BB60C and BB60A Application Programming Interface (API)

Programmers Reference Manual

BB60C and BB60A Application Programming Interface (API)
Programmers Reference Manual

1502 SE Commerce Ave, Suite 101
Battle Ground, WA
Phone 360-217-0112

October 5, 2020

Requirements, Operation, Function Definitions, Examples

Table of Contents

Overview 4

Contact Information 4

Build/Version Notes 4

What’s New in Version 4.0 4
What’s New in Version 3.0 5
What’s New in version 2.0 5
Updating from Version 1.2 or Later 5

Requirements 5

Theory of Operation 6

Opening a Device 6
Configuring the Device 6
Initiating the Device 6
Retrieve Data from the Device 7
Abort the Current Mode 7
Closing the Device 7
Calibration 7

Modes of Operation 8

Swept Analysis 8
Real-Time Analysis 9
I/Q Streaming 9
Audio Demodulation 10
Scalar Network Analysis 10

Multi-Threading 11

Multiple Devices and Inter-Process Device Management 11

API Functions 11

Common Status Codes 11
bbGetSerialNumberList 12
bbOpenDeviceBySerialNumber 12
bbOpenDevice 13
bbCloseDevice 14
bbConfigureAcquisition 14
bbConfigureCenterSpan 14
bbConfigureIQCenter 15
bbConfigureLevel 16
bbConfigureGain 16
bbConfigureSweepCoupling 17
bbConfigureProcUnits 19
bbConfigureIO 19
bbConfigureDemod 20
bbConfigureIQ 21
bbConfigureIQDataType 22
bbConfigureRealTime 22
bbConfigureRealTimeOverlap 23
bbInitiate 23

bbFetchTrace 24
bbFetchRealTimeFrame 25
bbFetchAudio 26
bbGetIQ 27
bbQueryTraceInfo 28
bbQueryRealTimeInfo 29
bbQueryRealTimePoi 29
bbQueryStreamInfo 29
bbGetIQCorrection 30
bbAbort 30
bbPreset 30
bbSelfCal 31
bbSyncCPUtoGPS 32
bbGetDeviceType 32
bbGetSerialNumber 33
bbGetFirmwareVersion 33
bbGetDeviceDiagnostics 33
bbAttachTg 34
bbIsTgAttached 34
bbConfigTgSweep 34
bbStoreTgThru 35
bbSetTg 35
bbSetTgReference 36
bbGetTgFreqAmpl 36
bbGetAPIVersion 37
bbGetErrorString 37

Appendix 38

Device Connection Errors 38
    Firmware Version 7 (BB60C) 38
Code Examples 38
Manual Gain/Attenuation 38
I/Q Data Types 39
Using a GPS Receiver to Time-Stamp Data 39
    Code Example 40
Bandwidth Tables 40

Overview

The manual is a reference for the Signal Hound BB60C/A application programming interface (API). The API provides a set of C routines used to control the Signal Hound BB60A and BB60C. The API is C ABI compatible, so it can be called from several other languages and environments such as Java, C#, Python, C++, MATLAB, and LabVIEW.

This manual will describe the requirements and knowledge needed to program to the API. If you are new to the API you should read the sections in this order: *Build/Version Notes*, *Requirements, Theory of Operation*, and *Modes of Operation*.

If you want to start programming immediately, see the code examples found in the *examples/* directory.

The Build/Version Notes details the available builds for the API and notes major changes to API versions.
The Requirements section details the physical and operational needs to use the API.
The Theory of Operation section details how to interface the device and covers every major component a program will implement when interfacing a Signal Hound broadband device.
The Modes of Operation section attempts to teach you how to use the device in each of its operational modes, from the required functions, to interpreting the data the device returns.
The API Functions section covers every function in depth. The knowledge learned in the *Theory and Modes of Operation* sections will help you navigate the API functions.

Contact Information

We are interested in your feedback and questions. Please report any issues/bugs as soon as possible. We will maintain the most up to date API on our website.

All programming and API related questions should be directed to aj@signalhound.com
All hardware/specification related questions should be directed to justin@signalhound.com

Build/Version Notes

Windows builds for x86 and x64 are available.
The Windows builds are compiled with Visual Studio 2012 and any application using this library will require distributing the VS2012 redistributable libraries.
64-bit Linux versions are available on our downloads page. The Linux API is not always current.
Build notes for the Linux API are provided in the Linux SDK found on the Signal Hound website.
See the distributed change log in the Spike installer for API version specific changes.

What's New in Version 4.0

Added support for the BB60C-2, a minor hardware revision to the BB60C. The non-deprecated functions have not changed, so integrating the new BB60C-2 devices should require no software changes other than updating the API DLL in the project.

Additionally, several deprecated functions have been removed. There are alternatives for each deprecated function.

- `bbFetchRaw`: use `bbGetIQ` instead.
- `bbConfigureWindow`: use `bbConfigureSweepCoupling` instead.
- `bbQueryTimestamp`: use `bbGetIQ` instead.

What's New in Version 3.0

Version 3.0 coincides with the release of the Spike™, Signal Hounds latest spectrum analyzer software. With this release comes the ability to open specific BB60C devices. BB60A devices lack the firmware to perform this task. See bbGetSerialNumberList and bbOpenDeviceBySerialNumber for more information.

What's New in version 2.0

Version 2.0 and greater introduces support for the BB60C as well as numerous performance improvements and flexible I/Q data streaming (see Modes of Operation: I/Q Streaming). The API can target both the BB60A and BB60C with virtually no changes to how one interfaces the API.

Updating from Version 1.2 or Later

This section contains notes of interest for users who are currently using version 1.2 of the API and who are updating their code base to use version 2.0.

- Function names and the API file names may have changed from an earlier version. This is due to making the API device agnostic. Functionally the API remains the same, so simply updating to the newer naming scheme will be all that is necessary to interface a newer API.
- Intermediate frequency (IF) streaming has been replaced with I/Q streaming but IF streaming can still be performed. See bbInitiate for more information on how to set up IF streaming.
- bbQueryDiagnostics has been deprecated and replaced with bbGetDeviceDiagnostics. This change removes unnecessary access to hardware diagnostic information specific to the BB60A.

Requirements

Windows Development Requirements

- Windows 7/8/10
- Windows C/C++ development tools/environment. Preferably Visual Studio 2008 or later. If Visual Studio 2012 is not used, then the VS2012 redistributables will need to be installed.
- The API header file. (bb_api.h)
- The API library (bb_api.lib) and dynamic library (bb_api.dll) files.
- The *ftd2xx.dll* will need to be present in the working directory or present in the system path.

Linux Development Requirements

- 64-bit Linux operating system. (Tested and developed on Ubuntu 18.04 and CentOS7)
- Modern C++ compiler (Built using g++)
- The API header file. (Included in SDK)
- The API shared library (Included in SDK)
- FTDI USB shared library (Included in SDK and available for download from manufacturer)
- libusb-1.0 shared library (Available from most package managers)
- Administrator(root) access to either run applications which use the API or install rules to allow non-root access to the device.

General Requirements

- A basic understanding of RF spectrum analysis.
- A Signal Hound BB60 spectrum analyzer.

- Dual/Quad core Intel i-series processors, preferably 3rd generation (Ivy Bridge) and later. Real-time analysis may be inadequate on hardware less performant than this. Most aspects other than real-time analysis will perform as expected with no issues on the suggested hardware.

Theory of Operation

The flow of any program interfacing a BB series device will be as follows.

1) Open a BB60 device using the bbOpen** functions.
2) Configure the device for a measurement.
3) Initiate the device and specify the mode of operation.
4) Retrieve data from the device.
5) Abort the current mode of operation.
6) Repeat steps 1-5 if desired.
7) Close the device.

- Calibration

The API provides functions for each step in this process. Each step is described in more detail below.

Opening a Device

Before attempting to open a device programmatically, it must be physically connected to a USB 3.0 port with the provided cable. Ensure the front facing LED is solid green. Once the device is connected it can be opened. The function bbOpenDevice provides this functionality. This function returns an integer handle to the device which was opened. Up to eight devices may be connected and interfaced through our API using the handle. The integer handle returned is required for every function call in the API, as it uniquely identifies which device you are interfacing.

Configuring the Device

Once the device is opened, it must be configured. The API provides several configuration routines and measurement modes. Most of this manual discusses configuring the device. In the **Modes of Operation** section, each operating mode and its relevant configuration routines are discussed. All configuration functions will modify a devices' global state. Device state is discussed more in the next section (**Initiating the Device**). The API provides configurations routines for groupings of related variables. Each configuration function is described in depth in the **API functions** section. All relevant configuration routines should be invoked prior to initialization to ensure a proper device state. Certain functions will enforce boundary conditions, and will note when either a parameter is invalid or has been clamped to the min/max possible value. Ensuring each routine configures successfully is required to ensure proper device operation. Different modes of operation will necessitate different boundary conditions. Each function description will detail these boundaries. We have also provided helpful macros in the header file to help check against these boundaries.

Initiating the Device

Each device has two states.

1) A global state set through the API configuration routines.
2) An operational/running state.

All configuration functions modify the global state which does not immediately affect the operation of the device. Once you have configured the global state to your liking, you may initiate the device into a mode of operation, in which the global state is copied into the running state. At this point, the running state is separate and not affected by future configuration function calls.

The spectrum analyzer has multiple modes of operation. The bbInitiate function is used to initialize the device and enter one of the operational modes. The device can only be in one operational mode at a time. If bbInitiate is called on a device that is already initialized, the current mode is aborted before entering the new specified mode. The operational modes are described in the **Modes of Operation** section.

Retrieve Data from the Device

Once a device has been successfully initialized you can begin retrieving data from the device. Every mode of operation returns different types and different amounts of data. The **Modes of Operation** section will help you determine how to collect data from the API for any given mode. Helper routines are also used for certain modes to determine how much data to expect from the device.

Abort the Current Mode

Aborting the operation of the device is achieved through the bbAbort function. This causes the device to cancel any pending operations and return to an idle state. Calling bbAbort explicitly is never required. If you attempt to initiate an already active device, bbAbort will be called for you. Also if you attempt to close an active device, bbAbort will be called. There are a few reasons you may wish to call bbAbort manually though.

- Certain modes combined with certain settings consume large amounts of resources such as memory and the spawning of many threads. Calling bbAbort will free those resources.
- Certain modes such as real-time spectrum analysis consume many CPU cycles, and they are always running in the background whether or not you are collecting and using the results they produce.
- Aborting an operational mode and spending more time in an idle state may help to reduce power consumption.

Closing the Device

When you are finished, you must call bbCloseDevice. This function attempts to safely close the USB 3.0 connection to the device and clean up any resources which may be allocated. A device may also be closed and opened multiple times during the execution of a program. This may be necessary if you want to change USB ports, or swap a device.

Calibration

Calibration is an important part of the device's operation. The device is temperature sensitive and it is important a device is re-calibrated when significant temperature shifts occur (+/- 2 °C). Signal Hound spectrum analyzers are streaming devices and as such cannot automatically calibrate itself without interrupting operation/communication (which may be undesirable). Therefore, we leave calibration to the programmer. The API provides two functions for assisting with live calibration, bbGetDeviceDiagnostics and bbSelfCal. bbGetDeviceDiagnostics can be used to retrieve the internal device temperature at any time after the device has been opened. If the device ever deviates from its temperature by 2 degrees Celcius or more we suggest calling bbSelfCal. Calling bbSelfCal requires the device be open and idle. After a self-calibration occurs, the global device state is undefined. It is necessary to reconfigure the device before continuing operation. One self-calibration is performed upon opening the device.

Note*:* The BB60C does not require the use of bbSelfCal for device calibration. Instead, for the BB60C, if the device deviates in temperature, simply call bbInitiate again which will re-calibrate the device at its current operating temperature.

Modes of Operation

Now that we have seen how a typical application interfaces with the API, let's examine the different modes of operation the API provides. Each mode will accept different configurations and have different boundary conditions. In the next sections you will see how to interact with the API in each mode.

For a more in-depth examination of each mode of operation (read: *theory*) refer to the Signal Hound BB60 user manual.

Swept Analysis

Swept analysis represents the most traditional form of spectrum analysis. This mode offers the largest amount of configuration options, and returns traditional frequency domain sweeps. A frequency domain sweep displays amplitude on the vertical axis and frequency on the horizontal axis.

The configuration routines which affect the sweep results are

- `bbConfigureAcquisition() – Configuring the detector and linear/log scaling`
- `bbConfigureCenterSpan() – Configuring the frequency range`
- `bbConfigureLevel() – Configuring reference level and internal attenuators`
- `bbConfigureGain() – Configuring internal amplifiers`
- `bbConfigureSweepCoupling() – Configuring RBW/VBW/Window function/sweep time`
- `bbConfigureProcUnits() – Configure VBW processing`

Once you have configured the device, you will initialize the device using the `BB_SWEEPING` flag.

This mode is driven by the programmer, causing a sweep to be collected only when the program requests one through the `bbFetchTrace()` functions. The length of the sweep is determined by a combination of resolution bandwidth, video bandwidth and sweep time.

Once the device is initialized you can determine the characteristics of the sweep you will be collecting with `bbQueryTraceInfo().` This function returns the length of the sweep, the frequency of the first bin, and the bin size (difference in frequency between any two samples). You will need to allocate two arrays of memory, representing the minimum and maximum values for each frequency bin.

Now you are ready to call `bbFetchTrace()`. This is a blocking call that does not begin collecting and processing data until it is called. Typical sweep times might range from 10ms – 100ms, but certain settings can take much more time (full spans, low RBW/VBWs).

Determining the frequency of any point returned is determined by the function below, where 'n' starts at zero for the first sample point.

$$Frequency\ of\ n'th\ sample\ point\ in\ returned\ sweep = startFreq + n * binSize$$

Real-Time Analysis

The API provides the functionality of a real-time spectrum analyzer for the full instantaneous bandwidth of the device (20MHz for the BB60A, 27MHz for the BB60C). Using FFTs at an overlapping rate of 50%, the spectrum results have no blind time (100% probability of intercept) for events as short as 4.8us at full amplitude (at 631kHz RBW). The RBW shape is restricted to the Nuttall window, and VBW is not configurable.

The configuration routines which affect the spectrum results are

- `bbConfigureAcquisition() – Configure detector and linear/logarithmic scale`
- `bbConfigureCenterSpan() – Configure center frequency and span of no more than the devices maximum instantaneous bandwidth, specified in the header macros.`
- `bbConfigureLevel() – Configure reference level and attenuators`
- `bbConfigureGain() – Specify internal amplifiers`
- `bbConfigureSweepCoupling() – Specify RBW`
- `bbConfigureRealTime() – Specify the real-time update rate and frame scale`

The number of sweep results far exceeds a program's capability to acquire, view, and process, therefore the API combines sweeps results for a user specified amount of time. It does this in two ways. One, is the API either max holds or averages the sweep results into a standard sweep.

Also, the API creates an image frame which acts as a density map for every sweep result processed during a period. Both the sweep and density map are returned at rate specified by the function *bbConfigureRealTime.*

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

For a full example of using real-time see the code examples.

I/Q Streaming

The API can be used to stream I/Q data up to 40MS/s. 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 IF flatness and RF leveling. I/Q data returned as 16-bit shorts is provided as full scale, only correcting for IF flatness and the user must apply a correction value to recover the fully amplitude corrected I/Q data.

The I/Q data can be decimated by powers of 2 up to a decimation of 8192. The API provides users a customizable bandpass filter cutoff frequency at any sample rate.

The I/Q data stream can be tuned to any frequency within the BB60 frequency range.

See the following functions for configuring and retrieving I/Q data.

- `bbConfigureIQCenter() – Set the center frequency of the I/Q data stream.`
- `bbConfigureLevel() – Set the measurement reference level.`

- `bbConfigureIO()` – Configure the BNC ports of the BB60.
- `bbConfigureIQ()` – Specify the decimation and bandwidth of the I/Q data stream.
- `bbConfigureIQDataType()` – Specify return type of I/Q data.
- `bbInitiate()` – Start streaming.
- `bbQueryStreamInfo()` – Get I/Q rate info.
- `bbGetIQCorrection()` – Get I/Q scaling for 16-bit acquisitions.
- `bbGetIQ()` – Retrieve I/Q data.

Once configured, initialize the device with the BB_STREAMING mode and the BB_STREAM_IQ flag. Data acquisition begins immediately. The API buffers ~3/4 second worth of I/Q samples in a circular buffer. Samples can be retrieved with the bbGetIQ function. If you wish to retrieve all samples, it is the responsibility of the user's application to poll the samples fast enough to prevent the APIs internal buffers from accumulating too much I/Q data. We suggest a separate polling thread and synchronized data structure (buffer) for retrieving the samples and using them in your application.

NOTE: Decimation and filtering occur on the PC and can be processor intensive on certain hardware. Please characterize the processor load.

Audio Demodulation

Audio demodulation can be achieved using `bbConfigureDemod()`, `bbFetchAudio()`, and `bbInitiate()`. See `bbConfigureDemod()` to see which types of demodulation can be performed. Settings such as gain, attenuation, reference level, and center frequency affect the underlying signal to be demodulated.

`bbConfigureDemod()` is used to specify the type of demodulation and the characteristics of the filters. Once desired settings are chosen, use `bbInitiate()` to begin streaming data. Once the device is streaming it is possible to continue to change the audio settings via `bbConfigureDemod()` as long as the updated center frequency is not +/- 8 MHz of the value specified when `bbInitiate()` was called. The center frequency is specified in `bbConfigureDemod()`.

Once the device is streaming, use `bbFetchAudio()` to retrieve 4096 audio samples for an audio sample rate of 32k.

Scalar Network Analysis

When a Signal Hound tracking generator is paired together with a BB60C or BB60A spectrum analyzer, the products can function as a scalar network analyzer to perform insertion loss measurements, or return loss measurements by adding a directional coupler. Throughout this document, this functionality will be referred to as tracking generator (or TG) sweeps.

Scalar Network Analysis can be realized by following these steps

1. Ensure a Signal Hound BB60C spectrum analyzer and tracking generator is connected to your PC.
2. Open the spectrum analyzer through normal means.
3. Associate a tracking generator to a spectrum analyzer by calling bbAttachTg. At this point, if a TG is present, it is claimed by the API and cannot be discovered again until bbCloseDevice is called.
4. Configure the device as normal, setting sweep frequencies and reference level (or manually setting gain and attenuation).

5. Configure the TG sweep with the bbConfigTgSweep function. This function configures TG sweep specific parameters.
6. Call bbInitiate with the BB_TG_SWEEP mode flag.
7. Get the sweep characteristics with bbQueryTraceInfo.
8. Connect the BB and TG device into the final test state without the DUT and perform one sweep with bbFetchTrace. After one full sweep has returned, call bbStoreTgThru with the TG_THRU_0DB flag.
9. (Optional) Configure the setup again still without the DUT but with a 20dB pad inserted into the system. Perform an additional full sweep and call bbStoreTgThru with the TG_THRU_20DB.
10. Once store through has been called, insert your DUT into the system and then you can freely call the get sweep function until you modify the configuration or settings.

If you modify the test setup or want to re-initialize the device with a new configuration, the store through must be performed again.

Multi-Threading

The BB60 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. Not synchronizing your function calls will lead to undefined behavior.

Multiple Devices and Inter-Process Device Management

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

If you wish to use the API in multiple processes it is the user's responsibility to manage a list of devices to prevent the possibility of opening a device twice from two different processes. Two processes communicating to the same device will result in undefined behavior. The two functions responsible for opening new devices are not thread safe and access to those functions must also be restricted by the programmer. One possible way to manage inter-process information like this is to use a named mutex on a Windows system.

API Functions

Common Status Codes

All API functions return the type bbStatus. This section documents some of the more common status codes. Negative error codes represent errors and are suffixed with 'Err'. When an error code is returned, the operation requested did not complete. Positive status codes are warnings and indicate the operation completed, but the user might need to take some action. For a full list of status codes, see the API header file.

bbNoError	Function returned successfully. No warnings or errors.
bbNullPtrErr	Returned when one or more required pointer parameter is null.

bbDeviceNotOpenErr	Returned when the device handle provided does not match an open or known device.
bbInvalidParameterErr	Returned when one or more parameters provided does not match the range of possible values.
bbDeviceNotConfiguredErr	Returned if the device is not properly configured for the desired action. Often occurs when the device needs to be configured for a specific measurement mode before taking an action.
bbClampedToLowerLimit	One or more parameters was clamped to a minimum lower limit.
bbClampedToUpperLimit	One or more parameters was clamped to a maximum upper limit.
bbAdjustedParameter	One ore more parameters to clamped to a minimum or maximum limit.

bbGetSerialNumberList

Get a list of available devices connected to the PC

```
bbStatus bbGetSerialNumberList(int serialNumbers[8], int *deviceCount);
```

Parameters

serialNumbers	A pointer to an array of at minimum 8 contiguous integers. It is undefined behavior if this array pointed to by *serialNumbers* is not 8 integers in length.
deviceCount	Pointer to an integer.

Description

This function returns the devices that are unopened in the current process. Up to 8 devices may be returned. The serial numbers of the unopened devices are returned for BB60Cs and a zero is returned for each BB60A. The array will be populated starting at index 0 of the provided array. The integer pointed to by *deviceCount* will equal the number of devices reported by this function upon returning.

bbOpenDeviceBySerialNumber

Open one Signal Hound device

```
bbStatus bbOpenDeviceBySerialNumber(int *device, int serialNumber);
```

Parameters

device	Pointer to an integer. If successful, the integer pointed to by device will contain a valid device number which can be used to identify a device for successive API function calls.
serialNumber	User provided serial number.

Description

The function attempts to open the device with serial number specified by the *serialNumber* parameter. Only BB60C devices can be opened by specifying the serial number. If the serial number specified is zero, the first BB60A found will be opened. If a device cannot be found matching the provided serial number, the function will return unsuccessful. If a device is opened successfully, a handle to the device will be returned through the *device* pointer which can be used to target that device for other API calls.

The function when successful is a blocking call and takes about 3 seconds to finish.

If you wish to target multiple devices or wish to target devices across processes, see **Multiple Devices and Inter-process Device Management**.

Return Values

bbNoError	No error, device number opened and returned successfully.
bbDeviceNotOpenErr	The device was unable to open. This can be returned for many reasons such as the device is not physically connected, eight devices are already open or there is an issue with the USB 3.0 connection.
bbUncalibratedDevice	This message is returned as a warning and notes the device has not been calibrated. If you see this warning, contact Signal Hound.

bbOpenDevice

Open one Signal Hound broadband device

```
bbStatus bbOpenDevice(int *device);
```

Parameters

device	If successful, a device number is returned. This number is used for all successive API function calls.

Description

This function attempts to open the first BB60A/C it detects. If a device is opened successfully, a handle to the device will be returned through the *device* pointer which can be used to target that device for other API calls.

This function when successful, takes about 3 seconds to perform.

If you wish to target multiple devices or wish to target devices across processes, see **Multiple Devices and Inter-process Device Management**.

Return Values

bbNoError	No error, device number opened and returned successfully.
bbDeviceNotOpenErr	The device was unable to open. This can be returned for many reasons such as the device is not physically connected, eight devices are already open or there is an issue with the USB 3.0 connection.
bbUncalibratedDevice	This message is returned as a warning and notes the device has not been calibrated. If you see this warning, contact Signal Hound.

bbCloseDevice

Close a BB60 device

```
bbStatus bbCloseDevice(int device);
```

Description

This function is called when you wish to close a connection with a device. Any resources the device has allocated will be freed and the USB 3.0 connection to the device is terminated. The device closed will be released and will become available to be opened again.

bbConfigureAcquisition

Change the detector type and choose between linear or log scaled returned sweeps

```
bbStatus bbConfigureAcquisition(int device, unsigned int detector, unsigned int
scale);
```

Parameters

detector	Specifies the video detector. The two possible values for detector type are BB_AVERAGE and BB_MIN_AND_MAX.
scale	Specifies the scale in which sweep results are returned int. The four possible values for *scale* are BB_LOG_SCALE, BB_LIN_SCALE, BB_LOG_FULL_SCALE, and BB_LIN_FULL_SCALE.

Description

The *detector* parameter specifies how to produce the results of the signal processing for the final sweep. Depending on settings, potentially many overlapping FFTs will be performed on the input time domain data to retrieve a more consistent and accurate final result. When the results overlap *detector* chooses whether to average the results together, or maintain the minimum and maximum values. If averaging is chosen, the *min* and *max* trace arrays returned from `bbFetchTrace()` will contain the same averaged data.

The *scale* parameter will change the units of returned sweeps. If BB_LOG_SCALE is provided, sweeps will be returned as dBm values, If BB_LIN_SCALE is return, the returned units will be in milli-volts. If the full scale units are specified, no corrections are applied to the data and amplitudes are taken directly from the full scale input.

Return Values

bbInvalidDetectorErr	The detector type provided does not match the list of accepted values.
bbInvalidScaleErr	The scale provided does not match the list of accepted values.

bbConfigureCenterSpan

Change the center and span frequencies `bbStatus bbConfigureCenterSpan(int device, double center, double span);`

Parameters

center	Center frequency in hertz.
span	Span in hertz. If you are configuring the device for I/Q streaming use a valid default value (such as 20MHz) here as the span. The span in this function is ignored for I/Q streaming but the API will still require a valid span be provided. See the bbConfigureIQ function for configuring the I/Q bandwidth.

Description

This function configures the operating frequency band of the broadband device. Start and stop frequencies can be determined from the center and span.

- start = center – (span / 2)
- stop = center + (span / 2)

The values provided are used by the device during initialization and a more precise start frequency is returned after initiation. Refer to the `bbQueryTraceInfo()` function for more information.

Each device has a specified operational frequency range. These limits are specified in the `BB60_MIN_FREQ`, `BB60_MAX_FREQ`, and `BB60_MAX_SPAN` macros. The *center* and *span* provided cannot specify a sweep outside of this range.

There is also an absolute minimum operating span of 20 Hz, but 200kHz is a suggested minimum.

Certain modes of operation have specific frequency range limits. Those mode dependent limits are tested against during `bbInitiate()` and not here..

Return Values

bbInvalidSpanErr	The span provided is less than the minimum acceptable span.
bbFrequencyRangeErr	The calculated start or stop frequencies fall outside of the operational frequency range of the specified device.

bbConfigureIQCenter

Set the center frequency of the device for I/Q acquisition

`bbStatus bbConfigureIQCenter(int device, double centerFreq);`

Parameters

centerFreq	Center frequency in Hertz.

Description

Avoids the need to set a valid span for I/Q acquisition. When switching back to sweep acquisition from I/Q acquisition, you will need to call the bbConfigureCenterSpan function again.

Return Values

See bbConfigureCenterSpan.

bbConfigureLevel

Change the attenuation and reference level of the device

```
bbStatus bbConfigureLevel(int device, double ref, double atten);
```

Parameters

ref	Reference level in dBm.
atten	Attenuation setting in dB. It is recommended to use `BB_AUTO_ATTEN`.

Description

When automatic *atten* is selected, the API uses the *ref* provided to choose the best gain settings for an input signal with amplitude equal to reference level. If an *atten* other than `BB_AUTO_ATTEN` is specified using `bbConfigureLevel()`, the *ref* parameter is ignored.

The *atten* parameter controls the RF input attenuator, and is adjustable from 0 to 30 dB in 10 dB steps. The RF attenuator is the first gain control device in the front end.

When attenuation is automatic, the attenuation and gain for each band is selected independently. When attenuation is not automatic, a flat attenuation is set across the entire spectrum.

It is recommended to set the gain and attenuation to automatic and set the reference level 5 dB higher than the expected input power level.

Return Values

bbReferenceLevelErr	The reference level provided exceeds 20 dBm.
bbAttenuationErr	The attenuation value provided exceeds 30 db.

bbConfigureGain

Change the RF/IF gain path in the device

```
bbStatus bbConfigureGain(int device, int gain);
```

Parameters

gain	A gain setting. It is recommended to use `BB_AUTO_GAIN`.

Description

To return the device to automatically choose the best gain setting, call this function with a gain of

`BB_AUTO_GAIN`.

The gain choices for each device range from 0 to `BB#_MAX_GAIN`.

When `BB_AUTO_GAIN` is selected, the API uses the reference level provided in `bbConfigureLevel()` to choose the best gain setting for an input signal with amplitude equal to the reference level provided.

After the RF input attenuator (0-30 dB), the RF path contains an additional amplifier stage after band filtering, which is selected for medium or high gain and bypassed for low or no gain.

Additionally, the IF has an amplifier which is bypassed only for a gain of zero.

For the highest gain settings, additional amplification in the ADC stage is used.

Return Values

bbInvalidGainErr	This is returned if the gain value is outside the range of possible inputs.

bbConfigureSweepCoupling

Configure sweep processing characteristics

```
bbStatus bbConfigureSweepCoupling(int device, double rbw, double vbw, double
sweepTime, unsigned int rbwShape, unsigned int rejection);
```

Parameters

rbw	Resolution bandwidth in Hz. RBWs can be set to arbitrary values, but may be limited by mode of operation and span. See the bandwidth tables in the Appendix for the Real-time RBW limitations.
vbw	Video bandwidth (VBW) in Hz. VBW must be less than or equal to RBW. VBW can be arbitrary. For best performance use RBW as the VBW. When VBW is set equal to RBW, no VBW filtering is performed.
sweepTime	Suggest a sweep time in seconds. In sweep mode, this value specifies how long the BB60 should sample spectrum for the configured sweep. Larger sweep times may increase the odds of capturing spectral events at the cost of slower sweep rates. The range of possible *sweepTime* values run from 1ms -> 100ms or [0.001 – 0.1].
rbwShape	The possible values for *rbwShape* are BB_RBW_SHAPE_NUTTALL, BB_RBW_SHAPE_FLATTOP, and BB_RBW_SHAPE_CISPR. This choice determines the window function used and the bandwidth cutoff of the RBW filter. BB_RBW_SHAPE_NUTTALL is default and unchangeable for real-time operation.
rejection	The possible values for rejection are BB_NO_SPUR_REJECT and BB_SPUR_REJECT.

Description

The resolution bandwidth represents the bandwidth of spectral energy represented in each frequency bin. For example, with an RBW of 10 kHz, the amplitude value for each bin would represent the total energy from 5 kHz below to 5 kHz above the bin's center. For standard bandwidths, the API uses the 3 dB points to define the RBW. For the CISPR RBW shape, 6dB bandwidths are used.

The video bandwidth is applied after the signal has been converted to frequency domain as power, voltage, or log units. It is implemented as a simple rectangular window, averaging the amplitude readings for each frequency bin over several overlapping FFTs. A signal whose amplitude is modulated at a much higher frequency than the VBW will be shown as an average, whereas amplitude modulation at a lower frequency will be shown as a minimum and maximum value.

Nuttall RBW shapes represent the bandwidths from a single power-of-2 FFT using our sample rate of 80 MSPS and the Nuttall window function. Each RBW is half of the previous. Using Nuttall RBWs can give you the lowest possible bandwidth for any given sweep time, and minimizes processing power. However, scalloping losses of up to 0.8 dB, occurring when a signal falls in between two bins, can cause problems for some types of measurements. While RBW can be set arbitrarily when using the Nuttall window, see the bandwidth table in the Appendix for a list of the RBWs that exhibit the most efficient use of processing.

The Flat-top RBW shape achieves arbitrary RBW values by created variable sized bandwidth flat-top window shapes. Flat-top windows provide low scalloping loss for the most accurate RF measurements.

The CISPR RBW shape uses a Gaussian window and zero-padding to achieve arbitrary RBWs. The RBW is calculated using the 6dB cutoff value.

sweepTime applies to standard swept analysis, and is ignored for other operating modes. If in sweep mode, *sweepTime* is the amount of time the device will spend collecting data before processing. Increasing this value is useful for capturing signals of interest or viewing a more consistent view of the spectrum. Increasing *sweepTime* can have a large impact on the amount of resources used by the API due to the increase of data needing to be stored and the amount of signal processing performed. For this reason, increasing *sweepTime* also decreases the rate at which you can acquire sweeps.

Rejection can be used to optimize certain aspects of the signal. Default is BB_NO_SPUR_REJECT, and should be used in most cases. If you have a steady CW or slowly changing signal, and need to minimize image and spurious responses from the device, use BB_SPUR_REJECT. *Rejection* is ignored outside of standard swept analysis.

Return Values

bbBandwidthErr	*rbw* fall outside device limits.
	vbw is greater than resolution bandwidth.
bbInvalidBandwidthTypeErr	*rbwShape* is not one of the accepted values.

bbConfigureProcUnits

Configure video processing unit type

`bbStatus bbConfigureProcUnits(int device, unsigned int units);`

Parameters

units	The possible values are `BB_LOG`, `BB_VOLTAGE`, `BB_POWER`, and `BB_BYPASS`.

Description

The *units* provided determines what unit type video processing occurs in. The chart below shows which unit types are used for each *units* selection.

For "average power" measurements, `BB_POWER` should be selected. For cleaning up an amplitude modulated signal, `BB_VOLTAGE` would be a good choice. To emulate a traditional spectrum analyzer, select `BB_LOG`. To minimize processing power, select `BB_BYPASS`.

BB_LOG	dBm
BB_VOLTAGE	mV
BB_POWER	mW
BB_BYPASS	No video processing

Return Values

bbInvalidVideoUnitsErr	The value for *units* did not match any known value.

bbConfigureIO

Configure the two I/O ports on a device

`bbStatus bbConfigureIO(int device, unsigned int port1, unsigned int port2);`

Parameters

port1	The first BNC port may be used to input or output a 10 MHz time base (AC or DC coupled), or to generate a general purpose logic high/low output. Please refer to the example below. All possible values for this port are found in the header file and are prefixed with "BB_PORT1"
port2	Port 2 can accept an external trigger or generate a logic output. Port 2 is always DC coupled. All possible values for this port are found in the header file and are prefixed with "BB_PORT2."

Description

NOTE: This function can only be called when the device is idle (not operating in any mode). To ensure the device is idle, call `bbAbort()`.

There are two configurable BNC connector ports available on the device. Both ports functionality are changed with this function. For both ports, ‘0’ is the default and can be supplied through this function to return the ports to their default values. Specifying a ‘0’ on port 1 returns the device to an internal time base and outputs the time base AC coupled. Specifying ‘0’ on port 2 emits a DC coupled logic low.

For external 10 MHz timebases, best phase noise is achieved by using a low jitter 3.3V CMOS input.

Configure combinations

Port 1 IO	For port 1 only a coupled value must be ‘OR’ed together with a port type. Use the ‘|’ operator to combine a coupled type and a port type.
BB_PORT1_AC_COUPLED	Denotes an AC coupled port
BB_PORT1_DC_COUPLED	Denotes a DC coupled port
BB_PORT1_INT_REF_OUT	Output the internal 10 MHz timebase
BB_PORT1_EXT_REF_IN	Accept an external 10MHz time base
BB_PORT1_OUT_LOGIC_LOW	
BB_PORT1_OUT_LOGIC_HIGH	
Port 2 IO	
BB_PORT2_OUT_LOGIC_LOW	
BB_PORT2_OUT_LOGIC_HIGH	
BB_PORT2_IN_TRIGGER_RISING_EDGE	When set, the device is notified of a rising edge
BB_PORT2_IN_TRIGGER_FALLING_EDGE	When set, the device is notified of a falling edge

Return Values

bbDeviceNotIdleErr	This is returned if the device is currently operating in a mode. The device must be idle to configure ports.

Example

This example shows how to configure an AC external reference input into port 1 and a emit a logic high on port 2. Note the ‘|’ operation is used to specify the AC couple.

```
bbConfigureIO (
  myDeviceNumber,
  BB_PORT1_AC_COUPLED | BB_PORT1_EXT_REF_IN, // AC external reference in on port 1
  BB_PORT2_OUT_LOGIC_HIGH                    // Output DC logic high on port 1
);
```

bbConfigureDemod

Configure audio demodulation operation

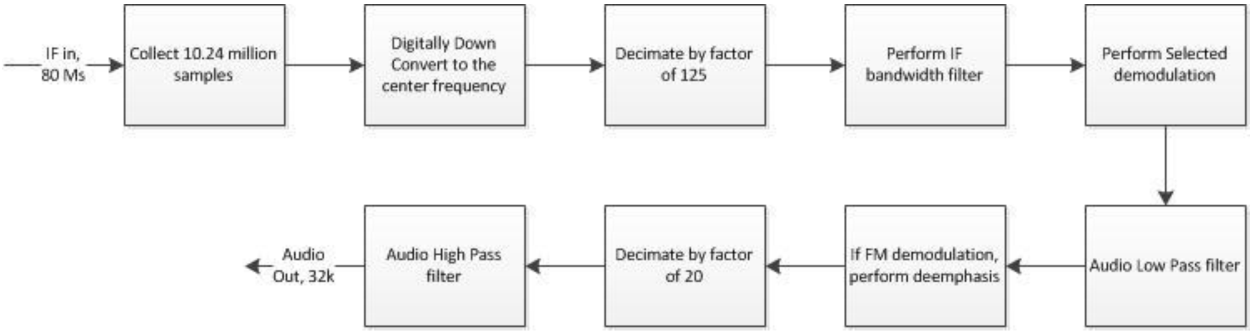
```
bbStatus bbConfigureDemod(int device, int modulationType, double freq, float IFBW,
float audioLowPassFreq, float audioHighPassFreq, float FMDeemphasis);
```

Parameters

modulationType	Specifies the demodulation scheme, possible values are BB_DEMOD_AM/FM/Upper sideband (USB)/Lower Sideband (LSB)/CW.
freq	Center frequency. For best results, re-initiate the device if the center frequency changes +/- 8MHz from the initial value.
IFBW	Intermediate frequency bandwidth centered on freq. Filter takes place before demodulation. Specified in Hz. Should be between 500Hz and 500kHz.
audioLowPassFreq	Post demodulation filter in Hz. Should be between 1kHz and 12kHz Hz.
audioHighPassFreq	Post demodulation filter in Hz. Should be between 20 and 1000Hz.
FMDeemphasis	Specified in micro-seconds. Should be between 1 and 100.

Description

Below is the overall flow of data through our audio processing algorithm.


IF in,
80 Ms
Collect 10.24 million samples
Digitally Down Convert to the center frequency
Decimate by factor of 125
Perform IF bandwidth filter
Perform Selected demodulation
Audio Low Pass filter
If FM demodulation, perform deemphasis
Decimate by factor of 20
Audio High Pass filter
Audio Out, 32k

This function can be called while the device is active.

Return Values

Note: If any of the boundary conditions are not met, this function will return with no error but the values will be clamped to its boundary values.

bbConfigureIQ

Configure the digital IQ data stream

```
bbStatus bbConfigureIQ(int device, int downsampleFactor, double bandwidth);
```

Parameters

downsampleFactor	Specify a decimation rate for the 40MS/s IQ digital stream.
bandwidth	Specify a bandpass filter width on the IQ digital stream.

Description

Decimation Rate	Sample Rate (IQ pairs/s)	Maximum Bandwidth
1	40 MS/s	27 MHz
2	20 MS/s	17.8 MHz
4	10 MS/s	8.0 MHz
8	5 MS/s	3.75 MHz
16	2.5 MS/s	2.0 MHz
32	1.25 MS/s	1.0 MHz
64	625 kS/s	0.5 MHz
128	312.5 kS/s	0.25 MHz
256	156.250 kS/s	140 kHz
512	78.125 kS/s	65 kHz
1024	39062.5 S/s	30 kHz
2048	19531.25 S/s	15 kHz
4096	9765.625 S/s	8 kHz
8192	4882.8125 S/s	4 kHz

This function is used to configure the digital IQ data stream. A decimation factor and filter bandwidth are able to be specified. The decimation rate divides the IQ sample rate directly while the *bandwidth* parameter further filters the digital stream.

For each given decimation rate, a maximum bandwidth value is specified to account for sufficient filter roll off. That table is above. See `bbGetIQ` for retrieving IQ data.

bbConfigureIQDataType

Specify the data type of the I/Q data returned from the API

```
bbConfiguredIQDataType(int device, bbDataType dataType);
```

Parameters

dataType	Data type can be specified either as 32-bit complex floats or 16-bit complex shorts.

Description

See Appendix: I/Q Data Types for more information.

bbConfigureRealTime

Configure the real-time frame parameters

```
bbStatus bbConfigureRealTime(int device, double frameScale, int frameRate);
```

Parameters

frameScale	Specifies the height in dB of the real-time frame. The value is ignored if the scale is linear. Possible values range from [10 – 200].

frameRate	Specifies the rate at which frames are generated in real-time mode, in frames per second. Possible values range from [4 – 30], where four means a frame is generated every 250ms and 30 means a frame is generated every ~33 ms.

Description

The function allows you to configure additional parameters of the real-time frames returned from the API. If this function is not called a scale of 100dB is used and a frame rate of 30fps is used. For more information regarding real-time mode see Modes of Operation : Real-Time Analysis.

bbConfigureRealTimeOverlap

Configure the real-time processing overlap rate

```
bbStatus bbConfigureRealTimeOverlap(int device, double advanceRate);
```

Parameters

advanceRate	FFT advance rate. See description.

Description

By setting the advance rate users can control the overlap rate of the FFT processing in real-time spectrum analysis. The *advanceRate* parameter specifies how far the FFT window slides through the data for each FFT as a function of FFT size. An *advanceRate* of 0.5 specifies that the FFT window will advance 50% the FFT length for each FFT for a 50% overlap rate. Specifying a value of 1.0 would mean the FFT window advances the full FFT length meaning there is no overlap in real-time processing. The default value is 0.5 and the range of acceptable values are between [0.5, 10]. Increasing the advance rate reduces processing considerably but also increases the 100% probability of intercept of the device.

bbInitiate

Change the operating state of the device

```
bbStatus bbInitiate(int device, unsigned int mode, unsigned int flag);
```

Parameters

mode	The possible values for *mode* are BB_SWEEPING, BB_REAL_TIME, BB_AUDIO_DEMOD, and BB_STREAMING.
flag	The default value should be zero. If the mode is equal to BB_STREAMING, the flag should be set to BB_STREAM_IQ (0). *flag* can be used to inform the API to time stamp data using an external GPS receiver. Mask the bandwidth flag ('|' in C) with BB_TIME_STAMP to achieve this. See **Appendix:Using a GPS Receiver to Time-Stamp Data** for information on how to set this up.

Description

`bbInitiate()` configures the device into a state determined by the *mode* parameter. For more information regarding operating states, refer to the **Theory of Operation** and **Modes of Operation** sections. This function calls `bbAbort()` before attempting to reconfigure. It should be noted, if an error is returned, any past operating state will no longer be active.

Return Values

bbInvalidParameterErr	The value for *mode* did not match any known value. In real-time mode, this value may be returned if the span limits defined in the API header are broken. Also in real-time mode, this error will be returned if the resolution bandwidth is outside the limits defined in the API header.
bbAllocationLimitError	This value is returned when the API is unable to allocate the necessary memory to prepare the device for operation. The API is often liberal with memory allocation due to the sheer amount of data being processed. All memory allocation occurs in `bbInitiate()` and deallocation occurs in `bbAbort()`.
bbBandwidthErr	This error is returned if your RBW is larger than your span. (Sweep Mode)

bbFetchTrace

Get one sweep from a configured and initiated device

```
bbStatus bbFetchTrace(int device, int arraySize, double *min, double *max);
bbStatus bbFetchTrace_32f(int device, int arraySize, float *min, float *max);
```

Parameters

arraySize	A provided arraySize. This value must be equal to or greater than the *traceSize* value returned from `bbQueryTraceInfo()`.
min	Pointer to a double buffer, whose length is equal to or greater than *traceSize* returned from `bbQueryTraceInfo()`.
max	Pointer to a double buffer, whose length is equal to or greater than *traceSize* returned from `bbQueryTraceInfo()`.

Description

Returns a minimum and maximum array of values relating to the current mode of operation. If the *detectorType* provided in `bbConfigureAcquisition()` is BB_AVERAGE, the array will be populated with the same values. Element zero of each array corresponds to the *startFreq* returned from `bbQueryTraceInfo()`.

Return Values

bbADCOverflow	This warning is returned when the ADC detects clipping of the input signal. This occurs when the maximum voltage has been reached. Signal analysis and reconstruction become issues on clipped signals. To prevent this, a combination of increasing attenuation, decreasing gain, or increasing reference level (when gain is automatic) will allow for more headroom.
bbUncalSweep	This warning is returned if the sweep returned is invalid due to either USB data loss during device acquisition or being unable to keep up with the necessary processing to create the sweep. The sweep may be discarded, and the function can be called again. This warning will only be returned if the device firmware is at version 7 or greater. If you have a device with an earlier firmware version and believe you are having USB data loss issues, contact Signal Hound.
bbPacketFramingErr	This error occurs when data loss or miscommunication has occurred between the device and the API. During normal operation we do not expect this error to occur. If you find this error occurs frequently, it may be indicative of larger issues. If this error is returned, the data returned is undefined. The device should be power cycled manually or with the *bbPreset* routine.
bbDeviceConnectionErr	Device connection issues were present in the acquisition of this sweep. See **Error Handling: Device Connection Errors.**
bbUSBTimeoutErr	The USB transfer timed out during the requested sweep. Causes may include a faulty USB cable or high processor/kernel load. See **Error Handling: Device Connection Errors**

bbFetchRealTimeFrame

Retrieve one real-time sweep and frame

```
bbStatus bbFetchRealTimeFrame(int device, float *minSweep float *maxSweep, float
*frame, float *alphaFrame);
```

Parameters

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

Description

This function is used to retrieve the real-time sweeps, frame, and alphaFrame. This function should be used instead of *bbFetchTrace* for real-time mode. The sweep arrays should be 'N' values long, where N is the sweep length returned from *bbQueryTraceInfo.* The frame and alphaFrame should be WxH values long where W and H are the values returned from *bbQueryRealTimeInfo.* For more information see Modes of Operation: Real-Time Analysis.

Return Values

bbDeviceConnectionErr	Device connection issues were present in the acquisition of this sweep. See **Error Handling: Device Connection Errors.**
bbUSBTimeoutErr	The USB transfer timed out during the requested sweep. Causes may include a faulty USB cable or high processor/kernel load. See **Error Handling: Device Connection Errors**
bbPacketFramingErr	This error occurs when data loss or miscommunication has occurred between the device and the API. During normal operation we do not expect this error to occur. If you find this error occurs frequently, it may be indicative of larger issues. If this error is returned, the data returned is undefined. The device should be power cycled manually or with the *bbPreset* routine.
bbADCOverflow	This warning is returned when the ADC detects clipping of the input signal. This occurs when the maximum voltage has been reached. Signal analysis and reconstruction become issues on clipped signals. To prevent this, a combination of increasing attenuation, decreasing gain, or increasing reference level (when gain is automatic) will allow for more headroom.

bbFetchAudio

Retrieve 4096 audio samples

```
bbStatus bbFetchAudio(int device, float *audio);
```

Parameters

audio	Pointer to an array of 4096 32-bit floating point values

Description

If the device is initiated and running in the audio demodulation mode, the function is a blocking call which returns the next 4096 audio samples. The approximate blocking time for this function is 128 ms if called again immediately after returning. There is no internal buffering of audio, meaning the audio will be overwritten if this function is not called in a timely fashion. The audio values are typically -1.0 to 1.0, representing full-scale audio. In FM mode, the audio values will scale with a change in IF bandwidth.

Return Values

bbDeviceConnectionErr	Device connection issues were present in the acquisition of audio. See **Error Handling : Device Connection Errors**.

bbGetIQ

Retrieve a block of I/Q data

```
bbStatus bbGetIQ(int device, bbIQPacket *pkt);
bbStatus bbGetIQUnpacked(int device, float *iqData, int iqCount, int *triggers, int
triggerCount, int purge, int *dataRemaining, int *sampleLoss, int *sec, int *nano);
```

Parameters

pkt	Pointer to a bbIQPacket structure

Description

This function retrieves one block of I/Q data as specified by the bbIQPacket struct. The members of the bbIQPacket struct and how they affect the acquisition are described below.

- **iqData –** Set by user, pointer to an array of 32-bit complex floating point values. Complex values are interleaved real-imaginary pairs. This must point to a contiguous block of iqCount complex pairs.
- **iqCount –** Set by user, specify the number of I/Q data pairs to return.
- **triggers –** Set by user, a pointer to an array of integers. If the external trigger input is active, and a trigger occurs during the acquisition time, triggers will be populated with values which are relative indices into the iqData array where external triggers occurred.
- **triggerCount –** Set by user, The size of the triggers array.
- **purge –** Set by user, specify whether to discard any samples acquired by the API since the last time and bbGetIQ function was called. Set to BB_TRUE if you wish to discard all previously acquired data, and BB_FALSE if you wish to retrieve the contiguous I/Q values from a previous call to this function.
- **dataRemaining –** Set by API, how any I/Q samples are still left buffered in the API.
- **sampleLoss –** Set by API, returns BB_TRUE or BB_FALSE for whether the API had to drop data due to the internal circular buffer filling. Will always return false if purge is true.
- **sec –** Set by API, the seconds since epoch representing the timestamp of the first sample in the returned array.
- **nano –** Set by API, the nanoseconds representing the timestamp of the first sample in the returned array.

The timestamps returned will either be synchronized to the GPS if it was properly configured or the PC system clock if not. For timestamps generated by the system clock, one should only use the first timestamp collected and use the index and sample rate to determine the time of a n individual sample.

The BB60 will report ~5k triggers per second. Use an adequate size trigger buffer if you wish to receive all potential triggers. If the API has more triggers to report than the size of the buffer provided, any excess triggers will be discarded.

bbGetIQUnpacked provides a method for retrieving I/Q data without needing the *bbIQPacket* struct. Each parameter in *bbGetIQUnpacked* has a one-to-one mapping to variables found in the *bbIQPacket* struct. This function serves as a convenience for creating bindings in various programming languages and environments such as Python, C#, LabVIEW, MATLAB, etc. This function is implemented by taking the parameters provided into the *bbIQPacket* struct and calling *bbGetIQ*.

Return Values

bbPacketFramingErr	This error occurs when data loss or miscommunication has occurred between the device and the API. During normal operation we do not expect this error to occur. If you find this error occurs frequently, it may be indicative of larger issues. If this error is returned, the data returned is undefined. The device should be power cycled manually or with the `bbPreset()` routine.
bbADCOverflow	This warning is returned when the ADC detects clipping of the input signal. This occurs when the maximum voltage has been reached. Signal analysis and reconstruction become issues on clipped signals. To prevent this, try a combination of increasing attenuation and decreasing gain.
bbDeviceConnectionErr	Device connection issues were present in the acquisition of data. See **Error Handling: Device Connection Errors.**

bbQueryTraceInfo

Returns values needed to query and analyze traces

```
bbStatus bbQueryTraceInfo(int device, unsigned int *traceLen, double *binSize, double
*start);
```

Parameters

traceLen	A pointer to an unsigned int. If the function returns successfully *traceLen* will contain the size of arrays returned by bbFetchTrace.
binSize	A pointer to a 64bit floating point variable. If the function returns successfully, *binSize* will contain the frequency difference between two sequential bins in a returned sweep. In Zero-Span mode, *binSize* refers to the difference between sequential samples in seconds.
start	A pointer to a 64bit floating point variable. If the function returns successfully, *start* will contain the frequency of the first bin in a returned sweep. In Zero-Span mode, *start* represents the exact center frequency used by the API.

Description

This function should be called to determine sweep characteristics after a device has been configured and initiated. For zero-span mode, startFreq and binSize will refer to the time domain values. In zero-span mode *startFreq* will always be zero, and *binSize* will be equal to sweepTime/traceSize.

bbQueryRealTimeInfo

Query the frame size of the real-time frame

```
bbStatus bbQueryRealTimeInfo(int device, int *frameWidth, int *frameHeight);
```

Parameters

frameWidth	Pointer to a 32-bit signed integer.
frameHeight	Pointer to a 32-bit signed integer.

Description

This function should be called after initializing the device for Real-Time mode. This device returns the frame size of the real-time frame configured.

bbQueryRealTimePoi

Get the configured 100% probability of intercept of the device configured for real-time spectrum analysis

```
bbStatus bbQueryRealTimePoi(int device, double *poi);
```

Parameters

poi	Pointer to double. See description.

Description

When this function returns successfully, the value *poi* points to will contain the 100% probability of intercept duration in seconds of the device as currently configured in real-time spectrum analysis. The device must actively be configured and initialized in the real-time spectrum analysis mode.

bbQueryStreamInfo

Retrieve values need to query and analyze an I/Q data stream

```
bbQueryStreamInfo(int device, int *return_len, double *bandwidth, int
*samples_per_sec);
```

Parameters

return_len	The number of I/Q samples pairs which will be returned by calling `bbFetchRaw.` This value is part of the deprecated bbFetchRaw function and is not needed if using the bbGetIQ function.
bandwidth	The bandpass filter bandwidth, width in Hz. Width is specified by the 3dB rolloff points.
samples_per_sec	The number of I/Q pairs to expect per second. (This parameter is no longer required for I/Q streaming. You can pass NULL for this parameter if you are configuring the device for I/Q data).

Description

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

bbGetIQCorrection

Get correction constant for 16-bit complex short I/Q acquisitions

```
bbStatus bbGetIQCorrection(int device, float *correction);
```

Parameters

corrections	Pointer to 32-bit float. The value correction points to will contain a scalar which can be used to convert full scale I/Q data 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. This function should be called after configuring and initiating the device for I/Q acquisitions.

bbAbort

Stop the current mode of operation

```
bbStatus bbAbort(int device);
```

Description

Stops the device operation and places the device into an idle state.

bbPreset

Trigger a device reset

```
bbStatus bbPreset(int device);
```

Description

This function exists to invoke a hard reset of the device. This will function similarly to a power cycle(unplug/re-plug the device). This might be useful if the device has entered an undesirable or unrecoverable state. Often the device might become unrecoverable if a program closed unexpectedly, not allowing the device to close properly. This function might allow the software to perform the reset rather than ask the user perform a power cycle.

Viewing the traces returned is often the best way to determine if the device is operating normally. To utilize this function, the device must be open. Calling this function will trigger a reset which happens after 2 seconds. Within this time you must call `bbCloseDevice()` to free any remaining resources and release the device serial number from the open device list. From the time of the `bbPreset()` call, we suggest 3 to more seconds of wait time before attempting to re-open the device.

Example

```
// Notes: Invoking a sleep in the main thread of execution may be undesirable
//   in a GUI application. This function is best performed in a separate thread.
// The amount of time to Sleep is dependent on how fast the device will register
//   on your machine after it resets. A longer sleep time may be preferred or multiple
//   attempts to open the device until it returns bbNoError
bool PresetRoutine() {

  bbPreset( myID );
  bbCloseDevice( myID );

  Sleep(3000); // Windows sleep function

  // Alternative 1: Assume it's ready
  if( bbOpenDevice( &myID ) == bbNoError )
    return true;
  else
    return false;


  // Alternative 2: Try a few times, it may not be ready at first
  int trys = 0;
  while(trys++ < 3) {
    if( bbOpenDevice( &myID ) == bbNoError )
      return true;
    else
      Sleep(500);
  }
  return false;
}
```

bbSelfCal

Calibrate the device for significant temperature changes. BB60A only

`bbStatus bbSelfCal(int device);`

Description

This function causes the device to recalibrate itself to adjust for internal device temperature changes, generating an amplitude correction array as a function of IF frequency. This function will explicitly call `bbAbort()` to suspend all device operations before performing the calibration, and will return the device in an idle state and configured as if it was just opened. The state of the device should not be assumed, and should be fully reconfigured after a self-calibration.

Temperature changes of 2 degrees Celsius or more have been shown to measurably alter the shape/amplitude of the IF. We suggest using `bbGetDeviceDiagnostics()` to monitor the device's temperature and perform self-calibrations when needed. Amplitude measurements are not guaranteed to be accurate otherwise, and large temperature changes (10˚C or more) may result in adding a dB or more of error.

Because this is a streaming device, we have decided to leave the programmer in full control of when the device in calibrated. The device is calibrated once upon opening the device through `bbOpenDevice()` and is the responsibility of the programmer after that.

Note:
After calling this function, the device returns to the default state. Currently the API does not retain state prior to the calling of `bbSelfCal()`. Fully reconfiguring the device will be necessary.

bbSyncCPUtoGPS

Synchronize a GPS reciever with the API

```
bbStatus bbSyncCPUtoGPS(int comPort, int baudRate);
```

Parameters:

comPort	Com port number for the NMEA data output from the GPS reciever.
baudRate	Baud Rate of the Com port.

Description:

This function is currently not support on the Linux operating system.

The connection to the COM port is only established for the duration of this function. It is closed when the function returns. Call this function once before using a GPS PPS signal to time-stamp RF data. The synchronization will remain valid until the CPU clock drifts more than ¼ second, typically several hours, and will re-synchronize continually while streaming data using a PPS trigger input.

This function calculates the offset between your CPU clock time and the GPS clock time to within a few milliseconds, and stores this value for time-stamping RF data using the GPS PPS trigger. This function ignores time zone, limiting the calculated offset to +/- 30 minutes. It was tested using an FTS 500 from Connor Winfield at 38.4 kbaud. It uses the "$GPRMC" string, so you must set up your GPS to output this string.

Return Values:

bbGPSErr	Returned when no GPS reciever was found, unable to establish communication with the specified port, or unable to decipher the GPRMC string.

bbGetDeviceType

Retrieve the model type of a device handle

```
bbStatus bbGetDeviceType(int device, int *type);
```

Parameters

type	Pointer to an integer to receive the model type.

Description

This function may be called only after the device has been opened. If the device successfully opened, *type* will contain the model type of the device pointed to by *handle.*

Possible values for *type* are `BB_DEVICE_NONE`, `BB_DEVICE_BB60A`, and `BB_DEVICE_BB60C`. These values can be found in the bb_api header file.

bbGetSerialNumber

Retrieve the serial number of the device

```
bbStatus bbGetSerialNumber(int device, unsigned int *sid);
```

Parameters

sid	Pointer to unsigned int which will be assigned the serial number of the broadband device specified with *device.*

Description

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

bbGetFirmwareVersion

Determine the firmware version of a SignalHound broadband device

```
bbStatus bbGetFirmwareVersion(int device, int *version);
```

Parameters

version	Pointer to an integer, will contain the firmware version of the specified device if this function returns successfully.

Description

Use this function to determine which version of firmware is associated with the specified device.

bbGetDeviceDiagnostics

Retrieve the current internal device characteristics

```
bbStatus bbGetDeviceDiagnostics(int device, float *temperature, float *usbVoltage,
float *usbCurrent);
```

Parameters

temperature	Pointer to 32-bit float. If the function is successful *temperature* will point to the current internal device temperature, in degrees Celsius. See "bbSelfCal" for an explanation on why you need to monitor the device temperature.

voltageUSB	USB operating voltage, in volts. Acceptable ranges are 4.40 to 5.25 V.
currentUSB	USB current draw, in milliamps.

Description

Pass *null* to any parameter you do not wish to query.

The device temperature is updated in the API after each sweep is retrieved. The temperature is returned in Celsius and has a resolution of 1/8th of a degree.

A USB voltage of below 4.4V may cause readings to be out of spec. Check your cable for damage and USB connectors for damage or oxidation.

bbAttachTg

Pairs and open BB60 spectrum analyzer with a Signal Hound tracking generator

```
bbStatus bbAttachTg(int device);
```

Description

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

Return Values

bbNotSupportedErr	The device specified does not have the firmware version necessary to support performing tracking generator sweeps.

bbIsTgAttached

Determine if a Signal Hound tracking generator is paired with the specified device

```
bbStatus bbIsTgAttached(int device, bool *attached);
```

Parameters

attached	Pointer to a boolean variable. If this function returns successfully, the variable attached points to will contain a true/false value as to whether a tracking generator is paired with the spectrum analyzer.

Description

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

bbConfigTgSweep

Configure a tracking generator sweep

```
bbStatus bbConfigureTgSweep(int device, int sweepSize, boolHighDynamicRange, bool
passiveDevice);
```

Parameters

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

Description

This function configures the tracking generator sweeps. Through this function you can request a sweep size. The sweep size is the number of discrete points returned in the sweep over the configured span. The final value chosen by the API can be different than the requested size by a factor of 2 at most. The dynamic range of the sweep is determined by the choice of *highDynamicRange* and *passiveDevice.* A value of *true* for both provides the highest dynamic range sweeps. Choosing *false* for *passiveDevice* suggests to the API that the device under test is an active device (amplification).

bbStoreTgThru

Perform a store thru

```
bbStatus bbStoreTgThru(int device, int flag);
```

Parameters

flag	Specify the type of store thru. Possible values are TG_THRU_0DB and TG_THRU_20DB.

Description

This function, with flag set to TG_THRU_0DB, notifies the API to use the next trace as a thru (your 0 dB reference). Connect your tracking generator RF output to your spectrum analyzer RF input. This can be accomplished using the included SMA to SMA adapter, or anything else you want the software to establish as the 0 dB reference (e.g. the 0 dB setting on a step attenuator, or a 20 dB attenuator you will be including in your amplifier test setup).

After you have established your 0 dB reference, a second step may be performed to improve the accuracy below -40 dB. With approximately 20-30 dB of insertion loss between the spectrum analyzer and tracking generator, call saStoreTgThru with flag set to TG_THRU_20DB. This corrects for slight variations between the high gain and low gain sweeps.

bbSetTg

Set the frequency and amplitude output of a paired tracking generator

```
bbStatus bbSetTg(int device, double frequency, double amplitude);
```

Parameters

frequency	Set the frequency, in Hz, of the TG output
amplitude	Set the amplitude, in dBm, of the TG output

Description

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

Return Values

bbTrackingGeneratorNotFound	A tracking generator was not found to be paired with the device specified.
bbDeviceNotConfiguredErr	The API is currently configured and initiated for tracking generator sweeps and the tracking generator cannot be controlled at this time.

bbSetTgReference

Configure the timebase for the TG44 and TG124

```
bbStatus bbSetTgReference(int device, int reference);
```

Parameters

reference	A valid time base setting value. Possible values are TG_REF_UNUSED, TG_REF_INTERNAL_OUT, and TG_REF_EXTERNAL_IN

Description

Configure the time base for the tracking generator attached to the device specified. When TG_REF_UNUSED is specified additional frequency corrections are applied. If using an external reference or you are using the TG time base frequency as the frequency standard for your system, you will want to specify TG_REF_INTERNAL_OUT or TG_REF_EXTERNAL_IN so the additional corrections are not applied.

Return Values

bbTrackingGeneratorNotFound	A tracking generator is not attached to the device.
bbDeviceNotConfigured	The tracking generator is actively sweeping and cannot be configured.

bbGetTgFreqAmpl

Retrieve the last set TG configuration

```
bbStatus bbGetTgFreqAmpl(int device, double *frequency, double *amplitude);
```

Parameters

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

Description

Retrieve the last set TG output parameters the user set through the saSetTg function. The setTg function must have been called for this function to return valid values. If the TG was used to perform scalar network analysis at any point, this function will not return valid values until the setTg function is called again.

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

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

Return Values

bbTrackingGeneratorNotFound	No tracking generator has been attached to the BB device.
bbDeviceNotConfiguredErr	The API is currently configured and initiated for tracking generator sweeps and the tracking generator cannot be controlled at this time.

bbGetAPIVersion

Get an API software version string

```
const char* bbGetAPIVersion();
```

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. An example string is below .. ['3' | '.' | '0' | '.' | '1' | '1' | '\0'] = "3.0.11"

bbGetErrorString

Produce an error string from an error code

```
const char* bbGetErrorString(bbStatus code);
```

Parameters

code	A bbStatus value returned from an API call.

Description

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

Return Values

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

Appendix

Device Connection Errors

The API issues errors when fatal connection issues are present during normal operation of the device. The two major errors in this category are **bbPacketFramingErr** and **bbDeviceConnectionErr**. These errors are reported on fetch routines, as these routines contain most major device I/O.

bbPacketFramingErr – Packet framing issues can occur in low power settings or when large interrupts occur on the PC (typically large system interrupts). This error can be handled by manually cycling the device power, or programmatically by using the preset routine.

bbDeviceConnectionErr – Device connection errors are the result of major USB issues most commonly being the device has lost power (unplugged). These errors should be handled by completely closing the software and cycling the device power, or, if you wish for the software to remain open, call the function bbCloseDevice before cycling the device power and re-opening the device as usual.

Firmware Version 7 (BB60C)

This firmware update has increased the stability of the BB60C on PCs that have a heavy CPU load or in instances where there is random data loss over USB. Currently in the instance where a PC kernel load is high, the PC may not be able to service the USB causing data loss to occur, and the BB60C API reports device connection issues, requiring a device preset. This can also happen if there is (random) data loss over USB 3.0 (rarer). This update resolves these issues by not failing on the connection issues and issuing a warning on the fetch trace functions notifying a user to discard the sweep and try again. The benefit of this is that the API and device remain connected and do not require a preset, often taking about 6 seconds. If you have a device that requires a firmware update, contact Signal Hound.

Code Examples

Code examples have been moved to the SDK.
https://signalhound.com/software/signal-hound-software-development-kit-sdk/

Manual Gain/Attenuation

Gain and attenuation are used to control the path the RF takes through the device. Selecting the proper gain and attenuation settings greatly affect the dynamic range of the resulting signal. When gain and attenuation are set to automatic, the reference level is used to control the internal amplifiers and attenuators. Choosing a reference level slightly above the maximum expected power level ensures the device engages the best possible configuration. Manually configuring gain and attenuation should only be used after testing and observation.

Additionally, when gain and attenuation are set to auto in sweep mode, the API can optimize the gain and attenuation across the frequency range of the device. When using manual settings, it will use the same gain/atten values across the entire span, which may be sub-optimal.

I/Q Data Types

I/Q data is returned from the bbGetIQ function either as 32-bit complex floats or 16-bit complex shorts depending on the data type set in bbConfigureIQDataType. 16-bit shorts are twice as memory efficient as floats 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 bbGetIQCorrection.
    a. re32f *= correction;
    b. im32f *= correction;
3) Calculate power
    a. Sample Power (dBm) = 10.0 * log10(re32f*re32f + im32f*im32f);

Using a GPS Receiver to Time-Stamp Data

With minimal effort it is possible to determine the absolute time (up to 50ns) of the ADC samples. This functionality is only available when the device is configured for IF or I/Q streaming. Additionally, this functionality is only available for Windows operating systems. It does not work on Linux.

What you will need:

1) GPS Receiver capable providing NMEA data, specifically the GPRMC string, and a 1PPS output. (Tested with Connor Winfield Xenith TBR FTS500)
2) The NMEA data must be provided via RS232 (Serial COM port) only once during application startup, releasing the NMEA data stream for other applications such as a “Drive Test Solution” to map out signal strengths.

Order of Operations:

1) Ensure correct operation of your GPS receiver.
2) Connect the 1PPS receiver output to port 2 of the device.

3) Connect the RS232 receiver output to your PC.
4) Determine the COM port number and baud rate of the data transfer over RS232 to your PC.
5) Open the device via bbOpenDevice
6) Ensure the RS232 connection is not open.
7) Use bbSyncCPUtoGPS to synchronize the API timing with the current GPS time. This function will release the connection when finished.
8) Configure the device for I/Q streaming.
9) Before initiating the device, use bbConfigureIO and configure port 2 for an incoming rising edge trigger via `BB_PORT2_IN_TRIGGER_RISING_EDGE`.
10) Call bbInitiate`(id, BB_STREAMING, BB_TIME_STAMP)`. The `BB_TIME_STAMP` argument will tell the API to look for the 1PPS input trigger for timing.
11) If initiated successfully you can now fetch I/Q data and timestamps with bbGetIQ. The timestamp returned will be the time of the first sample in the array of data collected.

Code Example

See the examples folder in the API download for an example of timestamping the I/Q data using a GPS receiver.

Bandwidth Tables

For Nuttall RBW shapes, this table shows the possible RBWs and their corresponding FFT sizes. RBWs that are between these values use zero-padding to achieve the selected RBW.

Native Bandwidths (Hz)		FFT size
10.10e6		16
5.050e6		32
2.525e6		64
1.262e6		128
631.2e3	Largest Real-Time RBW	256
315.6e3		512
157.1e3		1024
78.90e3		2048
39.45e3		4096
19.72e3		8192
9.863e3		16384
4.931e3		32768
2.465e3	Smallest Real-Time RBW	65536
1.232e3		131072
616.45		262144
308.22		524288
154.11		1048576
154.11		1048576
77.05		2097152
38.52		4194304
19.26		8388608
9.63		16777549
4.81		33554432
2.40		67108864

1.204		134217728
0.602		268435456
0.301		536870912

Flat-top RBWs and FFT size

It is possible to determine the FFT length used by the API when utilizing flat-top RBWs. The function below returns the FFT length for an arbitrary RBW. A custom flat-top window with variable bandwidth is built to modify the signal bandwidth beyond just FFT length.

```
int fft_size_from_flattop_rbw(double rbw)
{
    double min_bin_sz = rbw / 3.2;
    double min_fft = 80.0e6 / min_bin_sz;
    int order = (int)ceil(log2(min_fft));

    return pow2(order);
}
```