Reference Guide
74-0044

# ThinkRF RTSA C++ API v1.5.1

## Reference Guide

**Wed Jul 3 2019**

thinkRF
monitor. detect. analyze

# Contents

# 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 `Examples` directory.

## 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 v1.5.1 currently supports the following RTSA products and their associated models (408, 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/

## 1.3 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 built using the RTSA C-API. The C++ API download 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 SCPI Command Related

- wsaSetSCPI()
- wsaGetSCPI_s()
- wsaGetSCPI()
- wsaSetAttenuation()
- wsaGetAttenuation()
- wsaSetReferencePLL()
- wsaGetReferencePLL()

## 2.2 GNSS Functions

This section contains functions applied only for RTSA with GNSS modules.

### 2.2.1 GNSS Status and Settings

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

### 2.2.2 GNSS Context Packets Retrieval

- wsaSetGNSSConstellation()

## 2.3 Data or Trace Capture Functions

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

## 2.4 Data Processing Functions

### 2.4.1 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()

### 2.4.2 Power Spectral Density (PSD) Computation

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

## 2.5 Sweep-Device Functions

Functions in this sections apply to sweep-device capture and processing only.

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

## 2.6 Utility Functions

- wsaGetRFESpan()
- wsaErrorMessage_s()
- wsaErrorMessage()
- getBuildInfo()

# Chapter 3

# Data Structure Index

## 3.1 Data Structures

Here are the data structures with brief descriptions:

# Chapter 4

# File Index

## 4.1 File List

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

# Chapter 5

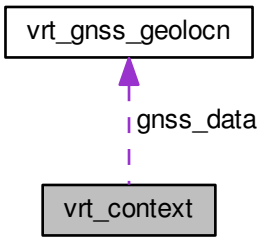
# Data Structure Documentation

## 5.1 vrt_context Struct Reference

This structure holds some of the VRT context packets's information.

`#include <wsaInterface.h>`

Collaboration diagram for vrt_context:




### Data Fields

- int32_t indicator_field
- uint64_t bandwidth
- uint64_t cent_freq
- int64_t freq_offset
- double gain_if
- double gain_rf
- int16_t reference_level
- 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:**

simpleGNSSPacketsCaptureExample.cpp.

### 5.1.2 Field Documentation

#### 5.1.2.1 indicator_field

`vrt_context::indicator_field`

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

#### 5.1.2.2 bandwidth

`vrt_context::bandwidth`

The bandwidth of the spectrum under the current settings, in Hz

#### 5.1.2.3 cent_freq

`vrt_context::cent_freq`

The center frequency of the RTSA, in Hz

#### 5.1.2.4 freq_offset

`vrt_context::freq_offset`

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

#### 5.1.2.5 gain_if

`vrt_context::gain_if`

The IF gain component of the RTSA, in dB

#### 5.1.2.6 gain_rf

`vrt_context::gain_rf`

The RF gain component of the RTSA, in dB

#### 5.1.2.7 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.

#### 5.1.2.8 gnss_data

`vrt_context::gnss_data`

A vrt_gnss_geolocn struct storing the GNSS data

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.

`#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

#### 5.2.2.1 mfr_oui

`vrt_gnss_geolocn::mfr_oui`

Manufacturer OUI of the GNSS.

#### 5.2.2.2 posfix_sec

`vrt_gnss_geolocn::posfix_sec`

The timestamp in second of the position fix.

#### 5.2.2.3 posfix_psec

`vrt_gnss_geolocn::posfix_psec`

The fractional timestamp in picosecond of the position fix.

#### 5.2.2.4 latitude

`vrt_gnss_geolocn::latitude`

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

**Examples:**

simpleGNSSPacketsCaptureExample.cpp.

#### 5.2.2.5 longitude

`vrt_gnss_geolocn::longitude`

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

**Examples:**

simpleGNSSPacketsCaptureExample.cpp.

#### 5.2.2.6 altitude

`vrt_gnss_geolocn::altitude`

The altitude of the geolocation, in meters

**Examples:**

simpleGNSSPacketsCaptureExample.cpp.

#### 5.2.2.7 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

#### 5.2.2.8 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.

#### 5.2.2.9 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.

#### 5.2.2.10 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.

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.

`#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.

See RTSA product's Programmer's Guide for more information.

**Examples:**

simpleGNSSPacketsCaptureExample.cpp.

### 5.3.2 Field Documentation

#### 5.3.2.1 packet_type

`vrt_header::packet_type`

The VRT packet types: IF data, context or extension

#### 5.3.2.2 stream_id

`vrt_header::stream_id`

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

#### 5.3.2.3 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.

#### 5.3.2.4 timestamp_psec

`vrt_header::timestamp_psec`

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

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 overiview and formatting information for the C++ API document.

### 6.1.1 Detailed Description

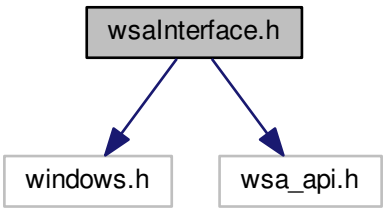
Provide overiview and formatting information for the C++ API document.

## 6.2 wsaInterface.h File Reference

The API header file with functions and their definition.

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

Include dependency graph for wsaInterface.h:




### Data Structures

- struct vrt_gnss_geolocn

  *This structure to hold VRT Formatted GNSS Geolocation.*

- struct vrt_header

  *This structure holds some of the VRT's header information.*
- struct vrt_context

  *This structure holds some of the VRT context packets's 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 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_pkt)
- int16_t wsaGetFFTSize (int32_t samples_per_packet, uint32_t stream_id, int32_t *array_size)
- int16_t wsaComputeFFT (int32_t samples_per_packet, 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.

## 6.2.2 Function Documentation

### 6.2.2.1 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() 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.cpp, and sweepDevicePeakFind.cpp.

### 6.2.2.2 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, simpleGNSSPackets↩CaptureExample.cpp, simpleHIF.cpp, simplePSDFunctionsExample.cpp, simpleRead.cpp, simpleReadH↩DR.cpp, streamExample.cpp, sweepDeviceCalculateChannelPower.cpp, sweepDeviceCalculateOccupied↩Bandwidth.cpp, sweepDeviceCaptureSweepSpectrum.cpp, and sweepDevicePeakFind.cpp.

#### 6.2.2.3 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, simpleGNSSPackets↩CaptureExample.cpp, simpleHIF.cpp, simplePSDFunctionsExample.cpp, simpleRead.cpp, simpleReadH↩DR.cpp, streamExample.cpp, sweepDeviceCalculateChannelPower.cpp, sweepDeviceCalculateOccupied↩Bandwidth.cpp, sweepDeviceCaptureSweepSpectrum.cpp, and sweepDevicePeakFind.cpp.

#### 6.2.2.4 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, simpleGNSSPackets↩CaptureExample.cpp, simpleHIF.cpp, simplePSDFunctionsExample.cpp, simpleRead.cpp, simpleReadH↩DR.cpp, streamExample.cpp, sweepDeviceCalculateChannelPower.cpp, sweepDeviceCalculateOccupied↩Bandwidth.cpp, sweepDeviceCaptureSweepSpectrum.cpp, and sweepDevicePeakFind.cpp.

### 6.2.2.5 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, sweepDevice↩CalculateOccupiedBandwidth.cpp, sweepDeviceCaptureSweepSpectrum.cpp, and sweepDevicePeakFind.cpp.

#### 6.2.2.6 wsaSetAttenuation()

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

Sets the attenuator's value. 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, and simpleGNSSPacketsCaptureExample.cpp.

#### 6.2.2.7 wsaGetAttenuation()

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

Gets the attenuator's value. See your RTSA's Programmer's Guide for the values. 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

#### 6.2.2.8 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

**Examples:**

simpleGNSSPacketsCaptureExample.cpp.

#### 6.2.2.9 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.

#### 6.2.2.10 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.

**Note:** Do not use this function if other VRT packets are also interested, see wsaReadVRTData() or wsaReadGNS↩ SContext().

**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 (milliseconds) |
| out | *stream_id* | - A pointer to store the IF data packet's stream id |
| out | *spectral_inversion* | - A pointer to to store the spectral inversion state, needed for when compute spectral data. If 1, the spectral data needs to be reverse. See Programmer's Guide. |
| 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 (pico seconds) |
| out | *reference_level* | - A pointer to store reference level to calibrate 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:**

largeBlockCaptureWithPowerFns.cpp, largeBlockCaptureWithPowerFnsAndDD.cpp, simplePSDFunctions↩ Example.cpp, simpleRead.cpp, simpleReadHDR.cpp, and streamExample.cpp.

#### 6.2.2.11 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_pkt )
```

Reads and returns the next available VRT data packet in the socket (see the product's Programmer's Guide for more info on the VRT protocol). If only wish to retrieve VRT IF data (I/Q time domain data), see wsaReadData() function.

These info (& the Programmer's Guide) are useful for understanding the output:

1. Use 'packet_type' member of 'header' struct to determine VRT pakcet types (IF_PACKET_TYPE, CONTEX↩ T_PACKET_TYPE, or EXTENSION_PACKET_TYPE).
2. Use 'stream_id' member of 'vrt_context_pkt' struct to determine what type of data this is (RECEIVER_STRE↩ AM_ID, DIGITIZER_STREAM_ID, EXTENSION_STREAM_ID, I16Q16_DATA_STREAM_ID, I16_DATA_ST↩ REAM_ID or I32_DATA_STREAM_ID)
3. For R57x0 with GNSS feature, might want to use GNSS_INDICATOR_MASK with 'indicator_field' of 'vrt_↩ context_pkt' struct. To run a long loop in order to gather GNSS context pkt, if the function's output is WS↩ A_WARNING_SOCKETTIMEOUT, could ignore that result. See the example called "simpleGNSSPackets↩ CaptureExample.cpp" for example.

**Parameters**

| | | |
|---|---|---|
| in | *wsa_handle* | - A 64-bit integer pointer which holds the device info |
| in | *samples_per_packet* | - The number of samples to read per VRT IF packet |
| in | *timeout* | - A value used to set the socket read timeout (milliseconds) |
| out | *header* | - A vrt_header struct pointer containing the VRT pkt information |
| 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 store the spectral inversion state, needed for when compute spectral data. If 1, the spectral data needs to be reverse. See RTSA Programmer's Guide. |
| out | *vrt_context_pkt* | - A vrt_context struct pointer containing the VRT context information |

**Returns**

0 on success or a negative value on error

**Examples:**

simpleGNSSPacketsCaptureExample.cpp.

#### 6.2.2.12 wsaGetFFTSize()

```
int16_t wsaGetFFTSize (
            int32_t samples_per_packet,
            uint32_t stream_id,
            int32_t * array_size )
```

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

**Parameters**

| | | |
|---|---|---|
| in | *samples_per_packet* | - The sample size of the time domain data |
| in | *stream_id* | - The stream id of the time domain data |
| out | *array_size* | - A pointer to store the size of the array |

**Returns**

0 on success, or a negative number on error

**Examples:**

largeBlockCaptureWithPowerFns.cpp, largeBlockCaptureWithPowerFnsAndDD.cpp, simpleGNSSPackets↩CaptureExample.cpp, simplePSDFunctionsExample.cpp, simpleRead.cpp, simpleReadHDR.cpp, and stream↩Example.cpp.

### 6.2.2.13 wsaComputeFFT()

```
int16_t wsaComputeFFT (
            int32_t samples_per_packet,
            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.

This function does the following:

- normalized the data
- applied DC Offset correction on I & Q data
- performed Hann Windowing and FFT
- compute PSD with calibrated ref level adjustment
- perform spectral inversion on spectra_data if spectral_inversion is 1.

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

**Parameters**

| | | |
|---|---|---|
| in | *samples_per_packet* | - 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 |

**Parameters**

| | | |
|---|---|---|
| in | *reference_level* | - The reference level to calibrate the spectral data (dBm) |
| in | *spectral_inversion* | - A pointer to to store the spectral inversion state. When 1, the fft_buffer data has been inverted. |
| 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:**

largeBlockCaptureWithPowerFns.cpp, largeBlockCaptureWithPowerFnsAndDD.cpp, simpleGNSSPackets↩CaptureExample.cpp, simplePSDFunctionsExample.cpp, simpleRead.cpp, simpleReadHDR.cpp, and stream↩Example.cpp.

#### 6.2.2.14 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 wsa↩ComputeFFT(). Use for a non-sweep-device capture mode.
Recommend to use this function for peak finding as SH/SHN/HDR's IF center frequency is not at 0 IF.

**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) |
| 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:**

largeBlockCaptureWithPowerFns.cpp, largeBlockCaptureWithPowerFnsAndDD.cpp, simpleGNSSPackets↩CaptureExample.cpp, and simplePSDFunctionsExample.cpp.

#### 6.2.2.15 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.
spectral_data could be derived from I/Q data by calling:

- wsaComputeFFT() for trace capture and manual sweep mode (using SCPI sweep setup)
- wsaCaptureSpectrum() for sweep device capture

**Parameters**

| | | |
|---|---|---|
| in | *start_bin* | - The first bin where the channel power calculation should take place |
| in | *stop_bin* | - The last bin 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 simplePSD↩FunctionsExample.cpp.

#### 6.2.2.16 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. spectral_data of I/Q data could be derived from:

- in trace block capture mode, call wsaComputeFFT(), but recommend to apply wsaComputePSDUsableData() on the spectral_data before calling this function.
- with sweep-device mode, call wsaCaptureSpectrum().

**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 simplePSD↩ FunctionsExample.cpp.

### 6.2.2.17 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.
Notes:

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

- If the spectral data is not derived from wsaComputeFFT(), make sure the spectral inversion has been taken care of first.

**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 |
| 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:**

largeBlockCaptureWithPowerFns.cpp, largeBlockCaptureWithPowerFnsAndDD.cpp, and simplePSD↩FunctionsExample.cpp.

#### 6.2.2.18 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().

#### 6.2.2.19 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 being programmed into the device. 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 IBW (i.e. 40 MHz for SH & 10 MHz for SHN), and they are useful for plotting.

**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 sweep start freq programmed in the device |
| out | *fstop_actual* | - the actual sweep stop freq programmed in the device |
| 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.

#### 6.2.2.20 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.
**Notes:**

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() for this function.

**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 | *spectral_data* | - Array to hold the output power spectral density data |

**Returns**

0 on success, or a negative number on error

**Examples:**

sweepDeviceCaptureSweepSpectrum.cpp.

#### 6.2.2.21 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.
**Note:** you must acquire read status before calling 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:**

sweepDevicePeakFind.cpp.

#### 6.2.2.22 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.
**Notes:**

- You must acquire read status before calling this function
- 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.

#### 6.2.2.23 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.
**Notes:**

- You must acquire read status before calling this function
- 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.

#### 6.2.2.24 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.

#### 6.2.2.25 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.

#### 6.2.2.26 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

**Examples:**

simpleGNSSPacketsCaptureExample.cpp.

#### 6.2.2.27 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 or & 0 off |

**Returns**

0 on success or a negative value on error

**Examples:**

simpleGNSSPacketsCaptureExample.cpp.

#### 6.2.2.28 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.

#### 6.2.2.29 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.

#### 6.2.2.30 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

#### 6.2.2.31 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.

#### 6.2.2.32 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

#### 6.2.2.33 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) |

**Returns**

0 on success or a negative value on error

#### 6.2.2.34 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 info). If wish to retrieve both VRT IF data (I/Q time domain data) and VRT context packets, see wsaReadVRTData() function.

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* | - A 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.

#### 6.2.2.35 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 |
| 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:**

largeBlockCaptureWithPowerFns.cpp, simpleGNSSPacketsCaptureExample.cpp, and simpleRead.cpp.

#### 6.2.2.36 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, simpleGNSSPackets↩CaptureExample.cpp, simplePSDFunctionsExample.cpp, simpleRead.cpp, streamExample.cpp, sweep↩DeviceCalculateChannelPower.cpp, sweepDeviceCalculateOccupiedBandwidth.cpp, sweepDeviceCapture↩SweepSpectrum.cpp, and sweepDevicePeakFind.cpp.

#### 6.2.2.37 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.

#### 6.2.2.38 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.

```
#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 = "HDR";
    float sample_rate;
    float input_freq = 100.0; // 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 getting RFE span info
    uint32_t span_bw;
    int64_t span_center;
    int64_t span_fstart, span_fstop;

    // variables for data 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 timeout = 5000;

    // 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;
float *spectral_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 -1;
}

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 -1;
}
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;

// set the SPP
sprintf(cmd_str, "TRACE:SPP %d", spp);
result = wsaSetSCPI(wsaHandle, cmd_str);

// set the PPB
sprintf(cmd_str, "TRACE:BLOCK:PACKETS %d", ppb);
result = wsaSetSCPI(wsaHandle, cmd_str);


//*********
// Initialize buffers and start data capture
//*********
i_data = (int16_t *)malloc(sizeof(int16_t) * block_samples);
if (i_data == NULL) {
    printf("Failed to allocate memory for the block.\n");
    return -1;
}

q_data = (int16_t *)malloc(sizeof(int16_t) * block_samples);
if (q_data == NULL) {
    printf("Failed to allocate memory for the block.\n");
    free(i_data);
    return -1;
}

// initiate the block data capture
result = wsaSetSCPI(wsaHandle, ":TRACE:BLOCK:DATA?");
```

```
if (result < 0) {
    free(i_data);
    free(q_data);
    return -1;
}

// loop to read the date & stitch it together
for (i = 0; i < ppb; i++) {
    // 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 -1;
    }

    // append each new spp data to the collective array
    memcpy(i_data+(i * spp), i16_data, spp * sizeof(int16_t));
    memcpy(q_data+(i * spp), q16_data, spp * sizeof(int16_t));
}
printf("Data bandwidth: %llu Hz\n", bandwidth);

// determine FFT size required
result = wsaGetFFTSize(block_samples, stream_id, &fft_size);

result = wsaGetRFESpan(wsaHandle, rfe_mode, &span_bw, &span_center, &span_fstart, &
  span_fstop);
actual_rbw = (float)span_bw / fft_size;
// if 'block_size' is used in place of 'fft_size', note that if bandwith is
// of IDATA only, it has to scaled up by 2 to reflect 'IQ' rate
//if ((span_bw == WSA_WB_IDATA_BW) || (span_bw == WSA_NB_IDATA_BW)
//  actual_rbw = actual_rbw * 2;
printf("For SPP %d & PPB %d capture block, bw %d Hz, the actual RBW is: %lf Hz\n", spp, ppb, span_bw,
  actual_rbw);

// 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");
    free(i_data);
    free(q_data);
    return -1;
}
printf("FFT (or PSD) data size: %d\n\n", fft_size);

// compute the FFT & PSD, with spectral data inverted when needed
result = wsaComputeFFT(block_samples,
                    fft_size,
                    (int32_t) stream_id,
                    reference_level,
                    spectral_inversion,
                    i_data,
                    q_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(i_data);
    free(q_data);
    free(spectral_data);
    return -1;
}

free(i_data);
free(q_data);

//*********
// Does some PSD data processing
```

```
    //*********

    // 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 -1;
    }
    printf("Usable %s data info with freq shift:\n\t BW %f MHz, fstart %f MHz, fstop %f MHz, data size %d\n
      \n",
        rfe_mode,
        (float)usable_bandwidth / MHZ, (float)usable_fstart / MHZ,
        (float)usable_fstop / MHZ, usable_data_size);


    // 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 -1;
    }

    // display the peak power, as well as the frequency location of the peak power
    printf("Peak detected: Frequency: %0.2f MHz, Power: %0.2f dBm\n\n",
        (float)peak_freq / MHZ - freq_shift, peak_power);

    free(spectral_data);

    // 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 -1;
    }
    printf("Channel Power for 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 -1;
        }
        printf("Channel Power between %.2f and %.2f MHz: %0.6f dBm\n\n",
            (float)user_fstart / MHZ, (float)user_fstop / MHZ, channel_power);
    }

    // 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 -1;
    }
    // Display the channel power information
    printf("%f%% Occupied bandwith: %f MHz\n\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 -1;
    }

    // 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 then computes the FFT & PSD and follows with some power related computation within user's specified frequency range.

```
#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
    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 timeout = 5000;

    // 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;
    float *spectral_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 -1;
```

```
    }

    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 -1;
    }
    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 -1;
    }

    // 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 -1;
    }

    // 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 -1;
    }

    result = wsaSetAttenuation(wsaHandle, user_attenuation);
    if (result < 0) {
        wsaErrorMessage_s(result, err_msg);
        printf("Error msg: %s\n", err_msg);
        return -1;
    }

    // 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 -1;
    }

    // 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 -1;
    }

    //*********
    // 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 -1;
    }
    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;

// result = wsaGetRFESpan(wsaHandle, rfe_mode, &span_bw, &span_center, &span_fstart, &span_fstop);
// actual_rbw = (float)span_bw / block_samples;
// if bandwith is of I-data only, it has to scaled up by 2 to reflect 'IQ' rate
//if (span_bw == WSA_WB_IDATA_BW)
//  actual_rbw = actual_rbw * 2;
actual_rbw = (float)sample_rate / (block_samples * user_decimation);
printf("For SPP %d & PPB %d capture block, the actual RBW is: %lf Hz\n", spp, ppb, actual_rbw);

// set the SPP
sprintf(cmd_str, "TRACE:SPP %d", spp);
result = wsaSetSCPI(wsaHandle, cmd_str);

// set the PPB
sprintf(cmd_str, "TRACE:BLOCK:PACKETS %d", ppb);
result = wsaSetSCPI(wsaHandle, cmd_str);


//*********
// Initialize buffers and start data capture
//*********
i_data = (int16_t *)malloc(sizeof(int16_t) * block_samples);
if (i_data == NULL) {
    printf("Failed to allocate memory for the block.\n");
    return -1;
}

q_data = (int16_t *)malloc(sizeof(int16_t) * block_samples);
if (q_data == NULL) {
    printf("Failed to allocate memory for the block.\n");
    free(i_data);
    return -1;
}

// initiate the block data capture
result = wsaSetSCPI(wsaHandle, ":TRACE:BLOCK:DATA?");
if (result < 0) {
    free(i_data);
    free(q_data);
    return -1;
}

// loop to read the data & stitch it together
for (i = 0; i < ppb; i++) {
    // 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 -1;
    }

    // append each new spp data to the collective array
    memcpy(i_data+(i * spp), i16_data, spp * sizeof(int16_t));
    memcpy(q_data+(i * spp), q16_data, spp * sizeof(int16_t));
}
printf("Data bandwidth: %llu Hz\n", bandwidth);

// determine FFT size required
result = wsaGetFFTSize(block_samples, 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");
        free(i_data);
        free(q_data);
        return -1;
    }
    printf("FFT (or PSD) data size: %d\n\n", fft_size);

    // compute the FFT & PSD, with spectral data inverted when needed
    result = wsaComputeFFT(block_samples,
                        fft_size,
                        (int32_t) stream_id,
                        reference_level,
                        spectral_inversion,
                        i_data,
                        q_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(i_data);
        free(q_data);
        free(spectral_data);
        return -1;
    }

    free(i_data);
    free(q_data);

    //*********
    // Does some PSD data processing
    //*********

    // 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 -1;
    }
    printf("Usable %s data info with freq shift:\n\t BW %f MHz, fstart %f MHz, fstop %f MHz, data size %d\n
      \n",
        rfe_mode,
        (float)usable_bandwidth / MHZ, (float)usable_fstart / MHZ,
        (float)usable_fstop / MHZ, usable_data_size);


    // 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 -1;
    }
    printf("Peak detected (before frequency shift adjusted): Frequency: %0.2f MHz, Power: %0.2f dBm\n",
        (float)peak_freq / MHZ, peak_power);
```

```
printf("Peak detected (after frequency shift adjusted): Frequency: %0.2f MHz, Power: %0.2f dBm\n\n",
    (float)peak_freq / MHZ - 2*freq_shift, peak_power);

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 -1;
}
printf("Channel Power for 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 -1;
    }
    printf("Channel Power between %.6f and %.6f MHz: %0.6f dBm\n\n",
        (float)user_fstart / MHZ, (float)user_fstop / MHZ, channel_power);
}


// 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 -1;
}
// Display the channel power information
printf("%f%% Occupied bandwith: %f MHz\n\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 -1;
}

// 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.

This example applies to RTSA products with GNSS module only.

```
#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     100*MHZ
#define DEC   0
#define ATT   0
#define SPP   16384
#define PPB   5

#define LOOPS 5 // seconds
#define TIMEOUT 100 // 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 -1;
    }

    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 -1;
    }
    printf("\tID: %s - %s\n\n", ip, scpi_result);
```

```
    fflush(stdout);

    result = runGNSSExampleTest(wsaHandle);
    if (result < 0) {
        printf("Failed to do GNSS packet capture tests!\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 -1;
    }
}

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;

    int32_t fft_size = 0;
    int64_t peak_freq;
    float peak_power;
    int i = 0, k = 0;
    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;


    /*
     * Buffers to store the decoded I & Q from the raw data
     */
    int16_t i16_data[SPP];
    int16_t q16_data[SPP];
    int32_t i32_data[SPP];
    int16_t *i_data;
    int16_t *q_data;
    float *spectral_data;
    uint8_t spectral_inversion;

    /*
     * 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 gnss_ref[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
    //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
     */
    result = wsaGNSSEnable(wsaHandle, 1);
    result = wsaGetGNSSStatus(wsaHandle, &gnss_enable);

    result = wsaSetReferencePLL(wsaHandle, "GNSS"); // to ensure it's set
    result = wsaGetReferencePLL(wsaHandle, gnss_ref);
    printf("GNSS Reference source: %s\n", gnss_ref);
    if (strcmp(gnss_ref, "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);

/*
    // 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(dev, "GPS", "BEIDOU");
    result = wsaGetGNSSConstellation(dev, 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 {
        // loop through all of the network VRT packets until an IF data packet is received
        // read the data
        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 -1;
        }
```

```
    // handle context packets
    if (has_gnss_data) {
        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 the SPP
sprintf(cmd_str, "TRACE:SPP %d", SPP);
result = wsaSetSCPI(wsaHandle, cmd_str);

// set the PPB
sprintf(cmd_str, "TRACE:BLOCK:PACKETS %d", PPB);
result = wsaSetSCPI(wsaHandle, cmd_str);

// start a new LOOPS time
time(&run_start_time);
do {
    //*********
    // Initialize buffers and start VRT data capture
    //*********
    i_data = (int16_t *)malloc(sizeof(int16_t) * SPP * PPB);
    if (i_data == NULL) {
        printf("Failed to allocate memory for the block.\n");
        return -1;
    }

    q_data = (int16_t *)malloc(sizeof(int16_t) * SPP * PPB);
    if (q_data == NULL) {
        printf("Failed to allocate memory for the block.\n");
        free(i_data);
        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;
    }

    // initiate the block data capture
    result = wsaSetSCPI(wsaHandle, ":TRACE:BLOCK:DATA?");
    if (result < 0) {
        free(i_data);
        free(q_data);

        wsaErrorMessage_s(result, err_msg);
        printf("Error msg: %s\n", err_msg);
        return -1;
    }

    // Loop to get all the requested packets SPP per packet & stitch together
    for (k = 0; k < PPB; k++) {
        // loop through all of the network VRT packets until all IF data packets are received
        while (1) {
            result = wsaReadVRTData(wsaHandle, SPP, TIMEOUT, &header, i16_data, q16_data,
```

```
i32_data,
            &spectral_inversion, &vrt_context_pkt);
        if ((result < 0) && (result != WSA_WARNING_SOCKETTIMEOUT)) {
            printf("Failed to read data!\n");
            free(i_data);
            free(q_data);

            wsaErrorMessage_s(result, err_msg);
            printf("Error msg: %s\n", err_msg);
            return -1;
        }

        // handle IF packet types
        if (header.packet_type == IF_PACKET_TYPE) {
            // append each new SPP data to the collective array
            memcpy(i_data + (k * SPP), i16_data, SPP * sizeof(int16_t));
            memcpy(q_data + (k * SPP), q16_data, SPP * sizeof(int16_t));

            // print only once the same info of all the IF packets:
            if (i == 0 && k == 0) {
                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);
            }

            break;
        }
        // handle context packets
        else {
            if ((header.stream_id == DIGITIZER_STREAM_ID) &&
                ((vrt_context_pkt.indicator_field & GNSS_INDICATOR_MASK) == GNSS_INDICATOR_MASK)) {
                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++;
            }
            continue;
        }
    } // end save data while loop
}


// do some FFT & PSD processing on each set of data
// Note that code could be written to combine the whole block before
// calling FFT
result = wsaGetFFTSize(SPP * PPB, header.stream_id, &fft_size);
spectral_data = (float *)malloc(sizeof(float) * fft_size);
if (spectral_data == NULL) {
    free(i_data);
    free(q_data);

    wsaErrorMessage_s(result, err_msg);
    printf("Error msg: %s\n", err_msg);
    return -1;
}

result = wsaComputeFFT(SPP * PPB,
    fft_size,
    header.stream_id,
    vrt_context_pkt.reference_level,
    spectral_inversion,
    i_data,
    q_data,
    i32_data,  // ignore in this example
    spectral_data);
if (result < 0) {
    printf("Failed to compute FFT!\n");
    free(i_data);
```

```
            free(q_data);
            free(spectral_data);

            wsaErrorMessage_s(result, err_msg);
            printf("Error msg: %s\n", err_msg);
            return -1;
        }

        // An example signal processing applied to the FFT data
        result = wsaPSDPeakFind(wsaHandle, RFE_MODE, spectral_inversion,
            fft_size, spectral_data, &peak_freq, &peak_power);

        // Remove the frequency shift value to get the exact input freq
        printf("%d Peak freq & Power: %f MHz, %f dBm\n", i, (float)(peak_freq - 2 * vrt_context_pkt.
      freq_offset) / MHZ, peak_power);

        // Free up the current block mem allocation
        free(i_data);
        free(q_data);
        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);
    if (((total_pkt - run_time) < 0) || ((total_pkt - run_time) > 1))

    printf("All capture done.\n");

    return 0;
}
```

## 7.4 simpleHIF.cpp

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

```
#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.

```
#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 samples_per_packet = 1024;
    char * rfe_mode = "SH";
    const float input_freq = 100; // in MHz
    int attenuation = 0;
    int16_t i16_data[samples_per_packet];
    int16_t q16_data[samples_per_packet];
    int32_t i32_data[samples_per_packet];
    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;
    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 -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);

// 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", samples_per_packet);
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,
                     samples_per_packet,
                     &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 -1;
}

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\n", spectral_inversion);

// determine FFT size required
result = wsaGetFFTSize(samples_per_packet, 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;
}

// compute the FFT & PSD, with data inverted when needed
result = wsaComputeFFT(samples_per_packet,
                    fft_size,
                    (int32_t) stream_id,
                    reference_level,
                    spectral_inversion,
                    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 -1;
}

//*********
// 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 -1;
}
// 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 -1;
}
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 -1;
}
// 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 -1;
}
// 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 -1;
}
// 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 -1;
}
// 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 -1;
    }

    // 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.

```
#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 samples_per_packet = 1024;
    char * rfe_mode = "SH";
    const float input_freq = 2450.0; // in MHz
    int16_t i16_data[samples_per_packet];
    int16_t q16_data[samples_per_packet];
    int32_t i32_data[samples_per_packet];
    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;
    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 -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);

// 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 -1;
}

// 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 -1;
}

// set the frequency
sprintf(cmd_str, "FREQ:CENT %f MHZ", input_freq);
result = wsaSetSCPI(wsaHandle, cmd_str);

// set the SPP
sprintf(cmd_str, "TRACE:SPP %d", samples_per_packet);
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,
                     samples_per_packet,
                     &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 -1;
}

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,
    spectral_inversion, &span_bw, &span_center, &span_fstart, &span_fstop);
if (result < 0) {
    wsaErrorMessage_s(result, err_msg);
    printf("Error msg: %s\n", err_msg);
    return -1;
}
```

```
    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(samples_per_packet, 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;
    }

    // compute the FFT & PSD
    result = wsaComputeFFT(samples_per_packet,
                        fft_size,
                        (int32_t) stream_id,
                        reference_level,
                        spectral_inversion,
                        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 -1;
    }

    // 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 -1;
    }

    // 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.

```
#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 samples_per_packet = 1024;
    int16_t i16_data[samples_per_packet];
    int16_t q16_data[samples_per_packet];
    int32_t i32_data[samples_per_packet];
    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 -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);

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,
                             samples_per_packet,
                             &timestamp_sec,
                             &timestamp_psec,
                             &reference_level,
                             &bandwidth,
                             &center_frequency);
if (result < 0) {
    printf(wsaErrorMessage(result));
    printf("\n");
    return -1;
}

//determine FFT size required
```

```
    result = wsaGetFFTSize(samples_per_packet, 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
    result = wsaComputeFFT(samples_per_packet,
                        fft_size,
                        (int32_t) stream_id,
                        reference_level,
                        spectral_inversion,
                        i16_data,
                        q16_data,
                        i32_data,
                        spectral_data);
    if (result < 0) {
        printf(wsaErrorMessage(result));
        printf("\n");
        free(spectral_data);
        return -1;
    }

    // disconnect from device
    result = wsaDisconnect(wsaHandle);
    if (result < 0) {
        printf(wsaErrorMessage(result));
        printf("\n");
        free(spectral_data);
        return -1;
    }

    // print out values
    for (int i = 0; i < (fft_size); i++)
        printf("%d, %0.2f \n", i, spectral_data[i]);

    // free FFT buffer
    free(spectral_data);
    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.

```
#include <tchar.h>
#include <conio.h>
#include <stdio.h>
#include <stdlib.h>
#include "wsaInterface.h"

int _tmain()
{
    // The device instance
    int64_t wsaHandle;
    char ip[50];

    // variables required for the read
    const int32_t samples_per_packet = 1024;
    char * rfe_mode = "SH";
    const float input_freq = 2450.0; // in MHz
    int decimation = 16;
    int attenuation = 10;
    int16_t i16_data[samples_per_packet];
    int16_t q16_data[samples_per_packet];
    int32_t i32_data[samples_per_packet];
    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;
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 -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");
if (result < 0) {
    wsaErrorMessage_s(result, err_msg);
    printf("Error msg: %s\n", err_msg);
    return -1;
}

// 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 -1;
}

// 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 -1;
}

// 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 -1;
}

// set the SPP
sprintf(cmd_str, "TRACE:SPP %d", samples_per_packet);
```

```
result = wsaSetSCPI(wsaHandle, cmd_str);
if (result < 0) {
    wsaErrorMessage_s(result, err_msg);
    printf("Error msg: %s\n", err_msg);
    return -1;
}

// 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 -1;
}

// set the attenuation
sprintf(cmd_str, "INPUT:ATT %d", attenuation);
result = wsaSetSCPI(wsaHandle, cmd_str);
if (result < 0) {
    wsaErrorMessage_s(result, err_msg);
    printf("Error msg: %s\n", err_msg);
    return -1;
}

// capture the data
result = wsaSetSCPI(wsaHandle, ":TRACE:STREAM:START");
if (result < 0) {
    wsaErrorMessage_s(result, err_msg);
    printf("Error msg: %s\n", err_msg);
    return -1;
}

// read the data
try
{
    for (int i = 0; i < 4 && !_kbhit(); i++) {
        memset(i16_data, 0, samples_per_packet * sizeof(int16_t));
        memset(q16_data, 0, samples_per_packet * sizeof(int16_t));
        memset(i32_data, 0, samples_per_packet * sizeof(int32_t));

        result = wsaReadData(wsaHandle,
            i16_data,
            q16_data,
            i32_data,
            timeout,
            &stream_id,
            &spectral_inversion,
            samples_per_packet,
            &timestamp_sec,
            &timestamp_psec,
            &reference_level,
            &bandwidth,
            &center_frequency);
        if (result < 0) {
            wsaErrorMessage_s(result, err_msg);
            printf("Error msg: %s\n", err_msg);
            break;
        }

        printf("[%d] stream_id: %08x\n", i, stream_id);
        printf("[%d] spectral_inversion: %d\n", i, spectral_inversion);
        printf("[%d] samples_per_packet: %d\n", i, samples_per_packet);
        printf("[%d] timestamp_sec: %d\n", i, timestamp_sec);
        printf("[%d] timestamp_psec: %lld\n", i, timestamp_psec);
        printf("[%d] reference_level: %d\n", i, reference_level);
        printf("[%d] bandwidth: %lld\n", i, bandwidth);
        printf("[%d] center_frequency: %lld\n", i, center_frequency);
        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");

        // determine FFT size required
        result = wsaGetFFTSize(samples_per_packet, stream_id, &fft_size);

        // allocate data for the spectral data variable
        spectral_data = (float*)malloc(sizeof(float) * fft_size);
        printf("[%d] fft_size: %d\n", i, fft_size);
```

```
            // compute the FFT
            result = wsaComputeFFT(samples_per_packet,
                fft_size,
                stream_id,
                reference_level,
                spectral_inversion,
                i16_data,
                q16_data,
                i32_data,
                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);
        }
    }
    catch (...)
    {
        printf("error");
        return -1;
    }

    result = wsaSetSCPI(wsaHandle, ":TRACE:STREAM:STOP");
    if (result < 0) {
        wsaErrorMessage_s(result, err_msg);
        printf("Error msg: %s\n", err_msg);
        return -1;
    }

    // disconnect from device
    result = wsaDisconnect(wsaHandle);
    if (result < 0) {
        wsaErrorMessage_s(result, err_msg);
        printf("Error msg: %s\n", err_msg);
        return -1;
    }

    return 0;
}
```

## 7.9 sweepDeviceCalculateChannelPower.cpp

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

```
#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

```
#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

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

```
#include <tchar.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 = 88000000;
    uint64_t fstop = 900000000;
    uint32_t rbw = 25000;
    char mode[20] = "SH";
    int32_t attenuator = 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 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
    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);

    // Some system clean up before starting
    result = wsaSetSCPI(wsaHandle, "*RST");
    result = wsaSetSCPI(wsaHandle, ":SYSTEM:ABORT");
    result = wsaSetSCPI(wsaHandle, ":SYSTEM:FLUSH");
```

```
    // request for data acquisition rights
    scpi_result = wsaGetSCPI(wsaHandle, ":SYSTEM:LOCK:REQUEST? ACQ");

    // 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: %s\n", err_msg);

        return -1;
    }
    printf("Sweep size: %d\nActual sweep frequencies: %lf MHz - %lf MHz\n\n", sweep_size,
        (float)fstart_actual / MHZ, (float)fstop_actual / MHZ);

    // 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");
    // Capture spectral data
    result = wsaCaptureSpectrum(wsaHandle,
        fstart,
        fstop,
        rbw,
        mode,
        attenuator,
        spectral_data);
    if (result < 0) {
        wsaErrorMessage_s(result, err_msg);
        printf("Error msg: %s\n", err_msg);
        free(spectral_data);
        return -1;
    }

    // Print out the spectral data
    for (uint32_t i = 0; i < sweep_size; i++)
        printf("%0.2f, ", spectral_data[i]);
    printf("\n");

    // 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.

```
#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 = 1900000000;
uint64_t fstop = 2450000000;
uint32_t rbw = 1000;
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