BB API

Generated by Doxygen 1.9.4

Chapter 1

BB60 API Reference

This manual is a reference for the Signal Hound BB60 application programming interface (API). This API supports the `BB60D`, `BB60C`, and BB60A Signal Hound devices. The API provides a set of C functions for making measurements with the BB60 devices. The API is C ABI compatible making is possible to be interfaced from most programming languages. See the code examples in the SDK for examples of calling the API from C++ and other programming languages and environments.

1.1 Examples

All code examples are located in the *examples/* folder in the SDK which can be downloaded at `www.signalhound.com/software-development-kit`.

1.2 Measurements

This section covers the main measurements available through the API.

- Swept Spectrum Analysis
- Real-Time Spectrum Analysis
- I/Q Streaming
- Audio Demodulation
- Scalar Network Analysis

Also see Theory of Operation for more information.

1.3 Build/Version Notes

1.3.1 Builds

Both Windows and Linux builds are available in the SDK.

Windows builds are available for both x86 and x64. The Windows builds are compiled with Visual Studio 2012 and any application using this library will require distributing the `VS2012 redistributable libraries`.

Linux versions are available for 64-bit Ubuntu 18.04 and CentOS7.

1.3.2 Versions

Versions are of the form *major.minor.revision*.

A *major* change signifies a significant change in functionality relating to one or more measurements, or the addition of significant functionality. Function prototypes have likely changed.

A *minor* change signifies additions that may improve existing functionality or fix major bugs but make no changes that might affect existing user's measurements. Function prototypes can change but do not change existing parameters meanings.

A *revision* change signifies minor changes or bug fixes. Function prototypes will not change. Users should be able to update by simply replacing the .DLL/.so.

See the change log in the SDK for detailed API version history.

1.3.2.1 What's New in Version 5.0

- Added support for the BB60D.
- For existing programs only utilizing BB60C/A devices, no software changes are required when using the new DLL.
- Some functions require different parameters when using the BB60D, such as bbConfigureIO.
- A new device type is returned from the bbGetDeviceType function to detect the BB60D when connected.
- New functions have been added specifically for the BB60D. These functions configure and enable the UART output for various types of antenna switching. See UART Antenna Switching.
- Some functions have been deprecated but will continue to be present in the API for existing applications. Alternatives have been provided for these functions.
  - bbConfigureRefLevel replaces bbConfigureLevel
  - bbConfigureGainAtten replaces bbConfigureGain
  - bbQueryIQParameters replaces bbQueryStreamInfo
- Several macros have been deprecated but remain in the API header file for existing applications. The updated and more descriptive macro names should be used.
- Added function bbGetSerialNumberList2, which also returns the device type. Enables the ability to target a specific device type when opening a device.

1.3.2.2 What's New in Version 4.0

- Added support for the BB60C-2, a minor hardware revision to the BB60C. The non-deprecated functions have not changed, so integrating the new BB60C-2 devices should require no software changes other than updating the API DLL in the project.
- Several deprecated functions have been removed. There are alternatives for each deprecated function.
  - bbFetchRaw: use bbGetIQ instead.
  - bbConfigureWindow: use bbConfigureSweepCoupling instead.
  - bbQueryTimestamp: use bbGetIQ instead.

1.3.2.3 What's New in Version 3.0

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

1.3.2.4 What's New in version 2.0

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

1.3.2.5 Updating from Version 1.2 or Later

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

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

1.4 Requirements

1.4.1 Windows Development Requirements

- Windows 7/8/10/11 (64-bit Windows 10 recommended)
- Windows C/C++ development tools and environment.
  - API was compiled using VS2012 and VS2019.
    - VS2012/VS2019 C++ redistributables are required.
- bb_api.h, API header file.
- bb_api.lib/bb_api.dll, These are the main API library files.
- ftd2xx.dll, Provided in SDK. Must be in the same directory as the bb_api.dll. Used for scalar network analysis sweeps with the Signal Hound tracking generator.

1.4.2 Linux Development Requirements

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

1.4.3 PC Requirements

- USB 3.0 connectivity provided through 4th generation or later Intel CPUs.
- (Recommended) Quad core Intel i5/i7 processors, 4th generation or later.
- (Minimum) Dual-core Intel i5/i7 processors, 4th generation or later.

1.5 UART Antenna Switching

The BB60D has a UART output on port 2. The UART port has 3 output modes:

1. Immediate, send a byte on demand.
2. Antenna switching during sweeps. During a sweep a UART byte will be transmitted at specific positions in the sweep, allowing multi-antenna coverage during a single sweep.
3. Pseudo-doppler switching during I/Q streaming. While I/Q streaming the UART can be configured to output bytes at specific intervals for pseudo-doppler direction finding applications.

In all 3 output modes the UART transmits 1 byte at a time, including start and stop bit for a total of 10 bits. The output baud rate can be specified from a list of available rates using the bbSetUARTRate function.

There is no UART input capability.

See bbConfigureIO, the UART functions, and the C++ programming examples for more information.

1.6 Thread Safety

The BB60 API is not thread safe. A multi-threaded application is free to call the API from any number of threads if the function calls are synchronized (i.e. using a mutex). Not synchronizing your function calls will lead to undefined behavior.

1.7 Multiple Devices and Multiple Processes

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

If you wish to use the API in multiple processes it is the user's responsibility to manage a list of devices to prevent the possibility of opening a device twice from two different processes. Two processes communicating to the same device will result in undefined behavior. One possible way to manage inter-process information like this is to use a named mutex on a Windows system.

If you wish to interface multiple devices on Linux, please see Multiple USB Devices.

1.8 Power State

The BB60D can be set into a low power state using the bbSetPowerState function provided by the API. Power will reduce to ∼1.25W from 6W when active. The device should not be actively performing any measurements when setting the device to the standby power state and should not perform any measurements while in the standby state. Before setting the device to standby, the device should be in the idle mode (by calling bbAbort) or configured for sweeping but not actively performing a sweep.

Here are some other power state transitions that occur:

- When the device is first connected to the PC, it starts in the low power power state.
- When the device is opened via the API, it automatically goes to the on state.
- When the device is closed via the API, it is sent to the standby power state.
- If the program crashes and the device is not able to be properly closed via the API, it can be left in the on state.

See the C++ programming examples for an example of configuring the power state.

1.9 Linux Notes

1.9.1 USB Throughput

By default, Linux applications cannot increase the priority of individual threads unless ran with elevated privilege (root). On Windows this issue does not exist, and the API will elevate the USB data acquisition threads to a higher priority to ensure USB data loss does not occur. On Linux, the user will need to run their application as root to ensure USB data acquisition is performed at a higher priority.

If this is not done, there is a higher risk of USB data loss for streaming modes such as I/Q, real-time, and fast sweep measurements on Linux.

In our testing, if little additional processing is occurring outside the API, 1 or 2 devices typically will not experience data loss due to this issue. Once the user application increases the processing load or starts performing I/O such as storing data to disk, the occurrence of USB data loss increases and the need to run the application as root increases.

1.9.2 Multiple USB Devices

There are system limitations when attempting to use multiple Signal Hound USB 3.0 devices∗ simultaneously on Linux operating systems. The default amount of memory allocated for USB transfers on Linux is 16MB. A single Signal Hound USB 3.0 device will stay within this allocation size, but two devices will exceed this limitation and can cause connection issues or will cause the software to crash.

The USB memory allocation size can be changed by writing to the file

```
/sys/module/usbcore/parameters/usbfs_memory_mb
```

A good value would be N ∗ 16 where N is the number of devices you plan on interfacing simultaneously. One way to write to this file is with the command:

```
sudo sh -c 'echo 32 > /sys/module/usbcore/parameters/usbfs_memory_mb'
```

where 32 can be replaced with any value you wish. You may need to restart the system for this change to take effect.

∗Includes both Signal Hound USB 3.0 spectrum analyzers and signal generators.

1.10 Other Programming Languages

The BB60 interface is C compatible which ensures it is possible to interface the API in most languages that can call C functions. These languages include C++, C#, Python, MATLAB, LabVIEW, Java, etc. Some examples of calling the API in these other languages are included in the code examples folder.

1.11 Device Connection Errors

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

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

bbDeviceConnectionErr – Device connection errors are the result of major USB issues, most commonly being the device has lost power (unplugged) or interruptions in the USB connection. These errors should be handled by closing and reopening the device with the bbCloseDevice/bbOpenDevice functions. In some cases, the device may need to be fully power cycled by disconnecting and reconnecting the USB cable.

1.11.1 Firmware Version 7 (BB60C)

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

1.12 Manual Gain/Attenuation

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

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

1.13 I/Q Data Types

I/Q data is returned from the bbGetIQ function either as 32-bit complex floats or 16-bit complex shorts depending on the data type set in bbConfigureIQDataType. 16-bit shorts are twice as memory efficient as floats but require more effort to convert to absolute amplitudes and may be less convenient to work with.

When data is returned as 32-bit complex floats, the data is scaled to mW and the amplitude can be calculated by the following equation:

```
samplePowerDBm = 10.0 * log10(re*re + im*im);
```

where `re` and `im` are the real and imaginary components of a single I/Q sample.

When data is returned as 16-bit complex shorts, the data is full scale and a correction must be applied before you can measure mW or dBm. Values range from [-32768 to +32767]. To measure the power of a sample using the complex short data type, three steps are required:

1. Convert from short to float.
   ```
   float re32f = ((float)re16s / 32768.0);
   float im32f = ((float)im16s / 32768.0); // This converts the short to a float in the range of [-1.0 to
   +1.0]
   ```
2. Scale the floats by the correction value returned from bbGetIQCorrection.
   ```
   re32f *= correction;
   im32f *= correction;
   ```
3. Calculate power
   ```
   samplePowerDBm = 10.0 * log10(re32f*re32f + im32f*im32f);
   ```

1.14 I/Q Filtering and Bandwidth Limitations

Users can control the baseband software filter when I/Q streaming. Software filtering is always enabled. For each decimation rate, a maximum bandwidth can be provided with the bbConfigureIQ function. The table below shows the sample rates and maximum bandwidth for each decimation rate.

Decimation Rate	Sample Rate (I/Q pairs/s)	Maximum Bandwidth
1	40 MS/s	27 MHz
2	20 MS/s	17.8 MHz
4	10 MS/s	8.0 MHz
8	5 MS/s	3.75 MHz
16	2.5 MS/s	2.0 MHz
32	1.25 MS/s	1.0 MHz
64	625 kS/s	0.5 MHz

Decimation Rate	Sample Rate (I/Q pairs/s)	Maximum Bandwidth
128	312.5 kS/s	0.25 MHz
256	156.250 kS/s	140 kHz
512	78.125 kS/s	65 kHz
1024	39062.5 S/s	30 kHz
2048	19531.25 S/s	15 kHz
4096	9765.625 S/s	8 kHz
8192	4882.8125 S/s	4 kHz

1.15 Using a GPS Receiver to Time-Stamp Data

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

What you will need:

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

Order of Operations:

1. Ensure correct operation of your GPS receiver.
2. Connect the 1PPS receiver output to port 2 of the device.
3. Connect the RS232 receiver output to your PC.
4. Determine the COM port number and baud rate of the data transfer over RS232 to your PC.
5. Open the device via bbOpenDevice
6. Ensure the RS232 connection is not open.
7. Use bbSyncCPUtoGPS to synchronize the API timing with the current GPS time. This function will release the connection when finished.
8. Configure the device for I/Q streaming.
9. Before initiating the device, use bbConfigureIO and configure port 2 for an incoming rising edge trigger via BB60C_PORT2_IN_TRIG_RISING_EDGE or BB60D_PORT2_IN_TRIG_RISING_EDGE.
10. Call bbInitiate(id, BB_STREAMING, BB_TIME_STAMP). The BB_TIME_STAMP argument will tell the API to look for the 1PPS input trigger for timing.
11. If initiated successfully you can now fetch I/Q data and timestamps with bbGetIQ. The timestamp returned will be the time of the first sample in the array of data collected.

1.15.1 Code Example

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

1.16 Calculating FFT Size

For a given RBW, the FFT size used for sweeps can be calculated with the following algorithm:

```
double BestFFTSize = (80.0e6 * WindowBW) / RBW;
double ActualFFTSize = RoundUpToNextPowerOf2(BestFFTSize) * C;
double ZeroPadding = ActualFFTSize - BestFFTSize;
```

Where

- `WindowBW` equals 2.02 for Nuttall windows, 3.7702 for flattop windows and 2.65 for CISPR windows.
- `C` equals 2 for CISPR windows and 1 for Nuttall and flattop windows.

1.17 Contact Information

For technical support, email `support@signalhound.com`.

For sales, email `sales@signalhound.com`.

Chapter 2

Theory of Operation

Any application using the BB60 API will follow these steps to perform measurements with a device:

1. Open a device and receive a handle to the device resources.
2. Configure the device.
3. Acquire measurement data.
4. Stop acquisitions and abort the current measurement.
5. Repeat steps 2-5 for further measurements if desired.
6. Close the device.
7. (Recalibration)

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

2.1 Opening a Device

Before opening a BB60, the device must be connected to the PC using the USB 3.0 cable, and the front panel LED should be solid green (or solid red to indicate a BB60D on standby). The bbOpenDevice function will attempt to open the device, and if successful return an integer handle that can be used to interface the device for the remainder of your program.

You can either have the API open the first valid BB60 device found on the system or specify the serial number of the device you wish to open. Get a list of all devices connected to the system using the bbGetSerialNumberList function.

Up to 8 devices may be connected to the API.

2.2 Configuring the Device

Once the device is open, the next step is to configure the device for a measurement. The available measurement modes are swept analysis, real-time analysis, I/Q streaming, and scalar network analysis using a Signal Hound tracking generator. Each mode has specific configurations routines, which set a temporary configuration state. Once all configuration routines have been called, calling the bbInitiate function copies the temporary configuration state into the active measurement state and the device is ready for measurements. The provided code examples showcase how to configure the device for each measurement mode.

2.3 Acquiring Measurement Data

After the device has been successfully configured, the API provides several functions for acquiring measurements. Only certain measurements are available depending on the active measurement mode. For example, I/Q data acquisition is not available when the device is in a sweep measurement mode. See the provided examples in the SDK.

2.4 Stopping the Measurements

Stopping all measurements is achieved through the bbAbort function. This causes the device to cancel or finish any pending operations and return to an idle state. Calling bbAbort is never required, as it is called by default if you attempt to change the measurement mode or close the device, but it can be useful to do this.

- Certain measurement modes can consume large amounts of resources such as memory and CPU usage. Returning to an idle state will free those resources.
- Returning to an idle state will help reduce power consumption.

2.5 Closing the Device

When finished making measurements, you can close the device and free all resources related to the device with the bbCloseDevice function. Once closed, the device will appear in the open device list again. It is possible to open and close a device multiple times during the execution of a program.

2.6 Recalibration

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

Note: The BB60C and BB60D do not require the use of bbSelfCal for device calibration. Instead, if the device deviates in temperature, simply call bbInitiate again which will re-calibrate the device at its current operating temperature.

Chapter 3

Measurement Types

Please also see the C++ programming examples for an example of interfacing the device for each measurement type.

3.1 Swept Spectrum Analysis

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

3.1.1 Example

For a list of all examples, please see the *examples/* folder in the SDK.

```
#include "bb_api.h"
#include <iostream>
#include <vector>
#ifdef _WIN32
#pragma comment(lib, "bb_api.lib")
#endif
/*
This example demonstrates using the API to perform a sweep
*/
void bbExampleSweep()
{
    int handle;
    bbStatus status = bbOpenDevice(&handle);
    if(status != bbNoError) {
        std::cout « "Issue opening device\n";
        std::cout « bbGetErrorString(status) « "\n";
        return;
    }
    // Configure a sweep from 850MHz to 950MHz with a 10kHz
    //  RBW/VBW and an expected input of at most -20dBm.
    bbConfigureRefLevel(handle, -20.0);
    bbConfigureCenterSpan(handle, 900.0e6, 100.0e6);
    bbConfigureSweepCoupling(handle, 10.0e3, 10.0e3, 0.001, BB_RBW_SHAPE_FLATTOP, BB_NO_SPUR_REJECT);
    bbConfigureAcquisition(handle, BB_AVERAGE, BB_LOG_SCALE);
    bbConfigureProcUnits(handle, BB_POWER);
    // Configuration complete, initialize the device
    status = bbInitiate(handle, BB_SWEEPING, 0);
    if(status < bbNoError) {
        std::cout « "Error configuring device\n";
        std::cout « bbGetErrorString(status);
        exit(-1);
    }
    // Get sweep characteristics and allocate memory for sweep
    uint32_t sweepSize;
    double binSize, startFreq;
    bbQueryTraceInfo(handle, &sweepSize, &binSize, &startFreq);
    std::vector<float> sweep(sweepSize);
    // Get the sweep
    // Pass NULL for the min parameter.  For most scenarios, the min sweep can be ignored.
    status = bbFetchTrace_32f(handle, sweepSize, 0, sweep.data());
    if(status != bbNoError) {
        std::cout « "Sweep status:  " « bbGetErrorString(status) « "\n";
    }
    // If the sweep status is not an error, the sweep is now stored in the sweep
    //   vector.  The frequency of the first index in the vector is startFreq.
    // To get the frequency of any bin in the vector use the equation
    // freqHz = startFreq + binSize * index;
    // From this point, you can continue to get sweeps with this configuration
    //   by calling bbFetchTrace again, or reconfigure and initiate the
    //   device for a new sweep configuration.
    // Finished/close device
    bbAbort(handle);
    bbCloseDevice(handle);
}
```

3.1.2 Configuration

The configuration routines which affect the sweep results are:

- bbConfigureCenterSpan
- bbConfigureRefLevel
- bbConfigureSweepCoupling
- bbConfigureAcquisition
- bbConfigureProcUnits

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

3.1.3 Usage

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

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

Now you can call bbFetchTrace. This is a blocking call that does not begin the sweep until the function is called.

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

```
Frequency of nth sample point in returned sweep = startFreq + n * binSize
```

3.2 Real-Time Spectrum Analysis

The API provides the functionality of a real-time spectrum analyzer for the full instantaneous bandwidth of the device (20MHz for the BB60A, 27MHz for the BB60C and BB60D). Using FFTs at an overlapping rate of 50%, the spectrum results have no blind time (100% probability of intercept) for events as short as 4.8us at full amplitude (at 631kHz RBW). The RBW shape is restricted to the Nuttall window, and VBW is not configurable.

3.2.1 Example

For a list of all examples, please see the *examples/* folder in the SDK.

```
#include "bb_api.h"
#include <iostream>
#include <vector>
#ifdef _WIN32
#pragma comment(lib, "bb_api.lib")
#endif
/*
This example illustrates how to perform real-time spectrum analysis
with the API. The sweep and and persistence frames are retrieved.
*/
void bbExampleRealTime()
{
    int handle;
    bbStatus status = bbOpenDevice(&handle);
    if(status != bbNoError) {
        std::cout « "Issue opening device\n";
        std::cout « bbGetErrorString(status) « "\n";
        return;
    }
    // Configure a 27MHz real-time stream at a 2.44GHz center
    bbConfigureRefLevel(handle, -20.0);
    bbConfigureCenterSpan(handle, 2.44e9, 20.0e6);
    bbConfigureSweepCoupling(handle, 10.e3, 10.0e3, 0.001,
        BB_RBW_SHAPE_NUTTALL, BB_NO_SPUR_REJECT);
    bbConfigureAcquisition(handle, BB_MIN_AND_MAX, BB_LOG_SCALE);
    // Configure a frame rate of 30fps and 100dB scale for the frame
    bbConfigureRealTime(handle, 100.0, 30);
    // Configuration complete, initialize the device
    status = bbInitiate(handle, BB_REAL_TIME, 0);
    if(status < bbNoError) {
        std::cout « "Error configuring device\n";
        std::cout « bbGetErrorString(status);
        exit(-1);
    }
    // Get sweep characteristics and allocate memory for sweep and
    // real-time frames.
    uint32_t sweepSize;
    double binSize, startFreq;
    bbQueryTraceInfo(handle, &sweepSize, &binSize, &startFreq);
    int frameWidth, frameHeight;
    bbQueryRealTimeInfo(handle, &frameWidth, &frameHeight);
    std::vector<float> sweep(sweepSize);
    std::vector<float> frame(frameWidth * frameHeight);
    std::vector<float> alphaFrame(frameWidth * frameHeight);
    // Retrieve roughly 1 second worth of real-time persistence frames and sweeps.
    // Ignore min sweep, pass NULL as parameter.
    int frameCount = 0;
    while(frameCount++ < 30) {
        bbFetchRealTimeFrame(handle, nullptr, sweep.data(), frame.data(), alphaFrame.data());
    }
    // Finished/close device
    bbAbort(handle);
    bbCloseDevice(handle);
}
```

3.2.2 Configuration

The configuration routines which affect the spectrum results are:

- bbConfigureRealTime
- bbConfigureRealTimeOverlap
- bbConfigureCenterSpan
- bbConfigureRefLevel
- bbConfigureAcquisition
- bbConfigureSweepCoupling

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

3.2.3 Usage

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

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

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

3.3 I/Q Streaming

The API can be used to stream I/Q data up to 40 MS/s. I/Q data can be retrieved as 32-bit complex floats or 16-bit complex shorts. I/Q data provided as 32-bit floats are corrected for IF flatness and RF leveling. I/Q data returned as 16-bit shorts is provided as full scale, only correcting for IF flatness and the user must apply a correction value to recover the fully amplitude corrected I/Q data.

3.3.1 Example

For a list of all examples, please see the *examples/* folder in the SDK.

```
#include "bb_api.h"
#include <iostream>
#include <vector>
#ifdef _WIN32
#pragma comment(lib, "bb_api.lib")
#endif
/*
This example demonstrates how to configure, initialize, and retrieve data
in I/Q streaming mode.  I/Q streaming mode provides continuous I/Q data
at a fixed center frequency with a selectable sample rate.
*/
void bbExampleIQStreaming()
{
    // Connect device
    int handle;
    bbStatus status = bbOpenDevice(&handle);
    if(status != bbNoError) {
        std::cout « "Issue opening device\n";
        std::cout « bbGetErrorString(status) « "\n";
        return;
    }
    // Configure the measurement parameters
    // Specify 32-bit floating point complex values
    bbConfigureIQDataType(handle, bbDataType32fc);
    // Set center frequency
    bbConfigureIQCenter(handle, 1.0e9);
    // Set reference level, maximum expected input amplitude
    bbConfigureRefLevel(handle, -20.0);
    // Set a sample rate of 40MS/s and a bandwidth of 27MHz
    bbConfigureIQ(handle, 1, 27.0e6);
    // Initiate the device, once this function returns the device
    // will be streaming I/Q.
    status = bbInitiate(handle, BB_STREAMING, BB_STREAM_IQ);
    if(status != bbNoError) {
        std::cout « "Initiate error\n";
        std::cout « bbGetErrorString(status) « "\n";
        bbCloseDevice(handle);
        return;
    }
    // Get I/Q stream characteristics
    double sampleRate, bandwidth;
    bbQueryIQParameters(handle, &sampleRate, &bandwidth);
    // Allocate memory for BLOCK_SIZE number of complex values
    // This is the number of I/Q samples we will request each function call.
    const int BLOCK_SIZE = 262144;
    std::vector<float> buffer(BLOCK_SIZE * 2);
    // Perform the capture, pass NULL for any parameters we don't care about
    status = bbGetIQUnpacked(handle, buffer.data(), BLOCK_SIZE, 0, 0,
        BB_FALSE, 0, 0, 0, 0);
    // Check status here
    // At this point, BLOCK_SIZE IQ data samples have been retrieved and
    //   stored in the buffer array.  Any processing on the data should happen here.
    // Continue to call bbGetIQ with the purge flag set to false, which ensures
    //   I/Q data is continuous from the last call.
    for(int i = 0; i < 10; i++) {
        status = bbGetIQUnpacked(handle, buffer.data(), BLOCK_SIZE, 0, 0,
            BB_FALSE, 0, 0, 0, 0);
        // Check status here
        // Do processing
    }
    // When done, stop streaming and close device.
    bbAbort(handle);
    bbCloseDevice(handle);
}
```

3.3.2 Configuration

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

- bbConfigureIQCenter
- bbConfigureRefLevel
- bbConfigureIO – Used for external triggering.
- bbConfigureIQ
- bbConfigureIQDataType
- bbConfigureIQTriggerSentinel
- bbQueryIQParameters
- bbGetIQCorrection
- bbGetIQ/bbGetIQUnpacked

Once configured, initialize the device with the BB_STREAMING flag.

3.3.3 Usage

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

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

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

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

3.3.4 External Triggering

External trigger information can be retrieved when I/Q streaming. Trigger information is provided through the triggers buffer in the bbGetIQ function.

3.3.4.1 Example

For a list of all examples, please see the *examples/* folder in the SDK.

```
#include "bb_api.h"
#include <complex>
#include <iostream>
#include <vector>
#ifdef _WIN32
#pragma comment(lib, "bb_api.lib")
#endif
/*
This example demonstrates using the BB60 to look for an external trigger.
This example waits for an external trigger, and captures N I/Q samples after
that trigger.  If no trigger arrives this example will loop indefinitely.
*/
const int TRIGGER_SENTINEL = -1;
void bbExampleIQExtTrigger()
{
    int handle;
    bbStatus status = bbOpenDevice(&handle);
    if(status != bbNoError) {
        std::cout « "Issue opening device\n";
        std::cout « bbGetErrorString(status) « "\n";
        return;
    }
    int deviceType;
    bbGetDeviceType(handle, &deviceType);
    // Configure BNC port 2 for rising edge trigger detection
    if(deviceType == BB_DEVICE_BB60D) {
        // BB60D
        bbConfigureIO(handle, BB60D_PORT1_DISABLED, BB60D_PORT2_IN_TRIG_RISING_EDGE);
    } else {
        // BB60C/A devices
        bbConfigureIO(handle, 0, BB60C_PORT2_IN_TRIG_RISING_EDGE);
    }
    // Now configure the measurement parameters
    // We want 32-bit floating point complex values
    bbConfigureIQDataType(handle, bbDataType32fc);
    // Set center frequency to 1GHz
    bbConfigureIQCenter(handle, 1.0e9);
    // Set reference level
    bbConfigureRefLevel(handle, -20.0);
    // Set a sample rate of 40.0e6 / 2 = 20.0e6 MS/s and bandwidth of 15 MHz
    bbConfigureIQ(handle, 2, 15.0e6);
    // By default the sentinel is zero, set to -1
    bbConfigureIQTriggerSentinel(TRIGGER_SENTINEL);
    // Initiate the device, once this function returns the device
    // will be streaming I/Q.
    status = bbInitiate(handle, BB_STREAMING, BB_STREAM_IQ);
    if(status != bbNoError) {
        std::cout « "Initiate error\n";
        std::cout « bbGetErrorString(status) « "\n";
        bbCloseDevice(handle);
        return;
    }
    // Get I/Q stream characteristics
    double sampleRate, bandwidth;
    bbQueryIQParameters(handle, &sampleRate, &bandwidth);
    // This is how much data we want after the external trigger
    const int N = 1e6;
    // I/Q capture buffer
    std::vector<std::complex<float» buffer(N);
    // We only care about the first trigger we see.  If you need more triggers,
    //   this can be an array.
    int triggerPos = 0;
    while(true) {
        // Perform the capture, pass NULL for any parameters we don't care about
        status = bbGetIQUnpacked(handle, &buffer[0], N, &triggerPos, 1,
            BB_FALSE, 0, 0, 0, 0);
        // At this point, N I/Q samples have been retrieved and
        //  stored in the buffer array.  If any triggers were seen during the capture
        //  of the returned samples, the trigger parameter will contain an index into
        //  the buffer array at which the trigger was seen.
        // A trigger value not equal to the sentinel means a trigger is present
        if(triggerPos != TRIGGER_SENTINEL) {
            // We have a trigger, now finish capture
            break;
        }
    }
    // Unless trigger was at beginning, we still need to capture some data
    //  to retrive N samples.
    int samplesAfterTrigger = N - triggerPos;
    int samplesLeft = N - samplesAfterTrigger;

    // Move the samples after the trigger to the beginning of the buffer.
    for(int i = 0; i < samplesAfterTrigger; i++) {
        buffer[i] = buffer[i + triggerPos];
    }
    // Get the rest of the contiguous samples
    bbGetIQUnpacked(handle, &buffer[samplesAfterTrigger], samplesLeft,
        0, 0, BB_FALSE, 0, 0, 0, 0);
    // When done, stop streaming and close device.
    bbAbort(handle);
    bbCloseDevice(handle);
}
```

3.3.4.2 Usage

If a trigger buffer is provided to bbGetIQ, any external trigger events seen during the acquisition of the returned I/Q data will be placed in the trigger buffer. External trigger events are returned as indices into the I/Q data at which the trigger event occurred. For example, if 1000 I/Q samples are requested and a trigger buffer of size 3 is provided, and the function returns with the trigger buffer set to [12,300,876], this indicates that an external trigger event occurred at I/Q sample index 12, 300, and 876 in the I/Q data returned from this function call.

If fewer external triggers were seen during the I/Q acquisition than the size of the trigger buffer provided, the remainder of the trigger buffer is set to the sentinel value. The default sentinel value is 0, so for example, if a trigger buffer of size 3 is provided, and only a single trigger event was seen, the trigger buffer will return [N, 0, 0] where N is the single trigger index returned.

If more trigger events were seen during the I/Q acquisition than the size of the trigger buffer, those trigger events that cannot fit in the buffer are discarded.

A note on trigger sentinel values: the default sentinel value of 0 does not allow the detection of triggers occurring at the first sample point. If this is an issue, set the sentinel value to -1 or some other negative value which cannot be normally returned. The default value of 0 is the result of historical choices and will remain the default value.

3.4 Audio Demodulation

3.4.1 Example

For a list of all examples, please see the *examples/* folder in the SDK.

```
#include "bb_api.h"
#include <iostream>
#include <vector>
#ifdef _WIN32
#pragma comment(lib, "bb_api.lib")
#endif
/*
This example demonstrates how to configure, initialize, and retrieve data
in audio demodulation mode.
*/
void bbExampleAudioDemod()
{
    // Connect device
    int handle;
    bbStatus status = bbOpenDevice(&handle);
    if(status != bbNoError) {
        std::cout « "Issue opening device\n";
        std::cout « bbGetErrorString(status) « "\n";
        return;
    }
    // Configure the measurement parameters
    // Set the demodulation scheme, center frequency, IF bandwidth,
    //   post demodulation low & hi pass filter frequencies, and FM deemphasis in microseconds
    status = bbConfigureDemod(handle, BB_DEMOD_FM, 97.1e6, 120.0e3, 8.0e3, 20.0, 75.0);
    // Initiate the device, once this function returns the device
    //   will be streaming demodulated audio data.
    status = bbInitiate(handle, BB_AUDIO_DEMOD, 0);
    if(status != bbNoError) {
        std::cout « "Initiate error\n";
        std::cout « bbGetErrorString(status) « "\n";
        bbCloseDevice(handle);
        return;
    }
    // Allocate memory for 4096 audio samples for an audio sample rate of 32k.
    // This is the number of audio samples we will acquire each function call.
    std::vector<float> buffer(4096);
    // Perform the capture
    status = bbFetchAudio(handle, buffer.data());
    // Check status here
    // At this point, 4096 audio samples have been retrieved and
    //   stored in the buffer array.  Any processing on the data should happen here.
    // Continue to call bbFetchAudio.
    // While 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.
    for(int i = 0; i < 10; i++) {
        status = bbFetchAudio(handle, buffer.data());
        // Check status here
        // Do processing
    }
    // When done, stop streaming and close device.
    bbAbort(handle);
    bbCloseDevice(handle);
}
```

3.4.2 Configuration

Initialize the device with the BB_AUDIO_DEMOD flag.

3.4.3 Usage

When audio is being performed, no other measurements can take place. If you need the I/Q data and the ability to demodulate audio, consider using the I/Q streaming functionality and performing the audio demodulation on the I/Q data from there.

Once the device is streaming audio it is possible to continue to change the audio settings via bbConfigureDemod if the updated center frequency does not exceed +/- 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.

3.5 Scalar Network Analysis

When a Signal Hound tracking generator is paired together with a BB60D, BB60C, or BB60A spectrum analyzer, the products can function as a scalar network analyzer to perform insertion loss measurements or return loss measurements by adding a directional coupler. Throughout this document, this functionality will be referred to as tracking generator (or TG) sweeps.

3.5.1 Example

For a list of all examples, please see the *examples/* folder in the SDK.

```
#include "bb_api.h"
#include <iostream>
#include <vector>
#ifdef _WIN32
#pragma comment(lib, "bb_api.lib")
#endif
// This example demonstrates how to use the API to perform a single tracking generator sweep.
// See the manual for a full description of each step of the process in the
// Scalar Network Analysis section.
void bbExampleScalarNetworkAnalysis()
{
    // Connect device
    int handle;
    bbStatus status = bbOpenDevice(&handle);
    if(status != bbNoError) {
        std::cout « "Issue opening device\n";
        std::cout « bbGetErrorString(status) « "\n";
        return;
    }
    if(bbAttachTg(handle) != bbNoError) {
        std::cout « "Unable to find tracking generator\n";
        return;
    }
    // Sweep some device at 900 MHz center with 1 MHz span
    bbConfigureCenterSpan(handle, 900.0e6, 1.0e6);
    bbConfigureAcquisition(handle, BB_MIN_AND_MAX, BB_LOG_SCALE);
    bbConfigureRefLevel(handle, -10.0);
    bbConfigureSweepCoupling(handle, 1.0e3, 1.0e3, 0.001, BB_RBW_SHAPE_FLATTOP, BB_SPUR_REJECT);
    bbConfigureProcUnits(handle, BB_POWER);
    // Additional configuration routine
    // Configure a 100 point sweep
    // The size of the sweep is a suggestion to the API, it will attempt to get near the requested size
    // Optimized for high dynamic range and passive devices
    bbConfigTgSweep(handle, 100, true, true);
    // Configuration complete, initialize the device
    status = bbInitiate(handle, BB_TG_SWEEPING, 0);
    if(status < bbNoError) {
        std::cout « "Error configuring device\n";
        std::cout « bbGetErrorString(status);
        exit(-1);
    }
    // Get sweep characteristics and allocate memory for sweep
    uint32_t sweepSize;
    double binSize, startFreq;
    bbQueryTraceInfo(handle, &sweepSize, &binSize, &startFreq);
    std::vector<float> sweep(sweepSize);
    // Create test set-up without DUT present
    // Get one sweep
    // Pass NULL for the min parameter.  For most scenarios, the min sweep can be ignored.
    status = bbFetchTrace_32f(handle, sweepSize, 0, sweep.data());
    if(status != bbNoError) {
        std::cout « "Sweep status:  " « bbGetErrorString(status) « "\n";
        return;
    }
    // Store baseline
    bbStoreTgThru(handle, TG_THRU_0DB);
    // Should pause here, and insert DUT into test set-up
    bbFetchTrace_32f(handle, sweepSize, 0, sweep.data());
    // From here, you can sweep several times without needing to restore the thru.
    // Once you change your setup, you should reconfigure the device and
    //   store the thru again without the DUT inline.
    // Finished/close device
    bbAbort(handle);
    bbCloseDevice(handle);
}
```

3.5.2 Configuration and Usage

Initialize the device with the BB_TG_SWEEPING flag.

Scalar Network Analysis can be realized by following these steps:

1. Ensure the Signal Hound BB60 spectrum analyzer and tracking generator are connected to your PC.

2. Open the spectrum analyzer through the bbOpenDevice function.
3. Associate a tracking generator to a spectrum analyzer by calling bbAttachTg. At this point, if a TG is present, it is claimed by the API and cannot be discovered again until bbCloseDevice is called.
4. Configure the device as normal, setting sweep frequencies and reference level (or manually setting gain and attenuation).
5. Configure the TG sweep with the bbConfigTgSweep function. This function configures TG sweep specific parameters.
6. Call bbInitiate with the BB_TG_SWEEPING mode flag.
7. Get the sweep characteristics with bbQueryTraceInfo.
8. Connect the BB and TG device into the final test state without the DUT and perform one sweep with bbFetchTrace. After one full sweep has returned, call bbStoreTgThru with the TG_THRU_0DB flag.
9. (Optional) Configure the setup again still without the DUT but with a 20dB pad inserted into the system. Perform an additional full sweep and call bbStoreTgThru with the TG_THRU_20DB.
10. Once store through has been called, insert your DUT into the system and then you can freely call the get sweep function until you modify the configuration or settings.

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

Chapter 4

Data Structure Index

4.1 Data Structures

Here are the data structures with brief descriptions:

Chapter 5

File Index

5.1 File List

Here is a list of all documented files with brief descriptions:

Chapter 6

Data Structure Documentation

6.1 bbIQPacket Struct Reference

```
#include <bb_api.h>
```

Data Fields

- void ∗ iqData
- int iqCount
- int ∗ triggers
- int triggerCount
- int purge
- int dataRemaining
- int sampleLoss
- int sec
- int nano

6.1.1 Detailed Description

Used to encapsulate I/Q data and metadata. See bbGetIQ.

6.1.2 Field Documentation

6.1.2.1 iqData

```
void* iqData
```

Pointer to an array of 32-bit complex floating-point values. Complex values are interleaved real-imaginary pairs. This must point to a contiguous block of iqCount complex pairs.

6.1.2.2 iqCount

```
int iqCount
```

Number of I/Q data pairs to return.

6.1.2.3 triggers

```
int* triggers
```

pointer to an array of integers. If the external trigger input is active, and a trigger occurs during the acquisition time, triggers will be populated with values which are relative indices into the iqData array where external triggers occurred. Any unused trigger array values will be set to zero.

6.1.2.4 triggerCount

```
int triggerCount
```

Size of the triggers array.

6.1.2.5 purge

```
int purge
```

Specifies whether to discard any samples acquired by the API since the last time and bbGetIQ function was called. Set to BB_TRUE if you wish to discard all previously acquired data, and BB_FALSE if you wish to retrieve the contiguous I/Q values from a previous call to this function.

6.1.2.6 dataRemaining

```
int dataRemaining
```

How many I/Q samples are still left buffered in the API. Set by API.

6.1.2.7 sampleLoss

```
int sampleLoss
```

Returns BB_TRUE or BB_FALSE. Will return BB_TRUE when the API is required to drop data due to internal buffers wrapping. This can be caused by I/Q samples not being polled fast enough, or in instances where the processing is not able to keep up (underpowered systems, or other programs utilizing the CPU) Will return BB_TRUE on the capture in which the sample break occurs. Does not indicate which sample the break occurs on. Will always return false if purge is true. Set by API.

6.1.2.8 sec

```
int sec
```

Seconds since epoch representing the timestamp of the first sample in the returned array. Set by API.

6.1.2.9 nano

```
int nano
```

Nanoseconds representing the timestamp of the first sample in the returned array. Set by API.

The documentation for this struct was generated from the following file:

- bb_api.h

Chapter 7

File Documentation

7.1 bb_api.h File Reference

API functions for the BB60 spectrum analyzers.

Data Structures

- struct bbIQPacket

Macros

- #define BB_TRUE (1)
- #define BB_FALSE (0)
- #define BB_DEVICE_NONE (0)
- #define BB_DEVICE_BB60A (1)
- #define BB_DEVICE_BB60C (2)
- #define BB_DEVICE_BB60D (3)
- #define BB_MAX_DEVICES (8)
- #define BB_MIN_FREQ (9.0e3)
- #define BB_MAX_FREQ (6.4e9)
- #define BB_MIN_SPAN (20.0)
- #define BB_MAX_SPAN (BB_MAX_FREQ - BB_MIN_FREQ)
- #define BB_MIN_RBW (0.602006912)
- #define BB_MAX_RBW (10100000.0)
- #define BB_MIN_SWEEP_TIME (0.00001)
- #define BB_MAX_SWEEP_TIME (1.0)
- #define BB_MIN_RT_RBW (2465.820313)
- #define BB_MAX_RT_RBW (631250.0)
- #define BB_MIN_RT_SPAN (200.0e3)
- #define BB60A_MAX_RT_SPAN (20.0e6)
- #define BB60C_MAX_RT_SPAN (27.0e6)
- #define BB_MIN_USB_VOLTAGE (4.4)
- #define BB_MAX_REFERENCE (20.0)
- #define BB_AUTO_ATTEN (-1)
- #define BB_MAX_ATTEN (3)
- #define BB_AUTO_GAIN (-1)

- #define BB_MAX_GAIN (3)
- #define BB_MIN_DECIMATION (1)
- #define BB_MAX_DECIMATION (8192)
- #define BB_IDLE (-1)
- #define BB_SWEEPING (0)
- #define BB_REAL_TIME (1)
- #define BB_STREAMING (4)
- #define BB_AUDIO_DEMOD (7)
- #define BB_TG_SWEEPING (8)
- #define BB_NO_SPUR_REJECT (0)
- #define BB_SPUR_REJECT (1)
- #define BB_LOG_SCALE (0)
- #define BB_LIN_SCALE (1)
- #define BB_LOG_FULL_SCALE (2)
- #define BB_LIN_FULL_SCALE (3)
- #define BB_RBW_SHAPE_NUTTALL (0)
- #define BB_RBW_SHAPE_FLATTOP (1)
- #define BB_RBW_SHAPE_CISPR (2)
- #define BB_MIN_AND_MAX (0)
- #define BB_AVERAGE (1)
- #define BB_LOG (0)
- #define BB_VOLTAGE (1)
- #define BB_POWER (2)
- #define BB_SAMPLE (3)
- #define BB_DEMOD_AM (0)
- #define BB_DEMOD_FM (1)
- #define BB_DEMOD_USB (2)
- #define BB_DEMOD_LSB (3)
- #define BB_DEMOD_CW (4)
- #define BB_STREAM_IQ (0x0)
- #define BB_DIRECT_RF (0x2)
- #define BB_TIME_STAMP (0x10)
- #define BB60C_PORT1_AC_COUPLED (0x00)
- #define BB60C_PORT1_DC_COUPLED (0x04)
- #define BB60C_PORT1_10MHZ_USE_INT (0x00)
- #define BB60C_PORT1_10MHZ_REF_OUT (0x100)
- #define BB60C_PORT1_10MHZ_REF_IN (0x8)
- #define BB60C_PORT1_OUT_LOGIC_LOW (0x14)
- #define BB60C_PORT1_OUT_LOGIC_HIGH (0x1C)
- #define BB60C_PORT2_OUT_LOGIC_LOW (0x00)
- #define BB60C_PORT2_OUT_LOGIC_HIGH (0x20)
- #define BB60C_PORT2_IN_TRIG_RISING_EDGE (0x40)
- #define BB60C_PORT2_IN_TRIG_FALLING_EDGE (0x60)
- #define BB60D_PORT1_DISABLED (0)
- #define BB60D_PORT1_10MHZ_REF_IN (1)
- #define BB60D_PORT2_DISABLED (0)
- #define BB60D_PORT2_10MHZ_REF_OUT (1)
- #define BB60D_PORT2_IN_TRIG_RISING_EDGE (2)
- #define BB60D_PORT2_IN_TRIG_FALLING_EDGE (3)
- #define BB60D_PORT2_OUT_LOGIC_LOW (4)
- #define BB60D_PORT2_OUT_LOGIC_HIGH (5)
- #define BB60D_PORT2_OUT_UART (6)
- #define BB60D_UART_BAUD_4_8K (0)
- #define BB60D_UART_BAUD_9_6K (1)
- #define BB60D_UART_BAUD_19_2K (2)

- #define BB60D_UART_BAUD_38_4K (3)
- #define BB60D_UART_BAUD_14_4K (4)
- #define BB60D_UART_BAUD_28_8K (5)
- #define BB60D_UART_BAUD_57_6K (6)
- #define BB60D_UART_BAUD_115_2K (7)
- #define BB60D_UART_BAUD_125K (8)
- #define BB60D_UART_BAUD_250K (9)
- #define BB60D_UART_BAUD_500K (10)
- #define BB60D_UART_BAUD_1000K (11)
- #define BB60D_MIN_UART_STATES (2)
- #define BB60D_MAX_UART_STATES (8)
- #define TG_THRU_0DB (0x1)
- #define TG_THRU_20DB (0x2)
- #define TG_REF_UNUSED (0)
- #define TG_REF_INTERNAL_OUT (1)
- #define TG_REF_EXTERNAL_IN (2)

Enumerations

- enum bbStatus {
  bbInvalidModeErr = -112 , bbReferenceLevelErr = -111 , bbInvalidVideoUnitsErr = -110 , bbInvalidWindowErr = -109 ,
  bbInvalidBandwidthTypeErr = -108 , bbInvalidSweepTimeErr = -107 , bbBandwidthErr = -106 , bbInvalidGainErr = -105 ,
  bbAttenuationErr = -104 , bbFrequencyRangeErr = -103 , bbInvalidSpanErr = -102 , bbInvalidScaleErr = -101 ,
  bbInvalidDetectorErr = -100 , bbInvalidFileSizeErr = -19 , bbLibusbError = -18 , bbNotSupportedErr = -17 ,
  bbTrackingGeneratorNotFound = -16 , bbUSBTimeoutErr = -15 , bbDeviceConnectionErr = -14 , bbPacketFramingErr = -13 ,
  bbGPSErr = -12 , bbGainNotSetErr = -11 , bbDeviceNotIdleErr = -10 , bbDeviceInvalidErr = -9 ,
  bbBufferTooSmallErr = -8 , bbNullPtrErr = -7 , bbAllocationLimitErr = -6 , bbDeviceAlreadyStreamingErr = -5 ,
  bbInvalidParameterErr = -4 , bbDeviceNotConfiguredErr = -3 , bbDeviceNotStreamingErr = -2 , bbDeviceNotOpenErr = -1 ,
  bbNoError = 0 , bbAdjustedParameter = 1 , bbADCOverflow = 2 , bbNoTriggerFound = 3 ,
  bbClampedToUpperLimit = 4 , bbClampedToLowerLimit = 5 , bbUncalibratedDevice = 6 , bbDataBreak = 7 ,
  bbUncalSweep = 8 , bbInvalidCalData = 9 }
- enum bbDataType { bbDataType32fc = 0 , bbDataType16sc = 1 }
- enum bbPowerState { bbPowerStateOn = 0 , bbPowerStateStandby = 1 }

Functions

- BB_API bbStatus bbGetSerialNumberList (int serialNumbers[BB_MAX_DEVICES], int ∗deviceCount)
- BB_API bbStatus bbGetSerialNumberList2 (int serialNumbers[BB_MAX_DEVICES], int device↩ Types[BB_MAX_DEVICES], int ∗deviceCount)
- BB_API bbStatus bbOpenDevice (int ∗device)
- BB_API bbStatus bbOpenDeviceBySerialNumber (int ∗device, int serialNumber)
- BB_API bbStatus bbCloseDevice (int device)
- BB_API bbStatus bbSetPowerState (int device, bbPowerState powerState)
- BB_API bbStatus bbGetPowerState (int device, bbPowerState ∗powerState)
- BB_API bbStatus bbPreset (int device)
- BB_API bbStatus bbPresetFull (int ∗device)
- BB_API bbStatus bbSelfCal (int device)
- BB_API bbStatus bbGetSerialNumber (int device, uint32_t ∗serialNumber)
- BB_API bbStatus bbGetDeviceType (int device, int ∗deviceType)

- BB_API bbStatus bbGetFirmwareVersion (int device, int ∗version)
- BB_API bbStatus bbGetDeviceDiagnostics (int device, float ∗temperature, float ∗usbVoltage, float ∗usb↩ Current)
- BB_API bbStatus bbConfigureIO (int device, uint32_t port1, uint32_t port2)
- BB_API bbStatus bbSyncCPUtoGPS (int comPort, int baudRate)
- BB_API bbStatus bbSetUARTRate (int device, int rate)
- BB_API bbStatus bbEnableUARTSweeping (int device, const double ∗freqs, const uint8_t ∗data, int states)
- BB_API bbStatus bbDisableUARTSweeping (int device)
- BB_API bbStatus bbEnableUARTStreaming (int device, const uint8_t ∗data, const uint32_t ∗counts, int states)
- BB_API bbStatus bbDisableUARTStreaming (int device)
- BB_API bbStatus bbWriteUARTImm (int device, uint8_t data)
- BB_API bbStatus bbConfigureRefLevel (int device, double refLevel)
- BB_API bbStatus bbConfigureGainAtten (int device, int gain, int atten)
- BB_API bbStatus bbConfigureCenterSpan (int device, double center, double span)
- BB_API bbStatus bbConfigureSweepCoupling (int device, double rbw, double vbw, double sweepTime, uint32_t rbwShape, uint32_t rejection)
- BB_API bbStatus bbConfigureAcquisition (int device, uint32_t detector, uint32_t scale)
- BB_API bbStatus bbConfigureProcUnits (int device, uint32_t units)
- BB_API bbStatus bbConfigureRealTime (int device, double frameScale, int frameRate)
- BB_API bbStatus bbConfigureRealTimeOverlap (int device, double advanceRate)
- BB_API bbStatus bbConfigureIQCenter (int device, double centerFreq)
- BB_API bbStatus bbConfigureIQ (int device, int downsampleFactor, double bandwidth)
- BB_API bbStatus bbConfigureIQDataType (int device, bbDataType dataType)
- BB_API bbStatus bbConfigureIQTriggerSentinel (int sentinel)
- BB_API bbStatus bbConfigureDemod (int device, int modulationType, double freq, float IFBW, float audio↩ LowPassFreq, float audioHighPassFreq, float FMDeemphasis)
- BB_API bbStatus bbInitiate (int device, uint32_t mode, uint32_t flag)
- BB_API bbStatus bbAbort (int device)
- BB_API bbStatus bbQueryTraceInfo (int device, uint32_t ∗traceLen, double ∗binSize, double ∗start)
- BB_API bbStatus bbQueryRealTimeInfo (int device, int ∗frameWidth, int ∗frameHeight)
- BB_API bbStatus bbQueryRealTimePoi (int device, double ∗poi)
- BB_API bbStatus bbQueryIQParameters (int device, double ∗sampleRate, double ∗bandwidth)
- BB_API bbStatus bbGetIQCorrection (int device, float ∗correction)
- BB_API bbStatus bbFetchTrace_32f (int device, int arraySize, float ∗traceMin, float ∗traceMax)
- BB_API bbStatus bbFetchTrace (int device, int arraySize, double ∗traceMin, double ∗traceMax)
- BB_API bbStatus bbFetchRealTimeFrame (int device, float ∗traceMin, float ∗traceMax, float ∗frame, float ∗alphaFrame)
- BB_API bbStatus bbGetIQ (int device, bbIQPacket ∗pkt)
- BB_API bbStatus bbGetIQUnpacked (int device, void ∗iqData, int iqCount, int ∗triggers, int triggerCount, int purge, int ∗dataRemaining, int ∗sampleLoss, int ∗sec, int ∗nano)
- BB_API bbStatus bbFetchAudio (int device, float ∗audio)
- BB_API bbStatus bbAttachTg (int device)
- BB_API bbStatus bbIsTgAttached (int device, bool ∗attached)
- BB_API bbStatus bbConfigTgSweep (int device, int sweepSize, bool highDynamicRange, bool passive↩ Device)
- BB_API bbStatus bbStoreTgThru (int device, int flag)
- BB_API bbStatus bbSetTg (int device, double frequency, double amplitude)
- BB_API bbStatus bbGetTgFreqAmpl (int device, double ∗frequency, double ∗amplitude)
- BB_API bbStatus bbSetTgReference (int device, int reference)
- BB_API const char ∗ bbGetAPIVersion ()
- BB_API const char ∗ bbGetProductID ()
- BB_API const char ∗ bbGetErrorString (bbStatus status)

7.1.1 Detailed Description

API functions for the BB60 spectrum analyzers.

This is the main file for user accessible functions for controlling the BB60 spectrum analyzers.

7.1.2 Macro Definition Documentation

7.1.2.1 BB_TRUE

```
#define BB_TRUE (1)
```

Used for boolean true when integer parameters are being used.

7.1.2.2 BB_FALSE

```
#define BB_FALSE (0)
```

Used for boolean false when integer parameters are being used.

7.1.2.3 BB_DEVICE_NONE

```
#define BB_DEVICE_NONE (0)
```

Device type: No Device. See bbGetDeviceType.

7.1.2.4 BB_DEVICE_BB60A

```
#define BB_DEVICE_BB60A (1)
```

Device type: BB60A. See bbGetDeviceType.

7.1.2.5 BB_DEVICE_BB60C

```
#define BB_DEVICE_BB60C (2)
```

Device type: BB60C. See bbGetDeviceType.

7.1.2.6 BB_DEVICE_BB60D

```
#define BB_DEVICE_BB60D (3)
```

Device type: BB60D. See bbGetDeviceType.

7.1.2.7 BB_MAX_DEVICES

```
#define BB_MAX_DEVICES (8)
```

Maximum number of devices that can be interfaced in the API. See bbGetSerialNumberList, bbGetSerialNumberList2.

7.1.2.8 BB_MIN_FREQ

```
#define BB_MIN_FREQ (9.0e3)
```

Minimum frequency (Hz) for sweeps, and minimum center frequency for I/Q measurements. See bbConfigureCenterSpan, bbConfigureIQCenter.

7.1.2.9 BB_MAX_FREQ

```
#define BB_MAX_FREQ (6.4e9)
```

Maximum frequency (Hz) for sweeps, and maximum center frequency for I/Q measurements. See bbConfigureCenterSpan, bbConfigureIQCenter.

7.1.2.10 BB_MIN_SPAN

```
#define BB_MIN_SPAN (20.0)
```

Minimum span (Hz) for sweeps. See bbConfigureCenterSpan.

7.1.2.11 BB_MAX_SPAN

```
#define BB_MAX_SPAN (BB_MAX_FREQ - BB_MIN_FREQ)
```

Maximum span (Hz) for sweeps. See bbConfigureCenterSpan.

7.1.2.12 BB_MIN_RBW

```
#define BB_MIN_RBW (0.602006912)
```

Minimum RBW (Hz) for sweeps. See bbConfigureSweepCoupling.

7.1.2.13 BB_MAX_RBW

```
#define BB_MAX_RBW (10100000.0)
```

Maximum RBW (Hz) for sweeps. See bbConfigureSweepCoupling.

7.1.2.14 BB_MIN_SWEEP_TIME

```
#define BB_MIN_SWEEP_TIME (0.00001)
```

Minimum sweep time in seconds. See bbConfigureSweepCoupling.

7.1.2.15 BB_MAX_SWEEP_TIME

```
#define BB_MAX_SWEEP_TIME (1.0)
```

Maximum sweep time in seconds. See bbConfigureSweepCoupling.

7.1.2.16 BB_MIN_RT_RBW

```
#define BB_MIN_RT_RBW (2465.820313)
```

Minimum RBW (Hz) for device configured in real-time measurement mode. See bbConfigureSweepCoupling.

7.1.2.17 BB_MAX_RT_RBW

```
#define BB_MAX_RT_RBW (631250.0)
```

Maximum RBW (Hz) for device configured in real-time measurement mode. See bbConfigureSweepCoupling.

7.1.2.18 BB_MIN_RT_SPAN

```
#define BB_MIN_RT_SPAN (200.0e3)
```

Minimum span (Hz) for device configured in real-time measurement mode. See bbConfigureCenterSpan.

7.1.2.19 BB60A_MAX_RT_SPAN

```
#define BB60A_MAX_RT_SPAN (20.0e6)
```

Maximum span (Hz) for BB60A device configured in real-time measurement mode. See bbConfigureCenterSpan.

7.1.2.20 BB60C_MAX_RT_SPAN

```
#define BB60C_MAX_RT_SPAN (27.0e6)
```

Maximum span (Hz) for BB60C/D device configured in real-time measurement mode. See bbConfigureCenterSpan.

7.1.2.21 BB_MIN_USB_VOLTAGE

```
#define BB_MIN_USB_VOLTAGE (4.4)
```

Minimum USB voltage. See bbGetDeviceDiagnostics.

7.1.2.22 BB_MAX_REFERENCE

```
#define BB_MAX_REFERENCE (20.0)
```

Maximum reference level in dBm. See bbConfigureRefLevel.

7.1.2.23 BB_AUTO_ATTEN

```
#define BB_AUTO_ATTEN (-1)
```

Automatically choose attenuation based on reference level. See bbConfigureGainAtten.

7.1.2.24 BB_MAX_ATTEN

```
#define BB_MAX_ATTEN (3)
```

Maximum attentuation. Valid values [0,3] or -1 for auto. See bbConfigureGainAtten.

7.1.2.25 BB_AUTO_GAIN

```
#define BB_AUTO_GAIN (-1)
```

Automatically choose gain based on reference level. See bbConfigureGainAtten.

7.1.2.26 BB_MAX_GAIN

```
#define BB_MAX_GAIN (3)
```

Maximum gain. Valid values [0,3] or -1 for auto. See bbConfigureGainAtten.

7.1.2.27 BB_MIN_DECIMATION

```
#define BB_MIN_DECIMATION (1)
```

No decimation. See bbConfigureIQ.

7.1.2.28 BB_MAX_DECIMATION

```
#define BB_MAX_DECIMATION (8192)
```

Maimumx decimation for I/Q streaming. See bbConfigureIQ.

7.1.2.29 BB_IDLE

```
#define BB_IDLE (-1)
```

Measurement mode: Idle, no measurement. See bbInitiate.

7.1.2.30 BB_SWEEPING

```
#define BB_SWEEPING (0)
```

Measurement mode: Swept spectrum analysis. See bbInitiate.

7.1.2.31 BB_REAL_TIME

```
#define BB_REAL_TIME (1)
```

Measurement mode: Real-time spectrum analysis. See bbInitiate.

7.1.2.32 BB_STREAMING

```
#define BB_STREAMING (4)
```

Measurement mode: I/Q streaming. See bbInitiate.

7.1.2.33 BB_AUDIO_DEMOD

```
#define BB_AUDIO_DEMOD (7)
```

Measurement mode: Audio demod. See bbInitiate.

7.1.2.34 BB_TG_SWEEPING

```
#define BB_TG_SWEEPING (8)
```

Measurement mode: Tracking generator sweeps for scalar network analysis. See bbInitiate.

7.1.2.35 BB_NO_SPUR_REJECT

```
#define BB_NO_SPUR_REJECT (0)
```

Turn off spur rejection. See bbConfigureSweepCoupling.

7.1.2.36 BB_SPUR_REJECT

```
#define BB_SPUR_REJECT (1)
```

Turn on spur rejection. See bbConfigureSweepCoupling.

7.1.2.37 BB_LOG_SCALE

```
#define BB_LOG_SCALE (0)
```

Specifies dBm units of sweep and real-time spectrum analysis measurements. See bbConfigureAcquisition.

7.1.2.38 BB_LIN_SCALE

```
#define BB_LIN_SCALE (1)
```

Specifies mV units of sweep and real-time spectrum analysis measurements. See bbConfigureAcquisition.

7.1.2.39 BB_LOG_FULL_SCALE

```
#define BB_LOG_FULL_SCALE (2)
```

Specifies dBm units, with no corrections, of sweep and real-time spectrum analysis measurements. See bbConfigureAcquisition.

7.1.2.40 BB_LIN_FULL_SCALE

```
#define BB_LIN_FULL_SCALE (3)
```

Specifies mV units, with no corrections, of sweep and real-time spectrum analysis measurements. See bbConfigureAcquisition.

7.1.2.41 BB_RBW_SHAPE_NUTTALL

```
#define BB_RBW_SHAPE_NUTTALL (0)
```

Specifies the Nuttall window used for sweep and real-time analysis. See bbConfigureSweepCoupling.

7.1.2.42 BB_RBW_SHAPE_FLATTOP

```
#define BB_RBW_SHAPE_FLATTOP (1)
```

Specifies the Stanford flattop window used for sweep and real-time analysis. See bbConfigureSweepCoupling.

7.1.2.43 BB_RBW_SHAPE_CISPR

```
#define BB_RBW_SHAPE_CISPR (2)
```

Specifies a Gaussian window with 6dB cutoff used for sweep and real-time analysis. See bbConfigureSweepCoupling.

7.1.2.44 BB_MIN_AND_MAX

```
#define BB_MIN_AND_MAX (0)
```

Use min/max detector for sweep and real-time spectrum analysis. See bbConfigureAcquisition.

7.1.2.45 BB_AVERAGE

```
#define BB_AVERAGE (1)
```

Use average detector for sweep and real-time spectrum analysis. See bbConfigureAcquisition.

7.1.2.46 BB_LOG

```
#define BB_LOG (0)
```

VBW processing occurs in dBm. See bbConfigureProcUnits.

7.1.2.47 BB_VOLTAGE

```
#define BB_VOLTAGE (1)
```

VBW processing occurs in linear voltage units (mV). See bbConfigureProcUnits.

7.1.2.48 BB_POWER

```
#define BB_POWER (2)
```

VBW processing occurs in linear power units (mW). See bbConfigureProcUnits.

7.1.2.49 BB_SAMPLE

```
#define BB_SAMPLE (3)
```

No VBW processing. See bbConfigureProcUnits.

7.1.2.50 BB_DEMOD_AM

```
#define BB_DEMOD_AM (0)
```

Audio demodulation type: AM. See bbConfigureDemod.

7.1.2.51 BB_DEMOD_FM

```
#define BB_DEMOD_FM (1)
```

Audio demodulation type: FM. See bbConfigureDemod.

7.1.2.52 BB_DEMOD_USB

```
#define BB_DEMOD_USB (2)
```

Audio demodulation type: Upper side band. See bbConfigureDemod.

7.1.2.53 BB_DEMOD_LSB

```
#define BB_DEMOD_LSB (3)
```

Audio demodulation type: Lower side band. See bbConfigureDemod.

7.1.2.54 BB_DEMOD_CW

```
#define BB_DEMOD_CW (4)
```

Audio demodulation type: CW. See bbConfigureDemod.

7.1.2.55 BB_STREAM_IQ

```
#define BB_STREAM_IQ (0x0)
```

Default for BB_SWEEPING measurement mode. See bbInitiate.

7.1.2.56 BB_DIRECT_RF

```
#define BB_DIRECT_RF (0x2)
```

For BB60C/D devices. See bbInitiate.

7.1.2.57 BB_TIME_STAMP

```
#define BB_TIME_STAMP (0x10)
```

Time stamp data using an external GPS receiver. See Using a GPS Receiver to Time-Stamp Data and bbInitiate.

7.1.2.58 BB60C_PORT1_AC_COUPLED

```
#define BB60C_PORT1_AC_COUPLED (0x00)
```

Configure BB60A/C port 1 as AC coupled. This is the default. See bbConfigureIO.

7.1.2.59 BB60C_PORT1_DC_COUPLED

```
#define BB60C_PORT1_DC_COUPLED (0x04)
```

Configure BB60A/C port 1 as DC coupled. See bbConfigureIO.

7.1.2.60 BB60C_PORT1_10MHZ_USE_INT

```
#define BB60C_PORT1_10MHZ_USE_INT (0x00)
```

Use BB60A/C internal 10MHz reference. The internal reference is also output on port 1, but it is considered unused. See bbConfigureIO.

7.1.2.61 BB60C_PORT1_10MHZ_REF_OUT

```
#define BB60C_PORT1_10MHZ_REF_OUT (0x100)
```

Output BB60A/C internal 10 MHz reference on port 1. Use this setting when external equipment such as signal generators are using the 10MHz from the BB60. See bbConfigureIO.

7.1.2.62 BB60C_PORT1_10MHZ_REF_IN

```
#define BB60C_PORT1_10MHZ_REF_IN (0x8)
```

Use an external 10MHz provided on BB60A/C port 1. Best phase noise is achieved by using a low jitter 3.3V CMOS input. See bbConfigureIO.

7.1.2.63 BB60C_PORT1_OUT_LOGIC_LOW

```
#define BB60C_PORT1_OUT_LOGIC_LOW (0x14)
```

Output logic low on BB60A/C port 1. See bbConfigureIO.

7.1.2.64 BB60C_PORT1_OUT_LOGIC_HIGH

```
#define BB60C_PORT1_OUT_LOGIC_HIGH (0x1C)
```

Output logic high on BB60A/C port 1. See bbConfigureIO.

7.1.2.65 BB60C_PORT2_OUT_LOGIC_LOW

```
#define BB60C_PORT2_OUT_LOGIC_LOW (0x00)
```

Output logic low on BB60A/C port 2. See bbConfigureIO.

7.1.2.66 BB60C_PORT2_OUT_LOGIC_HIGH

```
#define BB60C_PORT2_OUT_LOGIC_HIGH (0x20)
```

Output logic high on BB60A/C port 2. See bbConfigureIO.

7.1.2.67 BB60C_PORT2_IN_TRIG_RISING_EDGE

```
#define BB60C_PORT2_IN_TRIG_RISING_EDGE (0x40)
```

Detect and report external triggers rising edge on BB60A/C port 2 when I/Q streaming. See bbConfigureIO.

7.1.2.68 BB60C_PORT2_IN_TRIG_FALLING_EDGE

```
#define BB60C_PORT2_IN_TRIG_FALLING_EDGE (0x60)
```

Detect and report external triggers falling edge on BB60A/C port 2 when I/Q streaming. See bbConfigureIO.

7.1.2.69 BB60D_PORT1_DISABLED

```
#define BB60D_PORT1_DISABLED (0)
```

Disable BB60D port 1. See bbConfigureIO.

7.1.2.70 BB60D_PORT1_10MHZ_REF_IN

```
#define BB60D_PORT1_10MHZ_REF_IN (1)
```

Discipline BB60D to an externally generated 10 MHz reference on port 1. See bbConfigureIO.

7.1.2.71 BB60D_PORT2_DISABLED

```
#define BB60D_PORT2_DISABLED (0)
```

Disable BB60D port 2. See bbConfigureIO.

7.1.2.72 BB60D_PORT2_10MHZ_REF_OUT

```
#define BB60D_PORT2_10MHZ_REF_OUT (1)
```

Output B660D internal 10MHz reference on port 2. See bbConfigureIO.

7.1.2.73 BB60D_PORT2_IN_TRIG_RISING_EDGE

```
#define BB60D_PORT2_IN_TRIG_RISING_EDGE (2)
```

Detect and report external triggers rising edge on BB60D port 2 when I/Q streaming. See bbConfigureIO.

7.1.2.74 BB60D_PORT2_IN_TRIG_FALLING_EDGE

```
#define BB60D_PORT2_IN_TRIG_FALLING_EDGE (3)
```

Detect and report external triggers falling edge on BB60D port 2 when I/Q streaming. See bbConfigureIO.

7.1.2.75 BB60D_PORT2_OUT_LOGIC_LOW

```
#define BB60D_PORT2_OUT_LOGIC_LOW (4)
```

Output logic low on BB60D port 2. See bbConfigureIO.

7.1.2.76 BB60D_PORT2_OUT_LOGIC_HIGH

```
#define BB60D_PORT2_OUT_LOGIC_HIGH (5)
```

Output logic high on BB60D port 2. See bbConfigureIO.

7.1.2.77 BB60D_PORT2_OUT_UART

```
#define BB60D_PORT2_OUT_UART (6)
```

Use when any BB60D UART output functionality is desired. See bbConfigureIO.

7.1.2.78 BB60D_UART_BAUD_4_8K

```
#define BB60D_UART_BAUD_4_8K (0)
```

Set 4800 baud rate for BB60D UART transmissions. See bbSetUARTRate.

7.1.2.79 BB60D_UART_BAUD_9_6K

```
#define BB60D_UART_BAUD_9_6K (1)
```

Set 9600 baud rate for BB60D UART transmissions. See bbSetUARTRate.

7.1.2.80 BB60D_UART_BAUD_19_2K

```
#define BB60D_UART_BAUD_19_2K (2)
```

Set 19200 baud rate for BB60D UART transmissions. See bbSetUARTRate.

7.1.2.81 BB60D_UART_BAUD_38_4K

```
#define BB60D_UART_BAUD_38_4K (3)
```

Set 38400 baud rate for BB60D UART transmissions. See bbSetUARTRate.

7.1.2.82 BB60D_UART_BAUD_14_4K

```
#define BB60D_UART_BAUD_14_4K (4)
```

Set 14400 baud rate for BB60D UART transmissions. See bbSetUARTRate.

7.1.2.83 BB60D_UART_BAUD_28_8K

```
#define BB60D_UART_BAUD_28_8K (5)
```

Set 28800 baud rate for BB60D UART transmissions. See bbSetUARTRate.

7.1.2.84 BB60D_UART_BAUD_57_6K

```
#define BB60D_UART_BAUD_57_6K (6)
```

Set 57600 baud rate for BB60D UART transmissions. See bbSetUARTRate.

7.1.2.85 BB60D_UART_BAUD_115_2K

```
#define BB60D_UART_BAUD_115_2K (7)
```

Set 115200 baud rate for BB60D UART transmissions. See bbSetUARTRate.

7.1.2.86 BB60D_UART_BAUD_125K

```
#define BB60D_UART_BAUD_125K (8)
```

Set 125000 baud rate for BB60D UART transmissions. See bbSetUARTRate.

7.1.2.87 BB60D_UART_BAUD_250K

```
#define BB60D_UART_BAUD_250K (9)
```

Set 250000 baud rate for BB60D UART transmissions. See bbSetUARTRate.

7.1.2.88 BB60D_UART_BAUD_500K

```
#define BB60D_UART_BAUD_500K (10)
```

Set 500000 baud rate for BB60D UART transmissions. See bbSetUARTRate.

7.1.2.89 BB60D_UART_BAUD_1000K

```
#define BB60D_UART_BAUD_1000K (11)
```

Set 1000000 baud rate for BB60D UART transmissions. See bbSetUARTRate.

7.1.2.90 BB60D_MIN_UART_STATES

```
#define BB60D_MIN_UART_STATES (2)
```

Minimum number of frequency/data pairs given in bbEnableUARTSweeping. See UART Antenna Switching.

7.1.2.91 BB60D_MAX_UART_STATES

```
#define BB60D_MAX_UART_STATES (8)
```

Maximum number of frequency/data pairs given in bbEnableUARTSweeping. See UART Antenna Switching.

7.1.2.92 TG_THRU_0DB

```
#define TG_THRU_0DB (0x1)
```

In scalar network analysis, use the next trace as a thru. See bbStoreTgThru.

7.1.2.93 TG_THRU_20DB

```
#define TG_THRU_20DB (0x2)
```

In scalar network analysis, improve accuracy with a second thru step. See bbStoreTgThru.

7.1.2.94 TG_REF_UNUSED

```
#define TG_REF_UNUSED (0)
```

Additional corrections are applied to tracking generator timebase. See bbSetTgReference.

7.1.2.95 TG_REF_INTERNAL_OUT

```
#define TG_REF_INTERNAL_OUT (1)
```

Use tracking generator timebase as frequency standard for system, and do not apply additional corrections. See bbSetTgReference.

7.1.2.96 TG_REF_EXTERNAL_IN

```
#define TG_REF_EXTERNAL_IN (2)
```

Use an external reference for TG124A, and do not apply additional corrections to timebase. See bbSetTgReference.

7.1.3 Enumeration Type Documentation

7.1.3.1 bbStatus

```
enum bbStatus
```

Status code returned from all BB API functions. Errors are negative and suffixed with 'Err'. Errors stop the flow of execution, warnings do not.

Enumerator

bbInvalidModeErr	Invalid mode
bbReferenceLevelErr	Reference level cannot exceed 20dBm
bbInvalidVideoUnitsErr	Invalid video processing units specified
bbInvalidWindowErr	Invalid window
bbInvalidBandwidthTypeErr	Invalid bandwidth type
bbInvalidSweepTimeErr	Invalid sweep time
bbBandwidthErr	Invalid bandwidth
bbInvalidGainErr	Invalid gain
bbAttenuationErr	Invalid attenuation
bbFrequencyRangeErr	Frequency range out of bounds
bbInvalidSpanErr	Invalid span
bbInvalidScaleErr	Invalid scale parameter
bbInvalidDetectorErr	Invalid detector type
bbInvalidFileSizeErr	Invalid file size
bbLibusbError	Unable to initialize libusb
bbNotSupportedErr	Attempting to perform an operation on a device that does not support it.
bbTrackingGeneratorNotFound	Tracking generator not found
bbUSBTimeoutErr	USB timeout error
bbDeviceConnectionErr	Device connection issues detected
bbPacketFramingErr	Device packet framing issues
bbGPSErr	GPS receiver not found or not configured properly
bbGainNotSetErr	Gain cannot be auto
bbDeviceNotIdleErr	Function could not complete because the instrument is actively configured for or making a measurement. Call bbAbort and try again.
bbDeviceInvalidErr	Invalid device
bbBufferTooSmallErr	Buffer provided too small
bbNullPtrErr	Returned when one or more required pointer parameter is null.
bbAllocationLimitErr	Allocation limit reached
bbDeviceAlreadyStreamingErr	Device is already streaming
bbInvalidParameterErr	Returned when one or more parameters provided does not match the range of possible values.
bbDeviceNotConfiguredErr	Returned if the device is not properly configured for the desired action. Often occurs when the device needs to be configured for a specific measurement mode before taking an action.
bbDeviceNotStreamingErr	Device not streaming
bbDeviceNotOpenErr	Returned when the device handle provided does not match an open or known device.
bbNoError	Function returned successfully. No warnings or errors.
bbAdjustedParameter	One or more parameters was clamped to a minimum or maximum limit.
bbADCOverflow	ADC overflow
bbNoTriggerFound	No trigger found
bbClampedToUpperLimit	One or more parameters was clamped to a maximum upper limit.
bbClampedToLowerLimit	One or more parameters was clamped to a minimum lower limit.
bbUncalibratedDevice	Device is uncalibrated
bbDataBreak	Break in data
bbUncalSweep	Sweep uncalibrated, data invalid or incomplete.
bbInvalidCalData	Invalid cal data, potentially corrupted

7.1.3.2 bbDataType

```
enum bbDataType
```

Specifies a data type for data returned from the API.

Enumerator

bbDataType32fc	32-bit complex floats
bbDataType16sc	16-bit complex shorts

7.1.3.3 bbPowerState

```
enum bbPowerState
```

Specifies device power state. See Power State for more information.

Enumerator

bbPowerStateOn	On
bbPowerStateStandby	Standby

7.1.4 Function Documentation

7.1.4.1 bbGetSerialNumberList()

```
BB_API bbStatus bbGetSerialNumberList (
              int serialNumbers[BB_MAX_DEVICES],
              int * deviceCount )
```

This function returns the serial numbers for all unopened devices.

The array provided is populated starting at index 0 up to BB_MAX_DEVICES. It is undefined behavior if the array provided contains fewer elements than the number of unopened devices connected. For this reason, it is recommended the array is BB_MAX_DEVICES elements in length. The integer *deviceCount* will contain the number of devices detected. Elements in the array beyond deviceCount are not modified.

Note: BB60A devices will have 0 returned as the serial number.

Parameters

out	*serialNumbers*	A pointer to an array of integers. Can be NULL.
out	*deviceCount*	A pointer to an integer. Will be set to the number of devices found on the system.

Returns

7.1.4.2 bbGetSerialNumberList2()

```
BB_API bbStatus bbGetSerialNumberList2 (
             int serialNumbers[BB_MAX_DEVICES],
             int deviceTypes[BB_MAX_DEVICES],
             int * deviceCount )
```

This function returns the serial numbers and device types for all unopened devices.

The arrays provided are populated starting at index 0 up to BB_MAX_DEVICES. It is undefined behavior if the arrays provided contain fewer elements than the number of unopened devices connected. For this reason, it is recommended the arrays are BB_MAX_DEVICES elements in length. The integer *deviceCount* will contain the number of devices detected. Elements in the arrays beyond deviceCount are not modified.

Note: BB60A devices will have 0 returned as the serial number.

Parameters

out	*serialNumbers*	A pointer to an array of integers. Can be NULL.
out	*deviceTypes*	A pointer to an array of integers. The possible device types returned are BB_DEVICE_BB60A, BB_DEVICE_BB60C, and BB_DEVICE_BB60D. Can be NULL.
out	*deviceCount*	A pointer to an integer. Will be set to the number of devices found on the system.

Returns

7.1.4.3 bbOpenDevice()

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

This function attempts to open the first unopened BB60 it detects. If a device is opened successfully, a handle to the device will be returned through the device pointer which can be used to target that device in subsequent API function calls.

When successful, this function takes about 3 seconds to return. During that time, the calling thread is blocked.

If you wish to target multiple devices or wish to target devices across processes, see Multiple Devices and Multiple Processes.

Parameters

out	*device*	Pointer to an integer. If successful, a device handle is returned. This handle is used for all successive API function calls.

Returns

7.1.4.4 bbOpenDeviceBySerialNumber()

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

The function attempts to open the device with the provided serial number. If no devices are detected with that serial, the function returns an error. If a device is opened successfully, a handle to the device will be returned through the device pointer which can be used to target that device in subsequent API function calls.

Only BB60C/D devices can be opened by specifying the serial number. If the serial number specified is 0, the first BB60A found will be opened.

When successful, this function takes about 3 seconds to return. During that time, the calling thread is blocked.

If you wish to target multiple devices or wish to target devices across processes, see Multiple Devices and Multiple Processes.

Parameters

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

Returns

7.1.4.5 bbCloseDevice()

```
BB_API bbStatus bbCloseDevice (
            int device )
```

This function closes a device, freeing internal allocated memory and USB 3.0 resources. The device closed will become available to be opened again.

This function will abort any active measurements.

Parameters

in	*device*	Device handle.

Returns

7.1.4.6 bbSetPowerState()

```
BB_API bbStatus bbSetPowerState (
            int device,
            bbPowerState powerState )
```

This function is used to set the power state of a BB60D device. This function will return an error for any non BB60D device. The device should not be performing any measurements when calling this function. The device can be configured for sweeps as long as it is not actively performing one.

The device should not perform any measurements while in the standby power state. Ideally no API functions calls should be performed between setting the device into standby power state and returning to the on power state.

See the Power State section for more details as well as the C++ programming example.

Parameters

in	*device*	Device handle.
in	*powerState*	Specify the on or standby power state.

Returns

7.1.4.7 bbGetPowerState()

```
BB_API bbStatus bbGetPowerState (
            int device,
            bbPowerState * powerState )
```

This function returns the current device power state.

See the Power State section for more details as well as the C++ programming example.

Parameters

in	*device*	Device handle.
out	*powerState*	Pointer to power state variable.

Returns

7.1.4.8 bbPreset()

```
BB_API bbStatus bbPreset (
              int device )
```

This function instructs the device to perform a power cycle. This might be useful if the device has entered an unresponsive state. The device must still be able to receive USB commands for this function to work. If the device receives the power cycle command, it will take roughly 3 seconds to fully power cycle.

An example of using this function is provided in the C++ example folder.

Parameters

in	*device*	Device handle.

Returns

7.1.4.9 bbPresetFull()

```
BB_API bbStatus bbPresetFull (
              int * device )
```

This function will fully preset, close, and reopen the device pointed to by the device parameter. This function will only work if the device can still receive USB commands.

Parameters

in	*device*	Pointer to a valid device handle.

Returns

7.1.4.10 bbSelfCal()

```
BB_API bbStatus bbSelfCal (
              int device )
```

This function is for the BB60A only.

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

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

Parameters

in	*device*	Device handle.

Returns

7.1.4.11 bbGetSerialNumber()

```
BB_API bbStatus bbGetSerialNumber (
            int device,
            uint32_t * serialNumber )
```

If this function returns successfully, the variable pointed to by serialNumber will contain the serial number of the specified device.

Parameters

in	*device*	Device handle.
out	*serialNumber*	Returns device serial number as unsigned integer.

Returns

7.1.4.12 bbGetDeviceType()

```
BB_API bbStatus bbGetDeviceType (
            int device,
            int * deviceType )
```

If the function returns successfully, the variable pointed to by deviceType will be one of several device type macro values, BB_DEVICE_NONE, BB_DEVICE_BB60A, BB_DEVICE_BB60C, or BB_DEVICE_BB60D. These macros are defined in the API header file.

Parameters

in	*device*	Device handle.
out	*deviceType*	Returns device type. Can be BB_DEVICE_NONE, BB_DEVICE_BB60A, BB_DEVICE_BB60C, BB_DEVICE_BB60D.

Returns

7.1.4.13 bbGetFirmwareVersion()

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

If the function returns successfully, the variable pointed to by version will be the firmware version of the specified device.

BB60 firmware version information is available on the BB60 downloads page on the Signal Hound website.

Parameters

in	*device*	Device handle.
out	*version*	Returns firmware version number as integer.

Returns

7.1.4.14 bbGetDeviceDiagnostics()

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

The device temperature is updated in the API after each sweep is retrieved, and periodically while I/Q streaming. The temperature is returned in Celsius and has a resolution of 1/8th of a degree. A USB voltage of below 4.4V may cause readings to be out of spec. Check your cable for damage and USB connectors for damage or oxidation.

Parameters

in	*device*	Device handle.
out	*temperature*	Returns device temperature as float.
out	*usbVoltage*	Returns USB voltage as float.
out	*usbCurrent*	Returns usb current as float.

Returns

7.1.4.15 bbConfigureIO()

```
BB_API bbStatus bbConfigureIO (
            int device,
            uint32_t port1,
            uint32_t port2 )
```

The device must be idle when calling this function.

This function configures the two I/O ports on the BB60 rear panel. The values passed to the port parameters will change depending on which device is connected. It is up to the developer to determine which device is connected and send the appropriate parameters.

For a full list of all possible parameters see below. For examples on basic configuration of the ports, see the C++ programming examples in the SDK.

7.1.4.15.1 BB60A/C

7.1.4.15.1.1 Port 1 Port 1 can be configured for 10 MHz ref in/out or as a logic output and accepts the following values:

- BB60C_PORT1_10MHZ_USE_INT Use internal 10 MHz reference. The internal reference is also output on port 1, but it is considered unused.
- BB60C_PORT1_10MHZ_REF_OUT Output internal 10 MHz reference. Use this setting when external equipment such as signal generators are using the 10 MHz from the BB60.
- BB60C_PORT1_10MHZ_REF_IN Use an external 10 MHz. Best phase noise is achieved by using a low jitter 3.3V CMOS input.
- BB60C_PORT1_OUT_LOGIC_LOW Output logic low.
- BB60C_PORT1_OUT_LOGIC_HIGH Output logic high.

Only one of the values above can be selected for port 1. In addition to one of the above values, port 1 can be configured as AC or DC coupled. If AC coupled is desired, simply use one of the macros defined above. If DC coupled is desired, bitwise or "|" one of the above values with BB60C_PORT1_DC_COUPLED.

7.1.4.15.1.2 Port 2 Port 2 can be configured as trigger input port or logic output port. Port 2 is always DC coupled and accepts the following values:

- BB60C_PORT2_OUT_LOGIC_LOW Output logic low.
- BB60C_PORT2_OUT_LOGIC_HIGH Output logic high.
- BB60C_PORT2_IN_TRIG_RISING_EDGE Detect and report external triggers rising edge when I/Q streaming.
- BB60C_PORT2_IN_TRIG_FALLING_EDGE Detect and report external triggers falling edge when I/Q streaming.

7.1.4.15.2 BB60D

7.1.4.15.2.1 Port 1 Port 1 can be configured for reference in. Port 1 is always AC coupled. The following values are accepted:

- BB60D_PORT1_DISABLED Disable port.
- BB60D_PORT1_10MHZ_REF_IN Discipline to an externally generated 10 MHz reference.

7.1.4.15.2.2 Port 2 Port 2 is used for 10 MHz out, trigger input, logic outputs, and UART outputs. Port 2 is always DC coupled. The following values are accepted:

- BB60D_PORT2_DISABLED Disable port.
- BB60D_PORT2_10MHZ_REF_OUT Output internal 10 MHz reference.
- BB60D_PORT2_IN_TRIG_RISING_EDGE Detect and report external triggers rising edge when I/Q streaming.
- BB60D_PORT2_IN_TRIG_FALLING_EDGE Detect and report external triggers falling edge when I/Q streaming.
- BB60D_PORT2_OUT_LOGIC_LOW Output logic low.
- BB60D_PORT2_OUT_LOGIC_HIGH Output logic high.
- BB60D_PORT2_OUT_UART Use when any UART output functionality is desired.

Parameters

in	*device*	Device handle.
in	*port1*	See description.
in	*port2*	See description.

Returns

7.1.4.16 bbSyncCPUtoGPS()

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

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

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

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

Parameters

in	*comPort*	COM port number for the NMEA data output from the GPS receiver.
in	*baudRate*	Baud Rate of the COM port.

Returns

7.1.4.17 bbSetUARTRate()

```
BB_API bbStatus bbSetUARTRate (
             int device,
             int rate )
```

BB60D only.

Sets the baud rate on all UART transmissions. Must be configured when the device is idle. Only the baud rates defined as macros are supported.

See the section on UART Antenna Switching for more information

Parameters

in	*device*	Device handle.
in	*rate*	One of any of the baud rate macros defined in the API header file. Macro names start with BB60D_UART_BAUD_.

Returns

7.1.4.18 bbEnableUARTSweeping()

```
BB_API bbStatus bbEnableUARTSweeping (
             int device,
             const double * freqs,
             const uint8_t * data,
             int states )
```

BB60D only.

Must be called when the device is idle prior to initiating the sweep.

Configures the receiver to transmit UART bytes at certain frequency thresholds. The BB60D steps the frequency spectrum in 20MHz steps, and thus resolution is +/- 20MHz on any given UART transmission.

Frequencies are sorted if they are not provided in increasing order. Regardless of the first frequency provided, the UART byte in position 0 is transmitted at the beginning of the sweep. We recommend simply setting the first frequency to 0Hz.

Port 2 must be configured as an UART output, see bbConfigureIO.

See the section on UART antenna switching for more information.

Parameters

in	*device*	Device handle.
in	*freqs*	Array of frequency thresholds at which to transmit a new UART byte. Size of array must be equal to the number of states.
in	*data*	Array of UART bytes to transmit at the given frequency threshold. Size of array must be equal to the number of states.
in	*states*	Number of freq/data pairs. Alternatively, the size of the provided arrays. Must be between [2,8].

Returns

7.1.4.19 bbDisableUARTSweeping()

```
BB_API bbStatus bbDisableUARTSweeping (
             int device )
```

BB60D only.

Disables UART sweep antenna switching and clears any states set in bbEnableUARTSweeping. Device must be idle when calling this function.

See the section on UART Antenna Switching for more information.

Parameters

in	*device*	Device handle.

Returns

7.1.4.20 bbEnableUARTStreaming()

```
BB_API bbStatus bbEnableUARTStreaming (
             int device,
             const uint8_t * data,
             const uint32_t * counts,
             int states )
```

BB60D only.

Configures the pseudo-doppler antenna switching for I/Q streaming. Must be configured when the device is idle.

When I/Q streaming, the device will cycle through the provided data/count pairs, transmitting the data at that state, then waiting for the specified number of 40MHz clocks. The specified wait count starts as soon as transmission of the start bit occurs.

An external trigger is inserted when the cycle restarts. This allows the ability to determine which I/Q samples correspond to which UART output states. There is a limitation on how quickly external triggers can be generated. The minimum spacing on triggers is ∼250us or 4kHz. If the UART streaming configuration produces triggers at a rate faster than this, you will not receive every triggers. In this situation, you will need to extrapolate missing triggers.

An example:

If two states are provided, with data = { 1, 8 } and counts { 10000, 10000 }, with a baud rate of 1M. While I/Q streaming, the UART byte of ‘1’ starts being transmitted, an external trigger is inserted when the start bit begins transmitting. The UART byte of 1 will take 10us to transmit, (10 bits, 8(data) + 1 (start bit) + 1(stop bit)) and the device will stream for 250us (10k / 40MHz) from when the start bit is transmitted. Once 250us have passed, the start bit for the transmission of ‘8’ is output. Again, it will stream for 250us with a time of 1us for transmitting the 10 bits. At that point it goes back to the first state with a ‘1’ being transmitted.

See the section on UART Antenna Switching for more information.

Parameters

in	*device*	Device handle.
in	*data*	Array of UART bytes to transmit.
in	*counts*	Array of 40MHz clock counts to remain at this state. Values cannot exceed 2^24. Values should also be longer than the time it takes to transmit 10 bits at the selected baud rate.
	states	Number of data/counts pairs. Alternatively, the size of the provided arrays. Must be between [2,8].

Returns

7.1.4.21 bbDisableUARTStreaming()

```
BB_API bbStatus bbDisableUARTStreaming (
             int device )
```

BB60D only.

Disables UART pseudo-doppler switching for I/Q streaming and clears any states set in bbEnableUARTStreaming. Device must be idle when calling this function. See the section on UART Antenna Switching for more information.

Parameters

in	*device*	Device handle.

Returns

7.1.4.22 bbWriteUARTImm()

```
BB_API bbStatus bbWriteUARTImm (
             int device,
             uint8_t data )
```

BB60D only.

Device must be idle when calling this function. Outputs provided byte immediately and returns when complete. Port 2 must be configured for UART output. See bbConfigureIO.

Parameters

in	*device*	Device handle.
in	*data*	Byte to transmit.

Returns

7.1.4.23 bbConfigureRefLevel()

```
BB_API bbStatus bbConfigureRefLevel (
             int device,
             double refLevel )
```

The reference level is used to control the sensitivity of the receiver.

It is recommended to set the reference level ∼5dB higher than the maximum expected input level.

The reference level is only used if the gain and attenuation are set to auto (default). It is recommended to leave gain/atten as auto.

Parameters

in	*device*	Device handle.
in	*refLevel*	Reference level in dBm.

Returns

7.1.4.24 bbConfigureGainAtten()

```
BB_API bbStatus bbConfigureGainAtten (
             int device,
             int gain,
             int atten )
```

This function is used to manually control the gain and attenuation of the receiver. Both gain and attenuation must be set to non-auto values to override the reference level.

It is recommended to leave them as auto. When left as auto, gain and attenuation can be optimized for each band independently. If manually chosen, a flat gain/attenuation is used across all bands.

Below is the gain and attenuation used with each setting.

Setting	BB60C	BB60D
gain = 0	0dB gain	0dB gain
gain = 1	5dB gain	5dB gain
gain = 2	30dB gain	15dB gain
gain = 3	35dB gain	20dB gain
atten = 0	0dB attenuation	0 dB attenuation
atten = 1	10dB attenuation	10dB attenuation
atten = 2	20dB attenuation	20dB attenuation
atten = 3	30dB attenuation	30dB attenuation

Parameters

`in`	*device*	Device handle.
`in`	*gain*	Gain should be between [-1,3] where -1 represents auto.
`in`	*atten*	Atten should be between [-1,3] where -1 represents auto.

Returns

7.1.4.25 bbConfigureCenterSpan()

```
BB_API bbStatus bbConfigureCenterSpan (
              int device,
              double center,
              double span )
```

See bbConfigureIQCenter for configuring I/Q streaming center frequency.

This function configures the sweep frequency range. Start and stop frequencies can be determined from the center and span.

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

During initialization a more precise start frequency and span is determined and returned in the bbQueryTraceInfo function.

The start/stop frequencies cannot exceed [9kHz, 6.4GHz].

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

Certain modes of operation have specific frequency range limits. Those mode dependent limits are tested against during bbInitiate.

Parameters

`in`	*device*	Device handle.
`in`	*center*	Center frequency in Hz.
`in`	*span*	Span in Hz.

Returns

7.1.4.26 bbConfigureSweepCoupling()

```
BB_API bbStatus bbConfigureSweepCoupling (
            int device,
            double rbw,
            double vbw,
            double sweepTime,
            uint32_t rbwShape,
            uint32_t rejection )
```

For standard bandwidths, the API uses the 3 dB points to define the RBW. For the CISPR RBW shape, 6dB bandwidths are used.

The video bandwidth is implemented as an IIR filter applied bin-by-bin to a sequence of overlapping FFTs. Larger RBW/VBW ratios require more FFTs.

The API uses the Stanford flattop window when FLATTOP is selected. The Nuttall window shape trades increased measurement speed for reduces measurement accuracy by adding up to 0.8dB scalloping losses. The CISPR window uses a Gaussian window with 6dB cutoff.

All windows use zero-padding to achieve arbitrary RBWs. Only powers of 2 FFT sizes are used in the API.

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

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

Parameters

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

Returns

7.1.4.27 bbConfigureAcquisition()

```
BB_API bbStatus bbConfigureAcquisition (
            int device,
            uint32_t detector,
            uint32_t scale )
```

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

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

Parameters

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

Returns

7.1.4.28 bbConfigureProcUnits()

```
BB_API bbStatus bbConfigureProcUnits (
            int device,
            uint32_t units )
```

The *units* provided determines scale 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_SAMPLE.

Macro	Unit
BB_LOG	dBm
BB_VOLTAGE	mV
BB_POWER	mW
BB_SAMPLE	No video processing

Parameters

in	*device*	Device handle.
in	*units*	The possible values are BB_LOG, BB_VOLTAGE, BB_POWER, BB_SAMPLE.

Returns

7.1.4.29 bbConfigureRealTime()

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

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

Parameters

in	*device*	Device handle.
in	*frameScale*	Specifies the height in dB of the real-time frame. The value is ignored if the scale is linear. Possible values range from [10 – 200].
in	*frameRate*	Specifies the rate at which frames are generated in real-time mode, in frames per second. Possible values range from [4 – 30], where four means a frame is generated every 250ms and 30 means a frame is generated every ∼33 ms.

Returns

7.1.4.30 bbConfigureRealTimeOverlap()

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

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

Parameters

in	*device*	Device handle.
in	*advanceRate*	FFT advance rate.

Returns

7.1.4.31 bbConfigureIQCenter()

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

Configure the center frequency for I/Q streaming.

When switching back to sweep acquisition from I/Q acquisition, you will need to call the bbConfigureCenterSpan function again.

The center frequency must be between [BB_MIN_FREQ, BB_MAX_FREQ]

Parameters

in	*device*	Device handle.
in	*centerFreq*	Center frequency in Hz.

Returns

7.1.4.32 bbConfigureIQ()

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

Configure the sample rate and bandwidth of the I/Q data stream.

The decimation rate divides the I/Q sample rate. The final sample rate is calculated with the equation `40MHz / downsampleFactor`. *downsampleFactor* must be a power of 2 between [1,8192].

bandwidth specifies the 3dB bandwidth of the I/Q data stream. *bandwidth* controls the filter cutoff of a FIR filter applied to the data stream prior to decimation.

For each given decimation rate, a maximum bandwidth value is specified to account for sufficient filter roll off. A table of maximum bandwidths can be found in I/Q Filtering and Bandwidth Limitations.

Parameters

in	*device*	Device handle.
in	*downsampleFactor*	Specify a decimation rate for the 40MS/s I/Q stream.
in	*bandwidth*	Specify a bandpass filter width for the I/Q stream.

Returns

7.1.4.33 bbConfigureIQDataType()

```
BB_API bbStatus bbConfigureIQDataType (
            int device,
            bbDataType dataType )
```

See I/Q Data Types for more information.

Parameters

in	*device*	Device handle.
in	*dataType*	Data type can be specified either as 32-bit complex floats or 16-bit complex shorts.

Returns

7.1.4.34 bbConfigureIQTriggerSentinel()

```
BB_API bbStatus bbConfigureIQTriggerSentinel (
            int sentinel )
```

See the I/Q Streaming for more information on triggering and how the trigger sentinel value is used.

Parameters

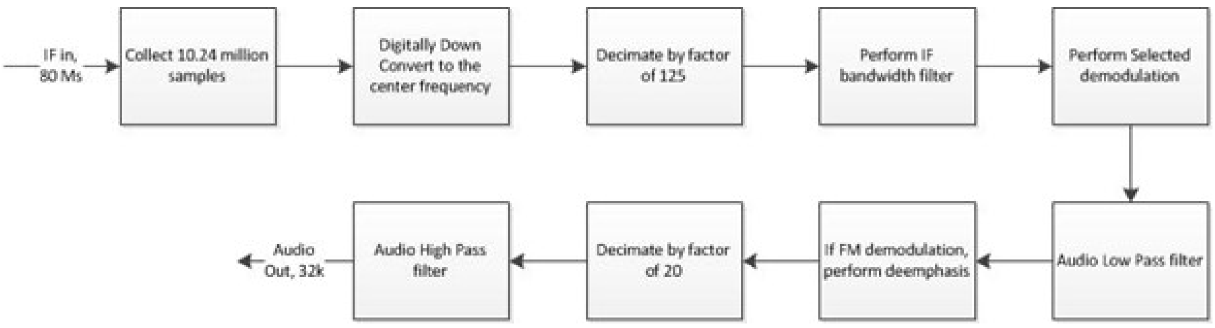
in	*sentinel*	Value used to fill the remainder of the trigger buffer when the trigger buffer provided is larger than the number of triggers returned. The default sentinel value is zero.

Returns

7.1.4.35 bbConfigureDemod()

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

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


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

This function can be called while the device is active.

Parameters

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

Returns

7.1.4.36 bbInitiate()

```
BB_API bbStatus bbInitiate (
             int device,
             uint32_t mode,
             uint32_t flag )
```

This function configures the device into a state determined by the *mode* parameter. For more information regarding operating states, refer to the Theory of Operation and Measurement Types 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.

Parameters

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

Returns

7.1.4.37 bbAbort()

```
BB_API bbStatus bbAbort (
            int device )
```

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

Parameters

in	*device*	Device handle.

Returns

7.1.4.38 bbQueryTraceInfo()

```
BB_API bbStatus bbQueryTraceInfo (
            int device,
            uint32_t * traceLen,
            double * binSize,
            double * start )
```

This function should be called to determine sweep characteristics after a device has been configured and initiated for sweep mode.

Parameters

in	*device*	Device handle.
out	*traceLen*	Pointer to uint32_t to contain the size of the sweeps returned from bbFetchTrace.
out	*binSize*	Pointer to double, to contain the frequency delta between two sequential bins in the returned sweep.
out	*start*	Pointer to double to contains the frequency of the first bin in the sweep.

Returns

7.1.4.39 bbQueryRealTimeInfo()

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

This function should be called after initializing the device for real-time mode.

Parameters

in	*device*	Device handle.
out	*frameWidth*	Pointer to uint32_t to contain the width of the real-time frame in bins.
out	*frameHeight*	Pointer to uint32_t to contain the height of the real-time frame in bins.

Returns

7.1.4.40 bbQueryRealTimePoi()

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

The device must be configured for real-time spectrum analysis to call this function.

Parameters

in	*device*	Device handle.
out	*poi*	Pointer to double to contain the 100% probability of intercept duration in seconds.

Returns

7.1.4.41 bbQueryIQParameters()

```
BB_API bbStatus bbQueryIQParameters (
             int device,
             double * sampleRate,
             double * bandwidth )
```

The device must be configured for I/Q streaming to call this function.

Parameters

in	*device*	Device handle.
out	*sampleRate*	Pointer to double to contain the I/Q streaming sample rate in Hz.
out	*bandwidth*	Pointer to double to contain the I/Q streaming bandwidth.

Returns

7.1.4.42 bbGetIQCorrection()

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

Retrieve the I/Q correction factor for I/Q streaming with 16-bit complex shorts. The device must be configured for I/Q streaming to call this function.

Parameters

in	*device*	Device handle.
out	*correction*	Pointer to float to contain the scalar value used to convert full scale I/Q data to amplitude corrected I/Q. The formulas for these conversions are in I/Q Data Types. Cannot be null.

Returns

7.1.4.43 bbFetchTrace_32f()

```
BB_API bbStatus bbFetchTrace_32f (
             int device,
             int arraySize,
             float * traceMin,
             float * traceMax )
```

Get one sweep in sweep mode. If the detector type is set to average, the *traceMin* array returned will equal max, in which case NULL can be used for the min parameter.

The first element returned in the sweep corresponds to the *startFreq* returned from bbQueryTraceInfo.

Parameters

in	*device*	Device handle.
in	*arraySize*	This parameter is deprecated and ignored.
out	*traceMin*	Pointer to a float array whose length should be equal to or greater than *traceSize* returned from bbQueryTraceInfo. Set to NULL to ignore this parameter.
out	*traceMax*	Pointer to a float array whose length should be equal to or greater than *traceSize* returned from bbQueryTraceInfo. Set to NULL to ignore this parameter.

Returns

7.1.4.44 bbFetchTrace()

```
BB_API bbStatus bbFetchTrace (
             int device,
             int arraySize,
             double * traceMin,
             double * traceMax )
```

Get one sweep in sweep mode. If the detector type is set to average, the *traceMin* array returned will equal max, in which case NULL can be used for the min parameter.

The first element returned in the sweep corresponds to the *startFreq* returned from bbQueryTraceInfo.

Parameters

in	*device*	Device handle.
in	*arraySize*	This parameter is deprecated and ignored.
out	*traceMin*	Pointer to a double array whose length should be equal to or greater than *traceSize* returned from bbQueryTraceInfo. Set to NULL to ignore this parameter.
out	*traceMax*	Pointer to a double array whose length should be equal to or greater than *traceSize* returned from bbQueryTraceInfo. Set to NULL to ignore this parameter.

Returns

7.1.4.45 bbFetchRealTimeFrame()

```
BB_API bbStatus bbFetchRealTimeFrame (
            int device,
            float * traceMin,
            float * traceMax,
            float * frame,
            float * alphaFrame )
```

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

Parameters

in	*device*	Device handle.
out	*traceMin*	If this pointer is non-null, the min held sweep will be returned to the user. If the detector is set to average, this array will be identical to the *traceMax* array.
out	*traceMax*	If this pointer is non-null, the max held sweep will be returned to the user. If the detector is set to average, this array contains the averaged results over the measurement interval.
out	*frame*	Pointer to a float array. If the function returns successfully, the contents of the array will contain a single real-time frame.
out	*alphaFrame*	Pointer to a float array. If the function returns successfully, the contents of the array will contain the alphaFrame corresponding to the frame. Can be NULL.

Returns

7.1.4.46 bbGetIQ()

```
BB_API bbStatus bbGetIQ (
            int device,
            bbIQPacket * pkt )
```

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

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

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

Parameters

in	*device*	Device handle.
out	*pkt*	Pointer to a bbIQPacket structure.

Returns

7.1.4.47 bbGetIQUnpacked()

```
BB_API bbStatus bbGetIQUnpacked (
            int device,
            void * iqData,
            int iqCount,
            int * triggers,
            int triggerCount,
            int purge,
            int * dataRemaining,
            int * sampleLoss,
            int * sec,
            int * nano )
```

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

Parameters

in	*device*	Device handle.
out	*iqData*	Pointer to an array of 32-bit complex floating-point values. Complex values are interleaved real-imaginary pairs. This must point to a contiguous block of *iqCount* complex pairs.
in	*iqCount*	Number of I/Q data pairs to return.
out	*triggers*	Pointer to an array of integers. If the external trigger input is active, and a trigger occurs during the acquisition time, triggers will be populated with values which are relative indices into the *iqData* array where external triggers occurred. Any unused trigger array values will be set to zero.
in	*triggerCount*	Size of the triggers array.
in	*purge*	Specifies whether to discard any samples acquired by the API since the last time a bbGetIQ function was called. Set to BB_TRUE if you wish to discard all previously acquired data, and BB_FALSE if you wish to retrieve the contiguous I/Q values from a previous call to this function.
out	*dataRemaining*	How many I/Q samples are still left buffered in the API.
out	*sampleLoss*	Returns BB_TRUE or BB_FALSE. Will return BB_TRUE when the API is required to drop data due to internal buffers wrapping. This can be caused by I/Q samples not being polled fast enough, or in instances where the processing is not able to keep up (underpowered systems, or other programs utilizing the CPU) Will return BB_TRUE on the capture in which the sample break occurs. Does not indicate which sample the break occurs on. Will always return false if *purge* is true.
out	*sec*	Seconds since epoch representing the timestamp of the first sample in the returned array.
out	*nano*	Nanoseconds representing the timestamp of the first sample in the returned array.

Returns

7.1.4.48 bbFetchAudio()

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

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.

Parameters

in	*device*	Device handle.
out	*audio*	Pointer to an array of 4096 32-bit floating point values.

Returns

7.1.4.49 bbAttachTg()

```
BB_API bbStatus bbAttachTg (
             int device )
```

This function connects a TG device and associates it with the specified BB60 device. Once the TG is opened, it cannot be opened in any other application until the BB60 is closed via bbCloseDevice.

Parameters

in	*device*	Device handle.

Returns

7.1.4.50 bbIsTgAttached()

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

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

Parameters

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

Returns

7.1.4.51 bbConfigTgSweep()

```
BB_API bbStatus bbConfigTgSweep (
            int device,
            int sweepSize,
            bool highDynamicRange,
            bool passiveDevice )
```

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

Parameters

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

Returns

7.1.4.52 bbStoreTgThru()

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

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

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

Parameters

in	*device*	Device handle.
in	*flag*	Specify the type of store thru. Possible values are TG_THRU_0DB, TG_THRU_20DB.

Returns

7.1.4.53 bbSetTg()

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

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

Parameters

in	*device*	Device handle.
in	*frequency*	Set the frequency, in Hz, of the TG output.
in	*amplitude*	Set the amplitude, in dBm, of the TG output.

Returns

7.1.4.54 bbGetTgFreqAmpl()

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

Retrieve the last set TG output parameters the user set through the bbSetTg function. The bbSetTg function must have been called for this function to return valid values. If the TG was used to perform scalar network analysis at any point, this function will not return valid values until the bbSetTg function is called again. If a previously set parameter was clamped in the bbSetTg function, this function will return the final clamped value. If any pointer parameter is null, that value is ignored and not returned.

Parameters

in	*device*	Device handle.
out	*frequency*	Pointer to a double that will contain the last set frequency of the TG output in Hz.
out	*amplitude*	Pointer to a double that will contain the last set amplitude of the TG output in dBm.

Returns

7.1.4.55 bbSetTgReference()

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

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

Parameters

in	*device*	Device handle.
in	*reference*	A valid time base setting value. Possible values are TG_REF_UNUSED, TG_REF_INTERNAL_OUT, TG_REF_EXTERNAL_IN.

Returns

7.1.4.56 bbGetAPIVersion()

```
BB_API const char * bbGetAPIVersion ( )
```

Get API version.

Returns

The returned string is of the form *major.minor.revision*. Ascii periods (".") separate positive integers. Major/↩ Minor/Revision are not guaranteed to be a single decimal digit. The string is null terminated. An example string: ['3' | '.' | '0' | '.' | '1' | '1' | '\0'] = "3.0.11"

7.1.4.57 bbGetProductID()

```
BB_API const char * bbGetProductID ( )
```

Get product ID.

Returns

The returned string is of the form ####-####.

7.1.4.58 bbGetErrorString()

```
BB_API const char * bbGetErrorString (
              bbStatus status )
```

Get an ascii string description of a provided status code.

Parameters

in	*status*	A bbStatus value returned from an API call.

Returns

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

7.2 bb_api.h

Go to the documentation of this file.

```
// Copyright (c).2014-2022, Signal Hound
// For licensing information, please see the API license in the software_licenses folder

#ifndef BB_API_H
#define BB_API_H

#if defined(_WIN32)
#ifdef BB_EXPORTS
#define BB_API __declspec(dllexport)
#else
#define BB_API __declspec(dllimport)
#endif

    // bare minimum stdint typedef support
#if _MSC_VER < 1700 // For VS2010 or earlier
        typedef signed char        int8_t;
        typedef short              int16_t;
        typedef int                int32_t;
        typedef long long          int64_t;
        typedef unsigned char      uint8_t;
        typedef unsigned short     uint16_t;
        typedef unsigned int       uint32_t;
        typedef unsigned long long uint64_t;
#else
#include <stdint.h>
#endif

#define BB_DEPRECATED(comment) __declspec(deprecated(comment))
#else // Linux
#define BB_API __attribute__((visibility("default")))

```

```
#include <stdint.h>

#if defined(__GNUC__)
#define BB_DEPRECATED(comment) __attribute__((deprecated))
#else
#define BB_DEPRECATED(comment) comment
#endif
#endif

#define BB_TRUE (1)
#define BB_FALSE (0)

#define BB_DEVICE_NONE (0)
#define BB_DEVICE_BB60A (1)
#define BB_DEVICE_BB60C (2)
#define BB_DEVICE_BB60D (3)

#define BB_MAX_DEVICES (8)

#define BB_MIN_FREQ (9.0e3)
#define BB_MAX_FREQ (6.4e9)

#define BB_MIN_SPAN (20.0)
#define BB_MAX_SPAN (BB_MAX_FREQ - BB_MIN_FREQ)

#define BB_MIN_RBW (0.602006912)
#define BB_MAX_RBW (10100000.0)

#define BB_MIN_SWEEP_TIME (0.00001) // 10us
#define BB_MAX_SWEEP_TIME (1.0) // 1s

#define BB_MIN_RT_RBW (2465.820313)
#define BB_MAX_RT_RBW (631250.0)
#define BB_MIN_RT_SPAN (200.0e3)
#define BB60A_MAX_RT_SPAN (20.0e6)
#define BB60C_MAX_RT_SPAN (27.0e6)

#define BB_MIN_USB_VOLTAGE (4.4)

#define BB_MAX_REFERENCE (20.0)

#define BB_AUTO_ATTEN (-1)
#define BB_MAX_ATTEN (3)
#define BB_AUTO_GAIN (-1)
#define BB_MAX_GAIN (3)

#define BB_MIN_DECIMATION (1)
#define BB_MAX_DECIMATION (8192)

#define BB_IDLE (-1)
#define BB_SWEEPING (0)
#define BB_REAL_TIME (1)
#define BB_STREAMING (4)
#define BB_AUDIO_DEMOD (7)
#define BB_TG_SWEEPING (8)

#define BB_NO_SPUR_REJECT (0)
#define BB_SPUR_REJECT (1)

#define BB_LOG_SCALE (0)
#define BB_LIN_SCALE (1)
#define BB_LOG_FULL_SCALE (2)
#define BB_LIN_FULL_SCALE (3)

#define BB_RBW_SHAPE_NUTTALL (0)
#define BB_RBW_SHAPE_FLATTOP (1)
#define BB_RBW_SHAPE_CISPR (2)

#define BB_MIN_AND_MAX (0)
#define BB_AVERAGE (1)

#define BB_LOG (0)
#define BB_VOLTAGE (1)
#define BB_POWER (2)
#define BB_SAMPLE (3)

#define BB_DEMOD_AM (0)
#define BB_DEMOD_FM (1)
#define BB_DEMOD_USB (2)
#define BB_DEMOD_LSB (3)
#define BB_DEMOD_CW (4)

#define BB_STREAM_IQ (0x0)
#define BB_DIRECT_RF (0x2)
#define BB_TIME_STAMP (0x10)

#define BB60C_PORT1_AC_COUPLED (0x00)
```

```
#define BB60C_PORT1_DC_COUPLED (0x04)
#define BB60C_PORT1_10MHZ_USE_INT (0x00)
#define BB60C_PORT1_10MHZ_REF_OUT (0x100)
#define BB60C_PORT1_10MHZ_REF_IN (0x8)
#define BB60C_PORT1_OUT_LOGIC_LOW (0x14)
#define BB60C_PORT1_OUT_LOGIC_HIGH (0x1C)
#define BB60C_PORT2_OUT_LOGIC_LOW (0x00)
#define BB60C_PORT2_OUT_LOGIC_HIGH (0x20)
#define BB60C_PORT2_IN_TRIG_RISING_EDGE (0x40)
#define BB60C_PORT2_IN_TRIG_FALLING_EDGE (0x60)

#define BB60D_PORT1_DISABLED (0)
#define BB60D_PORT1_10MHZ_REF_IN (1)
#define BB60D_PORT2_DISABLED (0)
#define BB60D_PORT2_10MHZ_REF_OUT (1)
#define BB60D_PORT2_IN_TRIG_RISING_EDGE (2)
#define BB60D_PORT2_IN_TRIG_FALLING_EDGE (3)
#define BB60D_PORT2_OUT_LOGIC_LOW (4)
#define BB60D_PORT2_OUT_LOGIC_HIGH (5)
#define BB60D_PORT2_OUT_UART (6)

#define BB60D_UART_BAUD_4_8K (0)
#define BB60D_UART_BAUD_9_6K (1)
#define BB60D_UART_BAUD_19_2K (2)
#define BB60D_UART_BAUD_38_4K (3)
#define BB60D_UART_BAUD_14_4K (4)
#define BB60D_UART_BAUD_28_8K (5)
#define BB60D_UART_BAUD_57_6K (6)
#define BB60D_UART_BAUD_115_2K (7)
#define BB60D_UART_BAUD_125K (8)
#define BB60D_UART_BAUD_250K (9)
#define BB60D_UART_BAUD_500K (10)
#define BB60D_UART_BAUD_1000K (11)

#define BB60D_MIN_UART_STATES (2)
#define BB60D_MAX_UART_STATES (8)

#define TG_THRU_0DB (0x1)
#define TG_THRU_20DB (0x2)

#define TG_REF_UNUSED (0)
#define TG_REF_INTERNAL_OUT (1)
#define TG_REF_EXTERNAL_IN (2)

typedef struct bbIQPacket {
    void *iqData;
    int iqCount;
    int *triggers;
    int triggerCount;
    int purge;
    int dataRemaining;
    int sampleLoss;
    int sec;
    int nano;
} bbIQPacket;

typedef enum bbStatus {
    bbInvalidModeErr             = -112,
    bbReferenceLevelErr          = -111,
    bbInvalidVideoUnitsErr       = -110,
    bbInvalidWindowErr           = -109,
    bbInvalidBandwidthTypeErr    = -108,
    bbInvalidSweepTimeErr        = -107,
    bbBandwidthErr               = -106,
    bbInvalidGainErr             = -105,
    bbAttenuationErr             = -104,
    bbFrequencyRangeErr          = -103,
    bbInvalidSpanErr             = -102,
    bbInvalidScaleErr            = -101,
    bbInvalidDetectorErr         = -100,

    bbInvalidFileSizeErr         = -19,
    bbLibusbError                = -18,
    bbNotSupportedErr            = -17,
    bbTrackingGeneratorNotFound  = -16,

    bbUSBTimeoutErr              = -15,
    bbDeviceConnectionErr        = -14,
    bbPacketFramingErr           = -13,
    bbGPSErr                     = -12,
    bbGainNotSetErr              = -11,
    bbDeviceNotIdleErr           = -10,
    bbDeviceInvalidErr           = -9,
    bbBufferTooSmallErr          = -8,
    bbNullPtrErr                 = -7,
    bbAllocationLimitErr         = -6,
    bbDeviceAlreadyStreamingErr  = -5,
    bbInvalidParameterErr        = -4,
    bbDeviceNotConfiguredErr     = -3,
    bbDeviceNotStreamingErr      = -2,
    bbDeviceNotOpenErr           = -1,

    bbNoError                    = 0,

    // Warnings/Messages

    bbAdjustedParameter          = 1,
    bbADCOverflow                = 2,
    bbNoTriggerFound             = 3,
    bbClampedToUpperLimit        = 4,
    bbClampedToLowerLimit        = 5,
    bbUncalibratedDevice         = 6,
    bbDataBreak                  = 7,
    bbUncalSweep                 = 8,
    bbInvalidCalData             = 9
} bbStatus;

typedef enum bbDataType {
    bbDataType32fc = 0,
    bbDataType16sc = 1
} bbDataType;

typedef enum bbPowerState {
    bbPowerStateOn = 0,
    bbPowerStateStandby = 1
} bbPowerState;

#ifdef __cplusplus
extern "C" {
#endif

BB_API bbStatus bbGetSerialNumberList(int serialNumbers[BB_MAX_DEVICES], int *deviceCount);

BB_API bbStatus bbGetSerialNumberList2(int serialNumbers[BB_MAX_DEVICES],
                                       int deviceTypes[BB_MAX_DEVICES],
                                       int *deviceCount);

BB_API bbStatus bbOpenDevice(int *device);

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

BB_API bbStatus bbCloseDevice(int device);

BB_API bbStatus bbSetPowerState(int device, bbPowerState powerState);

BB_API bbStatus bbGetPowerState(int device, bbPowerState *powerState);

BB_API bbStatus bbPreset(int device);

BB_API bbStatus bbPresetFull(int *device);

BB_API bbStatus bbSelfCal(int device);

BB_API bbStatus bbGetSerialNumber(int device, uint32_t *serialNumber);

BB_API bbStatus bbGetDeviceType(int device, int *deviceType);

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

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

BB_API bbStatus bbConfigureIO(int device, uint32_t port1, uint32_t port2);

BB_API bbStatus bbSyncCPUtoGPS(int comPort, int baudRate);

BB_API bbStatus bbSetUARTRate(int device, int rate);

BB_API bbStatus bbEnableUARTSweeping(int device, const double *freqs, const uint8_t *data, int states);

BB_API bbStatus bbDisableUARTSweeping(int device);

BB_API bbStatus bbEnableUARTStreaming(int device, const uint8_t *data, const uint32_t *counts, int states);

BB_API bbStatus bbDisableUARTStreaming(int device);

BB_API bbStatus bbWriteUARTImm(int device, uint8_t data);

BB_API bbStatus bbConfigureRefLevel(int device, double refLevel);

BB_API bbStatus bbConfigureGainAtten(int device, int gain, int atten);

BB_API bbStatus bbConfigureCenterSpan(int device, double center, double span);

BB_API bbStatus bbConfigureSweepCoupling(int device, double rbw, double vbw, double sweepTime,
                                         uint32_t rbwShape, uint32_t rejection);

BB_API bbStatus bbConfigureAcquisition(int device, uint32_t detector, uint32_t scale);

BB_API bbStatus bbConfigureProcUnits(int device, uint32_t units);

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

BB_API bbStatus bbConfigureRealTimeOverlap(int device, double advanceRate);

BB_API bbStatus bbConfigureIQCenter(int device, double centerFreq);

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

BB_API bbStatus bbConfigureIQDataType(int device, bbDataType dataType);

BB_API bbStatus bbConfigureIQTriggerSentinel(int sentinel);

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

BB_API bbStatus bbInitiate(int device, uint32_t mode, uint32_t flag);

BB_API bbStatus bbAbort(int device);

BB_API bbStatus bbQueryTraceInfo(int device, uint32_t *traceLen, double *binSize, double *start);

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

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

BB_API bbStatus bbQueryIQParameters(int device, double *sampleRate, double *bandwidth);

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

BB_API bbStatus bbFetchTrace_32f(int device, int arraySize, float *traceMin, float *traceMax);

BB_API bbStatus bbFetchTrace(int device, int arraySize, double *traceMin, double *traceMax);

BB_API bbStatus bbFetchRealTimeFrame(int device, float *traceMin, float *traceMax, float *frame, float
 *alphaFrame);

BB_API bbStatus bbGetIQ(int device, bbIQPacket *pkt);

BB_API bbStatus bbGetIQUnpacked(int device, void *iqData, int iqCount, int *triggers,
                                int triggerCount, int purge, int *dataRemaining,
                                int *sampleLoss, int *sec, int *nano);

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

BB_API bbStatus bbAttachTg(int device);

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

BB_API bbStatus bbConfigTgSweep(int device, int sweepSize, bool highDynamicRange, bool passiveDevice);

BB_API bbStatus bbStoreTgThru(int device, int flag);

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

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

BB_API bbStatus bbSetTgReference(int device, int reference);

BB_API const char* bbGetAPIVersion();

BB_API const char* bbGetProductID();

BB_API const char* bbGetErrorString(bbStatus status);

// Deprecated functions, use suggested alternatives

// Use bbConfigureRefLevel instead
BB_API bbStatus bbConfigureLevel(int device, double ref, double atten);
// Use bbConfigureGainAtten instead
BB_API bbStatus bbConfigureGain(int device, int gain);
// Use bbQueryIQParameters instead
BB_API bbStatus bbQueryStreamInfo(int device, int *return_len, double *bandwidth, int
 *samples_per_sec);

#ifdef __cplusplus
} // extern "C"
#endif

// Deprecated macros, use alternatives where available
```

```
#define BB60_MIN_FREQ (BB_MIN_FREQ)
#define BB60_MAX_FREQ (BB_MAX_FREQ)
#define BB60_MAX_SPAN (BB_MAX_SPAN)
#define BB_MIN_BW (BB_MIN_RBW)
#define BB_MAX_BW (BB_MAX_RBW)
#define BB_MAX_ATTENUATION (30.0) // For deprecated bbConfigureLevel function
#define BB60C_MAX_GAIN (BB_MAX_GAIN)
#define BB_PORT1_INT_REF_OUT (0x00)
#define BB_PORT1_EXT_REF_IN (BB60C_PORT1_10MHZ_REF_IN)
#define BB_RAW_PIPE (BB_STREAMING)
#define BB_STREAM_IF (0x1) // No longer supported
// Use new device specific port 1 macros
#define BB_PORT1_AC_COUPLED (BB60C_PORT1_AC_COUPLED)
#define BB_PORT1_DC_COUPLED (BB60C_PORT1_DC_COUPLED)
#define BB_PORT1_10MHZ_USE_INT (BB60C_PORT1_10MHZ_USE_INT)
#define BB_PORT1_10MHZ_REF_OUT (BB60C_PORT1_10MHZ_REF_OUT)
#define BB_PORT1_10MHZ_REF_IN (BB60C_PORT1_10MHZ_REF_IN)
#define BB_PORT1_OUT_LOGIC_LOW (BB60C_PORT1_OUT_LOGIC_LOW)
#define BB_PORT1_OUT_LOGIC_HIGH (BB60C_PORT1_OUT_LOGIC_HIGH)
// Use new device specific port 2 macros
#define BB_PORT2_OUT_LOGIC_LOW (BB60C_PORT2_OUT_LOGIC_LOW)
#define BB_PORT2_OUT_LOGIC_HIGH (BB60C_PORT2_OUT_LOGIC_HIGH)
#define BB_PORT2_IN_TRIGGER_RISING_EDGE (BB60C_PORT2_IN_TRIG_RISING_EDGE)
#define BB_PORT2_IN_TRIGGER_FALLING_EDGE (BB60C_PORT2_IN_TRIG_FALLING_EDGE)

#endif // BB_API_H
```

Index