Reference Guide
74-0044

ThinkRF RTSA C++ API v1.6.0

Reference Guide

Mon Dec 7 2020

thinkRF
monitor. detect. analyze

Contents

1 Introduction 1

1.1 Software and System Requirements . . . 1

1.1.1 Supported RTSA Product List . . . 1

1.2 How to Use the API . . . 2

1.2.1 Using the Library Directly . . . 2

1.2.2 Using the DLL . . . 2

1.2.3 Quick Start with Examples . . . 2

1.3 Contact Us . . . 2

2 RTSA C++ DLL API 3

2.1 Connection and Command Related Functions . . . 3

2.1.1 Connection . . . 3

2.1.2 Direct SCPI Command Related . . . 3

2.2 GNSS Functions . . . 4

2.2.1 GNSS Status and Settings . . . 4

2.2.2 GNSS Context Packets Retrieval . . . 4

2.3 VRT Data Capture Functions . . . 4

2.4 Trace Capture Functions . . . 4

2.5 Stream Capture Functions . . . 5

2.6 Manual Sweep Capture Functions . . . 5

2.7 Data Processing Functions . . . 5

2.7.1 FFT and Spectral Data Computation . . . 5

2.7.2 Power Spectral Density (PSD) Computation . . . 5

2.8 Sweep-Device Functions . . . 6

2.9 Utility Functions . . . 6

2.10 Known Issues . . . 6

3 Data Structure Index 7

3.1 Data Structures . . . 7

4 File Index 9

4.1 File List . . . 9

5 Data Structure Documentation **11**
5.1 vrt_context Struct Reference . . . 11
5.1.1 Detailed Description . . . 12
5.1.2 Field Documentation . . . 12
5.2 vrt_gnss_geolocn Struct Reference . . . 14
5.2.1 Detailed Description . . . 14
5.2.2 Field Documentation . . . 14
5.3 vrt_header Struct Reference . . . 17
5.3.1 Detailed Description . . . 17
5.3.2 Field Documentation . . . 17

6 File Documentation **19**
6.1 doxygen/api_doxy_doc.txt File Reference . . . 19
6.1.1 Detailed Description . . . 19
6.2 wsaInterface.h File Reference . . . 19
6.2.1 Detailed Description . . . 21
6.2.2 Function Documentation . . . 21

7 Example Documentation **53**
7.1 largeBlockCaptureWithPowerFns.cpp . . . 53
7.2 largeBlockCaptureWithPowerFnsAndDD.cpp . . . 56
7.3 simpleGNSSPacketsCaptureExample.cpp . . . 60
7.4 simpleHIF.cpp . . . 65
7.5 simplePSDFunctionsExample.cpp . . . 66
7.6 simpleRead.cpp . . . 70
7.7 simpleReadHDR.cpp . . . 72
7.8 streamExample.cpp . . . 74
7.9 sweepDeviceCalculateChannelPower.cpp . . . 77
7.10 sweepDeviceCalculateOccupiedBandwidth.cpp . . . 78
7.11 sweepDeviceCaptureSweepSpectrum.cpp . . . 79
7.12 sweepDevicePeakFind.cpp . . . 81

Index **83**

Chapter 1

Introduction

The ThinkRF Real-Time Spectrum Analyzer (RTSA) has the performance of traditional high-end lab spectrum analyzers at a fraction of the cost, size, weight and power consumption and is designed for distributed or remote deployment.

ThinkRF provides a C++ API, which is a library with high level interfaces to an RTSA device. It abstracts away the actual low level connection, controls or commands to the RTSA, and data extraction.

The C++ programming language is widely used for application development. By controlling the ThinkRF RTSA via the C++ API, simple scripts can be used for data acquisition as well as for application development in complex systems. A rich set of examples are also included with the API. See the **Examples** directory included with the API.

The API is to be used in conjunction with your RTSA's Programmer's Guide, which provides explanation for the SCPI commands, the VRT protocol supported by your RTSA and the important functional overview of your RTSA.

1.1 Software and System Requirements

To use the ThinkRF C++ API in your application, the following software is required:

- Windows 7/8/10 32-bit/64-bit operating system;
- Visual Studio 2010 Express or higher to directly run the API and/or examples;
- Access to an RTSA product. If don't have a unit, you may access also to one of ThinkRF's evaluation units on the internet from www.thinkrf.com/demo.

1.1.1 Supported RTSA Product List

C++ DLL API v1.5.0 or higher supports the following RTSA products and their associated models (408, 408P, 418, 427):

- R5500
- R5550
- R5700
- R5750
- Legacy products WSA5000

1.2 How to Use the API

1.2.1 Using the Library Directly

To use the API library directly in your C++ project, you need to include the header file, wsaInterface.h, in any files that will use any of the API functions and proper link to the wsaInterface.lib. The library or DLL also depends on others C API's ∗.h files provided in the **include** folder (they are not documented in this documentation).

1.2.2 Using the DLL

A DLL of the API has been provided for any application development that could not use the library directly, such as MATLAB, LabVIEW, C#, etc. Refers to your tool's help or documentation for how to include the DLL.

ThinkRF provides also MATLAB & LabVIEW APIs, which use this DLL, for your convenience. See `https↩://www.thinkrf.com/software-apis/`

1.2.3 Quick Start with Examples

The easiest way to get started with the API is to take a look at the examples provided in the "Examples" folder included with the API release package. The examples include different capture modes (block, stream and sweep-device) along with SCPI controls and data processing where applicable.

To use the examples, create a project in the Visual Studio and modify the project's Properties to point to the "include" path for "C/C++" and to wsaInterface.lib path for "Linker > Input". Only one example could be included in the project at a time.

After compiling successfully, the usage would be to run "<project_name>.exe" then follows the prompt of the executable to provide the device's IP and any other parameters as needed.

1.3 Contact Us

ThinkRF Support website provides online documents for resolving technical issues with ThinkRF products at `http://www.thinkrf.com/resources`.
For all customers who hold a valid end-user license, ThinkRF provides technical assistance 9 AM to 5 PM Eastern Time, Monday to Friday. Contact us at `http://www.thinkrf.com/support/` or by calling +1.613.369.5104.

Chapter 2

RTSA C++ DLL API

The RTSA C++ API is mostly a high-level wrapper of ThinkRF's C API, simplifying/grouping the multitudes of C API functions into more abstracted high level functions, making them easier to use for integration. The API is provided in a DLL as well as a lib library, allowing for use in programming languages that support them.

The API release package comprised of:

- wsaInterface.h – the header file with all the C++ API functions
- “include” folder – contains all of the C API header files needed for the C++ API
- “x32”/”x64” folders – contains compiled windows libraries, ∗.lib and .dll, for 32-bit/64-bit Windows OS, ∗ - spectively
- “Examples” folder – contains different ∗.cpp examples, serve to illustrative how to use the C++ API. This section lists the API functions provided in wsaInterface.h into subsections for easy navigation.

2.1 Connection and Command Related Functions

This section contains functions related to RTSA connection and SCPI command high level functions.

2.1.1 Connection

- wsaConnect()
- wsaDisconnect()

2.1.2 Direct SCPI Command Related

Use the following functions for any SCPI commands listed in the RTSA's Programmer's Guide.

- wsaSetSCPI()
- wsaGetSCPI_s() - Response output as a function's parameter
- wsaGetSCPI() - Response output as a return of the function

Some SCPI command wrappers:

- wsaSetAttenuation() - Use for trace/stream capture only, not applicable for sweep

- wsaGetAttenuation() - Use for trace/stream capture only, not applicable for sweep
- wsaSetReferencePLL()
- wsaGetReferencePLL()

2.2 GNSS Functions

This section contains functions applied only for RTSA devices come with a GNSS module.

2.2.1 GNSS Status and Settings

- wsaGNSSEnable()
- wsaGetGNSSStatus()
- wsaGetGNSSPosition()
- wsaGetGNSSFixSource()
- wsaSetGNSSAntennaDelay()
- wsaGetGNSSAntennaDelay()
- wsaSetGNSSConstellation()
- wsaGetGNSSConstellation()
- wsaSetReferencePPS()
- wsaGetReferencePPS()

2.2.2 GNSS Context Packets Retrieval

- wsaReadGNSSContext()

2.3 VRT Data Capture Functions

The functions in this section are used for reading out a VRT packet (IF data and/or context).

- wsaReadData() - Read only a VRT IF data packet each time
- wsaReadVRTData() - Read a VRT packet each time

2.4 Trace Capture Functions

This section consists of functions used for trace (block) data capture.

- wsaArmTraceCapture()
- wsaCaptureTraceAndContext()
- wsaCaptureSpectrumAndContext()

2.5 Stream Capture Functions

For stream capture, use the following functions and follow by functions from Data Processing Functions section as needed.

- wsaStreamStartWithID()
- wsaStreamStop()
- wsaCaptureStreamTraceAndContext()

2.6 Manual Sweep Capture Functions

For manual sweep capture, only time domain data is returned. These functions are useful for when wanting to do discontinous sweep bands scanning. If sweeping of a continous range is desired, consider using functions in Sweep-Device Functions instead.
The following functions are provided and could be follow by functions from Data Processing Functions section as needed.

- wsaManualSweepStart()
- wsaManualSweepStop()
- wsaCaptureManualSweepTraceAndContext()

2.7 Data Processing Functions

2.7.1 FFT and Spectral Data Computation

Once I/Q data are captured, whether as trace or sweep-device, these functions could be called. The spectral data output (power spectral density (PSD)) in dBm could be used by other power functions for further processing.

- wsaGetFFTSize()
- wsaComputeFFT()
- wsaComputeFFTWithCorrOptions()

2.7.2 Power Spectral Density (PSD) Computation

- wsaComputePSDUsableData()
- wsaPSDPeakFind()
- wsaComputePSDChannelPower()
- wsaComputePSDOccupiedBandwidth()

2.8 Sweep-Device Functions

Functions in this sections apply to sweep-device capture and processing only. **Warning:** When performing a sweep-device over a very large span (typically 7GHz or higher), usage of small RBWs under 10 Hz could result in a memory allocation failure as your computer might not have enough RAM space to handle the allocation.

- wsaGetSweepSize_s()
- wsaCaptureSpectrum()
- wsaPeakFind()
- wsaChannelPower()
- wsaOccupiedBandwidth()

2.9 Utility Functions

This section contains additional useful API functions.

- wsaGetRFESpan() - Determines the bandwidth, center, start and stop frequencies of the RFE mode.
- wsaErrorMessage_s() - Output is returned through a parameter
- wsaErrorMessage() - Output is returned as a function return
- getBuildInfo() - Get the version and date of the API DLL built

2.10 Known Issues

- Usage of some sweep's RBWs (such as 2kHz) for wsaCaptureSpectrum() could result in a slow sweep (when compared to 1kHz) due to the FFT function usage that is not optimized for non-power of 2 FFT length. Hence, when RBW values that give raise to FFT length at or closer to power of 2, the sweep data processing will be more optimal.
- No Windows Properties information is available for the DLL.

Chapter 3

Data Structure Index

3.1 Data Structures

Here are the data structures with brief descriptions:

vrt_context
This structure holds some of the VRT context packets's information. See RTSA product's Programmer's Guide for more information 11
vrt_gnss_geolocn
This structure to hold VRT Formatted GNSS Geolocation. See R57x0 Programmer's Guide for more information 14
vrt_header
This structure holds some of the VRT's header information as well as trailer information of the IF data. See RTSA product's Programmer's Guide for more information 17

Chapter 4

File Index

4.1 File List

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

wsaInterface.h
The API header file with functions and their definition 19

Chapter 5

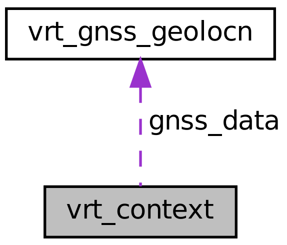
Data Structure Documentation

5.1 vrt_context Struct Reference

This structure holds some of the VRT context packets's information. See RTSA product's Programmer's Guide for more information.

`#include <wsaInterface.h>`

Collaboration diagram for vrt_context:



Data Fields

- uint32_t indicator_field
- uint64_t bandwidth
- uint64_t cent_freq
- int64_t freq_offset
- double gain_if
- double gain_rf
- int16_t reference_level
- uint8_t spectral_inversion
- uint8_t sample_loss_indicator
- uint8_t has_gnss_data
- struct vrt_gnss_geolocn gnss_data

5.1.1 Detailed Description

This structure holds some of the VRT context packets's information. See RTSA product's Programmer's Guide for more information.

Examples

largeBlockCaptureWithPowerFns.cpp, largeBlockCaptureWithPowerFnsAndDD.cpp, simpleGNSSPacketsCaptureExample.cpp, and streamExample.cpp.

5.1.2 Field Documentation

indicator_field

`vrt_context::indicator_field`

The context's indicator field, identifying which context info is in the VRT context packet

bandwidth

`vrt_context::bandwidth`

The 'full operating' bandwidth of the IF spectral data under the current settings, in Hz (ie. this is not the usable bandwidth, which has the filter roll-off removed)

Examples

streamExample.cpp.

cent_freq

`vrt_context::cent_freq`

The center frequency of the RTSA, in Hz

Examples

streamExample.cpp.

freq_offset

`vrt_context::freq_offset`

The frequency offset (shift) applied to the RTSA, in Hz

gain_if

`vrt_context::gain_if`

The IF gain component of the RTSA, in dB

gain_rf

`vrt_context::gain_rf`

The RF gain component of the RTSA, in dB

reference_level

`vrt_context::reference_level`

The reference level providing a power level reference so that the magnitude of the received data can be calculated by a user.

Examples

streamExample.cpp.

spectral_inversion

`vrt_context::spectral_inversion`

The spectral inversion state of the I-data, needed for when user computes the frequency mapping of the spectral data in SH/SHN mode without using the provided API functions.

Examples

streamExample.cpp.

sample_loss_indicator

`vrt_context::sample_loss_indicator`

An indicator indicating if there's any data lost occurred between this packet and the one just before it (no data loss within the packet).

Examples

streamExample.cpp.

has_gnss_data

`vrt_context::has_gnss_data`

An indicator for when gnss_data has new valid data (1) or no (0).

Examples

largeBlockCaptureWithPowerFns.cpp, largeBlockCaptureWithPowerFnsAndDD.cpp, and streamExample.cpp.

gnss_data

```
vrt_context::gnss_data
```

A vrt_gnss_geolocn struct storing the GNSS data

Examples

largeBlockCaptureWithPowerFns.cpp, largeBlockCaptureWithPowerFnsAndDD.cpp, and streamExample.cpp.

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

- wsaInterface.h

5.2 vrt_gnss_geolocn Struct Reference

This structure to hold VRT Formatted GNSS Geolocation. See R57x0 Programmer's Guide for more information.

```
#include <wsaInterface.h>
```

Data Fields

- uint32_t mfr_oui
- uint32_t posfix_sec
- uint64_t posfix_psec
- float latitude
- float longitude
- float altitude
- float speed_over_gnd
- float heading_angle
- float track_angle
- float magnetic_var

5.2.1 Detailed Description

This structure to hold VRT Formatted GNSS Geolocation. See R57x0 Programmer's Guide for more information.

Examples

simpleGNSSPacketsCaptureExample.cpp.

5.2.2 Field Documentation

mfr_oui

```
vrt_gnss_geolocn::mfr_oui
```

Manufacturer OUI of the GNSS.

Examples

largeBlockCaptureWithPowerFns.cpp, largeBlockCaptureWithPowerFnsAndDD.cpp, and streamExample.cpp.

posfix_sec

`vrt_gnss_geolocn::posfix_sec`

The timestamp in second of the position fix.

Examples

largeBlockCaptureWithPowerFns.cpp, largeBlockCaptureWithPowerFnsAndDD.cpp, and streamExample.cpp.

posfix_psec

`vrt_gnss_geolocn::posfix_psec`

The fractional timestamp in picosecond of the position fix.

Examples

largeBlockCaptureWithPowerFns.cpp, largeBlockCaptureWithPowerFnsAndDD.cpp, and streamExample.cpp.

latitude

`vrt_gnss_geolocn::latitude`

The latitude of the geolocation, with value ranging from -90.0 (South) to +90.0 (North) degrees.

Examples

largeBlockCaptureWithPowerFns.cpp, largeBlockCaptureWithPowerFnsAndDD.cpp, simpleGNSSPacketsCaptureExample.cpp, and streamExample.cpp.

longitude

`vrt_gnss_geolocn::longitude`

The longtitude of the geolocation, with value ranging from -180.0 (West) to +180.0 (East) degrees.

Examples

largeBlockCaptureWithPowerFns.cpp, largeBlockCaptureWithPowerFnsAndDD.cpp, simpleGNSSPacketsCaptureExample.cpp, and streamExample.cpp.

altitude

`vrt_gnss_geolocn::altitude`

The altitude of the geolocation, in meters

Examples

largeBlockCaptureWithPowerFns.cpp, largeBlockCaptureWithPowerFnsAndDD.cpp, simpleGNSSPacketsCaptureExample.cpp, and streamExample.cpp.

speed_over_gnd

`vrt_gnss_geolocn::speed_over_gnd`

The speed over ground of the geolocation, with value in the range of 0 to 65636 m/s and a resolution of 1.5e-5 m/s

Examples

largeBlockCaptureWithPowerFns.cpp, largeBlockCaptureWithPowerFnsAndDD.cpp, and streamExample.cpp.

heading_angle

`vrt_gnss_geolocn::heading_angle`

The heading angle of the GNSS, expresses the platform's orientation with respect to true North in decimal degrees. Its value ranges from 0.0 to +359.999999761582 degrees.

Examples

largeBlockCaptureWithPowerFns.cpp, largeBlockCaptureWithPowerFnsAndDD.cpp, and streamExample.cpp.

track_angle

`vrt_gnss_geolocn::track_angle`

The track angle of the GNSS, conveys the platform's direction of travel with respect to true North in decimal degrees. Its value ranges from 0.0 to +359.999999761582 degrees.

Examples

largeBlockCaptureWithPowerFns.cpp, largeBlockCaptureWithPowerFnsAndDD.cpp, and streamExample.cpp.

magnetic_var

`vrt_gnss_geolocn::magnetic_var`

The magnetic variation of the geolocation, expresses magnetic variation from true North in decimal degree, with value ranging from -180.0 (West) to +180.0 (East) degrees.

Examples

largeBlockCaptureWithPowerFns.cpp, largeBlockCaptureWithPowerFnsAndDD.cpp, and streamExample.cpp.

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

- wsaInterface.h

5.3 vrt_header Struct Reference

This structure holds some of the VRT's header information as well as trailer information of the IF data. See RTSA product's Programmer's Guide for more information.

```
#include <wsaInterface.h>
```

Data Fields

- uint8_t packet_type
- uint32_t stream_id
- uint32_t timestamp_sec
- uint64_t timestamp_psec

5.3.1 Detailed Description

This structure holds some of the VRT's header information as well as trailer information of the IF data. See RTSA product's Programmer's Guide for more information.

Examples

largeBlockCaptureWithPowerFns.cpp, largeBlockCaptureWithPowerFnsAndDD.cpp, simpleGNSSPacketsCaptureExample.cpp, and streamExample.cpp.

5.3.2 Field Documentation

packet_type

`vrt_header::packet_type`

The VRT packet types: IF data, context or extension

stream_id

`vrt_header::stream_id`

The VRT stream type: IF stream format (I16/Q16/I32), receiver, digitzer or extension context streams

Examples

streamExample.cpp.

timestamp_sec

`vrt_header::timestamp_sec`

The timestamp in seconds of the VRT packet, representing the number of seconds elapsed since 00:00 hours, Jan 1, 1970 UTC.

Examples

streamExample.cpp.

timestamp_psec

`vrt_header::timestamp_psec`

The fractional timestamp of the VRT packet, in picoseconds after the second

Examples

streamExample.cpp.

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

- wsaInterface.h

Chapter 6

File Documentation

6.1 doxygen/api_doxy_doc.txt File Reference

Provide overview and formatting information for the C++ API document using doxygen.

6.1.1 Detailed Description

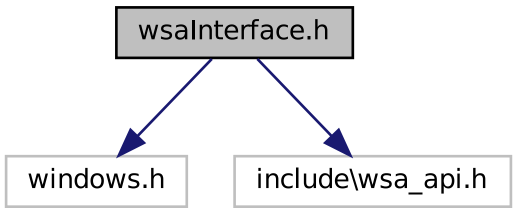
Provide overview and formatting information for the C++ API document using doxygen.

6.2 wsaInterface.h File Reference

The API header file with functions and their definition.

```
#include <windows.h>
#include "include\wsa_api.h"
```

Include dependency graph for wsaInterface.h:



Data Structures

- struct vrt_gnss_geolocn

    This structure to hold VRT Formatted GNSS Geolocation. See R57x0 Programmer's Guide for more information.

- struct vrt_header

  This structure holds some of the VRT's header information as well as trailer information of the IF data. See RTSA product's Programmer's Guide for more information.
- struct vrt_context

  This structure holds some of the VRT context packets's information. See RTSA product's Programmer's Guide for more information.

Functions

- int16_t wsaConnect (int64_t ∗wsa_handle, char ∗ip)
- int16_t wsaDisconnect (int64_t wsa_handle)
- int16_t wsaSetSCPI (int64_t wsa_handle, char ∗command)
- int16_t wsaGetSCPI_s (int64_t wsa_handle, char ∗command, char ∗response)
- char ∗ wsaGetSCPI (int64_t wsa_handle, char ∗command)
- int16_t wsaSetAttenuation (int64_t wsa_handle, int32_t att_value)
- int16_t wsaGetAttenuation (int64_t wsa_handle, int32_t ∗att_value)
- int16_t wsaSetReferencePLL (int64_t wsa_handle, char ∗ref_type)
- int16_t wsaGetReferencePLL (int64_t wsa_handle, char ∗ref_type)
- int16_t _stdcall wsaArmTraceCapture (int64_t wsa_handle, int32_t samples_per_packet, int32_t packets_↩per_block)
- int16_t wsaCaptureSpectrumAndContext (int64_t wsa_handle, int32_t samples_per_packet, int32_t packets↩_per_block, uint32_t timeout, struct vrt_context ∗vrt_context_data, struct vrt_header ∗header, float ∗spectral↩_data, uint32_t ∗spectral_size, int64_t ∗spectral_fstart, int64_t ∗spectral_fstop, uint32_t ∗spectral_bandwidth)
- int16_t wsaCaptureTraceAndContext (int64_t wsa_handle, int32_t samples_per_packet, int32_t packets_per↩_block, uint32_t timeout, struct vrt_context ∗vrt_context_data, struct vrt_header ∗header, int16_t ∗i16_data, int16_t ∗q16_data, int32_t ∗i32_data)
- int16_t wsaStreamStartWithID (int64_t wsa_handle, int32_t samples_per_packet, int32_t stream_start_id)
- int16_t wsaStreamStop (int64_t wsa_handle)
- int16_t wsaCaptureStreamTraceAndContext (int64_t wsa_handle, int32_t samples_per_packet, uint32_t time-out, struct vrt_context ∗vrt_context_data, struct vrt_header ∗header, uint32_t ∗stream_start_id, int16_t ∗i16↩_data, int16_t ∗q16_data, int32_t ∗i32_data)
- int16_t wsaManualSweepStart (int64_t wsa_handle, int64_t sweep_start_id)
- int16_t wsaManualSweepStop (int64_t wsa_handle)
- int16_t wsaCaptureManualSweepTraceAndContext (int64_t wsa_handle, int32_t samples_per_packet, int32↩_t packets_per_step, uint32_t timeout, struct vrt_context ∗vrt_context_data, struct vrt_header ∗header, uint32_t ∗sweep_start_id, int16_t ∗i16_data, int16_t ∗q16_data, int32_t ∗i32_data)
- int16_t wsaReadData (int64_t wsa_handle, int16_t ∗i16_data, int16_t ∗q16_data, int32_t ∗i32_data, uint32_t timeout, uint32_t ∗stream_id, uint8_t ∗spectral_inversion, int32_t samples_per_packet, uint32_t ∗timestamp↩_sec, uint64_t ∗timestamp_psec, int16_t ∗reference_level, uint64_t ∗bandwidth, uint64_t ∗center_frequency)
- int16_t wsaReadVRTData (int64_t wsa_handle, int32_t samples_per_packet, uint32_t timeout, struct vrt_header ∗header, int16_t ∗i16_data, int16_t ∗q16_data, int32_t ∗i32_data, uint8_t ∗spectral_inversion, struct vrt_context ∗vrt_context_data)
- int16_t wsaGetFFTSize (int32_t sample_size, uint32_t stream_id, int32_t ∗fft_size)
- int16_t wsaComputeFFTWithCorrOptions (int32_t sample_size, int32_t fft_size, uint32_t stream_id, int16↩_t reference_level, int8_t dc_offset_corr, int8_t iq_imbalance_corr, int16_t ∗i16_buffer, int16_t ∗q16_buffer, int32_t ∗i32_buffer, float ∗spectral_data)
- int16_t wsaComputeFFT (int32_t sample_size, int32_t fft_size, uint32_t stream_id, int16_t reference_level, uint8_t spectral_inversion, int16_t ∗i16_buffer, int16_t ∗q16_buffer, int32_t ∗i32_buffer, float ∗spectral_data)
- int16_t wsaPSDPeakFind (int64_t wsa_handle, char ∗rfe_mode, uint8_t spectral_inversion, uint32_t data_size, float ∗spectral_data, int64_t ∗peak_freq, float ∗peak_power)
- int16_t wsaComputePSDChannelPower (uint32_t start_bin, uint32_t stop_bin, uint32_t data_size, float ∗spectral_data, float ∗channel_power)

- int16_t wsaComputePSDOccupiedBandwidth (uint32_t rbw, uint32_t data_size, float ∗spectral_data, float occupied_percentage, uint64_t ∗occupied_bandwidth)
- int16_t wsaComputePSDUsableData (int64_t wsa_handle, char ∗rfe_mode, uint8_t spectral_inversion, float ∗spectral_data, uint32_t data_size, uint32_t ∗usable_bandwidth, int64_t ∗usable_fstart, int64_t ∗usable_fstop, float ∗usable_spectral_data, uint32_t ∗usable_data_size)
- int16_t wsaGetSweepSize (int64_t wsa_handle, uint64_t fstart, uint64_t fstop, uint32_t rbw, char ∗rfe_mode, int32_t attenuator, uint32_t ∗sweep_size)
- int16_t wsaGetSweepSize_s (int64_t wsa_handle, uint64_t fstart, uint64_t fstop, uint32_t rbw, char ∗rfe_mode, int32_t attenuator, uint64_t ∗fstart_actual, uint64_t ∗fstop_actual, uint32_t ∗sweep_size)
- int16_t wsaCaptureSpectrum (int64_t wsa_handle, uint64_t fstart, uint64_t fstop, uint32_t rbw, char ∗rfe_mode, int32_t attenuator, float ∗spectral_data)
- int16_t wsaPeakFind (int64_t wsa_handle, uint64_t fstart, uint64_t fstop, uint32_t rbw, char ∗rfe_mode, float ref_offset, int32_t attenuator, uint64_t ∗peak_freq, float ∗peak_power)
- int16_t wsaChannelPower (int64_t wsa_handle, uint64_t fstart, uint64_t fstop, uint32_t rbw, char ∗rfe_mode, float ref_offset, int32_t attenuator, float ∗channel_power)
- int16_t wsaOccupiedBandwidth (int64_t wsa_handle, uint64_t fstart, uint64_t fstop, uint32_t rbw, float occupied_percentage, char ∗rfe_mode, int32_t attenuator, uint64_t ∗occupied_bw)
- int16_t wsaSetReferencePPS (int64_t wsa_handle, char ∗pps_type)
- int16_t wsaGetReferencePPS (int64_t wsa_handle, char ∗pps_type)
- int16_t wsaGNSSEnable (int64_t wsa_handle, int32_t enable)
- int16_t wsaGetGNSSStatus (int64_t wsa_handle, uint8_t ∗status)
- int16_t wsaGetGNSSPosition (int64_t wsa_handle, float ∗latitude, float ∗longitude, float ∗altitude)
- int16_t wsaGetGNSSFixSource (int64_t wsa_handle, char ∗fix_source)
- int16_t wsaSetGNSSAntennaDelay (int64_t wsa_handle, int32_t delay)
- int16_t wsaGetGNSSAntennaDelay (int64_t wsa_handle, int32_t ∗delay)
- int16_t wsaSetGNSSConstellation (int64_t wsa_handle, char ∗constel1, char ∗constel2)
- int16_t wsaGetGNSSConstellation (int64_t wsa_handle, char ∗constellations)
- int16_t wsaReadGNSSContext (int64_t wsa_handle, uint32_t timeout, int32_t ∗new_gnss_data, struct vrt_gnss_geolocn ∗gnss_data)
- int16_t wsaGetRFESpan (int64_t wsa_handle, char ∗rfe_mode, uint32_t ∗span_bw, int64_t ∗span_center, int64_t ∗span_fstart, int64_t ∗span_fstop)
- void wsaErrorMessage_s (int16_t error_code, char ∗error_msg)
- char ∗ wsaErrorMessage (int16_t error_code)
- void getBuildInfo (char ∗build_info)

6.2.1 Detailed Description

The API header file with functions and their definition.

Copyright

6.2.2 Function Documentation

wsaConnect()

```
int16_t wsaConnect (
            int64_t * wsa_handle,
            char * ip )
```

Establishes a connection to the RTSA. At success, the handle remains open for future access by other functions.

Note: wsaDisconnect() must be called at the end of the program for each wsaConnect() successfully made.

Parameters

in	*wsa_handle*	- A 64-bit integer pointer which holds the device info.
in	*ip*	- A char pointer to the RTSA device's IP

Returns

0 on success, or a negative number on error

Examples

largeBlockCaptureWithPowerFns.cpp, largeBlockCaptureWithPowerFnsAndDD.cpp, simpleGNSSPacketsCaptureExample.cpp, simpleHIF.cpp, simplePSDFunctionsExample.cpp, simpleRead.cpp, simpleReadHDR.cpp, streamExample.cpp, sweepDeviceCalculateChannelPower.cpp, sweepDeviceCalculateOccupiedBandwidth.cpp, sweepDeviceCaptureSweepSpectrum and sweepDevicePeakFind.cpp.

wsaDisconnect()

```
int16_t wsaDisconnect (
            int64_t wsa_handle )
```

Closes an already established connection to the RTSA. This function must be called at the end of the program for each wsaConnect() made.

Parameters

in	*wsa_handle*	- A 64-bit integer pointer which holds the device info

Returns

0 on success, or a negative number on error

Examples

largeBlockCaptureWithPowerFns.cpp, largeBlockCaptureWithPowerFnsAndDD.cpp, simpleGNSSPacketsCaptureExample.cpp, simpleHIF.cpp, simplePSDFunctionsExample.cpp, simpleRead.cpp, simpleReadHDR.cpp, streamExample.cpp, sweepDeviceCalculateChannelPower.cpp, sweepDeviceCalculateOccupiedBandwidth.cpp, sweepDeviceCaptureSweepSpectrum and sweepDevicePeakFind.cpp.

wsaSetSCPI()

```
int16_t wsaSetSCPI (
            int64_t wsa_handle,
            char * command )
```

Sends a SCPI command to the device. The list of all the supported SCPI commands and their definition can be found in the RTSA Programmer's Guide.

Parameters

in	*wsa_handle*	- A 64-bit integer pointer which holds the device info.
in	*command*	- The command to be sent to the RTSA

Returns

0 on success, or a negative number on error

Examples

largeBlockCaptureWithPowerFns.cpp, largeBlockCaptureWithPowerFnsAndDD.cpp, simpleGNSSPacketsCaptureExample.cpp, simpleHIF.cpp, simplePSDFunctionsExample.cpp, simpleRead.cpp, simpleReadHDR.cpp, streamExample.cpp, sweepDeviceCalculateChannelPower.cpp, sweepDeviceCalculateOccupiedBandwidth.cpp, sweepDeviceCaptureSweepSpectrun and sweepDevicePeakFind.cpp.

wsaGetSCPI_s()

```
int16_t wsaGetSCPI_s (
            int64_t wsa_handle,
            char * command,
            char * response )
```

Sends a SCPI query command to the RTSA and retrieve the response for the sent query as a pointer. The list of all the supported SCPI commands and their definition can be found in the RTSA Programmer's Guide.

This fn is an alternative to the wsaGetSCPI() but having the output returned as a parameter instead of as the function return.

Parameters

in	*wsa_handle*	- A 64-bit integer pointer which holds the device info
in	*command*	- The command to be sent to the RTSA
out	*response*	- The query response output

Returns

0 on success, or a negative number on error

Examples

largeBlockCaptureWithPowerFns.cpp, largeBlockCaptureWithPowerFnsAndDD.cpp, simpleGNSSPacketsCaptureExample.cpp, simpleHIF.cpp, simplePSDFunctionsExample.cpp, simpleRead.cpp, simpleReadHDR.cpp, streamExample.cpp,

sweepDeviceCalculateChannelPower.cpp, sweepDeviceCalculateOccupiedBandwidth.cpp, sweepDeviceCaptureSweepSpectrum
and sweepDevicePeakFind.cpp.

wsaGetSCPI()

```
char* wsaGetSCPI (
             int64_t wsa_handle,
             char * command )
```

Sends a SCPI query command to the RTSA and return the response as a function output. The list of all the supported SCPI commands and their definition can be found in the RTSA Programmer's Guide.

Parameters

in	*wsa_handle*	- A 64-bit integer pointer which holds the device info
in	*command*	- The command to be sent to the RTSA

Returns

A string containing the returned query.

Examples

simpleGNSSPacketsCaptureExample.cpp, simpleHIF.cpp, simplePSDFunctionsExample.cpp, simpleRead.cpp, simpleReadHDR.cpp, streamExample.cpp, sweepDeviceCalculateChannelPower.cpp, sweepDeviceCalculateOccupiedBandwidt
sweepDeviceCaptureSweepSpectrum.cpp, and sweepDevicePeakFind.cpp.

wsaSetAttenuation()

```
int16_t wsaSetAttenuation (
             int64_t wsa_handle,
             int32_t att_value )
```

Sets the attenuator's value for a "trace" capture mode (block or streaming). See the RTSA's Programmer's Guide for the values.

- For WSA5000: use values 0 (Off) and 1 (In) for 20dB.
- For RTSA5XX0: use 0, 10, 20, or 30 dB.

Parameters

in	*wsa_handle*	- A 64-bit integer pointer which holds the device info
in	*att_value*	- An integer containing the attenuation value

Returns

0 on success, or a negative number on error

Examples

largeBlockCaptureWithPowerFnsAndDD.cpp, simpleGNSSPacketsCaptureExample.cpp, and streamExample.cpp.

wsaGetAttenuation()

```
int16_t wsaGetAttenuation (
            int64_t wsa_handle,
            int32_t * att_value )
```

Gets the attenuator's value for a "trace" capture mode (block or streaming). For WSA5000, value 0 (Off) and 1 (In) refers to 20 dB attenuation state.

Parameters

in	*wsa_handle*	- A 64-bit integer pointer which holds the device info
out	*att_value*	- An integer pointer to store the attenuator's value

Returns

0 on successful, or a negative number on error

wsaSetReferencePLL()

```
int16_t wsaSetReferencePLL (
            int64_t wsa_handle,
            char * ref_type )
```

Sets the reference PLL's type, whether INT, EXT or GNSS, depending on your product.

Parameters

in	*wsa_handle*	- A 64-bit integer pointer which holds the device info
in	*ref_type*	- A char pointer to store the reference PLL's type

Returns

0 on successful, or a negative number on error

wsaGetReferencePLL()

```
int16_t wsaGetReferencePLL (
            int64_t wsa_handle,
            char * ref_type )
```

Gets the current reference PLL's type used.

Parameters

in	*wsa_handle*	- A 64-bit integer pointer which holds the device info
out	*ref_type*	- A char pointer to store the reference PLL's type

Returns

0 on successful, or a negative number on error

Examples

simpleGNSSPacketsCaptureExample.cpp.

wsaArmTraceCapture()

```
int16_t _stdcall wsaArmTraceCapture (
            int64_t wsa_handle,
            int32_t samples_per_packet,
            int32_t packets_per_block )
```

Sets the block capture parameters (:TRACE:SPP & :TRACE:BLOCK:PACKETS) to the RTSA and arms the capture (:TRACE:BLOCK:DATA?).

This function is needed for wsaReadData(), wsaReadVRTData() (if reading IF data also), wsaCaptureTraceAndContext() and wsaCaptureSpectrumAndContext().

Parameters

in	*wsa_handle*	- A 64 - bit integer pointer which holds the device info
in	*samples_per_packet*	- Number of IF samples in each VRT packet
in	*packets_per_block*	- Number of IF VRT packets per block to read

Returns

0 on success or a negative value on error

Examples

largeBlockCaptureWithPowerFns.cpp, largeBlockCaptureWithPowerFnsAndDD.cpp, and simpleGNSSPacketsCaptureExample.c

wsaCaptureSpectrumAndContext()

```
int16_t wsaCaptureSpectrumAndContext (
            int64_t wsa_handle,
            int32_t samples_per_packet,
            int32_t packets_per_block,
            uint32_t timeout,
            struct vrt_context * vrt_context_data,
            struct vrt_header * header,
            float * spectral_data,
            uint32_t * spectral_size,
            int64_t * spectral_fstart,
            int64_t * spectral_fstop,
            uint32_t * spectral_bandwidth )
```

Reads the block of IF data, computes and returns its power spectral density (PSD) data, with context data stored in structs. The spectral data returned is of "usable range" (i.e. the filter roll-off section has been trimmed).

This high-level function uses wsaCaptureTraceAndContext(), wsaComputeFFTWithCorrOptions() (with DC offset correction on & IQ imbalance correction is on for ZIF only), and wsaComputePSDUsableData() functions.

Remarks

- Must call wsaArmTraceCapture() before calling this fucntion to arm the capture
- Since this function does a 'block' capture, it loops to wait until the number of requested are captured; therefore, if a context information, such as the GNSS context, comes more than once during the capture, only the latest one will be returned in vrt_context_data.
- If only interested in GNSS context packets for RTSA with GNSS module, see wsaReadGNSSContext().

Parameters

in	*wsa_handle*	- A 64-bit integer pointer which holds the device info
in	*samples_per_packet*	- Number of IF samples in each VRT packet
in	*packets_per_block*	- Number of IF VRT packets per block to read
in	*timeout*	- A value used to set the socket read timeout (msec)
out	*vrt_context_data*	- A vrt_context struct containing the context information of the current capture
out	*header*	- A vrt_header struct storing the header info and time-stamp of the first IF VRT packet returned (when data capture occurred)
out	*spectral_data*	- Float pointer to an array containing the (trimmed) usable PSD or spectral data (in dBm)
out	*spectral_size*	- The number of samples of the spectral_data array
out	*spectral_fstart*	- Unsigned 64-bit integer pointer containing the (usable) start frequency of the spectral_data returned (in Hz)
out	*spectral_fstop*	- Unsigned 64-bit integer pointer containing the (usable) stop frequency of the spectral_data returned (in Hz)
out	*spectral_bandwidth*	- Unsigned 32-bit integer containing the (usable) bandwidth of the (usable) spectral_data returned (in Hz)

Returns

0 on success or a negative value on error

Examples

largeBlockCaptureWithPowerFns.cpp, largeBlockCaptureWithPowerFnsAndDD.cpp, and simpleGNSSPacketsCaptureExample.c

wsaCaptureTraceAndContext()

```
int16_t wsaCaptureTraceAndContext (
            int64_t wsa_handle,
            int32_t samples_per_packet,
            int32_t packets_per_block,
            uint32_t timeout,
            struct vrt_context * vrt_context_data,
            struct vrt_header * header,
            int16_t * i16_data,
            int16_t * q16_data,
            int32_t * i32_data )
```

Reads and returns a "block" of IF data (I/Q time domain data), with context data stored in structs.

wsaComputeFFTWithCorrOptions() and wsaComputePSDUsableData() functions could be used on the captured IQ data to get the PSD spectral data. Or just use wsaCaptureSpectrumAndContext().

Remarks

- Must call wsaArmTraceCapture() before calling this fucntion to arm the capture.
- Since this function does a 'block' capture, it loops to wait until the number of requested are captured; therefore, if a context information, such as the GNSS context, comes more than once during the capture, only the latest one will be returned in vrt_context_data.
- If only interested in GNSS context packets for RTSA with GNSS module, not the IF Data, see wsaReadGNSSContext().

Parameters

in	*wsa_handle*	- A 64-bit integer pointer which holds the device info
in	*samples_per_packet*	- Number of IF samples in each VRT packet
in	*packets_per_block*	- Number of IF VRT packets per block to read
in	*timeout*	- A socket read timeout value (milliseconds)
out	*vrt_context_data*	- A vrt_context struct containing the (latest) context information of the current capture
out	*header*	- A vrt_header struct storing the header data and time-stamp of the first IF VRT packet captured (when data capture occurred)
out	*i16_data*	- A pointer to a 16-bit buffer to hold the 16-bit idata
out	*q16_data*	- A pointer to a 16-bit buffer to hold the 16-bit qdata when available
out	*i32_data*	- A pointer to a 32-bit buffer to hold the 32-bit idata when HDR mode is used

Returns

0 on success or a negative value on error

wsaStreamStartWithID()

```
int16_t wsaStreamStartWithID (
            int64_t wsa_handle,
            int32_t samples_per_packet,
            int32_t stream_start_id )
```

Sets the number of stream samples per VRT packet to output, then initiates the capture and streaming of IQ data in the RTSA with the use of an ID to mark the start of that Stream (see Programmer's Guide on :TRACE:STREAM:START command).

Remarks

1. Must acquire read access, such as: wsaGetSCPI_s(wsaHandle, ":SYSTEM:LOCK:REQUEST? ACQ", query_result);
2. For R5xx0 products, as the capture rate within the device is much faster than the transfer rate, a decimation is highly recommended to be used with streaming. Use [:SENSe]:DECimation with wsaSetSCPI().

Parameters

in	*wsa_handle*	- A 64-bit integer pointer which holds the device info
in	*samples_per_packet*	- Number of IF samples in each VRT packet returned
in	*stream_start_id*	- An ID to mark the beginning of new data packets belonging to this new stream start

Returns

0 on success or a negative value on error

Examples

streamExample.cpp.

wsaStreamStop()

```
int16_t wsaStreamStop (
            int64_t wsa_handle )
```

Stop the stream mode, flush the internal memory of the device, and clear any remaining data packets in the network.

Parameters

in	*wsa_handle*	- A 64-bit integer pointer which holds the device info

Returns

0 on success or a negative value on error

Examples

streamExample.cpp.

wsaCaptureStreamTraceAndContext()

```
int16_t wsaCaptureStreamTraceAndContext (
            int64_t wsa_handle,
            int32_t samples_per_packet,
            uint32_t timeout,
            struct vrt_context * vrt_context_data,
            struct vrt_header * header,
            uint32_t * stream_start_id,
            int16_t * i16_data,
            int16_t * q16_data,
            int32_t * i32_data )
```

Reads and returns 1 packet of IF data (I/Q time domain data) of spp size during streaming, with packet's info and context stored in structs. Check the sample_loss_indicator in the vrt_context for if there're data loss occurred between this current packet and the one just before it (there's no data loss within the packet).

wsaComputeFFTWithCorrOptions() and wsaComputePSDUsableData() functions could be used on the captured IQ data to get the PSD spectral data. Or just use wsaCaptureSpectrumAndContext(). Note thouth doing data analysis during stream capture could affect its throughput rate.

Remarks

1. Must start with wsaStreamStartWithID() (see its notes) and end with wsaStreamStop()
2. The trailer of each packet has a "Sample Loss Indicator" (see the VRT section of the RTSA's Programmer's Guide), recommend to tracks this indicator. Also, make sure you get the latest firmware as the indicator was not available in some early versions.
3. If only interested in GNSS context packets for RTSA with GNSS module, not the IF Data, see wsaReadGNSSContext().

Parameters

in	*wsa_handle*	- A 64-bit integer pointer which holds the device info
in	*samples_per_packet*	- Number of IF samples in each VRT packet
in	*timeout*	- A socket read timeout value (milliseconds)
out	*vrt_context_data*	- A vrt_context struct containing the context information of the current capture
out	*header*	- A vrt_header struct storing the header data and time-stamp of the IF VRT packet captured
out	*stream_start_id*	- A pointer storing the stream start ID received from the VRT extension context, should match that set by wsaStreamStartWithID()
out	*i16_data*	- A pointer to a 16-bit buffer to hold the 16-bit idata
out	*q16_data*	- A pointer to a 16-bit buffer to hold the 16-bit qdata when available
out	*i32_data*	- A pointer to a 32-bit buffer to hold the 32-bit idata when HDR mode is used

Returns

0 on success or a negative value on error

Examples

streamExample.cpp.

wsaManualSweepStart()

```
int16_t wsaManualSweepStart (
            int64_t wsa_handle,
            int64_t sweep_start_id )
```

For starting a "manually" setup sweep (using SCPI commands) with a specified sweep start ID. Not needed for sweepDevice related functions below.

Remarks

Must acquire read access, such as: wsaGetSCPI_s(wsaHandle, ":SYSTEM:LOCK:REQUEST? ACQ", query↩ _result);

Parameters

in	*wsa_handle*	- A 64-bit integer pointer which holds the device info
in	*sweep_start↩ _id*	- A sweep start ID, will be returned in a VRT Extension context, used as a marker to indicate the start of the sweep data being sent back from the device.

Returns

0 on success or a negative value on error

wsaManualSweepStop()

```
int16_t wsaManualSweepStop (
            int64_t wsa_handle )
```

For stopping a "manually" setup sweep (using SCPI commands), flush on-device memory, and attempt to clear remaining packets in the socket. Not needed for sweepDevice related functions below.

Parameters

in	*wsa_handle*	- A 64-bit integer pointer which holds the device info

Returns

0 on success or a negative value on error

wsaCaptureManualSweepTraceAndContext()

```
int16_t wsaCaptureManualSweepTraceAndContext (
            int64_t wsa_handle,
            int32_t samples_per_packet,
            int32_t packets_per_step,
            uint32_t timeout,
            struct vrt_context * vrt_context_data,
            struct vrt_header * header,
            uint32_t * sweep_start_id,
            int16_t * i16_data,
            int16_t * q16_data,
            int32_t * i32_data )
```

This function is for a "manually" setup sweep (using SCPI commands) only. Not applicable for sweepDevice related functions below. It reads and returns "1" block (trace) of IF data (I/Q time domain data) of "samples_per_packet * packets_per_step" size during "a sweep step" with packet's info and context stored in structs. Make sure samples↩ _per_packet & packets_per_step are the same values used for :SWEEP:ENTRY:SPP & :SWEEP:ENTRY:PPB, respectively. If a manual sweep setup consists of many entries, keep them the same for all entries to simplify the capture loop.

Remarks

1. Must start with wsaManualSweepStart() (see its notes) and end with wsaManualSweepStop()
2. If only interested in GNSS context packets for RTSA with GNSS module, not the IF Data, see wsaReadGNSSContext().

Parameters

in	*wsa_handle*	- A 64-bit integer pointer which holds the device info
in	*samples_per_packet*	- Number of IF samples in each VRT packet
in	*packets_per_step*	- Number of packets (form a block) per sweep step
in	*timeout*	- A socket read timeout value (milliseconds)
out	*vrt_context_data*	- A vrt_context struct containing the context information of the current capture
out	*header*	- A vrt_header struct storing the header data and time-stamp of the IF VRT packet captured
out	*sweep_start_id*	- A pointer storing the sweep start ID received from the VRT extension context, should match that set by wsaManualSweepStart()
out	*i16_data*	- A pointer to a 16-bit buffer to hold the 16-bit idata
out	*q16_data*	- A pointer to a 16-bit buffer to hold the 16-bit qdata when available
out	*i32_data*	- A pointer to a 32-bit buffer to hold the 32-bit idata when HDR mode is used

Returns

0 on success or a negative value on error

wsaReadData()

```
int16_t wsaReadData (
            int64_t wsa_handle,
            int16_t * i16_data,
            int16_t * q16_data,
            int32_t * i32_data,
            uint32_t timeout,
            uint32_t * stream_id,
            uint8_t * spectral_inversion,
            int32_t samples_per_packet,
            uint32_t * timestamp_sec,
            uint64_t * timestamp_psec,
            int16_t * reference_level,
            uint64_t * bandwidth,
            uint64_t * center_frequency )
```

Reads and returns "one" VRT IF data packet (I/Q time domain data), with info from the corresponding context packets parsed into the parameters. Recommends using higher level functions wsaCaptureTraceAndContext() or wsaCaptureSpectrumAndContext() instead if capturing a 'block' of data.

Remarks

1. Do not use this function if other VRT packets are also interested, see wsaReadVRTData() or wsaReadGNSSContext().
2. This function does not set the **samples_per_packet** to the RTSA. It is the user's responsibility to configure the RTSA with the correct **samples_per_packet** before initiating a capture, by calling wsaArmTraceCapture() for instance.

Parameters

in	*wsa_handle*	- A 64-bit integer pointer which holds the device info
out	*i16_data*	- A pointer to a 16-bit buffer to hold the 16-bit idata
out	*q16_data*	- A pointer to a 16-bit buffer to hold the 16-bit qdata when available
out	*i32_data*	- A pointer to a 32-bit buffer to hold the 32-bit idata when HDR mode is used
in	*timeout*	- A value used to set the socket read timeout (millisecs)
out	*stream_id*	- A pointer to store the IF data packet's stream id
out	*spectral_inversion*	- A pointer to to indicate the spectral inversion state of the I-data only, needed for when mapping the frequencies of the spectral data in SH/SHN mode without this API functions.
in	*samples_per_packet*	- The number of samples to read
out	*timestamp_sec*	- A pointer to hold the timestamp value (seconds)
out	*timestamp_psec*	- A pointer to hold the timestamp value (picosecs)
out	*reference_level*	- A pointer to store reference level for calibrating the spectral data (dBm)
out	*bandwidth*	- The bandwidth of the IF data (Hz)
out	*center_frequency*	- The center frequency of the IF data (Hz)

Returns

0 on success or a negative value on error

Examples

simplePSDFunctionsExample.cpp, simpleRead.cpp, and simpleReadHDR.cpp.

wsaReadVRTData()

```
int16_t wsaReadVRTData (
                int64_t wsa_handle,
                int32_t samples_per_packet,
                uint32_t timeout,
                struct vrt_header * header,
                int16_t * i16_data,
                int16_t * q16_data,
                int32_t * i32_data,
                uint8_t * spectral_inversion,
                struct vrt_context * vrt_context_data )
```

Reads and returns the next available VRT data packet in the socket, which could be context or IF (I/Q) data packet. See the product's Programmer's Guide for more info on the VRT protocol. This function is useful for when intending to handle the VRT packets by your own application instead of using the capture methods provided.

The VRT context info is useful for understanding the output:

- Use 'packet_type' member of 'header' struct to determine VRT packet types (IF_PACKET_TYPE, CONTEX←T_PACKET_TYPE, or EXTENSION_PACKET_TYPE).
- Use 'stream_id' member of 'vrt_context_data' struct to determine what type of data this is (RECEIVER_STR←EAM_ID, DIGITIZER_STREAM_ID, EXTENSION_STREAM_ID, I16Q16_DATA_STREAM_ID, I16_DATA_S←TREAM_ID or I32_DATA_STREAM_ID).
- For R57x0 with GNSS feature, to run a long loop in order to gather GNSS context pkt, if the function's output is WSA_WARNING_SOCKETTIMEOUT, could ignore that warning to keep looping. See the example called "simpleGNSSPacketsCaptureExample.cpp" for example.

Remarks

1. This function does not set the **samples_per_packet** to the RTSA. It is the user's responsibility to configure the RTSA with the correct **samples_per_packet** before initiating a capture, by calling wsaArmTraceCapture() for instance.
2. If capturing a 'block' of data along with the context is desired, use higher level function wsaCaptureTraceAndContext() or wsaCaptureSpectrumAndContext() instead.
3. If only wish to retrieve 1 VRT IF packet (I/Q time domain data) at a time, see wsaReadData() function.

Parameters

in	*wsa_handle*	- A 64-bit integer pointer which holds the device info
in	*samples_per_packet*	- Number of samples to read per VRT IF packet
in	*timeout*	- The socket read timeout value (milliseconds)
out	*header*	- A vrt_header struct pointer containing the VRT header info
out	*i16_data*	- A pointer to a 16-bit buffer to hold the 16-bit idata
out	*q16_data*	- A pointer to a 16-bit buffer to hold the 16-bit qdata when available
out	*i32_data*	- A pointer to a 32-bit buffer to hold the 32-bit idata when HDR mode is used
out	*spectral_inversion*	- A pointer to to indicate the spectral inversion state of the I-data only, needed for when mapping the frequencies of the spectral data in SH/SHN mode without this API functions.
out	*vrt_context_data*	- A vrt_context struct pointer containing the VRT context information

Returns

0 on success or a negative value on error

wsaGetFFTSize()

```
int16_t wsaGetFFTSize (
            int32_t sample_size,
            uint32_t stream_id,
            int32_t * fft_size )
```

Get the required buffer size to store the FFT data. This function is used with wsaComputeFFT() & wsaComputeFFTWithCorrOptions() functions.

Parameters

in	*sample_size*	- The sample size of the time domain data
in	*stream_id*	- The stream id of the time domain data
out	*fft_size*	- A pointer to store the size of the FFT array

Returns

0 on success, or a negative number on error

Examples

simplePSDFunctionsExample.cpp, simpleRead.cpp, simpleReadHDR.cpp, and streamExample.cpp.

wsaComputeFFTWithCorrOptions()

```
int16_t wsaComputeFFTWithCorrOptions (
            int32_t sample_size,
            int32_t fft_size,
            uint32_t stream_id,
            int16_t reference_level,
            int8_t dc_offset_corr,
            int8_t iq_imbalance_corr,
            int16_t * i16_buffer,
            int16_t * q16_buffer,
            int32_t * i32_buffer,
            float * spectral_data )
```

Compute the FFT of the given time domain data and return the Power Spectral Density (PSD) data, with options to do time-domain IQ corrections. I16/Q16 is for ZIF, SH, SHN, & DD where applicable, & I32 is for HDR mode.

This function does the following:

- normalized the data
- applies DC Offset correction on I & Q data if specified
- applies IQ imbalance correction if specified, recommend to enable for ZIF
- performed Hann Windowing and FFT
- compute PSD with calibrated ref level adjustment

To get only usable data within the whole spectrum after this function, see wsaComputePSDUsableData() function.

Parameters

`in`	*sample_size*	- The number of samples to be computed
`in`	*fft_size*	- The size of the FFT buffer (SH/SHN without decimation resulted in FFT size = 1/2 SPP). Use wsaGetFFTSize() function to get the FFT size.
`in`	*stream_id*	- The IF data packet's stream id
`in`	*reference_level*	- The reference level to calibrate the spectral data (dBm)
`in`	*dc_offset_corr*	- Set 1 to turn on DC offset correction for the IQ data, or 0 to turn it off
`in`	*iq_imbalance_corr*	- Set 1 to turn on IQ imbalance correction for the ZIF mode (recommended), or 0 to turn it off.
`in`	*i16_buffer*	- A pointer to a 16-bit buffer to hold the 16-bit idata
`in`	*q16_buffer*	- A pointer to a 16-bit buffer to hold the 16-bit qdata
`in`	*i32_buffer*	- A pointer to a 32-bit buffer to hold the 32-bit idata of HDR mode when used
`out`	*spectral_data*	- A pointer to store the PSD or spectral data

Returns

0 on success or a negative value on error

Examples

simplePSDFunctionsExample.cpp, simpleRead.cpp, simpleReadHDR.cpp, and streamExample.cpp.

wsaComputeFFT()

```
int16_t wsaComputeFFT (
            int32_t sample_size,
            int32_t fft_size,
            uint32_t stream_id,
            int16_t reference_level,
            uint8_t spectral_inversion,
            int16_t * i16_buffer,
            int16_t * q16_buffer,
            int32_t * i32_buffer,
            float * spectral_data )
```

Compute the FFT of the given time domain data and return the Power Spectral Density (PSD) data. I16/Q16 is for ZIF, SH, SHN, & DD where applicable, & I32 is for HDR mode.

Note: Recommend using wsaComputeFFTWithCorrOptions() instead.

This function does the following:

- normalized the data
- applied DC Offset correction on I & Q data *always*
- applied IQ imbalance correction *always*
- performed Hann Windowing and FFT
- compute PSD with calibrated ref level adjustment

- spectral_inversion is taken care of in time-domain (started in v1.5.2), the variable is kept here for backward compatibility.

To get only usable data within the whole spectrum after this function, see wsaComputePSDUsableData() function.

Parameters

in	*sample_size*	- The number of samples to be computed
in	*fft_size*	- The size of the FFT buffer (SH/SHN without decimation resulted in FFT size = 1/2 SPP). Use wsaGetFFTSize() function to get the FFT size.
in	*stream_id*	- The IF data packet's stream id
in	*reference_level*	- The reference level to calibrate the spectral data (dBm)
in	*spectral_inversion*	- Indicator for if the spectral data is inverted (1) or not (0)
in	*i16_buffer*	- A pointer to a 16-bit buffer to hold the 16-bit idata
in	*q16_buffer*	- A pointer to a 16-bit buffer to hold the 16-bit qdata
in	*i32_buffer*	- A pointer to a 32-bit buffer to hold the 32-bit idata of HDR mode when used
out	*spectral_data*	- A pointer to store the PSD or spectral data

Returns

0 on success or a negative value on error

wsaPSDPeakFind()

```
int16_t wsaPSDPeakFind (
             int64_t wsa_handle,
             char * rfe_mode,
             uint8_t spectral_inversion,
             uint32_t data_size,
             float * spectral_data,
             int64_t * peak_freq,
             float * peak_power )
```

Find the peak within the "usable (operating) bandwidth" range of the spectral (PSD) data returned from wsaComputeFFTWithCorrOptions() function. Recommend to use this function for peak finding as SH/SHN/H←DR's IF center frequency is not at 0 IF.

However, if wsaComputePSDUsableData() is applied to spectral data after wsaComputeFFTWithCorrOptions(), a regular peak (maximum value) search would be sufficient.

Note: If frequency shift is used, peak freq would need to adjust accordingly (ex. peak freq = peak_freq - freq_shift).

Parameters

in	*wsa_handle*	- A 64-bit integer pointer which holds the device info.
in	*rfe_mode*	- String containing the RFE mode
in	*spectral_inversion*	- The spectral inversion indicator for the given frequency. Indicator is provided from wsaReadData().
in	*data_size*	- The number of samples inside the spectral data array
in	*spectral_data*	- A floating point array containing the spectral data (in dBm)

Parameters

out	*peak_freq*	- An unsigned 64-bit integer to store the peak's frequency (in Hz)
out	*peak_power*	- A floating point to store the peak's power level (in dBm)

Returns

0 on success or a negative value on error

Examples

simplePSDFunctionsExample.cpp.

wsaComputePSDChannelPower()

```
int16_t wsaComputePSDChannelPower (
             uint32_t start_bin,
             uint32_t stop_bin,
             uint32_t data_size,
             float * spectral_data,
             float * channel_power )
```

Calculate the channel power of a range in the given spectral (PSD) data.

Parameters

in	*start_bin*	- The first bin within the spectral data where the channel power calculation should take place
in	*stop_bin*	- The last bin within the spectral data where the channel power calculation should take place
in	*data_size*	- The number of spectral data points (bins)
in	*spectral_data*	- A floating point array containing the spectral data (in dBm)
out	*channel_power*	- A floating point pointer to store the channel power (in dBm)

Returns

0 on success or a negative value on error

Examples

largeBlockCaptureWithPowerFns.cpp, largeBlockCaptureWithPowerFnsAndDD.cpp, and simplePSDFunctionsExample.cpp.

wsaComputePSDOccupiedBandwidth()

```
int16_t wsaComputePSDOccupiedBandwidth (
             uint32_t rbw,
             uint32_t data_size,
             float * spectral_data,
             float occupied_percentage,
             uint64_t * occupied_bandwidth )
```

Calculate the occupied power bandwidth for the specified occupied percentage of the given spectral (PSD) data.

Parameters

in	*rbw*	- A 32-bit integer containing the resolution BW (RBW) value of the data (in Hz). Usually, RBW = spectral BW / spectral data size.
in	*spectral_data*	- Float pointer array containing the spectral data (in dBm)
in	*data_size*	- The number of samples inside the spectral data array
in	*occupied_percentage*	- The channel power percentage (in %) to be used for calculating the corresponding occupied bandwidth
out	*occupied_bandwidth*	-An unsigned 64-bit pointer to hold the bandwidth (in Hz)

Returns

0 on success or a negative value on error

Examples

largeBlockCaptureWithPowerFns.cpp, largeBlockCaptureWithPowerFnsAndDD.cpp, and simplePSDFunctionsExample.cpp.

wsaComputePSDUsableData()

```
int16_t wsaComputePSDUsableData (
                int64_t wsa_handle,
                char * rfe_mode,
                uint8_t spectral_inversion,
                float * spectral_data,
                uint32_t data_size,
                uint32_t * usable_bandwidth,
                int64_t * usable_fstart,
                int64_t * usable_fstop,
                float * usable_spectral_data,
                uint32_t * usable_data_size )
```

Calculates and returns the usable PSD related data for the given info. This is useful for if, for instances, data are to be plotted or to find power info.

Note: This function is not applicable for sweep-device capture mode or data.

Parameters

in	*wsa_handle*	- A 64-bit integer pointer which holds the device info
in	*rfe_mode*	- String containing the RFE mode
in	*spectral_inversion*	- The spectral inversion indicator for the given frequency. Indicator is provided from wsaReadData().
in	*spectral_data*	- Float pointer array containing the spectral data (in dBm) to be trimmed to usable data
in	*data_size*	- The number of samples inside the spectral data array

Parameters

out	*usable_bandwidth*	- Unsigned 32-bit integer containing the (usable) bandwidth of the (usable) spectral_data returned (in Hz)
out	*usable_fstart*	- Unsigned 64-bit integer containing the (usable) start frequency of the (usable) spectral_data returned (in Hz)
out	*usable_fstop*	- Unsigned 64-bit integer containing the (usable) stop frequency of the (usable) spectral_data returned (in Hz)
out	*usable_spectral_data*	- Float pointer to an array containing the (trimmed) usable spectral_data (in dBm)
out	*usable_data_size*	- The number of samples inside the usable_spectral_data array

Returns

0 on success or a negative value on error

Examples

simplePSDFunctionsExample.cpp.

wsaGetSweepSize()

```
int16_t wsaGetSweepSize (
             int64_t wsa_handle,
             uint64_t fstart,
             uint64_t fstop,
             uint32_t rbw,
             char * rfe_mode,
             int32_t attenuator,
             uint32_t * sweep_size )
```

Get sweep size, superseded by wsaGetSweepSize_s().

wsaGetSweepSize_s()

```
int16_t wsaGetSweepSize_s (
             int64_t wsa_handle,
             uint64_t fstart,
             uint64_t fstop,
             uint32_t rbw,
             char * rfe_mode,
             int32_t attenuator,
             uint64_t * fstart_actual,
             uint64_t * fstop_actual,
             uint32_t * sweep_size )
```

This function supersedes the wsaGetSweepSize() function as it returning also the actual fstart & fstop values of the sweep data output. These values are different than the user's requested freqs as the user's sweep range might not be a multiple of the RFE mode's operating IBW (i.e. 40 MHz for SH & 50 MHz for DD), and they are useful for plotting, for example, the spectral data output of wsaCaptureSpectrum().

Parameters

in	*wsa_handle*	- A 64-bit integer pointer which holds the device info
in	*fstart*	- Start frequency of the sweep (Hz)
in	*fstop*	- Stop frequency of the sweep (Hz)
in	*rbw*	- RBW of the sweep (Hz)
in	*rfe_mode*	- RFE mode of the sweep (SH or SHN)
in	*attenuator*	- The attenuator value (in dBm)
out	*fstart_actual*	- the actual start freq of the sweep data output
out	*fstop_actual*	- the actual stop freq of the sweep data output
out	*sweep_size*	- Size of sweep data points for the given configuration

Returns

0 on success, or a negative number on error

Examples

sweepDeviceCaptureSweepSpectrum.cpp.

wsaCaptureSpectrum()

```
int16_t wsaCaptureSpectrum (
            int64_t wsa_handle,
            uint64_t fstart,
            uint64_t fstop,
            uint32_t rbw,
            char * rfe_mode,
            int32_t attenuator,
            float * spectral_data )
```

Perform a sweep device capture and compute PSD data based on a given sweep configuration.

Remarks

1. You must acquire read status before calling this function
2. Call wsaGetSweepSize_s() to determine the array sweep_size required to store the resulted spectral data.
3. DO NOT used the actual fstart & fstop returned from wsaGetSweepSize_s() as the input parameters for this function.
4. However, use the actual fstart & fstop returned from wsaGetSweepSize_s() to do frequency mapping for the output spectral_data.

Warning: When performing a sweep-device over a very large span (typically 7GHz or higher), usage of small RBWs under 10 Hz could result in a memory allocation failure as your computer might not have enough RAM space to handle the allocation.

Parameters

in	*wsa_handle*	- A 64-bit integer pointer which holds the device info
in	*fstart*	- Start frequency of the sweep (Hz)

Parameters

in	*fstop*	- Stop frequency of the sweep (Hz)
in	*rbw*	- RBW of the sweep (Hz)
in	*rfe_mode*	- RFE mode of the sweep (SH or SHN)
in	*attenuator*	- The attenuator value (in dBm)
out	*spectral_data*	- Array to hold the output power spectral density data

Returns

0 on success, or a negative number on error

Examples

sweepDeviceCaptureSweepSpectrum.cpp.

wsaPeakFind()

```
int16_t wsaPeakFind (
            int64_t wsa_handle,
            uint64_t fstart,
            uint64_t fstop,
            uint32_t rbw,
            char * rfe_mode,
            float ref_offset,
            int32_t attenuator,
            uint64_t * peak_freq,
            float * peak_power )
```

Perform a sweep device catpure and find the peak value within the specified range.

Remarks

1. You must acquire read status before calling this function
2. DO NOT used the actual fstart & fstop returned from wsaGetSweepSize_s() as the input parameters for this function.

Parameters

in	*wsa_handle*	- A 64-bit integer pointer which holds the device info
in	*fstart*	- An unsigned 64-bit integer containing the start frequency (Hz)
in	*fstop*	- An unsigned 64-bit integer containing the stop frequency (Hz)
in	*rbw*	- An unsigned 64-bit integer containing the RBW value of the captured data (in Hz)
in	*rfe_mode*	- A string containing the RFE mode (SH or SHN)
in	*ref_offset*	- A float used to apply any additional offset such as cable loss (dBm, positive value for gain, negative value for loss)
in	*attenuator*	- An integer to hold the attenuator value
out	*peak_freq*	- An unsigned 64-bit integer to store the frequency of the peak (in Hz)
out	*peak_power*	- A flloating point pointer to store the power level of the peak (in dBm)

Returns

0 on success or a negative value on error

Examples

sweepDeviceCaptureSweepSpectrum.cpp, and sweepDevicePeakFind.cpp.

wsaChannelPower()

```
int16_t wsaChannelPower (
             int64_t wsa_handle,
             uint64_t fstart,
             uint64_t fstop,
             uint32_t rbw,
             char * rfe_mode,
             float ref_offset,
             int32_t attenuator,
             float * channel_power )
```

Perform a sweep device data capture and calculate the channel power within the specific range.

Remarks

1. You must acquire read status before calling this function
2. DO NOT used the actual fstart & fstop returned from wsaGetSweepSize_s() as the input parameters for this function.
3. Use wsaComputePSDChannelPower() instead if wish to calculate on existing data

Parameters

in	*wsa_handle*	- A 64-bit integer pointer which holds the device info
in	*fstart*	- An unsigned 64-bit integer containing the start frequency (Hz)
in	*fstop*	- An unsigned 64-bit integer containing the stop frequency (Hz)
in	*rbw*	- A 64-bit integer containing the RBW value of the captured data (in Hz)
in	*rfe_mode*	- A string containing the RFE mode (SH or SHN)
in	*ref_offset*	- A float used to apply any additional offset, such as cable loss (dBm)
in	*attenuator*	- An integer to hold the attenuator value
out	*channel_power*	- A flloating point pointer to store the channel power (in dBm)

Returns

0 on success or a negative value on error

Examples

sweepDeviceCalculateChannelPower.cpp.

wsaOccupiedBandwidth()

```
int16_t wsaOccupiedBandwidth (
             int64_t wsa_handle,
             uint64_t fstart,
             uint64_t fstop,
             uint32_t rbw,
             float occupied_percentage,
             char * rfe_mode,
             int32_t attenuator,
             uint64_t * occupied_bw )
```

Perform data capture and calculate the occupied bandwidth.

Remarks

1. You must acquire read status before calling this function
2. DO NOT used the actual fstart & fstop returned from wsaGetSweepSize_s() as the input parameters for this function.
3. Use wsaComputePSDOccupiedBandwidth() instead if wish to calculate on existing data

Parameters

in	*wsa_handle*	- A 64-bit integer pointer which holds the device info
in	*fstart*	- An unsigned 64-bit integer containing the start frequency (Hz)
in	*fstop*	- An unsigned 64-bit integer containing the stop frequency (Hz)
in	*rbw*	- A 64-bit integer containing the RBW value of the captured data (in Hz)
in	*occupied_percentage*	- How much of the channel power percentage should be in the occupied bandwidth
in	*rfe_mode*	- A string containing the RFE mode (SH or SHN)
in	*attenuator*	- An integer to hold the attenuator value
out	*occupied_bw*	- An unsigned 64-bit pointer to hold the bandwidth (MHz)

Returns

0 on success or a negative value on error

Examples

sweepDeviceCalculateOccupiedBandwidth.cpp.

wsaSetReferencePPS()

```
int16_t wsaSetReferencePPS (
             int64_t wsa_handle,
             char * pps_type )
```

Sets the reference PPS's type, whether 'EXT' or 'GNSS', depending on your product. This function is to be used with RTSA with GNSS module only.

Parameters

in	*wsa_handle*	- A 64-bit integer pointer which holds the device info.
in	*pps_type*	- A char pointer to store the PPS'S type.

Returns

0 on successful, or a negative number on error.

wsaGetReferencePPS()

```
int16_t wsaGetReferencePPS (
            int64_t wsa_handle,
            char * pps_type )
```

Gets the current reference PPS's type set. This function is to be used with RTSA with GNSS module only.

Parameters

in	*wsa_handle*	- A 64-bit integer pointer which holds the device info.
out	*pps_type*	- A char pointer to store the PPS'S type.

Returns

0 on successful, or a negative number on error.

Examples

simpleGNSSPacketsCaptureExample.cpp.

wsaGNSSEnable()

```
int16_t wsaGNSSEnable (
            int64_t wsa_handle,
            int32_t enable )
```

Enable or disable the GNSS module of the RTSA product with GNSS

Parameters

in	*wsa_handle*	- A 64-bit integer pointer which holds the device info
in	*enable*	- Enable (1) or disable (0) the GNSS module

Returns

0 on success or a negative value on error

wsaGetGNSSStatus()

```
int16_t wsaGetGNSSStatus (
            int64_t wsa_handle,
            uint8_t * status )
```

Return the status of the GNSS module of the RTSA product with GNSS, whether GNSS is on (1) or off (0).

Parameters

in	*wsa_handle*	- A 64-bit integer pointer which holds the device info
out	*status*	- An uint8_t pointer storing the status, 1 for on & 0 off

Returns

0 on success or a negative value on error

Examples

simpleGNSSPacketsCaptureExample.cpp.

wsaGetGNSSPosition()

```
int16_t wsaGetGNSSPosition (
            int64_t wsa_handle,
            float * latitude,
            float * longitude,
            float * altitude )
```

Return the GNSS position (latitude (degrees), longitude (degrees), and altitude (meters)) of the RTSA product with GNSS.

Parameters

in	*wsa_handle*	- A 64-bit integer pointer which holds the device info
out	*latitude*	- A float pointer to the latitude (degrees) value
out	*longitude*	- A float pointer to the longitude (degrees) value
out	*altitude*	- A float pointer to the altitude (meters) value

Returns

0 on success or a negative value on error

Examples

simpleGNSSPacketsCaptureExample.cpp.

wsaGetGNSSFixSource()

```
int16_t wsaGetGNSSFixSource (
              int64_t wsa_handle,
              char * fix_source )
```

This funtion determines if there's a GNSS fix through the :GNSS:REFerence? command. The GNSS module updates the fix information approximately every second. This function reflects the moment the query has been received.

Note: When an INT (no fix) value is returned, it means that the GNSS module is being disciplined by an internal 10 MHz reference oscilator, instead of the GNSS module.

Parameters

in	*wsa_handle*	- A 64-bit integer pointer which holds the device info
out	*fix_source*	- A char pointer storing the fix source, GNSS or INT.

Returns

0 on success or a negative value on error

Examples

simpleGNSSPacketsCaptureExample.cpp.

wsaSetGNSSAntennaDelay()

```
int16_t wsaSetGNSSAntennaDelay (
              int64_t wsa_handle,
              int32_t delay )
```

Set antenna cable delay for the RTSA with GNSS module, in nanoseconds.

Parameters

in	*wsa_handle*	- A 64-bit integer pointer which holds the device info
in	*delay*	- A int32_t delay input, in nanoseconds. Valid values are -32768 to 32767.

Returns

0 on success or a negative value on error

wsaGetGNSSAntennaDelay()

```
int16_t wsaGetGNSSAntennaDelay (
              int64_t wsa_handle,
              int32_t * delay )
```

Get antenna cable delay for the RTSA with GNSS module.

Parameters

in	*wsa_handle*	- A 64-bit integer pointer which holds the device info
out	*delay*	- An int32_t pointer storing the delay, in nanoseconds.

Returns

0 on success or a negative value on error

Examples

simpleGNSSPacketsCaptureExample.cpp.

wsaSetGNSSConstellation()

```
int16_t wsaSetGNSSConstellation (
            int64_t wsa_handle,
            char * constel1,
            char * constel2 )
```

Set the constellation option(s) for the RTSA with GNSS module. The options supported are "GPS", "GLONASS", & "BEIDOU". One or two options could be chosen.

Parameters

in	*wsa_handle*	- A 64-bit integer pointer which holds the device info
in	*constel1*	- A char pointer to store the 1st constellation option (required).
in	*constel2*	- A char pointer to store the 2nd constellation option. This is optional, & use "" if not used.

Returns

0 on success or a negative value on error

wsaGetGNSSConstellation()

```
int16_t wsaGetGNSSConstellation (
            int64_t wsa_handle,
            char * constellations )
```

Get the constellation option(s) currently set in the RTSA with GNSS module.

Parameters

in	*wsa_handle*	- A 64-bit integer pointer which holds the device info
out	*constellations*	- A char pointer storing the option(s), comma separated if more than 1 returned

Returns

0 on success or a negative value on error

Examples

simpleGNSSPacketsCaptureExample.cpp.

wsaReadGNSSContext()

```
int16_t wsaReadGNSSContext (
             int64_t wsa_handle,
             uint32_t timeout,
             int32_t * new_gnss_data,
             struct vrt_gnss_geolocn * gnss_data )
```

This function applies to RTSA with GNSS module only.

Reads and returns the next available VRT GNSS context packet in the socket (see the product's Programmer's Guide for more info on the VRT protocol). If wish to retrieve both VRT IF data (I/Q time domain data) and VRT context packets, see wsaCaptureSpectrumAndContext() or wsaCaptureTraceAndContext() function instead.

To run a long loop in order to gather GNSS context pkts, if the function's output is WSA_WARNING_SOCKETTIM←EOUT, could ignore that result in the checking. See the example called "simpleGNSSPacketsCaptureExample.cpp" for usage.

Parameters

in	*wsa_handle*	- A 64-bit integer pointer which holds the device info
in	*timeout*	- A value used to set the socket read timeout (milliseconds)
out	*new_gnss_data*	- An int32_t pointer indicating if there's new GNSS data
out	*gnss_data*	- A vrt_gnss_geolocn struct pointer containing the VRT GNSS context information

Returns

0 on success or a negative value on error

Examples

simpleGNSSPacketsCaptureExample.cpp.

wsaGetRFESpan()

```
int16_t wsaGetRFESpan (
             int64_t wsa_handle,
             char * rfe_mode,
             uint32_t * span_bw,
             int64_t * span_center,
             int64_t * span_fstart,
             int64_t * span_fstop )
```

Determines the bandwidth, center, start and stop frequencies of the RFE mode. The output bandwidth, start & stop frequencies are of the RFE 'full' bandwidth (ie. not of the useable/operating range).

This function is useful as in some RFE modes, the IF center frequency is not at half the bandwidth so the start & stop calculation are taken cared of.

Note: This function relies on the latest device configuration, which means that frequency shift and decimation have been taken into consideration.

Parameters

in	*wsa_handle*	- A 64-bit integer pointer which holds the device info

Parameters

in	*rfe_mode*	- String containing the RFE mode
out	*span_bw*	- Unsigned 32-bit integer containing the bandwidth (in Hz)
out	*span_center*	- Signed 64-bit integer containing the center frequency (in Hz)
out	*span_fstart*	- Signed 64-bit integer containing the start frequency (in Hz)
out	*span_fstop*	- Signed 64-bit integer containing the stop frequency (in Hz)

Returns

0 on success or a negative value on error

Examples

simpleGNSSPacketsCaptureExample.cpp, and simpleRead.cpp.

wsaErrorMessage_s()

```
void wsaErrorMessage_s (
             int16_t error_code,
             char * error_msg )
```

Get an error message corresponding to an error code.
This function is an alternative to the wsaErrorMessage() but having the output returned as a parameter instead of the function return.

Parameters

in	*error_code*	- Error code
out	*error_msg*	- The error message string output corresponding to the error code given

Returns

None

Examples

largeBlockCaptureWithPowerFns.cpp, largeBlockCaptureWithPowerFnsAndDD.cpp, simpleGNSSPacketsCaptureExample.cpp, simplePSDFunctionsExample.cpp, simpleRead.cpp, streamExample.cpp, sweepDeviceCalculateChannelPower.cpp, sweepDeviceCalculateOccupiedBandwidth.cpp, sweepDeviceCaptureSweepSpectrum.cpp, and sweepDevicePeakFind.cpp.

wsaErrorMessage()

```
char* wsaErrorMessage (
             int16_t error_code )
```

Returns an error message corresponding to an error code.

Parameters

in	*error_code*	- Error code

Returns

A string containing the error message corresponding to the error code

Examples

simpleHIF.cpp, simpleReadHDR.cpp, and streamExample.cpp.

getBuildInfo()

```
void getBuildInfo (
            char * build_info )
```

Returns through build_info the build version and date of this library.

Note: This is an un-official DLL version method.

Parameters

out	*build_info*	- A char pointer storing the build version info

Returns

None

Chapter 7

Example Documentation

7.1 largeBlockCaptureWithPowerFns.cpp

An example that demonstrates how to read a large block of data packet in SH mode. The example then computes the FFT & PSD and follows with some power related computation within user's specified frequency range.

Copyright

```
#include <tchar.h>
#include <stdio.h>
#include "wsaInterface.h"
int _tmain()
{
    // The device instance
    int64_t wsaHandle;
    char ip[50];
    // device settings
    const int32_t spp = 8000;    // samples per VRT packet
    int32_t ppb = 1;             // number of VRT packets per block
    int32_t ppb_max = 1;
    uint32_t block_samples;
    char * rfe_mode = "SH";
    float sample_rate;
    float input_freq = 1003.0; // in MHz
    float freq_shift = 0.0; // in MHs
    uint32_t decimation = 1;
    // User related values
    int32_t user_rbw;
    float actual_rbw;
    uint64_t user_fstart, user_fstop;
    uint32_t user_start_bin, user_stop_bin;
    // variables for data read
    uint32_t timeout = 5000;
    struct vrt_header pkt_header;
    struct vrt_context vrt_context_data;
    // for PSD functions
    uint32_t usable_bandwidth;
    int64_t usable_fstart, usable_fstop;
    float *usable_spectral_data;
    uint32_t usable_data_size;
    int64_t peak_freq = 0;
    float peak_power = 0;
    float channel_power;
    float occupied_percentage = 90.5;
    uint64_t occupied_bandwidth;
    // Values to store function results
    int16_t result;
    char err_msg[1024];
    char cmd_str[256];
    char resp_str[512];
    char *query_result = resp_str;
    int i = 0;
```

```
// grab IP from user
printf("Enter an IP address: ");
scanf_s("%s", ip, (unsigned)_countof(ip));
printf("IP received: %s\n\n", ip);
printf("Enter RBW in Hz: ");
scanf_s("%s", resp_str, (unsigned)_countof(resp_str));
if (atoi(resp_str) > 0) {
    user_rbw = atoi(resp_str);
}
else {
    printf("Invalid RBW provided. Must be a positive integer and greater than 0 Hz.\n");
    return -1;
}
printf("RBW received:  %d Hz\n\n", user_rbw);
printf("Enter CENTER FREQ, in MHz (0 to use default): ");
scanf_s("%s", resp_str, (unsigned)_countof(resp_str));
if (atol(resp_str) > 0)
    input_freq = atof(resp_str);
printf("Center frequency received: %lf MHz\n\n", input_freq);
printf("Enter occupied bandwidth percentage, in %%: ");
scanf_s("%s", resp_str, (unsigned)_countof(resp_str));
if (atol(resp_str) > 0)
    occupied_percentage = (float)atof(resp_str);
printf("Percentage received:  %lf %%\n\n", occupied_percentage);
//*********
// Connect to the device & configure
//*********
// connect to device
result = wsaConnect(&wsaHandle, ip);
if (result < 0) {
    wsaErrorMessage_s(result, err_msg);
    printf("Error msg: %s\n", err_msg);
    return result;
}
printf("\nCurrent device status: \n");
// retrieve the *IDN command
result = wsaGetSCPI_s(wsaHandle, "*IDN?", query_result);
if (result < 0) {
    wsaErrorMessage_s(result, err_msg);
    printf("Error msg: %s\n", err_msg);
    return result;
}
printf("\tID: %s - %s\n\n", ip, query_result);
fflush(stdout);
// reset the device to default settings
// NOTE: using *RST affects the hardware settling time for DD mode, which
// takes ~ 2sec before data becomes useful
result = wsaSetSCPI(wsaHandle, "*RST");
// acquire read access
result = wsaGetSCPI_s(wsaHandle, ":SYSTEM:LOCK:REQUEST? ACQ", query_result);
// flush the device's internal memory
result = wsaSetSCPI(wsaHandle, ":SYSTEM:FLUSH");
// set the frequency
sprintf(cmd_str, "FREQ:CENT %f MHZ", input_freq);
result = wsaSetSCPI(wsaHandle, cmd_str);
// set the input mode
sprintf(cmd_str, "INPUT:MODE %s", rfe_mode);
result = wsaSetSCPI(wsaHandle, cmd_str);
// set the decimation
sprintf(cmd_str, "DECIMATION %d", decimation);
result = wsaSetSCPI(wsaHandle, cmd_str);
// set the frequency shift
sprintf(cmd_str, "FREQ:SHIFT %lf MHZ", freq_shift);
result = wsaSetSCPI(wsaHandle, cmd_str);
//*********
// Set the block size basing on PPB & SPP
//*********
if (!strcmp(rfe_mode, "HDR"))
    sample_rate = (float)WSA_NB_SAMPLE_RATE;
else
    sample_rate = (float)WSA_WB_SAMPLE_RATE;
// Determine the PPB basing on the RBW & SPP used
// Since this is a DD capture mode, bandwidth is 62.5 MHz
ppb = (int32_t) round(sample_rate / (spp * user_rbw * decimation));
if (ppb == 0)
    ppb = 1;
// Determine maximum PPB for the given SPP used
result = wsaGetSCPI_s(wsaHandle, "TRACE:BLOCK:PACKETS? MAX", query_result);
if (result < 0) {
    printf("Error %d, failed querry cmd\n", result);
}
```

```
    ppb_max = atoi(query_result);
    // Verify that ppb does not exceed the max, else set to the max allow
    if (ppb > ppb_max)
        ppb = ppb_max;
    block_samples = spp * ppb;
    // allocate memory for the usable spectral data
    usable_spectral_data = (float*)malloc(sizeof(float) * block_samples);
    if (usable_spectral_data == NULL) {
        printf("failed to locate memory.\n");
        return -1;
    }
    // Set and arm the capture
    result = wsaArmTraceCapture(wsaHandle, spp, ppb);
    if (result < 0) {
        free(usable_spectral_data);
        wsaErrorMessage_s(result, err_msg);
        printf("Error msg: %s\n", err_msg);
        return result;
    }
    // Get the trace data and compute its spectral result
    result = wsaCaptureSpectrumAndContext(wsaHandle, spp, ppb, timeout,
        &vrt_context_data, &pkt_header, usable_spectral_data, &usable_data_size,
        &usable_fstart, &usable_fstop, &usable_bandwidth);
    if ((result < 0) && (result != WSA_WARNING_SOCKETTIMEOUT)) {
        free(usable_spectral_data);
        wsaErrorMessage_s(result, err_msg);
        printf("Error msg: %s\n", err_msg);
        return result;
    }
    actual_rbw = usable_bandwidth / usable_data_size;
    printf("For SPP %d & PPB %d capture block, the actual RBW is: %lf Hz\n", spp, ppb, actual_rbw);
    // print GNSS context when available for RTSA with GNSS module
    if (vrt_context_data.has_gnss_data) {
        printf("\nGot GNSS Context VRT Info:\n");
        printf("\tManufacture OUI:    %08x\n", vrt_context_data.gnss_data.mfr_oui);
        printf("\tTimestamp of Position Fix: %d sec, %llu psec\n",
            vrt_context_data.gnss_data.posfix_sec, vrt_context_data.gnss_data.posfix_psec);
        printf("\tLatitude:           %lf degs\n", vrt_context_data.gnss_data.latitude);
        printf("\tLongtitude:         %lf degs\n", vrt_context_data.gnss_data.longitude);
        printf("\tAltitude:           %lf meters\n", vrt_context_data.gnss_data.altitude);
        printf("\tSpeed Over Ground:  %lf m/sec\n", vrt_context_data.gnss_data.speed_over_gnd);
        printf("\tHeading Angle:      %lf degs\n", vrt_context_data.gnss_data.heading_angle);
        printf("\tTrack Angle:        %lf degs\n", vrt_context_data.gnss_data.track_angle);
        printf("\tMagnetic Variation: %lf degs\n\n", vrt_context_data.gnss_data.magnetic_var);
    }
    // Compute the occupied bandwidth of the given % of the usable spectral data
    result = wsaComputePSDOccupiedBandwidth((uint64_t)actual_rbw,
        usable_data_size,
        usable_spectral_data,
        occupied_percentage,
        &occupied_bandwidth);
    if (result < 0) {
        wsaErrorMessage_s(result, err_msg);
        printf("Error msg: %s\n", err_msg);
        free(usable_spectral_data);
        return result;
    }
    // Display the channel power information
    printf("%f%% Occupied bandwith: %f MHz\n\n", occupied_percentage, (float)occupied_bandwidth / MHZ);
    // Compute the channel power for the whole usable spectral data
    result = wsaComputePSDChannelPower(
                0,
                usable_data_size,
                usable_data_size,
                usable_spectral_data,
                &channel_power);
    if (result < 0) {
        wsaErrorMessage_s(result, err_msg);
        printf("Error msg: %s\n", err_msg);
        free(usable_spectral_data);
        return result;
    }
    printf("Channel Power for the whole usable bandwidth (%.2f): %0.6f dBm\n\n",
        (float)usable_bandwidth / MHZ, channel_power);
    /******
     * Compute channel power for the user's range
     * Note, this example uses 'center_frequency' + 'freq_shift' for user's range
     ******/
    // 1st, Assign defaults to usable freqs with freq shift & then get user's inputs
    user_fstart = (uint64_t)usable_fstart;
    user_fstop = (uint64_t)usable_fstop;
    printf("The usable frequency range (with freq shifted) is %lf MHz - %lf MHz\n",
        (float)usable_fstart / MHZ, (float)usable_fstop / MHZ);
    printf("Enter a START FREQ for channel power computation, in MHz (0 to use default): ");
    scanf_s("%s", resp_str, (unsigned)_countof(resp_str));
    if (atol(resp_str) > 0)
        user_fstart = (uint64_t)(atof(resp_str) * MHZ);
    printf("Enter a STOP FREQ for channel power computation, in MHz (0 to use default): ");
    scanf_s("%s", resp_str, (unsigned)_countof(resp_str));
    if (atol(resp_str) > 0)
        user_fstop = (uint64_t)(atof(resp_str) * MHZ);
    printf("Frequencies received: %lf - %lf MHz\n\n",
        (float)user_fstart / MHZ, (float)user_fstop / MHZ);
    // 2nd, then verify user's frequencies with respect to adjusted usable range
    if (((int64_t)user_fstart < usable_fstart) || ((int64_t)user_fstop > usable_fstop)) {
        printf("ERROR: User's start and/or stop frequency is not within usable range.\n");
        printf("No channel power is computed.\n\n");
    }
    else {
        // 3rd, convert user's start & stop frequencies to the corresponding bins
        user_start_bin = (uint32_t)((user_fstart - usable_fstart) / actual_rbw);
        user_stop_bin = (uint32_t)((user_fstop - usable_fstart) / actual_rbw);
        //printf("%d %d\n", user_start_bin, user_stop_bin);
        result = wsaComputePSDChannelPower(
            user_start_bin,
            user_stop_bin,
            usable_data_size,
            usable_spectral_data,
            &channel_power);
        if (result < 0) {
            wsaErrorMessage_s(result, err_msg);
            printf("Error msg: %s\n", err_msg);
            free(usable_spectral_data);
            return result;
        }
        printf("Channel Power between %.2f and %.2f MHz: %0.6f dBm\n\n",
            (float)user_fstart / MHZ, (float)user_fstop / MHZ, channel_power);
    }
    // disconnect from device
    result = wsaDisconnect(wsaHandle);
    if (result < 0) {
        wsaErrorMessage_s(result, err_msg);
        printf("Error msg: %s\n", err_msg);
        free(usable_spectral_data);
        return result;
    }
    // free data buffer
    free(usable_spectral_data);
    system("PAUSE");
    return 0;
}
```

7.2 largeBlockCaptureWithPowerFnsAndDD.cpp

An example that demonstrates how to read a large block of data packet in DD mode (which covers 9kHz - 62.50 MHz range & center freq at 0).

Decimation of 4 and frequency shift -17.5 MHz are applied to bring the device's center frequency to 17.5 MHz and to cover the user's desired range of 10-25 MHz. (If decimation of 8 is used, the range covered would be narrower 12.5 MHz wide).

The example section starting from setting "TRACE:SPP" to calling wsaComputePSDUsableData() has been replaced by calling wsaCaptureSpectrumAndContext() instead.

Copyright

(C) ThinkRF Corporation 2015-2020. All rights reserved.

```
#include <tchar.h>
#include <stdio.h>
#include "wsaInterface.h"
int _tmain()
{
    // The device instance
    int64_t wsaHandle;
    char ip[50];
    // User related values
    int32_t user_rbw;
    float actual_rbw;
    int64_t user_fstart, user_fstop;
    uint32_t user_decimation = 1;
    uint32_t user_start_bin, user_stop_bin;
    int32_t user_attenuation = 0;
    // device settings
    const int32_t spp = 8192;   // samples per VRT packet
    int32_t ppb = 1;            // number of VRT packets per block
    int32_t ppb_max = 1;
    uint32_t block_samples;
    char * rfe_mode = "DD";
    float sample_rate = (float)WSA_WB_SAMPLE_RATE;
    float freq_shift = -17.5;   // MHz
    // variables for getting RFE span info
/*  uint32_t span_bw;
    int64_t span_center;
    int64_t span_fstart, span_fstop;*/
    // variables for data read
    uint32_t timeout = 5000;
    // to store outputs from the read
/*  int16_t i16_data[spp];
    int16_t q16_data[spp];
    int32_t i32_data[spp];  // for HDR mode
    int16_t *i_data;
    int16_t *q_data;
    uint32_t stream_id;
    uint8_t spectral_inversion;
    uint32_t timestamp_sec;
    uint64_t timestamp_psec;
    int16_t reference_level;
    uint64_t bandwidth;
    uint64_t center_frequency;

    // required buffer size for FFT & spectral data
    int32_t fft_size;
    float *spectral_data;*/
    struct vrt_header pkt_header;
    struct vrt_context vrt_context_data;
    // for PSD functions
    uint32_t usable_bandwidth;
    int64_t usable_fstart;
    int64_t usable_fstop;
    float *usable_spectral_data;
    uint32_t usable_data_size;
    int64_t usable_fstart_shifted;
    int64_t usable_fstop_shifted;
    int64_t peak_freq = 0;
    float peak_power = 0;
    float channel_power;
    float occupied_percentage = 90.5;
    uint64_t occupied_bandwidth;
    // Values to store function results
    int16_t result;
    char err_msg[1024];
    char cmd_str[256];
    char resp_str[512];
    char *query_result = resp_str;
    int i = 0;
    // grab IP from user
    printf("Enter an IP address: ");
    scanf_s("%s", ip, (unsigned)_countof(ip));
    printf("IP received: %s\n\n", ip);
    printf("Enter RBW in Hz: ");
    scanf_s("%s", resp_str, (unsigned)_countof(resp_str));
    if (atoi(resp_str) > 0) {
        user_rbw = atoi(resp_str);
    }
    else {
        printf("Invalid RBW provided. Must be a positive integer and greater than 0 Hz.\n");
        return -1;
    }
    printf("RBW received:  %d\n\n", user_rbw);
    printf("Enter an attenuation using one of values 0, 10, 20 or 30 (dB): ");
    scanf_s("%s", resp_str, (unsigned)_countof(resp_str));
    if ((atoi(resp_str) == 0) || (atoi(resp_str) == 10) ||
        (atoi(resp_str) == 20) || (atoi(resp_str) == 30)) {
        user_attenuation = atoi(resp_str);
    }
    else {
        printf("Invalid attenuation value, must be 0, 10, 20 or 30.\n");
        return -1;
    }
    printf("Attenuation received:  %d\n\n", user_attenuation);
    // For this example, allows only decimation of 4 or 8
    printf("Enter a decimation value 4 or 8: ");
    scanf_s("%s", resp_str, (unsigned)_countof(resp_str));
    if ((atoi(resp_str) == 4) || (atoi(resp_str) == 8)) {
        user_decimation = atoi(resp_str);
    }
    else {
        printf("Invalid decimation value, must be 4 or 8.\n");
        return -1;
    }
    printf("Decimation received:  %d\n\n", user_decimation);
    // defaulting freq values for decimation 4 or 8 for this example in particular
    if (user_decimation == 4) {
        user_fstart = 10 * MHZ;
        user_fstop = 25 * MHZ;
    } else {
        user_fstart = 11.25 * MHZ;
        user_fstop = 23 * MHZ;
    }
    printf("Enter START FREQ for channel power computation, in MHz (0 to use default): ");
    scanf_s("%s", resp_str, (unsigned)_countof(resp_str));
    if (atol(resp_str) > 0)
        user_fstart = (uint64_t)(atof(resp_str) * MHZ);
    printf("Enter STOP FREQ for channel power computation, in MHz (0 to use default): ");
    scanf_s("%s", resp_str, (unsigned)_countof(resp_str));
    if (atol(resp_str) > 0)
        user_fstop = (uint64_t)(atof(resp_str) * MHZ);
    printf("Frequencies received: %lf - %lf MHz\n\n", (float)user_fstart / MHZ, (float)user_fstop / MHZ);
    printf("Enter occupied bandwidth percentage, in %%: ");
    scanf_s("%s", resp_str, (unsigned)_countof(resp_str));
    if (atol(resp_str) > 0)
        occupied_percentage = (float)atof(resp_str);
    printf("Percentage received:  %lf %%\n\n", occupied_percentage);
    //*********
    // Connect to the device & configure
    //*********
    // connect to device and reset device to default settings
    result = wsaConnect(&wsaHandle, ip);
    if (result < 0) {
        wsaErrorMessage_s(result, err_msg);
        printf("Error msg: %s\n", err_msg);
        return result;
    }
    printf("\nCurrent device status: \n");
    // retrieve the *IDN command
    result = wsaGetSCPI_s(wsaHandle, "*IDN?", query_result);
    if (result < 0) {
        wsaErrorMessage_s(result, err_msg);
        printf("Error msg: %s\n", err_msg);
        return result;
    }
    printf("\tID: %s - %s\n\n", ip, query_result);
    fflush(stdout);
    // acquire read access
    result = wsaGetSCPI_s(wsaHandle, ":SYSTEM:LOCK:REQUEST? ACQ", query_result);
    if (result < 0) {
        wsaErrorMessage_s(result, err_msg);
        printf("Error msg: %s\n", err_msg);
        return result;
    }
    // flush the device's internal memory
    result = wsaSetSCPI(wsaHandle, ":SYSTEM:FLUSH");
    if (result < 0) {
        wsaErrorMessage_s(result, err_msg);
        printf("Error msg: %s\n", err_msg);
        return result;
    }
    // set the input mode
    sprintf(cmd_str, "INPUT:MODE %s", rfe_mode);
    result = wsaSetSCPI(wsaHandle, cmd_str);
    if (result < 0) {
        wsaErrorMessage_s(result, err_msg);
        printf("Error msg: %s\n", err_msg);
        return result;
    }
```

```
    result = wsaSetAttenuation(wsaHandle, user_attenuation);
    if (result < 0) {
        wsaErrorMessage_s(result, err_msg);
        printf("Error msg: %s\n", err_msg);
        return result;
    }
    // set the decimation
    sprintf(cmd_str, "DECIMATION %d", user_decimation);
    result = wsaSetSCPI(wsaHandle, cmd_str);
    if (result < 0) {
        wsaErrorMessage_s(result, err_msg);
        printf("Error msg: %s\n", err_msg);
        return result;
    }
    // set the frequency shift
    sprintf(cmd_str, "FREQ:SHIFT %lf MHZ", freq_shift);
    result = wsaSetSCPI(wsaHandle, cmd_str);
    if (result < 0) {
        wsaErrorMessage_s(result, err_msg);
        printf("Error msg: %s\n", err_msg);
        return result;
    }
    //*********
    // Set the block size basing on PPB & SPP
    //*********
    // Determine the PPB basing on the RBW & SPP used
    // Since this is a DD capture mode, bandwidth is 62.5 MHz
    ppb = (int32_t) round(sample_rate / (spp * user_rbw * user_decimation));
    if (ppb == 0)
        ppb = 1;
    // Determine maximum PPB for the given SPP used
    result = wsaGetSCPI_s(wsaHandle, "TRACE:BLOCK:PACKETS? MAX", query_result);
    if (result < 0) {
        wsaErrorMessage_s(result, err_msg);
        printf("Error msg: %s\n", err_msg);
        return result;
    }
    ppb_max = atoi(query_result);
    // Verify that ppb does not exceed the max, else set to the max allow
    if (ppb > ppb_max)
        ppb = ppb_max;
    block_samples = spp * ppb;
    // allocate memory for the usable spectral data
    usable_spectral_data = (float*)malloc(sizeof(float) * block_samples);
    if (usable_spectral_data == NULL) {
        printf("failed to locate memory.\n");
        return -1;
    }
    // Set and arm the capture
    result = wsaArmTraceCapture(wsaHandle, spp, ppb);
    if (result < 0) {
        free(usable_spectral_data);
        wsaErrorMessage_s(result, err_msg);
        printf("Error msg: %s\n", err_msg);
        return result;
    }
    // Get the trace data and compute its spectral result
    result = wsaCaptureSpectrumAndContext(wsaHandle, spp, ppb, timeout,
        &vrt_context_data, &pkt_header, usable_spectral_data, &usable_data_size,
        &usable_fstart, &usable_fstop, &usable_bandwidth);
    if ((result < 0) && (result != WSA_WARNING_SOCKETTIMEOUT)) {
        free(usable_spectral_data);
        wsaErrorMessage_s(result, err_msg);
        printf("Error msg: %s\n", err_msg);
        return result;
    }
    actual_rbw = usable_bandwidth / usable_data_size;
    printf("For SPP %d & PPB %d capture block, the actual RBW is: %lf Hz\n", spp, ppb, actual_rbw);
    // print GNSS context when available for RTSA with GNSS module
    if (vrt_context_data.has_gnss_data) {
        printf("\nGot GNSS Context VRT Info:\n");
        printf("\tManufacture OUI:    %08x\n", vrt_context_data.gnss_data.mfr_oui);
        printf("\tTimestamp of Position Fix: %d sec, %llu psec\n",
            vrt_context_data.gnss_data.posfix_sec, vrt_context_data.gnss_data.posfix_psec);
        printf("\tLatitude:           %lf degs\n", vrt_context_data.gnss_data.latitude);
        printf("\tLongtitude:         %lf degs\n", vrt_context_data.gnss_data.longitude);
        printf("\tAltitude:           %lf meters\n", vrt_context_data.gnss_data.altitude);
        printf("\tSpeed Over Ground:  %lf m/sec\n", vrt_context_data.gnss_data.speed_over_gnd);
        printf("\tHeading Angle:      %lf degs\n", vrt_context_data.gnss_data.heading_angle);
        printf("\tTrack Angle:        %lf degs\n", vrt_context_data.gnss_data.track_angle);
        printf("\tMagnetic Variation: %lf degs\n\n", vrt_context_data.gnss_data.magnetic_var);
    }
    // Compute the occupied bandwidth for the usable portion of spectral data
    result = wsaComputePSDOccupiedBandwidth((uint64_t) actual_rbw,
        usable_data_size,
        usable_spectral_data,
        occupied_percentage,
        &occupied_bandwidth);
    if (result < 0) {
        wsaErrorMessage_s(result, err_msg);
        printf("Error msg: %s\n", err_msg);
        free(usable_spectral_data);
        return result;
    }
    // Display the channel power information
    printf("%f%% Occupied bandwith: %f MHz\n\n", occupied_percentage, (float)occupied_bandwidth / MHZ);
    // Compute the channel power for the whole usable spectral data
    result = wsaComputePSDChannelPower(
                0,
                usable_data_size,
                usable_data_size,
                usable_spectral_data,
                &channel_power);
    if (result < 0) {
        wsaErrorMessage_s(result, err_msg);
        printf("Error msg: %s\n", err_msg);
        free(usable_spectral_data);
        return result;
    }
    printf("Channel Power for the whole usable bandwidth (%.2f): %0.6f dBm\n\n",
        (float)usable_bandwidth / MHZ, channel_power);
    // Mirror usable frequencies with fshift to the positive range
    usable_fstart_shifted = usable_fstart - 2 * freq_shift*MHZ;
    usable_fstop_shifted = usable_fstart_shifted + usable_bandwidth;
    // Then verify user's frequencies with respect to usable range
    if ((user_fstart < usable_fstart_shifted) || (user_fstop > usable_fstop_shifted)) {
        printf("ERROR: User's start and/or stop frequency is not within usable range.\n");
        printf("No channel power is computed.\n\n");
    }
    else {
        // convert user's start & stop frequencies to the corresponding bins
        // but use the adjusted usable fstart & fstop to keep the values positive
        user_start_bin = (uint32_t)((user_fstart - usable_fstart_shifted) / actual_rbw);
        user_stop_bin = (uint32_t)((user_fstop - usable_fstart_shifted) / actual_rbw);
        result = wsaComputePSDChannelPower(
            user_start_bin,
            user_stop_bin,
            usable_data_size,
            usable_spectral_data,
            &channel_power);
        if (result < 0) {
            wsaErrorMessage_s(result, err_msg);
            printf("Error msg: %s\n", err_msg);
            free(usable_spectral_data);
            return result;
        }
        printf("Channel Power between %.6f and %.6f MHz: %0.6f dBm\n\n",
            (float)user_fstart / MHZ, (float)user_fstop / MHZ, channel_power);
    }
    // disconnect from device
    result = wsaDisconnect(wsaHandle);
    if (result < 0) {
        wsaErrorMessage_s(result, err_msg);
        printf("Error msg: %s\n", err_msg);
        free(usable_spectral_data);
        return result;
    }
    // free data buffer
    free(usable_spectral_data);
    system("PAUSE");
    return 0;
}
```

7.3 simpleGNSSPacketsCaptureExample.cpp

A VRT packets read example code to illustrate how to use GNSS feature and get the GNSS VRT packets with or without doing data capture.

7.3 simpleGNSSPacketsCaptureExample.cpp

*** This example applies to RTSA products with GNSS module only. ***

Copyright

(C) ThinkRF Corporation 2015-2020. All rights reserved.

```
#ifdef _WIN32
#include <Windows.h>
#else
#include <unistd.h>
#endif
#include <stdio.h>
#include <time.h>
#include <tchar.h>
#include "wsaInterface.h"
 // Note: when DD mode is used right after reset state, it would take time for
 // DD mode to settle. Hence at least 2 seconds are needed before starting the
 // data capture.
#define RFE_MODE "SH"
#define FREQ     1003*MHZ
#define DEC   1
#define ATT   30
#define SPP   16384
#define PPB   2
#define LOOPS 2 // seconds
#define TIMEOUT 2000 // msec
int16_t runGNSSExampleTest(int64_t wsaHandle);
int _tmain()
{
    // Initialize WSA structure and IP setting
    int64_t wsaHandle;
    char ip[50];
    // Values to store function results
    int16_t result;
    char err_msg[1024];
    char resp_str[512];
    char *scpi_result = resp_str;
    int i = 0;
    // grab IP from user
    printf("Enter an IP address: ");
    scanf_s("%s", ip, (unsigned)_countof(ip));
    printf("IP received: %s\n\n", ip);
    // connect to device
    result = wsaConnect(&wsaHandle, ip);
    if (result < 0) {
        wsaErrorMessage_s(result, err_msg);
        printf("Error msg: %s\n", err_msg);
        return result;
    }
    printf("\nCurrent device status: \n");
    // retrieve the *IDN command
    result = wsaGetSCPI_s(wsaHandle, "*IDN?", scpi_result);
    if (result < 0) {
        wsaErrorMessage_s(result, err_msg);
        printf("Error msg: %s\n", err_msg);
        return result;
    }
    printf("\tID: %s - %s\n\n", ip, scpi_result);
    fflush(stdout);
    result = runGNSSExampleTest(wsaHandle);
    if (result < 0) {
        printf("Failed to complete GNSS packet capture!\n");
        wsaErrorMessage_s(result, err_msg);
        printf("Error msg: %s\n", err_msg);
    }
    // disconnect from device
    result = wsaDisconnect(wsaHandle);
    if (result < 0) {
        wsaErrorMessage_s(result, err_msg);
        printf("Error msg: %s\n", err_msg);
        return result;
    }
}
int16_t runGNSSExampleTest(int64_t wsaHandle)
{
    // Values to store function results
    int16_t result;
    char err_msg[1024];
    char cmd_str[256];
    char resp_str[512];
    char *scpi_result = resp_str;
    int i = 0;
    uint32_t m;
    int32_t total_pkt = 0;
    /*
     * For wsa_get_current_rfe_span_info testing
     */
    int64_t span_center;
    uint32_t span_bw;
    int64_t span_fstart, span_fstop;
    /*
     * For the spectral data output
     */
    float *spectral_data;
    uint32_t spectral_size;
    int64_t spectral_fstart;
    int64_t spectral_fstop;
    uint32_t spectral_bandwidth;
    int64_t peak_freq = 0;
    float peak_power = -200; // dbm
    uint32_t tmp_max_index = 0;
    float channel_power;
    uint64_t occupied_bandwidth;
    /*
     * For VRT Context
     */
    struct vrt_header header;
    struct vrt_context vrt_context_pkt;
    struct vrt_gnss_geolocn geolocn_data;
    int32_t has_gnss_data;
    /*
     * For GNSS
     */
    char ref_pll[16];
    char ref_pps[16];
    uint8_t gnss_enable;
    char gnss_fix_src[16];
    float latitude = 0.0, longitude = 0.0, altitude = 0.0;
    float new_latitude, new_longitude, new_altitude;
    int32_t delay = 0;
    char constels[32];
    /*
     * For running a 5 sec loop to get GNSS context pkts
     */
    time_t run_start_time;
    int32_t run_time = 0;
    /*
     * Configure the device
     */
     // reset the device to default settings
     // note: this could affect the GNSS module, requiring geolocation data being
     // reacquired
     //result = wsaSetSCPI(wsaHandle, "*RST");
      // acquire read access
    scpi_result = wsaGetSCPI(wsaHandle, ":SYSTEM:LOCK:REQUEST? ACQ");
    // flush the device's internal memory
    result = wsaSetSCPI(wsaHandle, ":SYSTEM:FLUSH");
    // set the frequency
    sprintf(cmd_str, "FREQ:CENT %llu", FREQ);
    result = wsaSetSCPI(wsaHandle, cmd_str);
    // set the input mode
    sprintf(cmd_str, "INPUT:MODE %s", RFE_MODE);
    result = wsaSetSCPI(wsaHandle, cmd_str);
    // set the decimation
    sprintf(cmd_str, "DECIMATION %d", DEC);
    result = wsaSetSCPI(wsaHandle, cmd_str);
    // set the attenuation
    result = wsaSetAttenuation(wsaHandle, ATT);
    printf("Block capture settings: %s, dec %d, att %d, spp %d, ppb %d\n",
        RFE_MODE, DEC, ATT, SPP, PPB);
    result = wsaGetRFESpan(wsaHandle, RFE_MODE, &span_bw, &span_center,
        &span_fstart, &span_fstop);
    printf("RFE span info: %lld, %d, %lld, %lld (Hz)\n\n", span_center, span_bw,
        span_fstart, span_fstop);
    /*
     * Set GNSS configuration and verify
     */
     // uncomment to use
     // Note: this would affect the GNSS module, causing geolocation data being
     // reacquired, which could take a few seconds
     /*result = wsaGNSSEnable(wsaHandle, 1);
```

```
     if (result < 0) {
         wsaErrorMessage_s(result, err_msg);
         printf("Error msg: %d %s\n", result, err_msg);
         return result;
     }*/
    result = wsaGetGNSSStatus(wsaHandle, &gnss_enable);
    if ((result < 0) || !gnss_enable) {
        printf("The GNSS module is not enabled. This programm will exit here.\n");
        return result;
    }
    //result = wsaSetReferencePLL(wsaHandle, "GNSS"); // uncomment to use
    result = wsaGetReferencePLL(wsaHandle, ref_pll);
    printf("GNSS Reference PLL source: %s\n", ref_pll);
    /*if (strcmp(ref_pll, "GNSS") != 0) {
        printf(" ==> ERROR: source returned does not matched with source set\n");
    }*/
    // Could add this to the data acquisition loop in case GNNS ref fix dropped
    result = wsaGetGNSSFixSource(wsaHandle, gnss_fix_src);
    printf("GNSS status: %s\n", (gnss_enable) ? "enabled" : "disabled");
    printf("Ref Fix Source: %s\n", gnss_fix_src);
    //result = wsaSetGNSSAntennaDelay(dev, 100);  // uncomment to use
    result = wsaGetGNSSAntennaDelay(wsaHandle, &delay);
    printf("Antenna cable delay: %d nsec\n\n", delay);
    //result = wsaSetReferencePPS(dev, "GNSS");  // uncomment to use
    result = wsaGetReferencePPS(wsaHandle, ref_pps);
    printf("GNSS Reference PPS source: %s\n\n", ref_pps);
    // Uncomment to use
    // Note: turning on this test would affect GNSS context info returned for
    // a few seconds while the system is re-acquiring the signals
    //result = wsaSetGNSSConstellation(wsaHandle, "GPS", "BEIDOU");
    result = wsaGetGNSSConstellation(wsaHandle, constels);
    /*  if(strcmp(constels, "GPS,BEIDOU") != 0) {
            printf("Constellation test failed, got %s\n\n", constels);
        }
    */
    printf("\n--------- Start the GNSS pkts capture (no data capture) for %d secs:\n",
        LOOPS);
    // Note: 1 GNSS context pkt is generated per second, so expecting up to
    // LOOPS+1 # of pkts
    time(&run_start_time);
    do {
        result = wsaReadGNSSContext(wsaHandle, TIMEOUT, &has_gnss_data, &geolocn_data);
        // if the error is a warning of timeout, ignore the error
        if ((result < 0) && (result != WSA_WARNING_SOCKETTIMEOUT)) {
            printf("Failed to read GNSS packet!\n");
            wsaErrorMessage_s(result, err_msg);
            printf("Error msg: %s\n", err_msg);
            return result;
        }
        // handle context packets
        if (has_gnss_data == 1) {
            printf("Got GNSS Context VRT Info:\n");
            printf("\tManufacture OUI:    %d 0x%08X\n",
                geolocn_data.mfr_oui, geolocn_data.mfr_oui);
            printf("\tTimestamp of Position Fix: %d sec, %llu psec\n",
                geolocn_data.posfix_sec, geolocn_data.posfix_psec);
            printf("\tLatitude:           %lf degs\n", geolocn_data.latitude);
            printf("\tLongtitude:         %lf degs\n", geolocn_data.longitude);
            printf("\tAltitude:           %lf meters\n", geolocn_data.altitude);
            printf("\tSpeed Over Ground:  %lf m/sec\n", geolocn_data.speed_over_gnd);
            printf("\tHeading Angle:      %lf degs\n", geolocn_data.heading_angle);
            printf("\tTrack Angle:        %lf degs\n", geolocn_data.track_angle);
            printf("\tMagnetic Variation: %lf degs\n\n", geolocn_data.magnetic_var);
            total_pkt++;
        }
        run_time = time(NULL) - run_start_time;
    } while (run_time <= LOOPS);
    printf("\nRun time: %d secs, got %d GNSS pkts\n", (run_time - 1), total_pkt);
    printf("--------- end GNSS context pkts only capture.\n");
    printf("\n--------- Start the block data & GNSS pkts capture test:\n");
    total_pkt = 0;
    /*
     * Set a trigger. Uncomment code to use.
     */
    //sprintf(cmd_str, "TRIG:LEVEL %llu,%llu,%d dBm", FREQ + MHZ, FREQ + (10 * MHZ / DEC), -100);
    //result = wsaSetSCPI(wsaHandle, cmd_str);
    //if (result < 0) {
    //  printf("Failed to set trigger level.\n");
    //  wsaErrorMessage_s(result, err_msg);
    //  printf("Error msg: %s\n", err_msg);
    //  return result;
```

```
//}
//sprintf(cmd_str, "TRIG:TYPE LEVEL");
//result = wsaSetSCPI(wsaHandle, cmd_str);
//if (result < 0) {
//  printf("Failed to set trigger type.\n");
//  wsaErrorMessage_s(result, err_msg);
//  printf("Error msg: %s\n", err_msg);
//  return result;
//}
int k = 0;
// start a new LOOPS time
time(&run_start_time);
do {
    //*********
    // Initialize buffer and start VRT data capture
    //*********
    spectral_data = (float *)malloc(sizeof(float) * SPP * PPB);
    if (spectral_data == NULL) {
        printf("Failed to allocate memory for the spectra data.\n");
        return -1;
    }
    // Get GNSS position, print only if new position data is available
    result = wsaGetGNSSPosition(wsaHandle, &new_latitude, &new_longitude, &new_altitude);
    if ((new_latitude != latitude) || (new_longitude != longitude) ||
        (new_altitude != altitude)) {
        printf("\nCurrent GNSS Position: %lf (deg), %lf (deg), %lf (meters)\n\n",
            new_latitude, new_longitude, new_altitude);
        latitude = new_latitude;
        longitude = new_longitude;
        altitude = new_altitude;
    }
    // Set and arm the capture
    result = wsaArmTraceCapture(wsaHandle, SPP, PPB);
    if (result < 0) {
        free(spectral_data);
        wsaErrorMessage_s(result, err_msg);
        printf("Error msg: %s\n", err_msg);
        return result;
    }
    // Get the trace data and compute its spectral result
    result = wsaCaptureSpectrumAndContext(wsaHandle, SPP, PPB, TIMEOUT,
        &vrt_context_pkt, &header, spectral_data, &spectral_size,
        &spectral_fstart, &spectral_fstop, &spectral_bandwidth);
    if ((result < 0) && (result != WSA_WARNING_SOCKETTIMEOUT)) {
        free(spectral_data);
        wsaErrorMessage_s(result, err_msg);
        printf("Error msg: %s\n", err_msg);
        return result;
    }
    if (i == 0) {
        // print some context info, only needed 1 time as they are the same
        // for the whole block
        printf("Receiver's Frequency: %llu Hz\n", vrt_context_pkt.cent_freq);
        printf("Ref Level: %d dbm\n", vrt_context_pkt.reference_level);
        printf("Bandwidth: %llu Hz\n", vrt_context_pkt.bandwidth);
        printf("Freq offset: %lld Hz\n\n", vrt_context_pkt.freq_offset);
        printf("Usable: BW %f MHz, fstart %f MHz, fstop %f MHz, data size %d\n\n",
            (float)spectral_bandwidth / MHZ, (float)spectral_fstart / MHZ,
            (float)spectral_fstop / MHZ, spectral_size);
    }
    // print GNSS context when available for RTSA with GNSS module
    if (vrt_context_pkt.has_gnss_data) {
        printf("\nGot GNSS Context VRT Info:\n");
        printf("\tManufacture OUI:     %08x\n", vrt_context_pkt.gnss_data.mfr_oui);
        printf("\tTimestamp of Position Fix: %d sec, %llu psec\n",
            vrt_context_pkt.gnss_data.posfix_sec, vrt_context_pkt.gnss_data.posfix_psec);
        printf("\tLatitude:           %lf degs\n", vrt_context_pkt.gnss_data.latitude);
        printf("\tLongtitude:         %lf degs\n", vrt_context_pkt.gnss_data.longitude);
        printf("\tAltitude:           %lf meters\n", vrt_context_pkt.gnss_data.altitude);
        printf("\tSpeed Over Ground:  %lf m/sec\n", vrt_context_pkt.gnss_data.speed_over_gnd);
        printf("\tHeading Angle:      %lf degs\n", vrt_context_pkt.gnss_data.heading_angle);
        printf("\tTrack Angle:        %lf degs\n", vrt_context_pkt.gnss_data.track_angle);
        printf("\tMagnetic Variation: %lf degs\n\n", vrt_context_pkt.gnss_data.magnetic_var);
        total_pkt++;
    }
    // If encountered socket timeout, that means no data, so should exit
    if (result == WSA_WARNING_SOCKETTIMEOUT) {
        free(spectral_data);
        break;
    }
    /*
```

```
         * Example signal processing applied to the spectral data
         * Uncomment to use as this slow down the capture process
         */
        //peak_power = -2000;
        //tmp_max_index = 0;
        // // Find the peak frequency of the usable spectral data
        //for (m = 0; m < spectral_size; m++) {
        //  if (spectral_data[m] > peak_power) {
        //      peak_power = spectral_data[m];
        //      tmp_max_index = m;
        //  }
        //}
        //peak_freq = spectral_fstart +
        //  (int64_t)((spectral_bandwidth / spectral_size) * tmp_max_index);
        //printf("Peak: Frequency: %0.2f MHz, Power: %0.2f dBm\n",
        //  (float)peak_freq / MHZ, peak_power);
        // Compute the channel power of the whole spectral
/*      result = wsaComputePSDChannelPower(
            0,
            spectral_size,
            spectral_size,
            spectral_data,
            &channel_power);
        if (result < 0) {
            wsaErrorMessage_s(result, err_msg);
            printf("Error msg: %s\n", err_msg);
            free(spectral_data);
            return result;
        }
        printf("Channel Power for usable bandwidth (%.2f): %0.6f dBm\n",
            (float)spectral_bandwidth / MHZ, channel_power);

        // Compute the occupied bandwidth for the usable portion of spectral data
        result = wsaComputePSDOccupiedBandwidth((uint64_t)(spectral_bandwidth / spectral_size),
            spectral_size,
            spectral_data,
            95.0,
            &occupied_bandwidth);
        if (result < 0) {
            wsaErrorMessage_s(result, err_msg);
            printf("Error msg: %s\n", err_msg);
            free(spectral_data);
            return result;
        }
        // Display the channel power information
        printf("%f%% Occupied bandwith: %f MHz\n\n", 95.0, (float)occupied_bandwidth / MHZ);
*/
        // Free up the current block mem allocation
        free(spectral_data);
        i++;
        run_time = time(NULL) - run_start_time;
    } while (run_time <= LOOPS);
    printf("\nRun time: %d secs, got %d GNSS pkts\n", (run_time - 1), total_pkt);

    printf("All capture done.\n");
    return result;
}
```

7.4 simpleHIF.cpp

An example script that demonstrates how to configure the RTSA for HIF usage.

Copyright

```
#include <iostream>
#include <tchar.h>
#include <stdio.h>
#include <stdlib.h>
#include "wsaInterface.h"
int _tmain()
{
    // The device instance
    int64_t wsaHandle;
    // Device IP
    char ip[50];
    // Values to store function results
    int16_t result;
    char input_str[512];
    char *scpi_result = input_str;
    // grab IP from user
    printf("Enter an IP address: ");
    scanf_s("%s", ip, (unsigned)_countof(ip));
    printf("IP received: %s\n\n", ip);
    // connect to device and reset device to default settings
    result = wsaConnect(&wsaHandle, ip);
    if (result < 0) {
        printf(wsaErrorMessage(result));
        printf("\n");
        return -1;
    }
    printf("Current device status: \n");
    // retrieve the *IDN command
    result = wsaGetSCPI_s(wsaHandle, "*IDN?", scpi_result);
    if (result < 0) {
        printf(wsaErrorMessage(result));
        printf("\n");
        return -1;
    }
    printf("\tID: %s - %s\n\n", ip, scpi_result);
    fflush(stdout);
    // reset the device
    result = wsaSetSCPI(wsaHandle, "*RST");
    // set the IQ path to connector
    result = wsaSetSCPI(wsaHandle, "OUTPUT:IQ:MODE CONNECTOR");
    // set the attenuation to 20 dB
    result = wsaSetSCPI(wsaHandle, "INPUT:ATT:VAR 20");
    // set the state of the PSFM Gain ( 0,0 =>  Low, 1,0 => Medium, 1,1 => High
    // set to Medium
    result = wsaSetSCPI(wsaHandle, ":INPUT:GAIN 1 1");
    result = wsaSetSCPI(wsaHandle, ":INPUT:GAIN 2 0");
    // set the frequency to 2 GHz
    result = wsaSetSCPI(wsaHandle, "FREQ:CENT 2 GHz");
    // retrieve the IF frequency
    scpi_result = wsaGetSCPI(wsaHandle, "FREQ:IF? -1");
    printf("LO Frequency: %s Hz\n", scpi_result);
    // close connection to device
    result = wsaDisconnect(wsaHandle);
    return 0;
}
```

7.5 simplePSDFunctionsExample.cpp

An example that demonstrates how to read a data packet (trace block capture), and does some power related computation.

Copyright

(C) ThinkRF Corporation 2015-2020. All rights reserved.

```
#include <tchar.h>
#include <stdio.h>
#include "wsaInterface.h"
int _tmain()
{
    // The device instance
    int64_t wsaHandle;
    char ip[50];
    // variables required for the read
    const int32_t spp = 8192; // samples per packet
    char * rfe_mode = "SH";
    const float input_freq = 1003; // in MHz
    int attenuation = 0;
    int16_t i16_data[spp];
    int16_t q16_data[spp];
    int32_t i32_data[spp];
    uint32_t timeout = 1000;
    // to store outputs from the read
    uint32_t stream_id;
    uint8_t spectral_inversion;
    uint32_t timestamp_sec;
    uint64_t timestamp_psec;
    int16_t reference_level;
    uint64_t bandwidth;
    uint64_t center_frequency;
    // required buffer size for FFT & spectral data
    int32_t fft_size;
    int8_t iq_imbalance_corr;
    float *spectral_data;
    // for PSD functions
    uint32_t rbw;
    uint32_t usable_bandwidth;
    int64_t usable_fstart;
    int64_t usable_fstop;
    float *usable_spectral_data;
    uint32_t usable_data_size;
    int64_t peak_freq = 0;
    float peak_power = 0;
    float channel_power;
    float occupied_percentage = 90.5;
    uint64_t occupied_bandwidth;
    // Values to store function results
    int16_t result;
    char err_msg[1024];
    char cmd_str[256];
    char response_str[512];
    char *scpi_result = response_str;
    // grab IP from user
    printf("Enter an IP address: ");
    scanf_s("%s", ip, (unsigned)_countof(ip));
    printf("IP received: %s\n\n", ip);
    // connect to device and reset device to default settings
    result = wsaConnect(&wsaHandle, ip);
    if (result < 0) {
        wsaErrorMessage_s(result, err_msg);
        printf("Error msg: %s\n", err_msg);
        return result;
    }
    printf("Current device status: \n");
    // retrieve the *IDN command
    result = wsaGetSCPI_s(wsaHandle, "*IDN?", scpi_result);
    if (result < 0) {
        wsaErrorMessage_s(result, err_msg);
        printf("Error msg: %s\n", err_msg);
        return result;
    }
    printf("\tID: %s - %s\n\n", ip, scpi_result);
    fflush(stdout);
    // reset the device to default settings
    result = wsaSetSCPI(wsaHandle, "*RST");
    // acquire read access
    scpi_result = wsaGetSCPI(wsaHandle, ":SYSTEM:LOCK:REQUEST? ACQ");
    // flush the device's internal memory
    result = wsaSetSCPI(wsaHandle, ":SYSTEM:FLUSH");
    // set the input mode
    sprintf(cmd_str, "INPUT:MODE %s", rfe_mode);
    result = wsaSetSCPI(wsaHandle, cmd_str);
    // set the frequency
    sprintf(cmd_str, "FREQ:CENT %f MHZ", input_freq);
    result = wsaSetSCPI(wsaHandle, cmd_str);
    // set the attenuation.  Note: for 408 model use "INPUT:ATT" instead
    sprintf(cmd_str, "INPUT:ATT:VAR %d", attenuation);
    result = wsaSetSCPI(wsaHandle, cmd_str);
    // set the SPP
    sprintf(cmd_str, "TRACE:SPP %d", spp);
    result = wsaSetSCPI(wsaHandle, cmd_str);
    // capture the data
    result = wsaSetSCPI(wsaHandle, ":TRACE:BLOCK:DATA?");
    // read the data
    result = wsaReadData(wsaHandle,
                    i16_data,
                    q16_data,
                    i32_data,
                    timeout,
                    &stream_id,
                    &spectral_inversion,
                    spp,
                    &timestamp_sec,
                    &timestamp_psec,
                    &reference_level,
                    &bandwidth,
                    &center_frequency);
    if (result < 0) {
        wsaErrorMessage_s(result, err_msg);
        printf("Error msg: %s\n", err_msg);
        return result;
    }
    printf("Captured timestamp: %d sec %llu psec\n", timestamp_sec, timestamp_psec);
    printf("Capture stream ID: %08x\n", stream_id);
    printf("Center frequency: %llu Hz\n", center_frequency);
    printf("Data bandwidth: %llu Hz\n", bandwidth);
    printf("Reference level: %d\n", reference_level);
    printf("Spectral inverstion status: %d\n\n", spectral_inversion);
    // determine FFT size required
    result = wsaGetFFTSize(spp, stream_id, &fft_size);
    // allocate memory for the spectral data variable
    spectral_data = (float*) malloc(sizeof(float) * fft_size);
    if (spectral_data == NULL) {
        printf("failed to locate memory.\n");
        return -1;
    }
    // turn on correction for when in ZIF mode
    if (!strcmp(rfe_mode, "ZIF"))
        iq_imbalance_corr = 1;
    else
        iq_imbalance_corr = 0;
    // compute the FFT & PSD
    result = wsaComputeFFTWithCorrOptions(spp,
        fft_size,
        (int32_t)stream_id,
        reference_level,
        1, // DC offset correction on
        iq_imbalance_corr,
        i16_data,
        q16_data,
        i32_data, // this is ignored in this example
        spectral_data);
    if (result < 0) {
        wsaErrorMessage_s(result, err_msg);
        printf("Error msg: %s\n", err_msg);
        free(spectral_data);
        return result;
    }
    //*********
    // Does some PSD data processing
    //*********
    // Find the peak frequency of the usable spectral data
    result = wsaPSDPeakFind(wsaHandle,
                rfe_mode,
                spectral_inversion,
                fft_size,
                spectral_data,
                &peak_freq,
                &peak_power);
    if (result < 0) {
        wsaErrorMessage_s(result, err_msg);
        printf("Error msg: %s\n", err_msg);
        free(spectral_data);
        return result;
    }
    // display the peak power, as well as the frequency location of the peak power
    printf("Peak: Frequency: %0.2f MHz, Power: %0.2f dBm\n", (float)peak_freq / MHZ, peak_power);
    // allocate memory for the usable spectral data
    usable_spectral_data = (float*)malloc(sizeof(float) * fft_size);
    if (usable_spectral_data == NULL) {
        printf("failed to locate memory.\n");
        free(spectral_data);
        return -1;
    }
    // compute the usable PSD data for the given info
    result = wsaComputePSDUsableData(wsaHandle,
        rfe_mode,
        spectral_inversion,
        spectral_data,
        fft_size,
        &usable_bandwidth,
        &usable_fstart,
        &usable_fstop,
        usable_spectral_data,
        &usable_data_size);
    if (result < 0) {
        wsaErrorMessage_s(result, err_msg);
        printf("Error msg: %s\n", err_msg);
        free(spectral_data);
        free(usable_spectral_data);
        return result;
    }
    printf("Usable: bw %f MHz, fstart %f MHz, fstop %f MHz, data size %d\n\n",
        (float)usable_bandwidth / MHZ, (float)usable_fstart / MHZ,
        (float)usable_fstop / MHZ, usable_data_size);
    free(spectral_data);
    // Compute the channel power for the usable spectral data
    result = wsaComputePSDChannelPower(
                0,
                usable_data_size,
                usable_data_size,
                usable_spectral_data,
                &channel_power);
    if (result < 0) {
        wsaErrorMessage_s(result, err_msg);
        printf("Error msg: %s\n", err_msg);
        free(usable_spectral_data);
        return result;
    }
    // Display the channel power information
    printf("Channel Power: %0.6f dBm\n", channel_power);
    rbw = (uint32_t) (usable_bandwidth / usable_data_size);
    occupied_percentage = 50.9;
    // Compute the occupied bandwidth for the usable portion of spectral data
    result = wsaComputePSDOccupiedBandwidth(rbw,
                usable_data_size,
                usable_spectral_data,
                occupied_percentage,
                &occupied_bandwidth);
    if (result < 0) {
        wsaErrorMessage_s(result, err_msg);
        printf("Error msg: %s\n", err_msg);
        free(usable_spectral_data);
        return result;
    }
    // Display the channel power information
    printf("%f%% Occupied bandwith: %f MHz\n", occupied_percentage, (float)occupied_bandwidth / MHZ);
    occupied_percentage = 96.5;
    // Compute the occupied bandwidth for the usable portion of spectral data
    result = wsaComputePSDOccupiedBandwidth(rbw,
        usable_data_size,
        usable_spectral_data,
        occupied_percentage,
        &occupied_bandwidth);
    if (result < 0) {
        wsaErrorMessage_s(result, err_msg);
        printf("Error msg: %s\n", err_msg);
        free(usable_spectral_data);
        return result;
    }
    // Display the channel power information
    printf("%f%% Occupied bandwith: %f MHz\n", occupied_percentage, (float)occupied_bandwidth / MHZ);
    occupied_percentage = 99.9;
    // Compute the occupied bandwidth for the usable portion of spectral data
    result = wsaComputePSDOccupiedBandwidth(rbw,
        usable_data_size,
        usable_spectral_data,
        occupied_percentage,
        &occupied_bandwidth);
    if (result < 0) {
        wsaErrorMessage_s(result, err_msg);
        printf("Error msg: %s\n", err_msg);
        free(usable_spectral_data);
        return result;
    }
    // Display the channel power information
    printf("%f%% Occupied bandwith: %f MHz\n", occupied_percentage, (float)occupied_bandwidth / MHZ);
    // disconnect from device
    result = wsaDisconnect(wsaHandle);
    if (result < 0) {
        wsaErrorMessage_s(result, err_msg);
        printf("Error msg: %s\n", err_msg);
        free(usable_spectral_data);
        return result;
    }
    // free data buffer
    free(usable_spectral_data);
    return 0;
}
```

7.6 simpleRead.cpp

An example illustrating how to read a VRT IF packet and compute the FFT.

Copyright

(C) ThinkRF Corporation 2015-2020. All rights reserved.

```
#include <tchar.h>
#include <stdio.h>
#include "wsaInterface.h"
int _tmain(int argc, _TCHAR* argv[])
{
    // The device instance
    int64_t wsaHandle;
    char ip[50];
    // variables required for the read
    const int32_t spp = 8192; // samples per packet
    char * rfe_mode = "SH";
    const float input_freq = 2450.0; // in MHz
    int16_t i16_data[spp];
    int16_t q16_data[spp];
    int32_t i32_data[spp];
    uint32_t timeout = 1000;
    // to store outputs from the read
    uint32_t stream_id;
    uint8_t spectral_inversion;
    uint32_t timestamp_sec;
    uint64_t timestamp_psec;
    int16_t reference_level;
    uint64_t bandwidth;
    uint64_t center_frequency;
    uint32_t span_bw;
    int64_t span_center;
    int64_t span_fstart, span_fstop;
    // required buffer size for FFT & spectral data
    int32_t fft_size;
    int8_t iq_imbalance_corr;
    float *spectral_data;
    // Values to store function results
    int i;
    int16_t result;
    char err_msg[1024];
    char cmd_str[256];
    char response_str[512];
    char *scpi_result = response_str;
    // grab IP from user
    printf("Enter an IP address: ");
    scanf_s("%s", ip, (unsigned)_countof(ip));
    printf("IP received: %s\n\n", ip);
    // connect to device and reset device to default settings
    result = wsaConnect(&wsaHandle, ip);
    if (result < 0) {
        wsaErrorMessage_s(result, err_msg);
        printf("Error msg: %s\n", err_msg);
        return result;
    }
    printf("Current device status: \n");
    // retrieve the *IDN command
    result = wsaGetSCPI_s(wsaHandle, "*IDN?", scpi_result);
    if (result < 0) {
        wsaErrorMessage_s(result, err_msg);
        printf("Error msg: %s\n", err_msg);
        return result;
    }
    printf("\tID: %s - %s\n\n", ip, scpi_result);
    fflush(stdout);
    // reset the device to default settings
    result = wsaSetSCPI(wsaHandle, "*RST");
    if (result < 0) {
        wsaErrorMessage_s(result, err_msg);
        printf("Error msg: %s\n", err_msg);
        return result;
    }
    // acquire read access
    scpi_result = wsaGetSCPI(wsaHandle, ":SYSTEM:LOCK:REQUEST? ACQ");
    // flush the device's internal memory
    result = wsaSetSCPI(wsaHandle, ":SYSTEM:FLUSH");
    // set the input mode
    sprintf(cmd_str, "INPUT:MODE %s", rfe_mode);
    result = wsaSetSCPI(wsaHandle, cmd_str);
    if (result < 0) {
        wsaErrorMessage_s(result, err_msg);
        printf("Error msg: %s\n", err_msg);
        return result;
    }
    // set the frequency
    sprintf(cmd_str, "FREQ:CENT %f MHZ", input_freq);
    result = wsaSetSCPI(wsaHandle, cmd_str);
    // set the trigger
    sprintf(cmd_str, ":TRIG:LEVEL %lf,%lf,%d\n", (input_freq-5)*MHZ, (input_freq+5)*MHZ, -80);
    result = wsaSetSCPI(wsaHandle, cmd_str);
    if (result < 0) {
        wsaErrorMessage_s(result, err_msg);
        printf("Error msg: %s\n", err_msg);
        return result;
    }
    sprintf(cmd_str, "TRIG:TYPE LEVEL\n");
    result = wsaSetSCPI(wsaHandle, cmd_str);
    if (result < 0) {
        wsaErrorMessage_s(result, err_msg);
        printf("Error msg: %s\n", err_msg);
        return result;
    }
    // set the SPP
    sprintf(cmd_str, "TRACE:SPP %d", spp);
    result = wsaSetSCPI(wsaHandle, cmd_str);
    // capture the data
    result = wsaSetSCPI(wsaHandle, ":TRACE:BLOCK:DATA?");
    // Added a loop for the case of triggering with no data returned
    while (1) {
        // read the data
        result = wsaReadData(wsaHandle,
                             i16_data,
                             q16_data,
                             i32_data,
                             timeout,
                             &stream_id,
                             &spectral_inversion,
                             spp,
                             &timestamp_sec,
                             &timestamp_psec,
                             &reference_level,
                             &bandwidth,
                             &center_frequency);
        if (result < 0) {
            if (result == WSA_WARNING_SOCKETTIMEOUT) {
                printf(".");
                fflush(stdout);
                continue;
            }
            wsaErrorMessage_s(result, err_msg);
            printf("Error msg: %s\n", err_msg);
            return result;
        }
        break;
    }
    printf("Captured time: %d sec %llu psec\n", timestamp_sec, timestamp_psec);
    printf("Capture stream ID: %08x\n", stream_id);
    printf("Center frequency: %llu Hz\n", center_frequency);
    printf("Data bandwidth: %llu Hz\n", bandwidth);
    printf("Reference level: %d\n", reference_level);
    printf("Spectral inverstion status: %d\n", spectral_inversion);
    // Find the frequency span
    result = wsaGetRFESpan(wsaHandle, rfe_mode,
        &span_bw, &span_center, &span_fstart, &span_fstop);
    if (result < 0) {
        wsaErrorMessage_s(result, err_msg);
        printf("Error msg: %s\n", err_msg);
        return result;
    }
    printf("Span info in Hz: bw: %0.2f, center: %0.2f, start freq: %0.2f, stop freq: %0.2f\n",
        (float)(span_bw) / MHZ, (float)(span_center) / MHZ,
        (float) (span_fstart) / MHZ, (float)(span_fstop) / MHZ);
    // determine FFT size required
    wsaGetFFTSize(spp, stream_id, &fft_size);
    // allocate memory for the spectral data variable
    spectral_data = (float*) malloc(sizeof(float) * fft_size);
    if (spectral_data == NULL) {
        printf("failed to locate memory.\n");
        return -1;
    }
    // turn on correction for when in ZIF mode
    if (!strcmp(rfe_mode, "ZIF"))
        iq_imbalance_corr = 1;
    else
        iq_imbalance_corr = 0;
    // compute the FFT & PSD
    result = wsaComputeFFTWithCorrOptions(spp,
                        fft_size,
                        (int32_t) stream_id,
                        reference_level,
                        1, // DC offset correction on
                        iq_imbalance_corr,
                        i16_data,
                        q16_data,
                        i32_data,
                        spectral_data);
    if (result < 0) {
        wsaErrorMessage_s(result, err_msg);
        printf("Error msg: %s\n", err_msg);
        free(spectral_data);
        return result;
    }
    // print out values
    printf("Spectral data:\n");
    for (i = 0; i < 10; i++)
        printf("%d, %0.2f; ", i+1, spectral_data[i]);
    printf("%d, %0.2f\n", fft_size, spectral_data[fft_size-1]);
    // disconnect from device
    result = wsaDisconnect(wsaHandle);
    if (result < 0) {
        wsaErrorMessage_s(result, err_msg);
        printf("Error msg: %s\n", err_msg);
        free(spectral_data);
        return result;
    }
    // free FFT buffer
    free(spectral_data);
    return 0;
}
```

7.7 simpleReadHDR.cpp

An example that demonstrates how to read a VRT If packet for HDR RFE mode, and compute the FFT/spectral data.

Copyright

```
#include <tchar.h>
#include <stdio.h>
#include "wsaInterface.h"
int _tmain()
{
    // The device instance
    int64_t wsaHandle;
    // variables required for the read
    const int32_t spp = 8192; // samples per packet
    int16_t i16_data[spp];
    int16_t q16_data[spp];
    int32_t i32_data[spp];
    uint32_t timeout = 1000;
    uint32_t stream_id;
    uint8_t spectral_inversion;
    uint32_t timestamp_sec;
    uint64_t timestamp_psec;
    int16_t reference_level;
```

```
uint64_t bandwidth;
uint64_t center_frequency;
//initialize buffer to store spectral data after FFT
float *spectral_data;
// required buffer size for spectral data
int32_t fft_size;
// Device IP
char ip[50];
// Values to store function results
int16_t result;
char input_str[512];
char *scpi_result = input_str;
// grab IP from user
printf("Enter an IP address: ");
scanf_s("%s", ip, (unsigned)_countof(ip));
printf("IP received: %s\n\n", ip);
// connect to device and reset device to default settings
result = wsaConnect(&wsaHandle, ip);
if (result < 0) {
    printf(wsaErrorMessage(result));
    printf("\n");
    return result;
}
printf("Current device status: \n");
// retrieve the *IDN command
result = wsaGetSCPI_s(wsaHandle, "*IDN?", scpi_result);
if (result < 0) {
    printf(wsaErrorMessage(result));
    printf("\n");
    return result;
}
printf("\tID: %s - %s\n\n", ip, scpi_result);
fflush(stdout);
result = wsaSetSCPI(wsaHandle, "*RST");
// acquire read access
scpi_result = wsaGetSCPI(wsaHandle, ":SYSTEM:LOCK:REQUEST? ACQ");
// set the input mode to HDR
result = wsaSetSCPI(wsaHandle, "INPUT:MODE HDR");
// set the frequency to 2 GHz
result = wsaSetSCPI(wsaHandle, "FREQ:CENT 2 GHz");
// set the SPP to 1024
result = wsaSetSCPI(wsaHandle, "TRACE:SPP 1024");
// capture the data
result = wsaSetSCPI(wsaHandle, ":TRACE:BLOCK:DATA?\n");
// read the data
result = wsaReadData(wsaHandle,
                    i16_data,
                    q16_data,
                    i32_data,
                    timeout,
                    &stream_id,
                    &spectral_inversion,
                    spp,
                    &timestamp_sec,
                    &timestamp_psec,
                    &reference_level,
                    &bandwidth,
                    &center_frequency);
if (result < 0) {
    printf(wsaErrorMessage(result));
    printf("\n");
    return result;
}
//determine FFT size required
result = wsaGetFFTSize(spp, stream_id, &fft_size);
// allocate data for the spectral data variable
spectral_data = (float*) malloc(sizeof(float) * fft_size);
if (spectral_data == NULL) {
    printf("failed to locate memory.\n");
    return -1;
}
// compute the FFT & PSD
result = wsaComputeFFTWithCorrOptions(spp,
    fft_size,
    (int32_t)stream_id,
    reference_level,
    1, // DC offset correction on
    0, // IQ imbalance correction is not needed for HDR
    i16_data, // this is ignored in this example
    q16_data, // this is ignored in this example
    i32_data,
        spectral_data);
    if (result < 0) {
        printf(wsaErrorMessage(result));
        printf("\n");
        free(spectral_data);
        return result;
    }
    // free FFT buffer
    free(spectral_data);
    // disconnect from device
    result = wsaDisconnect(wsaHandle);
    if (result < 0) {
        printf(wsaErrorMessage(result));
        printf("\n");
        return result;
    }
    // print out values
    for (int i = 0; i < (fft_size); i++)
        printf("%0.2f, ", spectral_data[i]);
    return 0;
}
```

7.8 streamExample.cpp

A simple example for how to do stream capture.In this example, the computeFFT is done within a loop. However, in a real application, the read data loop should focus on reading data only in order to increase the transfer (data throughput) rate and reduce drop packets due to memory overflow (as data is captured within the RTSA faster than transferring through a network). Therefore, process the data only after stream has been stopped.

Copyright

```
#include <tchar.h>
#include <conio.h>
#include <stdio.h>
#include <stdlib.h>
#include "wsaInterface.h"
#define DO_PRINT 0
#define DO_DSP 0
#define NUM_PKTS 1000
int _tmain()
{
    // The device instance
    int64_t wsaHandle;
    char ip[50];
    // variables required for the read
    const int32_t spp = 4096; // samples per packet
    char *rfe_mode = "SH";
    const float input_freq = 2450.0; // in MHz
    int decimation = 32;
    int attenuation = 10;
    int user_start_id;
    int16_t i16_data[spp];
    int16_t q16_data[spp];
    int32_t i32_data[spp];
    uint32_t timeout = 1000;
    // to store outputs from the read
    uint32_t stream_start_id;
    struct vrt_header pkt_header;
    struct vrt_context vrt_context_data;
    // required buffer size for FFT & spectral data
    int32_t fft_size;
    int8_t iq_imbalance_corr;
    float *spectral_data;
    // Values to store function results
    int16_t result;
    char err_msg[1024];
    char cmd_str[256];
    char response_str[512];
    char *scpi_result = response_str;
    // grab IP from user
    printf("Enter an IP address: ");
    scanf_s("%s", ip, (unsigned)_countof(ip));
    printf("IP received: %s\n\n", ip);
    // connect to device and reset device to default settings
    result = wsaConnect(&wsaHandle, ip);
    if (result < 0) {
        printf(wsaErrorMessage(result));
        printf("\n");
        return result;
    }
    printf("Current device status: \n");
    // retrieve the *IDN command
    result = wsaGetSCPI_s(wsaHandle, "*IDN?", scpi_result);
    if (result < 0) {
        wsaErrorMessage_s(result, err_msg);
        printf("Error msg: %s\n", err_msg);
        return result;
    }
    printf("\tID: %s - %s\n\n", ip, scpi_result);
    fflush(stdout);
    result = wsaSetSCPI(wsaHandle, "*RST");
    if (result < 0) {
        wsaErrorMessage_s(result, err_msg);
        printf("Error msg: %s\n", err_msg);
        return result;
    }

    // acquire read access
    scpi_result = wsaGetSCPI(wsaHandle, ":SYSTEM:LOCK:REQUEST? ACQ");
    result = wsaSetSCPI(wsaHandle, "SYSTEM:FLUSH");
    if (result < 0) {
        wsaErrorMessage_s(result, err_msg);
        printf("Error msg: %s\n", err_msg);
        return result;
    }
    // set the input mode
    sprintf(cmd_str, "INPUT:MODE %s", rfe_mode);
    result = wsaSetSCPI(wsaHandle, cmd_str);
    if (result < 0) {
        wsaErrorMessage_s(result, err_msg);
        printf("Error msg: %s\n", err_msg);
        return result;
    }
    // set the frequency
    sprintf(cmd_str, "FREQ:CENT %f MHZ", input_freq);
    result = wsaSetSCPI(wsaHandle, cmd_str);
    if (result < 0) {
        wsaErrorMessage_s(result, err_msg);
        printf("Error msg: %s\n", err_msg);
        return result;
    }
    // set decimation
    sprintf(cmd_str, ":SENSE:DEC %d", decimation);
    result = wsaSetSCPI(wsaHandle, cmd_str);
    if (result < 0) {
        wsaErrorMessage_s(result, err_msg);
        printf("Error msg: %s\n", err_msg);
        return result;
    }
    // set the attenuation
    result = wsaSetAttenuation(wsaHandle, attenuation);
    if (result < 0) {
        wsaErrorMessage_s(result, err_msg);
        printf("Error msg: %s\n", err_msg);
        return result;
    }
    // As this is a small stream test, assuming here that the data from
    // one packet to the next is continous & contiguous (no data loss)
    memset(i16_data, 0, spp * sizeof(int16_t));
    memset(q16_data, 0, spp * sizeof(int16_t));
    memset(i32_data, 0, spp * sizeof(int32_t));

    // turn on correction for when in ZIF mode
    if (!strcmp(rfe_mode, "ZIF"))
        iq_imbalance_corr = 1;
    else
        iq_imbalance_corr = 0;
    // get a random ID
    user_start_id = rand();
    // start the data streaming with an ID (some number)
    result = wsaStreamStartWithID(wsaHandle, spp, user_start_id);
    if (result < 0) {
        wsaErrorMessage_s(result, err_msg);
        printf("Error msg: %s\n", err_msg);
        return result;
    }
    // read the data
    try
    {
        for (int i = 0; i < NUM_PKTS && !_kbhit(); i++) {
            result = wsaCaptureStreamTraceAndContext(wsaHandle, spp, timeout,
                &vrt_context_data, &pkt_header, &stream_start_id,
                i16_data, q16_data, i32_data);
            if (result < 0) {
                wsaErrorMessage_s(result, err_msg);
                printf("Error msg: %s\n", err_msg);
                break;
            }
            if (i == 0) {
                if (stream_start_id != user_start_id) {
                    printf("Error: The start ID does not match value set.\n");
                    return -1;
                }
                printf("Stream context INFO received:\n");
                printf("\tVRT stream ID: %08x\n", pkt_header.stream_id);
                printf("\tSpectral inversion: %d\n", vrt_context_data.spectral_inversion);
                printf("\tSamples per packet: %d\n", spp);
                printf("\tReference level: %d\n", vrt_context_data.reference_level);
                printf("\tBandwidth: %lld\n", vrt_context_data.bandwidth);
                printf("\tCenter frequency: %lld\n\n", vrt_context_data.cent_freq);
            }
#if (DO_PRINT)
            printf("[%d] timestamp: %u sec, %llu psec; data loss: %u\n", i,
                pkt_header.timestamp_sec, pkt_header.timestamp_psec,
                vrt_context_data.sample_loss_indicator);
            printf("[%d] First 10 IQ data: [%d;%d]", i, i16_data[0], q16_data[0]);
            // print out values
            for (int j = 1; j < 10; j++) {
                printf(", [%d;%d]", i16_data[j], q16_data[j]);
            }
            printf("\n");
#else
            // Make sure to have the latest firmware to use the indicator
            if (vrt_context_data.sample_loss_indicator)
                printf("!(%d)", i);
            else
                printf(".");
            // print GNSS context when available for RTSA with GNSS module
            if (vrt_context_data.has_gnss_data) {
                printf("\nGot GNSS Context VRT Info:\n");
                printf("\tManufacture OUI:    %08x\n", vrt_context_data.gnss_data.mfr_oui);
                printf("\tTimestamp of Position Fix: %d sec, %llu psec\n",
                    vrt_context_data.gnss_data.posfix_sec, vrt_context_data.gnss_data.posfix_psec);
                printf("\tLatitude:           %lf degs\n", vrt_context_data.gnss_data.latitude);
                printf("\tLongtitude:         %lf degs\n", vrt_context_data.gnss_data.longitude);
                printf("\tAltitude:           %lf meters\n", vrt_context_data.gnss_data.altitude);
                printf("\tSpeed Over Ground:  %lf m/sec\n", vrt_context_data.gnss_data.speed_over_gnd);
                printf("\tHeading Angle:      %lf degs\n", vrt_context_data.gnss_data.heading_angle);
                printf("\tTrack Angle:        %lf degs\n", vrt_context_data.gnss_data.track_angle);
                printf("\tMagnetic Variation: %lf degs\n\n", vrt_context_data.gnss_data.magnetic_var);
            }
#endif
#if (DO_DSP)
            // determine FFT size required
            result = wsaGetFFTSize(spp, pkt_header.stream_id, &fft_size);
            //printf("[%d] fft_size: %d\n", i, fft_size);
            // allocate data for the spectral data variable
            spectral_data = (float*)malloc(sizeof(float) * fft_size);
            if (spectral_data == NULL) {
                printf("failed to locate memory.\n");
                return -1;
            }
            // compute the FFT & PSD
            // Note: processing data here as an example
            // it would slow down stream transfer rate
            result = wsaComputeFFTWithCorrOptions(spp,
                fft_size,
                (int32_t)pkt_header.stream_id,
                vrt_context_data.reference_level,
                1, // DC offset correction on
                iq_imbalance_corr,
                i16_data,
                q16_data,
                i32_data, // this is ignored in this example
                spectral_data);
            if (result < 0) {
                printf(wsaErrorMessage(result));
                printf("\n");
                free(spectral_data);
                break;
            }
            printf("[%d] First 10 Spectral data: %0.2f", i, spectral_data[0]);
            // print out values
            for (int k = 1; k < 10; k++) {
                printf(", %0.2f", spectral_data[k]);
            }
            printf("\n\n\n");
            // free FFT buffer
            free(spectral_data);
#endif
        }
    }
    catch (...)
    {
        printf("error");
        return -1;
    }
    result = wsaStreamStop(wsaHandle);
    if (result < 0) {
        wsaErrorMessage_s(result, err_msg);
        printf("Error msg: %s\n", err_msg);
        return result;
    }
    // disconnect from device
    result = wsaDisconnect(wsaHandle);
    if (result < 0) {
        wsaErrorMessage_s(result, err_msg);
        printf("Error msg: %s\n", err_msg);
        return result;
    }
    return 0;
}
```

7.9 sweepDeviceCalculateChannelPower.cpp

An example demonstrating how to calculate the channel power for a sweep-device capture.

Copyright

```
#include <tchar.h>
#include <stdio.h>
#include "wsaInterface.h"
int _tmain()
{
    // The device instance
    int64_t wsaHandle;
    // Device settings
    char ip[50];
    uint64_t fstart = 2250 * MHZ;
    uint64_t fstop = 2450 * MHZ;
    uint32_t rbw = 10000;
    char mode[20] = "SH";
    int32_t attenuator = 0;
    float ref_offset = 0;
    float channel_power = 0;
    // Values to store function results
    int16_t result;
    char err_msg[1024];
    char input_str[512];
    char *scpi_result = input_str;
    // grab IP from user
    printf("Enter an IP address: ");
    scanf_s("%s", ip, (unsigned)_countof(ip));
    printf("IP received: %s\n\n", ip);
    // connect to device and reset device to default settings
    result = wsaConnect(&wsaHandle, ip);
    if (result < 0) {
        wsaErrorMessage_s(result, err_msg);
        printf("Error msg: %s\n", err_msg);
        return -1;
    }
    printf("Current device status: \n");
    // retrieve the *IDN command
    result = wsaGetSCPI_s(wsaHandle, "*IDN?", scpi_result);
    if (result < 0) {
        wsaErrorMessage_s(result, err_msg);
        printf("Error msg: %s\n", err_msg);
        return -1;
    }
    printf("\tID: %s - %s\n\n", ip, scpi_result);
    fflush(stdout);
    result = wsaSetSCPI(wsaHandle, "*RST");
    result = wsaSetSCPI(wsaHandle, ":SYSTEM:ABORT");
    // acquire data read
    scpi_result = wsaGetSCPI(wsaHandle, ":SYSTEM:LOCK:REQUEST? ACQ");
    // call the sweep device's channel power computation
    result = wsaChannelPower(wsaHandle,
        fstart,
        fstop,
        rbw,
        mode,
        ref_offset,
        attenuator,
        &channel_power);
    if (result < 0) {
        wsaErrorMessage_s(result, err_msg);
        printf("Error msg: %s\n", err_msg);
        return -1;
    }
    // Display the channel power information
    printf("Channel Power %0.6f dBm\n", channel_power);
    result = wsaDisconnect(wsaHandle);
    if (result < 0) {
        wsaErrorMessage_s(result, err_msg);
        printf("Error msg: %s\n", err_msg);
        return -1;
    }
    system("PAUSE");
    return 0;
}
```

7.10 sweepDeviceCalculateOccupiedBandwidth.cpp

An example that demonstrates how to use the sweep-device's occupied bandwidth function

Copyright

(C) ThinkRF Corporation 2015-2020. All rights reserved.

```
#include <tchar.h>
#include <stdio.h>
#include "wsaInterface.h"
int _tmain()
{
    // The device instance
    int64_t wsaHandle;
    // Device settings
    char ip[50];
    uint64_t fstart = 2350000000;
    uint64_t fstop = 2450000000;
    uint32_t rbw = 10000;
    char mode[20] = "SH";
    float occupied_percentage = 92.5;
    int32_t attenuator = 0;
    // Variable to hold occupied bandwidth
    uint64_t occupied_bw = 0;
    // Values to store function results
    int16_t result;
    char err_msg[1024];
    char input_str[512];
    char *scpi_result = input_str;
    // grab IP from user
    printf("Enter an IP address: ");
    scanf_s("%s", ip, (unsigned)_countof(ip));
    printf("IP received: %s\n\n", ip);
    // connect to device and reset device to default settings
    result = wsaConnect(&wsaHandle, ip);
    if (result < 0) {
        wsaErrorMessage_s(result, err_msg);
        printf("Error msg: %s\n", err_msg);
        return -1;
    }
    printf("Current device status: \n");
    // retrieve the *IDN command
    result = wsaGetSCPI_s(wsaHandle, "*IDN?", scpi_result);
    if (result < 0) {
        wsaErrorMessage_s(result, err_msg);
        printf("Error msg: %s\n", err_msg);
        return -1;
    }
    printf("\tID: %s - %s\n\n", ip, scpi_result);
    fflush(stdout);
    result = wsaSetSCPI(wsaHandle, "*RST");
    result = wsaSetSCPI(wsaHandle, ":SYSTEM:ABORT");
    // acquire data read
    scpi_result = wsaGetSCPI(wsaHandle, ":SYSTEM:LOCK:REQUEST? ACQ");
    // call the peak find function
    result = wsaOccupiedBandwidth(wsaHandle,
        fstart,
        fstop,
        rbw,
        occupied_percentage,
        mode,
        attenuator,
        &occupied_bw);
    if (result < 0) {
        wsaErrorMessage_s(result, err_msg);
        printf("Error msg: %s\n", err_msg);
        return -1;
    }
    // display the peak power, as well as the frequency location of the peak power
    printf("Occupied Bandwidth %0.2f MHz\n", (float)occupied_bw / 1000000);
    result = wsaDisconnect(wsaHandle);
    if (result < 0) {
        wsaErrorMessage_s(result, err_msg);
        printf("Error msg: %s\n", err_msg);
        return -1;
    }
    system("PAUSE");
    return 0;
}
```

7.11 sweepDeviceCaptureSweepSpectrum.cpp

An example that demonstrates how to do data capture with sweep-device and get spectral data output.

Copyright

```
#include <tchar.h>
#include <time.h>
#include <stdio.h>
#include "wsaInterface.h"
int _tmain()
{
    // The device instance
    int64_t wsaHandle;
    // Device settings, change the IP to use it
    char ip[50];
    uint64_t fstart = 1 * MHZ;  // set to the min
    uint64_t fstop = 0;
    uint32_t rbw = 10000; // Min allow is ~2 Hz (no decimation)
    char mode[20] = "SH";
    int32_t attenuator = 0;
    // Variable to account for power offsets (cable loss, known gains, etc), dB
    float ref_offset = 0;
```

```
// The size of the expected spectral data
uint32_t sweep_size;
uint64_t fstart_actual;
uint64_t fstop_actual;
// Array to hold the spectral data
float *spectral_data;
// Variables to hold peak result
float peak_power = 0;
uint64_t peak_freq = 0;
// Variables to store function results
int16_t result;
char err_msg[1024];
char input_str[512];
char *scpi_result = input_str;
// Variables for timing how long the sweep capture take
clock_t run_start_time;
clock_t run_end_time;
float run_time = 0;
// grab IP from user
printf("Enter an IP address: ");
scanf_s("%s", ip, (unsigned)_countof(ip));
printf("IP received: %s\n\n", ip);
// Connect to device
result = wsaConnect(&wsaHandle, ip);
if (result < 0) {
    wsaErrorMessage_s(result, err_msg);
    printf("Error msg: %s\n", err_msg);
    return -1;
}
printf("Current device status: \n");
// retrieve the *IDN command
result = wsaGetSCPI_s(wsaHandle, "*IDN?", scpi_result);
if (result < 0) {
    wsaErrorMessage_s(result, err_msg);
    printf("Error msg: %s\n", err_msg);
    return -1;
}
printf("\tID: %s - %s %llu\n\n", ip, scpi_result, fstart);
fflush(stdout);
// Some system clean up before starting
result = wsaSetSCPI(wsaHandle, "*RST");
Sleep(1000); // since using DD range, allow settling time
result = wsaSetSCPI(wsaHandle, ":SYSTEM:ABORT");
if (result < 0) {
    wsaErrorMessage_s(result, err_msg);
    printf("Error msg: %s\n", err_msg);
    return -1;
}
result = wsaSetSCPI(wsaHandle, ":SYSTEM:FLUSH");
if (result < 0) {
    wsaErrorMessage_s(result, err_msg);
    printf("Error msg: %s\n", err_msg);
    return -1;
}
// request for data acquisition rights
scpi_result = wsaGetSCPI(wsaHandle, ":SYSTEM:LOCK:REQUEST? ACQ");
printf("ACQ request result: %s\n", scpi_result);
// Get the min/max value for a full spectrum sweep
result = wsaGetSCPI_s(wsaHandle, "FREQ:CENT? MAX", scpi_result);
fstop = strtoull(scpi_result, NULL, 0);
printf("Requested sweep frequencies: %lf MHz - %lf MHz, RBW: %u Hz\n",
    (float)fstart / MHZ, (float)fstop / MHZ, rbw);
// Retrieve the expected number of samples
result = wsaGetSweepSize_s(wsaHandle,
    fstart,
    fstop,
    rbw,
    mode,
    attenuator,
    &fstart_actual,
    &fstop_actual,
    &sweep_size);
if (result < 0) {
    wsaErrorMessage_s(result, err_msg);
    printf("Error msg in wsaGetSweepSize_s: %d %s\n", result, err_msg);
    return -1;
}
printf("Actual sweep frequencies: %lf MHz - %lf MHz\nSweep size: %d\n\n",
    (float)fstart_actual / MHZ, (float)fstop_actual / MHZ, sweep_size);
// Allocate memory for spectral data based on the number of samples
spectral_data = (float*)malloc(sizeof(float) * sweep_size);
```

```
    if (spectral_data == NULL) {
        printf("Failed to allocate memory\n");
        return -1;
    }
    printf("Start sweep capture\n");
    run_start_time = clock();
    // Capture spectral data
    result = wsaCaptureSpectrum(wsaHandle,
        fstart,
        fstop,
        rbw,
        mode,
        attenuator,
        spectral_data);
    run_end_time = clock();
    if (result < 0) {
        wsaErrorMessage_s(result, err_msg);
        printf("Error msg: %s\n", err_msg);
        free(spectral_data);
        return -1;
    }
    run_time = ((float)(run_end_time - run_start_time) / CLOCKS_PER_SEC) * 1000;
    printf(">>>> Capture run time: %.3f msec\n\n", run_time);
    // Print out some spectral data
    uint32_t i;
    for (i = 0; i < 100; i++) { //sweep_size; i++) {//
        printf("%0.2f, ", spectral_data[i]);
    }
    printf("\n\n");
    // perform a sweep device capture and call the peak find function
    result = wsaPeakFind(wsaHandle,
        fstart,
        fstop,
        rbw,
        mode,
        ref_offset,
        attenuator,
        &peak_freq,
        &peak_power);
    if (result < 0) {
        wsaErrorMessage_s(result, err_msg);
        printf("Error msg: %s\n", err_msg);
        return -1;
    }
    // display the peak power, as well as the frequency location of the peak power
    printf("Peak Frequency %f MHz, Peak Power %0.3f dBm\n\n", (float)peak_freq / 1000000, peak_power);
    // Disconnect from device
    result = wsaDisconnect(wsaHandle);
    if (result < 0) {
        wsaErrorMessage_s(result, err_msg);
        printf("Error msg: %s\n", err_msg);
        free(spectral_data);
        return -1;
    }
    system("PAUSE");
    free(spectral_data);
    return 0;
}
```

7.12 sweepDevicePeakFind.cpp

An example that demonstrates how to use the sweep-device's peak find function.

Copyright

(C) ThinkRF Corporation 2015-2020. All rights reserved.

```
#include <tchar.h>
#include <stdio.h>
#include "wsaInterface.h"
int _tmain()
{
    // The device instance
    int64_t wsaHandle;
    // Device settings
    char ip[50];
    uint64_t fstart = 100000;
    uint64_t fstop = 8000*MHZ;
    uint32_t rbw = 10000;
    char mode[20] = "SH";
    int32_t attenuator = 0;
    // Variable to account for power offsets (cable loss, known gains, etc)
    float ref_offset = 3;
    // Variables to hold peak result
    float peak_power = 0;
    uint64_t peak_freq = 0;
    // Values to store function results
    int16_t result;
    char err_msg[1024];
    char input_str[512];
    char *scpi_result = input_str;
    // grab IP from user
    printf("Enter an IP address: ");
    scanf_s("%s", ip, (unsigned)_countof(ip));
    printf("IP received: %s\n\n", ip);
    // connect to device and reset device to default settings
    result = wsaConnect(&wsaHandle, ip);
    if (result < 0) {
        wsaErrorMessage_s(result, err_msg);
        printf("Error msg: %s\n", err_msg);
        return -1;
    }
    printf("Current device status: \n");
    // retrieve the *IDN command
    result = wsaGetSCPI_s(wsaHandle, "*IDN?", scpi_result);
    if (result < 0) {
        wsaErrorMessage_s(result, err_msg);
        printf("Error msg: %s\n", err_msg);
        return -1;
    }
    printf("\tID: %s - %s\n\n", ip, scpi_result);
    fflush(stdout);
    result = wsaSetSCPI(wsaHandle, "*RST");
    result = wsaSetSCPI(wsaHandle, ":SYSTEM:ABORT");
    // acquire data read
    scpi_result = wsaGetSCPI(wsaHandle, ":SYSTEM:LOCK:REQUEST? ACQ");
    // perform a sweep device capture and compute the peak find function
    result = wsaPeakFind(wsaHandle,
                    fstart,
                    fstop,
                    rbw,
                    mode,
                    ref_offset,
                    attenuator,
                    &peak_freq,
                    &peak_power);
    if (result < 0) {
        wsaErrorMessage_s(result, err_msg);
        printf("Error msg: %s\n", err_msg);
        return -1;
    }
    // display the peak power, as well as the frequency location of the peak power
    printf("Peak Frequency %0.2f MHz, Peak Power %0.2f dBm \n", (float) peak_freq / 1000000, peak_power);
    result = wsaDisconnect(wsaHandle);
    if (result < 0) {
        wsaErrorMessage_s(result, err_msg);
        printf("Error msg: %s\n", err_msg);
        return -1;
    }
    system("PAUSE");
    return 0;
}
```

Index

altitude
  vrt_gnss_geolocn, 15

bandwidth
  vrt_context, 12

cent_freq
  vrt_context, 12

doxygen/api_doxy_doc.txt, 19

freq_offset
  vrt_context, 12

gain_if
  vrt_context, 12
gain_rf
  vrt_context, 12
getBuildInfo
  wsaInterface.h, 51
gnss_data
  vrt_context, 13

has_gnss_data
  vrt_context, 13
heading_angle
  vrt_gnss_geolocn, 16

indicator_field
  vrt_context, 12

latitude
  vrt_gnss_geolocn, 15
longitude
  vrt_gnss_geolocn, 15

magnetic_var
  vrt_gnss_geolocn, 16
mfr_oui
  vrt_gnss_geolocn, 14

packet_type
  vrt_header, 17
posfix_psec
  vrt_gnss_geolocn, 15
posfix_sec
  vrt_gnss_geolocn, 14

reference_level
  vrt_context, 13

sample_loss_indicator
  vrt_context, 13
spectral_inversion
  vrt_context, 13
speed_over_gnd
  vrt_gnss_geolocn, 16
stream_id
  vrt_header, 17

timestamp_psec
  vrt_header, 18
timestamp_sec
  vrt_header, 17
track_angle
  vrt_gnss_geolocn, 16

vrt_context, 11
  bandwidth, 12
  cent_freq, 12
  freq_offset, 12
  gain_if, 12
  gain_rf, 12
  gnss_data, 13
  has_gnss_data, 13
  indicator_field, 12
  reference_level, 13
  sample_loss_indicator, 13
  spectral_inversion, 13
vrt_gnss_geolocn, 14
  altitude, 15
  heading_angle, 16
  latitude, 15
  longitude, 15
  magnetic_var, 16
  mfr_oui, 14
  posfix_psec, 15
  posfix_sec, 14
  speed_over_gnd, 16
  track_angle, 16
vrt_header, 17
  packet_type, 17
  stream_id, 17
  timestamp_psec, 18

timestamp_sec, 17

wsaArmTraceCapture
wsaInterface.h, 26
wsaCaptureManualSweepTraceAndContext
wsaInterface.h, 31
wsaCaptureSpectrum
wsaInterface.h, 41
wsaCaptureSpectrumAndContext
wsaInterface.h, 26
wsaCaptureStreamTraceAndContext
wsaInterface.h, 29
wsaCaptureTraceAndContext
wsaInterface.h, 27
wsaChannelPower
wsaInterface.h, 43
wsaComputeFFT
wsaInterface.h, 36
wsaComputeFFTWithCorrOptions
wsaInterface.h, 35
wsaComputePSDChannelPower
wsaInterface.h, 38
wsaComputePSDOccupiedBandwidth
wsaInterface.h, 38
wsaComputePSDUsableData
wsaInterface.h, 39
wsaConnect
wsaInterface.h, 21
wsaDisconnect
wsaInterface.h, 22
wsaErrorMessage
wsaInterface.h, 50
wsaErrorMessage_s
wsaInterface.h, 50
wsaGetAttenuation
wsaInterface.h, 25
wsaGetFFTSize
wsaInterface.h, 34
wsaGetGNSSAntennaDelay
wsaInterface.h, 47
wsaGetGNSSConstellation
wsaInterface.h, 48
wsaGetGNSSFixSource
wsaInterface.h, 46
wsaGetGNSSPosition
wsaInterface.h, 46
wsaGetGNSSStatus
wsaInterface.h, 45
wsaGetReferencePLL
wsaInterface.h, 25
wsaGetReferencePPS
wsaInterface.h, 45
wsaGetRFESpan
wsaInterface.h, 49
wsaGetSCPI
wsaInterface.h, 24
wsaGetSCPI_s
wsaInterface.h, 23
wsaGetSweepSize
wsaInterface.h, 40
wsaGetSweepSize_s
wsaInterface.h, 40
wsaGNSSEnable
wsaInterface.h, 45
wsaInterface.h, 19
getBuildInfo, 51
wsaArmTraceCapture, 26
wsaCaptureManualSweepTraceAndContext, 31
wsaCaptureSpectrum, 41
wsaCaptureSpectrumAndContext, 26
wsaCaptureStreamTraceAndContext, 29
wsaCaptureTraceAndContext, 27
wsaChannelPower, 43
wsaComputeFFT, 36
wsaComputeFFTWithCorrOptions, 35
wsaComputePSDChannelPower, 38
wsaComputePSDOccupiedBandwidth, 38
wsaComputePSDUsableData, 39
wsaConnect, 21
wsaDisconnect, 22
wsaErrorMessage, 50
wsaErrorMessage_s, 50
wsaGetAttenuation, 25
wsaGetFFTSize, 34
wsaGetGNSSAntennaDelay, 47
wsaGetGNSSConstellation, 48
wsaGetGNSSFixSource, 46
wsaGetGNSSPosition, 46
wsaGetGNSSStatus, 45
wsaGetReferencePLL, 25
wsaGetReferencePPS, 45
wsaGetRFESpan, 49
wsaGetSCPI, 24
wsaGetSCPI_s, 23
wsaGetSweepSize, 40
wsaGetSweepSize_s, 40
wsaGNSSEnable, 45
wsaManualSweepStart, 30
wsaManualSweepStop, 31
wsaOccupiedBandwidth, 43
wsaPeakFind, 42
wsaPSDPeakFind, 37
wsaReadData, 32
wsaReadGNSSContext, 48
wsaReadVRTData, 33
wsaSetAttenuation, 24
wsaSetGNSSAntennaDelay, 47
wsaSetGNSSConstellation, 48

  wsaSetReferencePLL, 25
  wsaSetReferencePPS, 44
  wsaSetSCPI, 22
  wsaStreamStartWithID, 28
  wsaStreamStop, 29
wsaManualSweepStart
  wsaInterface.h, 30
wsaManualSweepStop
  wsaInterface.h, 31
wsaOccupiedBandwidth
  wsaInterface.h, 43
wsaPeakFind
  wsaInterface.h, 42
wsaPSDPeakFind
  wsaInterface.h, 37
wsaReadData
  wsaInterface.h, 32
wsaReadGNSSContext
  wsaInterface.h, 48
wsaReadVRTData
  wsaInterface.h, 33
wsaSetAttenuation
  wsaInterface.h, 24
wsaSetGNSSAntennaDelay
  wsaInterface.h, 47
wsaSetGNSSConstellation
  wsaInterface.h, 48
wsaSetReferencePLL
  wsaInterface.h, 25
wsaSetReferencePPS
  wsaInterface.h, 44
wsaSetSCPI
  wsaInterface.h, 22
wsaStreamStartWithID
  wsaInterface.h, 28
wsaStreamStop
  wsaInterface.h, 29