RSA306
Spectrum Analyzer
Application Programming Interface (API)

Programming Reference

077-1031-01

Tektronix

RSA306
Spectrum Analyzer
Application Programming Interface (API)

Programming Reference

Revision A

www.tektronix.com

077-1031-01

Tektronix

Copyright © Tektronix. All rights reserved. Licensed software products are owned by Tektronix or its subsidiaries or suppliers, and are protected by national copyright laws and international treaty provisions.

Tektronix products are covered by U.S. and foreign patents, issued and pending. Information in this publication supersedes that in all previously published material. Specifications and price change privileges reserved.

TEKTRONIX and TEK are registered trademarks of Tektronix, Inc.

Contacting Tektronix

Tektronix, Inc.
14150 SW Karl Braun Drive
P.O. Box 500
Beaverton, OR 97077
USA

For product information, sales, service, and technical support:

- In North America, call 1-800-833-9200.
- Worldwide, visit www.tektronix.com to find contacts in your area.

Table of Contents

Preface ii
API function groups 1
- ADC streaming 2
- Alignment 4
- Audio 5
- Connection 8
- Device operation 9
- Device status 15
- DPX 19
- IQ data 28
- IQ streaming 32
- Playback (R3F file format) 47
- Trigger 50
- Self test 54
- Spectrum 55
- System time reference 61

Example Python program 63
- File attachment 64

RSA306 Streaming Sample Data File Format 65
Index

Preface

This document contains the API function calls to interface with the RSA306 Signal Analyzer through Microsoft Windows.

The API driver is required to use the function calls. This driver is automatically installed with the installation of the SignalVu-PC software. If you wish to install the API driver without SignalVu-PC software, it is available on the Flash drive provided with the RSA306 Spectrum Analyzer. Open the RSA306 flash drive, and navigate to the API installer. The supplied driver is for the Microsoft Windows operating system.

Some examples of programing languages supported by this driver include: C, C++, and Python.

API function groups

This section contains the available function calls. The functions are grouped into the following categories:

- ADC streaming (See page 2.)
- Alignment (See page 4.)
- Audio (See page 5.)
- Connection (See page 8.)
- Device operation (See page 9.)
- Device status (See page 15.)
- DPX (See page 19.)
- IQ data (See page 28.)
- IQ streaming (See page 32.)
- Trigger (See page 50.)
- Self test (See page 54.)
- Spectrum (See page 55.)
- System time reference (See page 61.)

ADC streaming

NOTE. *Before calling the API function SetStreamADCToDiskEnabled(true), you must have made at least one call to Run() to initialize the channel correction data structures or the frame header information in at least one of your streamed files will be incomplete.*

After calling SetStreamADCToDiskEnabled(true), you must not make any changes to hardware settings until you call SetStreamADCToDiskEnabled(false) or until enough time has elapsed such that all automatically created streamed files are completely written to disk.

Table 1: ADC streaming functions

SetStreamADCToDiskEnabled	Starts and stops the ADC streaming operation.
Declaration:	ReturnStatus SetStreamADCToDiskEnabled(bool enabled);
Parameters:	
enabled:	Boolean value which specifies whether to start or stop streaming to disk.
Return Values:	
noError:	The operation has completed successfully.
errorStreamADC-ToDiskAlreadyStreaming:	ADC streaming is already in operation.
SetStreamADCToDiskPath	Sets the path for file saves.
Declaration:	ReturnStatus SetStreamADCToDiskPath(const char *path);
Parameters:	
path:	Character string defining the path the ADC data is saved to.
Return Values:	
noError:	The operation has completed successfully.
errorStreamADC-ToDiskAlreadyStreaming:	ADC streaming is already in operation.
SetStreamADCToDiskFilenameBase	Sets the base file name for file saves.
Declaration:	ReturnStatus SetStreamADCToDiskFilenameBase(const char *path);
Parameters:	
path:	Character string defining the base name of the ADC data files.
Return Values:	
noError:	The operation has completed successfully.
errorStreamADC-ToDiskAlreadyStreaming:	ADC streaming is already in operation.
Additional Detail	The base file name is combined with the path and a timestamp to generate a unique file name for this date and session.

Table 1: ADC streaming functions (cont.)

SetStreamADCToDiskMaxTime	Sets the maximum recording time for any single data file.
Declaration:	ReturnStatus SetStreamADCToDiskMaxTime(long milliseconds);
Parameters:	
milliseconds:	Sets the maximum recording time for ADC files.
Return Values:	
noError:	The operation has completed successfully.
errorStreamADC-ToDiskAlreadyStreaming:	ADC streaming is already in operation.
SetStreamADCToDiskMode	Sets the streaming mode.
Declaration:	ReturnStatus SetStreamADCToDiskMode(StreamingMode mode);
Parameters:	
filename:	A StreamingMode type that specifies whether the device is in StreamingModeRaw or StreamingModeFramed.
Return Values:	
noError:	The operation has completed successfully.
errorStreamADC-ToDiskAlreadyStreaming:	ADC streaming is already in operation.
Additional Detail	In StreamingModeRaw, the data file has only ADC samples. The frame footer is removed and the data header, describing the contents, is placed in an auxiliary file. In StreamingModeFramed, the header is the first 16k block in the data file and each frame is complete, including frame footers. Refer to RSA306 Streaming Sample Data File Format. (See page 65.)
SetStreamADCToDiskMaxFileCount	Sets the maximum number of files to open for streamed data.
Declaration:	ReturnStatus SetStreamADCToDiskMaxFileCount(int maximum);
Parameters:	
maximum:	Maximum number of files to save.
Return Values:	
noError:	The operation has completed successfully.
errorStreamADC-ToDiskAlreadyStreaming:	ADC streaming is already in operation.
GetStreamADCToDiskActive	Allows the current status of the ADC data streaming operation to be queried.
Declaration:	ReturnStatus GetStreamADCToDiskActive(bool *enabled);
Parameters:	
enabled:	Reports the current status of the ADC data streaming operation.
Return Values:	
noError:	The operation has completed successfully.

Alignment

Table 2: Alignment functions

GetDeviceTemperature	Returns the current temperature of the device.
Declaration:	ReturnStatus double GetDeviceTemperature();
Return Values:	
double:	The current temperature of the device in degrees Celcius.
RunAlignment	Calibrates the device for the current temperature.
Declaration:	ReturnStatus RunAlignment();
Return Values:	
noError:	The alignment has succeeded.
errorDataNotReady:	The device failed to wait for IQ data to be ready.
IsAlignmentNeeded	Determines if an alignment is needed or not.
Declaration:	ReturnStatus IsAlignmentNeeded(bool* needed);
Parameters:	
needed:	Pointer to a bool. Its value is true when an alignment is needed or false when an alignment is not needed.
Return Values:	
noError:	The function has completed successfully.
errorNotConnected:	The device is not connected.
Additional Detail	It is based on the difference between the current temperature and the temperature from the last alignment.

Audio

Table 3: Audio functions

AUDIO_StartAudio	Starts the audio demodulation output generation.
Declaration:	ReturnStatus AUDIO_StartAudio();
Return Values:	
noError:	The audio demodulation output generation has started.
AUDIO_StopAudio	Stops the audio demodulation output generation.
Declaration:	ReturnStatus AUDIO_StopAudio()
Return Values:	
noError:	The audio demodulation output generation has stopped.
AUDIO_GetData	Stores audio data in the data parameter.
Declaration:	ReturnStatus AUDIO_GetData(int16_t* data, uint16_t inSize, uint16_t* outSize);
Parameters:	
data:	Pointer to a 16 bit integer. Contains an array of audio data when the function completes.
inSize:	The maximum amount of audio data samples allowed. The outSize parameter will not exceed this value.
outSize:	The amount of audio data samples stored in the data array.
Return Values:	
noError:	The data parameter is filled with audio data.
Additional Detail	The outSize variable specifies the amount of audio samples stored in the array. The inSize value specifies the maximum amount of audio samples allowed.
AUDIO_SetMode	Sets the audio demodulation mode.
Declaration:	ReturnStatus AUDIO_SetMode(AudioDemodMode mode);
Parameters:	
mode:	

AudioDemodMode	Value
ADM_FM_8KHZ	0
ADM_FM_13KHZ	1
ADM_FM_75KHZ	2
ADM_FM_200KHZ	3
ADM_AM_8KHZ	4
ADM_MODE_NONE	5

Return Values:	
noError:	The audio demodulation mode has been successfully set.

Table 3: Audio functions (cont.)

AUDIO_GetMode — Queries the audio demodulation mode.

Declaration: ReturnStatus AUDIO_GetMode(AudioDemodMode* _mode);

Parameters:

_mode: Pointer to an AudioDemodMode. Contains the audio demodulation mode when the function completes.

AudioDemodMode	Value
ADM_FM_8KHZ	0
ADM_FM_13KHZ	1
ADM_FM_75KHZ	2
ADM_FM_200KHZ	3
ADM_AM_8KHZ	4
ADM_MODE_NONE	5

Return Values:

noError: The audio demodulation mode has been successfully queried.

Additional Detail The mode type is stored in the _mode parameter.

AUDIO_SetMute — Sets the mute status.

Declaration: ReturnStatus AUDIO_SetMute(bool mute);

Parameters:

mute: Mute status. This value can be either true or false.

Return Values:

noError: The mute status has been successfully set.

Additional Detail It does not affect the data processing or callbacks. If the mute parameter is true, the output speakers are muted. If the mute parameter is false, the output speakers are not muted.

AUDIO_GetMute — Queries the status of the mute operation.

Declaration: ReturnStatus AUDIO_GetMute(bool* _mute);

Parameters:

_mute: Pointer to a bool. Contains the mute status of the output speakers when the function completes.

Return Values:

noError: The mute status has been successfully queried.

Additional Detail If the value of _mute is true, the speaker output is muted. If the value of _mute is false, the speaker output is not muted. The status of the mute operation does not stop the audio processing or data callbacks.

Table 3: Audio functions (cont.)

AUDIO_SetVolume	Sets the volume value and must be a real number ranging from 0 to 1.
Declaration:	ReturnStatus AUDIO_SetVolume(float volume);
Parameters:	
volume:	Volume value. Range: 0.0 to 1.0.
Return Values:	
noError:	The volume has successfully been set.
Additional Detail	If the value is outside of the specified range, clipping occurs.
AUDIO_GetVolume	Queries the volume and must be a real value ranging from 0 to 1.
Declaration:	ReturnStatus AUDIO_GetVolume(float* _volume);
Parameters:	
_volume:	Pointer to a float. Contains a real number ranging from 0 to 1.
Return Values:	
noError:	The volume has been successfully queried.
Additional Detail	If the value is outside of the specified range, clipping occurs.

Connection

Table 4: Connection functions

Search	Searches for devices that can be connected to.
Declaration:	ReturnStatus Search(long deviceIDs[], wchar_t* deviceSerial[], int* numDevicesFound);
Parameters:	
deviceIDs:	Pointer to an array of long integers. Caller must allocate the array memory. On return, array contains a list device ID numbers.
deviceSerial:	Pointer to an array of wchar_t pointers. Caller must allocate the array memory. On return, array contains pointers to device serial number strings.
numDevicesFound:	Pointer to an integer. Contains the number of devices found by the search call.
Return Values:	
noError:	The search succeeded.
Additional Detail:	An array of device IDs is returned that correspond to specific devices.
Connect	Connects to a device specified by the deviceID parameter.
Declaration:	ReturnStatus Connect(long deviceID);
Parameters:	
deviceID:	Device ID found during the Search function call.
Return Values:	
noError:	The device has been connected.
errorTransfer:	The POST status could not be retrieved from the device.
errorTransfer:	The POST status could not be retrieved from the device.
errorIncompatibleFirmware:	The firmware version is incompatible with the API version.
errorNotConnected:	The device is not connected.
Additional Detail:	The deviceID value must be found by the Search function call.
ResetDevice	Reboots the specified device.
Declaration:	ReturnStatus ResetDevice(long deviceID);
Return Values:	
noError:	The device has been rebooted.
errorRebootFailure:	The reboot failed.
Disconnect	Stops data acquisition and disconnects from the connected device.
Declaration:	ReturnStatus Disconnect();
Return Values:	
noError:	The device has been disconnected.
errorDisconnectFailure:	The disconnect failed.

Device operation

Table 5: Device operation functions

Run	Starts data acquisition.
Declaration:	ReturnStatus Run();
Return Values:	
noError:	The device has begun data acquisition.
errorTransfer:	The device did not receive the command.
errorNotConnected:	The device is not connected.
Additional Detail:	To continually generate data, this function must be called after each data acquisition (call from GetIQData).
Stop	Stops data acquisition.
Declaration:	ReturnStatus Stop();
Return Values:	
noError:	The data acquisition has stopped.
errorTransfer:	The device did not receive the command.
errorNotConnected:	The device is not connected.
Additional Detail:	This function must be called when changes are made to values that affect the signal.
GetRunState	Queries the run mode.
Declaration:	ReturnStatus GetRunState(RunMode* runMode);
Parameters:	
runMode:	Pointer to a RunMode type. Contains the run state when the function completes. The run state can be either stopped or running.
Return Values:	
noError:	The run state has been queried.
errorNotConnected:	The device is not connected.
Additional Detail:	The value is stored in the runMode parameter.
	When the device mode is in the stopped state, the device does not gather data. It waits until the device enters run mode to acquire data.
Preset	This function sets the trigger mode to freeRun, the center frequency to 1.5 GHz, the span to 40 MHz, the IQ record length to 1024 samples and the reference level to 0 dBm.
Declaration:	ReturnStatus Preset();
Return Values:	
noError:	The preset values have been set.
errorNotConnected:	The device is not connected.

Table 5: Device operation functions (cont.)

SetReferenceLevel	Sets the reference level.
Declaration:	ReturnStatus SetReferenceLevel(double refLevel);
Parameters:	
refLevel:	Reference level measured in dBm. Range: –130 dBm to 30 dBm.
Return Values:	
noError:	The reference level value has been set.
errorNotConnected:	The device is not connected.
Additional Detail:	The reference level adjusts the vertical scaling of the signal.
GetReferenceLevel	Queries the reference level.
Declaration:	ReturnStatus GetReferenceLevel(double* refLevel);
Parameters:	
refLevel:	Pointer to a double. Contains the reference level when the function completes. Range: –130 dBm to 30 dBm.
Return Values:	
noError:	The IQ record length value has been set.
errorNotConnected:	The device is not connected.
Additional Detail:	The value is stored in the refLevel parameter.
SetCenterFreq	Sets the center frequency value.
Declaration:	ReturnStatus SetCenterFreq(double cf);
Parameters:	
cf:	Pointer to a double. Contains the center frequency when the function completes. Range: 0 Hz to 6.2 GHz.
Return Values:	
noError:	The center frequency has been queried.
errorNotConnected:	The device is not connected.
GetCenterFreq	Queries the center frequency.
Declaration:	ReturnStatus GetCenterFreq(double* cf);
Parameters:	
cf:	Pointer to a double. Contains the center frequency when the function completes. Range: 0 Hz to 6.2 GHz.
Return Values:	
noError:	The center frequency has been queried.
errorNotConnected:	The device is not connected.
Additional Detail:	The center frequency determines the center location for the spectrum view.

Table 5: Device operation functions (cont.)

GetTunedCenterFreq	Queries the center frequency used by the hardware.
Declaration:	ReturnStatus GetTunedCenterFreq(double* cf);
Parameters:	
cf:	Pointer to a double. Contains the tuned center frequency when the function completes.
Return Values:	
noError:	The tuned center frequency has been queried.
Additional Detail:	This value needs an offset to match the desired center frequency.
GetMaxCenterFreq	Queries the maximum center frequency.
Declaration:	ReturnStatus GetMaxCenterFreq(double* maxCF);
Parameters:	
maxCF:	Pointer to a double. Contains the maximum center frequency when the function completes.
Return Values:	
noError:	The maximum centerFrequency value has been queried.
errorNotConnected:	The device is not connected.
Additional Detail:	The value is stored in the maxCF parameter.
SetIQBandwidth	Sets the IQ bandwidth value.
Declaration:	ReturnStatus SetIQBandwidth(double iqBandwidth);
Parameters:	
iqBandwidth:	IQ bandwidth value measured in Hz. Range: 100 Hz to 6.2 GHz.
Return Values:	
noError:	The IQ bandwidth value has been set.
errorNotConnected:	The device is not connected.
Additional Detail:	The IQ bandwidth value determines the range of data to be examined. In the frequency domain, the IQ bandwidth value is the span.
GetIQBandwidth	Queries the IQ bandwidth value.
Declaration:	ReturnStatus GetIQBandwidth (double* iqBandwidth);
Parameters:	
iqBandwidth:	Pointer to a double. Contains the IQ bandwidth value when the function completes.
Return Values:	
noError:	The IQ bandwidth has been queried.
errorNotConnected:	The device is not connected.
Additional Detail:	The value is stored in the iqBandwidth parameter. The IQ bandwidth value determines the range of data to be examined. In the frequency domain, the IQ bandwidth value is the span. When this value is above 40 MHz, the data will be stitched.

Table 5: Device operation functions (cont.)

GetIQSampleRate	This value is 56 MHz.
Declaration:	ReturnStatus GetIQSampleRate(double* iqSampleRate);
Parameters:	
iqSamplingRate:	Pointer to a double. Contains the IQ sampling frequency when the function completes.
Return Values:	
noError:	The IQ sampling frequency was successfully queried.
SetIQRecordLength	Sets the amount of IQ data samples in each data acquisition.
Declaration:	ReturnStatus SetIQRecordLength(long recordLength);
Parameters:	
recordLength:	IQ record length. This value is measured in samples. Range: 2 – 104.8576 M samples.
Return Values:	
noError:	The IQ record length value has been set.
errorNotConnected:	The device is not connected.
GetIQRecordLength	Queries the IQ record length.
Declaration:	ReturnStatus GetIQRecordLength(long* recordLength);
Parameters:	
recordLength:	Pointer to a long. Contains the amount of IQ data samples to be generated with each acquisition. Range: 2 – 104.8576 M samples.
Return Values:	
noError:	The IQ record length has been queried.
errorNotConnected:	The device is not connected.
Additional Detail:	The value is stored in the recordLength parameter.

Table 5: Device operation functions (cont.)

SetExternalRefEnable	Enables or disables the external reference.
Declaration:	ReturnStatus SetExternalRefEnable(bool exRefEn);
Parameters:	
exRefEn:	Enables or disables the external reference.
Return Values:	
noError:	The external reference has been enabled or disabled.
errorNotConnected:	The device is not connected.
errorTimeout:	The operation has not finished after 2 seconds.
Additional Detail:	When the exRefEn parameter is true, the external reference is enabled. When the exRefEn parameter is false, the external reference is disabled. When the external reference is enabled, an external reference signal must be connected to the “Ref In” port. The signal must have a frequency of 10 MHz with a +10 dBm maximum amplitude. This signal is used by the local oscillators to mix with the input signal. When the external reference is disabled, the local oscillators provide a reference signal to mix with the input signal.
GetExternalRefEnable	Queries the state of the external reference.
Declaration:	ReturnStatus GetExternalRefEnable(bool* exRefEn);
Parameters:	
exRefEn:	Pointer to a bool. Contains the status of the external reference when the function completes.
Return Values:	
noError:	The function has completed successfully.
Additional Detail:	When the external reference is enabled, the exRefEn parameter will be true. When the external reference is disabled, the exRefEn parameter will be false.
GetMaxIQBandwidth	Queries the maximum bandwidth value.
Declaration:	ReturnStatus GetMaxIQBandwidth(double* maxBandwidth);
Parameters:	
maxBandwidth:	Pointer to a double. It contains the maximum bandwidth value when the function completes.
Return Values:	
noError:	The maximum bandwidth value has been queried.
errorNotConnected:	The device is not connected.
Additional Detail:	The value is stored in the maxBandwidth parameter.

Table 5: Device operation functions (cont.)

GetMinIQBandwidth	Queries the minimum span value.
Declaration:	ReturnStatus GetMinIQBandwidth(double* minBandwidth);
Parameters:	
minBandwidth:	Pointer to a double. Contains the minimum bandwidth value when the function completes. This value is passed by reference. After the function has completed, the value will contain the minimum bandwidth value.
Return Values:	
noError:	The minimum bandwidth value has been queried.
errorNotConnected:	The device is not connected.
Additional Detail:	The value is stored in the minBandwidth parameter.
GetMaxAcquisitionSamples	Returns the maximum number of ADC samples allowed by the system.
Declaration:	ReturnStatus GetMaxAcquisitionSamples (unsigned long* maxSamples);
Parameters:	
maxCF:	Pointer to an unsigned long. Contains the maximum center frequency when the function completes.
Return Values:	
noError:	The maximum maxSamples value has been queried.

Device status

Table 6: Device status functions

GetAPIVersion	Stores the API version number in the apiVersion parameter.
Declaration:	ReturnStatus GetAPIVersion(char* apiVersion);
Parameters:	
apiVersion:	String that contains the API version number when the function completes.
Return Values:	
noError:	The API version number has been successfully stored in the apiVersion parameter.
Additional Detail:	The API version number has the form: “majorNumber.minorNumber.revision-Number”. For example: “3.4.0145”: 3 = major number, 4 = minor number, 0145 = revision number
GetDeviceNomenclature	Stores the name of the device in the nomenclature parameter.
Declaration:	ReturnStatus GetDeviceNomenclature(char* nomenclature);
Parameters:	
nomenclature:	String that contains the name of the device when the function completes.
Return Values:	
noError:	The string name has been set.
GetDeviceSerialNumber	Stores the serial number of the device in the serialNum parameter.
Declaration:	ReturnStatus GetDeviceSerialNumber(char* serialNum);
Parameters:	
serialNum:	String that contains the serial number of the device when the function completes.
Return Values:	
noError:	The device serial number has been set.
errorNotConnected:	The device is not connected.
GetFirmwareVersion	Stores the firmware version number in the fwVersion parameter.
Declaration:	ReturnStatus GetFirmwareVersion(char* fwVersion);
Parameters:	
fwVersion:	String that contains the firmware version number when the function completes.
Return Values:	
noError:	The firmware version has been stored in the variable.
errorNotConnected:	The device is not connected.
Additional Detail:	The firmware version number has the form: “Vmajor.minor”. For example: “V3.4”: major = 3, minor = 4

Table 6: Device status functions (cont.)

GetFPGAVersion	Stores the FPGA version number in the fpgaVersion parameter.
Declaration:	ReturnStatus GetFPGAVersion(char* fpgaVersion);
Parameters:	
fpgaVersion:	String that contains the FGPA version number when the function completes.
Return Values:	
noError:	The FPGA version number has been stored in the variable.
errorNotConnected:	The device is not connected.
Additional Detail:	The FPGAVersion has the form: “Vmajor.minor”. For example: “V3.4”: major = 3, minor = 4
GetHWVersion	Stores the hardware version in a string. It has the form: “V versionNumber”.
Declaration:	ReturnStatus GetHWVersion(char* hwVersion);
Parameters:	
hwVersion:	String that contains the hardware version when the function completes.
Return Values:	
noError:	The HW version number is stored in the hwVersion parameter.
errorNotConnected:	The device is not connected.

Table 6: Device status functions (cont.)

GetErrorString	Returns a string that corresponds to the ReturnStatus value specified by the status parameter.
Declaration:	ReturnStatus const char* GetErrorString(ReturnStatus status);
Parameters:	
status:	A ReturnStatus value.
Return Values:	
noError:	No Error
errorNotConnected:	Not Connected
errorTimeout	Timeout Error
errorTransfer	Transfer Error
errorFileOpen	Opening File
errorParameter	Parameter Error
errorDataNotReady	Data Not Ready
errorInvalidCalibConstant-FileFormat	Invalid Calibration Constant File Format
errorMismatchCalibCon-stantsSize	Mismatched Calibration Constant Size
errorFailed	Failed
errorCRC	CRC Error
errorWriteCalConfigHeader	Cal Config Header Write
errorWriteCalConfigData	Cal Config Data Write
errorReadCalConfigHeader	Cal Config Header Read
ErrorReadCalConfigData	Cal Config Data Read
errorReadCalConfigHeader	Cal Config Header Read
errorReadCalConfigData	Cal Config Data Read
errorEraseCalConfig	Cal Config Erase
errorCalConfigFileSize	Cal Config File Size
errorChangeToFlashMode	Changing to Flash Mode
errorChangeToRunMode	Changing to Run Mode
errorIncompatibleFirmware	Incompatible Firmware
errorStreamADCToDisk-FileOpen	Failure to open ADC streaming data file
errorStreamADC-ToDiskAlreadyStreaming	Cannot change ADC file streaming parameters while streaming is active
errorStreamADCToDiskBad-Path	Nonexistant path or insufficient privileges to open file in specified path
errorStreamADCToDisk-ThreadFailure	ADC streaming to disk thread failure
errorRebootFailure	Failed to reboot instrument
errorLOLockFailure	Local oscillator did not achieve lock

Table 6: Device status functions (cont.)

errorPOSTFailureFPGALoad	Device internal HW configuration failure
errorPOSTFailureHiPower	USB port power insufficient for device
errorPOSTFailureI2C	Device internal control bus test failure
errorPOSTFailureGPIF	Device internal data bus test failure
errorPOSTFailureUsbSpeed	Device failed to connect as USB 3.0
errorPlaceholder	Placeholder Error
notImplemented	Not Implemented

DPX

Table 7: DPX functions

SetDPXEnabled	Enables or disables DPX.
Declaration:	ReturnStatus SetDPXEnabled(bool enabled);
Parameters:	
enabled:	Enables or disables DPX.
Return Values:	
noError:	DPX has been successfully enabled or disabled.
Additional Detail:	If the value is true, DPX is enabled. If the value is false, DPX is disabled.
GetDPXEnabled	Checks the status of DPX.
Declaration:	ReturnStatus GetDPXEnabled(bool* enabled);
Parameters:	
enabled:	Pointer to a bool. It queries the state of the DPX mode.
Return Values:	
noError:	The operation completed successfully.
Additional Detail:	If DPX is enabled, the enabled parameter will be true. If DPX is disabled, the enabled parameter will be false.

Table 7: DPX functions (cont.)

DPX_SetParameters

Sets the DPX span, resolution bandwidth, trace pointer per pixel, Y-axis units, maximum Y-axis value, minimum Y-axis value, infinite persistence, persistence time and show only trigger frame.

Declaration:

ReturnStatus DPX_SetParameters(double fspan, double rbw, int32_t bitmapWidth, int32_t tracePtsPerPixel, VerticalUnitTypes yUnit, double yTop, double yBottom, bool infinitePersistence, double persistenceTimeSec, bool showOnlyTrigFrame);

Parameters:

fspan: Span measured in Hz.

This value must be greater than or equal to 0 and less than or equal to 40 MHz.

rbw: Resolution bandwidth measured in Hz.

This value must be greater than or equal to 0.

bitmapWidth: Bitmap width measured in pixels.

This value must be greater than 0 and less than or equal to 801.

tracePtsPerPixel: Trace points per pixel.

Valid values are: 1, 3, 5.

yUnit: Units of the Y-axis.

VerticalUnitTypes	Value
VerticalUnit_dBm	0
VerticalUnit_Watt	1
VerticalUnit_Volt	2
VerticalUnit_Amp	3

yTop: The maximum value on the Y-axis in yUnit.

This value must be higher than yBottom.

yBottom: The minimum value on the Y-axis in yUnit.

infinitePersistence: Enables or disables infinite persistence. It causes every data point to remain on the screen when enabled.

persistenceTimeSec: The amount of time that previous signals remain on the screen.

showOnlyTrigFrame: Enables or disables showing only trigger frames.

Return Values:

noError: The function has executed successfully.

DPX_Configure

Enables or disables the DPX spectrum and DPX spectrogram modes.

Declaration:

ReturnStatus DPX_Configure(bool enableSpectrum, bool enableSpectrogram);

Parameters:

enableSpectrum: Enables or disables spectrum. If the variable is true, spectrum is enabled. If the variable is false, spectrum is disabled.

enableSpectrogram: Enables or disables spectrogram. If the variable is true, spectrum is enabled. If the variable is false, spectrum is disabled.

Return Values:

noError: The function has executed successfully.

Table 7: DPX functions (cont.)

DPX_GetSettings

Queries the current DPX settings.

Declaration:

ReturnStatus DPX_GetSettings(DPX_SettingStruct *dpxSettings);

Parameters:

dpxSettings: Pointer to DPX_SettingsStruct.

DPX_SettingsStruct
bool enableSpectrum
bool enableSpectrogram
int32_t bitmapWidth
int32_t bitmapHeight
int32_t traceLength
float decayFactor
double actualRBW

Return Values:

noError: The function has executed successfully.

DPX_SetSpectrumTraceType

Specifies one of the three traces with the traceIndex parameter and sets its trace type with the type parameter.

Declaration:

ReturnStatus DPX_SetSpectrumTraceType(int32_t traceIndex, TraceTypes type);

Parameters:

traceIndex: Specifies the trace to be set. It can be 0, 1, or 2.

type: A value of type TraceTypes.

TraceTypes	Value
TraceTypeAverage	0
TraceTypeMax	1
TraceTypeMaxHold	2
TraceTypeMin	3
TraceTypeMinHold	4

Return Values:

noError: The function has executed successfully.

DPX_FindRBWRange

Queries the valid RBW range based on span.

Declaration:

ReturnStatus DPX_FindRBWRange(double fspan, double *minRBW, double *maxRBW);

Parameters:

fpsan: Span measured in Hz. This value must be greater than or equal to 0.

minRBW: Returns minimum RBW in Hz.

maxRBW: Returns maximum RBW in Hz.

Return Values:

noError: The function has executed successfully.

Table 7: DPX functions (cont.)

DPX_ResetDPx	Clears the spectrum bitmap, resets the spectrum traces, resets the spectrogram bitmap, resets the spectrogram traces, resets the processing time, resets the processing timers, sets the FFT count to 0, sets the frame count to 0, sets the last FFT timestamp to 0 and sets the FFT interval index to 0.
Declaration:	ReturnStatus DPX_ResetDPx();
Return Values:	
noError:	The function has executed successfully.
DPX_GetFrameInfo	Queries the frame count and FFT count.
Declaration:	ReturnStatus DPX_GetFrameInfo(int64_t* frameCount, int64_t* fftCount);
Parameters:	
frameCount:	Pointer to a 64 bit integer. Contains the frame count value when the function completes.
fftCount:	Pointer to a 64 bit integer. Contains the FFT count when the function completes.
Return Values:	
noError:	The function has executed successfully.
DPX_SetSogramParameters	Sets the amount of time that each spectrogram line represents and the power range of the spectrogram.
Declaration:	ReturnStatus DPX_SetSogramParameters(double timePerBitmapLine, double timeResolution, double maxPower, double minPower);
Parameters:	
timePerBitmapLine:	The amount of time per bitmap line.
timeResolution:	The amount of time that each spectrogram line represents.
maxPower:	The maximum power of the spectrogram.
minPower:	The minimum power of the spectrogram.
Return Values:	
noError:	The function has executed successfully.
DPX_SetSogramTraceType	Sets the DPX spectrogram trace type.
Declaration:	ReturnStatus DPX_SetSogramTraceType(TraceTypes traceType);
Parameters:	
traceType:	A value of type TraceTypes.

TraceTypes	Value
TraceTypeAverage	0
TraceTypeMax	1
TraceTypeMaxHold	2
TraceTypeMin	3
TraceTypeMinHold	4

Return Values:	
noError:	The function has executed successfully.
Additional Detail:	A trace keeps track of a specific measurement. The DPX spectrogram can keep track of the maximum value, the minimum value or the average value. If the max hold or min hold traces are selected, an error occurs.

Table 7: DPX functions (cont.)

DPX_GetSogramSettings	Queries DPX spectrogram bitmap width, bitmap height, trace line time and bitmap line time.
Declaration:	ReturnStatus DPX_GetSogramSettings(DPX_SogramSettingsStruct *sogramSettings);
Parameters:	
sogramSettings:	Pointer to DPX_SogramSettingsStruct.

DPX_SogramSettingsStruct
int32_t bitmapWidth
int32_t bitmapHeight
double sogramTraceLineTime
double sogramBitmapLineTime

Return Values:	
noError:	The function has executed successfully.
DPX_GetSogramHiResLineCountLatest	Queries the amount of high resolution lines in the DPX spectrogram.
Declaration:	ReturnStatus DPX_GetSogramHiResLineCountLatest(int32_t* lineCount);
Parameters:	
lineCount:	Pointer to a 32 bit integer. Contains the amount of high resolution lines in the spectrogram when the function completes.
Return Values:	
noError:	The function has executed successfully.
DPX_GetSogramHiResLineTriggered	Checks the DPX high resolution spectrogram line specified by the lineIndex parameter.
Declaration:	ReturnStatus DPX_GetSogramHiResLineTriggered(bool* triggered, int32_t lineIndex);
Parameters:	
triggered:	Pointer to a bool. If the specified high resolution line is triggered, the value will be true. If the high resolution line is not triggered, the value will be false.
lineIndex:	The index of the high resolution spectrogram line.
Return Values:	
noError:	The function has executed successfully.
Additional Detail:	If this line is triggered, the triggered parameter will be true. If the line is not triggered, the triggered parameter will be false.

Table 7: DPX functions (cont.)

DPX_GetSogramHiResLineTimes-tamp	Queries the spectrogram high resolution timestamp.
Declaration:	ReturnStatus DPX_GetSogramHiResLineTimestamp(double* timestamp, int32_t lineIndex);
Parameters:	
timestamp:	Pointer to a double. Contains the timestamp value when the function completes.
lineIndex:	The index of the high resolution spectrogram line.
Return Values:	
noError:	The function has executed successfully.
Additional Detail:	The timestamp is started by the FPGA.
DPX_GetSogramHiResLine	Queries the high resolution line specified by the lineIndex parameter.
Declaration:	ReturnStatus DPX_GetSogramHiResLine(int16_t* vData, int32_t* vDataSize, int32_t lineIndex, double* dataSF, int32_t tracePoints, int32_t firstValidPoint);
Parameters:	
vData:	Pointer to a 16 bit integer. Contains the data stored in the spectrogram line.
vDataSize:	Pointer to a 32 bit integer. Contains the amount of elements in the vData parameter.
lineIndex:	The spectrogram line index.
dataSF:	Pointer to a double. Contains the scale factor.
tracePoints:	The amount of trace points.
firstValidPoint:	First valid trace point.
Return Values:	
noError:	The function has executed successfully.
Additional Detail:	The data stored at the specified line is stored in the vData parameter.
WaitForDPXDataReady	Waits for the DPX data to be ready to be queried.
Declaration:	ReturnStatus WaitForDPXDataReady(int timeoutMsec, bool* ready);
Parameters:	
timeoutMsec:	Timeout value measured in ms.
ready:	Pointer to a bool. Its value determines the status of the data.
Return Values:	
noError:	The function has executed successfully.
Additional Detail:	If the data is not ready and the timeout value is exceeded, the ready parameter will be false. Otherwise, the data is ready for acquisition and the ready parameter will be true.

Table 7: DPX functions (cont.)

DPX_GetFrameBuffer	This function returns the DPX Frame Buffer containing the latest DPX bitmaps and traces.
Declaration:	ReturnStatus DPX_GetFrameBuffer(DPX_FrameBuffer* frameBuffer);
Parameters:	
frameBuffer:	Pointer to a DPX_FrameBuffer struct.
Return Values:	
noError:	The function has executed successfully.
DPX_FinishFrameBuffer	This function specifies that the frame is finished. It must be called before the next frame will be available.
Declaration:	ReturnStatus DPX_FinishFrameBuffer();
Return Values:	
noError:	The function has executed successfully.
DPX_IsFrameBufferAvailable	This function checks DPX frame availability.
Declaration:	ReturnStatus DPX_IsFrameBufferAvailable(bool* frameAvailable);
Parameters:	
frameAvailable:	Pointer to a bool.
Return Values:	
noError:	The function has executed successfully.
Additional Detail:	Specifies the frame availability with the frameAvailable parameter. If the frameAvailable value is true, the frame is available. If the frameAvailable value is false, the frame is not available.

Table 8: DPX_FrameBuffer description

DPX_FrameBuffer	Description
int32_t fftPerFrame	Number of FFT performed in this frame.
int64_t fftCount	Total number of FFT performed since DPx acquisition started.
int64_t frameCount	Total number of DPx frames since DPx acquisition started.
double timestamp	Acquisition timestamp of this frame.
uint32_t acqDataStatus	Acquisition data status. See AcqDataStatus enum.
double minSigDuration	Minimum signal duration in seconds for 100% POI.
bool minSigDurOutOfRange	Minimum signal duration out of range.
int32_t spectrumBitmapWidth	Spectrum bitmap width in pixels.
int32_t spectrumBitmapHeight	Spectrum bitmap height in pixels.
int32_t spectrumBitmapSize	Total number of pixels in Spectrum bitmap (spectrumBitmapWidth * spectrumBitmapHeight).
int32_t spectrumTraceLength	Number of trace points in Spectrum trace.
int32_t numSpectrumTraces	Number of Spectrum traces.
bool spectrumEnabled	True, DPX Spectrum is enable. False, DPX Spectrum is disabled. See DPX_Configure.

Table 8: DPX_FrameBuffer description (cont.)

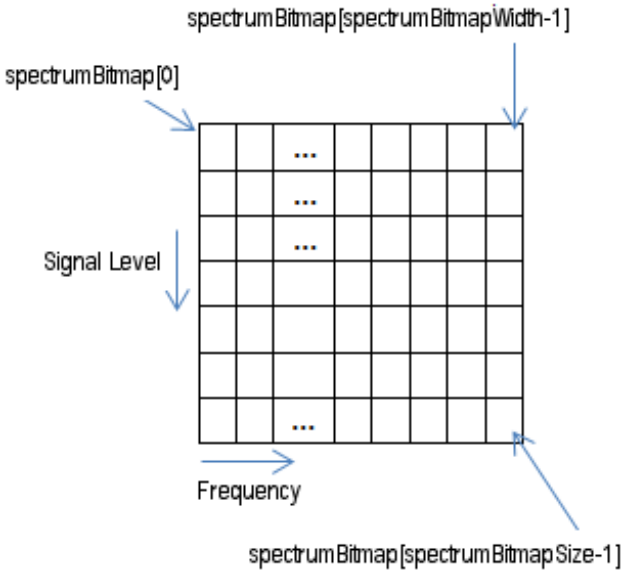
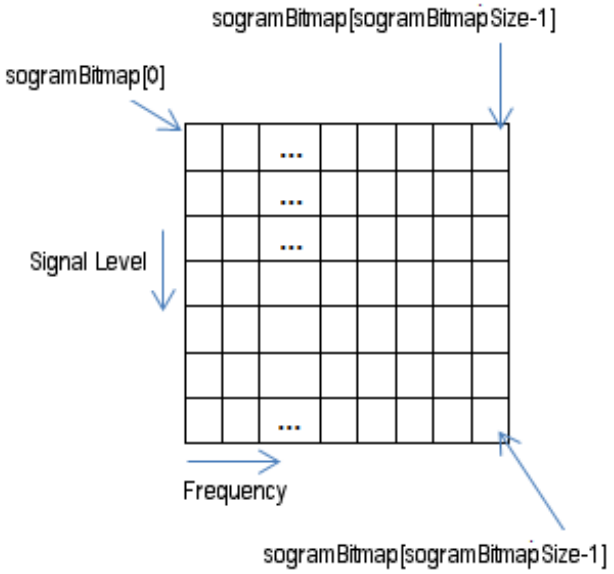
DPX_FrameBuffer	Description
bool spectrogramEnabled	True, DPX Spectogram is enable. False, DPX Spectogram is disabled. See DPX_Configure.
float* spectrumBitmap	DPX Spectrum bitmap array. Each value represents the hit count of each pixel in the DPX Spectrum bitmap. The first element in the array represents the upper left corner of the bitmap and the second element represents the pixel to the right of the first pixel. The last element represents the lower right corner of the bitmap. The following diagram shows the Spectrum bitmap and spectrumBitmap array indexes. The x axis in the bitmap represents spectrum frequency and the y axis represents spectrum signal level. 
float** spectrumTraces	Spectrum traces array. The first n elements represents spectrum trace 0 and the next n elements represents spectrum trace 1 and so forth, where n is the value of spectrumTraceLength (see SPECTRUM_SetSettings). Each trace point represents the spectrum power in Watts.
int32_t sogramBitmapWidth	Spectrogram bitmap width in pixels.
int32_t sogramBitmapHeight	Spectrogram bitmap height in pixels.
int32_t sogramBitmapSize	Total number of pixels in Spectrogram bitmap (sogramBitmapWidth * sogramBitmapHeight).
int32_t sogramBitmapNumValidLines	Number of valid horizontal lines (spectrums) in Spectrogram bitmap.

Table 8: DPX_FrameBuffer description (cont.)

DPX_FrameBuffer	Description
uint8_t* sogramBitmap	Spectrogram map array. Each element represent the scaled signal level in the increment of: (maxPower – minPower) / 254 where maxPower and minPower are the parameters from DPX_SetSogramParameters(). If the pixel value is 0, it represents signal level <= minPower. If the pixel value is 254, it represents signal level >= maxPower. The first row in the spectrogram bitmap represents the spectrum with the latest time and the last row in the bitmap represents the oldest spectrum. 
double* sogramBitmapTimestampArray	Spectrogram bitmap timestamps. Each element in the array represents the timestamp of each row in the bitmap. The first element represents the latest spectrum and the last element represents the oldest spectrum.
int16_t* sogramBitmapContainTriggerArray	Spectrogram bitmap trigger. Each element in the array indicates if trigger occurred during spectrum acquisition in the bitmap. A value of 1 indicates trigger occurred and a value of 0 indicates no trigger occurred. The first element represents the latest spectrum and the last element represents the oldest spectrum.

IQ data

Table 9: IQ data functions

GetIQHeader	This function queries the current IQ header.
Declaration:	ReturnStatus GetIQHeader(IQHeader* header);
Parameters:	
header:	Pointer to a value of type IQHeader. IQHeader types: ■ uint16_t acqDataStatus ■ uint64_t acquisitionTimestamp ■ uint32_t frameID ■ uint16_t trigger1Index ■ uint16_t trigger2Index ■ uint16_t timeSyncIndex
Return Values:	
noError:	The header has been successfully queried.
errorDataNotReady:	The output data is not ready.
Additional Detail:	The header parameter is populated with the acquisition timestamp, the acquisition data status, the frame ID, the time sync index, the trigger 1 and trigger 2 indexes.

Table 9: IQ data functions (cont.)

GetIQData — Queries the IQ data in interleaved mode.

Declaration: ReturnStatus GetIQData(float* iqData, int startIndex, int length);

Parameters:

iqData: Pointer to a float. Contains I-data and Q-data at alternating indexes of the array when the function completes.

startIndex: Starting index of the IQ record. The sum of the startIndex and length must be less than the IQ record length.

length: Amount of samples of IQ data to acquire. The sum of the startIndex and length must be less than the IQ record length.

Return Values:

noError: The I data and Q data have been stored in the iqData buffer.

errorDataNotReady: There is not enough IQ data acquired to fill the IQ data record length.

Additional Detail: The I-data values are stored at even indexes of the iqData buffer, and the Q-data values are stored at odd indexes of the iqData buffer. The I-data value are the real part, and the Q-data values are the imaginary part of the complex IQ data.

The image below illustrates the iqData buffer and its conversion to IQ data.

iqData Buffer, length = 2

Index	0	1	2	3
Contents	I_0	Q_0	I_1	Q_1

Actual IQ Data, length = 2

Index	0	1
Contents	$I_0 + Q_0\sqrt{-1}$	$I_1 + Q_1\sqrt{1}$

Table 9: IQ data functions (cont.)

GetIQDataDeinterleaved — Passes two arrays by reference.

Declaration: ReturnStatus GetIQDataDeinterleaved(float* iData, float* qData, int startIndex, int length);

Parameters:

iData: Pointer to a float. Contains an array of I-data when the function completes.

qData: Pointer to a float. Contains an array of Q-data when the function completes. The Q-data is not imaginary.

startIndex: Starting index of the IO record. The sum of the startIndex and length must be less than the IQ record length.

length: Amount of samples of IQ data to acquire. The sum of the startIndex and length must be less than the IQ record length.

Return Values:

noError: The IQ record length has been queried.

errorDataNotReady: There is not enough IQ data acquired to fill the IQ data record length.

Additional Detail: When complete, the iData array is filled with I- data and the qData array is filled with Q-data. The Q-data is not imaginary.

See the following illustration.

iData, length =2:

Index	0	1	2	3
Contents	I_0	Q_0	I_1	Q_1

qData, length =2:

Index	0	1
Contents	Q_0	Q_1

Actual IQ Data, length =2:

Index	0	1
Contents	$I_0 + Q_0\sqrt{-1}$	$I_1 + Q_1\sqrt{-1}$

Table 9: IQ data functions (cont.)

GetIQDataCplx	Passes an array of structs by reference.
Declaration:	ReturnStatus GetIQDataCplx(Cplx32* iqData, int startIndex, int length);
Parameters:	
iqData:	Pointer to an array of Cplx32 structs. Contains the IQ data when the function completes.
startIndex:	Starting index of the IQ record. The sum of the startIndex and length must be less than the IQ record length.
length:	Amount of samples of IQ data to acquire. The sum of the startIndex and length must be less than the IQ record length.
Return Values:	
noError:	The IQ record length has been queried.
errorDataNotReady:	There is not enough IQ data acquired to fill the IQ data record length.
Additional Detail:	When complete, the iqData array is filled with I-data and Q-data. See the following illustration. iqData, length = 2 Index: 0 — 1 Contents: I_0, Q_0 — I_1, Q_1 Actual IQ Data, length =2: Index: 0 — 1 Contents: $I_0 + Q_0\sqrt{-1}$ — $I_1 + Q_1\sqrt{-1}$
WaitForIQDataReady	Waits for the data to be ready to be queried.
Declaration:	ReturnStatus WaitForIQDataReady(int timeoutMsec, bool* ready);
Parameters:	
timeoutMsec:	Timeout value measured in ms.
ready:	Pointer to a bool. Its value determines the status of the data.
Return Values:	
noError:	The function has executed successfully.
Additional Detail:	If the data is not ready and the timeout value is exceeded, the ready parameter will be false. Otherwise, the data is ready for acquisition and the ready parameter will be true.

IQ streaming

NOTE. *When IQ Streaming is active, it should be the only API data processing function in operation. No other processing function (DPx, IQ Block, Audio Demod, IF streaming or Spectrum) should be running at the same time. If other data processing is active, it may overload the computer processing capability, causing gaps or dropouts in the streamed IQ data. There can also be conflicts in some control parameters between IQ Streaming and the other processing operations.*

Most IQSTREAM control parameters should only be set/changed while IQStream processing is in its Stopped state (IQSTREAM_Stop()). Changing parameters while IQStream processing is running does not correctly apply the new values.

Table 10: IQ Streaming functions

IQSTREAM_SetAcqBandwidth	Sets the users request for the acquisition bandwidth of the output IQ data stream samples.
Declaration:	ReturnStatus IQSTREAM_SetAcqBandwidth(double bwHz_req);
Parameters:	
bwHz_req:	Requested acquisition bandwidth of IQ Streaming output data, in Hz.
Return Values:	
noError:	The requested value was accepted
Additional Detail:	No checking of the input value is done by this function. See the table in IQSTREAM_GetAcqParameters() for the mapping of requested bandwidth to actual output bandwidth provided. ***NOTE.*** *The Acq Bandwidth setting should only be changed when the instrument is in the* ***global*** *Stopped state. The new BW setting does not take effect until the global system state is cycled from Stopped to Running.*
IQSTREAM_GetAcqParameters	Reports the processing parameters of IQ output bandwidth and sample rate resulting from the users requested bandwidth.
Declaration:	ReturnStatus IQSTREAM_GetAcqParameters(double* bwHz_act, double* srSps);
Parameters:	
bwHz_act:	Ptr-to-double. Returns actual acquisition bandwidth of IQ Streaming output data, in Hz.
srSps:	Ptr-to-double. Returns actual sample rate of IQ Streaming output data, in Samples/sec
Return Values:	
noError:	The query was successful
Additional Detail:	This table gives the mapping of requested bandwidth to actual output bandwidth and sample rate.

Requested BW	Output BW	Output Sample Rate
BW ≤ 5 MHz	5 MHz	7.0 Msps
5 MHz < BW ≤ 10 MHz	10 MHz	14.0 Msps
10 MHz < BW ≤ 20 MHz	20 MHz	28.0 Msps
BW > 20 MHz	40 MHz	56.0 Msps

Table 10: IQ Streaming functions (cont.)

IQSTREAM_SetOutputConfiguration

Sets the output data destination and IQ data type.

Declaration:

ReturnStatus IQSTREAM_SetOutputConfiguration(IQSOUTDEST dest, IQSOUTDTYPE dtype);

Parameters:

dest:

Destination (sink) for IQ sample output. Valid settings:

dest value	Destination
IQSOD_CLIENT	Client application
IQSOD_FILE_TIQ	TIQ format file (.tiq extension)
IQSOD_FILE_SIQ	SIQ format file with header and data combined in one file (.siq extension)
IQSOD_FILE_SIQ_SPLIT	SIQ format with header and data in separate files (.siqh and .siqd extensions)

dtype:

Output IQ data type. Valid settings:

dtype value	Data type
IQSODT_SINGLE	32-bit single precision floating point (not valid with TIQ file destination)
IQSODT_INT32	32-bit integer
IQSODT_INT16	16-bit integer

Return Values:

noError: The requested settings were accepted.

errorIQStreamInvalidFile-DataType: Invalid selection of TIQ file and Single data type together.

Additional Detail:

The destination can be the client application, or files of different formats. The IQ data type can be chosen independently of the file format. IQ data values are stored in interleaved I/Q/I/Q order regardless of the destination or data type.

NOTE. *TIQ format files only allow Int32 or Int16 data types, not Single.*

IQSTREAM_SetIQDataBufferSize

Sets the requested size in IQ sample pairs of the IQ record returned to the client.

Declaration:

ReturnStatus IQSTREAM_SetIQDataBufferSize(int reqSize);

Parameters:

reqSize: Requested size of IQ output data buffer in IQ sample pairs. 0 resets to default.

Return Values:

noError: The value was accepted.

Additional Detail:

Any size can be requested, but only a limited set of actual sizes are available. Client must use IQSTREAM_GetIQDataBufferSize() to determine the actual maximum size which will be returned. The nearest available size smaller or equal to the requested size will be used.

If the default output size is acceptable, this function does not need to be used.

Table 10: IQ Streaming functions (cont.)

IQSTREAM_GetIQDataBufferSize

Returns the maximum number of IQ sample pairs which will be returned by the IQSTREAM_GetIQData () function.

Declaration:

ReturnStatus IQSTREAM_GetIQDataBufferSize(int* maxSize);

Parameters:

maxSize: Ptr-to-int. Returns maximum size IQ output data buffer required when using client IQ access. Size value is in IQ sample pairs.

Return Values:

noError: The query was successful.

Additional Detail:

The requested size value can be increased or decreased using the IQSTREAM_SetIQDataBufferSize () function. Available size values are integer multiples of 65,424 (integer multiplier range 1..8), with default size of 2*65242 = 130,848 IQ samples. The client should use the value returned by this function. Do not assume the above sizes will remain fixed.

The client application must allocate a buffer large enough to accept maxSize IQ data pairs returned when the IQSTREAM_GetIQData() function is called. The required allocated buffer sizes are given below:

Data Type	IQ Buffer data type	Required Client Buffer size
Single	Cplx32	maxSize * size(Cplx32)
Int32	CplxInt32	maxSize * size(CplxInt32)
Int16	CplxInt16	maxSize * size(CplxInt16)

Example C code client buffer allocation code (using either malloc() or new is acceptable):

Single: Cplx32* pCplx32 = new Cplx32[maxSize];

Int32: CplxInt32* pCplxInt32 = malloc(maxSize*sizeof(CplxInt32));

Int16: CplxInt16* pCplxInt16 = malloc(maxSize*sizeof(CplxInt16));

Example client function use:

int maxSize;

IQSTREAM_SetIQDataBufferSize *(500000); // request 500,000 IQ sample pairs*

IQSTREAM_GetIQDataBufferSize (&maxSize); // maxSize = 261696 returned

Cplx32 pIQdata = new Cplx32[maxSize];*

IQSTREAM_GetIQData(pIQdata, &iqlen, &iqinfo);

Table 10: IQ Streaming functions (cont.)

IQSTREAM_SetDiskFilenameBase	Sets the base filename for file output.
Declaration:	ReturnStatus IQSTREAM_SetDiskFilenameBaseW(const wchar_t* filenameBaseW) ReturnStatus IQSTREAM_SetDiskFilenameBase(const char* filenameBase);
Parameters:	
filenameBase:	Base filename for file output. This can include drive/path, as well as the common base filename portion of the file. The filename base should not include a file extension, as the file writing operation will automatically append the appropriate one for the selected file format. Wide (2 byte) or standard (byte) char string function versions are available. Other than accepting different type strings, they do the same operation.
Return Values:	
noError:	The setting was accepted.
Additional Detail:	The complete output filename has the following format: <filenameBase><suffix><.ext> <filenameBase>: as set by this function <suffix>: as set by filename suffix control in IQSTREAM_SetDiskFilenameSuffix() <.ext>: as set by destination control in IQSTREAM_SetOutputConfiguration(), [.tiq, .siq, .siqh+.siqd] If separate data and header files are generated, the same path/filename is used for both, with different extensions to indicate the contents. ReturnStatus = IQSTREAM_SetDiskFilenameBaseW(const wchar_t* filenameBaseW)

Table 10: IQ Streaming functions (cont.)

IQSTREAM_SetDiskFilenameSuffix

Sets the control that determines what, if any, filename suffix is appended to the output base filename.

Declaration:

ReturnStatus IQSTREAM_SetDiskFilenameSuffix(int suffixCtl);

Parameters:

suffixCtl: Sets the filename suffix control value.

Return Values:

noError: The setting was accepted.

Additional Detail:

See description of IQSTREAM_SetDiskFilename() for the full filename format.

***suffixCtl* value**	**Suffix generated**
IQSSDFN_SUFFIX_NONE (-2)	None. Base filename is used without suffix. *(Note that the output filename will not change automatically from one run to the next, so each output file will overwrite the previous one unless the filename is explicitly changed by calling the Set function again.)*
IQSSDFN_SUFFIX_TIMESTAMP (-1)	String formed from file creation time Format: "-YYYY.MM.DD.hh.mm.ss.msec" *(Note this time is not directly linked to the data timestamps, so it should not be used as a high-accuracy timestamp of the file data!)*
≥ 0	5 digit auto-incrementing index, initial value = *suffixCtl*. Format: "-nnnnn" *(Note index auto-increments by 1 each time IQSTREAM_Start() is invoked with file data destination setting.)*

Below are examples of output filenames generated with different suffixCtl settings. Multiple filenames show suffix auto-generation behavior with each IQSTREAM_Start(). The most recent suffixCtl setting remain in effect until changed by another function call.

(Assume <filenameBase> is "myfile" and TIQ file format is selected.)

***suffixCtl* value**	**Full Filename** (and behavior with multiple runs)
IQSSDFN_SUFFIX_NONE	"myfile.tiq" "myfile.tiq" "myfile.tiq" ...
IQSSDFN_SUFFIX_TIMESTAMP	"myfile-2015.04.15.09.33.**12.522**.tiq" "myfile-2015.04.15.09.33.**14.697**.tiq" "myfile-2015.04.15.09.33.**17.301**.tiq" ...
10 4	"myfile-0001**0**.tiq" "myfile-0001**1**.tiq" "myfile-0001**2**.tiq" ... "myfile-0000**4**.tiq" "myfile-0000**5**.tiq" ...

Table 10: IQ Streaming functions (cont.)

IQSTREAM_SetDiskFileLength

Sets the time length of IQ data written to an output file.

Declaration:

ReturnStatus IQSTREAM_SetDiskFileLength(int msec);

Parameters:

msec: Length of time in milliseconds to record IQ samples to file.

Return Values:

noError: The setting was accepted.

Additional Detail:

See IQSTREAM_GetDiskFileWriteStatus() to find how to monitor file output status to determine when it is active and completed.

msec value	File store behavior
0	No time limit on file output. File storage is terminated when *IQSTREAM_Stop()* is called.
> 0	File output ends after this number of milliseconds of samples stored. File storage can be terminated early by calling *IQSTREAM_Stop().*

IQSTREAM_Start()

Initializes IQ Stream processing and initiates data output.

Declaration:

ReturnStatus IQSTREAM_Start();

Parameters:

(none):

Return Values:

noError: The operation was successful

errorBufferAllocFailed: Internal buffer allocation failed (memory unavailable)

errorIQStreamFileOpen-Failed: Output file open (create) failed.

Additional Detail:

If the data destination is the client application, data will become available soon after the Start() function is invoked. Even if triggering is enabled, the data will begin flowing to the client without need for a trigger event. The client must begin retrieving data as soon after Start() as possible.

If the data destination is file, the output file is created, and if triggering is not enabled, data starts to be written to the file immediately. If triggering is enabled, data will not start to be written to the file until a trigger event is detected. ForceTrigger() can be used to generate a trigger event if the specified one does not occur.

IQSTREAM_Stop()

This function terminates IQ Stream processing and disables data output.

Declaration:

ReturnStatus IQSTREAM_Stop();

Parameters:

(none):

Return Values:

noError: The operation was successful.

Additional Detail:

If the data destination is file, file writing is stopped and the output file is closed.

Table 10: IQ Streaming functions (cont.)

IQSTREAM_GetEnabled()	This function returns the current IQ Stream processing state.
Declaration:	ReturnStatus IQSTREAM_GetEnabled(bool* enabled);
Parameters:	
enabled:	Ptr-to-bool. Returns the current IQ Stream processing enable status. True: IQ Stream processing is active False: IQ Stream processing is inactive
Return Values:	
noError:	The query was successful.
IQSTREAM_GetIQData()	Allows the client application to retrieve IQ data blocks generated by the IQ Stream processing.
Declaration:	ReturnStatus IQSTREAM_GetIQData(void* iqdata, int* iqlen, IQSTRMIQINFO* iqinfo);
Parameters:	
iqdata:	Pointer to buffer to return IQ sample data block.
iqlen:	Ptr-to-int. Returns number of IQ data pairs returned in iqbuffer. 0 indicates no data available.
iqinfo:	Ptr-to-struct. Returns a structure containing information about the IQ data block. (Set value to NULL if the info struct is not wanted).
Return Values:	
noError:	The query was successful
Additional Detail:	Allows the client application to retrieve IQ data blocks generated by the IQ Stream processing. Data blocks are copied out to the buffer pointed to by iqdata, which must be allocated by the client large enough to hold the output record. See IQSTREAM_GetIQDataBufferSize() to get the required buffer size. The underlying data buffer organization is interleaved I/Q data pairs of the data type configured. It is recommended to use the correct “complex” data type: Cplx32 (Single data type), CplxInt32 (Int32), CplxInt16 (Int16) to simplify accessing the data, although any buffer pointer type will be accepted. iqlen returns the number of IQ sample pairs copied out to the buffer. The returned value will be 0 if no data is available. The client can poll the function, waiting for iqlen>0 to indicate data is available. If possible, the client should not do this in a “tight loop” to avoid heavily loading the processor while waiting for data. IMPORTANT: Client applications must retrieve the data blocks at a fast enough rate to avoid backing up a large amount of data within the API, which can result in loss of data. The minimum retrieval rate can be calculated as (srSps /maxSize). For example, with a sample rate of 56 Msps (40 MHz Acq BW) and IQ block maxSize of 130,848 samples (default), blocks must be retrieved at an average rate of no less than 56e6/130848 = 428 blocks/sec, or less than 2.34 msecs/block. The interval can be increased by requesting larger blocks sizes, or decreased if desired.

Table 10: IQ Streaming functions (cont.)

The API has an internal buffer which can hold up to 100 msec of output IQ samples at 40 MHz, to allow the client to occasionally take longer than the average required output rate. But if the client output retrieval rate continually averages less than the required rate, the buffer will eventually overflow and data will be lost. The same output buffer is used for all output sample rates so the buffer's effective time-size increases for lower sample rates (2x for 20 MHz, 4x for 10 MHz, etc).

iqinfo returns a copy of an IQSTRMIQINFO structure with the following content:

Item	Description
timestamp	Timestamp of first sample of block
triggerCount	Number of trigger events occurring during block. Maximum of 100 trigger events per block.
triggerIndices	List of sample indices where trigger(s) occurred, triggerCount in length. This list is stored in an internal static buffer and is overwritten on each call to IQSTREAM_GetIQData(). To preserve it longer, the client must copy the values to an external buffer before the next call.
scaleFactor	Scale factor to convert Int16 or Int32 data types to standard voltage values. This value is set to 1.0 for Single data type since those values are already scaled to voltage.
acqStatus	Acquisition status flags for this block and entire run interval. Individual bits are used as indicators as follows: Individual Retrieved Block status (Bits0-15, starting from LSB): Bit0: 1=Input overrange Bit1: (unused, always 0) Bit2: 1=Input buffer>75% full (IQStream processing heavily loaded) Bit3: 1=Input buffer overflow (IQStream processing overloaded, data loss has occurred) Bit4: 1=Output buffer>75% full (Client falling behind unloading data) Bit5: 1=Output buffer overflow (Client unloading data too slow, data loss has occurred) Bit6-Bit15: (unused, always 0) Entire run summary status ("sticky bits") The bits in this range are essentially the same as Bits0-15, except once they are set (->1) they remain set for the entire run interval. They can be used to determine if any of the status events occurred at any time during the run. (Bits16-31, starting from LSB): Bit16: 1=Input overrange Bit17: (unused, always 0) Bit18: 1=Input buffer>75% full (IQStream processing heavily loaded) Bit19: 1=Input buffer overflow (IQStream processing overloaded, data loss has occurred) Bit20: 1=Output buffer>75% full (Client falling behind unloading data) Bit21: 1=Output buffer overflow (Client unloading data too slow, data loss has occurred) Bit22-Bit31: (unused, always 0)

IQSTREAM_ClearAcqStatus() can be called to clear the "sticky" bits during the run if it is desired to reset them.

Table 10: IQ Streaming functions (cont.)

IQSTREAM_GetDiskFileWriteStatus()	Allows monitoring the progress of file output.
Declaration:	ReturnStatus IQSTREAM_GetDiskFileWriteStatus(bool* isComplete, bool *isWriting);
Parameters:	
isComplete:	Ptr-to-bool. Returns whether the IQ Stream file output writing complete.
isWriting:	Ptr-to-bool. Returns whether the IQ Stream processing has started writing to file (useful when triggering is in use). (Input NULL if no return value is desired).
Return Values:	
noError:	The query was successful.
Additional Detail:	The returned values indicate when the file output has started and completed. These become valid after IQSTREAM_Start() is invoked, with any file output destination selected. isComplete: false: indicates that file output is not complete. true: indicates file output is complete. isWriting: false: indicates file writing is not in progress. true: indicates file writing is in progress. When untriggered, this value is true immediately after Start() is invoked. For untriggered configuration, isComplete is all that needs to be monitored. When it switches from false->true, file output has completed. Note that if “infinite” file length is selected (file length parameter msec=0), isComplete only changes to true when the run is stopped with IQSTREAM_Stop(). If triggering is used, isWriting can be used to determine when a trigger has been received. The client application can monitor isWriting, and if a maximum wait period has elapsed while it is still false, the output operation can be aborted. isWriting behaves the same for both finite and infinite file length settings. The indicators sequence is as follows (assumes a finite file length setting): Untriggered operation: IQSTREAM_Start() => File output in progress: [isComplete =false, isWriting =true] => File output complete: [isComplete =true, isWriting =true] Triggered operation: IQSTREAM_Start() => Waiting for trigger, File writing not started: [isComplete=false, isWriting =false] => Trigger event detected, File writing in progress: [isComplete=false, isWriting =true] => File output complete: [isComplete=true, isWriting =true]

Table 10: IQ Streaming functions (cont.)

IQSTREAM_GetFileInfo() — Returns an information structure about the previous file output operation.

Declaration: ReturnStatus IQSTREAM_GetFileInfo(IQSTRMFILEINFO* fileinfo);

Parameters:

fileinfo: Ptr-to-struct. Returns a structure of information about the file output operation.

Return Values:

noError: The query was successful.

Additional Detail: This information is intended to be queried after the file output operation has completed. It can be queried during file writing as an ongoing status, but some of the results may not be valid at that time.

IQSTRMFILEINFO structure content:

Item	Description
numberSamples	Number of IQ sample pairs written to the file.
sample0Timestamp	Timestamp of the first sample written to file.
triggerSampleIndex	Sample index where the trigger event occurred. This is only valid if triggering has been enabled. Set to 0 otherwise.
triggerTimestamp	Timestamp of the trigger event. This is only valid if triggering has been enabled. Set to 0 otherwise.
filenames	Ptrs-to-wchar_t strings of the filenames of the output files: filenames[IQSTRM_FILENAME_DATA_IDX]: data filename filename[IQSTRM_FILENAME_HEADER_IDX] : header filename If data and header output are in the same file, the strings will be identical. The string storage is in an internal static buffer, overwritten with each call to the function.

Table 10: IQ Streaming functions (cont.)

acqStatus	Acquisition status flags for the run interval. Individual bits are used as indicators as follows: NOTE: *Bits0-15 indicate status for each internal write block, so may not be very useful. Bits 16-31 indicate the entire run status up to the time of query.* <u>Individual Internal Write Block status</u> (Bits0-15, starting from LSB): Bit0: 1=Input overrange Bit1: (unused, always 0) Bit2: 1=Input buffer>75% full (IQStream processing heavily loaded) Bit3: 1=Input buffer overflow (IQStream processing overloaded, data loss has occurred) Bit4: 1=Output buffer>75% full (File output falling behind writing data) Bit5: 1=Output buffer overflow (File output too slow, data loss has occurred) Bit6-Bit15: (unused, always 0) <u>Entire run summary status ("sticky bits")</u> The bits in this range are essentially the same as Bits0-15, except once they are set (->1) they remain set for the entire run interval. They can be used to determine if any of the status events occurred at any time during the run. (Bits16-31, starting from LSB): Bit16: 1=Input overrange Bit17: (unused, always 0) Bit18: 1=Input buffer>75% full (IQStream processing heavily loaded) Bit19: 1=Input buffer overflow (IQStream processing overloaded, data loss has occurred) Bit20: 1=Output buffer>75% full (File writing falling behind data generation) Bit21: 1=Output buffer overflow (File writing too slow, data loss has occurred) Bit22-Bit31: (unused, always 0)

IQSTREAM_ClearAcqStatus() can be called to clear the "sticky" bits during the run if it is desired to reset them.

NOTE. *If acqStatus indicators show "Output buffer overflow", it is likely that the disk is too slow to keep up with writing the data generated by IQ Stream processing. Use a faster disk (SSD recommended), or a smaller Acq BW which generates data at a lower rate.*

IQSTREAM_ClearAcqStatus()

Resets the "sticky" status bits of the acqStatus info element during an IQ Streaming run interval.

Declaration:

void IQSTREAM_ClearAcqStatus();

Parameters:

none:

Return Values:

none:

Additional Detail:

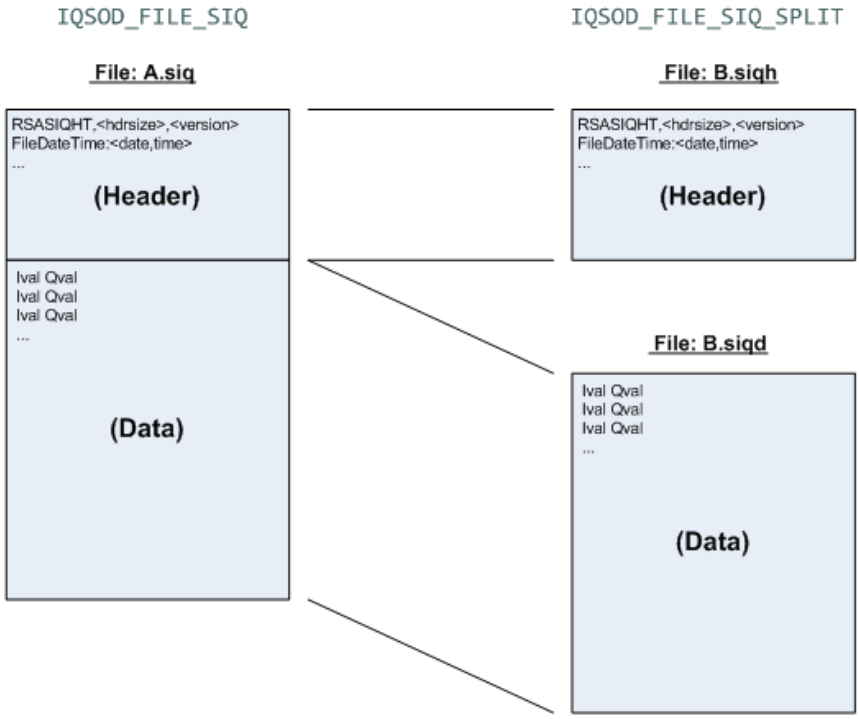
This is affective for both client and file destination runs.

IQ Streaming SIQ/SIQH/SIQD File Formats

IQ Streaming file output to can be configured as IQSOD_FILE_SIQ or IQSOD_FILE_SIQ_SPLIT using the IQSTREAM_SetOutputConfiguration() function dest (destination) parameter. This section describes the SIQ/SIQH/SIQD output files' content and format.

If IQSOD_FILE_SIQ format is selected, a single file with extension **.siq** is generated, containing both header information and sample data. If IQSOD_FILE_SIQ_SPLIT is selected, two files are generated: a text file containing the header information, with extension **.siqh**; and binary data file with the sample data content, with extension **.siqd**.

The header information format is the same in both **.siq** and **.siqh** file. Likewise, the data content format is the same in the **.siq** and **.siqd** files. The choice of combined or split files is a user preference, and does not affect the actual file content. When split files are selected, the filename portion of both files, excluding the extension, will be identical.



Header Block. The Header consists of lines of 8-bit ASCII text characters, each line terminated by a LF/CR (0x0D/0x0A) control character pair.

Example Header Block:

```
RSASIQHT:1024,1
FileDateTime:2015-04-29T10:12:33.170
Hardware:RSA306-Q000004
Software/Firmware:3.6.0034-V1.7-V1.1-V3
ReferenceLevel:0.00
CenterFrequency:100000000.00
SampleRate:56000000.00
AcqBandwidth:40000000.00
NumberSamples:56000
NumberFormat:IQ-Int16
```

DataScale:6.2660977E-005
DataEndian:Little
RecordUtcSec:001430327553.177054669
RecordUtcTime:2015-04-29T17:12:33.177054669
RecordLclTime:2015-04-29T10:12:33.177054669
TriggerIndex:0
TriggerUtcSec:001430327553.177054669
TriggerUtcTime:2015-04-29T17:12:33.177054669
TriggerLclTime:2015-04-29T10:12:33.177054669
AcqStatus:0x00000000

Header Identifier. The Header Identifier is always the first line of the header block. It is the only fixed location item in the header section. In addition to the fixed Header identifier string (RSASIQHT), it also contains the header size and version.

(Line1): RSASIQHT<:headerSizeInBytes>,<versionNumber>

Example: Header size: 1024 bytes, Version: 1

RSASIQHT:1024,1

In combined .siq files, the headerSizeInBytes value indicates the starting location (in bytes from the beginning of the file) of the Data section. This value should always be read and used as an index to the Data, as it may vary from file to file. Not all of the header may be needed for header content. Unused header range is filled with space characters (0x20) from the last piece of useful header data to the end of the header itself. In .siqd files, data always starts with the first byte, so the header size value should be ignored then.

The versionNumber is used to indicate different header content formats. Initially there is only one header format, version number = 1. However, it may change in future SW releases, so should be verified when decoding header information.

Header Information. Following the Header Identifier are lines with parameters describing the associated Data block values.

Each line has the format:

<InfoIDstring>:<InfoValueString>

The Header Information entries may be in any order. The table below describes the Header information content.

Table 11: IQ Streaming header content

Header Info Item:	Header Info Value:	Example:
FileDateTime:<fileDateTime>* *<fileDateTime> value only indicates the time the file was created. It is not an accurate timestamp of the data stored in the file.	<fileDateTime>: File creation date and time. Format: YYYY-MM-DDThh-hh-ss.msec	FileDateTime:2015-04-29T10:12:33.170
Hardware:<InstrNom>-<SerNum>	<InstrNom>: Instrument Nomenclature <SerNum>: Instrument Serial number	Hardware:RSA306-B010114
Software/Firmware:<Versions>	<Versions>: (API_SW)-(USB_FW)-(FPGA_FW)-(BoardID)	Software/Firmware:3.6.0034-V1.7-V1.1-V3
ReferenceLevel:<RefLeveldBm>	<RefLeveldBm>: Instrument Reference Level setting in dBm	ReferenceLevel:0.00

Table 11: IQ Streaming header content (cont.)

Header Info Item:	Header Info Value:	Example:
CenterFrequency:<CFinHz>	<CFinHz> Instrument Center Frequency setting in Hertz	CenterFrequency:100000000.00
SampleRate:<SRinSamples/sec>	<SRinSamples/sec>: Data sample rate in samples/second	SampleRate:56000000.00
AcqBandwidth:<BWinHz>	<BWinHz>: Acquisition (flat) Bandwidth of Data in Hertz, centered at 0 Hz (IQ baseband)	AcqBandwidth:40000000.00
NumberSamples:<numSamples>	<numSamples>: Number of IQ sample pairs stored in Data block	NumberSamples:56000
NumberFormat:<format>	<format>: Data block sample data format: IQ-Single: IQ pairs, each in one Single precision float (4 bytes per I or Q value) IQ-Int32: IQ pairs, each in one 32-bit integer (4 bytes per I or Q value) IQ-Int16: IQ pairs, each in one 16-bit integer (2 bytes per I or Q value)	NumberFormat:IQ-Int16
DataScale:<scaleFactor>	<scaleFactor>: Scale factor to convert In32 or Int16 I and Q values into “volts into 50 ohms”	DataScale:6.2660977E-005
DataEndian:<endian>	<endian>: Indicates Data block values stored in Little or Big Endian order	DataEndian:Little
RecordUtcSec:<recordUtcSec>	<recordUtcSec>: UTC Timestamp of first IQ sample in Data block record. Format: seconds.nanoseconds since Midnite, Jan 1, 1970 (UTC time).	RecordUtc-Sec:001430327553.177054669
RecordUtcTime:<recordUtcTime>	<recordUtcTime>: UTC Timestamp of first IQ sample in Data block record. Format: YYYY-MM-DDThh:mm:ss.nanoseconds (UTC time).	RecordUtcTime:2015-04-29T17:12:33.177054669
RecordLclTime:<recordLclTime>	<recordLclTime>: Local Timestamp of first IQ sample in Data block record. Format: YYYY-MM-DDThh:mm:ss.nanoseconds (Local time).	RecordLclTime:2015-04-29T17:12:33.177054669

Table 11: IQ Streaming header content (cont.)

Header Info Item:	Header Info Value:	Example:
TriggerIndex:<sampleIndex>	<sampleIndex>: IQ Sample index in Data block where trigger event occurred. If triggering is not enabled, sampleIndx is set to 0 (first sample of record).	TriggerIndex:21733
TriggerUtcSec:<triggerUtcSec>	<triggerUtcSec>: UTC Timestamp of trigger event. Format: seconds.nanoseconds since Midnite, Jan 1, 1970 (UTC time). If triggering is not enabled, this value is equal to RecordUtcSec.	TriggerUtc-Sec:001430327553.177054669
TriggerUtcTime:<triggerUtcTime>	<triggerUtcTime>: UTC Timestamp of trigger event. Format: YYYY-MM-DDThh:mm:ss.nanoseconds (UTC time). If triggering is not enabled, this value is equal to RecordUtcTime.	TriggerUtcTime:2015-04-29T17:12:33.177054669
TriggerLclTime:<triggerLclTime>	<triggerLclTime>: Local Timestamp of trigger event. Format: YYYY-MM-DDThh:mm:ss.nanoseconds (Local time). If triggering is not enabled, this value is equal to RecordLclTime.	TriggerLclTime:2015-04-29T17:12:33.177054669
AcqStatus:<acqStatusWord>	<acqStatusWord>: Hexidecimal value of acquisition and file status. Individual bits in this word indicate various status types. For detailed description, see acqStatus item in the IQSTREAM_GetFileInfo() function description. A value of 0x00000000 indicates no problems during file acquisition and storage.	AcqStatus:0x00000000

Data Block. Data block format is the same for all SIQx file selections. It consists of IQ sample pairs in alternating I/Q order as shown here:

I(0) Q(0) I(1) Q(1) I(2) Q(2) …. I(N-2) Q(N-2) I(N-1) Q(N-1)

where N equals the NumberSamples parameter value.

Each IQ Sample pair forms a complex baseband time-domain sample, at the sample rate given by the header block SampleRate parameter.

Each I and Q value is represented by a binary number in the data format specified by the header block NumberFormat parameter (Single, Int32 or Int16), with “endian-ness” specified by the DataEndian parameter.

Int32 and Int16 I and Q samples values can be scaled to “volts into 50 ohms” form by multiplying each integer value by the header block DataScale parameter value. Single values are prescaled to the correct form, so do not need to be multiplied by the scale factor (it is set to 1.0 to indicate this).

Playback (R3F file format)

These functions pertain to the palyback of files recorded with the RSA306. The RSA306 can record using two data structures, formatted or raw.

Recordings created using the formatted data structure create a single file (.r3f) that contain a single configuration info block, followed by a block of data and status information. The file contains the ADC output from the digitizer with enough metadata about the system state to reconstruct the IQ data stream.

Recordings created using the raw data structure create two files; a header file (.r3h) and a raw data file (.r3a).

The API can only play back files in the .r3f format.

Table 12: Playback functions

PLAYBACK_OpenDiskFile	Opens a .r3f file on disk and prepares the system for playback according to the parameters passed.
Declaration:	ReturnStatus PLAYBACK_OpenDiskFile(const wchar_t * fileName, int startPercentage, int stopPercentage, double skipTimeBetweenFullAcquisitions, bool loopAtEndOfFile, bool emulateRealTime);
Parameters:	
filename:	The Unicode name of an accessible disk file in .r3f format. The file must exist and you must have read permission to its contents.
startPercentage:	The starting location in the file from which to commence playback. Units are in percent of the total file length. File playback will skip the portion of the file prior to Start Position whenever it plays the file from the beginning, including repeatedly skipping that portion of the file if loop mode is enabled. Minimum allowed value: 0 Maximum allowed value: 99 Units: percentage
stopPercentage:	The stopping location in the file at which playback terminates. Units are in percent of total file length. File playback will skip the portion of the file after Stop Position to the end of the file, including skipping it every time the file plays if loop mode is enabled. Minimum allowed value: 1 Maximum allowed value: 100 Units: percentage
skipTimeBetweenFullAcquisitions:	The amount of time to skip in the file in order to accomplish fast-forwarding. The playback mechanism will play a contiguous slice of the file contents, the size of which is determined by the needs of the active measurements. Once that slice has been processed, file playback will skip a section of data roughly corresponding to Skip time, then start processing a new slice. Please note that skip time is not completely arbitrary – it is rounded up and discretized to the nearest USB data frame boundary, approximately 73 µs. Minimum allowed value: 0 (implies no portion of the file is skipped) Maximum allowed value: undefined, determined by the actual length of the input file. Units: time in seconds, rounded up to the nearest ~73 µs unit.

Table 12: Playback functions (cont.)

loopAtEndOfFile:	Controls if the file playback automatically wraps around to the start position when the stop position is reached during playback. Allowed values: true (loop at end of file) loops the file indefinitely until a stop request is received. false (do not loop and end of file) terminates playback when the stop position (or end of file) is reached.
emulateRealTime:	This setting, when true, puts the system in a real time emulation mode. Data is processed in a fashion indistinguishable from a live connection to an RSA device. A 60 second recording will take ~60 seconds to replay, and there is no guarantee that every frame of data is processed by the system. This mode is particularly useful for replaying files that contain audio data that you wish to hear. When set to false, the system uses a deterministic playback method that processes every frame of data. Deterministic playback is significantly more time consuming and should only be used for analyzing small significant portions of a file. Be aware that real time emulation mode is dependent on sufficient hardware processing power in order to read the data at the full necessary data rate (an SSD drive is typically necessary) and for the data processing demands of the streamed playback data. Allowed values: true for emulating real time playback, false for deterministic playback.
Return Values:	
noError:	The file successfully opened for playback.
errorStreamedFileOpenFailure:	The file could not be opened. Check the file for existence, access permissions, non-zero length, or other issues which might interfere with its use.
errorStreamedFileInvalidHeader:	The metadata stored in the file by the API appears to be corrupt. This data is necessary for playback to match the circumstances under which it was captured.
errorStreamingInvalidParameters:	One of the parameters passed to the function was out of range. Verify the ranges and types of parameters.
Additional Detail:	Once playback has commenced (via a call to Run() or equivalent), the system behaves much as it would when connected to actual hardware.

Table 12: Playback functions (cont.)

PLAYBACK_HasReplayCompleted	Determine if a file being replayed has reached the end of the file contents.
Declaration:	ReturnStatus PLAYBACK_HasReplayCompleted(bool * isCompleted);
Parameters:	
isCompleted:	a pointer to a Boolean which will return true if file playback has completed or false otherwise. Note that in loop back mode, a file will never report true from a call to isCompleted().
Return Values:	
noError:	The operation completed successfully.
PrepareForRun	Performs all of the internal tasks necessary to put the system in a known state ready to stream data, but does not actually initiate data transfer.
Declaration:	ReturnStatus PrepareForRun();
Parameters:	
None	
Return Values:	
noError:	The system is ready to start streaming data.
Additional Detail:	During file playback mode, this is useful to allow other parts of your application to prepare to receive data before starting the transfer. (See StartFrameTransfer). This is in comparison to the Run() function, which immediately starts data streaming without waiting for a Go signal.
StartFrameTransfer	Starts data transfer.
Declaration:	ReturnStatus StartFrameTransfer();
Parameters:	
None	
Return Values:	
noError:	System transfer has started.
errorTransfer:	Data transfer could not be initiated.
Additional Detail:	This is typically used as the trigger to start data streaming after a call to PrepareForRun(). If the system is in the stopped state, this call will place it back into the run state with no changes to any internal data or settings, and data streaming will begin assuming the system is not experiencing an error.

Trigger

Table 13: Trigger functions

ForceTrigger	Forces the device to trigger.
Declaration:	ReturnStatus ForceTrigger();
Return Values:	
noError:	The operation completed successfully.
Additional Detail:	When a device is triggered, its data before and after the trigger are used to update the signal. The amount of data used before and after the trigger is determined by the trigger position value.
SetTriggerPositionPercent	Sets the trigger position percentage.
Declaration:	ReturnStatus SetTriggerPositionPercent(double trigPosPercent);
Parameters:	
trigPosPercent:	Trigger position percentage. Range: 1% to 99%.
Return Values:	
noError:	The trigger position percent has been set.
errorNotConnected:	The device is not connected.
Additional Detail:	This value determines how much data to store before and after a trigger event. The stored data is used to update the signal's image when a trigger occurs.
GetTriggerPositionPercent	Queries the trigger position percent.
Declaration:	ReturnStatus GetTriggerPositionPercent(double* trigPosPercent);
Parameters:	
trigPosPercent:	Pointer to a double. Contains the trigger position percent value when the function completes.
Return Values:	
noError:	The trigger position percent has been queried.
errorNotConnected:	The device is not connected.
Additional Detail:	The value is stored in the trigPosPercent parameter.
SetTriggerMode	Sets the trigger mode.
Declaration:	ReturnStatus SetTriggerMode(TriggerMode mode);
Parameters:	
mode:	This variable describes the trigger mode. It can be in either freeRun or triggered mode.

TriggerMode	Value
freeRun	0
triggered	1

Return Values:	
noError:	The trigger mode has been set.
errorNotConnected:	The device is not connected.
Additional Detail:	When the device is in freeRun, it continually gathers data. When the device is in triggered mode, it will not acquire new data unless it is triggered.

Table 13: Trigger functions (cont.)

GetTriggerMode	Queries the trigger mode.
Declaration:	ReturnStatus GetTriggerMode(TriggerMode* mode);
Parameters:	
mode:	Pointer to a TriggerMode type. Contains a trigger mode value when the function completes. The mode value can be freeRun or triggered.
Return Values:	
noError:	The trigger mode has been set.
errorNotConnected:	The device is not connected.
Additional Detail:	The value is stored in the mode parameter.
	When the trigger mode is set to freeRun, the signal is continually updated. When the trigger mode is set to triggered, the data is only updated when a trigger occurs.
SetTriggerTransition	Sets the trigger transition detection.
Declaration:	ReturnStatus SetTriggerTransition(TriggerTransition transition);
Parameters:	
transition:	A TriggerTransition type. It can be set to TriggerTransitionLH, TriggerTransitionHL, or TriggerTransitionEither.
Return Values:	
noError:	The trigger transition mode has been set.
errorNotConnected:	The device is not connected.
Additional Detail:	When the trigger transition is set to low-to-high, the trigger occurs when the signal changes from a low input level to a high input level. Likewise for high-to-low mode. The transition type can also be set to trigger on either low-to-high or high-to-low transitions.
GetTriggerTransition	Queries the current trigger transition mode.
Declaration:	ReturnStatus GetTriggerTransition(TriggerTransition *transition);
Parameters:	
transition:	A pointer to a TriggerTransition type. Contains a trigger transition mode value when the function completes. The mode value can be TriggerTransitionLH, TriggerTransitionHL, or TriggerTransitionEither.
Return Values:	
noError:	The trigger transition mode has been queried.
errorNotConnected:	The device is not connected.
Additional Detail:	When the trigger transition is set to low-to-high, the trigger occurs when the signal changes from a low input level to a high input level. Likewise for high-to-low mode. The transition type can also be set to trigger on either low-to-high or high-to-low transitions.

Table 13: Trigger functions (cont.)

SetTriggerSource	Sets the trigger source.
Declaration:	ReturnStatus SetTriggerSource(TriggerSource source);
Parameters:	
source:	A TriggerSource type. It can be set to TriggerSourceExternal or TriggerSourceIFPowerLevel.
Return Values:	
noError:	The trigger source has been set.
errorNotConnected:	The device is not connected.
Additional Detail:	When the trigger source is set to external, acquisition triggering looks at the external trigger input for a trigger signal. When the trigger mode is set to IF power level, the power of the signal itself causes a trigger to occur.
GetTriggerSource	Queries the trigger mode.
Declaration:	ReturnStatus GetTriggerSource(TriggerSource *source);
Parameters:	
source:	Pointer to a TriggerSource type. Contains a trigger source value when the function completes. The source value can be TriggerSourceExternal or TriggerSourceIFPowerLevel.
Return Values:	
noError:	The trigger source has been queried.
errorNotConnected:	The device is not connected.
Additional Detail:	The value is stored in the source parameter. When the trigger source is set to external, acquisition triggering looks at the external trigger input for a trigger signal. When the trigger mode is set to IF power level, the power of the signal itself causes a trigger to occur.
SetIFPowerTriggerLevel	Sets the IF power detection level.
Declaration:	ReturnStatus SetIFPowerTriggerLevel(double level);
Parameters:	
level:	A double type. This parameter sets the detection power level for the IF power trigger source.
Return Values:	
noError:	The trigger level has been set.
errorNotConnected:	The device is not connected.
Additional Detail:	When set to the IF power level trigger source, a trigger occurs when the signal power level crosses this detection level.

Table 13: Trigger functions (cont.)

GetIFPowerTriggerLevel	Queries the trigger mode.
Declaration:	ReturnStatus GetIFPowerTriggerLevel(double *level);
Parameters:	
level:	A double type. This parameter contains the detection power level for the IF power trigger source.
Return Values:	
noError:	The trigger mode has been queried.
errorNotConnected:	The device is not connected.
Additional Detail:	The value is stored in the mode parameter. When the trigger mode is set to freeRun, the signal is continually updated. When the trigger mode is set to triggered, the data is only updated when a trigger occurs.

Self test

Table 14: Self test functions

POST_QueryStatus	Queries the device for the last POST status.
Declaration:	ReturnStatus POST_QueryStatus ();
Return Values:	
noError:	The alignment has succeeded.
errorPOSTFailureFPGALoad:	The FPGA failed to load.
errorPOSTFailureHiPower:	USB Power is insufficient.
errorPOSTFailureI2C:	The I2C bus has failed.
errorPOSTFailureGPIF:	The GPIF bus has failed.
errorPOSTFailureUsbSpeed:	The device has been connected to a incompatible USB port, such as USB 2.0.

Spectrum

Table 15: Spectrum functions

SPECTRUM_SetEnable	Sets the enable status.
Declaration:	ReturnStatus SPECTRUM_SetEnable(bool enable);
Parameters:	
enable:	Enable or disable Spectrum measurement. This value can be either true or false.
Return Values:	
noError:	The function has completed successfully.
Additional Detail:	If the enable parameter is true, the spectrum measurement is enabled. If the enable parameter is false, the spectrum is disabled. When the spectrum measurement is enabled, the IQ acquisition is disabled.
SPECTRUM_GetEnable	Queries the enable status.
Declaration:	ReturnStatus SPECTRUM_GetEnable (bool* enable);
Parameters:	
enable:	Pointer to a bool. Contains the enable status of the spectrum.
Return Values:	
noError:	The enable status has been successfully queried.
Additional Detail:	If the value of enable is true, the spectrum measurement is enabled. If the value of enable is false, the spectrum measurement is disabled.
SPECTRUM_SetDefault	Sets the spectrum settings to default settings.
Declaration:	ReturnStatus SPECTRUM_SetDefault();
Parameters:	
none	
Return Values:	
noError:	The function has completed successfully.
Additional Detail:	This does not change the spectrum enable status. The following are the default settings: ■ Span: 40 MHz ■ RBW: 300 kHz ■ Enable VBW: false ■ VBW: 300 kHz ■ Trace Length: 801 ■ Window: Kaiser ■ Vertical Unit: dBm ■ Trace1: Enable, +Peak ■ Trace2: Disable, -Peak ■ Trace3: Disable, Average

Table 15: Spectrum functions (cont.)

SPECTRUM_SetSettings — Sets the spectrum settings.

Declaration: ReturnStatus SPECTRUM_SetSettings(Spectrum_Settings settings);

Parameters:

settings: Spectrum settings.

Spectrum_Settings
double span
double rbw
bool enableVBW
double vbw
int traceLength
SpectrumWindows window
SpectrumVerticalUnits verticalUnit

SpectrumWindows	**Value**
SpectrumWindow_Kaiser	0
SpectrumWindow_Mil6dB	1
SpectrumWindow_BlackmanHarris	2
SpectrumWindow_Rectangular	3
SpectrumWindow_FlatTop	4
SpectrumWindow_Hann	5
SpectrumVerticalUnits	
SpectrumVerticalUnit_dBm	0
SpectrumVerticalUnit_Watt	1
SpectrumVerticalUnit_Volt	2
SpectrumVerticalUnit_Amp	3
SpectrumVerticalUnit_dBmV	4

Return Values:

noError: The function has completed successfully.

errorNotConnected: The device is not connected.

Table 15: Spectrum functions (cont.)

SPECTRUM_GetSettings	Queries the spectrum settings.
Declaration:	ReturnStatus SPECTRUM_GetSettings(Spectrum_Settings *settings);
Parameters:	
settings:	Pointer to Spectrum settings. Returns the current settings.

Spectrum_Settings
double span
double rbw
bool enableVBW
double vbw
int traceLength
SpectrumWindows window
SpectrumVerticalUnits verticalUnit
double actualStartFreq
double actualStopFreq
double actualFreqStepSize
double actualRBW
double actualVBW
double actualNumIQSamples

Return Values:	
noError:	The function has completed successfully.
Additional Detail:	In addition to user settings, the Spectrum_Setting structure also returns some internal setting values.

Table 15: Spectrum functions (cont.)

SPECTRUM_SetTraceType — Sets the trace settings.

Declaration: ReturnStatus SPECTRUM_SetTraceType(SpectrumTraces trace, bool enable, SpectrumDetectors detector);

Parameters:

trace: One of the spectrum traces.

SpectrumTraces	Value
SpectrumTrace1	0
SpectrumTrace2	1
SpectrumTrace3	2

enable: Enable trace output. This value can be either true or false.

detector: Detector type.

SpectrumDetectors	Value
SpectrumDetector_PosPeak	0
SpectrumDetector_NegPeak	1
SpectrumDetector_AverageVRMS	2
SpectrumDetector_Sample	3

Return Values:

noError: The function has completed successfully.

errorNotConnected: The device is not connected.

SPECTRUM_GetTraceType — Queries the trace settings.

Declaration: ReturnStatus SPECTRUM_GetTraceType(SpectrumTraces trace, bool *enable, SpectrumDetectors *detector);

Parameters:

trace: One of the spectrum trace. See SPECTRUM_SetTraceType().

enable: Pointer to bool. It returns the enable status of the trace.

detector: Pointer to SpectrumDetectors. It returns the detector type of the trace. See SPECTRUM_SetTraceType().

Return Values:

noError: The function has completed successfully.

Table 15: Spectrum functions (cont.)

SPECTRUM_GetLimits — Queries the limits of the spectrum settings.

Declaration: ReturnStatus SPECTRUM_GetLimits(Spectrum_Limits *limits);

Parameters:

limits: Return the spectrum setting limits.

Spectrum_Limits	Description	64 bit API limit	32 bit API limit
double maxSpan	Maximum Span	6.2 GHz	6.2 GHz
double minSpan	Minimum Span	1 kHz	100 kHz
double maxRBW	Maximum RBW	10 MHz	10 MHz
double minRBW	Minimum RBW	10 Hz	100 Hz
double maxVBW	Maximum VBW	10 MHz	10 MHz
double minVBW	Minimum VBW	1 Hz	100 Hz
double maxTraceLength	Maximum Trace Length	64001	64001
double minTraceLength	Minimum Trace Length	801	801

Return Values:

noError: The limits have been successfully queried.

SPECTRUM_WaitForDataReady — Waits for the spectrum data to be ready to be queried.

Declaration: ReturnStatus SPECTRUM_WaitForDataReady(int timeoutMsec, bool *ready);

Parameters:

timeoutMsec: Timeout value in msec.

ready: Pointer to a bool. Its value indicates the status of the data.

Return Values:

noError: The data ready status has been successfully queried.

Additional Detail: If the data is not ready and the timeout value is exceeded, the ready parameter will be false. Otherwise, the spectrum data is ready and the ready parameter will be true.

Table 15: Spectrum functions (cont.)

SPECTRUM_GetTrace

This function queries the spectrum trace data.

Declaration:

ReturnStatus SPECTRUM_GetTrace(SpectrumTraces trace, int maxTracePoints, float *traceData, int *outTracePoints);

Parameters:

trace:

One of the spectrum trace.

SpectrumTraces	Value
SpectrumTrace1	0
SpectrumTrace2	1
SpectrumTrace3	2

maxTracePoints:

Maximum number of trace points to be retrieved. The traceData array should be at least this size.

traceData:

Return spectrum trace data.

outTracePoints:

Pointer to int. Returns the actual number of valid trace points in traceData array.

Return Values:

noError:

The trace data has been successfully queried.

SPECTRUM_GetTraceInfo

This function queries the spectrum result information.

Declaration:

ReturnStatus SPECTRUM_GetTraceInfo(Spectrum_TraceInfo *traceInfo);

Parameters:

traceInfo:

Return spectrum trace result information.

Spectrum_TraceInfo
uint64_t timestamp
uint16_t acqDataStatus

For timestamp, see REFTIME_GetTimeFromTimestamp() for converting from timestamp to time.

For acqDataStatus bits definition are:

AcqDataStatus	Value
adcOverrange	0x1
refFreqUnlock	0x2
lo1Unlock	0x4
lo2Unlock	0x8
lowSupplyVoltage	0x10
adcDataLost	0x20
event1pps	0x40
eventTrig1	0x80
eventTrig2	0x100

Return Values:

noError:

The trace information has been successfully queried.

System time reference

These functions support manipulation of data time and timestamp information based on the internal time/timestamp association. The internal time association is automatically initialized when the instrument is connected, and aligned to the current local time based on the Windows OS time function.

Table 16: System Time Reference functions

REFTIME_GetTimestampRate()	Returns value of the clock rate of the continuously running timestamp counter in the instrument.
Declaration:	ReturnStatus IQSTREAM_GetTimestampRate(uint64_t* refTimestampRate);
Parameters:	
refTimestampRate:	Ptr-to-uint64_t. Returns timestamp counter clock rate.
Return Values:	
noError:	The query was successful.
Additional Detail:	This function can be used for calculations on timestamp values.
REFTIME_GetTimeFromTimes-tamp()	The input timestamp value is converted to equivalent second and nanosecond component values, using the current internal reference time/timestamp association.
Declaration:	ReturnStatus IQSTREAM_GetTimeFromTimestamp(uint64_t i_timestamp, time_t* o_timeSec, uint64_t* o_timeNsec);
Parameters:	
i_timestamp:	Timestamp counter time to convert to time values.
o_timeSec:	Ptr-to-time_t. Returns time value seconds component.
o_timeNsec:	Ptr-to-uint64_t. Returns time value nanoseconds component.
Return Values:	
noError:	The query was successful.
Additional Detail:	The timeSec value is the number of seconds elapsed since midnight (00:00:00), Jan 1, 1970, UTC. The timeNsec value is the number of nanoseconds into the specified second.
REFTIME_GetTimestampFrom-Time ()	The input time specified by the second and nanosecond component values is converted to the equivalent timestamp value, using the current internal reference time/timestamp association.
Declaration:	ReturnStatus IQSTREAM_GetTimestampFromTime (time_t i_timeSec, uint64_t i_timeNsec, uint64_t* o_timestamp);
Parameters:	
i_timeSec:	Time-seconds component to convert to timestamp.
i_timeNsec:	Time-nanoseconds component to convert to timestamp.
o_timestamp:	Ptr-to-uint64_t. Returns equivalent timestamp value.
Return Values:	
noError:	The query was successful.
Additional Detail:	The timeSec value is the number of seconds elapsed since midnight (00:00:00), Jan 1, 1970, UTC. The timeNsec value is the number of nanoseconds into the specified second.

Table 16: System Time Reference functions (cont.)

REFTIME_GetCurrentTime ()	Returns the current RSA API system time (in second and nanoseconds components), and the corresponding current timestamp value.
Declaration:	ReturnStatus IQSTREAM_GetCurrentTime (time_t* o_timeSec, uint64_t* o_timeNsec, uint64_t* o_timestamp);
Parameters:	
o_timeSec:	Ptr-to-time_t. Returns seconds component of current time. (Input NULL argument value if return value not desired).
o_timeNsec:	Ptr-to-uint64_t. Returns nanoseconds component of current time. (Input NULL argument value if return value not desired).
o_timestamp:	Ptr-to-uint64_t. Returns timestamp of current time. (Input NULL argument value if return value not desired).
Return Values:	
noError:	The query was successful.
Additional Detail:	The timeSec value is the number of seconds elapsed since midnight (00:00:00), Jan 1, 1970, UTC. The timeNsec value is the number of nanoseconds into the specified second. The time and timestamp values are accurately aligned with each other at the time of the function call.
REFTIME_GetIntervalSinceRef-TimeSet ()	Returns the number of seconds that have elapsed since the internal RSA API time and timestamp association was set.
Declaration:	ReturnStatus QSTREAM_GetIntervalSinceRefTimeSet (double* sec);
Parameters:	
sec:	Ptr-to-double. Returns seconds since the internal Reference time/timestamp association was last set.
Return Values:	
noError:	The query was successful.

Example Python program

The example program provided (as an attachment to this PDF document) sets up the basic acquisition parameters, and then shows Spectrum and raw IQ vs Time displays. It allows you to change several parameters on the fly, like Ref Level, IQBandwidth, and Center Frequency. It also allows you to enable external triggering.

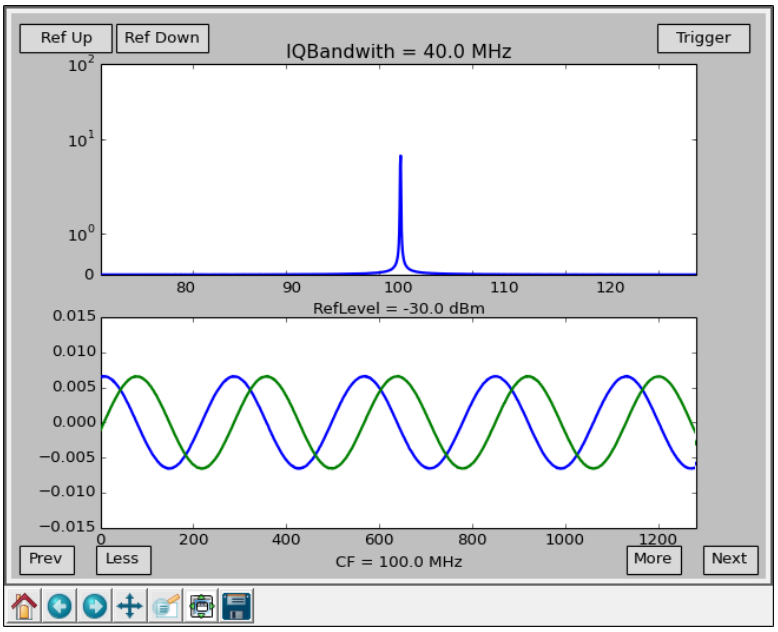
The program was written with Python 2.7. To use this example, the NumPy, Matplotlib, Dateutil, and Pyparsing libraries need to be installed along with Python 2.7.

These are the basics steps, in order, the example program accomplishes:

- Import necessary Python plotting and processing libraries
- Import the RSA300API DLL
- Search for, and connect to the device
- Set IQ Record Length
- Set CF
- Set Ref Level
- Set Trigger Position
- Set IQ Bandwidth
- Define function for getting IQ Data from the device
  - Set the device to Run
  - Wait for IQ Data to be ready
  - Get IQ Data
  - Process IQ Data into spectrum
  - Return IQ and spectrum data
- Define functions for updating the plots
- Initialize plots
- Define functions for all of the buttons
- Initialize buttons
- Start animating plots and display them to the screen

When the program exits, Stop and Disconnect from the device

Following is a picture of the program when it is running. Ref Up and Ref Down step the Ref Level up and down. Prev and Next change the CF by 10 MHz, and More and Less adjust the IQBandwidth. Trigger enables external triggering.



File attachment

The ***Python Programming Example.txt*** attachment is an actual program file created with Python 2.7. The Python file extension (.py) was replaced with the text extension (.txt) to enable easier access to the file. If you save or copy the file, you can replace the file extension with the Python (.py) extension.

NOTE. *Typically, Adobe Acrobat uses a paper clip icon to display attachments. Other PDF file viewers may use other indicators for attachments. If needed, refer to the PDF viewer's documentation.*

RSA306 Streaming Sample Data File Format

Streaming Data Files

Streaming ADC data can be stored to disk file in two file formats.

- Formatted file type combines ADC samples with auxiliary information (configuration and USB data transport framing) in the same file.
- Raw file type places the ADC samples and auxiliary information into separate files. The ADC data file contains only the raw ADC data, the non-data framing portions of the USB data transport stream are not stored.

In both file storage formats, ADC samples are stored in the same basic format:

- 16-bit signed integers in 2 bytes
- Unscaled for signal path gain, and uncorrected for internal IF signal path channel amplitude and phase deviations

Filename Extensions

Formatted files use a file extension of ".r3f".

Raw files use a file extension of ".r3a" for the raw ADC sample data files, and ".r3h" for the configuration ("header") info files.

Formatted File Content

Formatted files (extension: .r3f) contain a single Configuration info block, followed by a blocks of data and status information. Each data block is called a frame. A frame is 16384 bytes (16kB) in size. Formatted files can only contain complete frames, not partial ones. Figure 1 shows the structure of the formatted data file.

The Config info block applies to all sample data within the file. Its content is described further below.

Data frames contain ADC sample data, and transport stream footer data. The ADC data can be accessed directly by indexing past the Config block info to the first data frame. The 8178 16-bit ADC data samples from that frame can be extracted. Then the 28 byte footer is skipped over to reach the start of the next frame where the next 8178 data samples can be extracted. This is repeated until data from all frames in the file is extracted. The location and sizes of the frame contents are specified by descriptor values in the Config info block, allowing a configurable reader function to determine the file structure at the time it reads the file, rather than having the values hard-coded.

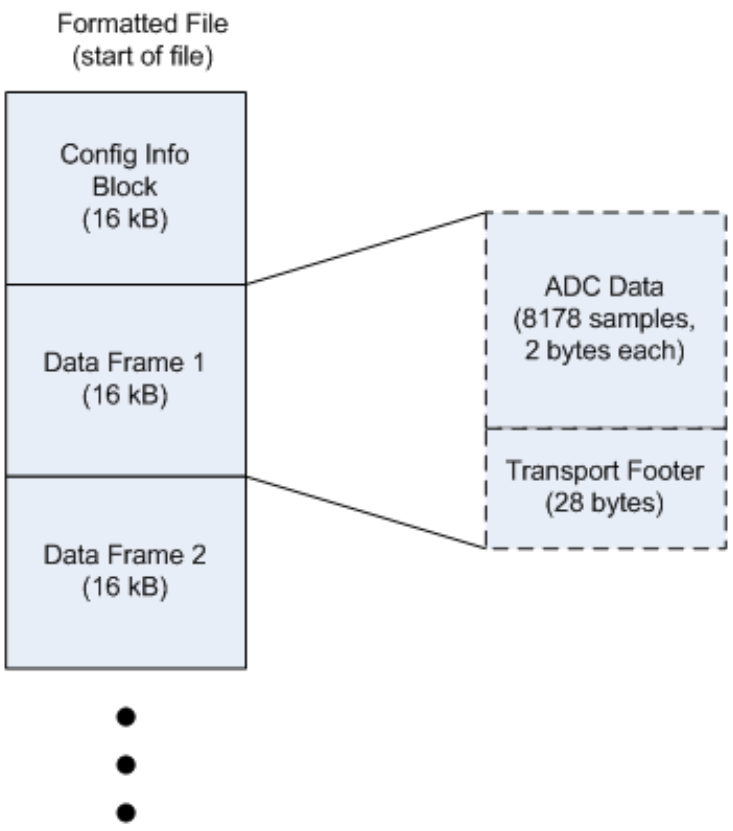


Figure 1: Formatted data file

Footers contain information about the samples in that frame. These include trigger indicators, frame counters and other synchronization information. Footer information can be ignored if only the raw ADC data is needed.

Raw File Content

Raw data files (extension: .r3a) contain only ADC samples. The samples are contiguous, with all transport frame information removed before storage. No knowledge other than the basic 16-bit/2 byte sample format is needed to read this data from the file.

The associated header file (extension: .r3h), if available, contains the Config data which can be used to interpret and scale the ADC data samples for further processing. This is the same file stored by a Formatted data file in the initial header block, except the data structure descriptor information is “zeroed” since there is only ADC data in the data file.

Configuration Information Block

The Configuration Information Block (AKA “header”) is a 16 kB (16384 bytes) block of non-sample data. The same header format is used for both Framed and Raw file storage formats. The header contains information about the acquisition settings and HW configuration used to set up the data. It also contains data to use for gain scaling and IF channel frequency response correction.

In Framed file format, the header block is inserted at the beginning of the file, before the sample data content, which also contains the USB transport framing. In Raw format, the entire header block is contained in a separate file from the sample data.

Data in the header is encoded as either ASCII character strings or binary data, in fixed location fields. This is so that users can access each item by indexing to the fixed location rather than requiring a parser like XML to interpret it.

The File Format value indicates the overall revision level of the file format.

NOTE. *All strings are "null-terminated" (0x00 byte following the final string character). If in a fixed length field, the unused portion of the field is filled with 0x00 byte values.*

"EOB" means "End-of-Block".

Table 17: (Category) specifications

Offset (Byte)	Size (Bytes)	Content	Description
File ID: (512 bytes)			
0	27	File ID String	Fixed String: "Tektronix RSA300 Data File"
	(to EOB)	Reserved	(filled with 0x00)
Version Info: (512 bytes)			
512	4	Endian-check	0x12345678 (int32)
	4	File Format Version	V.V.V.V (V=1 byte, 1.0.0.0 initially)
	4	API SW Version	V.V.V.V (V=1 byte, fill unused with 0)
	4	FX3 FW Version	V.V.V.V (V=1 byte, fill unused with 0)
	4	FPGA FW Version	V.V.V.V (V=1 byte, fill unused with 0)
	64	Device S/N	Serial Number String (fill with 0x00 to end)
	(to EOB)	Reserved	(filled with 0x00)
Instrument State: (1k bytes)			
1024	8	Reference Level	dBm (double)
	8	RF Center Frequency	Hz (double)
	8	Device temperature	Deg C (double)
	4	Alignment State	0=Not Aligned, 1=Aligned
	4	Freq Ref State	0=Internal, 1=External
	4	Trigger Mode	0=FreeRun, 1=Triggered
	4	Trigger Source	0=External, 1=Power
	4	Trigger Transition	1=Rising Edge,2=Falling Edge
	8	Trigger Level	dBm (double)
	(to EOB)	Reserved	(filled with 0x00)

Table 17: (Category) specifications (cont.)

Offset (Byte)	Size (Bytes)	Content	Description
Data Format: (1k bytes)			
2048	4	File Data Type	161 = 16 bit integer ADC samples
	6 * 4	File Data Structure Descriptor	(Note: These items describe the frame structure of the Formatted .r3f file with 16-bit ADC IF samples and transport framing; for others file formats, these items are filled with 0 values) All items are int32 types (4 bytes). Default values for initial framed ADC storage format are shown • Offset to first frame (bytes): 16384 • Size of frame (bytes): 16384 • Offset to sample data in frame (bytes): 0 • Number of samples in frame: 8178 • Offset to non-sample data in frame (bytes): 16356 • Size of non-sample data in frame (bytes): 28
	8	Center Frequency at Sampled Data IF	Hz (double) (ADC samples: 28 MHz + LO offset
	8	Sample Rate	Samples/sec (double) (ADC samples: 112e6
	8	Bandwidth	Usable Bandwidth (double) (ADC samples: 40e6
	4	File Data Corrected	0=uncorrected
	4	Ref Time - Wall Time Type	0=Local
	7 * 4	Ref Time - Wall Time	Ref Time: (7 values, each int32) Year, Month, Date, Hour, Minute, Second, Nanoseconds (Note: Nanoseconds is set to 0 initially)
	8	Ref Time - Sample Count	Ref Time: FPGA Sample Count (uint64)
	8	Ref Time - Sample Ticks Per Second	Ref Time: FPGA Sample counter ticks per second (uint64) (112,000,000)
	(to EOB)	Reserved	(filled with 0x00)
Signal Path: (1k bytes)			
3072	8	Sample Gain Scaling Factor	(Factor which scales the data (ADC or IQ) samples to “Volts terminated in 50 ohm” values.) Volts/ADC-levels (double) for ADC samples
	8	Signal Path delay	Seconds (double)
	(to EOB)	Reserved	(filled with 0x00)

Table 17: (Category) specifications (cont.)

Offset (Byte)	Size (Bytes)	Content	Description
Channel Correction: (8k bytes)			
4096	4	Channel Correction Type	0=LF, 1=IF
	252	Reserved	(fill with 0x00s)
	4	Number of Table Entries	Nt (int32, Nt(max) = 501)
	501 * 4	Frequency Table	Hz (float, first Nt points of table)
	501 * 4	Amplitude Table	dB (float, first Nt points of table)
	501 * 4	Phase Table	Degrees (float, first Nt points of table)
	(to EOB)	Reserved	(filled with 0x00)
Reserved: (4k bytes)			
12288	(to EOB)	Reserved	(filled with 0x00)

Index

A

ADC streaming functions, 2
Alignment functions, 4
Audio functions, 5
AUDIO_GetData, 5
AUDIO_GetMode, 6
AUDIO_GetMute, 6
AUDIO_GetVolume, 7
AUDIO_SetMode, 5
AUDIO_SetMute, 6
AUDIO_SetVolume, 7
AUDIO_StartAudio, 5
AUDIO_StopAudio, 5

C

Connect, 8
Connection functions, 8

D

Device operation functions, 9
Device status functions, 15
Disconnect, 8
DPX functions, 19
DPX_Configure, 20
DPX_FindRBWRange, 21
DPX_FinishFrameBuffer, 25
DPX_GetSettings, 21
DPX_GetSogramHiResLine, 24
DPX_GetSogramHiResLineCount-Latest, 23
DPX_GetSogramHiResLineTimes-tamp, 24
DPX_GetSogramHiResLineTrig-gered, 23
DPX_GetSogramSettings, 23
DPX_IsFrameBufferAvailable, 25
DPX_ResetDPx, 22
DPX_SetParameters, 20
DPX_SetSogramParameters, 22
DPX_SetSogramTraceType, 22
DPX_SetSpectrumTraceType, 21

F

ForceTrigger, 50

G

GetAPIVersion, 15
GetCenterFreq, 10
GetDeviceNomenclature, 15
GetDeviceSerialNumber, 15
GetDeviceTemperature, 4
GetDPXEnabled, 19
GetErrorString, 17
GetExternalRefEnable, 13
GetFirmwareVersion, 15
GetFPGAVersion, 16
GetFrameBuffer, 25
GetFrameInfo, 22
GetHWVersion, 16
GetIFPowerTriggerLevel, 53
GetIQBandwidth, 11
GetIQData, 29
GetIQDataCplx, 31
GetIQDataDeinterleaved, 30
GetIQHeader, 28
GetIQRecordLength, 12
GetIQSampleRate, 12
GetMaxAcquisitionSamples, 14
GetMaxCenterFreq, 11
GetMaxIQBandwidth, 13
GetMinIQBandwidth, 14
GetReferenceLevel, 10
GetRunState, 9
GetStreamADCToDiskActive, 3
GetTriggerMode, 51
GetTriggerPositionPercent, 50
GetTriggerSource, 52
GetTriggerTransition, 51
GetTunedCenterFreq, 11

I

IQ data functions, 28
IQ Streaming functions, 32
IQSTREAM_ClearAcqStatus(), 42
IQSTREAM_GetAcqParameters, 32
IQSTREAM_GetDiskFileWriteSta-tus(), 40
IQSTREAM_GetEnabled(), 38
IQSTREAM_GetFileInfo(), 41
IQSTREAM_GetIQData(), 38
IQSTREAM_GetIQDataBuffer-Size, 34
IQSTREAM_SetAcqBandwidth, 32
IQSTREAM_SetDiskFileLength, 37
IQSTREAM_SetDiskFilename-Base, 35
IQSTREAM_SetDiskFilenameSuf-fix, 36
IQSTREAM_SetIQDataBuffer-Size, 33
IQSTREAM_SetOutputConfigura-tion, 33
IQSTREAM_Start(), 37
IQSTREAM_Stop(), 37
IsAlignmentNeeded, 4

P

Playback functions, 47
PLAYBACK_HasReplay-Completed, 49
PLAYBACK_OpenDiskFile, 47
POST_QueryStatus, 54
PrepareForRun, 49
Preset, 9

R

REFTIME_GetCurrentTime (), 62
REFTIME_GetIntervalSinceRef-TimeSet (), 62
REFTIME_GetTimeFromTimes-tamp(), 61
REFTIME_GetTimestampFromTime (), 61
REFTIME_GetTimestampRate(), 61
ResetDevice, 8
Run, 9
RunAlignment, 4

S

Search, 8
Self test functions, 54
SetCenterFreq, 10
SetDPXEnabled, 19
SetExternalRefEnable, 13
SetIFPowerTriggerLevel, 52
SetIQBandwidth, 11
SetIQRecordLength, 12
SetReferenceLevel, 10

SetStreamADCToDiskEnabled, 2
SetStreamADCToDiskFilenameBase, 2
SetStreamADCToDiskMaxFileCount, 3
SetStreamADCToDiskMaxTime, 3
SetStreamADCToDiskMode, 3
SetStreamADCToDiskPath, 2
SetTriggerMode, 50
SetTriggerPositionPercent, 50
SetTriggerSource, 52
SetTriggerTransition, 51
Spectrum functions, 55
SPECTRUM_GetEnable, 55
SPECTRUM_GetLimits, 59
SPECTRUM_GetSettings, 57
SPECTRUM_GetTrace, 60
SPECTRUM_GetTraceInfo, 60
SPECTRUM_GetTraceType, 58
SPECTRUM_SetDefault, 55
SPECTRUM_SetEnable, 55
SPECTRUM_SetSettings, 56
SPECTRUM_SetTraceType, 58
SPECTRUM_WaitForDataReady, 59
StartFrameTransfer, 49
Stop, 9
System Time Reference functions, 61

T

Trigger functions, 50

W

WaitForDPXDataReady, 24
WaitForIQDataReady, 31