TEST EQUIPMENT PLUS

# Signal Hound Broadband Spectrum Analyzer Application Programming Interface(API)

## Programmers Reference Manual

Version 1.2.0

**Justin Crooks & Andrew Montgomery**

**10/31/2013**

Requirements, Operation, Function Definitions, Examples

# Table of Contents

# Overview

This manual is a programmer's reference for the Signal Hound Broadband API. The API is a C-based set of routines used to control the Signal Hound broadband devices.

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

The *Build/Version Notes* details each library build by Operating System.
The *Requirements* section details the physical and operational needs to use the API.
The *Theory of Operation* section details how to interface this 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 operation modes, from the required functions, to understanding the data the device returns.
The *API Functions* section details 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. |

# 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@teplus.com
All hardware/specification related questions should be directed to justin@teplus.com

You can also contact us via phone, 1-800-260-8378. Listen for the extensions for AJ or Justin.

## Build/Version Notes

Two Windows builds are available for 32 and 64 bit operating systems. The builds are compiled with Visual Studio 2012. Distributing an application using this library will require installing the VS2012 redistributable libraries.

A 64-bit Linux library is now available. The library is built on Ubuntu 12.04 using g++ version 2.6. The libusb libraries are required, they can be downloaded from the libusb website or installed via your package manager. libusb version 1.0 or greater is required.

## Requirements

**Windows Development Requirements**

Below is a list of requirements needed to begin.

1) Windows 7. The API is untested outside Windows 7 although has shown to be stable on Windows 8.
2) Windows C/C++ development tools/environment. Preferably Visual Studio 2008 or later.
3) The bb_api.h API header file.
4) The API library(.lib) and dynamic library(.dll) files.

**Linux Development Requirements**

1) libusb 1.0 or greater
2) The *bb_api* header and shared object file
3) Knowledge of linking shared libraries

**General Requirements**

1) A basic understanding of RF Spectrum Analysis.
2) A Signal Hound broadband spectrum analyzer.
3) Dual / Quad core Intel i-series processors, preferrably 2$^{nd}$ 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 modest hardware.

## Theory of Operation

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

1) Open a USB 3.0 connected device.

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 SCPI's *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. Once the device is connected it can be opened. The function bbOpenDevice() provides this functionality. This function returns an integer ID to the device which was opened. Up to eight devices may be connected and interfaced through our API using the ID's. The integer ID 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 device has 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 configure functions will modify the devices' global state. Device state is discussed more in the next section (Initiating the device). The API provides configure routines for groupings of related variables. Each function is described in depth in the API functions section. All relevant configuration routines should be invoked. Also note there are boundary conditions when configuring the device. Some variables will have different boundary conditions given the mode you will later be trying to operate in. Each function description will detail these boundaries. We have also provided helpful #defines in the API header file to help check against these boundaries. All configure routines will provide error/warning messages when these boundaries are crossed.

# Initiate the Device

Each device has two states.

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

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

The broadband spectrum analyzers have multiple modes of operation. The bbInitiate() function is used to enter one of the operational states. The device can only be in one operational state at a time. If

bbInitiate() is called on a device that is already running, the current state is aborted before entering the new specified state. These modes are described in the “Modes of Operation” section.

## Retrieve Data from the Device

Once a device has been successfully initiated you can begin retrieving data from the device. Every mode of operation returns different types and amounts of data. Some modes return a constant amount of data and some return varying sizes of data. The *Modes of Operation* section will help you determine how to collect data from the API. While this is the general flow of information for each operating mode, beware that each mode has its own nuances. Specific operational modes are detailed more in the *Modes of Operation* section.

## 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 abort 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 alongside 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 reduce power consumption.

## Close 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 devices are temperature sensitive and it is important a device is always 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, *bbQueryDiagnostics* and *bbSelfCal. bbQueryDiagnostics* 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 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.

## Modes of Operation

Now that we have seen how a typical application interfaces with the device, let's examine the different modes of operation a Signal Hound spectrum analyzer 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 broadband spectrum analyzer 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 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
- bbConfigureCenterSpan
- bbConfigureLevel
- bbConfigureGain
- bbConfigureSweepCoupling
- bbConfigureWindow
- bbConfigureProcUnits

Once you have configured the device, you will initiate it using the BB_SWEEPING mode.

This mode is driven by the programmer, causing a sweep to be collected only when the program requests one through th *bbFetchTrace* function. The length of the spectra and number of spectra used to compose the sweep are determined by a combination of resolution bandwidth, video bandwidth and sweep time.

Once the device is initiated you can determine the characteristics of the data you will be collecting with *bbQueryTraceInfo.* This function returns the length of the sweep, the frequency of the first sample and the bin size (difference in frequency between any two samples). You will need to allocate two arrays of memory, representing the Min and Max for each frequency point.

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 10 – 100 ms, 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 a 20MHz bandwidth. Through the use of FFTs at an overlapping rate of 75%, the spectrum results have no blind time (100% probability of intercept). Due to the demands in processing, restrictions are placed on resolution bandwidth, and video bandwidth is non-configurable.

The configuration routines which affect the spectrum results are

- bbConfigureAcquisition
- bbConfigureCenterSpan
    - Span must be less than or equal to 20MHz. Span restrictions are defined in the API header as BB_MIN/MAX_RT_RBW.
- bbConfigureLevel
- bbConfigureGain
- bbConfigureSweepCoupling
    - Restricted bandwidths, defined in the API header file as BB_MIN/MAX_RT_RBW.
    - Resolution bandwidths are restricted to the native values using the NUTALL window.
    - Video bandwidth is unconfigurable.

The number of spectrum results far exceeds a program's capability to acquire, view, and process, therefore the API combines results by min/max/averaging the spectrum density at each frequency bin for a specified amount of time. That time is determined by the *sweepTime* parameter in *bbConfigureSweepCoupling.*

Once configured, initiate the device in BB_REAL_TIME mode. The API immediately begins collecting spectrum. Use *bbQueryTraceInfo* to determine sweep characteristics and *bbFetchTrace* to collect sweeps. *bbFetchTrace* will block until data is ready. Because data is always being processed, the API uses a queue to store data until it is requested. It is possible for the queue to fill leading to a loss in data. Ensure your program can collect sweeps at the rate of *sweepTime* provided.

# Time-Gate Analysis

Time gate analysis allows you to capture a specific time slice of spectrum using external triggers. A *gate* represents the time you are interested in. An external trigger drives the capture of a *gate* of data. Signal analysis is performed on the gate similar to swept analysis mode. The user must specify the start of the gate relative to the trigger, and the length of the gate. The minimum gate length is dependent on your bandwidth settings. Using "native" bandwidths, with RBW = VBW, this is roughly 2.0 / RBW. *bbConfigureTimeGate* is used to characterize the *gate.*

Configuration is very similar to standard sweep mode with a few limitations. The maximum span allowed is 20MHz. The gate length must also be large enough to support the necessary processing. This means that your FFT size cannot be larger than the length of the gate. See *Appendix:Bandwidth Table* to determine FFT size. You also must specify a *timeout* period. This is the length of time the API will look for a trigger. If no trigger is found the final chunk of spectrum captured is used for analysis and *bbNoTriggerFound* warning is returned. It is also possible the trigger is found too late in the total capture not allowing enough room for the gate. This will also trigger the *bbNoTriggerFound* warning.

Sweeps characteristics are determined and acquired through the *bbQueryTraceInfo* and *bbQueryTrace* routines. *bbFetchTrace* looks for only one gate per function call.

Notes: Many of the boundary issues such as gates too small for processing, or gates being larger than the timeout period, are caught during *bbInitiate* and often a *bbInvalidParameter* error message is returned. For now, we leave the programmer to determine and assess gate characteristics. We encourage loosening specifications to assert proper functionality before "tightening" specs.

# Zero-Span

Zero-Span offers highly configurable AM and FM demodulation. Video and external triggers are available for synchronization as well as adjustable bandwidth and video filters.

Using zero-span, sweeps returned represent amplitude or frequency over time.

Configuration routines which affect sweeps are

- bbConfigureAcquisition
    - scale is necessary for amplitude demodulation
- bbConfigureCenterSpan
    - center frequency represents the frequency you want to view in the time domain, span is irrelevant here and is ignored.
- bbConfigureSweepCoupling
    - RBW represents the width of the bandpass filter located at the center frequency
    - VBW is the bandwidth of the filter performed on the demodulated trace
    - Sweep time is the length of the trace
- bbConfigureLevel
    - For AM and FM, reference level should be greater than your expected input signal, reference level helps determine the best gain and attenuation when using auto atten/gain.
- bbConfigureGain
- bbConfigureTrigger
    - Should be called even if you are using no trigger.
        - When using no trigger, trigger parameters are ignored.
    - Allows you to configure an external or video capture trigger or no trigger.
- bbConfigureIO
    - Applicable if utilizing external triggers

The *bbConfigureTrigger* routine can setup an external or video trigger. If an external trigger is desired you must also call *bbConfigureIO* to setup the BNC port for an external trigger.

When using triggers in this mode, you must specify a capture window (*timeout* paramter on bbConfigureTrigger). The capture window length specifies a continuous time period to look for a trigger before returning. If a trigger is found, a sweep is returned from the point of the trigger. If the trigger is found near the end of a capture window, there may not be enough data remaining in the window for a full sweep. In this event a trigger is still reported and the starting location is shifted so that a full sweep can be returned. To prevent this scenario from occuring frequently increase the length of the capture window.

Initiate the device with the BB_ZERO_SPAN value. Sweep characteristics are determined with the *bbQueryTraceInfo* function. Sweeps are retrieved with the *bbFetchTrace* function. *bbFetchTrace* is a blocking call, and can take up to multiple seconds depending on the settings used in *bbConfigureTrigger.*

# Raw Data

The API is capable of providing programmers with a continous stream of RF at 20 MHz and 7 MHz bandwidths. Both are unique in structure and are described below.

The device is capable of continuously streaming 20 MHz of IF bandwidth through the API. The IF stream is 16-bit signed ADC values, sampled at 80 MSPS. The API then scales the 16 bit values to 32-bit floats ranging from -1.0 to 1.0 representing the full scale. The 20 MHz of usable bandwidth is shown below in the frequency domain.

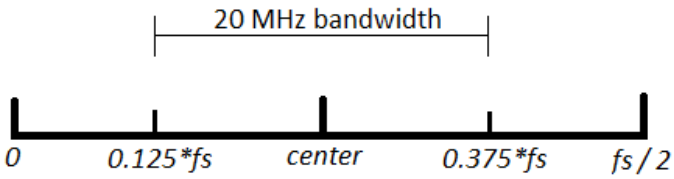



Figure 1 : 20MHz bandwidth shown in the spectrum domain.

Above is the typical result of a DFT on real data. *fs* is the sample rate, 80 million. A 20 MHz bandpass filter is centered on *center.* This leaves 20 MHz of usable spectrum centered on *center*. *center* is the frequency returned from *bbQueryStreamingCenter()* which can be called after an IF stream is opened.

7 MHz streaming was introduced to reduce the necessary hard drive solutions needed to save a 20 MHz bandwidth. 7 MHz streaming produces data to ~40 MB/s (if converted to signed shorts) and it becomes viable to save all data with no high speed disk solution. The data rate is also decreased by four and will decrease computational/analysis cost later when reviewing the data.

For the 7 MHz IQ data pipe the IF is downconverted to a baseband signal and filtered. The center frequency must be a multiple of 1kHz. The data returned is alternating IQ values at a rate of 10 million IQ pairs per second representing a bandwidth of 10 MHz. The 7 MHz of usable bandwidth is centered. The values returned are 32-bit floats which range from -1.0 to 1.0 representing the full scale.

Configuration routines used to prepare streaming are

- bbConfigureCenterSpan*()*
  - A center frequency which is a multiple of 400kHz is best for minimal spurious signals.
  - Span is ignored
- bbConfigureLevel*()*
  - See below for discussion on attenuation and reference level
- bbConfigureGain*()*
  - See below for discussion on gain

## 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. By default the data

returned in the streaming data mode is not amplitude corrected and gain and attenuation need to be manually chosen. When gain and attenuation are chosen, reference level is ignored.

Automatic gain and attenuation should only be used when you plan on utilizing the amplitude corrections retrieved from *bbFetchRawCorrections.* This is because automatic gain and attenuation are chosen for best dynamic range *after* corrections are applied and based on your suggested reference level. To learn more about how to perform amplitude corrections please see *bbFetchRawCorrections.*

Once configured, initiate the device in BB_RAW_PIPE mode with the proper bandwidth flag.(See *bbInitiate)* Data acquisition begins immediately. Use the *bbFetchRaw()* function to collect 299008 samples. This function is blocking when data is not ready. With a 20 MHz bandwidth, *bbFetchRaw* must be called approximately 267 times per second. Using the 7 MHz bandwidth, *bbFetchRaw* must be called approximately 67 times per second. The API queues up to 120ms of data in both bandwidths, before the queue becomes full and drops data. It is very important to have minimal processing in the data collection thread to prevent this from happening.

PLEASE NOTE: The downconversion from 20 MHz to 7 MHz is processor intensive, as it is performed entirely in software. Please characterize this processor load before attempting to perform any real-time analysis.

If you want to convert the values returned from *bbFetchRaw()* to a more space efficient data type, we recommend signed shorts, scaling the returned values by 32768.

Raw data mode 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).*

# Raw Sweep

Raw sweep mode is similar to regular sweep mode except the API performs no signal processing and just returns the ADC data. Essentially, this mode will take 20MHz steps while streaming. You will specify a starting center frequency, a step count, and a collection amount at each frequency. The device will collect 'n' samples at each step, and return the results in one large array. The data collected is the same type of data collected in the raw data mode (See above) except in this mode, signed shorts are returned instead of 32-bit floating point values.

You must configure the sweep via the *bbConfigureRawSweep* functions. There are various conditions that must be met to be a valid sweep. Amplitude corrections are not made on the data, so absolute amplitude cannot be determined.

Once configured, initiate the device in BB_RAW_SWEEP mode. You can begin collecting sweeps with the *bbFetchRawSweep* function. Sweeps sizes are determined from parameters used to configure the sweep. Sweeps are collected on demand, one per fetch. The data returned is in signed short format ranging from -32768 to 32768 where 32768 is the maximum possible digital level and 6dB below full scale. It is useful to adjust gain and attenuation to find the best dynamic range.

# Raw Sweep Loop

Raw sweep loop is similar to raw sweep except instead of returning traces on request, the device sweeps the spectrum as configured indefinitely. This eliminates the software setup time at the beginning of each sweep, and can drastically improve sweep speed for sweeps under 50 ms.

The device is configured similarly to raw sweep mode. The gain and attenuation must be chosen. (Gain cannot be auto) The device must be initated using BB_RAW_SWEEP_LOOP as the mode. After configuration the device is still idling until *bbStartRawSweepLoop* is called. Read the entry on *bbStartRawSweepLoop* to understand how data is collected in this mode.

Requesting the smallest dwell time at each step this mode is capable of sweeping ~25 GHz per second. Despite the name, this mode is in fact streaming data continuously, and incurs the ½ second abort time as described in the section *Sweeping versus Streaming.*

# 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.

## Sweeping versus Streaming

All modes of operation fall within two categories, sweeping and streaming. In any sweeping mode, the device operates only when requested. Usually requesting a trace triggers a single trace acquisition, otherwise the device and API are idle. Sweeping is very responsive and switching between difference types of sweep modes is very quick. Streaming modes are modes in which the API is continually receiving a stream of spectrum (20MHz) from the device. The characterization of the data is a result of the initial configuration. 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 stream mode is 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 or the raw sweep mode. From these modes you can simply detect the location of a signal of interest and quickly react by switching into a stream mode with appropriate settings.

## API Functions

# 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 find the first unopened broadband device and initalize it for use.

This function when successful, takes about 3 seconds to perform. The API maintains a list of opened devices by serial number. Therefore, calling *bbOpenDevice()* a second time will do nothing, assuming only one device is connected. The only way to remove a serial number from the open device list is to call *bbCloseDevice()* with the proper device ID or end the program execution.

This function must be called before any other calls are made to the device. Attempting to interface a device that is not open will return *bbDeviceNotOpenErr* errors. This function at most opens a single device. The *device* parameter returned will always be a value between 0-7. This value must be saved as it is used to for all subsequent API calls.

### Return Values

**bbNoError** — No error, device number opeened 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.

# 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

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 millivolts. If the full scale units are specified, no corrections are applied to the data and amplitudes are taken directly from the full scale input.

*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.

## Return Values

| | |
|---|---|
| **bbNoError** | No error. |
| **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.

Certain modes of operation have specific frequency range limits. Those mode dependent limits are tested against during initialization 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 bbConfigureCenterSpan(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 *gain* is selected, the API uses the *reference level* provided to choose the best gain settings for an input signal with amplitude equal to reference level. If a gain other than BB_AUTO_GAIN is specified using *bbConfigureGain*, the *reference level* 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. A set attenuation may produce a non-flat noise floor.

### 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

*Change the sweep coupling values*

```
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** | Sweep time in seconds.<br><br>In sweep mode, this value is how long the device collects data before it begins processing. Maximum values to be provided should be around 100ms.<br><br>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 for optimal viewing and use.<br><br>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, BB_SPUR_REJECT, and BB_BYPASS_RF. |

## 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_RBWs, 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 |

# bbConfigureTrigger

*Configure the Zero-Span trigger*

```
bbStatus bbConfigureTrigger(int device, unsigned int type, unsigned int edge, double
level, double timeout);
```

### Parameters

| | |
|---|---|
| **device** | Handle to the device being configured. |
| **type** | Specifies the type of trigger to use. Possible values are BB_NO_TRIGGER, BB_VIDEO_TRIGGER, BB_EXTERNAL_TRIGGER, and BB_GPS_PPS_TRIGGER. If an external signal is desired, BNC port 2 must be configured to accept a trigger (see *bbConfigureIO*). When BB_NO_TRIGGER is specified, the other parameters are ignored and this function sets only trigger type. |
| **edge** | Specifies the edge type of a video trigger. Possible values are BB_TRIGGER_RISING and BB_TRIGGER_FALLING. If you are using a trigger type other than a video trigger, this value is ignored but must be specified. |
| **level** | Level of the video trigger. The units of this value are determined by the demodulation type used when initiating the device. If demodulating AM, *level* is in dBm units, if demodulating FM, *level* is in Hz. |
| **timeout** | *timeout* specifies the length of a capture window in seconds. The capture window specifies the length of continuous time you wish to wait for a trigger. If no trigger is found within the window, the last *sweepTime* of data within the data is returned. The capture window must be greater than *sweepTime*. If it is not, it will be automatically adjusted to *sweepTime.* The *timeout/capture* window is applicable to both video and external triggering. |

### Description

Allows you to configure all zero-span trigger related variables. As with all configure routines, the changes made here are not reflected until the next initiate.

When a trigger is specified the sweep returned will start approximately 200 microseconds before the trigger event. This provide a slight view of occurances directly before the event. If no trigger event is found, the data returned at the end of the timeout period is returned.

### Return Values

| | |
|---|---|
| **bbNoError** | Configured successfully |
| **bbDeviceNotOpenErr** | The device specified is not open. |
| **bbInvalidParameterErr** | A parameter specified is not valid. |

# bbConfigureTimeGate

*Configure gate properties*

```
bbStatus bbConfigureTimeGate (int device, double delay, double length, double
timeout);
```

### Parameters

| | |
|---|---|
| **device** | Handle to the device being configured. |

| | |
|---|---|
| **delay** | The time in seconds, from the trigger to the beginning of the gate |
| **length** | The length in seconds, of the gate |
| **timeout** | The time in seconds to wait for a trigger. If no trigger is found, the last *length* will be used. |

### Description

Time gates are relative to an external trigger.

Therefore it is necessary to use *bbConfigureIO* to setup an external trigger.

### Return Values

| | |
|---|---|
| **bbNoError** | Device configured successfully |
| **bbDeviceNotOpenErr** | Device specified is not open. |
| **bbInvalidParameterErr** | A supplied parameter is unknown or out of range. |

## bbConfigureRawSweep

*Prepare the device to collect swept ADC data.*

```
bbStatus bbConfigureRawSweep(int device, int start, int ppf, int steps, int stepsize);
```

### Parameters

| | |
|---|---|
| **device** | Handle to the device being configured. |
| **start** | Frequency value in MHz representing the center of the first 20MHz step in the sweep. |
| **ppf** | Controls the amount of data to collect at each frequency. The number of samples at each frequency equals 18688 * ppf. |
| **steps** | Number of steps to take in the sweep |
| **stepsize** | Value must be BB_TWENTY_MHZ |

### Description

This function configures the sweep returned in BB_RAW_SWEEP mode. Read the restrictions below and view the example in the Appendix for further information.

### Restrictions

1) The first center frequency (*start* parameter) must be a multiple of 20MHz. This helps to reduce and eliminate the majority of spurious responses. This values should be a minimum of 20MHz.
2) *ppf * steps* must be a multiple of 16.
3) The final center frequency, obtained by the equation *(start + steps*20),* cannot be greater than 6000(6 GHz).

### Return Values

| | |
|---|---|
| **bbNoError** | The device was successfully configured. |
| **bbDeviceNotOpenErr** | The device specified is not open. |
| **bbInvalidParameter** | One or more of the functions requirements were not met. |

# 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_PORT_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.

```
1.  bbConfigureIO (
2.    myDeviceNumber,
3.    BB_PORT1_AC_COUPLED | BB_PORT1_EXT_REF_IN, // Catch AC external reference
4.    BB_PORT2_OUT_LOGIC_HIGH                    // Output DC logic high
5.  );
```

# 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 2kHz and 500kHz.

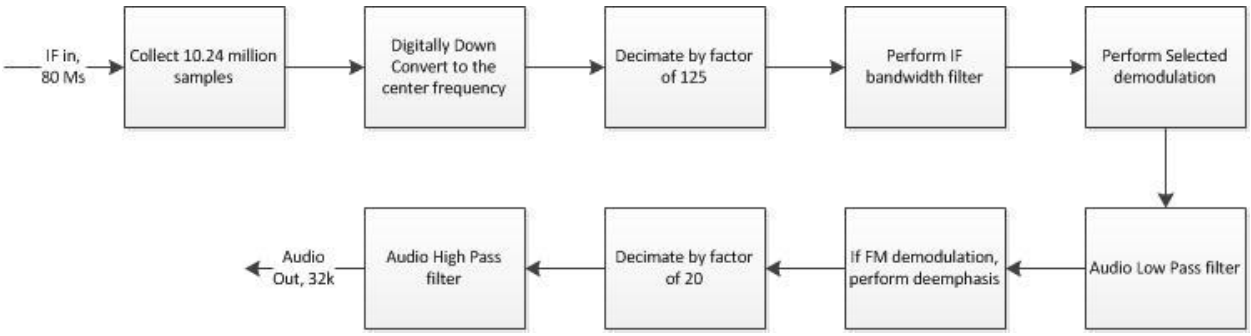
**audioLowPassFreq** Post demodulation filter in Hz. Should be between 1kHz and 12kHz Hz.

**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.




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.

# 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_ZERO_SPAN, BB_TIME_GATE, BB_RAW_SWEEP, BB_RAW_SWEEP_LOOP, BB_AUDIO_DEMOD, and BB_RAW_PIPE.

**flag** The default value is zero.

If mode equals BB_ZERO_SPAN, *flag* can be used to denote the type of modulation performed on the incoming signal. BB_DEMOD_AM and BB_DEMOD_FM are the two options.

If mode equals BB_RAW_PIPE, *flag* is used to denote the retreived bandwidth. BB_SEVEN_MHZ or BB_TWENTY_MHZ is used to change the desired capture bandwidth. *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.

Pay special attention to the bbInvalidParameterErr description below.

### 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.<br><br>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.<br><br>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 in extreme circumstances. The API currently limits the amount of RAM usage to 1GB. When exceptional parameters are provided, such as very low bandwidths, or long sweep times, this error may be returned. At this point you have reached the boundaries of the device. The processing algorithms are optimized for speed at the expense of space, which is the reason this can occur. |
| **bbBandwidthErr** | This error is returned if your RBW is larger than your span. (Sweep Mode) |

# 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. |

## bbQueryStreamingCenter

*Get the center frequency of a streaming device*

```
bbStatus bbQueryStreamingCenter(int device, double *center);
```

### Parameters

| | |
|---|---|
| **device** | Handle of an initialized device. |
| **center** | Pointer to a double which will receive the absolute center frequency of the streaming device. |

### Description

The function retrieves the center frequency of the 20 MHz IF bandwidth of a device currently initialized in raw pipe mode. The *center* returned is representative of ¼ of the IF sample rate. The 20 MHz of usable bandwidth is centered on this frequency.

### Return Values

| | |
|---|---|
| **bbNoError** | No error. |
| **bbNullPtrErr** | Pointer to *center* is null. |
| **bbDeviceNotOpenErr** | *device* is not a handle to an open device. |
| **bbDeviceNotConfiguredErr** | The device is not configured in the 20 MHz IF streaming mode. |

## bbFetchTrace

*Get  one sweep from a configured and initiated device*

```
bbStatus bbFetchTrace(int device, int arraySize, double *min, double *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. |
| **bbBufferTooSmallErr** | The *arraySize* parameter passed is less than the trace size returned from *bbQueryTraceInfo.* |
| **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. |
| **bbNoTriggerFound** | In time-gated analysis, if the spectrum returned is not representative of the gate specified, this warning is returned.<br><br>In zero-span analysis, if the device is configured to anticipate a video or external trigger, this warning is returned when the trigger condition has not been met for this trace. |
| **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. |

# 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 |

## bbFetchRawCorrections

*Retrieve information needed to make amplitude corrections on data collected in the raw-pipe mode*

```
bbStatus bbFetchRawCorrections(int device, float *corrections, int *index, double
*startFreq);
```

### Parameters

| | |
|---|---|
| **device** | Handle of an initialized device. |
| **corrections** | 32-bit float array of length 2048. Correction values are decibel. |
| **index** | Index into the *corrections* array where the correction data begins. |
| **startFreq** | Frequency associated with the correction at *index.* |

### Description

When this function returns successfully, the correction array will contain the frequency domain correction constants for the given bandwidth chosen. The corrections are modified based on temperature, gain, attenuation, and frequency. If any of these change, a new correction array should be requested. The correction array will only be generated again on a new *bbInitiate()*.

The correction arrays and returned values differ slightly depending on the 7 or 20 MHz bandwidth chosen. Each one is described in depth below.

**20 MHz**

The correction array represents 40 MHz of bandwidth where frequencies outside the requested 20 MHz are zeroed out. The first non-zero sample begins at *corrections[index].* The frequency at this index is *startFreq.* The bin size of each index is implied through 40 MHz divided by the length of the array,

(40.0e6 / 2048) = 19531.25 Hz. If an Fourier transform is applied on the IF data, the correction values will line up with the usable 20 MHz bandwdith.

**7MHz**
The correction array represents 10 Mhz of bandwdith where the usable 7 MHz is centered and all values outside the usable 7 MHz is zeroed. The *index* returned is the first non zero sample in the array. The *startFreq* returned is the frequency of the first sample in the array, *corrections[0]*. Every other sample's frequency can be determined with the bin size. The bin size for this array is (10.0e6 / 2048) = 4882.8125 Hz. If a complex Fourier Transform is applied to the IQ data, the correction values will line up with the usable 7 MHz bandwidth.

**Tips**
Time domain corrections of the signal's amplitude require two steps. First, an inverse Fourier Transform must be performed on the entire correction array (including zero'ed portions). This results in a 4096 sample kernel. Second, the kernel is used in convolution with the time domain data. If a larger/smaller kernel is desired, interpolate/extrapolate the correction array while it is in the frequency domain to the desired length. Lengths which are powers of two are suggested.

Frequency domain correction of the signal's amplitude requires you to first transform the raw data into the frequency domain. Performing an Fourier transform on the incoming data will yeild a frequency domain array that will align with the correction array. You can index the Transform results using the *index* returned from this function if you wish or apply the whole array. Remember that the corrections are in dB. If larger Transform sizes are desired, you can interpolate the correction array to the desired size. (Be aware! This will change the index of the first non-zero correction, but the results of the FFT will still align the with usable 20 MHz)

### Return Values

| | |
|---|---|
| **bbNoError** | Function returned successfully |
| **bbNullPtrErr** | One or more parameters provided is null |
| **bbDeviceNotOpenErr** | The device specified is not open |
| **bbDeviceNotConfigured** | The device specified is not currently configured in *raw-pipe* mode |

## bbFetchRaw

*Retrieve raw data from a streaming device*

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

### *Parameters*

| | |
|---|---|
| **device** | Handle of an initialized device. |
| **buffer** | A pointer to an array of 32-bit floating point values of length 299008. |
| **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

Fetch a chunk of data while the device is in the raw data pipe mode. A buffer will data represent either 299008/80,000,000 or 299008/20,000,000 seconds worth of time depending on the bandwidth selected at initiate. The structure of the data is described in *Modes of Operation:Raw Data.* To ensure full coverage the function must be called ~267 times with a 20 MHz bandwidth and ~67 with 7 MHz bandwidth. Only 120ms of data is buffered before data loss occurs. Ensure no additional processing or disk I/O is occuring in the same thread as the one acquiring the data to ensure no data loss.

The values in buffer range from -1.0 to 1.0. A value of 1.0 or -1.0 represents full scale. Values at or near 1.0 might indicate compression. Adjusting gain and attenuation is the best way to achieve the best dynamic range of values returned.

The *triggers* parameter can be NULL if you are not interested in trigger position, otherwise *triggers* should point to an array of 68 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 ~90 and 299007. (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.

The filters used to downconvert the 20 MHz IF to 7 MHz IQ cause the data to incur a 3.3 micro second delay which is accounted for in the trigger index values.

triggerArray[68] = [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: The ports on a broadband device need to be configured to receive external triggers to take advantage of the trigger array.

See the Code Examples for an example of this function.

The values returned are in the time domain and are uncorrected. (*See bbFetchRawCorrections() for information on making amplitude corrections on the raw data)*

## 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. |
| **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. |

# bbFetchRawSweep

*Retrieve a single sweep in the raw sweep mode*

```
bbStatus bbFetchRawSweep(int device, short *buffer);
```

## Parameters

| | |
|---|---|
| **device** | Handle of an initialized device. |
| **buffer** | Pointer to an array of signed short integers |

## Description

This function is used to collect a single sweep for a device configured in raw sweep mode. The length of the buffer provided is determined by the settings used to configure the device for raw sweep mode. This length can be determined using the equation.

$$Buffer\ Length = 18688 * ppf * steps$$

If the function returns successfully the array will contain a full sweep. The shorts will

## Return Values

| | |
|---|---|
| **bbNoError** | Function returned successfully. Buffer contains the entire sweep. |
| **bbNullPtrErr** | *buffer* is null |
| **bbDeviceNotOpenErr** | Device specified is not open |
| **bbDeviceNotConfiguredErr** | Device specified is not configured and initiated for raw sweeps. |
| **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. |

# bbStartRawSweepLoop

*Begin the raw sweep loop*

```
bbStatus bbStartRawSweepLoop(int device, short *buffer, int bufferLength, int
*bufferIndex, int *bufferCounter);
```

## Parameters

| | |
|---|---|
| **device** | Handle of an initialized device. |
| **buffer** | Pointer to an array of signed shorts. *Buffer* length specified by *bufferLength* |

| | |
|---|---|
| **bufferLength** | The number of shorts *buffer*. This number must be a multiple of 299008. |
| **bufferIndex** | Pointer to an unsigned integer. Will be updated to reflect the next index where the API will write data. |
| **bufferCounter** | Pointer to an unsigned integer. Will be incremented each time the buffer is filled. |

### Description

This function can be called after being configured for raw sweeps through *bbConfigureRawSweep* and initiated for the raw sweep loop mode. The function begins the sweep loops.

If this function returns successfully, the device begins sweeping immediately. *bufferIndex* and *bufferCounter* will be set to zero to begin. *buffer* will be filled with the incoming sweeps. When the end of the buffer is reached, the buffer is then filled from the beginning. The buffer length must be a multiple of 299008 shorts in length, which is the smallest transfer size.

Once started *bufferIndex* will be updated to represent where the next packet of data will be written. All data up to *buffer[bufferIndex]* are valid samples. *bufferCounter* is incremented each time the buffer has been filled. It is the responsibility of the programmer to maintain which samples relate to which portions of the spectrum. This is known to the programmer by the variables used in *bbConfigureRawSweep*. The length of the buffer does not have to be a multiple of the sweep size. If the *buffer* length is not a multiple of the sweep size, different portions of the *buffer* will correspond to different portions of the sweep depending on *bufferCounter.* It is recommended that the *buffer* provided be the largest possible due to the speed in which data is collected. Depending on the sweep characteristics, the user can expect to receive anywhere from 90 to 267 packets(299008 shorts) per second.

The location and length of *buffer* should not be modified until the device has stopped operation.

The device sweeps indefinitely until *bbAbort* or *bbCloseDevice* is called. When operation is suspended via *bbAbort,* the device must be reconfigured and initiated again before calling this function.

### Return Values

| | |
|---|---|
| **bbNoError** | The function returned successfully and is now sweeping. |
| **bbNullPtrErr** | Returned if either *buffer, bufferIndex,* or *bufferCounter* is null. |
| **bbDeviceNotOpenErr** | The device specified is not open. |
| **bbDeviceNotConfiguredErr** | The device has not be configured and initiated for raw sweep loop mode. |
| **bbDeviceAlreadyStreamingErr** | The device is already sweeping, indicating this function has already been called. |

# 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. |

## 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. |

# 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;
}
```

# bbQueryDiagnostics

*Retrieve the current internal device characteristics*

```
bbStatus bbQueryDiagnostics(int device, float *temperature, float *voltage1_8, float
*voltage1_2, float *voltageUSB, float *currentUSB);
```

### Parameters

| | |
|---|---|
| **device** | Handle of an open device. |
| **temperature** | Pointer to 32bit 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. |
| **voltage1_8** | Factory use only: Internal regulator. |
| **voltage1_2** | Factory use only: Internal regulator. |
| **voltageUSB** | USB operating voltage, in volts. Acceptable ranges are 4.40 to 5.25 V. |
| **currentUSB** | USB current draw, in mA. Acceptable ranges are 800 – 1000 mA. |

### 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 1/8$^{th}$ of a degree. A temperature above 70°C or below 0°C indicates your device is operating outside of its normal operating temperature, and may cause readings to be out of spec, and may damage the device.

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 |

# bbSelfCal

*Calibrate the device for significant temperature changes.*

```
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 *bbQueryDiagnostics* 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. |

# 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_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.

## bbGetAPIVersion

*Get 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.

## Appendix

# Code Examples

This sections 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 inline function for checking these status codes.

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

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

This example shows you how to set up the device for standard spectrum sweeps. The example begins by defining common variables used in the process. A *bbStatus* variable is used to catch warning or errors on some of the more important API calls such as opening the device, initiating the device, and retrieving sweeps. *devID* is used to store the handle to our open device. The remaining variables are used to define the characteristics of the sweeps returned from the device.

```
/* Open a device, configure it, and retrieve a sweep */

bbStatus status;
int devID, traceSize;
double *min, *max, binSize, startFreq;

/* Open the device, retrieve the device handle */
```

```
status = bbOpenDevice( &devID );
if( status != bbNoError )
  myErrorRoutine( status );

/* Simple configuration */

bbConfigureAcquisition(
  devID,
  BB_MIN_AND_MAX,
  BB_LOG_SCALE  // Log scaled results
);

bbConfigureCenterSpan(
  devID,
  900.0e6, // 900 MHz center
  20.0e6  // 20 MHz span
);

bbConfigureLevel(
  devID,
  0.0, // 0 db Reference level
  BB_AUTO_ATTEN // Automatically choose attenuation
);

bbConfigureGain(
  devID,
  BB_AUTO_GAIN
);

bbConfigureSweepCoupling(
  devID,
  10.0e3, // 10 kHz rbw
  10.0e3, // 10 kHz vbw
  0.001,  // 1 ms sweep acquisition
  BB_NATIVE_RBW, // Use native rbw
  BB_NO_SPUR_REJECT // No software spur rejection
);

bbConfigureWindow(
  devID,
  BB_BLACKMAN // Blackman windowing function
);

bbConfigureProcUnits(
  devID,
  BB_LOG // Video processing performed in logarithmic scale
);

status = bbInitiate(
  devID,
  BB_SWEEPING // Sweep mode
  0  // Use zero if not in zero-span mode
);

if( status != bbNoError )
  myErrorRoutine( status );

// Device initiated, get sweep information
bbQueryTraceInfo(
  devID,
  &traceSize, // Get trace size
```

```
  &binSize,   // Get freq per returned sample
  &startFreq  // Get accurate start frequency
);

min = new double[traceSize];
max = new double[traceSize];

// Continually fetch sweep information
while( yourProgramIsRunning ) {

  bbFetchTrace(
    devID,
    traceSize,
    min,
    max
  );

  displayTrace( min, max ); // Your custom routine

} /* while(programRunning) */


// Your custom error handling routine
void myErrorRoutine( bbStatus code ) {

  cerr << bbGetErrorString( code );
  handleError( code );

}
```

## Raw Data Pipe Example

This code snippet shows how you would open a device and retrieve raw data values. For brevity, error checking is left out. The configuration is much simplified due to no signal processing or corrections being performed on the data. The only configurations which modify the output are gain and attenuation. Remember that in the raw data pipe mode, the device produces samples at a rate of 80 million per second! This means that keeping up with the flow of data requires calling bbFetchRaw ~267 times per second! Any processing or data saving done in the same thread must be done quickly (~3ms) if you want no gaps in the data. The API accumulates 120ms of data before data loss happens.

```
/* Retrieving raw ADC values */
/* Also shows how to retrieve external trigger occurrences
   from the returned trigger array */

int devID;
float buf = new float[BB_RAW_PACKET_SIZE];
int triggers[68];               // From bbFetchRaw
std::vector<int> triggerList;   // Just triggers

bbOpenDevice( &devID );

bbConfigureCenterSpan(
  devID,
  2500.0e9,  // 2.5 GHz center
  20.0e6     // 20 MHz span is default for raw-pipe mode
);

bbConfigureLevel(
  devID,
```

```
  0.0,   // not applicable if gain is selected
  10.0   // 10 db attenuation
);

bbConfigureGain(
  devID,
  1      // low gain
);

bbConfigureIO(
  devID,
  0,                                  // default
  BB_PORT2_IN_TRIGGER_RISING_EDGE     // Catch rising edges
);

bbInitiate(
  devID,
  BB_RAW_PIPE,
  BB_TWENTY_MHZ  // Could also be BB_SEVEN_MHZ
);

while( streamingData ) {

  bbFetchRaw(
    devID,
    buf,           // spectrum
    triggers       // triggers
  );

  // Extract just the trigger positions into a vector
  int i = 0;
  while(triggers[i]) {
    triggerList.push_back(triggers[i]);
    i++;
  }

  doSomethingWithData( buf, triggerList ); // Save/process/display

} // while(streaming)
```

## Zero-Span Triggering

This code snippet shows you how to initialize and configure the device for zero-span mode with an external trigger. For brevity, error handling and unrelated configuration API calls are left out.

```
/* Configure Zero-Span and an external trigger */

int devID;

bbOpenDevice( &devID );

//
// Other configurations here
//
// bbConfigureAcquisition
// bbConfigureCenterSpan
// bbConfigureLevel
// bbConfigureSweepCoupling
//

// Configure our trigger, tell the device to use an external trigger
```

```
bbConfigureTrigger(
  devID,
  BB_EXTERNAL_TRIGGER,
  0,            // n/a for external trigger
  0,            // n/a for external trigger
  0.032         // wait up to 32 ms for trigger
);

// The device now expects an external trigger, we configure port 2 for a trigger
bbConfigureIO(
  devID,
  0,                                // Not using port1, 0 is default
  BB_PORT2_IN_TRIGGER_RISING_EDGE  // trigger on rising edge
);

bbInitiate(
  devID,
  BB_ZERO_SPAN, // Zero-Span mode
  BB_DEMOD_FM   // FM demodulation
);

/*
 The device is now ready for an external trigger.
 From here, we would get our trace information, and
 begin getting sweeps.
*/
```

## Raw Sweep Example

The example below shows how you can prepare the device to perform raw sweeps.

```
bbConfigureLevel(
  id,
  my_ref_level, /* Just above the max expected input */
  BB_AUTO_ATTEN
);

bbConfigureGain(
  id,
  BB_AUTO_GAIN
);

bbConfigureRawSweep(
  id,
  20,           /* First center at 20MHz */
  2,            /* 18688 * 2 samples at each step */
  16,           /* Take 16 steps */
  BB_TWENTY_MHZ /* Forced parameter */
);

/* Allocate enough samples for the full sweep */
short *sweep = new short[18688 * 2 * 16];

/* Setup the device */
bbInitiate(id, BB_RAW_SWEEP, 0);

/* Get one sweep */
bbFetchRawSweep(id, sweep);

/* 'sweep' variable now contains 16 20MHz streams
   with center frequencies of 20, 40, 60, … , 320 (MHz)
```

```
    Each 20MHz stream contains 18688*2 samples
    The stream format is the same IF format described
    in the raw data mode
*/
```

# 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 using the device in the "raw pipe" mode.

What's needed:

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 reciever.
2) Connect the 1PPS reciever output to port 2 of the device.
3) Connect the RS232 reciever output to your PC.
4) Determine the COM port number and baud rate of the data transfer over RS232.
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 raw pipe mode.
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_RAW_PIPE, 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 or 1/80000000.

### Code Example

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

```
/* Configure and prepare the device for time stamping */

int id;
float *data = new float[299008];

// Open Device as usual
bbOpenDevice( &id );

```

```
// Configuration
bbConfigureCenterSpan( id, 900.0e6, 20.0e6 );
bbConfigureLevel( id, 0, 10 );
bbConfigureGain( id, BB_HIGH_GAIN );

// The device MUST be ready to accept input triggers on port 2
// The 1 PPS trigger will be connected to port 2
bbConfigureIO( id, 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 baud rate
//   must be known
// Ensure the receiver is "locked"
bbSyncCPUtoGPS( 3, 38400 );

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

// We can now retrieve data
while( programRunning ) {

  int seconds, nanoseconds;
  char *timeString;

  // If we wanted we could collect the 1PPS triggers here
  bbFetchRaw(
    id,
    data,   // Collect one raw packet
    NULL    // Not interested in the triggers,
  );

  // Return the seconds and nanoseconds of the first sample in the last packet
  //   retrieved.
  bbQueryTimestamp( id, &seconds, &nanoseconds );

  // Function in <ctime> which returns a human readable string of the date/time
  timeString = ctime( (time_t*)(&seconds) );

  doSomething();
}
```

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
```

```
14.   double delTime = 1.0 / 80000000;
15.
16.   // Assuming zero based sample, get output nanos
17.   unsigned int outs = startSeconds;
18.   unsigned int outns = startNanos + delTime * sample;
19.
20.   // If nanos are greater than 1 billion, then we wrap
21.   if( outns > 1000000000 ) {
22.     outs++;
23.     outns -= 1000000000;
24.   }
25.
26.   *sampleSeconds = outs;
27.   *sampleNanos = outns;
28. }
```

# 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.

```
1.  int non_native_fft_from_rbw(double rbw)
2.  {
3.      double min_bin_sz = rbw / 3.2;
4.      double min_fft = 80.0e6 / min_bin_sz;
5.      int order = (int)ceil(log2(min_fft));
6.
7.      return pow2(order);
8.  }
```