Tektronix®

RSA306, RSA306B, and RSA500A/600A Series Spectrum Analyzers Application Programming Interface (API)

Programming Reference

Revision B

077-1031-02

Tektronix®

RSA306, RSA306B, and RSA500A/600A Series Spectrum Analyzers Application Programming Interface (API)

Programming Reference

Revision B

www.tek.com

077-1031-02

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.tek.com to find contacts in your area.

Table of Contents

Preface

This document describes the RSA API function calls to interface with the RSA306, RSA306B, RSA500A Series, and RSA600A Series Spectrum Analyzers 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. Open the flash drive, and navigate to the API installer. The supplied driver is for the Microsoft Windows operating system.

Programing languages supported by this driver include: C, C++, and Python. An example program written in Python is provided. (See page 89, *Programming file attachment*.)

This document supports API version 2. A compatibility chart from API version 1 to version 2 is provided. (See page 95, *RSA API version compatibility*.)

API function groups

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

Alignment functions

ALIGN_GetAlignmentNeeded	Determines if an alignment is needed or not.
Declaration:	ReturnStatus ALIGN_GetAlignmentNeeded(bool* needed);
Parameters:	
needed:	Pointer to a bool. True indicates an alignment is needed. False indicates 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.

ALIGN_GetWarmupStatus	Reports device warm-up status.
Declaration:	ReturnStatus ALIGN_GetWarmupStatus(bool* warmedUp);
Parameters:	
warmedUp:	Pointer to a bool. True indicates the device's warm-up interval has been reached. False indicates the warm-up interval has not been reached.
Return Values:	
noError:	The function has completed successfully.
Additional Detail:	Devices start in the "warm-up" state after initial power up until the internal temperature stabilizes. The warm-up interval is different for different devices.

ALIGN_RunAlignment	Runs the device alignment process.
Declaration:	ReturnStatus ALIGN_RunAlignment();
Return Values:	
noError:	The alignment has succeeded.
errorDataNotReady:	The alignment operation failed.

Audio functions

AUDIO_SetFrequencyOffset	Sets the audio demodulation carrier frequency offset from the Center Frequency.
Declaration:	ReturnStatus AUDIO_SetFrequencyOffset(double freqOffsetHz);
Parameters:	
freqOffsetHz:	Amount of frequency offset from the Center Frequency. Range: –20e6 ≤ freqOffsetHz ≤ 20e6
Return Values:	
noError:	The function completed successfully.
errorParameter:	Input parameter out of range.
Additional Detail:	This function allows the audio demodulation carrier frequency to be offset from the device's Center Frequency. This allows tuning different carrier frequencies without changing the Center Frequency. The audio demodulation is performed at a carrier frequency of (Center Frequency + freqOffsetHz). The freqOffsetHz is set to an initial value of 0 Hz at the time the device is connected.

AUDIO_GetFrequencyOffset	Queries the audio carrier frequency offset from the Center Frequency.
Declaration:	ReturnStatus AUDIO_GetFrequencyOffset(double* freqOffsetHz);
Parameters:	
freqOffsetHz:	Pointer to a double variable. Returns the current audio frequency offset from the Center Frequency in Hz.
Return Values:	
noError:	The function completed successfully.

AUDIO_GetEnable	Queries the audio demodulation run state.
Declaration:	ReturnStatus AUDIO_GetEnable(bool *enable);
Parameters:	
freqOffsetHz:	Pointer to bool variable. True indicates the audio demodulation is running. False indicates it is stopped.
Return Values:	
noError:	The query was successful.

AUDIO_GetData	Returns audio sample data in a user buffer.
Declaration:	ReturnStatus AUDIO_GetData(int16_t* data, uint16_t inSize, uint16_t* outSize);
Parameters:	
data:	Pointer to a 16 bit integer array. 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_GetMode	Queries the audio demodulation mode.
Declaration:	ReturnStatus AUDIO_GetMode(AudioDemodMode* _mode);
Parameters:	
_mode:	Pointer to AudioDemodMode mode. 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_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. True indicates the speaker output is muted. False indicates the speaker output is not muted.
Return Values:	
noError:	The mute status has been successfully queried.
Additional Detail:	The status of the mute operation does not stop the audio processing or data callbacks.

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.

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.

AUDIO_SetMute — Sets the mute status.

Declaration: ReturnStatus AUDIO_SetMute(bool mute);

Parameters:

mute: Mute status. True mutes the output speakers. False restores the output speaker sound.

Return Values:

noError: The mute status has been successfully set.

Additional Detail: It does not affect the data processing or callbacks.

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_Start	Starts the audio demodulation output generation.
Declaration:	ReturnStatus AUDIO_Start();
Return Values:	
noError:	The audio demodulation output generation has started.

AUDIO_Stop	Stops the audio demodulation output generation.
Declaration:	ReturnStatus AUDIO_Stop()
Return Values:	
noError:	The audio demodulation output generation has stopped.

Configure functions

CONFIG_GetCenterFreq	Queries the center frequency.
Declaration:	ReturnStatus CONFIG_GetCenterFreq(double* cf);
Parameters:	
cf:	Pointer to a double. Contains the center frequency when the function completes.
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.

CONFIG_GetExternalRefEnable	Queries the state of the external reference.
Declaration:	ReturnStatus CONFIG_GetExternalRefEnable(bool* exRefEn);
Parameters:	
exRefEn:	Pointer to a bool. Contains the status of the external reference when the function completes. True indicates the external reference is enabled. False indicates the external reference is disabled.
Return Values:	
noError:	The function has completed successfully.

CONFIG_GetExternalRefFrequency	Queries the frequency of the external reference.
Declaration:	ReturnStatus CONFIG_GetExternalRefFrequency(double* extFreq);
Parameters:	
extFreq:	Pointer to a double. On return, contains the frequency in Hz of the attached external reference input.
Return Values:	
noError:	The function has completed successfully.
errorExternalReferenceNotEnabled:	The external reference input is not in use.
Additional Detail:	The external reference input must be enabled for this function to return useful results.

CONFIG_GetMaxCenterFreq	Queries the maximum center frequency.
Declaration:	ReturnStatus CONFIG_GetMaxCenterFreq(double* maxCF);
Parameters:	
maxCF:	Pointer to a double. Contains the maximum center frequency when the function completes.
Return Values:	
noError:	The maximum center frequency value has been queried.
errorNotConnected:	The device is not connected.
Additional Detail:	The value is stored in the maxCF parameter.

CONFIG_GetMinCenterFreq	Queries the minimum center frequency.
Declaration:	ReturnStatus CONFIG_GetMinCenterFreq(double* minCF);
Parameters:	
minCF:	Pointer to a double. Contains the minimum center frequency when the function completes.
Return Values:	
noError:	The minimum center frequency value has been queried.
errorNotConnected:	The device is not connected.
Additional Detail:	The value is stored in the minCF parameter.

CONFIG_GetReferenceLevel	Queries the reference level.
Declaration:	ReturnStatus CONFIG_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 function has completed successfully.
errorNotConnected:	The device is not connected.
Additional Detail:	The value is stored in the refLevel parameter.

CONFIG_Preset	This function sets the trigger mode to Free Run, 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 CONFIG_Preset();
Return Values:	
noError:	The preset values have been set.
errorNotConnected:	The device is not connected.

CONFIG_SetCenterFreq	Sets the center frequency value.
Declaration:	ReturnStatus CONFIG_SetCenterFreq(double cf);
Parameters:	
cf:	Value to set Center Frequency, in Hz. The value must be within the range MinCF to MaxCF.
Return Values:	
noError:	The center frequency has been queried.
errorNotConnected:	The device is not connected.
Additional Detail:	When using the tracking generator, the tracking generator output (TRKGEN_SetOutputLevel) should be set prior to setting the center frequency.

CONFIG_SetExternalRefEnable	Enables or disables the external reference.
Declaration:	ReturnStatus CONFIG_SetExternalRefEnable(bool exRefEn);
Parameters:	
exRefEn:	Enables or disables the external reference. True enables the external reference. False 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 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.

CONFIG_SetReferenceLevel	Sets the reference level.
Declaration:	ReturnStatus CONFIG_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 setting controls the signal path gain and attenuation settings. The value should be set to the maximum expected signal input power level, in dBm. Setting the value too low may result in over-driving the signal path and ADC, while setting it too high results in excess noise in the signal.

CONFIG_GetAutoAttenuationEnable	RSA500A Series and RSA600A Series only. Queries signal path auto-attenuation enable state.
Declaration:	ReturnStatus CONFIG_GetAutoAttenuationEnable(bool *enable);
Parameters:	
enable:	Pointer to a bool. True indicates that auto-attenuation operation is enabled. False indicates it is disabled.
Return Values:	
noError:	The function completed successfully.
errorNotConnected:	The device is not connected.
Additional Detail:	This function returns the enable state value set by the last call to CONFIG_SetAutoAttenuationEnable(), regardless of whether it has been applied to the hardware yet.

CONFIG_SetAutoAttenuationEnable	RSA500A Series and RSA600A Series only. Sets the signal path auto-attenuation enable state.
Declaration:	ReturnStatus CONFIG_SetAutoAttenuationEnable(bool enable);
Parameters:	
enable:	True enables auto-attenuation operation. False disables it.
Return Values:	
noError:	The function completed successfully.
errorNotConnected:	The device is not connected.
Additional Detail:	When auto-attenuation operation is enabled, the RF Input Attenuator is automatically configured to an optimal value which accommodates input signal levels up to the Reference Level. Auto-attenuation operation bases the attenuator setting on the current Reference Level, Center Frequency and RF Preamplifier state. When the RF Preamplifier is enabled, the RF Attenuator setting is adjusted to account for the additional gain. Note that auto-attenuation state does not affect the RF Preamplifier state. The device Run state must be re-applied to apply the new state value to the hardware. At device connect time, the auto-attenuation state is initialized to enabled (true).

CONFIG_GetRFPreampEnable	RSA500A Series and RSA600A Series only. Queries the state of the RF Preamplifier.
Declaration:	ReturnStatus CONFIG_GetPreampEnable(bool *enable);
Parameters:	
enable:	Pointer to a bool. True indicates the RF Preamplifier is enabled. False indicates it is disabled.
Return Values:	
noError:	The function completed successfully.
errorNotConnected:	The device is not connected.
Additional Detail:	This function returns the RF Preamplifier enable state value set by the last call to CONFIG_SetRFPreampEnable(), regardless of whether it has been applied to the hardware yet.

CONFIG_SetRFPreampEnable	RSA500A Series and RSA600A Series only. Sets the RF Preamplifier enable state.
Declaration:	ReturnStatus CONFIG_SetRFPreampEnable(bool enable);
Parameters:	
enable:	True enables the RF Preamplifier. False disables it.
Return Values:	
noError:	The function completed successfully.
errorNotConnected:	The device is not connected.
Additional Detail:	This function provides direct control of the RF Preamplifier. The Preamplifier state is independent of the auto-attenuation state or RF Attenuator setting. The Preamplifier provides nominally 25 dB of gain when enabled, with gain varying over the device RF frequency range (refer to the device data sheet for detailed preamp response specifications). When the Preamplifier is enabled, the device Reference Level setting should be –15 dBm or lower to avoid saturating internal signal path components. The device Run state must be re-applied to cause a new state value to be applied to the hardware.

CONFIG_GetRFAttenuator	RSA500A Series and RSA600A Series only. Queries the setting of the RF Input Attenuator.
Declaration:	ReturnStatus CONFIG_GetRFAttenuator(double *value);
Parameters:	
value:	Pointer to a double. Returns the RF Input Attenuator setting value in dB.
Return Values:	
noError:	The function completed successfully.
errorNotConnected:	The device is not connected.
Additional Detail:	If auto-attenuation is enabled, the returned value is the current RF attenuator hardware configuration. If auto-attenuation is disabled (manual attenuation mode), the returned value is the last value set by CONFIG_SetRFAttenuator(), regardless of whether it has been applied to the hardware.

CONFIG_SetRFAttenuator	RSA500A Series and RSA600A Series only. Sets the RF Input Attenuator value manually.
Declaration:	ReturnStatus CONFIG_SetRFAttenuator(double value);
Parameters:	
value:	Setting to configure the RF Input Attenuator, in dB units.
Return Values:	
noError:	The function completed successfully
errorNotConnected:	The device is not connected.
Additional Detail:	This function allows direct control of the RF Input Attenuator setting. The attenuator can be set in 1 dB steps, over the range –51 dB to 0 dB. Input values outside the range are converted to the closest legal value. Input values with fractional parts are rounded to the nearest integer value, giving 1 dB steps. The device auto-attenuation state must be disabled for this control to have effect. Setting the attenuator value with this function does not change the auto-attenuation state. Use CONFIG_SetAutoAttenuationEnable() to change the auto-attenuation state. The device Run state must be re-applied to cause a new setting value to be applied to the hardware. Improper manual attenuator setting may cause signal path saturation, resulting in degraded performance. This is particularly true if the RF Preamplifier state is changed. When making significant attenuator or preamp setting changes, it is recommended to use auto-attenuation mode to set the initial RF Attenuator level for a desired Reference Level, then query the attenuator setting to determine reasonable values for further manual control.

Device functions

DEVICE_Connect	Connects to a device specified by the deviceID parameter.
Declaration:	ReturnStatus DEVICE_Connect(int 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.
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.

DEVICE_Disconnect	Stops data acquisition and disconnects from the connected device.
Declaration:	ReturnStatus DEVICE_Disconnect();
Return Values:	
noError:	The device has been disconnected.
errorDisconnectFailure:	The disconnect failed.

DEVICE_GetEnable	Queries the run state.
Declaration:	ReturnStatus DEVICE_GetEnable(bool* enable);
Parameters:	
enable:	Pointer to a bool variable. Returns the device run state. True indicates the device is in the Run state. False indicates it is in the Stop state.
Return Values:	
noError:	The run state has been queried.
errorNotConnected:	The device is not connected.
Additional Detail:	The value is stored in the enable parameter. The device only produces data results when in the Run state, when signal samples flow from the device to the host API.

DEVICE_GetErrorString	Returns a string that corresponds to the ReturnStatus value specified by the status parameter.
Declaration:	ReturnStatus const char* DEVICE_GetErrorString(ReturnStatus status);
Parameters:	
status:	A ReturnStatus value.
Return Values:	Pointer to a string corresponding to the status input value. ReturnStatus error codes are listed in the RSA_API.h interface file.

DEVICE_GetFPGAVersion	Stores the FPGA version number in the fpgaVersion parameter.
Declaration:	ReturnStatus DEVICE_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

DEVICE_GetFWVersion	Stores the firmware version number in the fwVersion parameter.
Declaration:	ReturnStatus DEVICE_GetFWVersion(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

DEVICE_GetHWVersion	Stores the hardware version in a string. It has the form: “V versionNumber”.
Declaration:	ReturnStatus DEVICE_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.

Obtaining a device’s nomenclature can be accomplished with similar functions. These functions are grouped together.

DEVICE_GetNomenclature	Stores the name of the device in the nomenclature parameter.
Declaration:	ReturnStatus DEVICE_GetNomenclature(char* nomenclature);
DEVICE_GetNomenclatureW	Stores the name of the device in the nomenclatureW parameter.
Declaration:	ReturnStatus DEVICE_GetNomenclatureW(wchar_t* nomenclatureW);
Parameters:	
nomenclature:	Char string that contains the name of the device when the function completes.
nomenclatureW:	Wchar_t string that contains the name of the device when the function completes.
Return Values:	
noError:	The string name has been set.

DEVICE_GetSerialNumber	Stores the serial number of the device in the serialNum parameter.
Declaration:	ReturnStatus DEVICE_GetSerialNumber(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.

DEVICE_GetAPIVersion	Stores the API version number in the apiVersion parameter.
Declaration:	ReturnStatus DEVICE_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

DEVICE_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 DEVICE_PrepareForRun();
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 DEVICE_StartFrameTransfer). This is in comparison to the Run() function, which immediately starts data streaming without waiting for a Go signal.

DEVICE_GetInfo	Retrieves multiple device and version information strings.
Declaration:	ReturnStatus DEVICE_GetInfo(DEVICE_INFO* devInfo);
Parameters:	
devInfo:	Pointer to DEVICE_INFO structure which contains the device and version information strings on return.
Return Values:	
noError:	The function has successfully completed.
errorNotConnected:	A device is not connected.
Additional Detail:	The device must be connected to perform this operation. The device Nomenclature, Serial Number, FW version, FPGA version, HW version, and the API SW version are returned in strings within the DEVICE_INFO structure. The caller must create an instance of this structure and pass a pointer to the function. The format of each information string is the same as those described in the individual DEVICE_Get... functions.

DEVICE_GetOverTemperatureStatus	Queries for device over-temperature status.
Declaration:	ReturnStatus DEVICE_GetOverTemperatureStatus(bool* overTemperature);
Parameters:	
overTemperature:	Pointer to a bool variable. Returns over-temperature status. True indicates the internal device temperature is above nominal safe operating range, and may result in reduced accuracy and/or damage to the device. False indicates the device temperature is within the safe operating range.
Return Values:	
noError:	The function has successfully completed.
errorNotConnected:	A device is not connected.
Additional Detail:	This function allows clients to monitor the device's internal temperature status when operating in high-temperature environments. If the over-temperature condition is detected, the device should be powered down or moved to a lower temperature area.

DEVICE_Reset	Reboots the specified device.
Declaration:	ReturnStatus DEVICE_Reset(int deviceID);
Return Values:	
noError:	The device has been rebooted.
errorRebootFailure:	The reboot failed.

DEVICE_Run	Starts data acquisition.
Declaration:	ReturnStatus DEVICE_Run();
Return Values:	
noError:	The device has begun data acquisition.
errorTransfer:	The device did not receive the command.
errorNotConnected:	The device is not connected.

Searching for devices can be accomplished with several similar functions. These functions are grouped together.

DEVICE_Search	Searches for connectable devices (user buffers)
Declaration:	ReturnStatus DEVICE_Search(int* numDevicesFound, int deviceIDs[], char deviceSerial[][DEVSRCH_SERIAL_MAX_STRLEN], char deviceType[][DEVSRCH_TYPE_MAX_STRLEN]);
DEVICE_SearchW	Searches for connectable devices (user buffers, w_char strings).
Declaration:	ReturnStatus DEVICE_SearchW(int* numDevicesFound, int deviceIDs[], wchar_t deviceSerial[][DEVSRCH_SERIAL_MAX_STRLEN], wchar_t deviceType[][DEVSRCH_TYPE_MAX_STRLEN]);
DEVICE_SearchInt	Searches for connectable devices (internal buffers).
Declaration:	ReturnStatus DEVICE_SearchInt(int* numDevicesFound, int* deviceIDs[], const char** deviceSerial[], const char** deviceType[]);

DEVICE_SearchIntW — Searches for connectable devices (internal buffers, w_char strings).

Declaration:

ReturnStatus DEVICE_SearchIntW(int* numDevicesFound, int* deviceIDs[], const wchar_t** deviceSerial[], const wchar_t** deviceType[]);

Parameters:

numDevicesFound: Pointer to an integer variable. Returns the number of devices found by the search call. A returned value of 0 indicates no devices found.

deviceIDs: Returns an array of device ID numbers, numDevicesFound entries.

deviceSerial: Returns an array of strings of device serial numbers, numDevicesFound entries. char or wchar_t strings are returned depending on the function used.

deviceType: Returns an array of strings of device types, numDevicesFound entries. char or wchar_t strings are returned depending on the function used. Valid device type strings are: "RSA306", "RSA306B", "RSA503A", "RSA507A","RSA603A","RSA607A"

Return Values:

noError: The search succeeded.

Additional Detail:

The *numDevicesFound* value indicates if any devices were detected. If this value is 0, the other returned items are not defined and should not be used.

Search functions with "Int" in their name return array items in static internal array buffers. Caller does not need to allocate these arrays externally. Internal result buffers remain valid until the next search operation is performed. Search functions without "Int" in the name require the caller to allocate external storage for result arrays.

Usage with user-supplied result buffers:

```
int numDev; int devID[RSA_API::DEVSRCH_MAX_NUM_DEVICES];
{char|wchar_t} devSN[RSA_API::DEVSRCH_MAX_NUM_DE-
VICES][RSA_API::DEVSRCH_SERIAL_MAX_STRLEN];
{char|wchar_t} devType[RSA_API::DEVSRCH_MAX_NUM_DE-
VICES][RSA_API::DEVSRCH_TYPE_MAX_STRLEN];
// Results returned in user-supplied buffers
rs = RSA_API::DEVICE_Search{W}(&numDev, devID, devSN, devType);
```

Usage with internal result buffers ("Int" functions):

```
int numDevices;
int* devID; // ptr to devID array
const {char|wchar_t}** devSN; // ptr to array of ptrs to devSN strings
const {char|wchar_t}** devType; // ptr to array of ptrs to devType strings
// Results returned in internal static buffers
rs = RSA_API::DEVICE_SearchInt{W}(&numDev, &devID, &devSN,
&devType);
```

DEVICE_StartFrameTransfer	Starts data transfer.
Declaration:	ReturnStatus DEVICE_StartFrameTransfer();
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 DEVICE_PrepareForRun. If the system is in the stopped state, this call places it back into the run state with no changes to any internal data or settings, and data streaming will begin assuming there are no errors.

DEVICE_Stop	Stops data acquisition.
Declaration:	ReturnStatus DEVICE_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.

DEVICE_GetEventStatus	Queries global device real-time event status.
Declaration:	ReturnStatus DEVICE_GetEventStatus(int eventID, bool* eventOccurred, uint64_t* eventTimestamp);
Parameters:	
eventID:	ID value identifying the event status to query. Valid IDs are: DEVEVENT_OVERRANGE (0) DEVEVENT_TRIGGER (1) DEVEVENT_1PPS (2)
eventOccurred:	Pointer to a boolean variable. True indicates the event has occurred. False indicates no event occurrence.
eventTimestamp:	Pointer to uint64_t variable returning the event occurrence timestamp. Only valid if eventOccurred indicates an event occurred.
Return Values:	
noError:	The function has successfully completed.
errorNotConnected:	A device is not connected.
Additional Detail:	The device should be in the Run state when this function is called. Event information is only updated in the Run state, not in the Stop state. Overrange event detection requires no additional configuration to activate. The event indicates that the ADC input signal exceeded the allowable range, and signal clipping has likely occurred. The reported timestamp value is the most recent USB transfer frame in which a signal overrange was detected. Trigger event detection requires the appropriate HW trigger settings to be configured. These include trigger Mode, Source (External or IF Power), Transition, and IF Power Level (if IF power trigger is selected). The event indicates that the trigger condition has occurred. The reported timestamp value is of the most recent sample instant when a trigger event was detected. The API ForceTrigger function can be used to simulate a trigger event. 1PPS event detection (RSA500A/600A only) requires the GNSS receiver to be enabled and have navigation lock. The event indicates that the 1PPS event has occurred. The reported timestamp value is of the most recent sample instant when the GNSS Rx 1PPS pulse rising edge was detected. Querying an event causes the information for that event to be cleared after its state is returned. Subsequent queries will report "no event" until a new one occurs. All events are cleared when the device state transitions from Stop to Run state.

DPX functions

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 DPX spectrum.
enableSpectrogram:	Enables or disables DPX spectrogram.
Return Values:	
noError:	The function has executed successfully.
Additional Detail:	This function must be called after any DPX settings have been changed and the device is in Stop state. This function configures all the DPX settings. See the following steps for an example of how to setup and acquire DPX data: **1.** Set the device in Stop state. **2.** Setup DPX settings. **3.** Call DPX_SetEnable() to enable DPX acquisition. **4.** Set the device in Run state. **5.** While the device is in Run state, call DPX_WaitForDataReady() to wait for DPX frame buffer available. **6.** When DPX frame is available, call DPX_GetFrameBuffer() to get DPX bitmaps and traces. **7.** Call DPX_FinishFrameBuffer() to indicate the caller has finished transferring the DPX frame data. **8.** Repeat waiting and getting the next DPX frame buffer. **9.** After DPX acquisition has completed and the device is in Stop state, you can use the following functions to get high resolution lines in the DPX spectrogram (if SPX spectrogram is enabled): DPX_GetSogramHiResLineCountLatest() DPX_GetSogramHiResLine() DPX_GetSogramHiResLineTimestamp() DPX_GetSogramHiResLineTriggered()

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_GetEnable	Checks the status of DPX.
Declaration:	ReturnStatus DPX_GetEnable(bool* enabled);
Parameters:	
enabled:	Pointer to a bool. It queries the state of the DPX mode. True indicates DPX is enabled. False indicates DPX is disabled.
Return Values:	
noError:	The operation completed successfully.
Additional Detail:	

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 DPX_FrameBuffer struct. See DPX_FrameBuffer table for descriptions. (See Table 1 on page 21.)
Return Values:	
noError:	The function has executed successfully.

Table 1: 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.
bool spectrogramEnabled	True, DPX Spectogram is enable. False, DPX Spectogram is disabled. See DPX_Configure.

Table 1: DPX_FrameBuffer description (cont.)

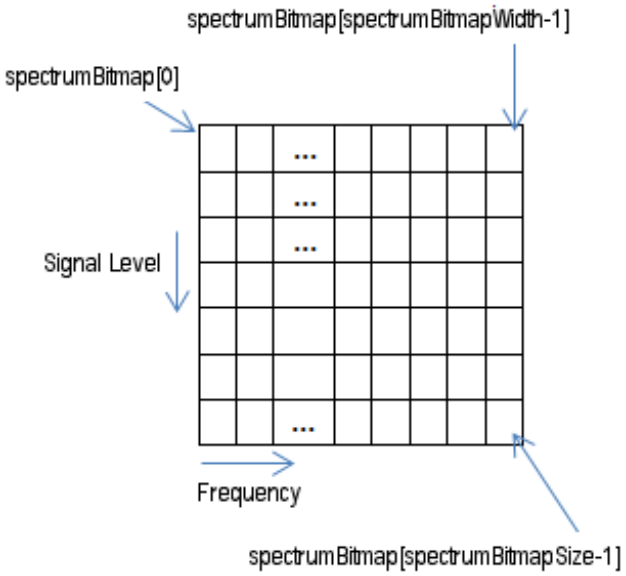
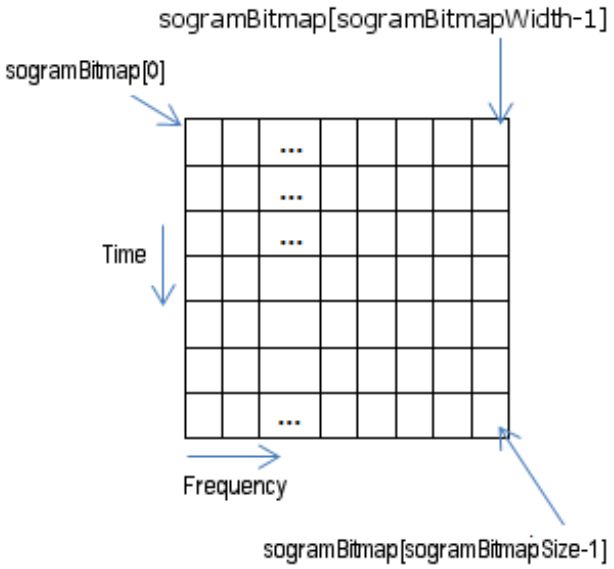
DPX_FrameBuffer	Description
float* spectrumBitmap	DPX Spectrum bitmap array. Each value represents the hit count of each pixel in the DPX Spectrum bitmap. The first element in the array represents the upper left corner of the bitmap and the second element represents the pixel to the right of the first 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. For example, if yTop = 0 dBm and yBottom = –100 dBm in DPX_SetParameters() and spectrumBitmapHeight in DPX_FrameBuffer = 201. The first row of the spectrumBitmap represents signal level from 0.25 dBm to –0.25 dBm and the bottom row of the spectrumBitmap represents signal level from –99.75 dBm to –100.25 dBm. 
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 1: DPX_FrameBuffer description (cont.)

DPX_FrameBuffer	Description
uint8_t* sogramBitmap	Spectrogram bitmap 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.

DPX_GetFrameInfo	Queries the latest 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 total number of DPX frames since DPx acquisition started.
fftCount:	Pointer to a 64 bit integer. Contains the total number of FFT performed since DPx acquisition started.
Return Values:	
noError:	The function has executed successfully.

DPX_GetRBWRange	Queries the valid RBW range based on span.
Declaration:	ReturnStatus DPX_GetRBWRange(double fspan, double *minRBW, double *maxRBW);
Parameters:	
fpsan:	Span measured in Hz. This value must be greater than 0.
minRBW:	Returns minimum RBW in Hz.
maxRBW:	Returns maximum RBW in Hz.
Return Values:	
noError:	The function has executed successfully.

DPX_GetSettings — Queries the current DPX settings.

Declaration: ReturnStatus DPX_GetSettings(DPX_SettingStruct *dpxSettings);

Parameters:

dpxSettings: Pointer to DPX_SettingsStruct.

DPX_SettingsStruct.

Item	Description
bool enableSpectrum	True if DPX spectrum is enabled; false if DPX spectrum is disabled
bool enableSpectrogram	True if DPX spectrogram is enabled; false if DPX spectrogram is disabled
int32_t bitmapWidth	DPX spectrum bitmap width in pixels
int32_t bitmapHeight	DPX spectrum bitmap height in pixels
int32_t traceLength	Number of trace points
float decayFactor	This is calculated based on persistenceTimeSec parameter in DPX_SetParameters(). During the decay process on each DPX frame, the hit count of each pixel in the DPX spectrum bitmap is multiplied by the decayFactor.
double actualRBW	Actual RBW in Hz

Return Values:

noError: The function has executed successfully.

Additional Detail: After changing DPX settings, DPX_Configure() must be called before this function will return valid DPX settings.

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 array. The array returns the data stored in the spectrogram high resolution line.
vDataSize:	Pointer to a 32 bit integer. Returns the amount of valid elements in the vData parameter array.
lineIndex:	The spectrogram line index.
dataSF:	Pointer to a double. Returns the scale factor. The spectrogram high resolution line signal level in dBm unit can be calculated by multiplying dataSF with the elements in vData array.
tracePoints:	The amount of trace points to return.
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. For example, if the firstValidPoint parameter is 10 and tracePoints parameter is 100, then the values of the high resolution line trace points from index 10 to 109 will be returned in the vData array in index 0 to 99. Since the spectrogram high resolution lines are updated continuously while DPX is acquiring, this function should be called when DPX is stopped.

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.
Additional Details:	Each high resolution line may be composed from multiple FFT acquisitions and the DPX acquisition can be stopped at any time. Therefore, the latest high resolution line may not contain all the FFTs in a high resolution line.

DPX_GetSogramHiResLineTimestamp	Queries the timestamp of a DPX spectrogram high resolution line.
Declaration:	ReturnStatus DPX_GetSogramHiResLineTimestamp(double* timestamp, int32_t lineIndex);
Parameters:	
timestamp:	Pointer to a double. Contains the timestamp value of the spectrogram high resolution line.
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. Since the spectrogram high resolution lines are updated continuously while DPX is acquiring, this function should be called when DPX is stopped.

DPX_GetSogramHiResLineTriggered	Queries the triggered status of a DPX spectrogram high resolution line.
Declaration:	ReturnStatus DPX_GetSogramHiResLineTriggered(bool* triggered, int32_t lineIndex);
Parameters:	
triggered:	Pointer to a bool. True indicates the specified high resolution line is triggered. False indicates the specified high resolution line is not triggered.
lineIndex:	The index of the high resolution spectrogram line.
Return Values:	
noError:	The function has executed successfully.
Additional Detail:	Since the spectrogram high resolution lines are updated continuously while DPX is acquiring, this function should be called when DPX is stopped.

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

Item	Description
int32_t bitmapWidth	DPX spectrogram bitmap width in pixels.
int32_t bitmapHeight	DPX spectrogram bitmap height in pixels.
double sogramTraceLineTime	Time per each DPX spectrogram high resolution trace line in seconds.
double sogramBitmapLineTime	Time per each DPX spectrogram bitmap line in seconds

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. True indicates the frame is available. False indicates the frame is not available.
Return Values:	
noError:	The function has executed successfully.
Additional Detail:	Refer to the DPX_FrameBuffer description table for more information. (See Table 1.)

DPX_Reset	Clears the spectrum bitmap, resets the spectrum traces, resets the spectrogram bitmap, resets the spectrogram traces, sets the FFT count to 0, and sets the frame count to 0.
Declaration:	ReturnStatus DPX_Reset();
Return Values:	
noError:	The function has executed successfully.

DPX_SetEnable	Enables or disables DPX.
Declaration:	ReturnStatus DPX_SetEnable(bool enabled);
Parameters:	
enabled:	True enables DPX. False disables DPX.
Return Values:	
noError:	DPX has been successfully enabled or disabled.

DPX_SetParameters

Sets the DPX span, resolution bandwidth, trace points 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 0 and less than or equal to 40 MHz.

rbw:

Resolution bandwidth measured in Hz.

This value must be greater than 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. The total number of trace points is equal to tracePtsPerPixel * bitmapWidth.

Valid values are: 1, 3, 5.

yUnit:

Units of the Y-axis.

VerticalUnitType	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. If true, DPX frame is only available when a trigger occurs. If false, DPX frame is available continuously.

Return Values:

noError:

The function has executed successfully.

DPX_SetSogramParameters	Sets the amount of time that each spectrogram line represents and the signal level 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 in seconds. Each bitmap line is composed of one or more spectrogram high resolution lines.
timeResolution:	The amount of time that each spectrogram high resolution line represents in seconds. This value must be greater than or equal to 1 ms.
maxPower:	The maximum signal level of the spectrogram bitmap in current Vertical Unit (yUnit in DPX_SetParameters).
minPower:	The minimum signal level of the spectrogram bitmap in current Vertical Unit (yUnit in DPX_SetParameters).
Return Values:	
noError:	The function has executed successfully.
Additional Detail:	See sogramBitmap item in DPX_FrameBuffer description table for the usage of maxPower and minPower. (See Table 1 on page 21.)

DPX_SetSogramTraceType	Sets the DPX spectrogram trace type.	
Declaration:	ReturnStatus DPX_SetSogramTraceType(TraceType traceType);	
Parameters:		
traceType:	A value of type TraceType.	
	TraceType	**Value**
	TraceTypeAverage	0
	TraceTypeMax	1
	TraceTypeMin	3
Return Values:		
noError:	The function has executed successfully.	
Additional Detail:	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.	

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, TraceType type);
Parameters:	
traceIndex:	Specifies the trace to be set. It can be 0, 1, or 2.
type:	A value of type TraceType.

TraceType	**Value**
TraceTypeAverage	0
TraceTypeMax	1
TraceTypeMaxHold	2
TraceTypeMin	3
TraceTypeMinHold	4

Return Values:	
noError:	The function has executed successfully.

DPX_WaitForDataReady	Waits for the DPX data to be ready to be queried.
Declaration:	ReturnStatus DPX_WaitForDataReady(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.

GNSS functions

The RSA500A Series and RSA600A Series devices include a Global Navigation Satellite System (GNSS) receiver (Telit SL869-V2) capable of tracking GPS, Glonass, or Beidou satellite navigation signals. The GNSS receiver provides status, position, and time messages in NMEA 0183 format, along with a high accuracy 1-Pulse-Per-Second (1PPS) timing pulse usable for internal signal timestamping. User access to the navigation message stream and 1PPS event are provided through API GNSS functions. User-controllable GNSS antenna power output is also provided.

GNSS_ClearNavMessageData	RSA500A Series and RSA600A Series only.
	Clears the navigation message data queue.
Declaration:	ReturnStatus GNSS_ClearNavMessageData();
Return Values:	
noError:	The function has successfully completed.
Additional Detail:	The data queue which holds GNSS navigation message character strings is emptied.

GNSS_Get1PPSTimestamp	RSA500A Series and RSA600A Series only.
	Queries the status of the internal 1PPS timing pulse.
Declaration:	ReturnStatus GNSS_Get1PPSTimestamp(bool* isValid, uint64_t* timestamp1PPS);
Parameters:	
isValid:	Pointer to bool. True indicates a new valid 1PPS pulse timestamp is available. False indicates it is not available.
timestamp1PPS:	Pointer to uint64_t. Returns the timestamp of the most recent 1PPS pulse.
Return Values:	
noError:	The function has successfully completed.
Additional Detail:	The internal GNSS receiver must be enabled and have navigation lock for this function to return useful information, otherwise it returns isValid = false. When isValid is true, it indicates that an internal 1PPS pulse has been detected. In that case, the timestamp1PPS value contains the internal timestamp of the 1PPS pulse. 1PPS pulses occur each second, so the user application should call this function at least once per second to retrieve the 1PPS information correctly. The 1PPS timestamp along with the decoded UTC time from the navigation messages can be used to set the API system time to GNSS-accurate time reference. See REFTIME_SetReferenceTime() for more information on setting reference time based on these values.

GNSS_GetAntennaPower	RSA500A Series and RSA600A Series only.
	Queries the GNSS antenna power output state.
Declaration:	ReturnStatus GNSS_GetAntennaPower(bool* powered);
Parameters:	
powered:	Pointer to a bool. True indicates the GNSS antenna power output is enabled. False indicates it is disabled.
Return Values:	
noError:	The function has successfully completed.
Additional Detail:	The returned value indicates the state set by GNSS_SetAntennaPower(), although the actual output state may be different. See the entry for GNSS_SetAntennaPower() for more information on GNSS antenna power control.

GNSS_GetEnable	RSA500A Series and RSA600A Series only.
	Queries the internal GNSS receiver enable state.
Declaration:	ReturnStatus GNSS_GetEnable(bool* enable);
Parameters:	
enable:	Pointer to a bool. True indicates the GNSS receiver is enabled. False indicates it is disabled.
Return Values:	
noError:	The function has successfully completed.

GNSS_GetHwInstalled	RSA500A Series and RSA600A Series only.
	Queries whether internal GNSS receiver HW is installed.
Declaration:	ReturnStatus GNSS_GetHwInstalled(bool *installed);
Parameters:	
installed:	Pointer to a bool. True indicates the GNSS receiver HW is installed. False indicates it is not installed.
Return Values:	
noError:	The function has successfully completed.
Additional Detail:	GNSS HW is only installed in RSA500A and RSA600A devices. All other devices will indicate no HW installed.

GNSS_GetNavMessageData	RSA500A Series and RSA600A Series only.
	Query for navigation message data.
Declaration:	ReturnStatus GNSS_GetNavMessageData(int* msgLen, const char** message);
Parameters:	
msgLen:	Pointer to int. Returns the number of chars in the message buffer. 0 indicates no chars available.
message:	Pointer to char. Returns a point to the API internal buffer containing navigation message characters. There will be msgLen chars in the buffer. The char string is terminated by a NULL char, not included in the msgLen count.
Return Values:	
noError:	The function has successfully completed.
Additional Detail:	The internal GNSS receiver must be enabled for this function to return useful data, otherwise it will always return msgLen = 0, indicating no data. The message output consists of contiguous segments of the ASCII character serial stream from the GNSS receiver, following the NMEA 0183 Version 3.0 standard. The character output rate is approximately 1000 characters per second, originating from an internal 9600 baud serial interface.
	The GNSS navigation message output includes RMC, GGA, GSA, GSV and other NMEA sentence types. The two character Talker Identifier following the starting "$" character may be "GP", "GL", "BD" or "GN" depending on the configuration of the receiver. The function does not decode the NMEA sentences. It passes them through in raw form, including all characters in the original serial stream.
	The message queue holding the message chars may overflow if this function is not called often enough to keep up with the data generation by the GNSS receiver. It is recommended to retrieve message data at least 4 times per second to avoid this overflow.

GNSS_GetSatSystem	RSA500A Series and RSA600A Series only.
	Queries the GNSS satellite system selection.
Declaration:	ReturnStatus GNSS_GetSatSystem(GNSS_SATSYS* satSystem);
Parameters:	
satSystem:	Pointer to GNSS_SATSYS type. Returns the ID of the currently selected system.
	See GNSS_SetSatSystem() entry for the ID information.
Return Values:	
noError:	The function has successfully completed.
errorFailed:	The function did not complete successfully. Returned parameter data is invalid.
Additional Detail:	This function should only be called when the GNSS Receiver is enabled. It will not return valid parameter data when the receiver is disabled.

GNSS_SetAntennaPower	RSA500A Series and RSA600A Series only.
	Sets the GNSS antenna power output state.
Declaration:	ReturnStatus GNSS_SetAntennaPower(bool powered);
Parameters:	
powered:	Sets the antenna power state. True enables the antenna power output. False disables it.
Return Values:	
noError:	The function has successfully completed.
Additional Detail:	The GNSS receiver must be enabled for antenna power to be output. If the receiver is disabled, antenna power output is also disabled, even when set to enabled state by this function. When antenna power is enabled, 3.0 VDC is switched to the antenna center conductor line for powering an external antenna. When disabled, the voltage source is disconnected from the antenna.

GNSS_SetEnable	RSA500A Series and RSA600A Series only.
	Enables or disables the internal GNSS receiver operation.
Declaration:	ReturnStatus GNSS_SetEnable(bool enable);
Parameters:	
enable:	True enables the GNSS receiver. False disables it.
Return Values:	
noError:	The function has successfully completed.
Additional Detail:	If the GNSS receiver functions are not needed, it should be disabled to conserve battery power.

GNSS_SetSatSystem

RSA500A Series and RSA600A Series only.

Sets the GNSS satellite system selection.

Declaration: ReturnStatus GNSS_SetSatSystem(GNSS_SATSYS satSystem);

Parameters:

satSystem: Sets the satellite systems used by the GNSS receiver. See below for details.

Return Values:

noError: The function has successfully completed.

errorFailed: The function did not complete successfully. Satellite system selection was not set.

Additional Detail: The GNSS receiver must be enabled to use this function.

The possible satellite system selections are:

ID Name	ID Value	Satellite systems used
GNSS_GPS_GLONASS	1	GPS + Glonass (default)
GNSS_GPS_BEIDOU	2	GPS + Beidou
GNSS_GPS	3	GPS only
GNSS_GLONASS	4	Glonass only
GNSS_BEIDOU	5	Beidou only

The satellite system selection limits the GNSS receiver to using only signals from the specified system(s). Use only a single ID type to configure the selection; do not combine IDs or values to get combinations not listed in the table.

Each time the GNSS receiver is enabled, the satellite system selection is set to the default value of GNSS_GPS_GLONASS (GPS+Glonass). Satellite system selections are not persistent or recallable, even within the same connection session. Any non-default setting must be explicitly applied after each receiver enable operation.

The setting can only be changed when the GNSS Receiver is enabled. If the function is called when the receiver is disabled, the selection is ignored and an error code is returned.

If the selected system(s) do not provide sufficient signal coverage at the antenna location, the GNSS receiver will not be able to acquire navigation lock. In most cases, the default selection provides the best coverage.

IF streaming functions

NOTE. *Before calling the API function IFSTREAM_SetEnable(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 IFSTREAM_SetEnable(true), you must not make any changes to hardware settings until you call IFSTREAM_SetEnable(false) or until enough time has elapsed such that all automatically created streamed files are completely written to disk.

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

Declaration: ReturnStatus IFSTREAM_SetDiskFilenameSuffix(int suffixCtl);

Parameters:

suffixCtl: Sets the filename suffix control value.

Note that the IFSSDFN_SUFFIX_TIMESTAMP setting is the default, and is applied automatically if the suffix control is not set after connection.

Return Values:

noError: The setting was accepted.

Additional Detail

The complete IF output filename has the following format:

<filePath><filenameBase><suffix><.ext>

where:

<filePath>,<filenameBase>: set by their associated IFSTREAM configuration functions

<suffix>: as set by filename suffix control using this function

<.ext>: as set by IFSTREAM file mode configurationfunction

[.r3f or .r3h+.r3a]

If separate data and header files are generated, the same path/filename is used for both, with different extensions to indicate the contents.

suffixCtl value	Suffix generated
IFSSDFN_SUFFIX_NONE (-2)	None. Filename is created without suffix. (Note that the output filename will not change automatically, so each output file will overwrite the previous one unless the filename is explicitly changed by calling the IF-STREAM_SetDiskFilenameBase() function.)
IFSSDFN_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.)
(Auto-increment index) ≥0	5 digit auto-incrementing index, initial value = suffixCtl. Format: “-nnnnn” (Note the index value auto-increments by 1 each time a new file is created.)

Below are examples of output filenames generated with different suffixCtl settings. Multiple filenames show suffix auto-generation behavior with each new file created. The most recent suffixCtl setting remains in effect until changed by calling this function with a different setting value.

(Assume <filePath>+<filenameBase> is “c:\myfile” and R3F file mode is selected.)

suffixCtl value	**Full Filename** (and behavior with multiple runs)
IFSSDFN_SUFFIX_NONE:	“c:\myfile.r3f” “c:\myfile.r3f” “c:\myfile.r3f”
IQSSDFN_SUFFIX_TIMESTAMP:	“c:\my-file-2015.04.15.09.33.12.522.r3f” “c:\myfile-2015.04.15.09.33.14.697.r3f” “c:\myfile-2015.04.15.09.33.17.301.r3f”
10:	“c:\myfile-00010.r3f” “c:\myfile-00011.r3f” “c:\myfile-00012.r3f”
4:	“c:\myfile-00004.r3f” “c:\myfile-00005.r3f”

IFSTREAM_GetActiveStatus	Allows the current status of the ADC data streaming operation to be queried.
Declaration:	ReturnStatus IFSTREAM_GetActiveStatus(bool *enabled);
Parameters:	
enabled:	Reports the current status of the ADC data streaming operation.
Return Values:	
noError:	The operation has completed successfully.

IFSTREAM_SetDiskFileCount	Sets the maximum number of files to open for streamed data.
Declaration:	ReturnStatus IFSTREAM_SetDiskFileCount(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.

IFSTREAM_SetDiskFileLength	Sets the maximum recording time for any single data file.
Declaration:	ReturnStatus IFSTREAM_SetDiskFileLength(int msec);
Parameters:	
msec:	Sets the maximum recording time for ADC files.
Return Values:	
noError:	The operation has completed successfully.
errorStreamADC-ToDiskAlreadyStreaming:	ADC streaming is already in operation.

IFSTREAM_SetDiskFileMode	Sets the streaming mode.
Declaration:	ReturnStatus IFSTREAM_SetDiskFileMode(StreamingMode mode);
Parameters:	
mode:	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 Streaming Sample Data File Format. (See page 90.)

IFSTREAM_SetDiskFilenameBase	Sets the base file name for file saves.
Declaration:	ReturnStatus IFSTREAM_SetDiskFilenameBase(const char *base);
Parameters:	
base:	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.

IFSTREAM_SetDiskFilePath	Sets the path for file saves.
Declaration:	ReturnStatus IFSTREAM_SetDiskFilePath(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.

IFSTREAM_SetEnable	Starts and stops the ADC streaming operation.
Declaration:	ReturnStatus IFSTREAM_SetEnable(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.

IQ block functions

IQBLK_GetIQAcqInfo — Queries the IQ acquisition status information for the most recently retrieved IQ Block record.

Declaration: ReturnStatus IQBLK_GetIQAcqInfo(IQBLK_ACQINFO* acqInfo);

Parameters:

acqInfo: Pointer to IQBLK_ACQINFO structure allocated by the caller.

Return Values:

noError: The function has completed successfully.

errorNotConnected: The device is not connected.

Additional Detail: IQBLK_GetIQAcqInfo() may be called after an IQ block record is retrieved with IQBLK_GetIQData(), IQBLK_GetIQDataInterleaved(), or IQBLK_GetIQDataComplex(). The returned information applies to the IQ record returned by the "GetData" functions.

The IQBLK_ACKINFO structure contains these items:

sample0Timestamp: uint64_t timestamp of the first sample of the IQ block record

triggerSampleIndex: uint64_t index to the sample corresponding to the trigger point

triggerTimestamp: uint64_t timestamp of the trigger sample

acqStatus: uint32_t word with acquistiion status bits. A status bit value of 1 indicates that event occurred during the signal acquisition, a value of 0 indicates no occurrence.

The valid status bits are described in the following Status Bits table.

Table 2: Status Bits

Status Bit	Description	
Bit 0:	IQBLK_STATUS_INPUT_OVERRANGE (mask=0x1):	ADC input overrange during acquisition
Bit 1:	IQBLK_STATUS_FREQREF_UNLOCKED (mask=0x2) :	Frequency Reference unlocked during acquisition
Bit 2:	IQBLK_STATUS_ACQ_SYS_ERROR (mask=0x4):	Internal oscillator unlocked or power failure during acquisition
Bit 3:	IQBLK_STATUS_DATA_XFER_ERROR (mask=0x8):	USB frame transfer error detected during acquisition

IQBLK_AcquireIQData	Initiates an IQ block record acquisition.
Declaration:	ReturnStatus IQBLK_AcquireIQData();
Return Values:	
noError:	The function has completed successfully.
errorNotConnected:	The device is not connected.
Additional Detail:	Executing this function initiates an IQ block record data acquisition. This function places the device in Run state if it is not already in that state. Before calling this function, all device acquisition parameters must be set to valid states. These include Center Frequency, Reference Level, any desired Trigger conditions, and the IQBLK Bandwidth and Record Length settings.

IQBLK_GetIQBandwidth	Queries the IQ bandwidth value.
Declaration:	ReturnStatus IQBLK_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.

IQBLK_GetIQData — Retrieves an IQ block data record in a single interleaved array format.

Declaration: ReturnStatus IQBLK_GetIQData(float* iqData, int* outLength, int reqLength);

Parameters:

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

outLength: Pointer to an integer variable. Returns the actual number of IQ sample pairs returned in iqData buffer.

reqLength: Number of IQ sample pairs requested to be returned in iqData. The maximum value of *reqLength* is equal to the *recordLength* value set in IQBLK_SetIQRecordLength(). Smaller values of *reqlLength* allow retrieving partial IQ records.

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

IQBLK_GetIQDataCplx — Retrieves an IQ block data record in Cplx32 array format.

Declaration: ReturnStatus IQBLK_GetIQDataCplx(Cplx32* iqData, int* outLength, int reqLength);

Parameters:

iqData: Pointer to an array of Cplx32 structs. Contains the IQ data when the function completes.

outLength: Pointer to an integer variable. Returns the actual number of complex IQ samples returned in iqData buffer.

reqLength: Number of IQ samples requested to be returned in iqData. The maximum value of reqLength is equal to the recordLength value set in IQBLK_SetIQRecordLength(). Smaller values of reqlLength allow retrieving partial IQ records.

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

IQBLK_GetIQDataDeinterleaved — Retrieves an IQ block data record in separate I and Q array format.

Declaration: ReturnStatus IQBLK_GetIQDataDeinterleaved(float* iData, float* qData, int* outLength, int reqLength);

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.

outLength: Pointer to an integer variable. Returns the actual number of I and Q sample values returned in iData and qData buffers.

reqLength: Number of IQ samples requested to be returned in iData and qData. The maximum value of reqLength is equal to the recordLength value set in IQBLK_SetIQRecordLength(). Smaller values of reqlLength allow retrieving partial IQ records.

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 with Q-data.

See the following illustration.

iData, length =2:

Index	0	1
Contents	I_0	I_1

qData, length =2:

Index	0	1
Contents	Q_0	Q_1

Actual IQ Data, length =2:

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

IQBLK_GetIQRecordLength — Queries the IQ record length.

Declaration: ReturnStatus IQBLK_GetIQRecordLength(int* recordLength);

Parameters:

recordLength: Pointer to an integer variable. Contains the number 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.

IQBLK_GetIQSampleRate	Queries the IQ sample rate value.
Declaration:	ReturnStatus IQBLK_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.
Additional Detail:	The IQ Sample Rate value depends on the IQ Bandwidth value set by IQBLK_SetIQBandwidth() function. Set the bandwidth value before querying the sample rate.

IQBLK_GetMaxIQBandwidth	Queries the maximum bandwidth value.
Declaration:	ReturnStatus IQBLK_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.

IQBLK_GetMaxIQRecordLength	Queries the maximum number of IQ samples which can be generated in one IQ block record.
Declaration:	ReturnStatus IQBLK_GetMaxIQRecordLength(int* maxSamples);
Parameters:	
maxCF:	Pointer to an integer. Contains the maximum IQ record length value when the function completes.
Return Values:	
noError:	The maxSamples value has been queried.

IQBLK_GetMinIQBandwidth	Queries the minimum bandwidth value.
Declaration:	ReturnStatus IQBLK_GetMinIQBandwidth(double* minBandwidth);
Parameters:	
minBandwidth:	Pointer to a double. Contains the minimum bandwidth value when the function completes.
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.

IQBLK_SetIQBandwidth	Sets the IQ bandwidth value.
Declaration:	ReturnStatus IQBLK_SetIQBandwidth(double iqBandwidth);
Parameters:	
iqBandwidth:	IQ bandwidth value measured in Hz. Range: Query the Min and Max IQ BW values for range.
Return Values:	
noError:	The IQ bandwidth value has been set.
errorNotConnected:	The device is not connected.
Additional Detail:	The IQ bandwidth must be set before the IQBLK_AcquireIQData() function is called.

IQBLK_SetIQRecordLength	Sets the number of IQ data samples to be generated by each IQ block acquisition.
Declaration:	ReturnStatus IQBLK_SetIQRecordLength(int recordLength);
Parameters:	
recordLength:	IQ record length. This value is measured in samples. Range: 2 — Max IQ record length. Query IQBLK_GetMaxIQRecordLength for value.
Return Values:	
noError:	The IQ record length value has been set.
errorNotConnected:	The device is not connected.

IQBLK_WaitForIQDataReady	Waits for the data to be ready to be queried.
Declaration:	ReturnStatus IQBLK_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. True indicates the data is ready for acquisition. False indicates the data is not ready and the timeout value is exceeded.
Return Values:	
noError:	The function has executed successfully.

IQ streaming functions

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, 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.

IQSTREAM_GetMaxAcqBandwidth	Queries the maximum IQ Bandwidth for IQ streaming.
Declaration:	ReturnStatus IQSTREAM_GetMaxAcqBandwidth(double* maxBandwidthHz);
Parameters:	
maxBandwidthHz:	Pointer to a double variable. Returns the maximum IQ bandwidth supported by IQ streaming.
Return Values:	
noError:	The function completed successfully.
Additional Detail:	The bandwidth value set in IQSTREAM_SetAcqBawndwidth() should be no larger than the value maxBandwidthHz returned by this function.

IQSTREAM_GetMinAcqBandwidth	Queries the minimum IQ Bandwidth for IQ streaming.
Declaration:	ReturnStatus IQSTREAM_GetMinAcqBandwidth(double* minBandwidthHz);
Parameters:	
minBandwidthHz:	Pointer to a double variable. Returns the minimum IQ bandwidth supported by IQ streaming.
Return Values:	
noError:	The function completed successfully.
Additional Detail:	The bandwidth value set in IQSTREAM_SetAcqBawndwidth() should be no smaller than the value minBandwidthHz returned by this function.

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.

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:

Pointer to a double. Returns actual acquisition bandwidth of IQ Streaming output data, in Hz.

srSps:

Pointer to a double. Returns actual sample rate of IQ Streaming output data, in Samples/sec

Return Values:

noError:

The query was successful.

Additional Detail:

This is 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

IQSTREAM_GetDiskFileInfo

Returns an information structure about the previous file output operation.

Declaration:

ReturnStatus IQSTREAM_GetDiskFileInfo(IQSTRMFILEINFO* fileinfo);

Parameters:

fileinfo:

Pointer to a 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.

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. **Individual Internal Write Block status** (Bits0-15, starting from LSB): Bit0: 1=Input overrange Bit1: 1=USB data stream discontinuity 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) **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: 1=USB data stream discontinuity 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_GetDiskFileWriteStatus	Allows monitoring the progress of file output.
Declaration:	ReturnStatus IQSTREAM_GetDiskFileWriteStatus(bool* isComplete, bool *isWriting);
Parameters:	
isComplete:	Pointer to a bool. Returns whether the IQ Stream file output writing complete.
isWriting:	Pointer to a 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]

IQSTREAM_GetEnable	This function returns the current IQ Stream processing state.
Declaration:	ReturnStatus IQSTREAM_GetEnable(bool* enabled);
Parameters:	
enabled:	Pointer to a bool. Returns the current IQ Stream processing enable status. True indicates IQ Stream processing is active. False indicates 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 iqbuffer. Returns IQ sample data block.
iqlen:	Pointer to an integer. Returns the number of IQ data pairs returned in iqbuffer. 0 indicates no data available.
iqinfo:	Pointer to a 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 is 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.

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.
triggerIndices	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 (Bits 0-15, starting from LSB): Bit0: 1=Input overrange Bit1: 1=USB data stream discontinuity 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: 1=USB data stream discontinuity 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.

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:

Pointer to an integer. 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);
```

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_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().

Sets the base filename for file output can be accomplished with similar functions. These functions are grouped together.	
IQSTREAM_SetDiskFilenameBase	Sets the base filename for file output (char string)
Declaration:	ReturnStatus IQSTREAM_SetDiskFilenameBase(const char* filenameBase);
IQSTREAM_SetDiskFilenameBaseW	Sets the base filename for file output (wchar_t string)
Declaration:	QSTREAM_SetDiskFilenameBaseW(const wchar_t* filenameBaseW)
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.
filenameBaseW:	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.
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.

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.)

Following 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	“myfile-00010.tiq” “myfile-00011.tiq” “myfile-00012.tiq” …
4	“myfile-00004.tiq” “myfile-00005.tiq” …

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.

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_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. TRIG_ForceTrigger() can be used to generate a trigger event if the specified one does not occur.

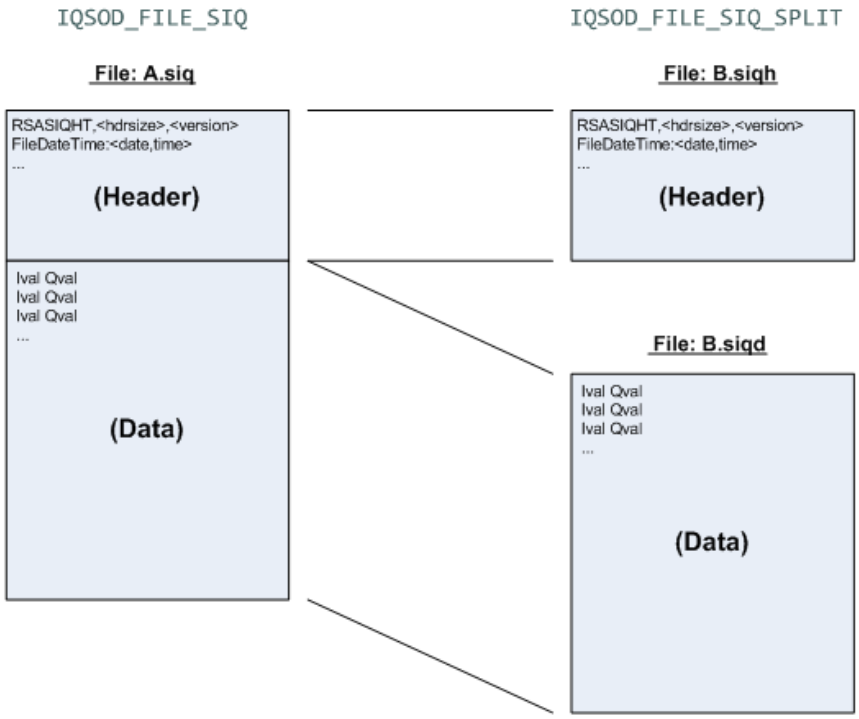
IQSTREAM_Stop	This function terminates IQ Stream processing and disables data output.
Declaration:	ReturnStatus IQSTREAM_Stop();
Parameters:	
(none):	
Return Values:	
noError:	The operation was successful.
Additional Detail:	If the data destination is file, file writing is stopped and the output file is closed.

IQ Streaming SIQ/SIQH/SIQD File Formats

IQ Streaming file outputs 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 3: 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
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

Table 3: IQ Streaming header content (cont.)

Header Info Item:	Header Info Value:	Example:
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
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

Table 3: IQ Streaming header content (cont.)

Header Info Item:	Header Info Value:	Example:
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_GetDiskFileInfo() 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 functions (R3F file format)

These functions pertain to the playback of files recorded with the RSA306, RSA306B, the RSA500A Series, and the RSA600A Series. The instruments 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.

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.

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.

PLAYBACK_GetReplayComplete	Determine if a file being replayed has reached the end of the file contents.
Declaration:	ReturnStatus PLAYBACK_GetReplayComplete(bool * complete);
Parameters:	
complete:	Pointer to a boolean. True indicates file playback has completed. False indicates it has not completed. Note that in loop back mode, a file will never report true from a call to PLAYBACK_GetReplayComplete().
Return Values:	
noError:	The operation completed successfully.

Power functions

POWER_GetStatus

RSA500A Series only.

Queries the device power and battery status information.

Declaration:

ReturnStatus POWER_GetStatus(POWER_INFO* powerInfo);

Parameters:

powerInfo:

Pointer to a POWER_INFO struct. On return, the structure contains the current power and battery status information. See Additional Detail below for structure content.

Return Values:

noError:

The status has been successfully queried.

errorMonitoringNotSupported:

The device does not support battery monitoring.

Additional Detail:

POWER_INFO structure content:

externalPowerPresent (boolean):

True indicates an external power supply is connected. False indicates no external power supply is connected.

batteryPresent (boolean):

True indicates a battery is installed in the device. False indicates no battery installed. If batteryPresent is false, the following battery-related status indicators are invalid and should be ignored.

batteryChargeLevel (double):

Indicates battery charge level in percent (0.0=fully discharged, 100.0=fully charged).

batteryOverTemperature (boolean):

During charge, the over temp alarm can be set if the pack exceeds 45 °C. The charger should stop charging when the alarm is set. If charging doesn't stop, the pack will open a resettable protection FET.

During discharge, the over temp alarm will set if the pack exceeds 60 °C. The pack will set the alarm bit, but if the temperature doesn't decrease, the pack will open a resettable protection FET and shut down the device.

batteryHardwareError (boolean):

True indicates the battery controller has detected an error in the battery hardware. False indicates the battery hardware is operating normally.

RSA600A Series devices can also return a result from this function. However, since they do not have an internal battery, they will always report the following status:

externalPowerPresent = true

batteryPresent = false

Spectrum functions

SPECTRUM_AcquireTrace	Initiates a spectrum trace acquisition
Declaration:	ReturnStatus SPECTRUM_AcquireTrace();
Return Values:	
noError:	The function has completed successfully.
errorNotConnected:	The device is not connected.
Additional Detail:	Executing this function initiates a spectrum trace acquisition. Before calling this function, all acquisition parameters must be set to valid states. These include Center Frequency, Reference Level, any desired Trigger conditions, and the SPECTRUM configuration settings.

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. True indicates the spectrum measurement is enabled. False indicates it is disabled.
Return Values:	
noError:	The enable status has been successfully queried.

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	– – [1]	– – [1]
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

The maximum span is device dependent.

Return Values:

noError: The limits have been successfully queried.

[1] The maximum span is device dependent.

SPECTRUM_GetSettings — Queries the spectrum settings.

Declaration: ReturnStatus SPECTRUM_GetSettings(Spectrum_Settings *settings);

Parameters:

settings: Pointer to Spectrum settings.
Returns the current settings with the following content:

Item	Description
double span	Span measured in Hz
double rbw	Resolution bandwidth measured in Hz
bool enableVBW	Enables or disables VBW
double vbw	Video bandwidth measured in Hz
int traceLength	Number of trace points
SpectrumWindows window	Windowing method used for the transform
SpectrumVerticalUnits verticalUnit	Vertical units
double actualStartFreq	Actual start frequency in Hz
double actualStopFreq	Actual stop frequency in Hz
double actualFreqStepSize	Actual frequency step size in Hz
double actualRBW	Actual RBW in Hz
double actualVBW	Not used.
int actualNumIQSamples	Actual number of IQ samples used for transform

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.

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. The trace data is in the unit of verticalUnit specified in the Spectrum_Settings structure.

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
adcDataLost	0x20

Return Values:

noError: The trace information has been successfully queried.

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 a 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.

SPECTRUM_SetDefault	Sets the spectrum settings to default settings.
Declaration:	ReturnStatus SPECTRUM_SetDefault();
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

SPECTRUM_SetEnable	Sets the enable status.
Declaration:	ReturnStatus SPECTRUM_SetEnable(bool enable);
Parameters:	
enable:	Enable or disable Spectrum measurement. True enables the spectrum measurement. False disables it.
Return Values:	
noError:	The function has completed successfully.
Additional Detail:	When the spectrum measurement is enabled, the IQ acquisition is disabled.

SPECTRUM_SetSettings — Sets the spectrum settings.

Declaration: ReturnStatus SPECTRUM_SetSettings(Spectrum_Settings settings);

Parameters:

settings: Spectrum settings.

Spectrum_Settings structure content:

Spectrum_Settings	Value
double span	Span measured in Hz
double rbw	Resolution bandwidth measured in Hz
bool enableVBW	Enables or disables VBW
double vbw	Video bandwidth measured in Hz
int traceLength	Number of trace points
SpectrumWindows window	Windowing method used for the transform
SpectrumVerticalUnits verticalUnit	Vertical units

SpectrumWindows	Value
SpectrumWindow_Kaiser	0
SpectrumWindow_Mil6dB	1
SpectrumWindow_BlackmanHarris	2
SpectrumWindow_Rectangular	3
SpectrumWindow_FlatTop	4
SpectrumWindow_Hann	5

SpectrumVerticalUnits	Value
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.

SPECTRUM_SetTraceType — Sets the trace settings.

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

Parameters:

trace: One of the spectrum traces.

Spectrum Traces	Value
SpectrumTrace1	0
SpectrumTrace2	1
SpectrumTrace3	2

enable: Enable trace output.

True enables trace output. False disables it.

detector: Detector type.

Spectrum Detectors	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_WaitForTraceReady — Waits for the spectrum trace data to be ready to be queried.

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

Parameters:

timeoutMsec: Timeout value in msec.

ready: Pointer to a bool.

True indicates the spectrum trace data is ready for acquisition. False indicates the data is not ready and the timeout value is exceeded.

Return Values:

noError: The trace data ready status has been successfully queried.

Time functions

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.

REFTIME_SetReferenceTime	Sets the RSA API time system association.
Declaration:	ReturnStatus REFTIME_SetReferenceTime(time_t refTimeSec, uint64_t refTimeNsec, uint64_t refTimestamp);
Parameters:	
efTimeSec:	Seconds component of the time system wall-clock reference time. Format is number of integer seconds elapsed since midnight (00:00:00), Jan 1, 1970, UTC.
refTimeNsec:	Nanosecond component of time system wall-clock reference time. Format is number of integer nanoseconds within the second specified in refTimeSec.
refTimestamp:	Timestamp counter component of time system reference time. Format is the integer timestamp count corresponding to the time specified by refTimeSec+refTimeNsec.
Return Values:	
noError:	The function completed successfully.
Additional Detail:	This function sets the RSA API time system association between a "wall-clock" time value and the internal timestamp counter. The wall-clock time is composed of refTimeSec+refTimeNsec, which specify a UTC time to nanosecond precision. refTimeSec represents the integer number of seconds elapsed since midnight (00:00:00), Jan 1, 1970, UTC and refTimeNsec represents a nanosecond offset within the refTimeSec second. refTimestamp represents the state of the device's internal timestamp counter at the wall-clock time specified by refTimeSec+refTimeNsec. At device connection, the API automatically initializes the time system using this function to associate current Windows system time with the current value of the timestamp counter. This setting does not give high-accuracy time alignment due to the uncertainty in Windows system time, but provides a basic time/timestamp association. The REFTIME functions then use this association for time calculations. To re-initialize the time system this way some time after connection, call the function with all arguments equal to 0. If a higher-precision time reference is available, such as GPS or GNSS receiver with 1PPS pulse output, or other precisely known time event, the API time system can be aligned to it by capturing the timestamp count of the event using the External trigger input. Then the timestamp value and corresponding wall-time value (sec+nsec) are associated using this function. This provides timestamp accuracy as good as the accuracy of the time + event alignment.

REFTIME_GetReferenceTime	Queries the RSA API system time association.
Declaration:	ReturnStatus REFTIME_GetReferenceTime(time_t* refTimeSec, uint64_t* refTimeNsec, uint64_t* refTimestamp);
Parameters:	
refTimeSec:	Pointer to time_t. Returns seconds component of reference time association. (Input NULL argument value if return value is not desired).
refTimeNsec:	Pointer to uint64_t. Returns nanoseconds component of reference time association. (Input NULL argument value if return value not desired).
refTimestamp:	Pointer to uint64_t. Returns counter timestamp of reference time association. (Input NULL argument value if return value not desired).
Return Values:	
noError:	The function completed successfully.
Additional Detail:	The refTimeSec value is the number of seconds elapsed since midnight (00:00:00), Jan 1, 1970, UTC.
	The refTimeNsec value is the number of nanoseconds offset into the refTimeSec second. refTimestamp is the timestamp counter value. These values are initially set automatically by the API system using Windows sytstem time, but may be modified by REFTIME_SetReferenceTime() function if a better reference time source is available.

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:	Pointer to time_t. Returns seconds component of current time. (Input NULL argument value if return value not desired).
o_timeNsec:	Pointer to uint64_t. Returns nanoseconds component of current time. (Input NULL argument value if return value not desired).
o_timestamp:	Pointer 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:	Pointer to a double. Returns seconds since the internal Reference time/timestamp association was last set.
Return Values:	
noError:	The query was successful.

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:	Pointer to time_t. Returns time value seconds component.
o_timeNsec:	Pointer 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:	Pointer 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.

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:	Pointer 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.

Tracking generator functions

TRKGEN_GetEnable	RSA500A Series and RSA600A Series only.
	This function queries the tracking generator enabled status.
Declaration:	ReturnStatus TRKGEN_GetEnable(bool *enable);
Parameters:	
enable:	Pointer to a bool. Stores the enable status of the tracking generator hardware.
	True indicates the tracking generator is enabled and powered on. False indicates the tracking generator is disabled and powered off.
Return Values:	
noError:	The enable status has been successfully queried.

TRKGEN_GetHwInstalled	RSA500A Series and RSA600A Series only.
	This function queries the hardware present status.
Declaration:	ReturnStatus TRKGEN_GetHwInstalled(bool *installed)
Parameters:	
enable:	Pointer to a bool. Stores the installed status of the tracking generator hardware.
	True indicates the tracking generator hardware is installed in the unit. False indicates the tracking generator is not installed.
Return Values:	
noError:	The installed status has been successfully queried.

TRKGEN_GetOutputLevel	RSA500A Series and RSA600A Series only.
	This function queries the output level of the tracking generator.
Declaration:	ReturnStatus TRKGEN_SetOutputLevel(double *level);
Parameters:	
level:	Pointer to a double. Returns the value of the tracking generator output level in dBm.
	Range: 43 dBm to –3dBm
Return Values:	
noError:	The output level was successfully queried.

TRKGEN_SetEnable	RSA500A Series and RSA600A Series only.
	This function sets the tracking generator enable status.
Declaration:	ReturnStatus TRKGEN_SetEnable(bool enable);
Parameters:	
enable:	Enable or disable the tracking generator and associated circuitry.
	True indicates the tracking generator and associated circuitry is enabled. False indicates the tracking generator is disabled and powered off.
Return Values:	
noError:	The enable status has been successfully set.

TRKGEN_SetOutputLevel	RSA500A Series and RSA600A Series only. This function sets the output power of the tracking generator in dBm.
Declaration:	ReturnStatus TRKGEN_SetOutputLevel(double level);
Parameters:	
level:	Requested output level of tracking generator in dBm. Range: -43 dBm to –3 dBm.
Return Values:	
noError:	The requested value was accepted.
Additional Detail:	The tracking generator output should be set prior to setting the center frequency. See the CONFIG_SetCenterFreq and CONFIG_Preset functions to set the center frequency.

Trigger functions

TRIG_ForceTrigger	Forces the device to trigger.
Declaration:	ReturnStatus TRIG_ForceTrigger();
Return Values:	
noError:	The operation completed successfully.

TRIG_GetIFPowerTriggerLevel	Queries the trigger power level.
Declaration:	ReturnStatus TRIG_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.

TRIG_GetTriggerMode	Queries the trigger mode.
Declaration:	ReturnStatus TRIG_GetTriggerMode(TriggerMode* mode);
Parameters:	
mode:	Pointer to 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.

TRIG_GetTriggerPositionPercent	Queries the trigger position percent.
Declaration:	ReturnStatus TRIG_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 trigger position setting only affects IQ Block and Spectrum acquisitions.

TRIG_GetTriggerSource	Queries the trigger source.
Declaration:	ReturnStatus TRIG_GetTriggerSource(TriggerSource *source);
Parameters:	
source:	Pointer to 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.

TRIG_GetTriggerTransition	Queries the current trigger transition mode.
Declaration:	ReturnStatus TRIG_GetTriggerTransition(TriggerTransition *transition);
Parameters:	
transition:	Pointer to 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.

TRIG_SetIFPowerTriggerLevel	Sets the IF power detection level.
Declaration:	ReturnStatus TRIG_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.

TRIG_SetTriggerMode	Sets the trigger mode.
Declaration:	ReturnStatus TRIG_SetTriggerMode(TriggerMode mode);
Parameters:	
mode:	This variable describes the trigger mode. It can be in either freeRun or triggered mode.

Trigger Mode	Value
freeRun	0
Untriggered	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.

TRIG_SetTriggerPositionPercent	Sets the trigger position percentage.
Declaration:	ReturnStatus TRIG_SetTriggerPositionPercent(double trigPosPercent);
Parameters:	
trigPosPercent:	Trigger position percentage. Range: 1% to 99%. Default setting is 50%.
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. The trigger position setting only affects IQ Block and Spectrum acquisitions.

TRIG_SetTriggerSource	Sets the trigger source.
Declaration:	ReturnStatus TRIG_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.

TRIG_SetTriggerTransition	Sets the trigger transition detection.
Declaration:	ReturnStatus TRIG_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.

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.

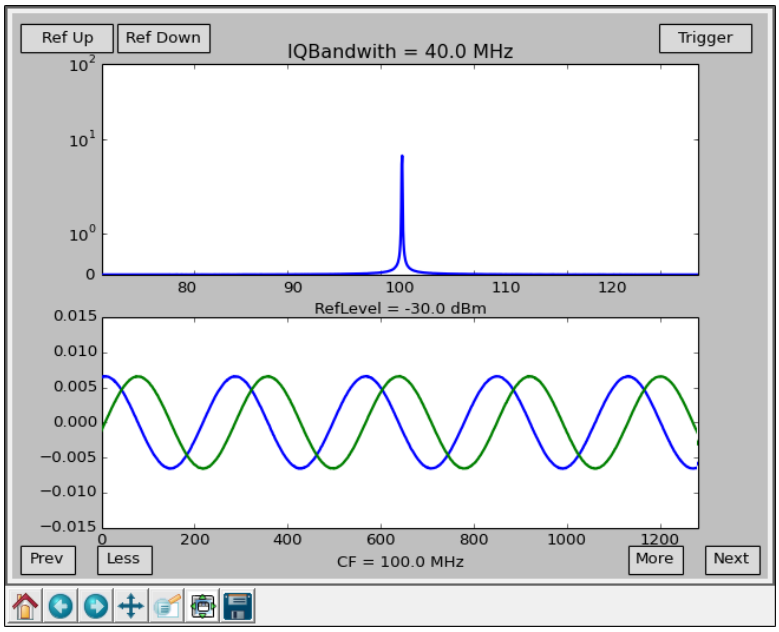
The program was written with Python 2.7. To use this example, the NumPy, Matplotlib, Dateutil, and Pyparsing libraries need to be installed along with Python 2.7.

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.



Programming 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.

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.

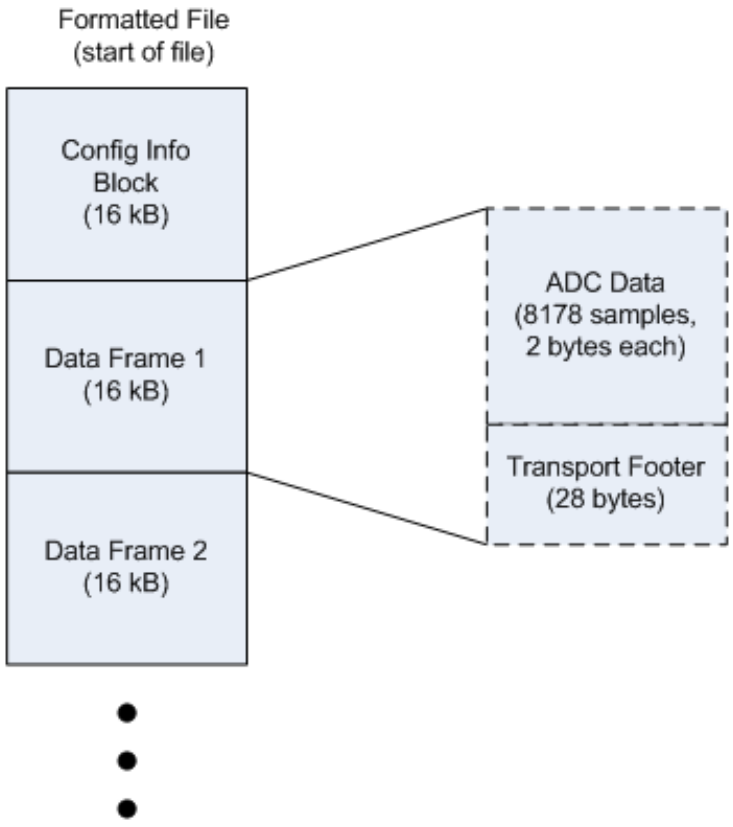


Figure 1: Formatted data file

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 4: (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.W (V=1 byte, W=2 bytes, 1.1.0 latest)
	4	API SW Version	V.V.W (V=1 byte, W=2 bytes, fill unused with 0)
	4	FX3 FW Version	V.V.W (V=1 byte, W=2 bytes, fill unused with 0)
	4	FPGA FW Version	V.V.W (V=1 byte, W=2 bytes, fill unused with 0)
	64	Device S/N	Serial Number String (fill with 0x00 to end)
	32	Device Nomenclature	The model number string of the device which stored the data, such as "RSA507A", up to 31 characters and 0x00 padded to the end. Introduced in V1.1 of the file.
	(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 4: (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)
	7 * 4	Ref Time – UTC time.	The same time as Wall time expressed as Universal Coordinated Time. Added with V1.1 of the R3F file spec. Ref Time: (7 values, each int32) Year, Month, Date, Hour, Minute, Second, Nanoseconds (Note: Nanoseconds is set to 0 initially)
	(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

Table 4: (Category) specifications (cont.)

Offset (Byte)	Size (Bytes)	Content	Description
	8	Signal Path delay	Seconds (double)
	(to EOB)	Reserved	(filled with 0x00)
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)

RSA API version compatibility

This document supports version 2 of the RSA API. API version 2 added prefix names to most functions and also provides additional functionality over API version 1. Some API version 1 function are not supported in API version 2.

Although most all of the API version 1 function calls work, it's recommended to use the API version 2 function calls.

API version 1 functions are accessed from "RSA300API.h" and API version 2 functions are accessed from "RSA_API.h". You should not intermix version 1 function calls with version 2 function calls in a source code file.

NOTE. *API version 1 functions are deprecated and will eventually be removed (not supported).*

The ***RSA_API version_compatibility.xlsx*** attachment is a compatibility spreadsheet to map the old version 1 function names to the new version 2 function names. The spreadsheet also indicates if the arguments or returns were modified in addition to changing the name.

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.

Index

F

G

I

P

S

T