VSG60 Application Programming Interface (API)

Programmers Reference Manual

VSG60 Application Programming Interface (API)
Programmers Reference Manual

1502 SE Commerce Ave, Suite 101
Battle Ground, WA
Phone 360-313-7997

July 15, 2020

Table of Contents

Overview

This manual is a reference for the Signal Hound SM200 spectrum analyzer programming interface (API). The API provides a low-level set of C routines for interfacing the SM200. The API is C ABI compatible making is possible to be interfaced from most programming languages. See the code examples folder to for examples of using the API in C++ and other various environments.

Contact Information

For all programming and technical questions, please email aj@signalhound.com.

For sales, email sales@signalhound.com.

Build/Version Notes

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 makes 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 DLL.

Version 1.0.0 – Official release

PC Requirements and Setup

Windows Development Requirements

- Windows 7/8/10, (7/10 recommended)
- Linux 64-bit (Ubuntu 18.04 recommended)
- (C/C++ only) Windows C/C++ development tools and environment. Preferably Visual Studio 2008 or later. If Visual Studio 2012 is not used, then the VS2012 redistributables will need to be installed.
- Library files **vsg_api.h**, **vsg_api.lib**, and **vsg_api.dll**.

PC and Other Requirements

- VSG60
- USB 3.0 connectivity provided through 4th generator or later Intel CPUs. 4th generation Intel CPU systems might require updating USB 3.0 drivers to operate properly.
- (Recommended) Quad core Intel i5 or i7 processor, 4th generation or later.
- (Minimum) Dual core Intel i5 or i7 processor, 4rd generation or later.

Code Examples

All code examples are provided in the Signal Hound SDK.

https://signalhound.com/software/signal-hound-software-development-kit-sdk/

Theory of Operation

There are two primary ways to generate waveforms with the VSG60 API.

1) (basic) Provide a complete waveform which the API will output once or repeat until stopped. This is the simplest method for generation. These methods are ideal for fixed frequency output waveforms up to many seconds in length.
2) (complex) Use the streaming functions which allow for long waveforms or complex sequences which might involve frequency hopping and level changes.

Opening a Device

Before any generation can occur, the device must be opened and initialized. Opening and initializing a device through the API is performed through the `vsgOpenDevice` or `vsgOpenDeviceBySerial` functions. These functions will perform the full initialization of the device and if successful, will return an integer handle which can be used to reference the device for the remainder of your program. See the list of all SM200 devices connected to the PC via the `vsgGetDeviceList` function.

Basic Signal Generation

Basic signal generation involves configuring the generator and then using one of the following functions

- vsgOutputWaveform
- vsgRepeatWaveform
- vsgOutputCW (convenience function)

Waveforms are provided as interleaved I/Q complex pairs. The API can output the waveform once using the `vsgOutputWaveform` function, or continually generate the waveform using the `vsgRepeatWaveform` function.

The `vsgOutputWaveform` is a blocking function which returns once fully output. When a waveform is on repeat, any changes in configuration will cause the waveform to be paused, and then restarted once reconfiguration has completed.

Submitting a trigger or I/Q data through the `vsgSubmitIQ` function will stop any waveforms on repeat.

Calling `vsgAbort` will end any active waveforms on repeat.

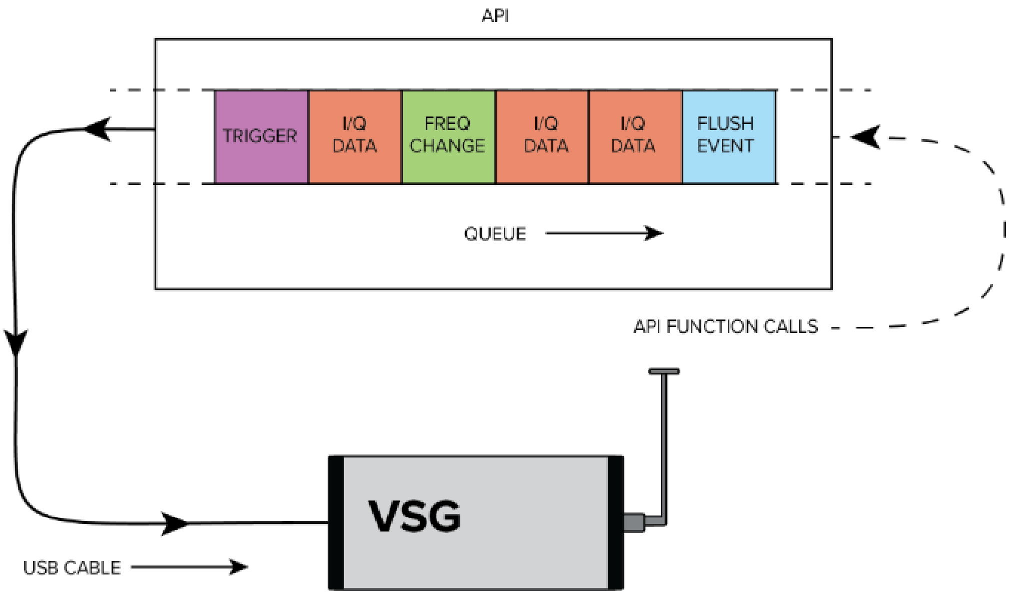
Complex Signal Generation (Streaming)

For long waveform generation or for transmitting a complex sequence of events such as frequency hopping or triggers, a collection of functions are available in the API which allow a user to buffer a sequence of configuration and transmit events. The following functions can be buffered/queued

- vsgSetFrequency
- vsgSetLevel
- vsgSubmitIQ
- vsgSubmitTrigger
- vsgFlushAndWait

Roughly 1/5th of a second of I/Q data can be buffered. If the buffer is full and `vsgSubmitIQ` data is called, the function blocks until space is available in the buffer.

Calling `vsgAbort` will cause all I/Q data in the buffer to be dumped and any pending frequency/level changes to be completed in the order received.



Closing the Device

When finished, you can close the device and free all resources related to the device with the `vsgCloseDevice` 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.

Recalibration

Recalibration is performed by calling the `vsgRecal` function which retrieves the current device temperature and recalibrates the device for the current device settings. This will interrupt any signal generation currently in progress.

Large temperature changes affect signal generation in the form of reduce amplitude accuracy and reduced spurious performance, and it is recommended to reconfigure the device after large environmental changes and during device warmup.

Recalibration can occur automatically during periods of activity that include frequency changes, but when generating the same signal for long periods of time, or after a long period of inactivity, a recalibration is recommended.

I/Q Data and Output Power

Waveforms are provided to the API as I/Q data. I/Q data should be provided as contiguous interleaved real, imaginary pairs.

Example: [re_1, im_1, re_2, im_2, …, re_N, im_N] would be an array of *N* I/Q samples, which would equal *2*N* contiguous floating-point values.

Each {re, im} pair represents a single sample. Each real and imaginary sample should be a 32-bit floating point value. A magnitude of 1.0 will equal the output power set by the user.

$$Magnitude\ of\ I/Q\ sample\ =\ \sqrt{real^{\,2} + imag^2}$$

To measure the output power of a sample in dBm, use the formula

$$Power\ of\ IQ\ sample = Output\ Power\ (vsgSetLevel) + 20 * log10(magnitude\ of\ IQ\ sample)$$

Internally I/Q samples are scaled to achieve the user selected output power. The default scaling is 0.5 and increases/decreases around this value to digitally scale where the internal amplifier and attenuator cannot. The internal scale can be queried through the API. Because we use a base scale of 0.5, this means magnitudes greater than 1.0 can be tolerated. Clipping occurs when either I or Q value exceeds 1.0 post scaling. The scaling is performed as such.

$$Scaled\ IQ = \{real * scaleFactor, imag * scaleFactor\}$$

If you have a waveform with amplitudes much greater than 1.0, it is recommended to query the scale to verify clipping won't occur or scale the entire waveform and adjust the output power to compensate.

Thread Safety

The VSG60 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. Ideally, a device should be interfaced from a single thread.

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 of the device directly 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 is to use a named mutex on a Windows system.

If you wish to interface multiple devices on Linux, see the Appendix: Linux Notes.

Status Codes and Error Handling

All functions return a `VsgStatus` error code. `VsgStatus` is an enumerated type representing the success of a given function call. The integer values associated with each status provides information about whether a function call succeeded or failed.

An integer value of zero indicates no error or warnings. Negative integer status values indicate errors and positive values represent warnings.

A descriptive string of each status type can be retrieved using the `vsgGetErrorString` function.

Functions

All functions other that initialization functions take a device handle as the first parameters. This integer is obtained after opening the device through either the vsgOpenDevice or vsgOpenDeviceBySerial function. This handle uniquely identifies the receiver for the duration of the application execution, or until vsgCloseDevice is called.

Each function returns an error code which can provide warnings or errors related to the execution of the function. There are many cases where you will need to monitor these codes to determine the success or failure of an operation. See a list of common error codes and their descriptions in the Appendix.

Common Error Codes

This sections documents some of the more common error codes and their meaning. For a full list of status codes, see the API header file and the API function list in this document.

Negative error codes represent errors and are suffixed with 'Err'. When an error code is returned, the operation requested did not complete. Positive error codes are warnings and indicate that the function/operation completed successfully, but the user might need to take some action.

`vsgNoError`	Returned when a function returns successfully.
`vsgInvalidDeviceErr`	Returned when the device handle provided does not match an open device.
`vsgSettingClamped`	Returned when one or more parameters was clamped to a valid range.
`vsgInvalidParameterErr`	Returned when one or more parameters is not valid. For instance, if an enum parameter does not match the set of possible values, this error code is returned.
`vsgNullPtrErr`	Returned when one ore more pointer parameters are NULL.

vsgGetAPIVersion

`const char* vsgGetAPIVersion();`

Return Values

`const char*`	The returned string is of the form major.minor.revision Ascii periods ('.') separate positive integers. Major/minor/revision are not guaranteed to be a single decimal digit. The string is null terminated. The string should not be modified or freed by the user. An example string is below… ['3' | '.' | '0' | '.' | '1' | '1' | '\0'] = "3.0.11"

vsgGetDeviceList

`VsgStatus vsgGetDeviceList(int *serials, int *deviceCount);`

Parameters

`serials`	Pointer to an array of integers.
`deviceCount`	Pointer to an int. The initial value should be the size of the serials array. If the function returns successfully, `deviceCount` will be set to the number devices found on the system and returned in the serials array. `deviceCount` will not exceed the initial size passed to the function.

Description

This function is used to retrieve the serial number of all unopened VSG60 devices connected to the PC. The serial numbers returned can then be used to open specific devices with the `vsgOpenDeviceBySerial` function.

When the function returns successfully, the `serials` array will contain `deviceCount` number of unique VSG60 serial numbers.

vsgOpenDevice

`VsgStatus vsgOpenDevice(int *handle);`

Parameters

`handle`	Pointer to integer to be used as a handle for the device.

Description

Claim the first unopened VSG60 detected on the system. If the device is opened successfully, a handle to the function will be returned through the device pointer. This handle can then be used to refer to this device for all future API calls.

This function has the same effect as calling `vsgGetDeviceList` and using the first device found to call `vsgOpenDeviceBySerial`.

Return Values

`vsgDeviceNotFoundErr`	Unable to find/open a VSG60 receiver.

vsgOpenDeviceBySerial

`VsgStatus vsgOpenDeviceBySerial(int *handle, int serialNumber);`

Parameters

`handle`	Pointer to integer to be used as a handle for the device.
`serialNumber`	Serial number of the device you wish to open.

Description

This function operates like `vsgOpenDevice` except it allows you to specify the device you wish to open. This function is often used in conjunction with `vsgGetDeviceList` when managing several VSG60 devices on one PC.

Return Values

See return values for `vsgOpenDevice`.

vsgCloseDevice

`VsgStatus vsgCloseDevice(int handle);`

Parameters

Description

This function should be called when you are finished with the VSG60. It will release all resources for the device and the device will become available again for use in the current process. The device handle specified will no longer point to a valid device and the device must be re-opened again to be used. This function should be called before the process exits.

Return Values

vsgPreset

```
VsgStatus vsgPreset(int handle);
```

Parameters

Description

Performs a full device preset. When this function returns, the hardware will have performed a full reset, the device handle will no longer be valid, the `vsgCloseDevice` function will have been called for the device handle, and the device will need to be re-opened again.

This function can be used to recover from an undesirable device state.

This function takes about 3 seconds to complete and return.

Return Values

vsgRecal

```
VsgStatus vsgRecal(int handle);
```

Parameters

Description

When operating the VSG60 for long periods of time with a fixed configuration, environmental changes leading to changes in the internal operating temperature of the VSG can cause signal drift leading to loss of amplitude accuracy and image rejection performance. This function aborts any current operation, and updates internal temperature corrections for the current configuration.

Return Values

vsgAbort

```
VsgStatus vsgAbort(int handle);
```

Parameters

Description

This function returns the device to an idle state.

If currently playing a waveform through the `vsgRepeatWaveform` function, the generation is stopped.

If the device is streaming in the complex generation mode, all I/Q data pending is discarded and all frequency/level changes are finished before returning.

When this function returns, the device will be in an idle state.

Return Values

vsgGetSerialNumber

```
VsgStatus vsgGetSerialNumber(int handle, int *serialNumber);
```

Parameters

`serialNumber`	Pointer to integer. If this function returns successfully, the integer pointed to will contain the specified devices serial number.

Description

This function returns the serial number of an open VSG60 device.

Return Values

vsgGetFirmwareVersion

`VsgStatus vsgGetFirmwareVersion(int handle, int *version)`

Parameters

`version`	Pointer to 32-bit int.

Description

Get the firmware version of an open device.

Return Values

vsgGetCalDate

`VsgStatus vsgGetCalDate(int handle, uint32_t *lastCalDate);`

Parameters

`lastCalDate`	Pointer to unsigned integer.

Description

This function returns the calibration date as the seconds since epoch.

Return Values

vsgReadTemperature

`VsgStatus vsgReadTemperature(int handle, float *temp);`

Parameters

`temp`	Pointer to float, to contain current device internal temperature in Celsius.

Description

If the device is not idle, the last read temperature is returned, otherwise the device temperature is read before returning.

Return Values

vsgSetRFOutputState

```
VsgStatus vsgSetRFOutputState(int handle, VsgBool enabled);
```

```
VsgStatus vsgGetRFOutputState(int handle, VsgBool *enabled);
```

Parameters

`enabled`	Set to `vsgTrue` to enable the RF output. Set to `vsgFalse` to disable the RF output.

Description

Use this function to disable the RF output of the VSG60. Even when the VSG is not transmitting, it might still be emitting spurious energy related to clock frequencies and the DC offset. Disabling the RF output will eliminate most spurious signals.

The RF output is enabled by default.

When the RF output is disabled the only way to enable it again is to call this function. Until then, all actions will continue to be performed but with the RF output disabled.

The vsgAbort function is called when this function is called. This means any signal actively being generated will be stopped when this function is called.

Return Values

vsgSetTimebase

```
VsgStatus vsgSetTimebase(int handle, VsgTimebaseState state);
```

```
VsgStatus vsgGetTimebase(int handle, VsgTimebaseState *state);
```

Parameters

`state`	Timebase state enum.

Description

Specify whether the VSG60 should use its internal 10MHz reference or one provided on the 10MHz reference BNC.

If external reference is selected and no reference is provided, the frequency error may be off by several PPM.

If the state provided matches the current state, the function returns immediately.

Any active waveforms are paused, and the stream is flushed before this operation takes place.

Return Values

vsgSetTimebaseOffset

```
VsgStatus vsgSetTimebaseOffset(int handle, double ppm);
```

```
VsgStatus vsgGetTimebaseOffset(int handle, double *ppm);
```

Parameters

`ppm`	Floating point value between [-2, +2]

Description

Adjust the VSG timebase by up to 2ppm. This adjustment will only last until the device is closed via vsgCloseDevice or the program is terminated.

If the value provided matches the currently set value, this function returns immediately.

Any active waveforms are paused, and the stream is flushed before this operation takes place.

Return Values

vsgSetFrequency

```
VsgStatus vsgSetFrequency(int handle, double frequency);

VsgStatus vsgGetFrequency(int handle, double *frequency);
```

Parameters

`frequency`	Set the output frequency in Hz.

Description

Sets the output frequency of the VSG.

This function can be used for complex streaming signal generation. In complex streaming mode, this operation takes 200us to complete.

This operation will occur even if the provided frequency matches the current frequency.

This operation may configure a recalibration (at no cost).

Return Values

vsgSetSampleRate

```
VsgStatus vsgSetSampleRate(int handle, double sampleRate);

VsgStatus vsgGetSampleRate(int handle, double *sampleRate);
```

Parameters

`sampleRate`	Set the VSG sample rate in Hz

Description

Sets the I/Q sample rate of the VSG. The streaming queue is flushed via vsgFlushAndWait before the sample rate is updated. If the supplied sample rate is the same sample rate, this function returns immediately.

A full sample rate change takes approximately 200-250ms.

Return Values

`vsgInvalidParameterErr`	If the sample rate provided is outside the acceptable range an error is returned instead of clamping.

vsgSetLevel

```
VsgStatus vsgSetLevel(int handle, double level);
```

```
VsgStatus vsgGetLevel(int handle, double *level);
```

Parameters

`level`	Desired output level in dBm.

Description

Set the output level of the VSG. Hardware attenuation, amplification, and digital scaling are used to achieve the requested output level. What values are used depend on the temperature calibration coefficients for individual device and the frequency of the output.

This function can be used for complex streaming signal generation. In complex streaming mode, this operation takes 10us to complete.

This operation will occur even if the provided level matches the current level.

This operation may configure a recalibration (at no cost).

Return Values

vsgSetAtten

```
VsgStatus vsgSetAtten(int handle, int atten);
```

Parameters

atten	Attenuator value between [-50, 20] in 2dB steps. (must be an even number).

Description

This function allows the customer to guarantee the configuration of the internal attenuator and amplifier directly by specifying a fixed system attenuation. Values that are positive utilize the amplifier to achieve the desired setting. A digital I/Q scale of 0.5 is used.

Return Values

vsgGetIQScale

```
VsgStatus vsgGetIQScale(int handle, double *iqScale);
```

Parameters

`iqScale`	Floating point value returned between [0.0, 1.0]

Description

Returns the currently used digital scale applied to the I/Q data before transmitting. The digital scaling is used in conjunction with the hardware amplifier and attenuator to achieve the desired output level.

This function does not interrupt any active waveforms or streaming generation.

Return Values

vsgSetIQOffset

`VsgStatus vsgSetIQOffset(int handle, int16_t iOffset, int16_t qOffset);`

`VsgStatus vsgGetIQOffset(int handle, int16_t *iOffset, int16_t *qOffset);`

Parameters

`iOffset`	User I offset between [-1024, +1024]
`qOffset`	User Q offset between [-1024, +1024]

Description

Specify an additional I/Q offset applied to the I/Q data before transmit. Used to fine improve carrier feedthrough. The offset lasts until the device is closed or the program is terminated.

`vsgFlushAndWait` is called at the beginning of this function.

If the supplied value matches the old value, this function returns immediately.

Return Values

vsgSetDigitalTuning

`VsgStatus vsgSetDigitalTuning(int handle, VsgBool enabled);`

`VsgStatus vsgGetDigitalTuning(int handle, VsgBool *enabled);`

Parameters

`enabled`	When set to `vsgTrue`, digital tuning is enabled.

Description

If the value is provided is, this function returns immediately with no effect.

`vsgFlushAndWait` is called before the operation occurs.

See the VSG60 product manual for a description of digital tuning.

Return Values

vsgSetTriggerLength

`VsgStatus vsgSetTriggerLength(int handle, double seconds);`

`VsgStatus vsgGetTriggerLength(int handle, double *seconds);`

Parameters

`seconds`	Time in seconds.

Description

Length of time the output trigger logic port remains high when a trigger is output in seconds. Default is 10us. (10.0e-6)

The range of acceptable values is [100ns, 1s]

This function does not interrupt any active waveforms or streaming generation.

Return Values

vsgSubmitIQ

`VsgStatus vsgSubmitIQ(int handle, float *iq, int len);`

Parameters

`iq`	Array of interleaved I/Q values.
`len`	Number of I/Q pairs in `iq` array.

Description

Submit an array of I/Q samples to be generated with the current configuration. If an ARB waveform is currently being transmitted via the `vsgRepeatWaveform` function, it is aborted, and the device starts operating in the streaming configuration.

This function should only be used for complex/streaming signal generation. For generating simple waveforms, use the `vsgOutputWaveform` and `vsgRepeatWaveform` functions.

This function will block until there is room in the processing and command queue.

See Complex Signal Generation for more information.

Return Values

vsgSubmitTrigger

`VsgStatus vsgSubmitTrigger(int handle);`

Parameters

Description

Submit a streaming trigger event. If an ARB waveform is currently being transmitted, it is aborted, and the device starts operating in the streaming configuration. If a trigger is already active when the new trigger is output, it is first toggled low before re-toggling high, resetting the trigger high period in the process.

Return Values

vsgFlushAndWait

`VsgStatus vsgFlushAndWait(int handle);`

Parameters

Description

Pushes all pending operations in a complex stream out to the device and blocks until they have been completed.

This function should be called after a sequence of streaming events to ensure continuous operation of the streaming events (no gaps).

When this function returns, the device will be idle.

Return Values

vsgOutputWaveform

`VsgStatus vsgOutputWaveform(int handle, float *iq, int len);`

Parameters

`iq`	Array of interleaved I/Q values.
`len`	Number of I/Q pairs in `iq` array.

Description

Output the I/Q waveform once and return. This function has the same effect as calling `vsgSubmitIQ` followed by `vsgFlushAndWait`. This function returns once the waveform has been transmitted. The device will be in an idle state when returned. If an ARB waveform is active, it is aborted before transmission.

Return Values

vsgRepeatWaveform

`VsgStatus vsgRepeatWaveform(int handle, float *iq, int len);`

Parameters

`iq`	Array of interleaved I/Q values.
`len`	Number of I/Q pairs in `iq` array.

Description

This function instructs the API to continually generate the provided waveform. A full copy of the waveform is made before generation occurs. This function blocks until signal generation has begun.

The repeated waveform is only stopped after calling the `vsgAbort` function or by submitting I/Q data or a trigger. Setting any other configuration value, such as frequency, level, etc. will only pause the waveform until the configuration completes and then generation starts again.

Return Values

vsgOutputCW

`VsgStatus vsgOutputCW(int handle);`

Parameters

Description

Convenience function which outputs a CW signal with the current frequency, level, and sample rate. This function has the same effect as calling `vsgRepeatWaveform` with a single I/Q value of {1,0}.

Return Values

vsgEnablePowerSavingCpuMode

`void vsgEnablePowerSavingCpuMode(VsgBool enabled);`

Description

See Power Saving CPU Mode.

vsgGetErrorString

`const char* vsgGetErrorString(VsgStatus status);`

Parameters

`status`	A valid `VsgStatus` enumeration.

Description

Retrieve a descriptive string of a `VsgStatus` enumeration. Useful for debugging and diagnostic purposes.

Return Values

`const char*`	A pointer to a null terminated string. The memory should not be freed/deallocated/modified.

Appendix

Linux Notes

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.

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.

Multiple Devices

There are limitations that apply when attempting to use multiple devices on Linux. The maximum amount of memory that can be allocated for USB transfers on Linux is 16MB. A single VSG60A can stay within this limitation, but two devices will exceed this limitation and can cause the API to crash when you do. The USB allocation limit 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.

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.

Other Programming Languages

The VSG60 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 VSG60 API in these other languages are included in the code examples folder.

The VSG60 API consists of several enumerated(enum) types, which are often used as parameters. These values can be treated as 32-bit integers when callings the API functions from other programming languages. You will need to match the enumerated values defined in the API header file.

Power Saving CPU Mode

Newer CPU models implement efficient power saving techniques that can interfere with and reduce USB bandwidth. If you are using one of these CPU models, you can experience issues with the VSG60 which might appear as data loss when inspecting the output of the VSG60. We are offering 2 potential solutions to this problem.

1) Enable the power saving CPU mode through the API. This has the effect of adding an artificial load to the API to keep the CPU from entering any low power CPU states that might affect the USB throughput. You will see an increase in CPU usage through this method.

2) Disable "C-States" in the BIOS of the PC. This prevents the OS from being able to put the CPU in these low power states which affect USB performance. This will increase power consumption of the PC which will affect battery life but will see lower CPU usage (since power saving CPU mode can be disabled).

The default state of this mode is disabled in the API.

PCs most affect are laptops and ultraportable devices running Windows 10.