SM200 Application Programming Interface (API)

Programmers Reference Manual

SM200 Application Programming Interface (API)
Programmers Reference Manual

1502 SE Commerce Ave, Suite 101
Battle Ground, WA
Phone 360.313.7997

November 2, 2020

Table of Contents

Overview

This manual is a reference for the Signal Hound SM200 spectrum analyzer programming interface (API). The API provides a low-level set of C routines for interfacing the SM200. The API is C ABI compatible making is possible to be interfaced from most programming languages. See the code examples folder to for examples of using the API in C++ and other various environments.

Contact Information

For all programming and technical questions, please email aj@signalhound.com.

For sales, email sales@signalhound.com.

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 DLL.

Version 1.0.3 – Official release, support for SM200A

Version 1.1.2 – First release with support for Linux operating systems (libusb backend)

Version 2.0.0 – Support for SM200B, LTE I/Q sample rates, and segmented I/Q captures.

Version 2.1.0 – Support for SM200C.

PC Requirements and Setup

Windows Development Requirements

- SM200A/B
  - Windows 7/8/10, (64-bit Win 7/10 recommended)
- SM200C
  - Windows 10 (64-bit recommended)
- (For C/C++ projects only) Windows C/C++ development tools and environment. Preferably Visual Studio 2008 or later. If Visual Studio 2012 is not used, then the VS2012 redistributables will need to be installed.
- Library files sm_api.h, sm_api.lib, and sm_api.dll

Linux Development Requirements

- Linux 64-bit
  - Ubuntu 18.04 (recommended)
  - CentOS 7
- libusb-1.0
- System GCC compiler
- SM200 library files, sm_api.h and libsm_api.so

PC and Other Requirements

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

- SM200A/SM200B
  - 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
  - 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.

Theory of Operation

Any application using the SM200 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)

Opening a Device

Opening a device changes based on the connection type.

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 SM200 devices connected to the PC via the `smGetDeviceList` function.

Opening a networked device is done through the `smOpenNetworkedDevice` function. A networked device can be configured to use specific network settings with the `smBroadcastNetworkConfig` function. All networked SM200 devices have a default network configuration.

Configuring the Device

Once the device is open, the next step is to configure the device for a measurement. The available measurement modes are swept analysis, real-time analysis, and I/Q streaming. 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.

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.

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.

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.

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.

Swept Spectrum Analysis

Swept spectrum analysis represents the common spectrum analyzer measurement of plotting amplitude over frequency. In this measurement mode, the API returns sweeps from the receiver. The API provides a simple interface through `smGetSweep` for acquiring single sweeps, or for high throughput sweep measurements, the `smStartSweep` / `smFinishSweep` functions. Both the sweep format and acquisition methods are described below.

Only 1 sweep configuration can be active at a time. Changing any sweep parameter requires reconfiguring the device with a new sweep configuration.

Sweep Format

Sweeps are returned from the API as 1-dimensional arrays of power values. Each array element corresponds to a specific frequency. The frequency of any given can be calculated as

$$Frequency\ of\ N'th\ sample\ in\ sweep = StartFreq + N * BinSize$$

where `StartFreq` and `BinSize` are reported in the `smGetSweepSettings` function.

Min and Max Sweep Arrays

Several functions in the SM200 API return two arrays for sweeps. They 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 SM200 API reduces this to either minmax and 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 enable, `sweepMin` and `sweepMax` will be identical arrays.

Sweeps are returned for swept analysis and real-time spectrum analysis. Generally, if you are not interested in either the min or max sweep, simply passing a NULL pointer for this parameter will tell the API you wish to ignore this sweep.

Most API users will only be interested in the `sweepMax` array as this will provide peak+ and average detector results. (pass `NULL` to the `sweepMin` array parameter)

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 most types of measurements but does not optimize for receiver sweep speed. USB latency can be very large compared to total acquisition/processing time. To eliminate USB latency, you will need to take advantage of the API sweep queuing mechanisms.

The sweep start/finish function provide a way to eliminate USB latencies between sweeps which allows the device to sustain the full sweep speed throughput. Using the `smStartSweep/smFinishSweep` functions you can start up to 'N' 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.

Note: When using the blocking `smGetSweep` function, the API utilizes the queued start/finish functions with a sweep index of zero (0). This means that if you want to mix the blocking and queue sweep acquisitions, avoid using index zero for queued sweeps.

Sweep Speed

This sweep speed is related to the parameter set in smSetSweepSpeed. See the description for smSetSweepSpeed for more information.

The SM200 has 3 sweeps speeds depending on the user's configuration. The sweep speed is primarily set by the user explicitly, 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.

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.

Normal – At this speed the SM200 steps the LO in 39.0625MHz steps. This mode offers better RF performance than fast sweep mode, with a sweep speed reduction of about 3X.

Fast – At this speed, the SM200 steps the LO in 156.25MHz steps to accomplish up to a 1THz sweep speed. There are additional restrictions that hold when utilizing the fast sweep. The max FFT size is 16K which limits RBW to about 30-60kHz depending on the spectrum window selected. Additionally, VBW and sweep time are not selectable when measuring at the fast sweep rate.

Real-Time Spectrum Analysis

The API provides methods for performing real-time spectrum analysis up to 160MHz in bandwidth.

Real-time spectrum analysis is accomplished for the SM200 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. Spans 40MHz and below are processed on the PC and lower RBWs can be set. See the Real-Time RBW Restrictions for more information.

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

Real-time measurements are performed over short consecutive time frames and returned to the user as frame and sweeps representing spectrum activity over these time periods. The duration of these time periods is ~30ms.

Real-Time Frame

Real-time spectrum analysis returns the sweep, frame, and alphaFrame.

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).

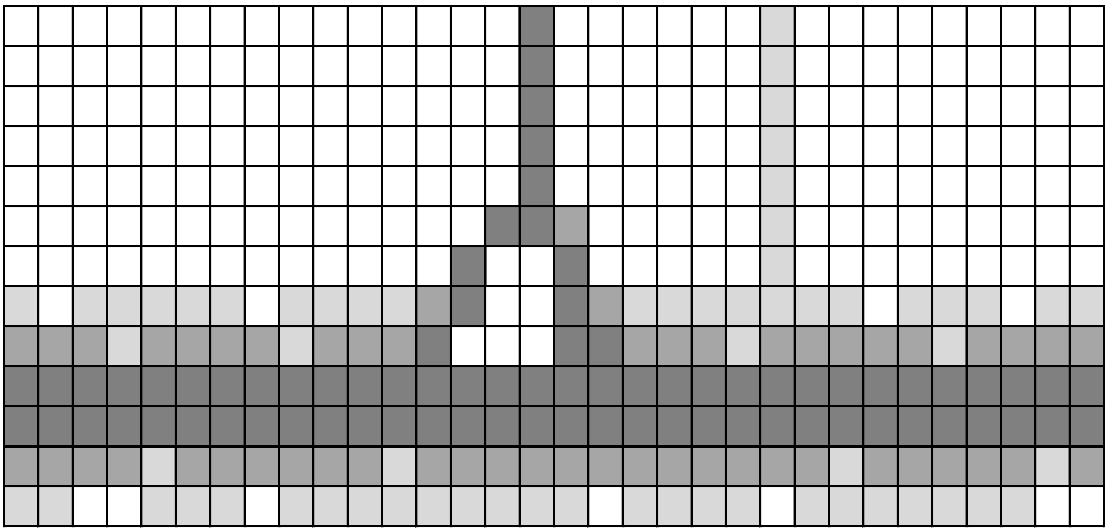

Figure 1: An example of a frame plotted as a gray scale image, mapping the density values between [0.0,1.0] to gray scale values between [0,255]. The frame shows a persistent CW signal near the center frequency and a short-lived CW signal.

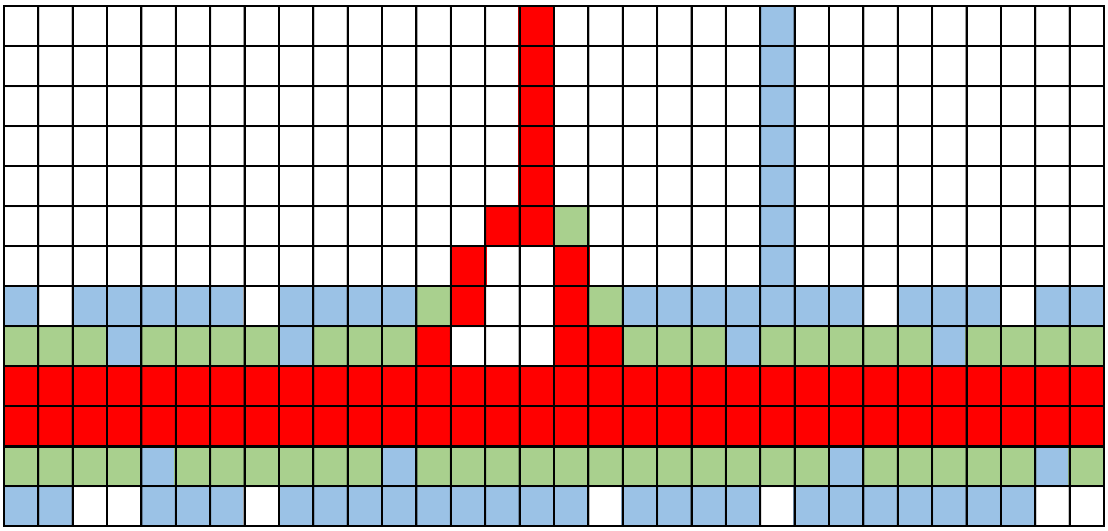

Figure 2: The same frame above as is plotted in Spike, where density values are mapped onto a color spectrum.

Real-Time Sweep

The sweeps returned in real-time spectrum analysis are the result of applying the detector to all FFTs that occur during the real-time frame period. A min/max detector will hold the min and maximum amplitudes seen during the frame period. The average detector will average all sweeps together during this period.

Streaming I/Q

The API provides the ability to stream I/Q samples up to the device's native sample rate or common LTE sample rates. I/Q data can be retrieved as 32-bit complex floats or 16-bit complex shorts. I/Q data provided as 32-bit floats are corrected for amplitude flatness, and I/Q imbalance. The I/Q data stream can be tuned to any frequency within the SM200 frequency range.

Sample Rate, Decimation, and Bandwidth

The I/Q data stream can be decimated by powers of two 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.

(SM200A/B 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.

(SM200C only) Custom cutoff frequencies are performed on the device with no CPU penalty, and as such these filters are always active.

For 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.

Polling Interface (I/Q)

The API for the I/Q data stream is a polling style interface, where the application must request I/Q data in blocks that will keep up with the device acquisition of data. The API 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 data loss does not occur.

See the following references for more information.

`smSetIQ***`

`smGetIQ`

Appendix: I/Q Acquisition

Segmented I/Q Acquisitions (SM200B only)

(See the API C++ examples in the SDK for segmented I/Q captures)

The SM200B has 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. The SM200B introduces 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 of the SM200B, configure a sequence of trigger acquisitions to capture low duty cycle signals.

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.
    a. Start a trigger sequence.
    b. Wait for it to finish.
    c. Retrieve the measurement info and data.
    d. Finish the capture. (Frees up resources)
    e. Repeat (go to step a.)

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.

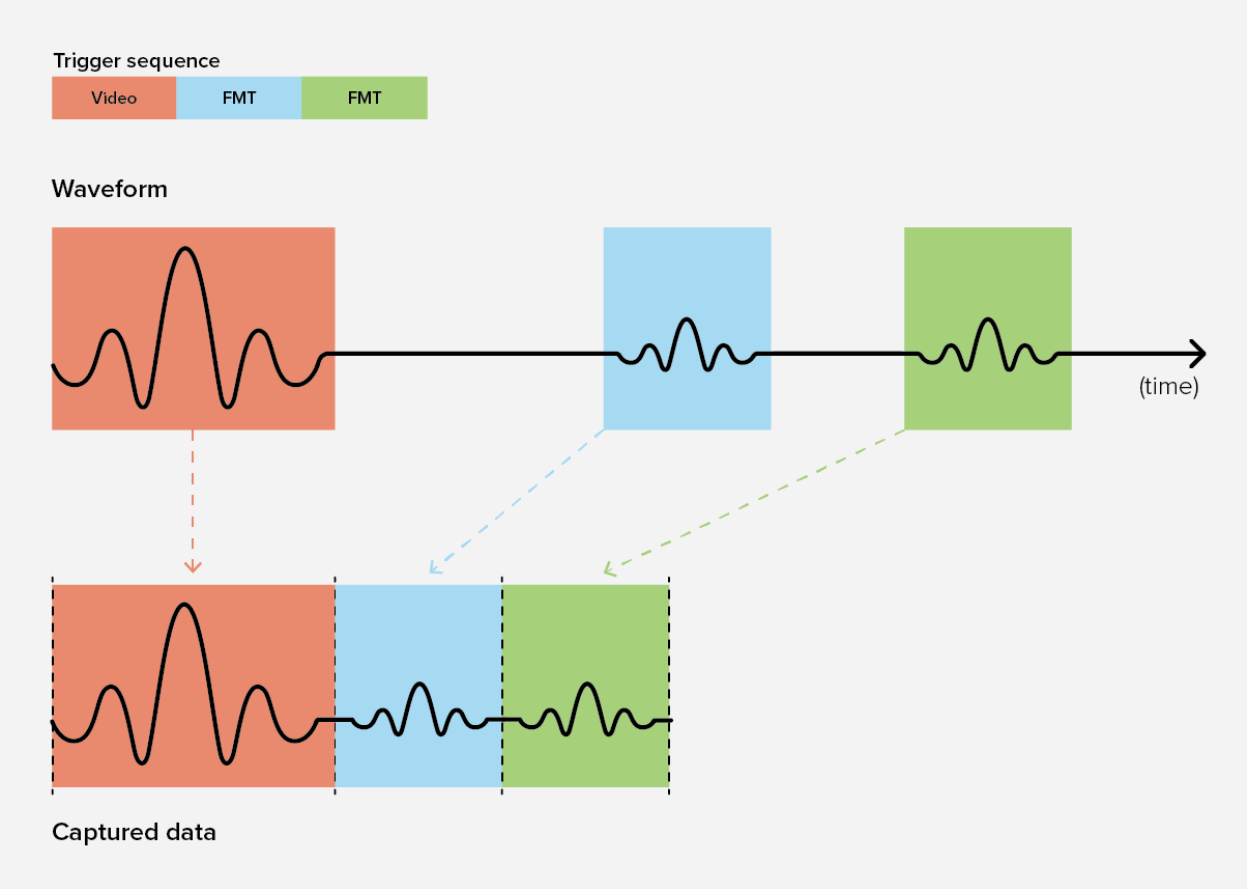

Figure 3: Trigger sequence captures 3 sparse signal events and ignores all other samples.

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

- video trigger type (level/edge)
- external trigger type (edge)
- FMT type (size/mask).

For each trigger in the sequence you can configure

1) The trigger type (using its one configuration)
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 SM200B to remain responsive.

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.

GPS

The internal SM200 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.

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.

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 the GPS data is valid, 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.

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 SM200 cannot achieve it)

The GPS will lock automatically with a GPS antenna attached, but for the GPS to discipline the SM200, 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 SM200 utilize a GPS lock to discipline the SM200 and store holdover values.

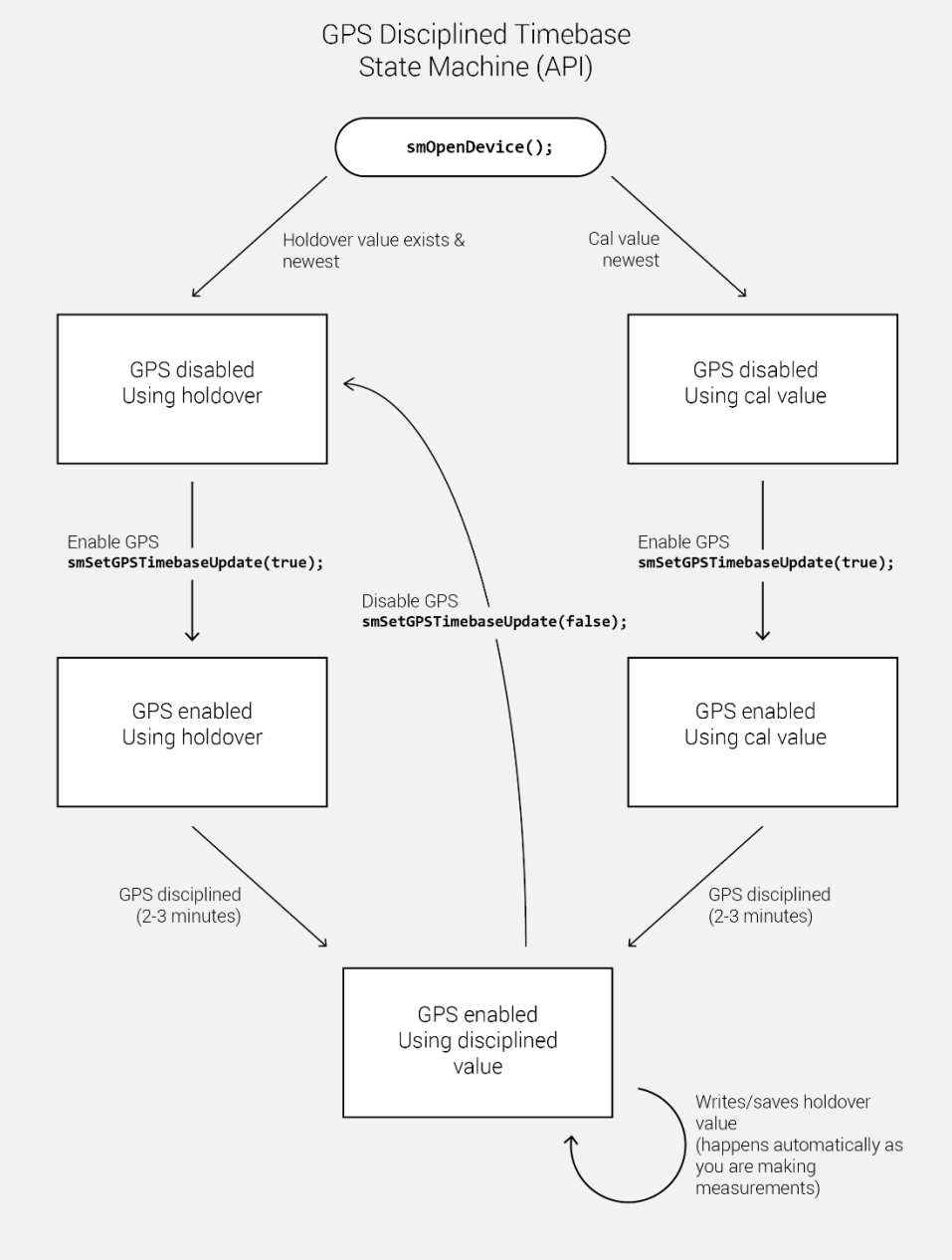

Figure 4: GPS Disciplining State Machine

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

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 SM200 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

GPIO

On the front panel of the SM200 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 a sweep based on frequency.

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

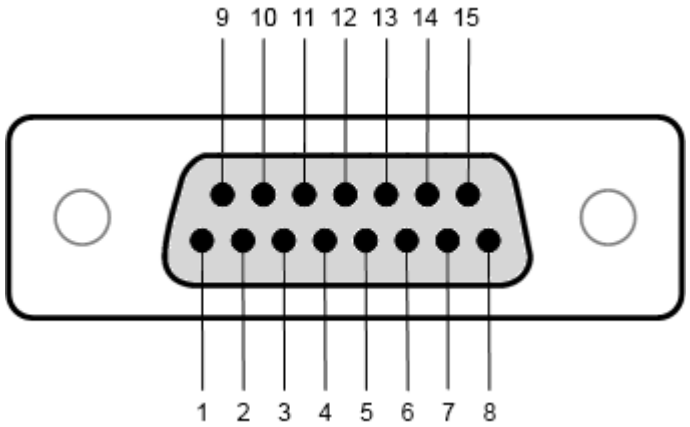

Figure 5: Front Panel Female DB15 Port on SM200

Pinout

1	GPIO(0)		9	GPIO(1)
2	GPIO(2)		10	GPIO(3)
3	V_{dd} in (1.8 to 3.3V)		11	3.3V out (max 30 mA)
4	GND		12	SPI SCLK
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 sections for more information.

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

GPIO Sweeps

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 update specific values. The frequency boundaries and GPIO output settings are configured with `smSetGPIOSweep`. GPIO sweep functionality affects standard swept analysis mode with > 40MHz span only. The frequency boundary resolution is 40MHz for 'normal' sweep speeds and 160MHz for 'fast' sweep speeds.

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.

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`.

SPI

Through the front panel DB15 port the SM200 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.

Power States

The SM200 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`.

Thread Safety

The SM200 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.

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 Appendix: Linux Notes.

Status Codes and Error Handling

All functions return an `SmStatus` error code. `SmStatus` is an enumerated type representing the success of a given function call. The integer values associated with each status provides information about whether a function call succeeded or failed.

An integer value of zero indicates no error or warnings. Negative integer status values indicate errors and positive values represent warnings.

A descriptive string of each status type can be retrieved using the `smGetErrorString` function.

Functions

All functions other that initialization functions take a device handle as the first parameters. This integer is obtained after opening the device through one of the API `smOpenDevice` and `smOpenDeviceBySerial` functions. This handle uniquely identifies the receiver for the duration of the application execution, or until `smCloseDevice` is called.

Each function returns an error code which can provide warnings or errors related to the execution of the function. There are many cases where you will need to monitor these codes to determine the success or failure of an operation. See a list of common error codes and their descriptions in the Appendix.

Common Error Codes

This sections documents some of the more common error codes and their meaning. For a full list of status codes, see the API header file and the API function list in this document.

Negative error codes represent errors and are suffixed with 'Err'. When an error code is returned, the operation requested did not complete. Positive error codes are warnings and indicate that the function/operation completed successfully, but the user might need to take some action.

`smNoError`	Returned when a function returns successfully.
`smInvalidDeviceErr`	Returned when the device handle provided does not match an open device.
`smSettingClamped`	Returned when one or more parameters what clamped to a valid range.
`smInvalidParameterErr`	Returned when one or more parameters is not valid. For instance, if an enum parameter does not match the set of possible values, this error code is returned.
`smNullPtrErr`	Returned when one or more required pointer parameters is NULL. Pointer parameters that can be set to NULL are noted in the parameter descriptions in the Functions section of this document.
`smConnectionLostErr`	Occurs when the PC is no longer able to communicate with the device. Connection must be reestablished by reopening the device. Ensure you call the smCloseDevice before reopening the device to let the system recover any allocated resources, failing to do so will result in memory leaks.
`smInvalidConfigurationErr`	This error occurs when the API is unable to complete an action for which the device is currently configured. For example, trying to retrieve a sweep when the device is configured for I/Q streaming.
`smSyncErr`	This error occurs when the API detects a data framing issue on the data coming from the device. In general, this type of issue is the result of data loss over USB/UDP, which would normally get reported as `smConnectionLostErr`. This error attempts to catch this situation when not the result of typical data loss. If this error is detected, recovery should be to reset the device, manually or through `smPreset`.

smGetDeviceList

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

Parameters

`serials`	Pointer to an array of integers. Must point to an array equal to in length or larger than the number of SM200 devices connected to the PC.
`deviceCount`	Pointer to integer. If the function returns successfully `deviceCount` will be set to the number devices found on the system.

Description

This function is for USB (SM200A/B) devices only.

This function is used to retrieve the serial number of all unopened USB SM200 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 SM200 serial numbers. Only `deviceCount` values will be modified.

This function will not return the serial numbers of any connected networked devices.

smBroadcastNetworkConfig

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

Parameters

`hostAddr`	This is the host IP address the broadcast message will be sent on.
`deviceAddr`	This is the address the device will use if it receives the broadcast message.
`port`	This is the port the device will use if it receives the broadcast message.
`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.

Description

This function is for networked (SM200C) 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`.

Return Values

`smNetworkErr`	Returned if unable to create/bind/connect/broadcast/send from a socket.

smOpenDevice

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

Parameters

device	Pointer to integer to be used as a handle for the device.

Description

This function is for USB (SM200A/B) devices only.

Claim the first unopened USB SM200 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 `smGetDeviceList` and using the first device found to call `smOpenDeviceBySerial`.

Return Values

smDeviceNotFoundErr	Unable to find/open an SM200 receiver.
smBootErr	Unable to finish initial boot sequence. If problem persists, contact Signal Hound.
smFPGABootErr	FPGA failed to initialize. If problem persists, contact Signal Hound.
smFx3RunErr	Unable to enter the final run program. If problem persists, contact Signal Hound.
smMaxDevicesConnectedErr	Cannot connect another device. Max number of devices reached.
smAllocationErr	Failed to allocate memory needed to initialize device.

smOpenDeviceBySerial

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

Parameters

device	Pointer to integer to be used as a handle for the device.
serialNumber	Serial number of the device you wish to open.

Description

This function is like `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 SM200 devices on one PC.

Return Values

See return values for smOpenDevice.

smOpenNetworkedDevice

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

Parameters

device	Pointer to integer to be used as a handle for the device upon a successful open.
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".
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.
port	Target device port.

Description

This function is for networked (SM200C) devices only.

Attempt 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 SM200C takes approximately 12 seconds to boot up after applying power. Until the device is booted, this function will return device not found.

Return Values

See return values for `smOpenDevice`.

smCloseDevice

```
SmStatus smCloseDevice(int device);
```

Description

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.

smPreset

```
SmStatus smPreset(int device);
```

Description

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.

smPresetSerial

```
SmStatus smPresetSerial(int serialNumber);
```

Description

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

`serialNumber`	Serial number of the device to preset.

Return Values

`smDeviceNotFound`	Cannot find the device specified.

smNetworkedSpeedTest

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

Parameters

`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.
`bytesPerSecond`	Pointer to double which when finished, will contain the measured bytes per second throughput between the device and PC.

Description

This function is for networked devices only. (SM200C)

Measure the network throughput between an SM200C and the PC. Useful for troubleshooting network throughput issues.

Return Values

`smInvalidConfigurationErr`	The device is not a networked device or the device is not idle.
`smConnectionLostErr`	Data loss occurred during the test. The device should be closed/re-opened.

smGetDeviceInfo

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

Parameters

`deviceType`	Pointer to SmDeviceType, to contain the device model number. Can be `NULL`.
`serialNumber`	Pointer to integer. If this function returns successfully, the integer pointed to will contain the specified devices serial number. Can be `NULL`.

Description

This function returns basic information about a specific SM200 receiver. Also see `smGetDeviceDiagnostics` and `smGetCalInfo`.

smGetFirmwareVersion

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

Parameters

major	Pointer to 32-bit int. Can be NULL. See description.
minor	Pointer to 32-bit int. Can be NULL. See description.
revision	Pointer to 32-bit int. Can be NULL. See description.

Description

Get the firmware version of an open device. The firmware version is of the form.

```
major.minor.revision
```

smGetDeviceDiagnostics

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

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

Parameters

voltage	Pointer to float, to contain measured device voltage. Can be NULL.
current	Pointer to float, to contain measured device current. Can be NULL.
temperature	Pointer to float, to contain current device internal temperature. Can be NULL.
diagnostics	Pointer to diagnostic struct.

Description

This function returns operational information about a specific SM200 receiver. Also see smGetDeviceInfo and smGetCalInfo.

smGetSFPDiagnostics

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

Parameters

temp	SFP+ reported temperature in C.
voltage	SFP+ reported voltage in V.
txPower	SFP+ reported transmit power in mW.
rxPower	SFP+ reported receive power in mW.

Description

Returns a number of diagnostic information for the SFP+ transceiver attached to the SM200C. If either the device is not an SM200C or the SFP+ does not communicate diagnostic information, the values returned will be zero.

Return Values

`smInvalidConfigurationErr`	The specified device is not an SM200C.

smSetPowerState

```
SmStatus smSetPowerState(int device, SmPowerState powerState);

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

Parameters

`powerState`	New power state.

Description

Change the power state of the SM200. The power state controls the power consumption of the SM200. See Power States for more information.

Return Values

`smInvalidConfigurationErr`	The device is not in a configuration in which you can change the power state. The device must be idle or in sweep mode to make power state changes.

smSetAttenuator

```
SmStatus smSetAttenuator(int device, int atten);

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

Parameters

`atten`	Attenuation value between [0,6] or -1

Description

Set the receiver attenuation. For more information, see Reference Level and Sensitivity. 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.

Return Values

`smInvalidParameterErr`	One or more parameters is not in the valid input range.

smSetRefLevel

```
SmStatus smSetRefLevel(int device, double refLevel);

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

Parameters

`refLevel`	Set the reference level of the receiver in dBm.

Description

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. The new reference level will not take effect until the device is reconfigured.

Return Values

`smSettingsClamped`	The reference level was clamped to a valid range.

smSetPreselector

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

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

Parameters

`enabled`	Specify whether to enable the SM200 preselector.

Description

Configure the SM200 preselector. This setting controls the preselector for all measurement modes. This setting will not take effect until the device is reconfigured.

Return Values

`smInvalidParameterErr`	The `enabled` parameter does not match the possible input values.

smSetGPIOState

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

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

Parameters

`lowerState`	Specify the direction of the lower 4 bits of the GPIO.
`upperState`	Specify the direction of the upper 4 bits of the GPIO.

Description

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

Return Values

`smInvalidParameterErr`	One or more invalid `SmGPIOState` parameters were provided.
`smInvalidConfigurationErr`	The device must either be in idle mode or in sweep mode (with no active sweeps).

smWriteGPIOImm

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

Parameters

`data` Data to write to the GPIO.

Description

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.

Return Values

`smInvalidConfigurationErr` The device is actively making measurements and cannot write to the GPIO.

smReadGPIOImm

```
SmStatus smReadGPIOImm(int device uint8_t *data);
```

Parameters

`data` Pointer to byte.

Description

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.

Return Values

`smInvalidConfigurationErr` The device is actively making measurements and cannot read the GPIO.

smWriteSPI

```
SmStatus smWriteAPI(int device, uint32_t data, int byteCount);
```

Parameters

`data` Up to 4 bytes of data to transfer.

`byteCount` Number of bytes to transfer.

Description

Output up to 4 bytes of data on the SPI data pins of the SM200.

Return Values

`smInvalidParameterErr` Byte count is not between [1,4].

`smInvalidConfigurationErr` Device is either streaming (I/Q/Real-time) or has active sweeps.

smSetGPIOSweepDisabled

```
SmStatus smSetGPIOSweepDisabled(int device);
```

Description

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

smSetGPIOSweep

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

Parameters

`steps`	Array of `SmGPIOStep` structs. The array must be `stepCount` in length.
`stepCount`	The number of steps to configure.

Description

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.

Return Values

`smInvalidParameterErr`	Invalid `stepCount` provided.

smSetGPIOSwitchingDisabled

```
SmStatus smSetGPIOSwitchingDisabled(int device);
```

Description

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.

smSetGPIOSwitching

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

Parameters

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

Description

Configures the GPIO switching functionality.

Return Values

`smInvalidParameterErr`	Invalid gpioSteps.

smSettingClamped	The count value was clamped to a value within the range of available counts.

smSetExternalReference

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

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

Parameters

enabled	When true, the 10MHz out port on the SM200 is enabled.

Description

The function allows you to enable the 10MHz out port on the SM200. If enabled, the current reference being used by the SM200 (as specified by smSetReference) will be output on the 10MHz out port.

Return Values

smInvalidConfigurationErr	The device is not currently in an idle state.

smSetReference

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

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

Parameters

reference	Specify the 10MHz reference for the SM200.

Description

Update the receiver to use either the internal time base reference or use a 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.

Return Values

smInvalidConfigurationErr	The device is not currently in an idle state. Call smAbort and try again.
smInvalidParameterErr	The reference parameter does not match the possible SmReference enum value.

smSetGPSTimebaseUpdate

```
SmStatus smSetGPSTimebaseUpdate(int device, SmBool enabled);

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

Parameters

enabled	Set to smTrue to enable automatic GPS timebase updates.

Description

This function must be called when the device is in an idle state. The ideal time to call this function is exactly one time after successfully opening a device. See Automatic GPS Timebase Discipline for more information.

Return Values

`smInvalidConfigurationErr`	The device is not in an idle state.

smGetGPSHoldoverInfo

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

Parameters

`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.
`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.

Description

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

smGetGPSState

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

Parameters

`GPSState`	Pointer to `SmGPSState` variable.

Description

Determine the locking status of the GPS. See the GPS section for more information.

smSetSweep***

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

SmStatus smSetSweepCenterSpan(int device, double centerFreqHz, double
spanHz);

SmStatus smSetSweepStartStop(int device, double startFreqHz, double
stopFreqHz);

SmStatus smSetSweepCoupling(int device, double rbw, double vbw, double
sweepTime);

SmStatus smSetSweepDetector(int device, SmDetector, SmVideoUnits videoUnits);

SmStatus smSetSweepScale(int device, SmScale scale);

SmStatus smSetSweepWindow(int device, SmWindowType window);

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

Parameters

`sweepSpeed`	Specify which device acquisition speed to use (if applicable). Auto prioritizes the fast speed possible, while normal prioritizes accuracy.
`centerFreqHz`	Specify the center frequency in Hz of the sweep.
`spanHz`	Specify the span in Hz of the sweep.
`startFreqHz`	Specify the start frequency of the sweep in Hz.
`stopFreqHz`	Specify the stop frequency of the sweep in Hz.
`rbw`	Resolution bandwidth in Hz.
`vbw`	Video bandwidth in Hz. Cannot be greater than rbw.
`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.
`detector`	Specify the detector setting of the sweep.
`videoUnits`	Specify the video processing units as either logarithmic, voltage, power, or sample.
`scale`	Specify the units of the returned sweep. Available units are either dBm or mV.
`window`	Specify the FFT window function.
`spurRejectEnabled`	Enable/disable the software image rejection algorithm. See Software Image Rejection for more information.

Description

Set of function which configure the sweep measurement mode of the receiver. These settings do not take effect until the device is reconfigured for sweep measurement mode.

Return Values

`smInvalidParameterErr`	One or more settings parameters are invalid. (i.e. Invalid enum value)
`smSettingClamped`	One or more parameter was clamped to a valid range.

smSetRealTime***

```
SmStatus smSetRealTimeCenterSpan(int device, double center, double span);

SmStatus smSetRealTimeRBW(int device, double rbw);

SmStatus smSetRealTimeDetector(int device, SmDetector detector);

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

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

Parameters

`center`	Specify the center frequency of the real-time band in Hz.
`span`	Specify the span of the real-time band in Hz.
`rbw`	Resolution bandwidth in Hz.
`detector`	Specify the detector setting of the sweep.
`scale`	Specify the units of the returned sweep. Available units are either dBm or mV.

`frameRef`	Sets the reference level of the real-time frame. (The amplitude of the highest pixel in the frame)
`frameScale`	Specify the height of the frame in dB. A common value is 100dB.
`window`	Specify the FFT window function.

Description

Set of functions which configure the receiver's real-time measurement mode. These settings do not take effect until the device is reconfigured.

Return Values

`smInvalidParameterErr`	One or more settings parameters are invalid. (i.e. Invalid enum value)
`smSettingClamped`	One or more parameter was clamped to a valid range.

smSetIQ***

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

SmStatus smSetIQDataType(int device, SmDataType iqDataType);

SmStatus smSetIQCenterFreq(int device, double centerFreqHz);

SmStatus smSetIQSampleRate(int device, int decimation);

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

SmStatus smSetIQExtTriggerEdge(int device, SmTriggerEdge edge);

SmStatus smSetIQQueueSize(int device, float ms);
```

Parameters

`sampleRate`	Specify the base sample rate of the I/Q acquisition. See Appendix: I/Q Samples Rates for more information.
`iqDataType`	Specify the I/Q data type returned. Can choose between 32-bit complex floats and 16-bit complex shorts. See Appendix: I/Q Data Types for more information.
`centerFreqHz`	Specify the center frequency of the I/Q acquisition in Hz.
`decimation`	Specify the decimation of the I/Q data as a power of two.
`enableSoftwareFilter`	Set to true to enable the software filter (SM200A/B only). This value is ignored for the SM200C (as the filter is always enabled).
`bandwidth`	Specify the bandwidth of the software filter in Hz.
`edge`	Trigger enumeration value. Specifies the edge on which the SM200 will detect triggers on the external trigger input port.
`ms`	Milliseconds, see description.

Description

These functions configure the receiver's IQ measurement mode. These settings do not take effect until the device is reconfigured.

`smSetIQQueueSize` controls the size of the queue of data that is being actively requested by the API to the SM200. For example, a queue size of 20ms means the API keeps up to 20ms of data requests active to the SM200. 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 small 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 Appendix: Linux notes. ms will be clamped to multiples of 2.62ms between 2 * 2.62ms and 16 * 2.62ms.

smSetSegIQ***

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

SmStatus smSetSegIQCenterFreq(int device, double centerFreqHz);

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

SmStatus smSetSegIQExtTrigger(int device, SmTriggerEdge extTriggerEdge);

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

SmStatus smSetSegIQSegmentCount(int device, int segmentCount);

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

Parameters

dataType	Specify whether the API will return 32-bit complex floats or 16-bit complex shorts.
centerFreqHz	Center frequency of the segmented I/Q capture.
triggerLevel	Video trigger level for video triggered captures in dBm.
triggerEdge	Video trigger edge type.
extTriggerEdge	External trigger edge type.
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.
frequencies	Array of *count* frequencies, specified as Hz, specifying the frequency points of the FMT mask.
ampls	Array of *count* amplitudes, specified as dBm, specifying the amplitude threshold limits of the FMT mask.
count	Number of FMT points in the *frequencies* and *ampls* arrays.
segmentCount	Specify the number of segments in the segmented I/Q capture. Specify this before setting each individual segment.
segment	Specify the segment to modify. Must be greater than or equal to zero, and less than the segmentCount.
triggerType	Specify the trigger type of the trigger segment.

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.
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.
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.

Description

(SM200B only)

These functions configure the SM200B segmented I/Q capture parameters. These functions must be called before configuring the device for segmented I/Q measurements. See Segmented I/Q Acquisitions for more information.

smSetAudio***

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

SmStatus smSetAudioType(int device, SmAudioType audioType);

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

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

Parameters

`centerFreqHz`	Center frequency in Hz.
`audioType`	Audio demodulation selection.
`ifBandwidth`	IF Bandwidth (RBW) in Hz.
`audioLpf`	Audio low pass frequency in Hz.
`audioHpf`	Audio high pass frequency in Hz.
`deemphasis`	FM deemphasis in us.

Description

Set of functions which configure the audio demodulation functionality of the API. These functions do not take effect until the receiver is reconfigured.

Return Values

`smSettingClamped`	One or more parameters was clamped to a valid range.
`smInvalidParameterErr`	One or more parameters was invalid. (I.E. invalid enum value)

smSetVrtPacketSize

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

Parameters

`samplesPerPkt`	Number of I/Q samples in each VRT data packet.

Description

This function specifies the number of I/Q samples to be obtained using smGetIQ and packed into each VRT data packet.

Return Values

smSetVrtStreamID

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

Parameters

`sid`	New stream identifier for the VRT information stream.

Description

This function sets the stream identifier, a value which is used to identify each VRT packet with the device.

Return Values

smConfigure

```
SmStatus smConfigure(int device, SmMode mode);
```

Parameters

`mode`	Specifies the mode of operation the API will be in if the function returns successfully.

Description

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.

Return Values

`smInvalidParameterErr`	The mode parameter does not match a valid `SmMode` value. If this error is returned, no change in device state takes place.

smGetCurrentMode

```
smStatus smGetCurrentMode(int device, SmMode *mode);
```

Parameters

`mode`	Pointer to `SmMode` variable.

Description

Retrieve the current device configuration.

smAbort

```
SmStatus smAbort(int device);
```

Description

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.

smGetSweepParameters

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

Parameters

`actualRBW`	Pointer to double. The RBW used internally in Hz. Can be `NULL`.
`actualVBW`	Pointer to double. The VBW used internally in Hz. Can be `NULL`.
`sweepSize`	Pointer to double. The length of the sweep (the number of frequency bins). Can be `NULL`.
`firstBinFreq`	Pointer to double. Frequency in Hz of the first bin in the sweep. Can be `NULL`.
`binSize`	Pointer to double. Frequency spacing in Hz, between each frequency bin in the sweep. Can be `NULL`.

Description

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.

Return Values

`smInvalidConfigurationErr`	The current measurement mode is not set to sweep.

smGetRealTimeParameters

```
SmStatus smGetRealTimeParameters(int device, double *actualRBW, int
*sweepSize, double *actualStartFreq, double *binSize, int *frameWidth, int
*frameHeight, double *poi);
```

Parameters

`actualRBW`	Pointer to double. The RBW used internally in Hz. Can be `NULL`.
`sweepSize`	Pointer to double. The length of the sweep, in frequency bins. Can be `NULL`.
`firstBinFreq`	Pointer to double. Frequency in Hz of the first bin in the sweep. Can be `NULL`.
`binSize`	Pointer to double. Frequency spacing in Hz, between each frequency bin in the sweep. Can be `NULL`.
`frameWidth`	Pointer to double. The width of the real-time frame. Can be `NULL`.
`frameHeight`	Pointer to double. The height of the real-time frame. Can be `NULL`.
`poi`	Pointer to double. 100% probability of intercept of a signal given the current configuration. Can be `NULL`.

Description

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.

Return Values

`smInvalidConfigurationErr`	The current measurement mode is not set to real-time.

smGetIQParameters

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

Parameters

`bandwidth`	Pointer to double. The bandwidth of the configure I/Q data stream. Can be `NULL`.
`sampleRate`	Pointer to double. The resulting sample rate of the receiver given the configuration parameters. Can be `NULL`.

Description

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.

Return Values

`smInvalidConfigurationErr`	The current measurement mode is not set to I/Q.

smGetIQCorrection

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

Parameters

`scale`	Pointer to 32-bit floating point value. If the function returns successfully, the value pointed to by scale will contain the 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 Appendix: I/Q Data Types. Cannot be null.

Description

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.

Return Values

`smInvalidConfigurationErr`	The device is currently not configured for I/Q streaming or segmented I/Q captures.

smSegIQGetMaxCaptures

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

Parameters

`maxCaptures`	Pointer to 32-bit integer. The maximum number of queued segmented acquisitions that can be active at any time.

Description

(SM200B only)

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 Segmented I/Q Acquisitions for more information.

Return Values

`smInvalidConfigurationErr`	Device is not configured for segmented I/Q captures.

smVrtContextPktSize

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

Parameters

`wordCount`	Pointer to unsigned 32-bit integer. The number of words in a VRT context packet. Can be `NULL`.

Description

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

Return Values

`smInvalidConfigurationErr`	The current measurement mode is not set to VRT.

smGetVrtPacketSize

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

Parameters

`SamplesPerPkt`	Pointer to unsigned 16-bit integer. The number of I/Q samples in a VRT data packet. Can be `NULL`.
`wordCount`	Pointer to unsigned 32-bit integer. The number of words in a VRT data packet. Can be `NULL`.

Description

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`.

Return Values

`smInvalidConfigurationErr`	The current measurement mode is not set to VRT.

smGetSweep

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

Parameters

`sweepMin`	Pointer to sweep min array. Can be `NULL`.
`sweepMax`	Pointer to sweep max array. Can be `NULL`.
`nsSinceEpoch`	Pointer to 64-bit integer. Represents nanoseconds since epoch. Can be `NULL`.

Description

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.

Return Values

`smDeviceNotOpenErr`	Device specified is not open.

smStartSweep

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

Parameters

`pos`	Sweep queue position.

Description

Starts a sweep at the queue pos. If successful, this function returns immediately.

Return Values

`smInvalidSweepPosition`	Invalid queue pos. Must be between [0, 16). May also receive this error if a sweep has already been started, but not finished at this queue position.

smFinishSweep

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

Parameters

`pos`	Sweep queue position.
`sweepMin`	Pointer to user allocated space for the min sweep. Can be set to `NULL`.
`sweepMax`	Pointer to user allocated space for the max sweep. Can be set to `NULL`.
`nsSinceEpoch`	Pointer to 64-bit integer. Represents nanoseconds since epoch Can be set to `NULL`.

Description

Finishes the sweep specified at the queue pos. This function blocks until the sweep is complete.

Return Values

smInvalidSweepPosition	Invalid queue pos. Must be between [0,16]. May also receive this error if a sweep has not been started at this queue position.

smGetRealTimeFrame

```
SmStatus smGetRealTimeFrame(int device, int frameWidth, int frameHeight,
float *frame, float *alphaFrame float *sweepMin, float *sweepMax, int
*frameCount, int64_t *nsSinceEpoch);
```

Parameters

frameWidth	The width of the 2D frame. This value should match the frame width returned from smGetRealTimeParameters.
frameHeight	The height of the 2D frame. This value should match the frame height returned from smGetRealTimeParameters.
frame	Pointer to memory for the real-time frame. Must be (frameWidth * frameHeight) floats in length. Can be NULL.
alphaFrame	Pointer to memory for the real-time alpha frame. Must be (frameWidth * frameHeight) floats long. Can be NULL.
sweepMin	Pointer to memory for the min sweep. Can be set to NULL.
sweepMax	Pointer to memory for the max sweep. Can be set to NULL.
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.
nsSinceEpoch	Pointer to int64_t. Nanoseconds since epoch for the returned frame. Can be NULL. 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.

Description

Retrieve a single real-time frame. See Real-Time Spectrum Analysis for more information.

Return Values

smInvalidConfigurationErr	The current measurement mode is not set to real-time.

smGetIQ

```
SmStatus smGetIQ(int device, void *iqBuf, int iqBufSize, double *triggers, int
triggerBufSize, int64_t *nsSinceEpoch, SmBool purge, int *sampleLoss, int
*samplesRemaining);
```

Parameters

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 Appendix: I/Q Data Types.
iqBufSize	Specifies the number of I/Q samples to be retrieved from the smGetIQ function. Must be greater than zero.

triggers	Pointer to user allocated array of doubles. The buffer must be at least triggerBufSize number of doubles long. The pointer can also be NULL to indicate you do not wish to receive external trigger information.
triggerBufSize	Specify the maximum number of external trigger events to receive. Once the maximum number of external triggers are recorded to the user buffer, any remaining triggers that occur within the collected I/Q block will be lost.
nsSinceEpoch	Nanoseconds since epoch. The time of the first I/Q sample returned. Can be NULL.
purge	When set to smTrue, any buffered I/Q data in the API is purged before returned beginning the I/Q block acquisition. See the section on Streaming I/Q Data for more detailed information.
sampleLoss	Set by the API when a sample loss condition occurs. If enough I/Q data has accumulated in the 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.
samplesRemaining	Set by the API, returns the number of samples remaining in the I/Q circular buffer. Can be NULL.

Description

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

Return Values

smInvalidParameterErr	iqBufSize was less than 1.
smInvalidConfigurationErr	Device specified is not configured in I/Q measurement mode.

smSegIQCapture***

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

SmStatus smSegIQCaptureWait(int device, int capture);

SmStatus smSegIQCaptureWaitAsync(int device, int capture, SmBool *completed);

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

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

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

SmStatus smSegIQCaptureFinish(int device, int capture);

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

Parameters

capture	Capture index. Must be between [0, maxCaptures-1].
segment	Segment index within capture. Must be between [0, segmentCount-1].

`completed`	smTrue when the capture specified is completed.
`timedOut`	smTrue when the segment specified was not triggered according to users configuration and a timeout occurred.
`nsSinceEpoch`	Nanosecond 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 SM200 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 SM200 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.
`iq`	User provide 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`.
`offset`	Offset into segment to retrieve.
`len`	Number of samples after the offset to retrieve.

Description

These functions are for the SM200B only.

These functions are used to perform segmented I/Q acquisitions. These functions can only be called after successfully configuring the device for segmented I/Q (via smConfigure).

CaptureStart initializes a capture with the given capture index. If no other captures are active, this capture begins immediately, otherwise all other captures are completed before beginning. To determine whether a capture is complete and ready to retrieve or has timed out, use the Wait functions. CaptureWait is a blocking function, and CaptureWaitAsync allows you to quickly query the capture status without blocking. Once the capture has completed, query the timeout status to determine for each segment in the capture, if the segment has timed out (for triggered acquisitions) or if it has completed successfully. If it has completed successfully, use the CaptureTime and CaptureRead commands to retrieve the timing and I/Q data. CaptureFinish frees the specific capture so that it can be started again.

CaptureFull is a convenience function for captures that have only 1 segment. They perform the full Start/Wait/Time/Read/Finish sequence for a capture.

See Segmented I/Q Acquisitions for more information.

Return Values

smSegIQLTEResample

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

Parameters

`input`	Pointer to input array. Input array should be interleaved I/Q samples retrieved from the segmented I/Q capture functions.
`inputLen`	Number of complex I/Q samples in input.
`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.

`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.
`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.

Description

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.

smGetAudio

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

Parameters

`audio`	Pointer to array of 1000 32-bit floats.

Description

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.

Return Values

`smInvalidConfigurationErr`	Device is not configured for audio demodulation.

smGetVrtContextPkt

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

Parameters

`words`	Pointer to user allocated buffer of unsigned integers. The buffer size must be at least wordCount 32-bit unsigned integers. Cannot be `NULL`.
`wordCount`	Set by the API, returns the number of words written to the `words` buffer. Can be `NULL`.

Description

Retrieve one VRT context packet.

Return Values

`smInvalidConfigurationErr`	Device is not configured in VRT mode.

smGetVrtPackets

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

Parameters

words	Pointer to user allocated buffer of unsigned integers. The buffer size must be at least wordCount 32-bit unsigned integers. Cannot be NULL.
wordCount	Set by the API, returns the number of words written to the words buffer. Can be NULL.
packetCount	Specifies the number of VRT data packets to be retrieved, each packed using the smGetIQ function. Must be greater than zero.
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.

Description

Retrieve one block of VRT data packets as specified by the user. This function blocks until the data requested is available.

Return Values

smInvalidConfigurationErr	Device is not configured in VRT mode.

smGetGPSInfo

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

Parameters

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.
updated	Pointer to boolean parameter. 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.
timeSec	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.
latitude	Latitude in decimal degrees. If the GPS is not locked, this value will be set to zero. Can be NULL.
longitude	Longitude in decimal degrees. If the GPS is not locked, this value will be set to zero. Can be NULL.
altitude	Altitude in meters. If the GPS is not locked, this value will be set to zero. Can be NULL.
nmea	Pointer to user allocated array of char. The length of this array is specified by the nmeaLen parameter. Can be set to NULL.
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.

Description

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.

Return Values

`smGpsNotLockedErr`	The GPS does not have a lock. No information is available, and this function will return without setting any parameters.

smWriteToGPS

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

Parameters

`mem`	The message to send to the GPS. The memory pointed to by this pointer should be contiguous.
`len`	The length of the message in bytes.

Description

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 present on the SM200 spectrum analyzers.

Messages provided are rounded/padded up to the next multiple of 4 bytes. The padded bytes are set to zero.

Return Values

`smInvalidConfigurationErr` The device does not have GPS write capability. Update the device firmware. This error can also be returned if the device is not idle when this function is called. The device cannot be actively making measurements when this function is called. If the device state is unknown, call `smAbort` prior to calling this function to ensure an idle state.

`smInvalidParameterErr`	Invalid len provided.

smSetFanThreshold

```
SmStatus smSetFanThreshold(int device, int temp);

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

Parameters

`temp`	Temperature in Celsius.

Description

Specify the temperature at which the SM200 fan should be enabled. This function has no effect if the SM200 does not have the fan assembly installed. The available temperature range is between [10-90] degrees.

This function must be called when the device is idle (no measurement mode active).

Return Values

`smInvalidConfigurationErr`	The device is not currently idle. Call `smAbort` before calling this function.
`smSettingClamped`	The temperature provided was clamped to an acceptable range.

smGetCalDate

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

Parameters

`lastCalDate`	Pointer to 64-bit unsigned integer.

Description

This function returns the calibration date as the seconds since epoch.

smGetAPIVersion

```
const char* smGetAPIVersion();
```

Return Values

`const char*`	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”

smGetErrorString

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

Parameters

`status`	A valid `SmStatus` enumeration.

Description

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

Return Values

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

Appendix

Code Examples

All code examples are distributed in the API download folder.

Linux Notes

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.

Multiple USB Devices

There are limitations that apply when attempting to use multiple devices on Linux. The maximum amount of memory that can be allocated for USB transfers on Linux is 16MB. A single SM200 can stay within this limitation, but two devices will exceed this limitation and can cause the API to crash when you do. The USB allocation limit 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.

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.

An alternative way to work around this for I/Q streaming specifically is to use the `smSetIQUSBQueueSize` function to set a smaller queue size. The memory used for I/Q streaming is roughly

*MB used for I/Q streaming = queue size in seconds * 200*

Example, queue size of 20ms

*MB used = 0.020 * 200 = 4MB.*

The default queue size is ~41.2ms, so two devices just exceed the 16MB allocation limit for I/Q streaming.

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 SM200C can keep in flight. While I/Q streaming, the SM200C 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.

Other Programming Languages

The SM200 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. Some examples of calling the SM200 API in these other languages are included in the code examples folder.

The SM200 API consists of several enumerated(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.

Real-Time RBW Restrictions

The table below outlines the RBW limitations in place in real-time mode.

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

I/Q Acquisition

I/Q Sample Rates

The table below outlines the available I/Q sample rates and corresponding decimations for both the USB and networked SM200s. See the software filter limitations in the following section for more information about filtering and bandwidth.

Decimation	Native Rate (SM200A/B) MS/s	LTE Rate* (SM200A/B) MS/s	Native Rate (SM200C) MS/s	LTE Rate (SM200C) 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.

I/Q Data Types

Data is returned from the `smGetIQ` function 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.

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.
    a. float re32f = ((float)re16s / 32768.0);
    b. float im32f = ((float)im16s / 32768.0);
        i. 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`.
    a. re32f *= correction;
    b. im32f *= correction;
3) Calculate power
    a. Sample Power (dBm) = 10.0 * log10(re32f*re32f + im32f*im32f);

I/Q Filtering and Bandwidth Limitations (SM200A/B only)

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

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

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

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.

Window Functions

Below are the window functions used in the SM200 API. The API uses zero-padding to achieve the requested RBW so the noise bandwidth in this table should not be directly used.

Type	Noise Bandwidth (bins)	Notes
Flat-Top	3.77	SRS flattop
Nuttall	2.02	
Kaiser	1.79	α = 3
Blackman	1.73	α = 0.16
Chebyshev	1.94	α = 5
Hamming	1.36	α = 0.54, β = 0.46
Gaussian6dB	2.64	σ = 0.1

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.

Software Spur Rejection

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

When enabled, the SM200 will sweep the frequency range twice using different LO and IF configurations. The two sweeps can be used to determine and eliminate spurious and mixer products generated by the SM200.

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.