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

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

35707 NE 86th Ave
La Center, WA 98629 USA
Phone 360.263.5006 • Fax 360.263.5007

Version 3.0
6/24/2015

Requirements, Operation, Function Definitions, Examples

Table of Contents

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 a number of 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, the appendix contains a number of code examples.

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.
The Appendix provides various code examples and tips.

Legend

Broadband Device	A Signal Hound BB-series spectrum analyzer.
Device	Shortened for *broadband device,* for brevity.
Instantaneous Bandwidth	The usable bandwidth at the intermediate frequency of the device. 20MHz for the BB60A. 27MHz for the BB60C.
RBW	Resolution Bandwidth
VBW	Video Bandwidth

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. We encourage any and all criticisms or ideas. We would love to hear how you might improve the API.

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

Build/Version Notes

Windows builds for x86 and x64 are available
As of Version 3.0.5 a 64-bit Linux API build is available.
The Windows builds are compiled with Visual Studio 2012 and any application using this library will require distributing the VS2012 redistributable libraries.
Build notes for the Linux API are provided in the Linux SDK found on the Signal Hound website.

What's New in Version 3.0

Version 3.0 coincides with the release of the Spike^{TM}, 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
- 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.

Linux Development Requirements

- 64-bit Linux operating system. (Tested and developed on Ubuntu 14.04)
- 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.

For full requirements and installation information, see the README in the Linux SDK.

General Requirements

- A basic understanding of RF spectrum analysis.
- A Signal Hound BB60 spectrum analyzer.
- Dual / Quad core Intel I-series processors, preferably 2nd generation (Sandy 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 broadband device will be as follows.

1) Open a USB 3.0 connected BB60A/BB60C.
2) Configure the device.
3) Initiate a device mode of operation
4) Retrieve data from the device
5) Abort the current mode of operation
6) Close the device

- Calibration

The API provides functions for each step in this process. We have strived to mimic the functionality and naming conventions of SCPIs *IviSpecAn* Class Specification where possible. It is not necessary to be familiar with this specification but those who are should feel comfortable with our API immediately. Let's look at each step in detail of a typical program interfacing a Signal Hound spectrum analyzer.

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 power light is lit on the device and 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 broadband device you are interfacing.

Configuring the Device

Once the device is opened, it must be configured. The API provides a number of configuration routines and many operating states. 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 configurations 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 re-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 broadband 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. Each mode will also provide data formatted to match the mode selected. In the next sections you will see how to interact with each mode.

For a more in-depth examination of each mode of operation (read: *theory*) refer to the Signal Hound BB60C/A 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/sweep time`
- `bbConfigureWindow() – Configuring window functions for certain RBWs`
- `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()` function. 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 where 'n' is a *zero* based sample point.

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

Real-Time Analysis

The API provides the functionality of an online real-time spectrum analyzer for the full instantaneous bandwidth of the device (20MHz for the BB60A, 27MHz for the BB60C). Through the use of FFTs at an overlapping rate of 50%, the spectrum results have no blind time (100% probability of intercept) for events as short as 50ns. Due to the demands in processing, restrictions are placed on resolution bandwidth. Video bandwidth is non-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 of time. Both the sweep and density map are returned at rate specified by the function *bbConfigureRealTime.* For a full example of using real-time see Appendix: Code Examples: Real-Time Mode.

I/Q Streaming

The API is capable of providing programmers with a continuous stream of digital I/Q samples from the device. The digital I/Q stream consists of interleaved 32-bit floating point I/Q pairs scaled to mV. The digital samples are amplitude corrected providing accurate measurements. The I/Q data rate at its highest is 40MS/s and can be decimated down by a factor of up to 128 (in powers of two). Each decimation value further reduces the overall bandwidth of the I/Q samples, so the API also provides a configurable band pass filter to control the overall passband of a given I/Q data stream. The I/Q data stream can also be tuned to an arbitrary frequency value.

Configuration routines used to prepare streaming are

- `bbConfigureCenterSpan() – Set the center frequency of the I/Q data stream.`
- `bbConfigureLevel() – See` *Gain and Attenuation in the Streaming Mode*
- `bbConfigureGain() – See` *Gain and Attenuation in the Streaming Mode*
  `bbConfigureIO() – Configure the BNC ports of the BB60.`
- `bbConfigureIQ() – Specify the decimation and bandwidth of the I/Q data stream.`

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 digital samples in circular buffers. It is the responsibility of the user application to poll the samples via `bbFetchRaw()` fast enough to prevent the circular buffers from wrapping. We suggest a separate polling thread and synchronized data structure (buffer) for retrieving the samples and using them in your application.

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

I/Q streaming is also the only mode in which you can time stamp data (See *Appendix:Using a GPS Reciever to Time-Stamp Data)* and determine external trigger locations (See `bbFetchRaw()`).

Gain and Attenuation in the Streaming Mode

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.

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 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.

Sweeping versus Streaming

All modes of operation fall within two categories, sweeping and streaming. In any sweeping mode, the device operates only when requested. For example, requesting a trace triggers a single trace acquisition, otherwise the device and API are idle. Sweeping is very responsive and switching between different types of sweep modes is very quick. Streaming modes are modes in which the API is continually receiving a stream of digitized IF from the device. The device is never idle in these modes. Once this process is started, it takes about ½ second to abort any streaming operation, to ensure all channels/pipes have been cleared and the device is ready for its next command.

Note: Entering a streaming mode is nearly instantaneous if the device is coming from an idle or sweep mode.

Depending on your application this ½ second abort time may not be acceptable (switching bands quickly/changing settings quickly). If you are interested in utilizing a streaming mode to fully characterize a signal of interest, a good approach might be to start in the standard sweep mode and switch to a streaming mode once you identified a frequency of interest.

Multi-Threading

The BB60 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 Inter-process Device Management

The API is capable of managing 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

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 max will 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. All unused values will be populated with the sentinel value of -1. The integer pointed to by *deviceCount* will equal the number of devices reported by this function upon returning.

Return Values

bbNoError	No error, function returned successfully.
bbNullPtrErr	One required pointer parameter to this function is null.

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.
bbNullPtrErr	The parameter device is null. The device is not opened.
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.
bbNullPtrErr	The parameter device is null. The device is not opened.

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 one Signal Hound broadband device

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

Parameters

device	Handle to the device being closed.

Description

This function is called when you wish to terminate 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.

Return Values

bbNoError	The device closed successfully.
bbDeviceNotOpenErr	The device specified is not open.

bbConfigureAcquisition

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

```
bbStatus bbConfigureAcquisition(int device, unsigned int detectorType, unsigned int
verticalScale);
```

Parameters

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

Description

detectorType 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 *detectorType* 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 *verticalScale* parameter will change the units of returned sweeps. If BB_LOG_SCALE is provided sweeps will be returned in amplitude unit dBm. 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

bbNoError	Function completed successfully.
bbDeviceNotOpenErr	The device handle provided points to a device that is not open.
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

device	Handle to the device being configured.
center	Center frequency in hertz.
span	Span in hertz.

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 `BB#_MIN_FREQ` and `BB#_MAX_FREQ`. 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

bbNoError	Device successfully configured.
bbDeviceNotOpenErr	The device handle provided points to a device that is not open.
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.

bbConfigureLevel

Change the attenuation and reference level of the device

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

Parameters

device	Handle to the device being configured.
ref	Reference level in dBm.
atten	Attenuation setting in dB. If attenuation provided is negative, attenuation is selected automatically.

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 automatic gain and attenuation and set the reference level to a value slighly higher than the expected inpu power level.

Return Values

bbNoError	Device successfully configured.
bbDeviceNotOpenErr	The device handle provided points to a device that is not open.
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

device	Handle to the device being configured.
gain	A gain setting.

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

bbNoError	Device successfully configured.
bbDeviceNotOpenErr	The device handle provided does not point to an open device.
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 rbwType, unsigned int rejection);
```

Parameters

device	Handle to the device being configured.
rbw	Resolution bandwidth in Hz. Use the bandwidth table in the appendix to determine good values to choose. As of 1.07 in non-native mode, RBW can be arbitrary. Therefore you may choose values not in the table and they will not clamp.
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.
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]. In the real-time configuration, this value represents the length of time data is collected and compounded before returning a sweep. Values for real-time should be between 16ms-100ms [0.016 – 0.1] for optimal use. In zero span mode this is the length of the returned sweep as a measure of time. Sweep times for zero span must range between 10us and 100ms. Values outside this range are clamped.

rbwType	The possible values for *rbwType* are `BB_NATIVE_RBW` and `BB_NON_NATIVE_RBW`. This choice determines which bandwidth table is used and how the data is processed. `BB_NATIVE_RBW` 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, or RBW, 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.

The video bandwidth, or VBW, 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.

Native RBWs represent the bandwidths from a single power-of-2 FFT using our sample rate of 80 MSPS and a high dynamic range window function. Each RBW is half of the previous. Using native 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.

Non-native RBWs use the traditional 1-3-10 sequence. As of version 1.0.7, non-native bandwidths are not restricted to the 1-3-10 sequence but can be arbitrary. Programmatically, non-native RBW's are achieved by creating variable sized bandwidth flattop windows.

sweepTime applies to regular sweep mode and real-time mode. 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* has a very 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.

In real-time, *sweepTime* refers to how long data is accumulated before returning a sweep. Ensure you are capable of retrieving as many sweeps that will be produced by changing this value. For instance, changing *sweepTime* to 32ms in real-time mode will return approximately 31 sweeps per second (1000/32).

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. If you have a signal between 300 MHz and 3 GHz, need the lowest possible phase noise, and do not need any image rejection, BB_BYPASS_RF can be used to rewire the front end for lowest phase noise.

Return Values

bbNoError	Device successfully configured.
bbDeviceNotOpenErr	The device handle provided points to a device that is not open.
bbBandwidthErr	*rbw* fall outside device limits.
	vbw is greater than resolution bandwidth.
bbInvalidBandwidthTypeErr	*rbwType* is not one of the accepted values.
bbInvalidParameterErr	*rejection* is not one of the accepted values.

bbConfigureWindow

Change the windowing function

```
bbStatus bbConfigureWindow(int device, unsigned int window);
```

Parameters

device	Handle to the device being configured.
window	The possible values for window are `BB_NUTALL`, `BB_BLACKMAN`, `BB_HAMMING`, and `BB_FLAT_TOP`.

Description

This changes the windowing function applied to the data before signal processing is performed. In real-time configuration the window parameter is permanently set to `BB_NUTALL`. The windows are only changeable when using the `BB_NATIVE_RBW` type in `bbConfigureSweepCoupling()`. When using `BB_NON_NATIVE_RBW`, a custom flattop window will be used.

Return Values

bbNoError	Device successfully configured
bbDeviceNotOpen	The device handle provided points to a device that is not open.
bbInvalidWindowErr	The value for *window* did not match any known value

bbConfigureProcUnits

Configure video processing unit type

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

Parameters

device	Handle to the device being configured.
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

bbNoError	Device successfully configured
bbDeviceNotOpen	The device handle provided points to a device that is not open.
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

device	Handle to the device being configured.
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 is capable of accepting an external trigger or generating 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

bbNoError	Device configured successfully.
bbDeviceNotOpenErr	Device specified is not open.
bbDeviceNotIdleErr	This is returned if the device is currently operating in a mode. The device must be idle to configure ports.
bbInvalidParameterErr	A parameter supplied is unknown.

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);
```

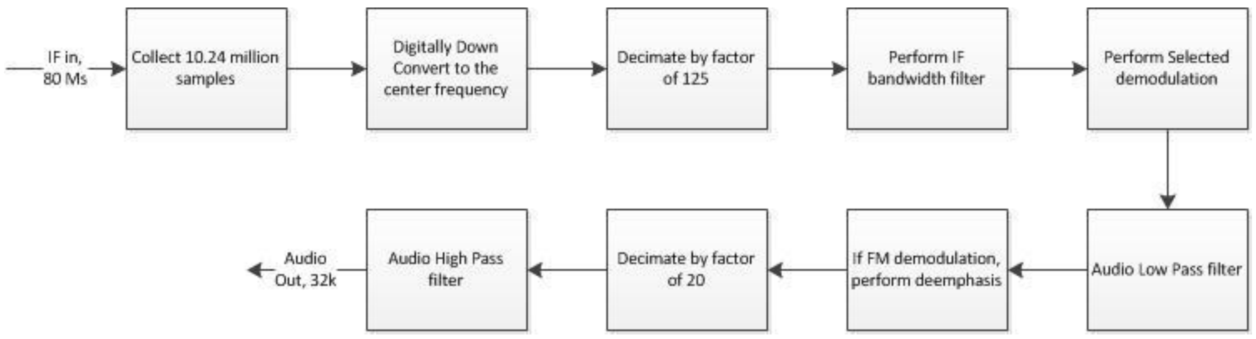
Parameters

device	Handle to the device being configured.
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

bbNoError	Function completed successfully
bbDeviceNotOpenErr	The device specified is not open.

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 I/Q data stream

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

Parameters

device	Handle to the device being configured.
downsampleFactor	Specify a decimation rate for the 40MS/s I/Q digital stream.
bandwidth	Specify a bandpass filter width on the I/Q digital stream.

Decimation Rate	Sample Rate (I/Q 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	0.625 MS/s	0.5 MHz
128	0.3125 MS/s	0.125 MHz

Description

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

For each given decimation rate, a maximum bandwidth value must be supplied to account for sufficient filter rolloff. That table is above. See `bbFetchRaw()` for polling the I/Q data stream.

See **Appendix: Code Examples: I/Q Streaming Example.**

Return Values

bbNoError	Function completed successfully.
bbDeviceNotOpenErr	The device specified is not open.
bbInvalidParameterErr	The downsample rate is outside the acceptable input range. The downsample rate is not a power of two.
bbClampedToLowerLimit	The bandpass filter width specified is lower than `BB_MIN_IQ_BW`
bbClampedToUpperLimit	Warning that the bandpass filter width was clamped to the maximum value allowed by the *downsampleFaction.*

bbConfigureRealTime

Configure the real-time frame parameters

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

Parameters

device	Handle to the device being configured.
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.

Return Values

bbNoError	Device sucessfully configured.
bbDeviceNotOpenErr	The device handle provided does not point to an open device.
bbAdjustedParameter	The parameter frameScale or frameRate fell outside the range of acceptable values and was clamped.

bbInitiate

Change the operating state of the device

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

Parameters

device	Handle to the device being configured.
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 can contain BB_STREAM_IQ for standard I/Q streaming or BB_STREAM_IF for direct IF digital samples. *flag* can be used to inform the API to time stamp data using an external GPS reciever. 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

bbNoError	Device successfully configured
bbDeviceNotOpenErr	The device handle provided does not point to an open device.
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. In time-gate analysis mode this error will be returned if span limits defined in the API header are broken. Also in time gate analysis, this error is returned if the bandwidth provided require more samples for processing than is allowed in the gate length. To fix this, increase rbw/vbw.
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

device	Handle of an initialized device.
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

bbNoError	Successful. pSweepDataMin/Max are populated with amplitude values.
bbNullPtrErr	If either *min* or *max* are null, bbNullPtrErr is returned immediately.
bbDeviceNotOpenErr	*device* is not a handle to an open device.
bbDeviceNotConfiguredErr	Returned if the device is idle or in BB_RAW_PIPE mode.

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.
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

Parameters

device	Handle to an initialized device configured in real-time mode.
sweep	Pointer to a floating point array. If the function returns successfully, the contents of the array will be a frequency sweep.
frame	Pointer to a floating point array. If the function returns successfully, the contents of the array will contain a single real-time frame.

Description

This function is used to retrieve one real-time frame and sweep. This function should be used instead of *bbFetchTrace* for real-time mode. The sweep array should be 'N' number of value long, where N is the sweep length returned from *bbQueryTraceInfo.* The frame 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

bbNoError	The function returned successfully.
bbDeviceNotOpenErr	The device specified is not open.
bbNullPtrErr	One or more of the supplied pointers are NULL.
bbDeviceNotConfiguredErr	The device specified is not currently configured for real-time mode.
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

device	Handle of an initialized device.
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

bbNoError	Function returned successfully
bbDeviceNotOpenErr	The device specified is not open
bbDeviceNotConfiguredErr	The device is not initiated and running the audio demodulation mode.
bbNullPtrErr	*audio* pointer is NULL
bbDeviceConnectionErr	Device connection issues were present in the acquisition of audio. See **Error Handling : Device Connection Errors**.

bbFetchRaw

Retrieve raw data from a streaming device

```
bbStatus bbFetchRaw(int device, float *buffer, int *triggers);
```

Parameters

device	Handle of a streaming device.
buffer	A pointer to a 32-bit floating point buffer. The contents of this buffer will be updated with interleaved I/Q digital samples when streaming I/Q or IF values ranging from -1/+1 full scale when streaming IF.
triggers	*triggers* is a pointer to an array of 68 integers representing external trigger information relative to the buffer. Read the description below for in-depth discussion.

Description

Retrieve the next array of I/Q samples in the stream. The length of the buffer provided to this function is the return length from `bbQueryStreamInfo()` * 2. `bbQueryStreamInfo()` returns the length as I/Q sample pairs. This function will need to be called ~73 times per second for any given decimation rate for the internal circular buffers not to fall behind. We recommend polling this function from a separate thread and not performing any other tasks on the polling thread to ensure the thread does not fall behind.

The buffer will be populated with alternating I/Q sample pairs scaled to mV. The time difference between each sample can be determined from the sample rate of the configured device.

The *triggers* parameter can be *null* if you are not interested in trigger position, otherwise *triggers* should point to an array of 64 32-bit integers. Starting at triggers[0], positive values will indicate positions within the returned *buffer* array where an external trigger occurred. The positions are zero based, meaning the positions will be between 0 and *bufferLen - 1*. (Note: the minimum trigger position detected is approximately 90) If no triggers occurred during the acquisition of the raw data, all values will be 0. If for example, 3 external triggers occurred during the acquisition, the first three values of the *triggers* array will be non-negative, and the remaining equal to 0. A returned trigger array might look like this.

triggerArray[64] = [917, 46440, 196264, 0, 0, …, 0];
This array indicates three external triggers were detected at *buffer[*917*], buffer[*46440*],* and *buffer[*196264*].* They will always be in increasing order.

Note: Trigger positions are relative to I/Q pairs, so a trigger position at 900 would refer to the I/Q pair at buffer[900*2] and buffer[900*2 + 1].

Note: The ports on a broadband device need to be configured to receive external triggers to take advantage of the trigger array.

See **Appendix: Code Examples: I/Q Streaming Example.**

Return Values

bbNoError	The device successfully began streaming.
bbDeviceNotOpenErr	*device* is not a handle to an open device.

bbDeviceNotConfiguredErr	The device has not been configured for retrieving raw data
bbNullPtrErr	This is returned if buffer is a null pointer.
bbDataBreak	Indicates that the data returned is not continuous with the data returned from the last call to bbFetchRaw. This occurs when three quarters of a second of data accumulates in the API. To avoid this warning, ensure your process is continually polling the API.
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

device	Handle of an initialized device.
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.

Return Values

bbNoError	Successful
bbNullPtrErr	If any pointer passed as a parameter is null, bbNullPtrErr will be returned and no values will be returned.
bbDeviceNotOpenErr	The *device* provided does not refer to an open device.
bbDeviceNotConfiguredErr	The device is not in a known operational state or is idle. This error will also be returned if the device is in BB_RAW_PIPE mode.

bbQueryRealTimeInfo

Query the frame size of the real-time frame

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

Parameters

device	Handle of an initialized device.
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.

Return Values

bbNoError	The function returned successfully.
bbDeviceNotOpenErr	The device specified is not open.
bbNullPtrErr	A pointer parameter supplied is equal to NULL.
bbDeviceNotConfiguredErr	The device specified is not configured for real-time mode.

bbQueryTimestamp

Retrieve an absolute time of a data packet

```
bbStatus bbQueryTimestamp(int device, unsigned int *seconds, unsigned int
*nanoseconds);
```

Parameters

device	Handle of an initialized device.
seconds	Seconds since midnight (00:00:00), January 1, 1970, coordinated universal time(UTC).
nanoseconds	nanoseconds between seconds and seconds + 1

Description

This function is used in conjunction with *bbSyncCPUtoGPS* and a GPS device to retrieve an absolute time for a data packet in raw pipe mode. This function returns an absolute time for the last packet retrieved from *bbFetchRaw.* See the Appendix:Code Examples for information on how to setup and interpret the time information.

Return Values

bbNoError	Successful
bbNullPtrErr	*seconds* or *nanoseconds* parameters are null.
bbDeviceNotOpenErr	*device* is not a handle to an open device.
bbDeviceNotConfiguredErr	The device is not configured and running in RAW_PIPE 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

device	Handle of the initialized device, which has been successfully initialized with the BB_STREAMING flag.
return_len	The number of I/Q samples pairs which will be returned by calling `bbFetchRaw()`.
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.

Description

Use this function to characterize the I/Q data stream.

See **Appendix: Code Examples: I/Q Streaming Example.**

Return Values

bbNoError	Function returned successfully.
bbDeviceNotOpenErr	*device* is not a handle to an open device.
bbDeviceNotConfiguredErr	The device specified is not currently streaming.

bbAbort

Stop the current mode of operation

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

Parameters

device	Handle of an initialized device.

Description

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

Return Values

bbNoError	The device has been successfully suspended.
bbDeviceNotOpenErr	The device indicated by *device* is not open.
bbDeviceNotConfiguredErr	The device is already idle.

bbPreset

Trigger a device reset

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

Parameters

device	Handle of an open 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.

Return Values

bbNoError	Function completed successfully, the device will be reset.
bbDeviceNotOpen	The device specified is not currently open.

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);`

Parameters

device	Handle of an open 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.

Return Values

bbNoError	The device was recalibrated successfully.

bbDeviceNotOpenErr	The device specified is either not open or valid.

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:

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:

bbNoError	Successful, describe what it means to be successful
bbDeviceNotOpenErr	No device is open at the time this function is called.
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, unsigned int *type);
```

Parameters

device	Handle of an open device.
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`, `BB_DEVICE_BB60C`, and `BB_DEVICE_BB124`. These values can be found in the bb_api header file.

Return Values

bbNoError	Successfully retrieved device type
bbDeviceNotOpenErr	The device specified is not open
bbNullPtrErr	The parameter *type* is null

bbGetSerialNumber

Retrieve the serial number of the device

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

Parameters

device	Handle of an open device.
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.

Return Values

bbNoError	Successfully retrieved the serial number. *sid* will contain the serial number.
bbDeviceNotOpenErr	The device specified is not open
bbNullPtrErr	The parameter *sid* is null

bbGetFirmwareVersion

Determine the firmware version of a SignalHound broadband device

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

Parameters

device	Handle of an open device.
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.

Return Values

bbNoError	Function returned successfully.
bbDeviceNotOpenErr	The device specified is not open.

bbNullPtrErr	The parameter *version* is null.

bbGetDeviceDiagnostics

Retrieve the current internal device characteristics

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

Parameters

device	Handle of an open device.
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.

Return Values

bbNoError	Successfully retrieved the temperature
bbDeviceNotOpenErr	Device specified is not currently open/valid

bbAttachTg

Pairs and open BB60C spectrum analyzer with a Signal Hound tracking generator

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

Parameters

device	Device handle.

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

bbNoError	The function returned successfully.
bbDeviceNotOpenErr	The device specified is not open.
bbNullPtrErr	The *attached* parameter is NULL.
bbNotSupportedErr	The devices specified is not a BB60C or the BB60C 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

device	Device handle.
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.

Return Values

bbNoError	The function returned successfully.
bbDeviceNotOpenErr	The device specified is not open.
bbNullPtrErr	The *attached* parameter is NULL.

bbConfigTgSweep

Configure a tracking generator sweep

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

Parameters

device	Device handle.
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).

Return Values

bbNoError	The function returned successfully.
bbDeviceNotOpenErr	The device specified is not open.
bbAdjustedParameter	Sweep size was clamped to the input range of [10, 1024].

bbStoreTgThru

Perform a store thru

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

Parameters

device	Device handle.
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.

Return Values

bbNoError	The function returned successfully.
bbDeviceNotOpenErr	The device specified is not open.
bbInvalidParameterErr	The *flag* parameter does not match any accepted value.
bbDeviceNotConfiguredErr	The device is not configured for tracking generator sweeps.

bbSetTg

Set the frequency and amplitude output of a paired tracking generator

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

Parameters

device	Device handle.
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

bbNoError	The function returned successfully.
bbDeviceNotOpenErr	The device specified is not open.
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.
bbAdjustedParameter	The frequency or amplitude was clamped to an upper or lower limit.

bbGetTgFreqAmpl

Retrieve the last set TG configuration

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

Parameters

device	Device handle.
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

bbNoError	The function returned successfully.
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 gauranteed to be a single decimal digit. The string is null terminated. An example string is below ..

['1' | '.' | '2' | '.' | '1' | '1' | '\0'] = "1.2.11"

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.

Error Handling

All API functions return the type *bbStatus*. *bbStatus* is an enumerated type representing the success of a given function call. The return values can be found in bb_api.h. There are three types of returned status codes.

1) No error : Represented with value bbNoError.
2) Error, interrupting function execution : Represented by a return value suffixed with "Err". All Error statuses are negative.
3) Warning : Each function may return a warning code. The system will still function but potentially in an undesirable state.

The best way to address issues is to check the return values of the API functions. An API function is provided to return a string representation of given status code for easy debugging.

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.

Appendix

Code Examples

This section contains some C examples for interacting with a device. Each example will have a short description describing the code in detail.

Common

All API functions return a status code responsible for reporting errors, warnings or success. It can be helpful to write a macro or function for checking these status codes.

```
#define CHECK_BB_STATUS(status)                      \
if(status != bbNoError) {                            \
        mylogErrorRoutine(bbGetErrorString(status)); \
        doErrorHandlingRoutine();                    \
    }
```

This macro can be used after each API call and can contain any error handling and reporting logic necessary. A macro such as this can clean up code, and keep logic in one location making error handling and reporting changes fast and easy.

Sweep Mode

Below is a simple example for configuring a BB60 to perform a single sweep. The example shows all the necessary functions needed to open, configure, initialize, get data from, and close the device.

```
int handle;
// Attempt to open a connected device
if(bbOpenDevice(&handle) != bbNoError) {
      myErrorRoutine("Device didn't open");
}

// Configure a sweep from 850MHz to 950MHz with an
//  RBW and VBW of 10kHz and an expected input of -20dBm or less
bbConfigureAcquisition(handle, BB_MIN_AND_MAX, BB_LOG_SCALE);
bbConfigureCenterSpan(handle, 900.0e6, 100.0e6);
bbConfigureLevel(handle, -20.0, BB_AUTO_ATTEN);
bbConfigureGain(handle, BB_AUTO_GAIN);
bbConfigureSweepCoupling(handle, 10.0e3, 10.0e3, 0.001, BB_NON_NATIVE_RBW,
BB_NO_SPUR_REJECT);
bbConfigureProcUnits(handle, BB_LOG);

// Configuration complete, initialize the device
if(bbInitiate(handle, BB_SWEEPING, 0)) {
      myErrorRoutine("Device couldn't initialize");
}

// Get sweep characteristics and allocate memory for sweep
unsigned int sweepSize;
double binSize, startFreq;
bbQueryTraceInfo(handle, &sweepSize, &binSize, &startFreq);

float *min = new float[sweepSize];
float *max = new float[sweepSize];

// Get one sweep
bbFetchTrace_32f(handle, sweepSize, min, max);

// Finished/close device
bbCloseDevice(handle);
```

Real-Time Mode

Below is an example of configuring and initializing the device for real-time mode. The example outlines the steps required to fully configure a BB60 device and retrieve the sweep size and frame size of a real-time session. One full sweep/frame is retrieved before closing the device.

```
int handle;
// Attempt to open a connected device
if(bbOpenDevice(&handle) != bbNoError) {
      myErrorRoutine("Device didn't open");
}
```

```
// Configure a 27MHz real-time stream at a 2.44GHz center
bbConfigureAcquisition(handle, BB_MIN_AND_MAX, BB_LOG_SCALE);
bbConfigureCenterSpan(handle, 2.44e9, 20.0e6);
bbConfigureLevel(handle, -20.0, BB_AUTO_ATTEN);
bbConfigureGain(handle, BB_AUTO_GAIN);
// 9.8kHz RBW, for real-time must specify a native BW value
bbConfigureSweepCoupling(handle, 9863.28125, 9863.28125, 0.001,
                         BB_NATIVE_RBW, BB_NO_SPUR_REJECT);
bbConfigureRealTime(handle, 100.0, 30);

// Initialize the device for real-time
if(bbInitiate(handle, BB_REAL_TIME, 0)) {
      myErrorRoutine("Device couldn't initialize");
}

// Get sweep characteristics and allocate memory for sweep
unsigned int sweepSize;
double binSize, startFreq;
bbQueryTraceInfo(handle, &sweepSize, &binSize, &startFreq);
int frameWidth, frameHeight;
bbQueryRealTimeInfo(handle, &frameWidth, &frameHeight);

float *sweep = new float[sweepSize];
float *frame = new float[frameWidth * frameHeight];

// Retrieve 1 real-time sweep/frame
bbFetchRealTimeFrame(handle, sweep, frame);

// Finished/close device
bbCloseDevice(handle);
```

I/Q Streaming Example

Below is the smallest example of using the BB60 and API for streaming I/Q data. The code snippet opens the device, fully configures the device for I/Q data streaming, initializes the device for streaming, retrieves one packet of data and then closes the device.

```
int handle;
bbOpenDevice(&handle);

// Configure an I/Q data stream
bbConfigureCenterSpan(handle, 2400.0e6, 20.0e6);
bbConfigureLevel(handle, -20.0 BB_AUTO_ATTEN);
bbConfigureGain(handle, BB_AUTO_GAIN);
// Set a sample rate of 20.0e6 MS/s and bandwidth of 15 MHz
bbConfigureIQ(handle, 2, 15.0e6);
// Initialize the device
bbInitiate(handle, BB_STREAMING, BB_STREAM_IQ);

// Allocate an array large enough for the I/Q stream configured
int buffer_len;
bbQueryStreamInfo(handle, &buffer_len, 0, 0);
float *iq_buffer;
// Allocate an array for ‘buffer_len’ alternating I/Q complex values
iq_buffer = new float[buffer_len * 2];

// Retrieve I/Q data packet
```

```
bbFetchRaw(handle, iq_buffer, 0);

// Done
bbCloseDevice(handle);
```

Another example below introduces the external trigger functionality. The code snippet is very similar to the one above, and shows the function call for configuring the BNC port to detect rising external triggers, and retrieving the triggers through the fetchRaw function.

```
int handle;
bbOpenDevice(&handle);

// Configure an I/Q data stream
bbConfigureCenterSpan(handle, 1000.0e6, 20.0e6);
bbConfigureLevel(handle, 0.0 BB_AUTO_ATTEN);
bbConfigureGain(handle, BB_AUTO_GAIN);
// Set a sample rate of 10.0e6 MS/s and bandwidth of 6 MHz
bbConfigureIQ(handle, 4, 6.0e6);
// Configure BNC port 2 for input rising edge trigger detection
bbConfigureIO(id, 0, BB_PORT2_IN_TRIGGER_RISING_EDGE);

// Initialize the device
bbInitiate(handle, BB_STREAMING, BB_STREAM_IQ);

// Allocate an array large enough for the I/Q stream configured
int buffer_len;
bbQueryStreamInfo(handle, &buffer_len, 0, 0);
float *iq_buffer;
// Allocate an array for 'buffer_len' alternating I/Q complex values
iq_buffer = new float[buffer_len * 2];
int trig_buffer[64];

// Retrieve I/Q data packet
bbFetchRaw(handle, iq_buffer, trig_buffer);

// Done
bbCloseDevice(handle);
```

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.

What you will need:

1) GPS Receiver capable providing NMEA data, specifically the GPRMC string, and a 1PPS output. (Tested with 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 data via `bbFetchRaw`. Calling the function bbQueryTimestamp will return the time of the first sample in the array of data collected from the last `bbFetchRaw`.
12) From the time retrieved, you can estimate the time of any sample knowing the difference in time between two samples is typically 12.5ns * decimation.

Code Example

Here we see a sample program following the steps mentioned above for setting up and retrieving time stamps for data.

```
// Open the device as usual
int handle;
bbOpenDevice(&handle);

// Configure an I/Q data stream as usual
bbConfigureCenterSpan(handle, 2400.0e6, 20.0e6);
bbConfigureLevel(handle, -20.0 BB_AUTO_ATTEN);
bbConfigureGain(handle, BB_AUTO_GAIN);
bbConfigureIQ(handle, 4, 6.0e6);

// Configure the device to accept input triggers on port 2
// The 1 PPS trigger will be connected to port 2
bbConfigureIO(handle, 0, BB_PORT2_IN_TRIGGER_RISING_EDGE);

// At this point the GPS receiver must be operational
// The RS232 connection cannot be open, and the COM port
//   and the baud rate must be known
// Ensure the receiver is locked
bbSyncCPUtoGPS(3, 38400);

// If syncCPUtoGPS returned successfully the device can now be initialized
//   and the RS232 connection should now be closed.
// Note: BB_TIME_STAMP is required so the device treats input triggers as the
//   GPS 1 PPS
bbInitiate(handle, BB_STREAMING, BB_STREAM_IQ | BB_TIME_STAMP);

// Allocate an array large enough for the I/Q stream configured
int buffer_len;
bbQueryStreamInfo(handle, &buffer_len, 0, 0);
float *iq_buffer;
iq_buffer = new float[buffer_len * 2];
```

```
int trig_buffer[64];

// Retrieve I/Q data packet
bbFetchRaw(handle, iq_buffer, trig_buffer);

// Retrieve the time of the first sample of the last packet retrieved
int seconds, nanoseconds;
bbQueryTimestamp(handle, &seconds, &nanoseconds);

// Done
bbCloseDevice(handle);
```

Additionally it may be helpful to write a function which determines the time of a single sample using the returned times from bbQueryTimestamp.

```
/*
  Retrieve the time of any sample in a packet
  To do this we need to know the starting time of the packet and
    the sample we are interested in
*/
void GetSampleTime(
  unsigned int startSeconds,    // In:  Seconds returned from QueryTimestamp
  unsigned int startNanos,      // In:  Nanoseconds returned from QueryTimestamp
  unsigned int sample,          // In:  Sample we are interested in, zero based
  unsigned int *sampleSeconds,  // Out: Seconds for interested sample
  unsigned int *sampleNanos )   // Out: Nanoseconds for interested sample
{
  // Amount of time between any two samples
  double delTime = 1.0 / 80000000;

  // Assuming zero based sample, get output nanos
  unsigned int outs = startSeconds;
  unsigned int outns = startNanos + delTime * sample;

  // If nanos are greater than 1 billion, then we wrap
  if( outns > 1000000000 ) {
    outs++;
    outns -= 1000000000;
  }

  *sampleSeconds = outs;
  *sampleNanos = outns;
}
```

Bandwidth Tables

In Native RBW mode, this table shows the possible RBWs and their corresponding FFT sizes. As of version 1.0.7 non-native bandwidths do not use this table. Non-native bandwidths can be arbitrary.

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

Non-Native RBWs and FFT size

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

```
int non_native_fft_from_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);
}
```