Table of Contents

5 Remote Control – Fundamentals 5.1

USB Driver Stack 5.1

Universal USB-Driver Interface 5.3

Functions for Setting the Sensor Mode 5.5

NrpChannelAssignment 5.5

NrpStartApplicationMode 5.5

Functions for Sending Commands and Data 5.6

NrpSendBinaryBlock 5.6

NrpSendCommand 5.6

Functions for Fetching Results and Data 5.7

NrpDataAvailable 5.7

NrpGetData 5.7

NrpGetBinaryResult 5.12

NrpGetBinaryResultLength 5.12

NrpGetBitfieldResult 5.12

NrpGetFloatArray 5.13

NrpGetFloatArrayLength 5.13

NrpGetFloatResult 5.13

NrpGetLongResult 5.14

NrpGetStringResult 5.14

Functions for Status and Error Detection 5.15

NrpEmptyAllBuffers 5.15

NrpEmptyErrorQueue 5.15

NrpGetLastError 5.15

NrpGetErrorText 5.17

NrpIsAlive 5.17

NrpGetTriggerState 5.17

NrpGetTriggerStateText 5.18

Callback Functions 5.18

Functions for Setting Callback Functions 5.20

NrpSetNotifyCallbackCommandAccepted 5.20

NrpSetNotifyCallbackDataAvailable 5.20

NrpSetNotifyCallbackDeviceChanged 5.20

NrpSetNotifyCallbackErrorOccured 5.20

NrpSetNotifyCallbackStateChanged 5.20

Windows Messages 5.21

USB-specific Commands 5.21
NrpOpenDriver 5.21
NrpCloseDriver 5.21
NrpClearDevice 5.21
NrpGetDeviceChangedDevName 5.21
NrpGetDeviceChangedMsgText 5.22
NrpGetDeviceID 5.23
NrpGetIDByLogicalName 5.23
NrpGetLogicalName 5.23
NrpGetManufacturerCode 5.23
NrpGetNrOfDevices 5.24
NrpGetProductID 5.24
NrpGetSerialNumber 5.24
NrpSDRGetDescriptor 5.24
NrpSelectDevice 5.25
Examples of Application 5.26
Single Average Power Measurement 5.26
Quick and Multiple Average Power Measurement 5.27
Measuring a Single Burst 5.27
Measurement of a Frame with Several Timeslots 5.28
Scope Mode for Displaying Periodic Signals 5.28
Scope Mode for Displaying a Single Process (Single-Shot Measurement) 5.29
Operating More Than One NRP Sensor on a PC 5.30

Figs.

Fig. 5-1 Description of dynamic link library NrpControl.dll 5.4

Tables

Table 5-1 Components of the USB driver stack 5.1
Table 5-2 Meaning of data types 5.7
Table 5-3 Meaning of group and parameter numbers 5.8
Table 5-4 Possible sensor error states 5.15
Table 5-5 Possible sensor trigger states 5.17
Table 5-6 Description of structure *USB_DEVICECHANGED_HANDLER_RESULT* 5.22

5 Remote Control – Fundamentals

USB Driver Stack

A USB driver stack is installed for sensors R&S NRP-Zxx at the same time as the NRP toolkit. The driver stack comprises the following files:

Table 5-1 Components of the USB driver stack

File	Function	Remarks
RSNRPZ11.sys RSNRPZ21.sys ...	USB device driver for R&S NRP sensors	
RSNRPZ11.inf RSNRPZ21.inf ...	Contains information on the installation of the USB device driver	
NRPFU.sys	USB device driver for R&S NRP sensors	Required for firmware update
NRPFU.inf	Contains information on the installation of the USB device driver	Required for firmware update
NrpControl.h	Contains the C declarations of all function calls	Should be included into the part of a C project in which the DLL is addressed [*])
NrpControl.lib	Contains the symbols for the linker exported by the DLL	Should be made known to the linker of the development environment [*])
NrpControl.dll	Holds the USB device driver to ensure more convenient communication with the sensor	Should be either in the same directory as the executable application or at least in a directory specified in the system path, e.g. %SYSTEM_ROOT%\system32; (%SYSTEM_ROOT% is an environment variable created by Windows™ that contains the Windows directory.) The file NrpControl.dll is copied to the system directory during the installation.

With the supplied components of the driver stack it is possible to address the sensor under various programming languages without involving much additional effort. In C, C++ and CVI projects, header file *NrpControl.h* and program library *NrpControl.lib* must be integrated.

Current versions of the dynamic link library *NrpControl.dll* have an integrated type library. Development environments that can access the type library are, for example, LabView and Visual Basic. The *IntelliSense* technique (automatic completion of instruction) is supported under Visual Basic.

To integrate the driver stack into Visual Basic projects, please proceed as follows when setting up the first project (description applies to the English-language version):

- In the *Project* menu, select *References* to open the *References* dialog window.
- Click the *Browse* button to open the file-selection dialog.
- Find the *NrpControl.dll* file in the system directory, and then click the *OK* button. The checkbox to the left of *NrpControl.dll* in the *References* list should now be enabled.
- Close the *References* dialog window by clicking *OK*.
- Press function key F2 to call the *Object Browser*, which lists all function calls in *NrpControl.dll*.

[*]) The files NrpControl.h and NrpControl.lib are saved to the directory C:\Program Files\Rohde&Schwarz\NRP-Toolkit\NRPControl during the installation of the NRP toolkit (installation performed with the setup type 'Typical').

For subsequent projects, all you need to do is enable the checkbox to the left of *NrpControl.dll* in the *References* list.

Universal USB-Driver Interface

The sensor R&S NRP-Zxx is addressed by the host PC via SCPI commands (see section 6 of this user manual). These commands are transmitted as ASCII strings to the sensor via the USB and interpreted by the sensor. The sensor sends measured values, parameters and other data in a binary block format.

The software developer can use the dynamic link library *NrpControl.dll* as a universal interface to remote-control the sensor R&S NRP-Zxx. This interface has an output command for SCPI commands, a command for sending binary blocks (for transmitting calibration data sets to the sensor) and several commands for querying the data sent by the sensor (measured values, device status, USB-specific information, etc.).

The binary block format used in the reverse direction should be interpreted depending on the type of data. If the sensor sends *FLOAT* values, for example, they should be either measured values or queried setting parameters. If they are setting parameters, the exact ones involved should be defined. The DLL provides mechanisms for this interpretation. Fig. 5-1 shows the basic operation of the DLL.

Within the DLL, the following data is managed in independent dynamic buffers:

- Errors that occurred
- Result descriptors (detailed information on the type of buffered data, e.g. measured values, parameters, limit values)
- Binary data
- *FLOAT* data
- *FLOAT* array data
- *LONG* data
- Bit-field data

These buffers are read out and emptied via special read functions. Functions are also available to delete these buffers. The buffers are organized as FIFO memories, i.e. the oldest data stored in a defined buffer is read and removed from the buffer.

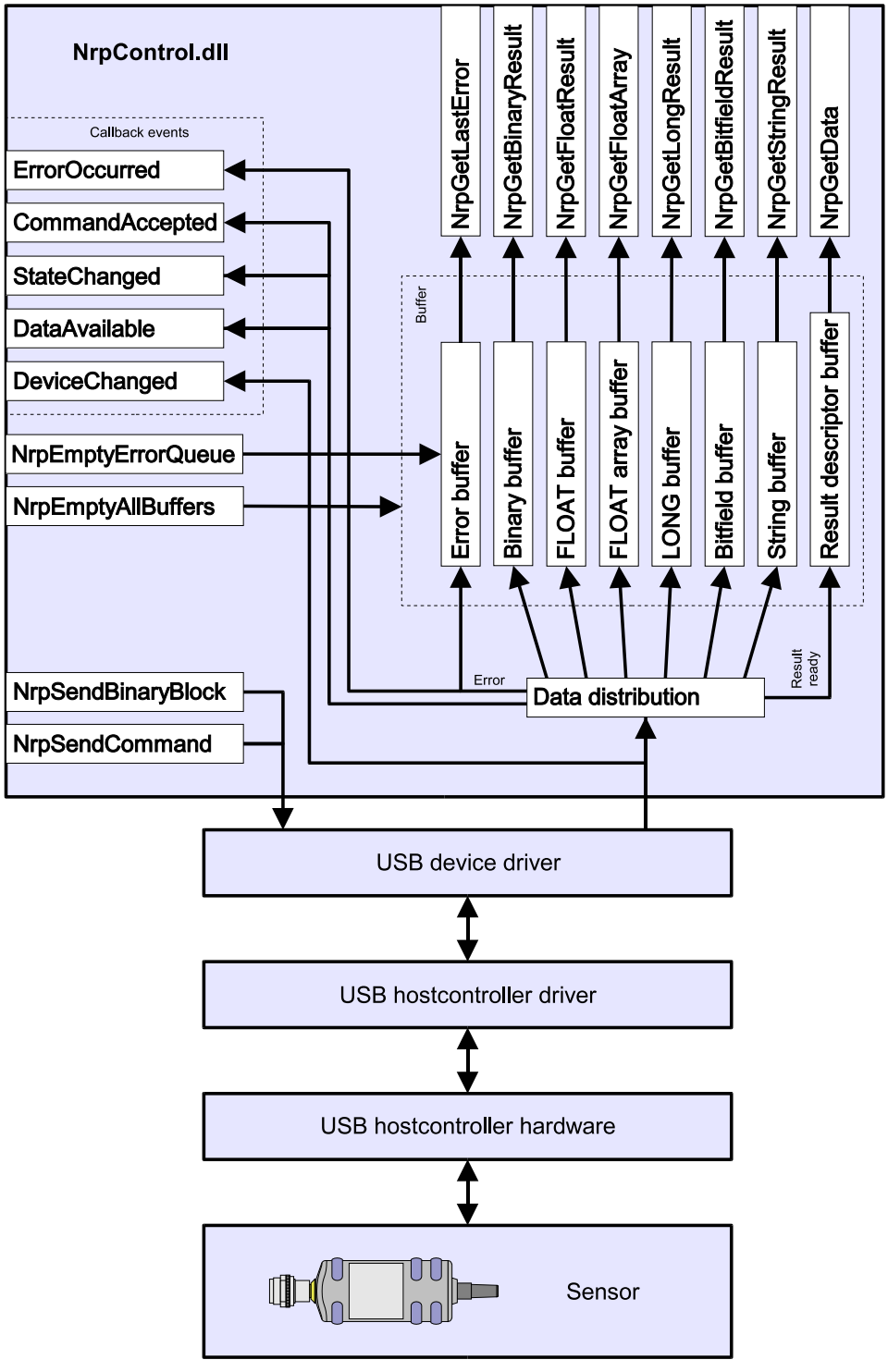

Fig. 5-1 Description of dynamic link library NrpControl.dll

Functions for Setting the Sensor Mode

NrpChannelAssignment

Calling *NrpChannelAssignment* opens a dialog window in which sensors can be assigned a straight-forward designation. This is to facilitate operation with several sensors. The sensors are uniquely identified by means of the type designation and the serial number.

Prototype: `void NrpChannelAssignment (void);`

NrpStartApplicationMode

NrpStartApplicationMode is used to set the sensor to the measurement mode.

After a power-on reset the boot loader first starts in the R&S NRP sensors. A firmware update can be performed even if the measurement firmware is defective. After a wait time of 10 s, the sensors automatically switch to the measurement mode. *NrpStartApplicationMode ()* immediately changes the mode.

Prototype: `void NrpStartApplicationMode (void);`

Functions for Sending Commands and Data

NrpSendBinaryBlock

NrpSendBinaryBlock is used to send a command, followed by a block of binary data as parameters to the sensor.

Prototype:

```
void NrpSendBinaryBlock (const char *i_pcCommand, void *i_pBuf,
                         long i_wCount);
```

Parameters:

	i_pcCommand	Pointer to the command to be sent (null-terminated string)
	i_pBuf	Pointer to the buffer containing the binary data
	i_wCount	Number of bytes to be sent

Example: `NrpSendBinaryBlock ("CALibration:DATA", caldata_ptr, 1234);`
The command *CALibration:DATA* is used to transmit a 1234-byte calibration data set to the flash memory of the sensor. *i_pBuf* points to the beginning of the memory area containing the calibration data.

NrpSendCommand

NrpSendCommand is used to send commands as ASCII strings to the sensor. The function then waits until either the sensor confirms command execution or the wait time exceeds the specified timeout. The function supplies 1 as a return value if the command was successfully executed and 0 if this is not the case.

If the value 0 is transmitted for *i_wTimeoutMs*, the function immediately returns the value 1 without waiting for a confirmation by the sensor.

Section 6 of this user manual provides an overview on the available commands.

Prototype: `long NrpSendCommand (const char *i_pcCommand, long i_wTimeoutMs);`

Parameter:

	i_pcCommand	Pointer to the command to be sent (null-terminated string)

Example: `NrpSendCommand ("*IDN?");`
The query **IDN?* is sent to the sensor.

Functions for Fetching Results and Data

NrpDataAvailable

NrpDataAvailable returns 1 if data is received from the sensor, and 0 if this is not the case and all internal buffers are empty. If a callback function is used to respond to the presence of data, it is not meaningful to use *NrpDataAvailable* at the same time.

Prototype: `long NrpDataAvailable (void);`

NrpGetData

NrpGetData determines the type of the result to be fetched (see Table 5-2) as well as the group number and the parameter number if the result is a parameter. The group number is the consecutive number of a group of associated parameters or commands. Different groups of parameters/commands are implemented for different sensor types depending on their characteristics. The parameter number is the consecutive number of a parameter/command within its group. Group and parameter numbers are 0 for measurement results.

Prototype:
`void NrpGetData (long *o_pDataType, long *o_pGroupNr, long *o_pParamNr);`

Parameters:
o_pDataType Pointer to the *LONG* variable that has to store the ordinal number of the data type
o_pGroupNr Pointer to the *LONG* variable that has to store the group number
o_pParamNr Pointer to the *LONG* variable that has to store the parameter number

Table 5-2 Meaning of data types

Data type (enum constant)	Ordinal number	Data to be fetched
DATA_BINARYBLOCK	0	A binary byte sequence can be fetched with *NrpGetBinaryResult* (calibration data set of the sensor with command *CALibration:DATA?*).
DATA_BITFIELDLIMIT	1	Three bit-field values can be fetched with *NrpGetBitfieldResult*. The first value is the default setting, the second value the bit mask of all permissible states and the third value is not used and assigned with 0.
DATA_BITFIELDPARAM	2	Three bit-field values for displaying discrete states (e.g. *OFF*, *ON*, *ONCE*) can be fetched with *NrpGetBitfieldResult*. The first value is a parameter value; the second and third values are not used and assigned with 0.
DATA_BITFIELDFEATURE	3	Three bit-field values for displaying implemented commands can be fetched with *NrpGetBitfieldResult*. The first value contains the information (a bit is set for each command or parameter within a command group, starting with bit 0 for parameter No. 1); the second and third values are not used and assigned with 0.
DATA_FLOATARRAY	4	An array of *FLOAT* values (power values) and an array of *LONG* values (indices) can be fetched with *NrpGetFloatArray*.
DATA_FLOATLIMIT	5	Three *FLOAT* values (default setting, upper limit value and lower limit value) can be fetched with *NrpGetFloatResult*.
DATA_FLOATPARAM	6	Three *FLOAT* values can be fetched with *NrpGetFloatResult*. The first value is a parameter value; the second and the third value are not used and assigned with 0.0.

Data type (enum constant)	Ordinal number	Data to be fetched
DATA_FLOATRESULT	7	Three *FLOAT* values (power value and two secondary measured values) can be fetched with *NrpGetFloatResult*.
DATA_LONGLIMIT	8	Three *LONG* values (default setting, upper limit value and lower limit value) can be fetched with *NrpGetLongResult*.
DATA_LONGPARAM	9	Three *LONG* values can be fetched with *NrpGetLongResult*. The first value is a parameter value; the second and third values are not used and assigned with 0.
DATA_STRING	10	A string can be fetched with *NrpGetStringResult*.

Table 5-3 Meaning of group and parameter numbers

Group number	Group of commands	Parameter number	Parameter or command
1	CALibration	1	CALibration:DATA
		2	CALibration:ZERO:AUTO
		3	CALibration:DATA:LENGth
2	INPut (currently reserved)		
3	SENSe	1	SENSe:AVERage:COUNt
		2	SENSe:AVERage:COUNt:AUTO
		3	SENSe:CORRection:DCYCle
		4	SENSe:AVERage:STATe
		5	SENSe:AVERage:TCONtrol
		6	SENSe:CORRection:OFFSet
		7	SENSe:CORRection:OFFSet:STATe
		8	SENSe:DCYCle:OFFSet:STATe
		9	SENSe:FREQuency
		10	SENSe:FUNCtion
		11	SENSe:EUNCertainty:STATe
		12	SENSe:RANGE
		13	SENSe:RANGE:AUTO
		14	SENSe:RANGE:CLEVel

Group number	Group of commands	Parameter number	Parameter or command
		15	SENSe:TIMing:EXCLude:STARt
		16	SENSe:TIMing:EXCLude:STOP
		17	SENSe:SAMPling
		18	SENSe:AVERage:COUNt:AUTO:MTIMe
		19	SENSe:AVERage:COUNt:AUTO:RESolution
		20	SENSe:AVERage:COUNt:AUTO:SLOT
		21	SENSe:AVERage:COUNt:AUTO:NSRatio
		22	SENSe:AVERage:COUNt:AUTO:TYPE
		23	SENSe:CORRection:SPDevice:STATe
		24	SENSe:SGAMma:MAGNitude
		25	SENSe:SGAMma:PHASe
		26	SENSe:SGAMma:CORRection:STATe
		27	SENSe:SGAMma:EUNCertainty
		28	SENSe:EUNCertainty:SGAMma:STATe
		29	SENSe:AVERage:RESet
4	SENSe:POWer:AVG	1	SENSe:POWer:AVG:APERture
		2	SENSe:POWer:AVG:BUFFer:SIZE
		3	SENSe:POWer:AVG:BUFFer:STATe
		4	SENSe:POWer:AVG:SMOothing:STATe
5	SENSe:POWer:TSLot:AVG	1	SENSe:POWer:TSLot:AVG:COUNt
		2	SENSe:POWer:TSLot:AVG:WIDTh
6	SENSe:POWer:BURSt:AVG	1	SENSe:POWer:BURSt:DTOLerance
7	SENSe:TRACe	1	SENSe:TRACe:AVERage:STATe
		2	SENSe:TRACe:OFFSet:TIMe
		3	SENSe:TRACe:POINts
		4	SENSe:TRACe:REALtime
		5	SENSe:TRACe:TIME
		6	SENSe:TRACe:AVERage:COUNt

Group number	Group of commands	Parameter number	Parameter or command
		7	SENSe:TRACe:AVERage:COUNt:AUTO
		8	SENSe:TRACe:AVERage:COUNt:AUTO:MTIMe
		9	SENSe:TRACe:AVERage:COUNt:AUTO:RESolution
		10	SENSe:TRACe:AVERage:COUNt:AUTO:POINt
		11	SENSe:TRACe:AVERage:COUNt:AUTO:NSRatio
		12	SENSe:TRACe:AVERage:COUNt:AUTO:TYPE
		13	SENSe:TRACe:AVERage:TCONtrol
8	SYSTem	1	SYSTem:INFO
		2	SYSTem:INITialize
		3	SYSTem:TRANsaction:BEGin
		4	SYSTem:TRANsaction:END
		5	SYSTem:MINPower
9	Trigger-system commands	1	ABORt
		2	INITiate:CONTinuous
		3	INITiate:IMMediate
		4	reserved
		5	reserved
		6	TRIGger:ATRigger:STATe
		7	TRIGger:COUNt
		8	TRIGger:DELay
		9	TRIGger:DELay:AUTO
		10	TRIGger:HOLDoff
		11	TRIGger:IMMediate
		12	TRIGger:LEVel
		13	TRIGger:SLOPe
		14	TRIGger:SOURce
		15	TRIGger:HYSTeresis

Group number	Group of commands	Parameter number	Parameter or command
10	SERVice	1	SERVice:DITHer
		2	SERVice:SAMPle
		3	SERVice:SIMCount
		4	SERVice:UNLock
		5	SERVice:CALibration:ZERO:NEG0
		6	SERVice:CALibration:ZERO:POS0
		7	SERVice:CALibration:ZERO:NEG1
		8	SERVice:CALibration:ZERO:POS1
		9	SERVice:CALibration:ZERO:NEG2
		10	SERVice:CALibration:ZERO:POS2
		11	SERVice:CALibration:DITHer
		12	SERVice:CALibration:DITHer:DATA
		13	SERVice:CALibration:TEMP
		14	SERVice:CALibration:TEMP:DATA
		15	SERVice:PARameter:RTEmp
		16	SERVice:PARameter:RNULL0
		17	SERVice:PARameter:RNULL1
		18	SERVice:PARameter:RNULL2
		19	SERVice:PARameter:RBAHN
		20	SERVice:PARameter:NREF
		21	SERVice:PARameter:ATHERM
		22	SERVice:PARameter:BTHERM
		23	SERVice:MVCorrection
		24	SERVice:CALibration:TEST
		25	SERVice:RCOunt
		26	SERVice:RESult
		27	SERVice:PARameter:CTHERM

Group number	Group of commands	Parameter number	Parameter or command
		28	SERVice:PARameter:DTHERM
		29	SERVice:PARameter:CJUNC
		30	SERVice:TDEScriptor?
		31	SERVice:TDEScriptor:LENGth?
11	IEEE 488.2 Common Commands	1	*RST
		2	*TRG
		3	*IDN?
		4	*TST?
12	TEST	1	TEST:SENSor

NrpGetBinaryResult

NrpGetBinaryResult reads from the binary buffer all response data blocks sent by the sensor since the function was last called and copies them to a buffer. The read bytes are deleted from the binary buffer. The oldest response data blocks are read out first. The function returns the number of bytes that are actually deleted from the binary buffer.

If the user transfers a null pointer as argument *o_pBuf* to the function, the function returns the number of bytes contained in the binary buffer.

Prototype: `long NrpGetBinaryResult (void *o_pBuf, long i_wCount);`

Parameters: *o_pBuf* Pointer to the buffer containing the result
i_wCount Buffer size in bytes

NrpGetBinaryResultLength

NrpGetBinaryResultLength returns the number of bytes contained in the binary buffer. Although the *NrpGetBinaryResult* function can also fulfill this task, a null pointer must be transferred, which causes problems in Visual Basic projects.

Prototype: `long NrpGetBinaryResultLength (void);`

NrpGetBitfieldResult

Parameters that can accept a specific number of discrete states (e.g. *OFF*, *ON* and *ONCE*) are coded as 32-bit bit fields. Each bit represents one of the possible states. In this way, the sensor can accept or exclude discrete states with dependent parameters (analogously to lower and upper limit with numeric parameters).

NrpGetBitfieldResult reads from the bit-field buffer the oldest bit-field blocks that have been sent since the function was last called by the sensor. The bit-field block read is deleted from the bit-field buffer. The bit fields are transferred as *LONG* variables.

The function returns 1 if read-out was successful and 0 if an error occurred during read-out.

Prototype:
```
long NrpGetBitfieldResult (long *o_pdwR1, long *o_pdwR2, long *o_pdwR3);
```

Parameters:

o_pdwR1	Pointer to the first of the three *LONG* variables that have to store the result.
o_pdwR2	Pointer to the second of the three *LONG* variables that have to store the result.
o_pdwR3	Pointer to the third of the three *LONG* variables that have to store the result.

NrpGetFloatArray

If the sensor is in the *buffered mode*, the results are transmitted in blocks, not individually. A particularly high transmission rate can thus be obtained. The sensor also returns its results in the form of *FLOAT* array blocks in the *Timeslot* and *Scope* modes. A fixed index is assigned to each *FLOAT* value in the array. *NrpGetFloatArray* copies a defined number of *FLOAT* values and their indices starting with the oldest values, from the DLL-internal float array buffer to a *FLOAT* or *LONG* array. The copied values are deleted from the float array buffer. The function returns the number of actually copied *FLOAT* values or indices.

If the user transfers a null pointer as *o_pFarr* argument to the function, the function supplies the number of *FLOAT* values and indices contained in the float array buffer.

Prototype:
```
long NrpGetFloatArray (float *o_pFarr , long *o_pIarr, long i_wCount);
```

Parameters:

o_pFarr	Pointer to the *FLOAT* array that has to store the *FLOAT* values
o_pIarr	Pointer to the *LONG* array that has to store the indices of *FLOAT* values
i_wCount	Length of array

NrpGetFloatArrayLength

NrpGetFloatArrayLength returns the number of *FLOAT* and index values contained in the float array buffer. Although the *NrpGetFloatArray* function can also fulfill this task, a null pointer must be transferred, which causes problems in Visual Basic projects.

Prototype: `long NrpGetFloatArrayLength (void);`

NrpGetFloatResult

NrpGetFloatResult copies the three *FLOAT* values contained in a *FLOAT* block to a *FLOAT* variable. If the *FLOAT* block is a measurement result, the first *FLOAT* value is the power value; the second and third *FLOAT* values are secondary measured values, e.g. noise and measurement uncertainty. The function returns 1 if read-out was successful and 0 if an error occurred during read-out.

Prototype:
```
long NrpGetFloatResult(float *o_pfR1, float *o_pfR2, float *o_pfR3);
```

Parameters:

o_pfR1	Pointer to the *FLOAT* variable that has to store the first *FLOAT* value
o_pfR2	Pointer to the *FLOAT* variable that has to store the second *FLOAT* value
o_pfR3	Pointer to the *FLOAT* variable that has to store the third *FLOAT* value

NrpGetLongResult

NrpGetFloatResult copies the three *LONG* values contained in a *LONG* block to a *FLOAT* variable. The function returns 1 if read-out was successful and 0 if an error occurred during read-out.

Prototype:
```
long NrpGetLongResult (long *o_pwR1, long *o_pwR2, long *o_pwR3);
```

Parameters:

	o_pwR1	Pointer to the *LONG* variable that has to store the first *LONG* value
	o_pwR2	Pointer to the *LONG* variable that has to store the second *LONG* value
	o_pwR3	Pointer to the *LONG* variable that has to store the third *LONG* value

NrpGetStringResult

NrpGetStringResult enables access to the string responses received until the function is called. A string response is usually received in several blocks, reassembled in the DLL and stored in the internal string buffer. The function copies a defined number of characters from this internal string buffer to a buffer. The copied characters are deleted from the internal string buffer. The function returns the number of characters actually read.

If the user transfers a null pointer as argument *o_pBuf* to the function, the function returns the number of characters contained in the string buffer. The terminating null character is not counted. If the terminating null character is to be read out, a buffer that is larger by 1 should be transferred.

Prototype: `long NrpGetStringResult (char *o_pBuf, long i_wCount);`

Parameters:

	o_pBuf	Pointer to the buffer in which the response has to be stored
	i_wCount	Buffer size in bytes

Functions for Status and Error Detection

NrpEmptyAllBuffers

NrpEmptyAllBuffers deletes all DLL-internal dynamic buffers.

Prototype: `void NrpEmptyAllBuffers (void);`

NrpEmptyErrorQueue

NrpEmptyErrorQueue deletes all errors contained in the error queue.

Prototype: `void NrpEmptyErrorQueue (void);`

NrpGetLastError

NrpGetLastError reads the oldest errors from the error queue. To empty the error queue, either all errors must be read out consecutively with *NrpGetLastError* or the function *NrpEmptyErrorQueue* or *NrpEmptyAllBuffers* must be called. A list of all possible error states is provided in Table 5-4.

Prototype: `long NrpGetLastError (void);`

Table 5-4 Possible sensor error states

Error (enum constant)	**Ordinal number**	**Meaning**
NRPERROR_NOERROR	0	This value is returned by the function *NrpGetLastError* if no error occurred.
NRPERROR_CALDATA_FORMAT	1	The format of the calibration data set to be transmitted to the sensor does not comply with the specification. Either the available calibration data set is not recognized as such or its version is not supported.
NRPERROR_OVERRANGE	2	The power of the test signal exceeds the range of the measurement channel selected manually (with *SENSe:RANGe:AUTO OFF*).
NRPERROR_NOTINSERVICEMODE	3	The command sent to the sensor is only available in the service mode but the sensor is not set to the service mode.
NRPERROR_CALZERO	4	Zeroing could not be performed because the RF power applied is too high.
NRPERROR_TRIGGERQUEUEFULL	5	In the *Burst Average* mode: The burst is too large and, as a result, the measurement was interrupted after the maximum permissible burst duration elapsed. The result thus obtained may deviate from the correct measured value.
NRPERROR_EVENTQUEUEFULL	6	The internal queue for trigger events is full. This error message indicates a software error since the sensor usually quits accepting trigger events when an overflow is expected.
NRPERROR_SAMPLEERROR	7	One or several samples could not be processed because the sensor did not react to the sample interrupt during the processing of a bus trigger, for example. Normally, this does not corrupt the measurement result, except in the realtime mode *(SENSe:TRACe:REALtime ON)*.

Error (enum constant)	Ordinal number	Meaning
NRPERROR_OVERLOAD	8	The power of the test signal exceeds the upper measurement limit of the sensor. The sensor may irreversibly be damaged.
NRPERROR_HARDWARE	9	A hardware error has occurred. Errors on the temperature sensor immediately generate this error message during operation. If an error was detected with test command **TST?* or *TEST:SENSor?*, the error message will be output until a new selftest detects no error. Errors in operating voltages are signalled separately (refer to *NRPERROR_VOLTAGE*).
NRPERROR_CHECKSUM	10	The checksum verification of the calibration data set to be loaded generated an error. The calibration data set was not loaded.
NRPERROR_ILLEGALSERIAL	11	An attempt was made to load a calibration data set whose serial number does not correspond to the serial number of the sensor. The calibration data set was not loaded.
NRPERROR_FILTERTRUNCATED	12	In the Auto-Averaging mode: The measurement was interrupted after the maximum measurement time (parameter *SENSe:AVERage:COUNt :AUTO:MTIMe* or *SENSe:TRACe:AVERage:COUNt:AUTO:MTIMe*) without obtaining the defined resolution or the defined signal/noise ratio.
NRPERROR_BURST_TOO_SHORT	13	In the *Burst Average* mode: A burst was too short to be correctly recognized.
NRPERROR_GENERIC	128	An error that cannot be defined more precisely has occurred.
NRPERROR_OVERMAX	129	The numeric parameter is greater than the permissible upper limit.
NRPERROR_UNDERMIN	130	The numeric parameter is smaller than the permissible lower limit.
NRPERROR_VOLTAGE	131	At least one of the internal supply voltages is not applied or is out of the permissible range.
NRPERROR_SYNTAX	132	The syntax of the command sent to the sensor is erroneous.
NRPERROR_MEMORY	133	The memory available is not sufficient. This error message indicates a firmware error.
NRPERROR_PARAMETER	134	This parameter is not allowed.
NRPERROR_TIMING	135	reserved
NRPERROR_NOTIDLE	136	The command sent to the sensor is not available during a measurement.
NRPERROR_UNKNOWNCOMMAND	137	The command sent to the sensor could not be interpreted.
NRPERROR_OUTBUFFERFULL	138	The transmit buffer of the sensor has overflowed because the data output by the sensor were not fetched.
NRPERROR_FLASHPROG	139	A flash memory programming error has occurred.
NRPERROR_CALDATANOTPRESENT	140	No valid calibration data set is present.

NrpGetErrorText

NrpGetErrorText calls the error message in plain text corresponding to the error code of the *NRPERROR* type. This message is stored as a non-null-terminated string in a buffer. The function returns the number of characters stored in the buffer.

If the user transfers a null pointer as argument *o_pBuf* to the function, the function returns the number of characters contained in the string buffer. The terminating null character is not counted. If the terminating null character is to be read out, a buffer that is larger by 1 should be transferred.

Prototype: `long NrpGetErrorText (long i_wErr, char *o_pBuf, long i_wCount);`

Parameters:

	i_wErr	Error code that was supplied by *NrpGetLastError*, for example
	o_pBuf	Pointer to the buffer in which the error text is to be stored
	i_wCount	Buffer size in bytes

NrpIsAlive

NrpIsAlive returns

- 0 if the selected active sensor is not ready for operation
- 1 if it is ready for operation and in the measurement mode
- 2 if the boot loader is active.

Prototype: `long NrpIsAlive (void);`

NrpGetTriggerState

NrpGetTriggerState returns the consecutive number of the current trigger system state (Table 5-5). The function *NrpGetTriggerStateText* can be used to obtain a short description in plain text of the trigger state.

Prototype: `long NrpGetTriggerState (void);`

Table 5-5 Possible sensor trigger states

Trigger state (enum constant)	**Ordinal number**	**Description**
SENSORHW_STATE_IDLE	0	No measurement is in progress and the sensor is ready for operation.
SENSORHW_STATE_WAIT_FOR_ARM	1	Reserved. The R&S NRP-Zxx has no ARM layer.
SENSORHW_STATE_WAIT_FOR_TRIGGER	2	A measurement was started and the sensor is waiting for a trigger event.
SENSORHW_STATE_MEASURING	3	A measurement is in progress.

NrpGetTriggerStateText

NrpGetTriggerStateText provides a short description in plain text of the trigger state determined with *NrpGetTriggerState*, for example.

If the user transfers a null pointer as argument *o_pBuf* to the function, the function returns the number of characters in the descriptive string. The terminating null character is not counted. If the terminating null character is to be read out, a buffer that is larger by 1 should therefore be transferred.

Prototype:
```
long NrpGetTriggerStateText (long i_wState, char *o_pBuf, long i_wCount);
```

Callback Functions

Callback functions that can be called by the DLL when relevant events occur can be defined to allow immediate and asynchronous evaluation of these events by the application. The following events can generate a response:

- *CommandAccepted*: A command sent to the sensor was accepted by the sensor.
- *DataAvailable*: Data from the sensor is available.
- *DeviceChanged*: A sensor was connected to the host controller or disconnected from the host controller.
- *ErrorOccurred*: The sensor has signalled an error.
- *StateChanged*: The trigger state of the sensor has changed.

The following functions are used to defined callback functions:
- *NrpSetNotifyCallbackCommandAccepted*
- *NrpSetNotifyCallbackDataAvailable*
- *NrpSetNotifyCallbackDeviceChanged*
- *NrpSetNotifyCallbackErrorOccurred*
- *NrpSetNotifyStateChanged*

These functions are described in detail in section "Functions for Setting Callback Functions" on page 5.19. The *function* type is defined as follows:
```
typedef void (__stdcall *function) (void *arg1, void *arg2);
```

The callback function to be transmitted should have the following form:
```
void __stdcall MyCallbackFunction (void *arg1, void *arg2)
{
  …
}
```

If the callback function is a class method it must be defined as *static*.

With events *CommandAccepted*, *DataAvailable*, *ErrorOccured* and *StateChanged*, the argument *arg1* is assigned *NULL* and need not be evaluated any longer. With *DeviceChanged*, a pointer is transmitted to a structure of the *USB_DEVICECHANGED_HANDLER_RESULT* type (see description of the function *NrpSetNotifyCallbackDeviceChanged* on page 5.19) that can be evaluated for differentiated event processing.

The argument *arg2* is used to transmit specific context information (e.g. reference to a class if the callback function should call a method of this class). The argument is specified on calling one of the *SetNotifyCallback* functions.

The following example shows a callback function for the *DeviceChanged* event that is implemented as class method in Visual C++. This is not a program that can be automatically executed. This applies in particular to class method *AddToProtocol* not presented here for outputting a string in a multiline edit control, for example.

Definition in the header file:

```
private:
  static void __stdcall DeviceChangedCallback(void* arg1, void* arg2);
```

Registration in the initialization section of the class:

```
NrpSetNotifyCallbackDeviceChanged(DeviceChangedCallback, this);
```

Implementation:

```
void CTestDlg::DeviceChangedCallback(void* arg1, void* arg2)
{
  CString txt, name, msg;
  long length, devices, devID, aliveState;

  USB_DEVICECHANGED_HANDLER_RESULT* hRes =
    (USB_DEVICECHANGED_HANDLER_RESULT*) arg1;

  CTestDlg* dialog = (CTestDlg*) arg2;

  length = NrpGetDeviceChangedMsgText(hRes->wEventTextHandle, NULL, 0);
  NrpGetDeviceChangedMsgText(hRes->wEventTextHandle, txt.GetBuffer(length),
    length);
  msg = "DeviceChangedMsgText:\t" + txt + "\r\n";
  dialog->AddToProtocol(msg);

  length = NrpGetDeviceChangedDevName(NULL, 0);
  NrpGetDeviceChangedDevName(name.GetBuffer(length), length);
  msg = "DeviceChangedDevName:\t" + name + "\r\n";
  dialog->AddToProtocol(msg);

  msg.Format("t\t%i Sensor(s) connected\r\n", NrpGetNrOfDevices());
  dialog->AddToProtocol(msg);
}
```

Functions for Setting Callback Functions

NrpSetNotifyCallbackCommandAccepted

NrpSetNotifyCallbackCommandAccepted defines a user-specific function that is called if a command sent to the sensor was successfully processed and acknowledged by the sensor (see section "Callback Functions" on page 5.17).

Prototype: `void NrpSetNotifyCallbackCommandAccepted (function f, void *arg);`

Parameters: *f* Function to be called
arg Argument for transferring context information

NrpSetNotifyCallbackDataAvailable

NrpSetNotifyCallbackDataAvailable defines a user-specific function that is called if measured values or other data from the sensor are available for retrieval (see section "Callback Functions" on page 5.17).

Prototype: `void NrpSetNotifyCallbackDataAvailable (function f, void *arg);`

Parameters: *f* Function to be called
arg Argument for transferring context information

NrpSetNotifyCallbackDeviceChanged

NrpSetNotifyCallbackDeviceChang defines a user-specific function that is called if a sensor is connected to the host controller or disconnected from the host controller (see section "Callback Functions" on page 5.17).

Prototype: `void NrpSetNotifyCallbackDeviceChanged (function f, void *arg);`

Parameters: *f* Function to be called
arg Argument for transferring context information

NrpSetNotifyCallbackErrorOccured

NrpSetNotifyCallbackErrorOccured defines a user-specific function that is called if an error occurred (see section "Callback Functions" on page 5.17).

Prototype: `void NrpSetNotifyCallbackErrorOccured (function f, void *arg);`

Parameters: *f* Function to be called
arg Argument for transferring context information

NrpSetNotifyCallbackStateChanged

NrpSetNotifyCallbackErrorOccured defines a user-specific function that is called if the trigger state of the sensor has changed (see section "Callback Functions" on page 5.18).

Prototype: `void NrpSetNotifyCallbackStateChanged (function f , void *arg);`

Parameters: *f* Function to be called
arg Argument for transferring context information

Windows Messages

The DLL sends and receives Windows™ messages. It requires that the calling program transfers these messages. The calling program is usually a GUI application based on the Windows API so that this condition is always fulfilled. However, if the program is a simple console application, this application will transfer no messages. The DLL offers the function *NrpMessageLoopBod*y to add a message loop in a console application. It should be cyclically called in the main loop of the console application to transfer the Windows™ messages sent and received by the DLL.

Prototype: `void NrpMessageLoopBody (void);`

USB-specific Commands

The following commands only concern the USB interface and do not refer to the sensor functionality.

NrpOpenDriver

NrpOpenDriver establishes the connection to the USB device driver.

Prototype: `void NrpOpenDriver (void);`

NrpCloseDriver

NrpCloseDriver terminates the connection to the USB device driver.

Prototype: `void NrpCloseDriver (void);`

Note: When the DLL is closed, any connection to the USB device driver is automatically cleared down.

NrpClearDevice

NrpClearDevice sends the *Vendor Request SET_DEVICE_CLEAR* to the sensor. This *Vendor Request* deletes the USB FIFOs and initializes the trigger system.

Prototype: `void NrpClearDevice (void);`

NrpGetDeviceChangedDevName

The callback function that is called by the *DeviceChanged* event supplies the structure *USB_DEVICECHANGED_HANDLER_RESULT* as an argument (see Table 8). One of the structure element is the pointer *pDeviceName* to the name of the device that has triggered the event. Since strings cannot be transferred directly as pointers in specific programming environments such as Visual Basic, the function *NrpGetDeviceChangedDevName* is offered as an alternative that copies the device name to a buffer. The function returns the number of actually copied characters.

If the user transfers a null pointer as argument *o_pBuf* to the function, the function returns the number of characters in the device-name string. The terminating null character is not counted. If the terminating null character is to be read out, a buffer that is larger by 1 should be transferred.

Prototype: `long NrpGetDeviceChangedDevName (char *o_pBuf, long i_wCount);`

Parameters: *o_pBuf* Pointer to the buffer that has to store the device name
i_wCount Buffer size in bytes

Table 5-6 Description of structure *USB_DEVICECHANGED_HANDLER_RESULT*

Type	Name	Description
long	wResult	The value is 1 if a device was assigned to the event and 0 if this is not the case (the string to which pDeviceName points is empty).
long	wEventType	Can take the values *SENSORHW_STATE_DEVICEARRIVAL* to *SENSORHW_STATE_DEVICEUNKNOWN*.
long	wEventTextHandle	Text handle that enables the function *NrpGetDeviceChangedMsgText* to determine the plain-text description of the event.
char*	pEventText	Pointer to the plain-text description of the event. This pointer should only be used in a C or C++ environment. The functions *NrpGetDeviceChangedMsgText* and *NrpGetDeviceChangedMsgTextLength* must be used for Visual Basic.
char*	pDeviceName	The device name that has triggered the event. This pointer should only be used in a C or C++ environment. The functions *NrpGetDeviceChangedDevName* and *NrpGetDeviceChangedDevNameLength* must be used for Visual Basic.

NrpGetDeviceChangedMsgText

The callback function that is called by the *DeviceChanged* event returns the structure *USB_DEVICECHANGED_HANDLER_RESULT* as an argument (see Table 5-6). One of the structure element is *wEventTextHandle*, a *LONG* value behind which a DLL-internal string-list index is concealed. *NrpGetDeviceChangedMsgText* is used to copy the event message stored in the string list to a buffer. The function returns the number of actually copied characters.

If the user transfers a null pointer as argument *o_pBuf* to the function, the function returns the number of characters in the event message. The terminating null character is not counted. If the terminating null character is to be read out, a buffer that is larger by 1 should be transferred.

Prototype:

```
long NrpGetDeviceChangedMsgText (long wEventTextHandle, char *o_pBuf,
                                 long i_wCount);
```

Parameters: *wEventTextHandle* String list index
o_pBuf Pointer to the buffer that has to store the event message
i_wCount Buffer size in bytes

NrpGetDeviceID

NrpGetDeviceID returns the consecutive number of the active sensor. Each connected sensor is assigned such a consecutive number. Counting starts with 0.

Prototype: `long NrpGetDeviceID (void);`

NrpGetIDByLogicalName

Each sensor has a unique combination of type designation and serial number that is automatically assigned a logical name when it is connected to the USB for the first time. These assignments are entered in the registry under *HKEY_CURRENT_USER\Software\ Rohde&Schwarz\NrpDll\Sensors*. They can be modified by means of function *NrpChannelAssignment*. *NrpGetIDByLogicalName* returns the consecutive number of a connected sensor if the transferred logical name was found in the registry. If the logical name was not found in the registry or if the respective sensor is not connected, the function returns –1.

Prototype: `long NrpGetIDByLogicalName (const char *i_pLogicalName);`

Parameters: *i_pLogicalName* Logical name of the sensor as null-terminated string

NrpGetLogicalName

Each sensor has a unique combination of type designation and serial number that is automatically assigned a logical name when it is connected to the USB for the first time. These assignments are entered in the registry under *HKEY_CURRENT_USER\Software\ Rohde&Schwarz\NrpDll\Sensors*. They can be modified by means of function *NrpChannelAssignment*. *NrpGetLogicalName* copies the logical name of the connected sensor with the consecutive number transferred as an argument to a buffer. The function returns the number of actually copied characters.

If the user transfers a null pointer as argument *o_pBuf* to the function, the function returns the number of characters in the logical name. The terminating null character is not counted. If the terminating null character is to be read out, a buffer that is larger by 1 should be transferred.

Prototype: `long NrpGetLogicalName (long i_wID, char *o_pBuf, long i_wCount);`

Parameters:

	i_wID	Consecutive number of the sensor
	o_pBuf	Pointer to the buffer that has to store the logical name
	i_wCount	Buffer size in bytes

NrpGetManufacturerCode

NrpGetManufacturerCode copies the manufacturer name of the connected sensor with the consecutive number transferred as an argument to a buffer. The function returns the number of actually copied characters.

If the user transfers a null pointer as argument *o_pBuf* to the function, the function returns the number of characters in the logical name. The terminating null character is not counted. If the terminating null character is to be read out, a buffer that is larger by 1 should be transferred.

Prototype:
```
long NrpGetManufacturerCode (long i_wID, char *o_pBuf, long i_wCount);
```

Parameters:	*i_wID*	Consecutive number of the sensor
	o_pBuf	Pointer to the buffer that has to store the manufacturer name
	i_wCount	Buffer size in bytes

NrpGetNrOfDevices

NrpGetNrOfDevices returns the number of sensors connected to the USB.

Prototype: `long NrpGetNrOfDevices (void);`

NrpGetProductID

NrpGetProductID copies the type designation of the connected sensor with the consecutive number transferred as an argument to a buffer. The function returns the number of actually copied characters.

If the user transfers a null pointer as argument *o_pBuf* to the function, the function returns the number of characters in the type designation. The terminating null character is not counted. If the terminating null character is to be read out, a buffer that is larger by 1 should therefore be transferred.

Prototype:
```
long NrpGetProductID(long i_wID, char *o_pBuf, long i_wCount);
```

Parameters:	*i_wID*	Consecutive number of the sensor
	o_pBuf	Pointer to the buffer that has to store the type designation
	i_wCount	Buffer size in bytes

NrpGetSerialNumber

NrpGetSerialNumber copies the serial number string of the connected sensor with the consecutive number transferred as an argument to a buffer. The function returns the number of actually copied characters.

If the user transfers a null pointer as argument *o_pBuf* to the function, the function returns the number of characters in the serial number string. The terminating null character is not counted. If the terminating null character is to be read out, a buffer that is larger by 1 should therefore be transferred.

Prototype:
```
long NrpGetProductID(long i_wID, char *o_pBuf, long i_wCount);
```

Parameters:	*i_wID*	Consecutive number of the sensor
	o_pBuf	Pointer to the buffer that has to store the serial number string
	i_wCount	Buffer size in bytes

NrpSDRGetDescriptor

NrpSDRGetDescriptor sends the *Standard Device Request GetDescriptor* to the sensor. This *Standard Device Request* is used to read device, configuration and string descriptors. Interface and endpoint descriptors cannot be read out directly but only together with the configuration descriptor. The function returns the error status according to Table 5-4. The structure of descriptors is described in detail in the technical literature on USB.

Prototype:
```
long NrpSDRGetDescriptor (void *o_pBuffer, long *io_wByteCount,
                          long i_wRecipient, long i_wDescrType,
                          long i_wDescrIndex, long i_wLangID);
```

Parameters:	*o_pBuffer*	Pointer to the buffer that has to store the descriptor data
	io_wByteCount	After being called: Buffer size in bytes After being called: Number of bytes actually stored in the buffer
	i_wRecipient	0 = device, 1 = interface, 2 = endpoint, 3 = other
	i_wDescrType	Type of descriptor (1 = device descriptor, 2 = configuration descriptor incl. associated interface, class and endpoint descriptors, 3 = string descriptor)
	i_wDescrIndex	Only with string descriptor: Index of the string for manufacturer name, product name or series number
	i_wLangID	Only with string descriptor: Language index (optional, 0x0409 = English), otherwise 0

NrpSelectDevice

NrpSelectDevice selects the connected sensor with the consecutive number transferred as an argument.

Prototype: `void NrpSelectDevice (long i_wID);`

Parameter: *i_wID* Consecutive number of the sensor

Examples of Application

The following examples illustrate the sensor functionality. The program module **USB-Terminal** of the NRP toolkit is suitable for learning the commands. The sensor is ready for operation immediately after it has been connected to the USB connector of the PC.

Commands must be sent to the sensor by means of the function *NrpSendCommand*. For clarity, only the SCPI command sequences are shown below. Only the short form of commands is used. For more details on the commands, refer to the command reference in section 6 of this user manual.

Before the measurement zeroing should be performed with the sensor with power switched off.

```
cal:zero:auto once      or      cal:zero:auto on
```

In addition, the RF carrier frequency (500 MHz in the following example) should be reported to the sensor for each measurement.

```
sens:freq 500e6
```

Single Average Power Measurement

In the simplest case, only the following commands are required to measure the average power:

`*rst`	resets the sensor to the default settings
`init:imm`	starts the measurement

An averaging filter is provided in all measurement modes. This filter can be operated with an averaging factor that is either manually set or automatically determined by the sensor. If the default setting is used, the sensor automatically determines the averaging factor. Two algorithms can be selected to determine the averaging factor. With the classic method, the required resolution is preset and the sensor determines the filter length using the power applied and the upper limit for the measurement time is complied with (parameter *SENSe:AVERage:COUNt:AUTO:MTIMe*). With the new fixed-noise method (*SENSe:AVERage:COUNt:AUTO:TYPE NSRatio*), however, an exactly defined noise component is complied with in the result.

The averaging filter can operate in two different ways. The parameter *SENSe:AVERage:TCONtrol* can accept the values *MOVing* and *REPeat*.

- The *REPeat* setting is usually of advantage in the remote-control mode. The average value is output when the filter is completely filled. In the *Continuous Average* mode, only one trigger event is required to start the measurement for filling the averaging filter; the other measurements are automatically performed. In the modes *Burst Average*, *Timeslot* and *Scope*, one trigger event is required for each measured value because of the time reference.
- With the *MOVing* setting, the average value is formed and output each time a measured value is shifted into the filter. This filter behaviour allows detection at an early stage of a trend in the displayed average value. This behaviour is also used in manual operation via the basic unit.

Example of configuration with manual averaging and *REPeating* behaviour:

```
sens:aver:coun:auto off
sens:aver:coun 8
sens:aver:tcon rep
```

Example of configuration with auto-averaging and fixed-noise method (noise component: 0.01 dB, upper limit for measuring time: 30 s):

```
sens:aver:coun:auto on
sens:aver:coun:auto:type nsr
sens:aver:coun:auto:nsr 0.01
sens:aver:count:auto:mtim 30
```

The averaging filter settings are the same for the modes *Continuous Average*, *Burst Average* and *Timeslot* and are accepted if the mode changes. The filter is specially configured for the *Scope* mode; the commands begin with *SENSe:TRACe:AVERage: …*

Quick and Multiple Average Power Measurement

For quick successive measurements, the sensor provides a result buffer with settable size. The content of this buffer is transferred to the control unit (basic unit/PC) after the buffer is filled with measured values. Since the content is transferred in *FLOAT* array blocks, the transmission is particularly time-saving. The following example shows the settings for the quickest possible operation:

`*rst`	resets the sensor to default settings
`sens:freq 500e6`	sets the RF carrier frequency (here 500 MHz)
`sens:aver:stat off`	switches off the averaging filter
`sens:pow:avg:aper 100e-6`	sets the shortest measurement window (here 100 µs)
`sens:pow:avg:buff:size 100`	the result buffer has to store 100 measured values
`sens:pow:avg:buff:stat on`	switches on the result buffer
`trig:sour imm`	free-running trigger
`trig:coun 100`	stores 100 measured values
`init:imm`	starts the measurement

The command `trig:count 100` ensures that 100 measurements are performed after a single measurement start with `init:imm` to fill the buffer. If, however, the default setting `trig:count 1` is used, `init:imm` would have to be called 100 times successively for the same number of results.

Measuring a Single Burst

In the *Burst Average* mode, the sensor can automatically detect individual bursts and measure their average power value. In this measurement mode, the sensor waits for a rising trigger edge and continues measuring until it recognizes a falling edge. If no burst end is detected within 50 ms, the sensor interrupts the measurement and outputs the error message *NRPERROR_TRIGGERQUEUEFULL*.

`*rst`	resets the sensor to default settings
`sens:func "pow:burs:avg"`	sets the *Burst Average* mode
`trig:lev 1e-6`	sets the trigger threshold to 1 µW
`sens:pow:burs:dtol 50e-6`	sets the dropout tolerance
`init:imm`	starts the measurement

In addition, sections at the burst beginning and end can be excluded (e.g. to eliminate the influence of flat signal edges or overshoots on the measurement result). In the following example, the first 20 µs and the last 30 µs of the burst are excluded from the measurement:

```
sens:tim:excl:star 20e-6
sens:tim:excl:stop 30e-6
```

Measurement of a Frame with Several Timeslots

The sensor provides the *Timeslot* mode for measuring the power in several successive timeslots in a fixed time frame. The number and duration of timeslots have to be set for the measurement. A measurement is triggered by the test signal *(TRIGger:SOURce INTernal)* or by an externally applied trigger signal *(TRIGger:SOURce EXTernal)*. An external trigger source is preferable to avoid measurement errors due to inaccurate triggering. Triggering via the USB *(TRIGger:SOURce BUS)* is basically possible, but it is not recommended because of the inaccurate time resolution.

The following example shows how the power can be measured in the eight timeslots of a GSM/EDGE signal:

`*rst`	resets the sensor to default settings
`sens:func "pow:tsl:avg"`	selects the *Timeslot* mode
`sens:freq 500e6`	sets the RF carrier frequency (here 500 MHz)
`sens:aver:coun:auto off`	deactivates auto-averaging
`sens:aver:coun 8`	sets averaging factor 8
`trig:sour ext`	selects an external trigger source (*int* can also be entered for internal triggering; the trigger threshold should then be defined with *TRIGger:LEVel*.)
`trig:slop pos`	triggering to the rising edge of the trigger signal
`sens:pow:tsl:avg:coun 8`	number of successive timeslots (here 8)
`sens:pow:tsl:avg:widt 570e-6`	width of a timeslot in seconds (here 570 µs = width of a GSM timeslot)
`init:imm`	starts the measurement

As in the *Burst* mode, the start and end sections of timeslots can be excluded from the measurement in the *Timeslot* mode. The setting parameters are the same as in the *Burst Average* mode:

`sens:tim:excl:star 50e-6`	the first 50 µs of each timeslot are blanked
`sens:tim:excl:stop 80e-6`	the last 80 µs of each timeslot are blanked

The parameters *SENSe:TIMing:EXCLude:STARt* and *STOP* are valid for both the *Burst Average* mode and the *Timeslot* mode, i.e. the settings performed in either of the two modes are accepted in the other mode when the mode changes.

Scope Mode for Displaying Periodic Signals

The sensor provides the *Scope* mode to display the time sequence of power in the same manner as with an oscilloscope. As in the *Timeslot* mode, triggering can be performed by the test signal or an externally applied trigger signal. An external trigger source is also preferable to avoid measurement errors due to inaccurate triggering.

The *Scope* mode provides separate parameters for the averaging filter.

To configure the sensor for a *Scope* measurement, the time to be covered by the *Scope* presentation and the number of intervals in which the presentation is to be resolved have to be specified. The smallest time resolution of the sensor is 10 µs. Ensure that the interval width (= time/number of intervals) does not exceed this time when setting the above parameters since otherwise the *Scope* presentation is corrupted.

The following example shows the use of the *Scope* mode, the trigger being derived from the test signal:

`*rst`	resets the sensor to default settings
`sens:func "xtim:pow"`	selects the *Scope* mode
`sens:freq 500e6`	sets the RF carrier frequency (here 500 MHz)
`sens:trac:aver:coun:auto off`	deactivates auto-averaging
`sens:trac:aver:coun 8`	sets averaging factor 8
`trig:sour int`	selects triggering by the test signal
`trig:slop pos`	triggering to the rising edge of the test signal
`trig:lev 1e-6`	sets the trigger threshold (here 1 µW)
`trig:hyst 3`	sets the trigger hysteresis (here 3 dB, i.e. new triggering due to the trigger threshold being exceeded is only possible if the test signal level has underranged the trigger threshold by at least 3 dB)
`sens:trac:time 20e-3`	time in seconds to be covered by the Scope presentation (here 20 ms)
`sens:trac:poin 200`	number of intervals in which the Scope presentation is to be resolved (here 200, i.e. resolution = 20 ms / 200 = 100 µs)
`init:imm`	starts the measurement

Scope Mode for Displaying a Single Process (Single-Shot Measurement)

A single non-periodic signal can be detected in the *Scope* mode if the parameter *SENSe:TRACe:REALtime ON* is set. Averaging is of course not possible with single processes. The averaging filter settings are therefore ignored (and not modified). In contrast to the last two modes, bus triggering is possible here since averaging does not cause several measurements to be superimposed.

The following example shows the settings for the single-shot measurements:

`*rst`	resets the sensor to default settings
`sens:func "xtim:pow"`	selects the *Scope* mode
`sens:freq 500e6`	sets the RF carrier frequency (here 500 MHz)
`sens:trac:time 20e-3`	time in seconds to be covered by the Scope presentation (here 20 ms)
`sens:trac:poin 200`	number of intervals in which the Scope presentation is to be resolved (here 200, i.e. resolution = 20 ms / 200 = 100 µs)
`sens:trac:real on`	activates the single-shot mode
`trig:sour ext`	selects an external trigger source (*int* can also be entered for internal triggering; the trigger threshold should then be defined with *TRIGger:LEVel*.)
`trig:slop pos`	triggering to the rising edge of the trigger signal
`init:imm`	starts the measurement

The following commands are required for a bus-triggered single-shot measurement:

`trig:sour hold`	ignores all trigger events, except *TRIGger:IMMediate*
`init:imm`	initializes the measurement; the trigger system waits for a trigger event
`trig:imm`	triggers the single-shot measurement

Note that the trigger delay (parameter *TRIGger:DELay*) is ignored due to the command *TRIGger:IMMediate*.

Operating More Than One NRP Sensor on a PC

The dynamic link library *NrpControl.dll* makes it possible to operate up to 127 sensors simultaneously, but measurements can only be performed sequentially (i. e. using only one sensor at a time). The sensor to be used for measuring is selected with the DLL function *NrpSelectDevice*. Subsequently it can be configured and the measurement started. All the other sensors which were not selected retain their previously set configurations, but they neither return measurement results nor can they be addressed.

To clearly identify all the sensors connected to a PC, they are assigned logical names which, for instance, can be compared with the addresses in an IEC bus measurement system. The logical names can be displayed and edited using the DLL function *NrpChannelAssignment*. This function opens a dialog window in which all the connected sensors (including sensors that were connected in the past but are no longer connected) are listed with their type designation, serial number and logical name. When sensors are initially connected to the PC, the USB driver assigns them a default name (Sensor1, Sensor2, Sensor3, etc) which can either be retained or changed.

Unfortunately sensors cannot be directly addressed using their logical name; the *i_wID* parameter that identifies the sensor is needed. This is a consecutive number (0, 1, 2, etc.) which the USB driver allocates relatively arbitrarily every time the PC is switched on or the sensor is changed. In such cases, this number must always be redetermined. This is done by using the *NrpGetIDByLogicalName* function, which determines the current consecutive number of a sensor from its logical name. If the desired sensor is not connected, the function returns the value –1.