R&S® NRP VISA Passport

User Manual

# Table of Contents

1 Driver Architecture ........................................ 3

2 Installation ........................................ 4

2.1 Message Log ........................................ 4

3 Using the NRP VISA Passport ........................................ 6

3.1 Resource Descriptor ........................................ 6

3.2 VISA Functions ........................................ 10

3.3 Return Data Format ........................................ 12

3.4 Specific SCPI Commands ........................................ 13

4 Examples ........................................ 15

4.1 Checking for Errors ........................................ 15

4.2 Wait for Completion ........................................ 16

4.3 Zeroing Sensor ........................................ 17

4.4 Continuous Average ........................................ 18

4.5 Reading Binary Data ........................................ 20

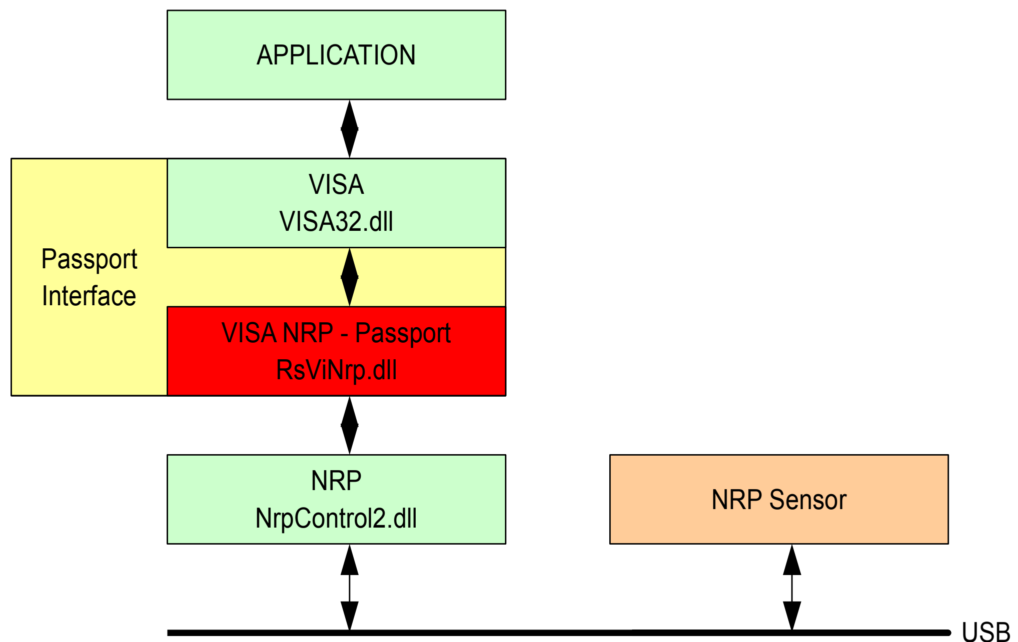
4.6 Measuring Trace ........................................ 22

4.7 Measuring CCDF with Average Power ........................................ 23

4.8 Further information ........................................ 25

# 1 Driver Architecture

The diagram below shows how the R&S®NRP VISA Passport fits into the overall VISA architecture. The application generally uses the VISA functions to access an instrument. When an instrument is opened a unique resource identifier is used to define the physical instrument link, for example, GPIB, LAN VXI-11, or USB. The R&S® NRP series power sensors use a proprietary binary protocol for the data transfer between the host PC and the sensor. Therefore a driver is required on the PC side to translate SCPI commands into the binary data required by the sensor. The so called 'VISA Passport' is the defined extension interface to install links between VISA and a custom drivers and provides a way to route SCPI commands to these drivers.




Note: Although the above picture illustrates only 32-bit modules the R&S®NRP VISA Passport driver is available in both 32-bit and 64-bit versions.

# 2 Installation

Prior to installing the R&S®NRP VISA Passport driver the following other software packages must be installed:

1) Install the R&S® NRP-Toolkit. This package provides the USB drivers for the NRP series power meters as well as the low level power sensor drivers.

2) Install the National Instruments VISA runtime. Currently only this VISA version is supported.

Once the above prerequisites are met the R&S® NRP VISA Passport can be installed. The setup is provided as a standard installation package which will guide you through the installation process. The name of the installation package looks like **`RsViNrp_x_y_z.exe`** where **`x`**, **`y`** and **`z`** identify the major-, minor- and sub-version of the current release (for example, at the time of this document being written, the current installation package is called **`RsViNrp_2_7_3.exe`**). Depending on the architecture of your system, either only the 32-bit version of R&S® NRP VISA Passport (**`RsViNrp.dll`**) or both 32-bit and 64-bit versions are installed (the 32-bit **`RsViNrp.dll`** under the **`%VXIPNPPATH%`** folder and the 64-bit **`RsViNrp.dll`** version under the **`%VXIPNPPATH64%`** folder).

If the R&S® NRP VISA Passport is updated from an earlier version the old version is removed first. In this case a reboot may be required after the installation is complete.

## 2.1 Message Log

The R&S® NRP VISA Passport component supports a logging mechanism for debugging purposes. The log level is set by changing a registry key as mentioned below. Changing can be made by following the steps listed below:

1) Start the registry editor by executing `regedit` on the command line.

2) Navigate to the following location:

```
HKEY_LOCAL_MACHINE / SOFTWARE / National Instruments /
NI-VISA / CurrentVersion / IoLibraries / RsViNrp.dll
```

3) Modifies the following entries as needed:

```
LOG_FILE        C:\RsViNrp.log
LOG_LEVEL       NONE
```

4) Close the registry editor tool.

The following log levels are supported:

| | |
|---|---|
| DEBUG | All messages are logged |
| INFO | Human readable messages, warning, and errors are logged |
| WARN | Warnings and errors are logged |
| ERROR | Only error messages are logged |
| FATAL | Only fatal errors are logged |
| NONE | Logging is off (default) |

# 3 Using the NRP VISA Passport

## 3.1 Resource Descriptor

In order to operate a device under VISA a so called VISA sessions must be opened. A VISA session is generally opened using the `viOpen` function. This function requires a resource descriptor string that clearly identifies the hardware interface as well as the instrument used with this VISA session. The format of the resource descriptor for the R&S® NRP series power sensors is as follows:

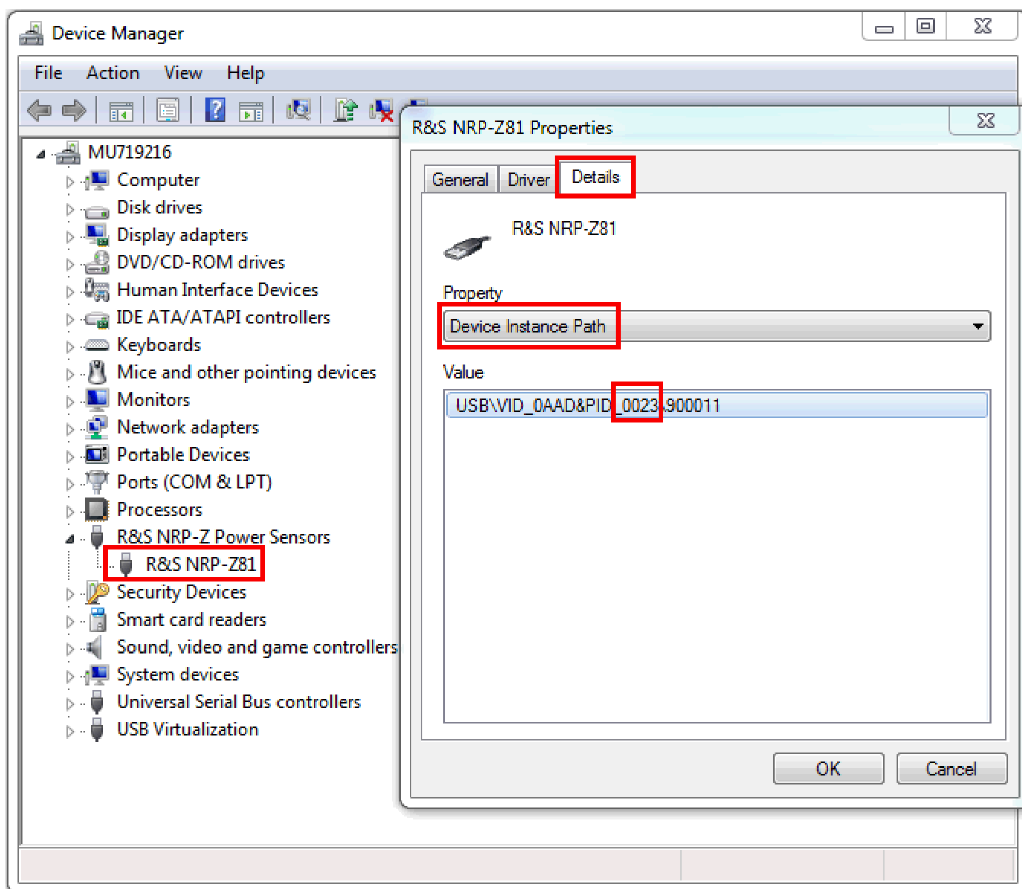
```
"RSNRP::<id>::<serial>[::INSTR]"
<id>        : USB device ID in hex. format
<serial>    : the sensor's serial number, e.g. 100123
```

Please note that the interface type is **RSNRP** and not **USB**. Using the **RSNRP** interface type instructs the VISA layer to route all further communication through the R&S® NRP VISA Passport driver to the corresponding power sensor(s).

**Example:**
A typical resource descriptor for an R&S® NRP-Z81 power sensor (which is characterized by a USB device ID of 0x0023) with serial number 124066 would look like

```
RSNRP::0x0023::124066::INSTR
```

The USB device ID of a certain R&S® power sensor type can be determined either by a lookup in the table below, or by using the Windows Device Manager and displaying the properties shown in the following picture.




| R&S Power Sensor | USB Device ID |
|---|---|
| FSH-Z1 | 0x000b |
| FSH-Z18 | 0x001a |
| NRP-Z11 | 0x000c |
| NRP-Z21 | 0x0003 |
| NRP-Z22 | 0x0013 |
| NRP-Z23 | 0x0014 |
| NRP-Z24 | 0x0015 |
| NRP-Z27 | 0x002f |
| NRP-Z28 | 0x0051 |
| NRP-Z31 | 0x002c |
| NRP-Z32 | 0x009a |

| R&S Power Sensor | USB Device ID |
|---|---|
| NRP-Z37 | 0x002d |
| NRP-Z41 | 0x0096 |
| NRP-Z51 | 0x0016 |
| NRP-Z52 | 0x0017 |
| NRP-Z55 | 0x0018 |
| NRP-Z56 | 0x0019 |
| NRP-Z57 | 0x0070 |
| NRP-Z58 | 0x00a8 |
| NRP-Z61 | 0x0097 |
| NRP-Z71 | 0x0098 |
| NRP-Z81 | 0x0023 |
| NRP-Z85 | 0x0083 |
| NRP-Z86 | 0x0095 |
| NRP-Z91 | 0x0021 |
| NRP-Z96 | 0x002e |
| NRP-Z98 | 0x0052 |
| NRP-Z92 | 0x0062 |
| NRP-Z211 | 0x00a6 |
| NRP-Z221 | 0x00a7 |
| NRPC18 | 0x00bf |
| NRPC33 | 0x00b6 |
| NRPC40 | 0x008f |
| NRPC50 | 0x0090 |
| NRPC18-B1 | 0x00c1 |
| NRPC33-B1 | 0x00c2 |
| NRPC40-B1 | 0x00c3 |

| R&S Power Sensor | USB Device ID |
|---|---|
| NRPC50-B1 | 0x00c4 |
| NRP8S | 0x00e2 |
| NRP8SN | 0x0137 |
| NRP18S | 0x0138 |
| NRP18SN | 0x0139 |
| NRP33S | 0x0145 |
| NRP33SN | 0x0146 |
| NRP33SN-V | 0x0168 |
| NRP40S | 0x015f |
| NRP40SN | 0x0160 |
| NRP50S | 0x0161 |
| NRP50SN | 0x0162 |
| NRP18S-10 | 0x0148 |
| NRP18SN-10 | 0x0149 |
| NRP18S-20 | 0x014a |
| NRP18SN-20 | 0x014b |
| NRP18S-25 | 0x014c |
| NRP18SN-25 | 0x014d |
| NRP6A | 0x0178 |
| NRP6AN | 0x0179 |
| NRP18A | 0x014e |
| NRP18AN | 0x014f |
| NRP18T | 0x0150 |
| NRP18TN | 0x0151 |
| NRP33T | 0x0152 |
| NRP33TN | 0x0153 |

| R&S Power Sensor | USB Device ID |
|---|---|
| NRP40T | 0x0154 |
| NRP40TN | 0x0155 |
| NRP50T | 0x0156 |
| NRP50TN | 0x0157 |
| NRP67T | 0x0158 |
| NRP67TN | 0x0159 |
| NRP110T | 0x015a |
| NRPM3 | 0x0195 |

## 3.2 VISA Functions

The R&S® NRP VISA Passport supports the following VISA functions and attributes:

- **viGetAttribute**

  VI_ATTR_RSRC_SPEC_VERSION

  VI_ATTR_INTF_INST_NAME

  VI_ATTR_MANF_ID

  VI_ATTR_MANF_NAME

  VI_ATTR_RSRC_NAME

  VI_ATTR_RSRC_MANF_NAME

  VI_ATTR_RSRC_MANF_ID

  VI_ATTR_RSRC_IMPL_VERSION

  VI_ATTR_RSRC_CLASS

  VI_ATTR_TMO_VALUE (in ms)

  VI_ATTR_USB_INTFC_NUM

  VI_ATTR_USB_PROTOCOL

  VI_ATTR_USB_SERIAL_NUM

- **viSetAttribute**

  VI_ATTR_TMO_VALUE (in ms)

- **viEnableEvent**

  VI_EVENT_SERVICE_REQ

The event will be generated if the status of the NRP series power sensor changes.

VI_EVENT_USB_INTR

This event will be generated if a power sensor gets connected or disconnected.

VI_EVENT_EXCEPTION

This event will be generated if an error occurs.

- **viDisableEvent**
- **viWrite**
- **viRead**
- **viAssertTrigger**
- **viReadSTB**
- **viClear**
- **viOpen**
- **viClose**
- **viParseRsrc**
- **viFindRsrc**

## 3.3 Return Data Format

The `viRead` function as well as the `FETCH?` query are both used to read data from the ower sensor. Typically this data was requested by a previous SCPI command and the result was stored in the driver cache. The format of the data returned by the two methods therefore depends on this SCPI query. The following data types are currently supported:

- Binary result (unsigned char)
- Integer value in ASCII format.
- Floating point number in exponential ASCII format.
- Multiple, comma separated floating point values in ASCII format.
- Fixed length binary data block according to SCPI standard:

[LC]: Number of characters in CO (length of length)

[CO]: Data block length in bytes

[B1]: 1st byte

[B2]: 2nd byte

[Bn]: nth byte

## 3.4 Specific SCPI Commands

The protocol between the low level NRP power sensor drivers and the NRP series power sensors is a proprietary binary format. As a result, the R&S® NRP VISA passport forwards SCPI commands to the low level drivers which translates them into binary information that the sensor can process. In turn, all results received from the sensor are cached by the driver and forwarded to the VISA layer as needed.

Some specific SCPI commands do not have a direct functional representation inside of the power sensor. Instead, this functionality is mimicked by the low level NRP power sensor driver layer:

**SYSTem:ERRor?**

This query returns the next error message from the driver's error queue.

**STATus:OPERation:CONDition?**

This query returns the status byte from the driver's status cache. The following table lists which bits reflect a certain sensor state:

| Bit | Function |
|---|---|
| 0 | 0 |
| 1 | 0 |
| 2 | 0 |
| 3 | 0 |
| 4 | sensor state is MEASURING |
| 5 | sensor state is WAIT_FOR_TRIGGER |
| 6 | 0 |
| 7 | 0 |

**FORMat ASCii**

This command sets the format of all returned numeric data to ASCII.

**FORMat REAL,32**

This command sets the format of all returned numeric data to floating point, 32-bit.

**SENSe:AVERage:COUNt?**

Note: Due to restrictions of the underlying, asynchronous communication protocol of the traditional NRP power sensors, the command to query the average count is disabled in the NRP VISA Passport driver. This limitation was necessary, because a sensor can send a change in its average count at any time, asynchronously during an active measurement, in case the auto-averaging feature was activated in the sensor. Such a signalization of an average count change could tamper with an available measurement result.

**FETCh?**

The query returns values from the driver's internal data cache. The format depends on the SCPI commands that were send before.

# 4 Examples

This section provides C-code examples that may be used as a starting point for own software projects. The code examples use the standard VISA functions, e.g. `viPrintf` or `viScanf` to exchange data with the power sensor.

## 4.1 Checking for Errors

The function below repeatedly polls the driver's error queue. The return code is `false` if errors occurred or `true` if the queue does not contain any errors.

```
bool checkErrors( ViSession SensorSes )
{
	bool bOK = true;

	static const int BUFFER_SIZE = 4096;
	char buf[BUFFER_SIZE] = {0};

	memset( buf, 0, sizeof(buf) );

	do
	{
		if( viPrintf( SensorSes, "SYST:ERR?\n" )!=0 )
			return false;

		if( viScanf( SensorSes, "%t", &buf )!=0 )
			return false;

		if( buf[0]=='0' )
			break;

		bOK = false;
		printf( "Error: '%s'\n", buf );

	} while( true );

	return bOK;
}
```

## 4.2 Wait for Completion

The function below polls the driver’s cache for the measurement completion state. The function times out after about 2 seconds.

```
bool waitForCompletion( ViSession SensorSes )
{
	int Status;

	static const int BUFFER_SIZE = 4096;
	char buf[BUFFER_SIZE] = {0};

	printf( "\n" );

	for( int i=0; i<20; i++ )
	{
		// sleep for 100 milliseconds
		Sleep( 100 );

		memset( buf, 0, sizeof(buf) );

		if( viPrintf( SensorSes, "STAT:OPER:COND?\n" )!=0 )
			return false;

		if( viScanf( SensorSes, "%t", &buf )!=0 )
			return false;

		Status = atoi( buf );

		printf( "." );

		if( Status==0 )
			break;
	}

	printf( "\n" );

	if( Status!=0 )
		return false;

	return true;
}
```

## 4.3 Zeroing Sensor

This example shows how to execute zero compensation on a power sensor. The command is

```
CAL:ZERO:AUTO ONCE
```

Depending on the type of sensor zero compensation may take 4 s to 8 s (or sometimes even longer). After executing the sensor's zero compensation an error check should follow.

The following snippet shows an implementation in C/C++ code.

```
bool zeroSensor( ViSession SensorSes )
{
	// This may take some sesonds...
	viPrintf( SensorSes, "CAL:ZERO:AUTO ONCE\n" );

	if ( !checkErrors( SensorSes ) )
	{
		printf( "Zeroing failed\n" );
		return false;
	}

	printf( "Zeroing successful\n" );
	return true;
}
```

## 4.4 Continuous Average

The example below measures the average power of a signal. The sampling window is set to 5 ms. Averaging is turned on with an average count of 16.

```
INIT:CONT OFF
SENS:FUNC "POW:AVG"
SENS:FREQ 1e9
SENS:AVER:COUN:AUTO OFF
SENS:AVER:COUN 16
SENS:AVER:STAT ON
SENS:AVER:TCON REP
SENS:POW:AVG:APER 5e-3

FORMAT ASCII
SYST:ERR?
0,"No Error"
INIT:IMM
STAT:OPER:COND?
0
FETCH?
1.001674e-002,0.000000e+000,0.000000e+000
```

The first set of commands configures the measurement mode as well as basic parameters. The complete list of commands varies between sensors and can be found in the sensor's operating manual.

When the sensor is used in manual filter mode it is required to turn the AUTO filter off explicitly. Otherwise the automatic filter takes precedence over the manual setting.

The FORMAT command sets the return data format to ASCII. After the configuration is complete the sensor's error queue should be checked for pending errors. This is done by calling the SYST:ERR? query until the error number is zero.

The following command INIT:IMM starts the measurement. The command returns immediately and therefore it is required to poll the measurement completion state before fetching the result. Polling the sensor state is done with the query STAT:OPER:COND?. A zero return value indicates that the result is available.

The following FETCH? query returns the measured power value. Since some R&S power sensor models can be configured to also return auxiliary values (min/max or random/max), the string returned by FETCH? contains three result values. In the example above no auxiliary values have been configured, therefore only the first value is used and contains the average power in Watts. (If you want to learn more about auxiliary values, search your sensor's operating manual for command SENS:AUX. However, bear in mind that not all sensors implement the SENS:AUX command. In such cases the second and third value would always be 0.0)

The following C-code example demonstrates how this measurement can be implemented.

```
bool contav( ViSession SensorSes )
{
      viPrintf( SensorSes, "INIT:CONT OFF\n"             );
      viPrintf( SensorSes, "SENS:FUNC \"POW:AVG\"\n"     );
      viPrintf( SensorSes, "SENS:FREQ 1e9\n"             );
      viPrintf( SensorSes, "SENS:AVER:COUN:AUTO OFF\n" );
      viPrintf( SensorSes, "SENS:AVER:COUN 16\n"         );
      viPrintf( SensorSes, "SENS:AVER:STAT ON\n"         );
      viPrintf( SensorSes, "SENS:AVER:TCON REP\n"        );
      viPrintf( SensorSes, "SENS:POW:AVG:APER 5e-3\n"   );
      viPrintf( SensorSes, "FORMAT ASCII\n"              );

      if( !checkErrors( SensorSes ) )
            return false;

      viPrintf( SensorSes, "INIT:IMM\n" );

      if( !waitForCompletion( SensorSes ) )
            return false;

      if( viPrintf( SensorSes, "FETCH?\n" ) != 0 )
            return false;

      char cData[256];
      memset( cData, 0, 256 );
      ViUInt32 iRetCnt;
      if( viBufRead( SensorSes, (ViPBuf)cData,
                     256, &iRetCnt ) != 0 )
            return false;

      // three comma seperated values
      printf( "Raw Result = '%s'\n", cData );

      double dResult = atof( cData );

      printf( "Pav = %10.2f dBm\n",
          30.0 + 10.0 * log10( fabs( dResult ) + 1e-32 ) );

      return true;
}
```

## 4.5 Reading Binary Data

The following function reads a block of binary data in Float32 format from the sensor. This function can therefore be used with trace or CCDF measurements.

```
bool readArray( ViSession SensorSes, float *pData, int iPoints )
{
      ViUInt32 iRetCnt = 0;

      if( pData==0 )
            return false;

      if( viPrintf( SensorSes, "FETCH?\n" ) != 0 )
            return false;

      int iBufLen = iPoints * 4 + 32;
      char *cTmpBuf = (char*)malloc( iBufLen );
      if( cTmpBuf==0 )
            return false;

      viSetAttribute( SensorSes, VI_ATTR_TERMCHAR_EN, false );
      if( viRead( SensorSes, (ViPBuf)cTmpBuf, iBufLen,
                  &iRetCnt ) != 0 )
      {
            delete cTmpBuf;
            viSetAttribute( SensorSes, VI_ATTR_TERMCHAR_EN,
                            true );
            return false;
      }
      viSetAttribute( SensorSes, VI_ATTR_TERMCHAR_EN, true );

      if( cTmpBuf[0]!='#' )
      {
            delete cTmpBuf;
            printf( "Unexpected prefix.\n" );
            return false;
      }

      int iLenLen = cTmpBuf[1] - '0';
      if( iLenLen<1 || iLenLen>9 )
      {
            delete cTmpBuf;
            printf( "Invalid length of length.\n" );
            return false;
      }

      char cLenBuf[10];
      memcpy( cLenBuf, &cTmpBuf[2], iLenLen );
      cLenBuf[iLenLen] = 0;
      int iDataLen = atoi( cLenBuf );
      if( iDataLen<=0 || iDataLen>iPoints*4 || (iDataLen%4)!=0 )
      {
```

```
            delete cTmpBuf;
            printf( "Invalid data length.\n" );
            return false;
        }

        memcpy( pData, cTmpBuf + 2 + iLenLen, iPoints*4 );
        delete cTmpBuf;

        if( (int)iRetCnt < iPoints*4+2+iLenLen )
        {
            printf( "Insufficient number of points.\n" );
            return false;
        }

        return true;
}
```

## 4.6 Measuring Trace

The code example below demonstrates how a simple trace measurement can be configured and how the measured data is read back to the host PC.

```
bool trace( ViSession SensorSes )
{
      viPrintf( SensorSes, "INIT:CONT OFF\n" );
      viPrintf( SensorSes, "SENS:FUNC \"XTIM:POW\"\n" );
      viPrintf( SensorSes, "SENS:FREQ 1e9\n" );
      viPrintf( SensorSes, "SENS:TRAC:POIN 200\n" );
      viPrintf( SensorSes, "SENS:TRAC:TIME 5e-3\n" );
      viPrintf( SensorSes, "SENS:TRAC:OFFS:TIME -500e-6\n" );
      viPrintf( SensorSes, "TRIG:SLOP POS\n" );
      viPrintf( SensorSes, "TRIG:LEV 1e-3\n" );
      viPrintf( SensorSes, "TRIG:SOUR INT\n" );
      viPrintf( SensorSes, "TRIG:DTIM 0\n" );
      viPrintf( SensorSes, "SENS:AVER:COUN 16\n" );
      viPrintf( SensorSes, "SENS:AVER:STAT ON\n" );
      viPrintf( SensorSes, "SENS:AVER:TCON REP\n" );
      viPrintf( SensorSes, "FORMAT REAL,32\n" );
      if( !checkErrors( SensorSes ) )
            return false;

      viPrintf( SensorSes, "INIT:IMM\n" );

      if( !waitForCompletion( SensorSes ) )
            return false;

      float FloatData[200];
      if( !readArray( SensorSes, FloatData, 200 ) )
            return false;

      // simple conversion to dBm
      double OutData[200];
      for( int i=0; i<200; i++ )
            OutData[i] = 30.0+10.0*log10(
                             fabs( FloatData[i] ) + 1e-32 );

      // … plot data …

      return true;
}
```

## 4.7 Measuring CCDF with Average Power

This code example demonstrates a CCDF measurement. This type of measurement is possible with the R&S NRP-Z8x series (NRP-Z81/Z85/Z86) of power sensors. Besides the power distribution values, this measurement can also return the average power within the measurement interval.

In this context you should be aware of a special behavior of the R&S NRP VISA Passport driver: Normally the measurement data can be read as an array of values. The resulting array is returned as an ASCII string of comma separated values, containing exactly as much points as have been selected for the measurement range. In order to also get the average power, you **must** select the **binary** result format. In that case you can read one datapoint more than have been selected for the measurement. The additional value contains the average power in W.

In the following example a CCDF measurement in the range of -25 dBm … +25 dBm is configured (50 dB range), distributed to 200 resulting points (= measurement values), and it is shown how the 201st value is read as the average power.

```
const ViUInt16 ErrQBit = 1<<2;
const ViUInt16 MAVBit  = 1<<4;
const ViUInt16 OperBit = 1<<7;

bool waitForMAV( ViSession sess )
{
      ViStatus stat;
      ViUInt16 uiSTB;
      static const int BUFFER_SIZE = 4096;
      char buf[BUFFER_SIZE] = {0};

      for( int i = 0; i < 500; i++ )
      {
            // sleep for 10 milliseconds
            Sleep( 10 );

            stat = viReadSTB( sess, &uiSTB );
            if ( (uiSTB & (MAVBit | OperBit)) == MAVBit )
                  break;
      }

      return ((uiSTB & MAVBit) == MAVBit);
}


bool CCDF( ViSession sess )
{
      viPrintf( sess, "*RST\n" );
      viPrintf( sess, "SENS:FUNC \"XPOW:CCDF\"\n" );
      viPrintf( sess, "SENS:FREQ 1e9\n" );
      viPrintf( sess, "SENS:BWID:VID \"FULL\"\n" );
      viPrintf( sess, "SENS:STAT:TIME 0.01\n" );
      viPrintf( sess, "SENS:STAT:OFFS:TIME 0\n" );
```

```
        viPrintf( sess, "SENS:STAT:EXCL:MID:TIME 0\n" );
        viPrintf( sess, "SENS:STAT:EXCL:MID:OFFS:TIME 0\n" );
        viPrintf( sess, "SENS:TRAC:AVER:STAT ON\n" );
        viPrintf( sess, "SENS:TRAC:AVER:TCON REP\n" );
        viPrintf( sess, "SENS:TRAC:AVER:COUN 128\n" );
        viPrintf( sess, "SENS:STAT:SCAL:X:RLEV -25\n" );
        viPrintf( sess, "SENS:STAT:SCAL:X:RANG 50\n" );
        viPrintf( sess, "SENS:STAT:SCAL:X:POIN 200\n" );
        viPrintf( sess, "TRIG:SOUR IMM\n" );

        // It is essential to select BINARY result format
        viPrintf( sess, "FORMAT REAL,32\n" );

        if( !checkErrors( sess ) )
              return false;

        // Starting a measurement...
        viPrintf( sess, "INIT:IMM\n" );

        // ...and waiting for the result
        if( !waitForMAV( sess ) )
              return false;

        // Reading the results. Be aware of the N+1 values!
        float FloatData[201];
        if( !readArray( sess, FloatData, 201 ) )
              return false;

        // FloatData[0]...FloatData[199] = meas data
        // FloatData[200]                = average power
        printf( "CCDF: Average Power = %.3f uW\n",
                1.0e6 * FloatData[200] );

        // … plot data …

        return true;
}
```

## 4.8 Further information

As mentioned earlier, the basic driver package for R&S® USB power sensors –the so called R&S® NRP-Toolkit– needs to be installed before you are able to communicate with the device(s). During the R&S® NRP-Toolkit installation you will have an option of installing the NRP-Toolkit software development kit (SDK). When selecting this option, the package (also) installs some VISA Passport example programs in C/C++ and C# source-code, which show how to construct your own applications utilizing NI-VISA and R&S® NRP VISA Passport. --- After the installation of the R&S® NRP-Toolkit-SDK see

`C:\ProgramData\Rohde-Schwarz\NRP-Toolkit-SDK\examples\Visa`

for further references.