Overview

This manual is a reference for the Signal Hound SM200A spectrum analyzer programming interface (API). The API provides a low-level set of C routines for interfacing the SM200A. The API is C ABI compatible making is possible to be interfaced from most programming languages. See the code examples folder to get started right away.

Contact Information

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

For sales, email sales@signalhound.com.

Build/Version Notes

Version 1.0.0 – Initial release

PC Requirements and Setup

Windows Development Requirements

- Windows 7/8/10
- (C/C++ 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

PC and Other Requirements

- SM200A
- USB 3.0 connectivity provided through native USB 3.0. Native USB 3.0 is provided through 3rd generation and later Intel CPUs. 3rd and 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, 3rd 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 and initializing a device through the API is performed 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 SM200 devices connected to the PC via the `smGetDeviceList` function.

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 IQ 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, IQ 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, 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 (TBD)

Recalibration is performed once each time the device is reconfigured. For instance, When the device is configured for IQ 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 IQ 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. Current operating temperatures can be measured with the `smGetDeviceDiagnostics` function.

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 both 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.

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 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 code examples.

Sweep Speed

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

The SM200A 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 operate in a slow sweep. 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 SM200A 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 SM200A steps the LO in 156.25MHz steps to accomplish up to a 1THz sweep speed.

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 SM200A 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 usually around 30ms.

Real-Time Frame

Real-time spectrum analysis provides two types of measurements, the sweep and the frame. The frame is a 2-dimensional grid representing frequency on the x-axis and amplitude levels on the y-axis.

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

The 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).

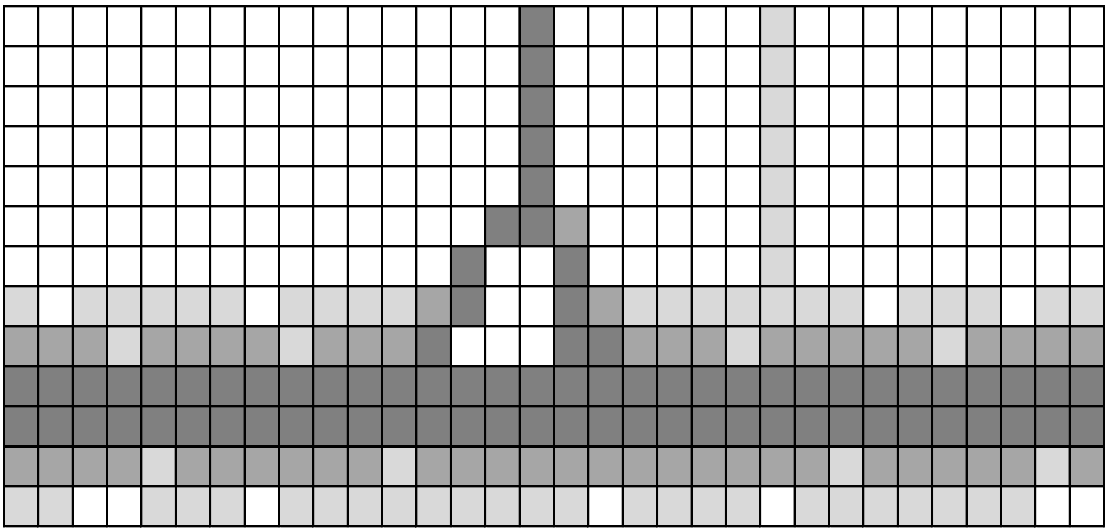

Figure 1: An example of a frame plotted as a gray scale image, mapping the density values between [0.0,1.0] to gray scale values between [0,255]. The frame shows a persistent CW signal near the center frequency and a short-lived CW signal.

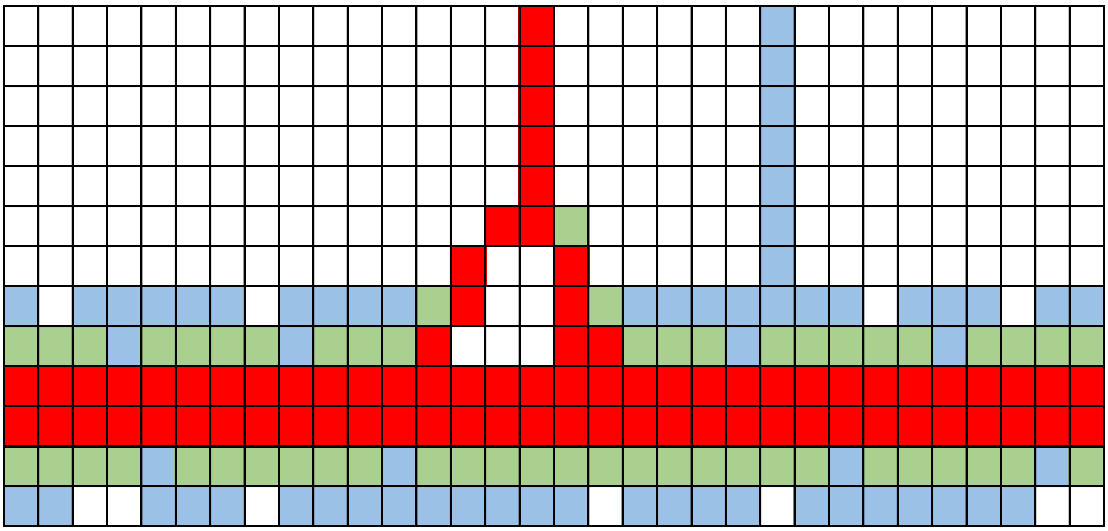

Figure 2: The same frame above as is plotted in Spike, where density values are mapped onto a color spectrum.

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 IQ Data

The API provides methods for acquiring up to 50MS/s streaming IQ data. IQ data is provided as interleaved 32-bit floating IQ samples scaled to mW. The IQ samples are corrected for amplitude flatness, IQ phase and amplitude imbalance, and phase flatness. The IQ data stream can be tuned to any frequency within the SM200 frequency range.

Sample Rate, Decimation, and Bandwidth

The IQ data stream can be decimated by powers of two down to 12.207kS/s (decimation of 4096). An optional software filter can be enabled for decimations between 1 and 8. Decimations up to 8 are performed on the SM200 FPGA using half band filters. Higher decimations are performed on the PC.

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 IQ bandwidth. Disabling the software filter will reduce CPU load of the IQ data stream at the cost of this aliasing.

For decimations above 8, the software filter is always enabled and the bandwidth of the filter is selectable. 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

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

Wideband IQ Acquisitions

The SM200A can be configured to perform 500MS/s short burst acquisitions up to 32k samples.

Reference Level and Sensitivity

There are two ways to set the sensitivity of the receiver, through the RF configuration routine `smSetRFConfig`, or through the reference level. The RF configuration routine allows full control over the preamp and attenuator settings of the receiver. When either the preamp or attenuator setting is set to auto, the reference level parameter takes over. The preamp and attenuator settings are 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 `smIsGPSLocked` 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, IQ 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.

Thread Safety

The SM200 API is not thread safe. A multi-threaded application is free to call the API from any number of threads as long as the function calls are synchronized. Not synchronizing your function calls will lead to undefined behavior.

Multiple Devices and Multiple Processes

The API is capable of managing multiple devices within one process. If 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 process 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.

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 succeded 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 openDevice() 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.

smGetDeviceList

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

Parameters

`serials`	Pointer to an array of integers.
`deviceCount`	Pointer to integer. The integer value should indicate the size of the serials array. If the function returns successfully, `deviceCount` will be set to the number devices found on the system. `deviceCount` will not exceed the initial size passed to the function.

Description

This function is used to retrieve the serial number of all unopened SM200 devices connected to the PC. The maximum number of devices to be returned is 8. 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.

Return Values

`smNullPtrErr`	One or more required pointer parameters was `NULL`.

smOpenDevice

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

Parameters

`device`	Pointer to integer.

Description

Claim the first unopened 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.

Return Values

`smDeviceNotFoundErr`	Unable to find/open an SM200 receiver.

smOpenDeviceBySerial

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

Parameters

`device`	Pointer to integer.
`serialNumber`	Serial number of the device you wish to open.

Description

This function is similar to `smOpenDevice` except it allows you to specify the device you wish to open. This function is often used in conjunction with `smGetDeviceList` when managing several SM200 devices on one PC.

Return Values

`smNullPtrErr`	One or more required pointer parameters was `NULL`.
`smDeviceNotFoundErr`	The device specified could not be found.

smCloseDevice

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

Parameters

`device`	Device handle.

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.

Return Values

`smInvalidDeviceErr`	The device handle specified does not point to an open device.

smPreset

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

Parameters

Description

Return Values

smGetDeviceInfo

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

Parameters

`device`	Device handle.
`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`.
`firmwareVersion`	Pointer to integer. If this function returns successfully, the integer pointed to will contain the specified devices firmware version. Can be `NULL`.

Description

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

Return Values

`smInvalidDeviceErr`	The device handle specified does not point to a valid device.

smGetDeviceDiagnostics

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

Parameters

device	Device handle.
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.

Description

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

Return Values

smInvalidDeviceErr	The device handle specified does not point to a valid device.

smSetRFConfig

```
SmStatus smSetRFConfig(int device, SmRFConfig config);
```

Parameters

device	Device handle.
config	RF configuration struct. For more information, see SmRFConfig.

Description

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

Return Values

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

smGetRFConfig

```
SmStatus smGetRFConfig(int device, SmRFConfig *config);
```

Parameters

device	Device handle.
config	Pointer to SmRFConfig struct.

Description

Retrieve the receiver sensitivity parameters.

Return Values

smNoError	No error.
smNulltPtrErr	One or more required pointer parameters is NULL.

smSetRefLevel

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

Parameters

`refLevel`	Set the reference level of the receiver.

Description

The reference level controls the sensitivity of the receiver by setting the attenuation and gain of the receiver to optimize measurements for signals at or below the reference level. This function will set the gain and attenuation settings set in `smSetRFConfig` to auto. 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.

smGetRefLevel

SmStatus smGetRefLevel(int device, double *refLevel);

Parameters

`refLevel`	Pointer to double.

Description

If this function returns successfully, `refLevel` will contain the current set reference level of the receiver.

Return Values

`smNullPtrErr`	One more required pointer parameters is NULL.

smSetPreselector

```
SmStatus smSetPreselector(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.

smGetPreselector

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

Parameters

`enabled`	Pointer to SmBool data type.

Description

Retrieve the last configured preselector state.

Return Values

`smNullPtrErr`	One or more required pointer parameters is NULL.

smConfigGpio

smWriteGpioImm

smReadGpioEvents

smSetGpioSweep

smWriteSPI

smSetExternalReference

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

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

Parameters

`device`	Device handle
`enabled`	When true, the 10MHz out port on the SM200A is enabled.

Description

The function allows you to enable the 10MHz out port on the SM200A. If enabled, the current reference being used by the SM200A (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

`device`	Device handle
`reference`	Specify the 10MHz reference for the SM200A.

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

`smNoError`	No error.
`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.

smIsGPSLocked

```
SmStatus smIsGPSLocked(int device, SmBool *isLocked);
```

Parameters

`device`	Device handle.
`isLocked`	Pointer to boolean value.

Description

Returns true if the GPS is currently locked. Use this function to determine if GPS location, NMEA messages, and time stamping are available.

Return Values

`smNoError`	No Error.
`smDeviceNotOpenErr`	Device specified is not open.
`smNullPtrErr`	One or more required pointer parameters are NULL.

smSetSweep***

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

SmStatus smSetSweepCenterSpan(int device, double center, double span);

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.
`center`	Specify the center frequency in Hz of the sweep.
`span`	Specify the span in Hz of the sweep.
`rbw`	Resolution bandwidth in Hz.

`vbw`	Video bandwidth in Hz. Cannot be greater than rbw.
`detector`	Specify the detector setting of the sweep.
`scale`	Specify the units of the returned sweep. Available units are either dBm or mV.
`videoUnits`	Specify the video processing units as either logarithmic, voltage, power, or sample.
`window`	Specify the FFT window function.
`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.

Description

Set of function which configure the sweep measurement mode of the receiver. 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.

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 smSetIQCaptureType(int device, SmIQCaptureType captureType);

SmStatus smSetIQCenterFreq(int device, double center);

SmStatus smSetIQSampleRate(int device, int decimation);

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

Parameters

`captureType`	Specify whether the device operates in the 500MS/s short burst acquisition mode or 50MS/s streaming mode.
`centerFreq`	Specify the center frequency of the IQ acquisition.
`decimation`	Specify the decimation of the 50MS/s streaming mode.
`enableSoftwareFilter`	Set to true to enable the software filter.
`bandwidth`	Specify the bandwidth of the software filter.

Description

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

smConfigure

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

Parameters

`device`	Device handle
`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.

smGetMeasMode

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

Parameters

device	Device handle.
mode	Pointer to a SmMode enum type.

Description

Retrieve the active device measurement mode.

Return Values

smNullPtrErr	One or more required pointer parameters is NULL.

smAbort

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

Parameters

device	Device handle.

Description

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

Return Values

smGetSweepParameters

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

Parameters

device	Device handle.
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.
windowBandwidth	Pointer to double. Window function bandwidth. Accounts for zero-padding. 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

`device`	Device handle.
`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`.
`framerate`	Pointer to double. The number of frames to be returned per seconds by the API. Can be `NULL`.
`windowBandwidth`	Pointer to double. Window function bandwidth. Accounts for zero-padding. 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 smGetStreamSettings(int device, double *sampleRate, double
*bandwidth);
```

Parameters

`device`	Device handle.
`bandwidth`	Pointer to double. The bandwidth of the configure IQ 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 IQ measurement mode parameters for an active IQ configuration. This function is typically called after a successful device configuration to retrieve the IQ stream parameters.

Return Values

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

smGetSweep

```
SmStatus smGetSweep(int device, float *sweepMin, float *sweepMax, SmTime
*time);
```

Parameters

`device`	Device handle.
`sweepMin`	Pointer to sweep min array. Can be NULL.
`sweepMax`	Pointer to sweep max array. Can be NULL.
`time`	Pointer to time struct. Can be NULL.

Description

Perform a single sweep. Block until the sweep completes.

Return Values

`smDeviceNotOpenErr`	Device specified is not open.

smStartSweep

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

Parameters

`device`	Device handle.
`pos`	Sweep queue position.

Description

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

Return Values

smFinishSweep

```
SmStatus smFinishSweep(int device, int pos, float *sweepMin, float *sweepMax,
SmTime *time);
```

Parameters

`device`	Device handle.
`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.
`time`	Pointer to user time struct. Can be set to NULL.

Description

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

Return Values

`smNoError`	Device handle.

smGetRealTimeFrame

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

Parameters

`device`	Device handle.
`frameWidth`	The width of the 2D frame. This value should match the frame width returned in the `SmRealTimeSettings` struct.
`frameHeight`	The height of the 2D frame. This value should match the frame height returned in the `SmRealTimeSettings` struct.
`float *frame`	Pointer to memory for the real-time frame. Must be `(frameWidth * frameHeight)` floats in length.
`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.
`timeSec`	Pointer to int64_t. Seconds 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.
`timeNs`	Pointer to int64_t. Ns of given second. 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, float *iqBuf, int iqBufSize, double *triggers, int
triggerBufSize, int64_t *timeSec, int64_t *timeNs, SmBool purge, int *overRange, int
*sampleLoss, int *samplesRemaining);
```

Parameters

`device`	Device handle.
`iqBuf`	Pointer to user allocated buffer of floats. The buffer size must be at least (iqBufSize * 2) 32-bit floats. Cannot be `NULL`.

iqBufSize	Specifies the number of IQ 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 IQ block will be lost.
timeSec	Seconds since epoch. The time of the first IQ sample returned. Can be NULL.
purge	When set to smTrue, any buffered IQ data in the API is purged before returned beginning the IQ block acquisition. See the section on Streaming IQ Data for more detailed information.
overRange	Set by the API when an input overload condition is detected. A detected event does not necessarily correlate to the IQ data block returned. Once returned, the internal overRange flag is cleared. Can be NULL.
sampleLoss	Set by the API when a sample loss condition occurs. If enough IQ 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 IQ circular buffer. Can be NULL.

Description

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

Return Values

smNullPtrErr	One or more required pointer parameters was NULL.
smInvalidParameterErr	iqBufSize was less than 1.
smInvalidConfigurationErr	Device specified is not configured in IQ measurement mode.

smGetGPSInfo

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

Parameters

device	Device handle.
refresh	When set to true and the device is not in a streaming mode, the API will request the latest GPS information.
updated	Pointer to boolean parameter. Will be set to true if the GPS position and NMEA data has been updated since the last time the user called this function. An updated time stamp will set the updated parameter to true. Can be set to NULL.

timeSec	Number of seconds since epoch as reported by the GPS NMEA sentences. Last reported value by the GPS. Can be NULL.
latitude	Latitude in decimal degrees. Can be NULL.
longitude	Longitude in decimal degrees. Can be NULL.
altitude	Altitude in meters. 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 string, 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.

Return Values

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

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.

Other Programming Languages

The SM200 interface is C compatible which ensures is 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

IQ Sample Rate Table

The table below outlines the available IQ decimations and corresponding sample rates. See the software filter limitations in the next section.

Decimation	Sample Rate	Software Filter	Downsampling
1 (Minimum)	50 MS/s	Optional	Hardware
2	25 MS/s	Optional	Hardware only
4	12.5 MS/s	Optional	Hardware only
8	6.25 MS/s	Optional	Hardware only
16	(50/16) MS/s	Always enabled	Hardware/Software
N = {32, 64, …}	(50/N) MS/s	Always enabled	Hardware/Software
4096 (Maximum)	(50/4096) MS/s	Always enabled	Hardware/Software

IQ Filtering and Bandwidth Limitations

The user can enable a baseband software filter on the IQ 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 rate. 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.

When the software filter is disabled the usable bandwidth of the IQ signal is

$$Maximum\ Bandwidth = \begin{cases} 41.5\ MHz, & decimation = 1 \\ Sample\ Rate * 0.8, & decimation > 1 \end{cases}$$

When the software filter is enabled the maximum bandwidth allowed is determined by the equation

$$Maximum\ Bandwidth = \begin{cases} 41.5\ MHz, & decimation = 1 \\ Sample\ Rate * 0.768, & decimation > 1 \end{cases}$$

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, but instead, use the `windowBandwidth` returned in the `SmSweepSettings` struct.

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