SmStatus smGetVrtPacketSize (
            int device,
            uint16_t * samplesPerPkt,
            uint32_t * wordCount )
```

Retrieve the number of words in a VRT data packet. Use this and a user-specified packet count to allocate an appropriately sized buffer for smGetVrtPackets.

Parameters

in	*device*	Device handle.
out	*samplesPerPkt*	Returns the number of I/Q samples in a VRT data packet.
out	*wordCount*	Returns the number of words in a VRT data packet.

Returns

13.3.2.6 smGetVrtPackets()

```
SM_API SmStatus smGetVrtPackets (
            int device,
            uint32_t * words,
            uint32_t * wordCount,
            uint32_t packetCount,
            SmBool purgeBeforeAcquire )
```

Retrieve one block of one or more VRT data packets. This function blocks until the data requested is available.

Parameters

in	*device*	Device handle.
out	*words*	Pointer to user allocated buffer. Returns the VRT packets. Must be as large as packetCount * packetSize words.
out	*wordCount*	The number of words written to the words buffer.
in	*packetCount*	The number of VRT data packets to retrive.
in	*purgeBeforeAcquire*	When set to smTrue, any buffered I/Q data in the API is purged before beginning the I/Q block acquisition. See the section on Streaming I/Q Data for more detailed information.

Returns

13.4 sm_api_vrt.h

Go to the documentation of this file.

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

#ifndef SM_API_VRT_H
#define SM_API_VRT_H

#include "sm_api.h"

SM_API SmStatus smSetVrtStreamID(int device, uint32_t sid);

SM_API SmStatus smGetVrtContextPktSize(int device, uint32_t *wordCount);

SM_API SmStatus smGetVrtContextPkt(int device, uint32_t *words, uint32_t *wordCount);

SM_API SmStatus smSetVrtPacketSize(int device, uint16_t samplesPerPkt);

SM_API SmStatus smGetVrtPacketSize(int device, uint16_t *samplesPerPkt, uint32_t *wordCount);

SM_API SmStatus smGetVrtPackets(int device,
                                uint32_t *words,
                                uint32_t *wordCount,
                                uint32_t packetCount,
                                SmBool purgeBeforeAcquire);

#endif
```

Index