BB60A Application Programming Interface(API)

Programmers Reference Manual

Justin Crooks & Andrew Montgomery

Version 1.0

Requirements, Operation, Function Definitions, Examples

Table of Contents

Overview

This manual is a programmer's reference for the BB60A API. The API is a C-based set of routines used to control the BB60A device.

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

The *Requirements* section details the physical and operational needs to use the API.
The *Theory of Operation* section is arguably the most important section of this document. It details how to interface this device and covers every major component a program will implement when interfacing the BB60A.
The *Modes of Operation* section attempts to teach you how to use the device in each of its operation modes, from the required functions, to understanding the data the device returns.
The *API Functions* section details every function in depth. The knowledge learned in the Theory and Modes of operation sections will help you navigate the API functions.
The *Appendix* provides various code examples and tips.

Contact Information

We are interested in your feedback and questions. Please report any issues/bugs as soon as possible. We will maintain the most up to date API on our website. We encourage any and all criticisms or ideas. We would love to hear how you might improve the API.

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

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

Requirements

Below is a list of requirements needed to begin.

1) Windows 7. The API is untested outside Windows 7.
2) Windows C/C++ development tools/environment. Preferably Visual Studio 2008 or later.
3) The BB60APIW32.h API header file.
4) The BB60 API library(.lib) and dynamic library(.dll) files.
5) A basic understanding of RF Spectrum Analysis.
6) A BB60A Spectrum analyzer.

Theory of Operation

The flow of any program interfacing the BB60A device will be as follows.

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

The API provides functions for each step in this process. We have strived to mimic the functionality and naming conventions of SCPI's *IviSpecAn* Class Specification where possible. It is not necessary to be familiar with this specification but those who are should feel comfortable with our API immediately. Let's look at each step in detail of a typical program interfacing the BB60A.

Opening a Device

Before attempting to open a device programmatically, it must be physically connected to a USB 3.0 port with the provided cable. Ensure the power light is lit on the device and is solid. Once the device is connected it can be opened. The function bbOpenDevice() provides this functionality. This function returns an integer ID to the device which was opened. Up to eight devices may be connected and interfaced through our API using the ID's. The integer ID is required for every function call in the API, as it uniquely identifies which BB60A you are interfacing.

Configuring the Device

Once the device is opened, it must be configured. Most of the functionality of this API is geared towards configuring the device. All configure functions will modify the devices' global state. Device state is discussed more in the next section (Initiating the device). The API provides configure routines for groupings of related variables. Each function is described in depth below. All relevant configuration routines should be invoked. Each configure routine should be examined below in detail. Also note the boundary conditions for each variable. Some variables will have different boundary conditions given the mode you will later be trying to operate in. Each function description below will detail these boundaries. We have also provided helpful #defines in the API header file to help check against these boundaries. All configure routines will provide error/warning messages when these boundaries are crossed.

Initiate the Device

Each device has two states.

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

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

The BB60A has multiple modes of operation. The bbInitiate() function is used to enter one of the operational states. The device can only be in one operational state at a time. These modes are described in the "Modes of Operation" section.

Retrieve Data from the Device

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

Abort the Current Mode

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

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

Close Device

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

Calibration

Calibration is an important part of the BB60A operation. The BB60A is temperature sensitive and it is important the device is always re-calibrated when significant temperature shifts occur (+/- 2 °C). The BB60A is a streaming device and as such cannot automatically calibrate itself without interrupting operation/communication (which may be undesirable). Therefore we leave calibration to the programmer. The API provides two functions for assisting with live calibration, *bbQueryDiagnostics* and *bbSelfCal. bbQueryDiagnostics* can be used to retrieve the internal device temperature at any time after the device has been opened. If the device ever deviates from its temperature we suggest calling *bbSelfCal.* Calling *bbSelfCal* requires the device be open and idle. After a self-calibration occurs, the global device state is undefined. It is necessary to reconfigure the device before continuing operation. One self-calibration is performed upon opening the device.

Modes of Operation

Now that we have seen how a typical application interfaces with the device, let's examine the different modes of operation the BB60A provides. Each mode will accept different configurations and have different boundary conditions. Each mode will also provide data formatted to match the mode selected. In the next sections you will see how to interact with each mode.

For a more in-depth examination of each mode of operation (read: *theory*) refer to the BB60A user manual.

Swept Analysis

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

The configuration routines which affect the sweep results are
- Acquisition
- CenterSpan
- Level
- Gain
- SweepCoupling
- Window
- ProcUnits

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

This mode is driven by the programmer, causing a sweep to be collected only when the program requests one. The number and size of Fourier transforms are determined by the combination of bandwidths and sweep time.

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

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

Determining the frequency of any point returned is determined by the function where 'n' is a zero based sample point.

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

Real-Time Analysis

The BB60A API provides the functionality of an online real-time spectrum analyzer for a 20MHz bandwidth. Through the use of FFTs at an overlapping rate of 75%, the spectrum results have no blind time (100% probability of intercept). Due to the demands in processing, restrictions are placed on resolution bandwidth, and video bandwidth is non-configurable.

The configuration routines which affect the spectrum results are
- Acquisition
- CenterSpan, span must be less than or equal to 20MHz. Span restrictions are defined in the API header as BB_MIN/MAX_RT_RBW.
- Level
- Gain
- SweepCoupling, restricted bandwidths, defined in the API header file as BB_MIN/MAX_RT_RBW.

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

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

Time-Gate Analysis

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

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

bbNoTriggerFound warning is returned. It is also possible the trigger is found too late in the total capture not allowing enough room for the gate. This will also trigger the noTrigger warning.

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

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

Zero-Span

Zero-Span allows you to view a signal in time domain. Zero Span mode returns amplitude or frequency on the Y-axis, and time is always on the X-axis. This is useful for viewing modulation or frequency steps.

Configuration routines which affect sweeps are
- CenterSpan, center represents the frequency you want to view in the time domain, span is irrelevant here and is ignored.
- SweepCoupling, rbw, vbw, and sweep time are all critical to returned sweeps
- Acquisition
- Atten
- Level
- Gain
- ProcUnits
- Trigger

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

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

Raw Data

Raw Data mode collects the raw, uncorrected spectrum at a 20MHz bandwidth. 16-bit signed ADC values, at 80 MSPS, are returned. A filter centered at 20 MHz, with a 20 MHz bandwidth, minimizes aliasing. Only the spectrum from approx. 10 to 30 MHz is valid.

Configuration routines which affect the output are
- CenterSpan
- Level
- Gain

Center frequency is clamped to the closest 400kHz value. It is very important to only select center frequencies that are multiples of 400kHz so you are gauranteed to get the center you want. The reasoning for this is to attempt to get you the cleanest signal possible.

Once configured, initiate the device in BB_RAW_PIPE mode. Data acquisition begins immediately. Use the *bbFetchRaw* function to collect 299008 samples. This function is blocking when data is not ready, and must be called approximately 267 times per second. The API queues up to 120ms of data (32 fetches) before the queue becomes full and drops data. It is very important to have minimal processing in the data collection thread to prevent this from happening.

If you want to convert these values to a more space efficient data type, we recommend signed shorts, scaling the returned values by 32768.

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

If you wish to retrieve absolute amplitude on the data, see *bbFetchRawCorrections*.

Raw Sweep

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

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

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

Raw Sweep Loop

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

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

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

Sweeping versus Streaming

All modes of operation fall within two categories, sweeping and streaming. In any sweeping mode, the device operates only when requested. Usually requesting a trace triggers a single trace acquisition, otherwise the device and API are idle. Sweeping is very responsive and switching between difference types of sweep modes is very quick. Streaming modes are modes in which the API is continually receiving a stream of spectrum (20MHz) from the device. The characterization of the data is a result of the initial configuration. The device is never idle in these modes. Once this process is started, it takes about ½ second to abort any streaming operation, to ensure all channels/pipes have been cleared and the device is ready for its next command.

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

Depending on your application this ½ second abort time may not be acceptable (switching bands quickly/changing settings quickly). If you are interested in utilizing a streaming mode to fully characterize a signal of interest, a good approach might be to start in the standard sweep mode or the raw sweep mode. From these modes you can simply detect the location of a signal of interest and quickly react by switching into a stream mode with appropriate settings.

API Functions

bbOpenDevice
Open a single BB60 device

bbStatus bbOpenDevice(
        int * *device*
);

Parameters:
device

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

Description:
This function when successful takes about 3 seconds to perform.

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

Return Values:
bbNoError

> No error, device number opened and returned successfully.

bbNullPtrErr

> The parameter *device* is null. The device is not opened.

bbDeviceNotOpenErr

The device was unable to open. This can be returned for many reasons such as, the device is not physically connected, eight devices are already open, or the USB 3.0 connection is faulty.

bbCloseDevice
Close a single BB60 device

bbStatus bbCloseDevice(
      int *device*
);

Parameters:
device

The number of the device you wish to close.

Description:
This function is called when you wish to terminate a connection with a BB60A device. Any resources the device has allocated and the USB 3.0 connection to the device is terminated. The device closed will be released and will become available to be opened again.

Return Values:
bbNoError

The device closed successfully.

bbDeviceNotOpenErr

The device specified through the *device* parameter is not open.

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

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

Parameters:
device

The device handle to the device being configured.

detectorType

The two possible values for detector type are BB_AVERAGE and BB_MIN_AND_MAX.

verticalScale

The four possible values for verticalScale are BB_LOG_SCALE, BB_LIN_SCALE, BB_LOG_FULL_SCALE, and BB_LIN_FULL_SCALE.

Description:

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

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

Return Values:

bbNoError

No error.

bbDeviceNotOpenErr

The device handle provided points to a device that is not open.

bbInvalidParameterErr

One or more parameters provided did not match the accepted inputs.

bbConfigureCenterSpan

Change the center and span frequencies

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

Parameters:

device

The device handle returned from bbOpenDevice

center

Center frequency in Hz

Span

Span in Hz

Description:

This function configures the operating frequency band of the BB60A device. Start and stop frequencies can be determined from the center and span.
- start = center – (span/2)
- stop = center+(span/2)

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

The BB60A device has an operating range of 9 kHz to 6 GHz. There are defines found in the API header you can use to check against. If the start or stop frequencies provided fall outside the operation range an error will be returned.

The BB60A device also has an absolute minimum operating span, as well as operation mode dependent minimums. The absolute minimum is caught here and operation dependent minimums are caught in bbInitiate.

Return Values:
bbNoError

Device successfully configured

bbDeviceNotOpenErr

The device handle provided points to a device that is not open.

bbInvalidParameterErr

A boundary condition was not met. The start or stop frequencies provided through center and span fall outside the operating range. The span may be too large, or too small.

bbConfigureLevel
Change the center and span frequencies

bbStatus bbConfigureCenterSpan(
        int *device,*
        double *ref,*
        double *atten*
);

Parameters:
device

The device handle returned from bbOpenDevice

ref

Reference level in either dBm or mV

atten

Attenuation setting in dB

Description:

The reference level provided should be in dBm or mV, depending on whether you plan to operate the device in logarithmic or linear scale respectively. Which scale is used is determined by the function *bbConfigureAcquisition*.

When automatic *gain* is selected, the API uses the *reference level* provided to choose the best gain settings for an input signal with amplitude equal to reference level. If a gain other than BB_AUTO_GAIN is specified using *bbConfigureGain*, the *reference level* parameter is ignored.

The *atten* parameter controls the RF input attenuator, and is adjustable from 0 to 31 dB in 1 dB steps. The RF attenuator is the first gain control device in the front end.

Return Values:
bbNoError

Device successfully configured

bbDeviceNotOpenErr

The device handle provided points to a device that is not open.

bbInvalidParameterErr

A provided parameter has fallen outside the device operation range. Attenuation must be between 0 and 30 dB.

bbConfigureGain
Change the RF/IF gain path in the device

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

Parameters:
device

The device handle returned from bbOpenDevice

gain

A gain setting. The possible values are BB_AUTO_GAIN, BB_NO_GAIN, BB_LOW_GAIN, BB_MED_GAIN, or BB_HIGH_GAIN.

Description:

Any changes to gain through this function will not be reflected until the next time the device is initiated. To return the device to automatically choose the best gain setting, call this function with a gain of BB_AUTO_GAIN.

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

After the RF input attenuator (0-31 dB), the RF path contains an additional amplifier stage after band filtering, which is selected for medium or high gain and bypassed for low or no gain.

Additionally, the IF has an amplifier which is bypassed only for BB_NO_GAIN.

For BB_HIGH_GAIN, additional amplification in the ADC stage is used.

Return Values:
bbNoError

Device successfully configured

bbDeviceNotOpenErr

The device handle provided does not point to an open device.

bbInvalidParameterErr

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

bbConfigureSweepCoupling
Change the sweep coupling values

bbStatus bbConfigureSweepCoupling(
    int *device,*
    double *rbw,*
    double *vbw,*
    double *sweepTime,*
    unsigned int *rbwType,*
    unsigned int *rejection*
);

Parameters:
device

The device handle returned from bbOpenDevice

rbw

Resolution bandwidth in Hz

vbw

Video bandwidth (VBW) in Hz. VBW must be less than or equal to RBW.

sweepTime

Sweep time in seconds. In sweep mode, this value is how long the BB60A collects data before it begins processing. In the real-time configuration, this value represents the length of time data is collected and compounded before returning a sweep. In zero span mode this is the amount of time in seconds you wish to display on the x-axis.

rbwType

The possible values for rbwType are BB_NATIVE_RBW and BB_NON_NATIVE_RBW. This choice determines which

	bandwidth table is used and how the data is processed. BB_NATIVE_RBW is default and unchangeable for real-time operation.
rejection	The possible values for rejection are BB_NO_SPUR_REJECT, BB_SPUR_REJECT, and BB_BYPASS_RF.

Description:

The resolution bandwidth, or RBW, represents the bandwidth of spectral energy represented in each frequency bin. For example, with an RBW of 10 kHz, the amplitude value for each bin would represent the total energy from 5 kHz below to 5 kHz above the bin's center. For standard bandwidths, the BB60A uses the 3 dB points to define the RBW.

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

Native RBWs represent the bandwidths from a single power-of-2 FFT using our sample rate of 80 MSPS and a high dynamic range window function. Each RBW is half of the previous. Using native RBWs can give you the lowest possible bandwidth for any given sweep time, and minimizes processing power. However, scalloping losses of up to 0.8 dB, occurring when a signal falls in between two bins, can cause problems for some types of measurements.

Non-native RBWs use the traditional 1-3-10 sequence. Programmatically, non-native RBWs are implemented by integrating the power across several adjacent bins in each FFT result, eliminating scalloping losses at the cost of a wider RBW and more processing time. Non-native bandwidths force video processing to use "power" units. Furthermore, the maximum achievable video bandwidth in non-native mode is limited by the algorithm, and is typically an order of magnitude lower than the RBW even if a higher VBW selected. If a high VBW is required, use native RBWs. If lower VBW is acceptable and a standard 1-3-10 sequence is desired, use non-native RBWs.

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

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

Rejection can be used to optimize certain aspects of the signal. Default is BB_NO_SPUR_REJECT, and should be used in most cases. If you have a steady CW or slowly changing signal, and need to minimize image and spurious responses from the BB60A, use BB_SPUR_REJECT. If you have a signal between 300

MHz and 3 GHz, need the lowest possible phase noise, and do not need any image rejection, BB_BYPASS_RF can be used to rewire the front end for lowest phase noise.

Return Values:
bbNoError

>Device successfully configured

bbDeviceNotOpenErr

>The device handle provided points to a device that is not open.

bbInvalidParameterErr

>Device unsuccessfully configured. There are a number of potential possibilities which will invoke this error.
>- sweepTime is less than BB_MIN_SWEEP_TIME
>- vbw is greater than rbw.
>- rbw/vbw falls outside bandwidth limits defined in BB60APIW32.h

bbConfigureWindow
Change the windowing function

bbStatus bbConfigureWindow(
>**int** *device,*
>**unsigned int** *window*

);

Parameters:
device

>The device number returned from bbOpenDevice

window

>The possible values for window are BB_NUTALL, BB_BLACKMAN, BB_HAMMING, and BB_FLAT_TOP.

Description:
This changes the windowing function applied to the data before signal processing is performed. In real-time configuration the window parameter is permanently set to BB_NUTALL.

Return Values:
bbNoError

>Device successfully configured

bbDeviceNotOpen

>The device handle provided points to a device that is not open.

bbInvalidParameterErr

>The value for window did not match any known value

bbConfigureProcUnits

Configure

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

Parameters:

device

      The device number returned from bbOpenDevice

units

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

Description:

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

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

BB_LOG	dBm
BB_VOLTAGE	mV
BB_POWER	mW
BB_BYPASS	No video processing

Return Values:

bbNoError

      Device successfully configured

bbDeviceNotOpen

      The device handle provided points to a device that is not open.

bbInvalidParameterErr

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

bbConfigureTrigger

Configure the Zero-Span trigger

bbStatus bbConfigureTrigger(
      int *device*,
      unsigned int *type*,
      unsigned int *edge*,

```
        double level,
        double timeout
);
```

Parameters

device

> Device ID

type

> Specifies the type of trigger to use. Possible values are
> BB_NO_TRIGGER, BB_VIDEO_TRIGGER,
> BB_EXTERNAL_TRIGGER, and BB_GPS_PPS_TRIGGER. If an
> external signal is desired, BNC port 2 must be configured to
> accept a trigger (see *bbConfigureIO*).

edge

> Specifies the edge type of a video trigger. Possible values
> are BB_TRIGGER_RISING and BB_TRIGGER_FALLING. If you
> are using a trigger type other than a video trigger, this
> value is ignored but must be specified.

level

> Level of the trigger. The units of this value are determined
> by the demodulation type used when initiating the device.
> If demodulating AM, *level* is in dBm units, if demodulating
> FM, *level* is in Hz.

timeout

> The amount of time you are willing to wait for a trigger in
> seconds. If that time is reached, a sweep over the last set of
> data is returned.

Description

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

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

Return Values

bbNoError

> Configured successfully

bbDeviceNotOpenErr

> The device specified is not open.

bbInvalidParameterErr

> Parameter specified is not valid.

bbConfigureTimeGate

Configure Time-Gate mode values

bbStatus bbConfigureTimeGate (
  int *device*,
  double *gateDelay*,
  double *gateLength*,
  double *timeout*
);

Parameters
device
       Device ID

gateDelay
       The time in seconds, from the trigger to the beginning of
       the gate

gateLength
       The length in seconds, of the gate

timeout
       The time in seconds to wait for a trigger. If no trigger is
       found, the last "gateLength" will be used.

Description
Allows you to configure the gate parameters used for time-gated analysis. As with all configure routines, the changes here are not reflected until the next initiate.

The gate is relative to an external trigger, therefore it is necessary to use *bbConfigureIO* to setup an external trigger.

Return Values
bbNoError
       Device configured successfully

bbDeviceNotOpenErr
       Device specified is not open.

bbInvalidParameterErr
       A supplied parameter is unknown or out of range.

bbConfigureRawSweep
Prepare the device to collect swept ADC data.

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

Parameters

device

Handle to the device

start

Frequency value in MHz representing the center of the first 20MHz bandwidth in the sweep. Start must be a multiple of 20.

ppf

Packets per frequency, the amount of data to collect at each frequency. The number of samples at each frequency equals 18688 * ppf.

steps

Number of steps to take in the sweep

stepsize

Value must be BB_TWENTY_MHZ

Description

This function is called prior to *bbInitiate* and prepares the device for the collection of data in the BB_RAW_SWEEP mode. This function allows you to detail the sweep characteristics, using a starting center frequency, and specifying the number of 20MHz steps to take, and the number of samples to collect at each frequency. Read the restrictions below and view the example in the Appendix for further information.

Restrictions

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

Return Values

bbNoError

The device was successfully configured.

bbDeviceNotOpenErr

The device specified is not open.

bbInvalidParameter

One or more of the functions requirements were not met.

bbConfigureIO

Configure the two I/O ports on the BB60 device

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

Parameters

device

port1

Device ID

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

port2

Port 2 is capable of accepting an external trigger or generating a logic output. Port 2 is always DC coupled. All possible values for this port are found in the header file and are prefixed with "BB_PORT2."

Description

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

There are two configurable BNC connector ports available on the BB60. Both ports functionality are changed with this function. For both ports, '0' is the default and can be supplied through this function to return the ports to their default values. Specifying a '0' on port 1 returns the device to an internal time base and outputs the time base AC coupled. Specifying '0' on port 2 emits a DC coupled logic low.

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

Configure combinations

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

Return Values

bbNoError

Device configured successfully.

bbDeviceNotOpenErr

Device specified is not open.

bbDeviceNotIdleErr

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

bbInvalidParameterErr

A parameter supplied is unknown.

Example

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

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

bbInitiate

Change the operating state of the device

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

Parameters:

device

The device number returned from bbOpenDevice

mode

The possible values for *mode* are BB_SWEEPING, BB_REAL_TIME, BB_ZERO_SPAN, and BB_RAW_PIPE.

flag

The default value is zero.

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

If mode equals BB_RAW_PIPE *flag* can be BB_TIME_STAMP which informs the API to time stamp the raw packets using a external GPS receiver. See *Appendix:Using a GPS Receiver to Time-Stamp Data* for information on how to set this up.

Description:

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

Pay special attention to the bbInvalidParameterErr description below.

Return Values:

bbNoError

Device successfully configured

bbDeviceNotOpenErr

The device handle provided does not point to an open device.

bbInvalidParameterErr

The value for *mode* did not match any known value.

In real-time mode, this value may be returned if the span limits defined in the API header are broken. Also in real-time mode, this error will be returned if the resolution bandwidth is outside the limits defined in the API header.

In time-gate analysis mode this error will be returned if span limits defined in the API header are broken. Also in time gate analysis, this error is returned if the bandwidth provided require more samples for processing than is allowed in the gate length. To fix this, increase rbw/vbw.

bbAllocationLimitError

This value is returned in extreme circumstances. The BB60A currently limits the amount of RAM usage to 1GB. When exceptional parameters are provided, such as very low bandwidths, or long sweep times, this error may be returned. At this point you have reached the boundaries of the device. The BB60A processing algorithms are optimized for speed at the expense of space, which is the reason this can occur.

bbQueryTraceInfo

Returns values needed to query and analyze traces

bbStatus bbQueryTraceInfo(
    int *device,*
    unsigned int **traceSize,*
    double **binSize,*
    double **startFreq*
);

Parameters:

device

      The device number returned from bbOpenDevice

traceSize

      A pointer to an unsigned int. If the function returns successfully traceSize will contain the size of arrays returned by bbFetchTrace.

binSize

      A pointer to a 64bit floating point variable. If the function returns successfully, binSize will contain the frequency difference between two sequential bins in a returned sweep. In Zero-Span mode, binSize refers to the difference between sequential samples in seconds.

startFreq

      A pointer to a 64bit floating point variable. If the function returns successfully, startFreq will contain the frequency of the first bin in a returned sweep. In Zero-Span mode, start frequency represents the exact center frequency used by the API.

Description:

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

Return Values:

bbNoError

      Successful

bbNullPtrErr

      If any pointer passed as a parameter is null, bbNullPtrErr will be returned and no values will be returned.

bbDeviceNotOpenErr

      The *device* provided does not refer to an open device.

bbDeviceNotConfiguredErr

      The device is not in a known operational state or is idle. This error will also be returned if the device is in BB_RAW_PIPE mode.

bbFetchTrace

Get one sweep from a configured and initiated device

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

Parameters:

device

> The number of an opened and configured device which you want to get a sweep on.

arraySize

> A provided arraySize. This value must be equal to or greater than the *traceSize* value returned from *bbQueryTraceInfo.*

min

> Pointer to a double buffer, whose length is equal to or greater than *traceSize* returned from *bbQueryTraceInfo.*

max

> Pointer to a double buffer, whose length is equal to or greater than *traceSize* returned from *bbQueryTraceInfo.*

Description:

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

Return Values:

bbNoError

> Successful. pSweepDataMin/Max are populated with amplitude values.

bbNullPtrErr

> If either *min* or *max* are null, bbNullPtrErr is returned immediately.

bbDeviceNotOpenErr

> *device* is not a handle to an open device.

bbDeviceNotConfiguredErr

> Returned if the device is idle or in BB_RAW_PIPE mode.

bbBufferTooSmallErr

> The *arraySize* parameter passed is less than the trace size returned from *bbQueryTraceInfo.*

bbADCOverflow

> This warning is returned when the ADC detects clipping of the input signal. This occurs when the maximum voltage has been reached. Signal analysis and reconstruction become issues on clipped signals. To prevent this, a combination of increasing attenuation, decreasing gain, or increasing reference level(when gain is automatic) will allow for more headroom.

bbNoTriggerFound

> In time-gated analysis, if the spectrum returned is not representative of the gate specified, this warning is returned.

bbPacketFramingErr

This error occurs when data loss or miscommunication has occurred between the device and the API. During normal operation we do not expect this error to occur. If you find this error occurs frequently, it may be indicative of larger issues. If this error is returned, the data returned is undefined. The device should be power cycled manually or with the *bbPreset* routine.

bbFetchRawCorrections

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

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

Parameters:
device

      Handle to an open and configured device.

corrections

      32-bit float array of length 2048. Correction values are decibel.

index

      Index into the *corrections* array where the correction data begins.

startFreq

      Frequency associated with the correction at *index.*

Description:
When this function returns successfully, the correction array will contain the frequency domain correction constants for the given 20 MHz band chosen. The corrections are modified based on temperature, gain, attenuation, and frequency. If any of these change, a new correction array should be requested. The correction array will only be generated again on a new *bbInitiate()*.

The correction array represents 40 MHz of bandwidth where frequencies outside the requested 20 MHz are zeroed out. The first non-zero sample begins at *corrections[index].* The frequency at this index is *startFreq.* The bin size of each index is implied through 40 MHz divided by the length of the array, (40.0e6 / 2048) = 19531.25 Hz.

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

Frequency domain correction of the signal's amplitude requires you to first transform the raw data into the frequency domain. If a 4096 length Fourier Transform is used on a portion of the time domain data, the result is 2048 samples representing frequencies from 0 to 40 MHz. (Remember, the sample rate is 80MHz). The 20MHz band requested will now line up with the non-zero corrections in the correction array. You can index the Transform results using the *index* returned from this function. If larger Transform sizes are desired, you can interpolate the correction array to the desired size. (Be aware! This will change the index of the first non-zero correction)

Return Values:
bbNoError

        Function returned successfully

bbNullPtrErr

        One or more parameters provided is null

bbDeviceNotOpenErr

        The device specified is not open

bbDeviceNotConfigured

        The device specified is not currently configured in *raw-pipe* mode

bbFetchRaw
Retrieve raw data from the BB60A

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

Parameters:
device

        The number of an open and configured for real-time device

buffer

        A pointer to an array of 32-bit floating point values of length 299008.

triggers

        triggers is a pointer to an array of 17 integers representing external trigger information relative to the buffer. Read the description below for in-depth discussion.

bbPacketFramingErr

        This error occurs when data loss or miscommunication has occurred between the device and the API. During normal operation we do not expect this error to occur. If you find this error occurs frequently, it may be indicative of larger issues. If this error is returned, the data returned is undefined. The device should be power cycled manually or with the *bbPreset* routine.

Description:

Fetch a chunk of data while the device is in the raw data pipe mode. A buffer of data represents 299008/80000000 seconds worth of data (~3.7376ms). The data will represent a 20MHz span at the specified center frequency. To ensure full spectrum coverage, this function must be called every 299008/80000000 seconds. The API buffers up to 32, 299008 chunks of raw data (~120ms) before it begins dropping data.

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

The *triggers* parameter can be NULL if you are not interested in trigger position, otherwise *triggers* should point to an array of 17 32-bit integers. Starting at triggers[0], positive values will indicate positions within the returned *buffer* array where an external trigger occurred. The positions are zero based, meaning the positions will be between ~90 and 299007. (Note: the minimum trigger position detected is approximately 90) If no triggers occurred during the acquisition of the raw data, all values will be 0. If for example, 3 external triggers occurred during the acquisition, the first three values of the *triggers* array will be non-negative, and the remaining equal to 0. A returned trigger array might look like this.

triggerArray[17] = [917, 46440, 196264, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0];
This array indicates three external triggers were detected at *buffer[*917*], buffer[*46440*],* and *buffer[*196264*].* They will always be in increasing order.

Note: The ports on the BB60A device need to be configured to receive external triggers to take advantage of the trigger array.

See the Code Examples for an example of this function.

The data points returned are floating point values ranging between -1.0 and 1.0. The values are in the time domain and are uncorrected. (*See bbFetchRawCorrections() for information on making amplitude corrections on the raw data*)

Return Values:
bbNoError
> The device successfully began streaming.

bbDeviceNotOpenErr
> *device* is not a handle to an open device.

bbDeviceNotConfiguredErr
> The device has not been configured for retrieving raw data

bbNullPtrErr
> This is returned if buffer is a null pointer.

bbFetchRawSweep
Retrieve a single sweep in the raw sweep mode

bbStatus bbFetchRawSweep (
      int *device,*
      short **buffer*

);

Parameters:
device

Device handle

buffer

Pointer to an array of signed short integers

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

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

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

Return Values:
bbNoError

Function returned successfully. Buffer contains the entire sweep.

bbNullPtrErr

buffer is null

bbDeviceNotOpenErr

Device specified is not open

bbDeviceNotConfiguredErr

Device specified is not configured and initiated for raw sweeps.

bbPacketFramingErr

This error occurs when data loss or miscommunication has occurred between the device and the API. During normal operation we do not expect this error to occur. If you find this error occurs frequently, it may be indicative of larger issues. If this error is returned, the data returned is undefined. The device should be power cycled manually or with the *bbPreset* routine.

bbStartRawSweepLoop
Begin the raw sweep loop

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

Parameters:

device

        Handle to an open device

buffer

        Pointer to an array of signed shorts. *Buffer* length specified by *bufferLength*

bufferLength

        The number of shorts *buffer*. This number must be a multiple of 299008.

bufferIndex

        Pointer to an unsigned integer. Will be updated to reflect the next index where the API will write data.

bufferCounter

        Pointer to an unsigned integer. Will be incremented each time the buffer is filled.

Description:

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

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

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

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

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

Return Values:

bbNoError

        The function returned successfully and is now sweeping.

bbNullPtrErr

        Returned if either *buffer, bufferIndex,* or *bufferCounter* is null.

bbDeviceNotOpenErr

The device specified is not open.

bbDeviceNotConfiguredErr

The device has not be configured and initiated for raw sweep loop mode.

bbDeviceAlreadyStreamingErr

The device is already sweeping, indicating this function has already been called.

bbQueryTimestamp

Retrieve an absolute time of a data packet

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

Parameters:

device

device handle

seconds

Seconds since midnight (00:00:00), January 1, 1970, coordinated universal time(UTC).

nanoseconds

nanoseconds between seconds and seconds+1

Description:

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

Return Values:

bbNoError

Successful

bbNullPtrErr

seconds or nanoseconds parameters are null.

bbDeviceNotOpenErr

device is not a handle to an open device.

bbDeviceNotConfiguredErr

The device is not configured and running in RAW_PIPE mode.

bbSyncCPUtoGPS

Synchronize a GPS reciever with the API

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

Parameters:

comPort

Com port number for the NMEA data output from the GPS reciever.

baudRate

Baud Rate of the Com port.

Description:

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

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

Return Values:

bbNoError

Successful, describe what it means to be successful

bbDeviceNotOpenErr

No device is open at the time this function is called.

bbGPSErr

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

bbAbort
Stop the current mode of operation

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

Parameters:

device

The number of an open and configured device

Description:

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

Return Values:

bbNoError

> The device has been successfully suspended.

bbDeviceNotOpenErr

> The device indicated by *device* is not open.

bbDeviceNotConfiguredErr

> The device is already idle.

bbPreset

Trigger a device reset

bbStatus bbPreset(
    int *device*
);

Parameters:

device

> An open device handle.

Description:

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

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

Return Values:

bbNoError

> Function completed successfully, the device will be reset.

bbDeviceNotOpen

> The device specified is not currently open.

Example

```
1.  // Notes: Invoking a sleep in the main thread of execution may be undesirable
2.  //   in a GUI application. This function is best performed in a separate thread.
3.  // The amount of time to Sleep is dependent on how fast the device will register
4.  //   on your machine after it resets. A longer sleep time may be preferred or multiple

5.  //   attempts to open the device until it returns bbNoError
```

```
6.  bool PresetRoutine() {
7.
8.    bbPreset( myID );
9.    bbCloseDevice( myID );
10.
11.   Sleep(3000); // Windows sleep function
12.
13.   // Alternative 1: Assume it's ready
14.   if( bbOpenDevice( &myID ) == bbNoError )
15.     return true;
16.   else
17.     return false;
18.
19.
20.   // Alternative 2: Try a few times, it may not be ready at first
21.   int trys = 0;
22.   while(trys++ < 3) {
23.     if( bbOpenDevice( &myID ) == bbNoError )
24.       return true;
25.     else
26.       Sleep(500);
27.   }
28.   return false;
29. }
```

bbSetUSBTimeout

Set a timeout for USB 3.0 communications

bbStatus bbSetUSBTimeout(
      int device,
      unsigned int *mode*
);

Parameters:
device

          The device handle

mode

          Enables or disables the USB timeout timer. Valid parameters are BB_ENABLE or BB_DISABLE. It is disabled by default.

Description:
This function exists to assist in the functionality of the BB60A device in the current USB 3.0 environment. In very rare circumstances we have encountered sporadic packet loss in the communication between the device and the host (PC). This behavior has been exhibited in very specific circumstances, usually a result of a USB driver/motherboard combination. Because much of the communication between the device and host results in blocking program execution, packet loss needs to be addressed.

Return Values:
bbNoError

          Successfully updated the timer

bbDeviceNotOpenErr

The device handle provided does not correlate to an open device.

bbInvalidParameterErr

The mode parameter is unrecognized. The current state of the timer remains unchanged.

bbQueryDiagnostics

Retrieve the current internal device characteristics

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

Parameters:

device

Device Handle

temperature

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

voltage1_8

Factory use only: Internal regulator.

voltage1_2

Factory use only: Internal regulator.

voltageUSB

USB operating voltage, in volts. Acceptable ranges are 4.40 to 5.25 V.

currentUSB

USB current draw, in mA. Acceptable ranges are 800 – 1000 mA.

Description:

Pass NULL to any parameter you do not wish to query.

The device temperature is updated in the API after each sweep is retrieved. The temperature is returned in Celsius and has a resolution of 1/8th of a degree. A temperature above 70°C or below 0°C indicates your device is operating outside of its normal operating temperature, and may cause readings to be out of spec, and may damage the device.

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

Return Values:

bbNoError

> Successfully retrieved the temperature

bbDeviceNotOpenErr

> Device specified is not currently open/valid

bbSelfCal

Calibrate the device for significant temperature changes.

bbStatus bbSelfCal(
      int *device*
);

Parameters:

device

> A device handle.

Description:

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

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

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

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

Return Values:

bbNoError

> The device was recalibrated successfully.

bbDeviceNotOpenErr

> The device specified is either not open or valid.

bbGetSerialNumber

Retrieve the serial number of the device

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

Parameters:

Description:

This function may be called only after the device has been opened. The serial number returned should match the number on the BB60 device case.

Return Values:

bbNoError

> Successfully retrieved the serial number. *sid* will contain the serial number.

bbDeviceNotOpenErr

> The device specified is not open

bbNullPtrErr

> The parameter *sid* is NULL

bbGetErrorString

Produce an error string from an error code

const char * bbGetErrorString(
      bbStatus *code*
);

Parameters:

code

> A bbStatus value returned from an API call.

Description:

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

Return Values:

const char *

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

Error Handling

All BB60 API functions return the type **bbStatus**. **bbStatus** is an enumerated type representing the success of a given function call. The return values can be found in BB60APIW32.h. There are three types of returned status codes.

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

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

Appendix

Code Examples

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

Common

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

```
1.  #define CHECK_BB_STATUS(status)          \
2.    if(status != bbNoError) {              \
3.      logError(bbGetErrorString(status));  \
4.      doErrorHandlingRoutine(status);      \
5.    }
```

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

Sweep Mode

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

```
1.  /* Open a device, configure it, and retrieve a sweep */
2.
3.  bbStatus status;
4.  int devID, traceSize;
5.  double *min, *max, binSize, startFreq;
6.
7.  /* Open the device, retrieve the device handle */
8.  status = bbOpenDevice( &devID );
9.  if( status != bbNoError )
10.   myErrorRoutine( status );
```

```
11.
12. /* Simple configuration */
13.
14. bbConfigureAcquisition(
15.    devID,
16.    BB_MIN_AND_MAX,
17.    BB_LOG_SCALE  // Log scaled results
18. );
19.
20. bbConfigureCenterSpan(
21.    devID,
22.    900.0e6, // 900 MHz center
23.    20.0e6  // 20 MHz span
24. );
25.
26. bbConfigureLevel(
27.    devID,
28.    0.0, // 0 db Reference level
29.    10.0 // 10 db Attenuation
30. );
31.
32. bbConfigureGain(
33.    devID,
34.    BB_AUTO_GAIN
35. );
36.
37. bbConfigureSweepCoupling(
38.    devID,
39.    10.0e3, // 10 kHz rbw
40.    10.0e3, // 10 kHz vbw
41.    0.001,  // 1 ms sweep acquisition
42.    BB_NATIVE_RBW, // Use native rbw
43.    BB_NO_SPUR_REJECT // No software spur rejection
44. );
45.
46. bbConfigureWindow(
47.    devID,
48.    BB_BLACKMAN // Blackman windowing function
49. );
50.
51. bbConfigureProcUnits(
52.    devID,
53.    BB_LOG // Spectrum analysis performed in logarithmic scale
54. );
55.
56. status = bbInitiate(
57.    devID,
58.    BB_SWEEPING // Sweep mode
59.    0  // Use zero if not in zero-span mode
60. );
61.
62. if( status != bbNoError )
63.    myErrorRoutine( status );
64.
65. // Device initiated, get sweep information
66. bbQueryTraceInfo(
67.    devID,
68.    &traceSize, // Get trace size
69.    &binSize,   // Get freq per returned sample
70.    &startFreq  // Get accurate start frequency
71. );
```

```
72.
73. min = new double[traceSize];
74. max = new double[traceSize];
75.
76. // Continually fetch sweep information
77. while( yourProgramIsRunning ) {
78.
79.    bbFetchTrace(
80.       devID,
81.       traceSize,
82.       min,
83.       max
84.    );
85.
86.    displayTrace( min, max ); // Your custom routine
87.
88. } /* while(programRunning) */
89.
90.
91. // Your custom error handling routine
92. void myErrorRoutine( bbStatus code ) {
93.
94.    cerr << bbGetErrorString( code );
95.    handleError( code );
96.
97. }
```

Raw Data Pipe Example

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

```
1.  /* Retrieving raw ADC values */
2.  /* Also shows how to retrieve external trigger occurrences
3.     from the returned trigger array */
4.
5.  int devID;
6.  float buf = new float[BB_RAW_PACKET_SIZE];
7.  int triggers[17];                // From bbFetchRaw
8.  std::vector<int> triggerList;    // Just triggers
9.
10. bbOpenDevice( &devID );
11.
12. bbConfigureCenterSpan(
13.    devID,
14.    2500.0e9,  // 2.5 GHz center
15.    20.0e6     // 20 MHz span is default for raw-pipe mode
16. );
17.
18. bbConfigureLevel(
19.    devID,
20.    0.0,   // not applicable if gain is selected
21.    10.0   // 10 db attenuation
22. );
```

```
23.
24. bbConfigureGain(
25.    devID,
26.    1       // low gain
27. );
28.
29. bbConfigureIO(
30.    devID,
31.    0,                                // default
32.    BB_PORT2_IN_TRIGGER_RISING_EDGE   // Catch rising edges
33. );
34.
35. bbInitiate(
36.    devID,
37.    BB_RAW_PIPE,
38.    0  // n/a
39. );
40.
41. while( streamingData ) {
42.
43.    bbFetchRaw(
44.       devID,
45.       buf,           // spectrum
46.       triggers       // triggers
47.    );
48.
49.    // Extract just the trigger positions into a vector
50.    int i = 0;
51.    while(triggers[i]) {
52.       triggerList.push_back(triggers[i]);
53.       i++;
54.    }
55.
56.    doSomethingWithData( buf, triggerList ); // Save/process/display
57.
58. } // while(streaming)
```

Zero-Span Triggering

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

```
1.  /* Configure Zero-Span and an external trigger */
2.
3.  int devID;
4.
5.  bbOpenDevice( &devID );
6.
7.  //
8.  // Other configurations here
9.  //
10. // bbConfigureAcquisition
11. // bbConfigureCenterSpan
12. // bbConfigureLevel
13. // bbConfigureSweepCoupling
14. //
15.
16. // Configure our trigger, tell the device to use an external trigger
17. bbConfigureTrigger(
18.    devID,
19.    BB_EXTERNAL_TRIGGER,
```

```
20.   0,              // n/a for external trigger
21.   0,              // n/a for external trigger
22.   0.032           // wait up to 32 ms for trigger
23. );
24.
25. // The device now expects an external trigger, we configure port 2 for a trigger
26. bbConfigureIO(
27.   devID,
28.   0,                                // Not using port1, 0 is default
29.   BB_PORT2_IN_TRIGGER_RISING_EDGE   // trigger on rising edge
30. );
31.
32. bbInitiate(
33.   devID,
34.   BB_ZERO_SPAN, // Zero-Span mode
35.   BB_DEMOD_FM   // FM demodulation
36. );
37.
38. /*
39.  The device is now ready for an external trigger.
40.  From here, we would get our trace information, and
41.  begin getting sweeps.
42. */
```

Using a GPS Receiver to Time-Stamp Data

With minimal effort it is possible to determine the absolute time (up to 50ns) of the ADC samples. This functionality is only available when using the device in the "raw pipe" mode.

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

Order of Operations:
1) Ensure correct operation of your GPS reciever.
2) Connect the 1PPS reciever output to port 2 of the BB60A.
3) Connect the RS232 reciever output to your PC.
4) Determine the COM port number and baud rate of the data transfer over RS232.
5) Open the device via *bbOpenDevice()*
6) Ensure the RS232 connection is not open.
7) Use *bbSyncCPUtoGPS()* to synchronize the API timing with the current GPS time. This function will release the connection when finished.
8) Configure the device for raw pipe mode.
9) Before initiating the device, use *bbConfigureIO* and configure port 2 for an incoming rising edge trigger via BB_PORT2_IN_TRIGGER_RISING_EDGE.
10) Call *bbInitiate(id, BB_RAW_PIPE, BB_TIME_STAMP)*. The BB_TIME_STAMP argument will tell the API to look for the 1PPS input trigger for timing.
11) If initiated successfully you can now fetch data via *bbFetchRaw().* Calling the function *bbQueryTimestamp()* will return the time of the first sample in the array of data collected from the last *bbFetchRaw().*

12) From the time retrieved, you can estimate the time of any sample knowing the difference in time between two samples is typically 12.5ns or 1/80000000.

Code Example

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

```
1.  /* Configure and prepare the device for time stamping */
2.
3.  int id;
4.  float *data = new float[299008];
5.
6.  // Open Device as usual
7.  bbOpenDevice( &id );
8.
9.  // Configuration
10. bbConfigureCenterSpan( id, 900.0e6, 20.0e6 );
11. bbConfigureLevel( id, 0, 10 );
12. bbConfigureGain( id, BB_HIGH_GAIN );
13.
14. // The device MUST be ready to accept input triggers on port 2
15. // The 1 PPS trigger will be connected to port 2
16. bbConfigureIO( id, 0, BB_PORT2_IN_TRIGGER_RISING_EDGE );
17.
18. // At this point, the GPS receiver must be operational
19. // The RS232 connection cannot be open, and the com port and baud rate
20. //    must be known
21. // Ensure the receiver is "locked"
22. bbSyncCPUtoGPS( 3, 38400 );
23.
24. // If syncCPUtoGPS returned successfully the device can now be initiated
25. //    and the RS232 connection should now be closed.
26. // Note: BB_TIME_STAMP is required so the device treats input triggers as the
27. //    GPS 1PPS
28. bbInitiate( id, BB_RAW_PIPE, BB_TIME_STAMP );
29.
30. // We can now retrieve data
31. while( programRunning ) {
32.
33.   int seconds, nanoseconds;
34.   char *timeString;
35.
36.   // If we wanted we could collect the 1PPS triggers here
37.   bbFetchRaw(
38.     id,
39.     data,   // Collect one raw packet
40.     NULL    // Not interested in the triggers,
41.   );
42.
43.   // Return the seconds and nanoseconds of the first sample in the last packet
44.   //    retrieved.
45.   bbQueryTimestamp( id, &seconds, &nanoseconds );
46.
47.   // Function in <ctime> which returns a human readable string of the date/time
48.   timeString = ctime( (time_t*)(&seconds) );
49.
50.   doSomething();
51. }
```

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

```
1.  /*
2.     Retrieve the time of any sample in a packet
3.     To do this we need to know the starting time of the packet and
4.        the sample we are interested in
5.  */
6.  void GetSampleTime(
7.     unsigned int startSeconds,     // In:  Seconds returned from QueryTimestamp
8.     unsigned int startNanos,       // In:  Nanoseconds returned from QueryTimestamp
9.     unsigned int sample,           // In:  Sample we are interested in, zero based
10.    unsigned int *sampleSeconds,   // Out: Seconds for interested sample
11.    unsigned int *sampleNanos )    // Out: Nanoseconds for interested sample
12. {
13.    // Amount of time between any two samples
14.    double delTime = 1.0 / 80000000;
15.
16.    // Assuming zero based sample, get output nanos
17.    unsigned int outs = startSeconds;
18.    unsigned int outns = startNanos + delTime * sample;
19.
20.    // If nanos are greater than 1 billion, then we wrap
21.    if( outns > 1000000000 ) {
22.       outs++;
23.       outns -= 1000000000;
24.    }
25.
26.    *sampleSeconds = outs;
27.    *sampleNanos = outns;
28. }
```

Bandwidth Tables

Internally the API uses bandwidth tables for its resolution and video bandwidths. When bandwidths are specified through the bbConfigureSweepCoupling routine, the provided values are clamped to the closest value found in the table. Below are the possible native and non-native bandwidth values.

Native Bandwidths (Hz)	Non-Native Bandwidths (Hz)	FFT size
10.10e6		16
5.050e6		32
2.525e6	1.0e7	64
1.262e6		128
631.2e3		256
315.6e3	3.0e6	512
157.1e3	1.0e6	1024
78.90e3		2048
39.45e3	3.0e5	4096
19.72e3	1.0e5	8192
9.863e3	3.0e4	16384
4.931e3		32768
2.465e3	1.0e4	65536
1.232e3	3.0e3	131072

616.45		262144
308.22		524288
154.11	1.0e3	1048576
154.11	3.0e2	1048576
77.05		2097152
38.52	1.0e2	4194304
19.26		8388608
9.63	3.0e1	16777549
4.81	1.0e1	33554432
2.40		67108864
1.204	3.0	134217728
0.602		268435456
0.301	1.0	536870912