Inhaltsübersicht

5 Fernbedienung – Grundlagen 5.1

USB-Treiberstack 5.1
Universelle USB-Treiber-Schnittstelle 5.3
Funktionen zum Einstellen der Messkopf-Betriebsart 5.5
NrpChannelAssignment 5.5
NrpStartApplicationMode 5.5
Funktionen zum Senden von Befehlen und Daten 5.6
NrpSendBinaryBlock 5.6
NrpSendCommand 5.6
Funktionen zum Abholen von Messergebnissen und Daten 5.7
NrpDataAvailable 5.7
NrpGetData 5.7
NrpGetBinaryResult 5.13
NrpGetBinaryResultLength 5.13
NrpGetBitfieldResult 5.13
NrpGetFloatArray 5.13
NrpGetFloatArrayLength 5.14
NrpGetFloatResult 5.14
NrpGetLongResult 5.14
NrpGetStringResult 5.15
Funktionen zur Status- und Fehlererkennung 5.16
NrpEmptyAllBuffers 5.16
NrpEmptyErrorQueue 5.16
NrpGetLastError 5.16
NrpGetErrorText 5.18
NrpIsAlive 5.18
NrpGetTriggerState 5.18
NrpGetTriggerStateText 5.19
Callback-Funktionen 5.19
Funktionen zum Setzen von Callback-Funktionen 5.21
NrpSetNotifyCallbackCommandAccepted 5.21
NrpSetNotifyCallbackDataAvailable 5.21
NrpSetNotifyCallbackDeviceChanged 5.21
NrpSetNotifyCallbackErrorOccured 5.21
NrpSetNotifyCallbackStateChanged 5.21
Windows-Messages 5.22

USB-spezifische Befehle ... 5.22
NrpOpenDriver ... 5.22
NrpCloseDriver ... 5.22
NrpClearDevice ... 5.22
NrpGetDeviceChangedDevName ... 5.22
NrpGetDeviceChangedMsgText ... 5.23
NrpGetDeviceID ... 5.24
NrpGetIDByLogicalName ... 5.24
NrpGetLogicalName ... 5.24
NrpGetManufacturerCode ... 5.24
NrpGetNrOfDevices ... 5.25
NrpGetProductID ... 5.25
NrpGetSerialNumber ... 5.25
NrpSDRGetDescriptor ... 5.25
NrpSelectDevice ... 5.26
Anwendungsbeispiele ... 5.27
Einfache Leistungsmittelwertmessung ... 5.27
Schnelle und mehrfache Leistungsmittelwertmessung ... 5.28
Messung eines einzelnen Bursts ... 5.28
Messung eines Frames mit mehreren Zeitfenstern (Timeslots) ... 5.29
Scopemodus zur Darstellung von periodischen Signalen ... 5.29
Scopemodus zur Darstellung eines einmaligen Vorgangs (Single-Shot-Messung) ... 5.30
Zum Umgang mit mehreren NRP-Messköpfen an einem PC ... 5.31

Bilder

Bild 5-1 Funktionsweise der Dynamic Link Library NrpControl.dll 5.4

Tabellen

Tabelle 5-1 Teile des USB-Treiberstacks 5.1
Tabelle 5-2 Bedeutung der Datentypen 5.7
Tabelle 5-3 Bedeutung der Gruppen- und Parameternummern 5.8
Tabelle 5-4 Mögliche Messkopf-Fehlerzustände 5.16
Tabelle 5-5 Mögliche Messkopf-Triggerzustände 5.18
Tabelle 5-6 Beschreibung der Struktur *USB_DEVICECHANGED_HANDLER_RESULT* 5.23

5 Fernbedienung – Grundlagen

USB-Treiberstack

Bei der Installation des NRP-Toolkit wird u. a. ein USB-Treiberstack für die Messköpfe R&S NRP-Zxx installiert. Dieser besteht aus folgenden Dateien:

Tabelle 5-1 Teile des USB-Treiberstacks

Datei	**Funktion**	**Bemerkung**
RSNRPZ11.sys RSNRPZ21.sys ...	USB-Device-Treiber für R&S NRP-Messköpfe	
RSNRPZ11.inf RSNRPZ21.inf ...	enthält Installationsinformationen für USB-Device-Treiber	
NRPFU.sys	USB-Device-Treiber für R&S NRP-Messköpfe	wird benötigt für Firmware-Update
NRPFU.inf	enthält Installationsinformationen für USB-Device-Treiber	wird benötigt für Firmware-Update
NrpControl.h	enthält die C-Deklarationen aller Funktionsaufrufe	muss in den Teil eines C-Projektes inkludiert werden, in dem die DLL angesprochen wird. *)
NrpControl.lib	enthält die von der DLL exportierten Symbole für den Linker	muss dem Linker der Entwicklungsumgebung bekannt gemacht werden. *)
NrpControl.dll	kapselt den USB-Device-Treiber, um eine komfortablere Kommunikation mit dem Messkopf zu ermöglichen	sollte entweder im selben Verzeichnis liegen wie die ausführbare Applikation, oder mindestens in einem Verzeichnis, das im Systempfad aufgeführt ist, z. B. %SYSTEM_ROOT%\system32; (%SYSTEM_ROOT% ist eine von Windows™ angelegte Umgebungsvariable, die das Windows-Verzeichnis enthält). Die Datei NrpControl.dll wird bei der Installation des NRP-Toolkits in das Systemverzeichnis kopiert.

Mit den mitgelieferten Bestandteilen des Treiberstacks ist es möglich, den Messkopf mit geringem Zusatzaufwand unter verschiedenen Programmiersprachen anzusprechen: In C-, C++- und CVI-Projekte müssen das Headerfile *NrpControl.h* und die Programmbibliothek *NrpControl.lib* eingebunden werden.

Aktuelle Versionen der *Dynamic Link Library NrpControl.dll* besitzen eine eingebundene Typbibliothek. Entwicklungsumgebungen, die auf die Typbibliothek zugreifen, sind z. B. LabView und Visual Basic. Unter Visual Basic wird die *IntelliSense*-Technik (automatische Anweisungsvervollständigung) unterstützt.

Um den Treiberstack in Visual-Basic-Projekte einzubinden, beim ersten Mal bitte folgendermaßen vorgehen (Beschreibung gilt für die englischsprachige Version):

- Im Menü *Project* den Eintrag *References* auswählen. Es öffnet sich das Dialogfenster *References*.
- Um Dateiauswahl-Dialog zu öffnen, Button *Browse* betätigen.
- Datei *NrpControl.dll* im Systemverzeichnis suchen, anschließend Button *OK* betätigen. Die Checkbox in der *References*-Liste vor dem Eintrag *NrpControl.dll* muss jetzt aktiviert sein.

*) Die Dateien NrpControl.h und NrpControl.lib werden bei der Installation des NRP-Toolkits im Verzeichnis c:\Programme\Rohde&Schwarz\NRP-Toolkit\NRPControl gespeichert (Installation mit der Einstellung 'Typical' durchgeführt).

- Dialogfenster *References* mit *OK* schließen.
- Mit der Funktionstaste F2 wird der *Object Browser* aufgerufen. Darin werden alle Funktionsaufrufe in *NrpControl.dll* angezeigt.

Bei weiteren Projekten muss nur die Checkbox in der *References*-Liste vor dem Eintrag *NrpControl.dll* aktiviert werden.

Universelle USB-Treiber-Schnittstelle

Der Messkopf R&S NRP-Zxx wird vom Host-PC über SCPI-Befehle (siehe Abschnitt 6 dieses Benutzerhandbuchs) angesprochen. Diese Befehle werden als ASCII-Strings über den USB zum Messkopf übertragen und von diesem interpretiert. Der Messkopf sendet dagegen Messwerte, Parameter und sonstige Daten in einem Binärblock-Format.

Dem Softwareentwickler wird mit der Dynamic Link Library *NrpControl.dll* eine universelle Schnittstelle angeboten, mit der er den Messkopf R&S NRP-Zxx fernsteuern kann. Diese Schnittstelle umfasst einen Ausgabebefehl für die SCPI-Befehle, einen Befehl zum Senden von Binärblöcken (zur Übertragung von Kalibrierdatensätzen in den Messkopf) und mehrere Befehle zum Abfragen der vom Messkopf gesendeten Daten (Messwerte, Gerätestatus, USB-spezifische Informationen usw.)

Die Interpretation des in Rückrichtung benutzten Binärblock-Formats muss in Abhängigkeit von der Art der Daten erfolgen. Sendet der Messkopf z. B. *FLOAT*-Werte, so kann es sich entweder um Messwerte oder um abgefragte Einstellparameter handeln. Sind es Einstellparameter, dann muss festgestellt werden, um welche genau es sich handelt. Die DLL stellt Mechanismen für diese Interpretation bereit. Bild 5-1 stellt die prinzipielle Funktionsweise der DLL dar.

Folgende Daten werden innerhalb der DLL in unabhängigen, dynamischen Puffern verwaltet:

- aufgetretene Fehler
- Ergebnis-Deskriptoren (nähere Informationen über die Art der gepufferten Daten, z. B. Messwerte, Parameter, Grenzwerte)
- binäre Daten
- *FLOAT*-Daten
- *FLOAT*-Array-Daten
- *LONG*-Daten
- Bitfeld-Daten

Ausgelesen und entleert werden diese Puffer über spezielle Lesefunktionen. Es existieren auch Funktionen zum gezielten Löschen dieser Puffer. Die Puffer sind als FIFO-Speicher organisiert, d. h. es werden beim Auslesen eines bestimmten Puffers stets die ältesten darin vorhandenen Daten ausgelesen und dabei aus dem Puffer entfernt.

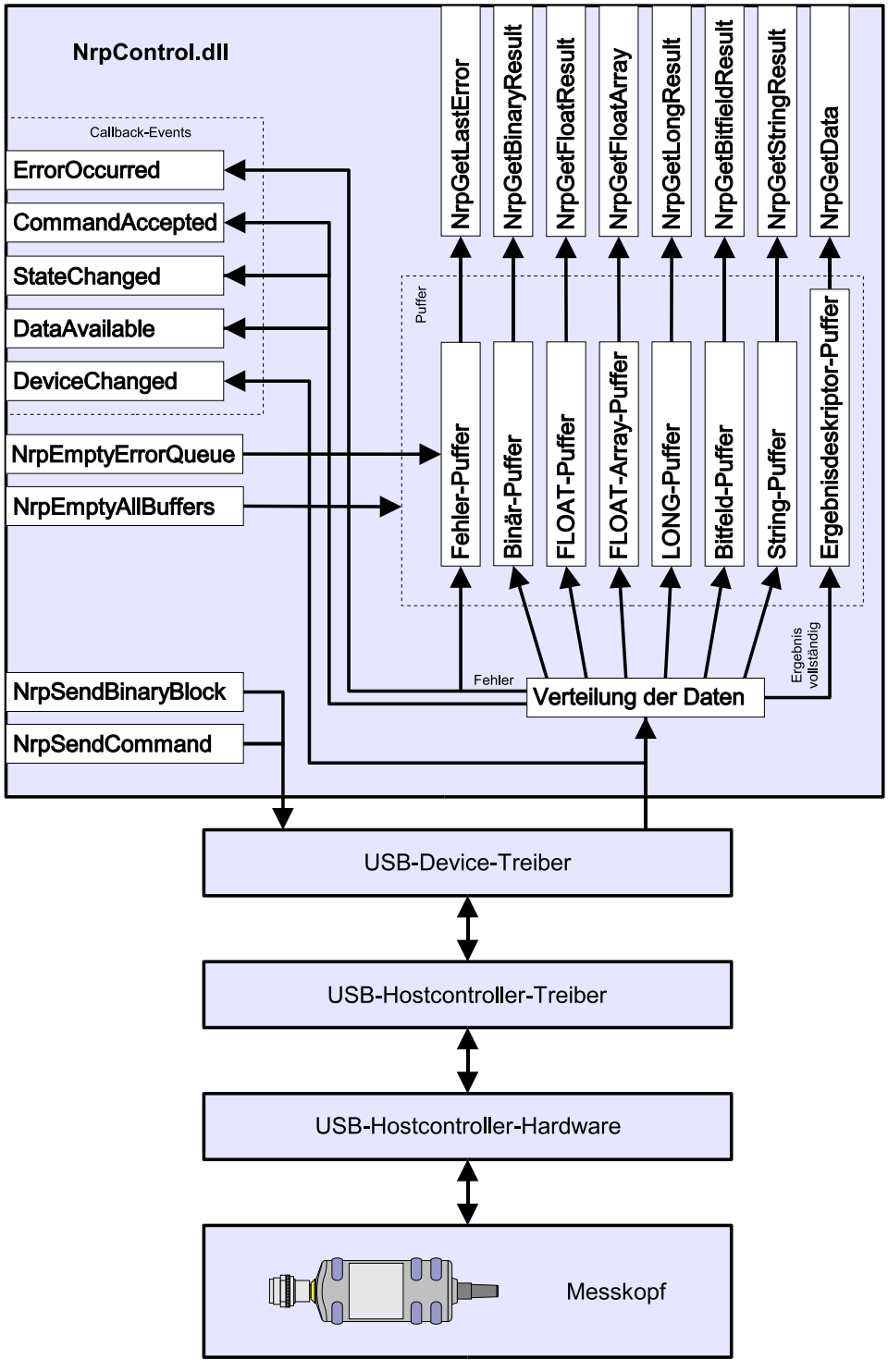

Bild 5-1 Funktionsweise der Dynamic Link Library NrpControl.dll

Funktionen zum Einstellen der Messkopf-Betriebsart

NrpChannelAssignment

Der Aufruf von *NrpChannelAssignment* öffnet einen Dialog, der es ermöglicht, Messköpfen eine leicht verständliche Bezeichnung zuzuordnen. Dies soll die Arbeit mit mehreren Messköpfen erleichtern. Dabei werden die Messköpfe mit Hilfe der Typenbezeichnung und Seriennummer eindeutig identifiziert.

Prototyp: `void NrpChannelAssignment (void);`

NrpStartApplicationMode

Mit *NrpStartApplicationMode* wird der Messkopf in den Messmodus versetzt.

Nach einem Power-on-Reset startet bei den R&S NRP-Messköpfen zunächst der Bootlader. Dies ermöglicht selbst bei einer defekten Mess-Firmware einen Firmware-Update. Nach einer Wartezeit von 10 s wechseln die Messköpfe automatisch in den Messmodus. Mit *NrpStartApplicationMode ()* erfolgt der Moduswechsel sofort.

Prototyp: `void NrpStartApplicationMode (void);`

Funktionen zum Senden von Befehlen und Daten

NrpSendBinaryBlock

Mit *NrpSendBinaryBlock* ist es möglich, einen Befehl, gefolgt von einem Block binärer Daten als Parameter, an den Messkopf zu senden.

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

Parameter: *i_pcCommand* Zeiger auf den zu sendenden Befehl (nullterminierter String)
i_pBuf Zeiger auf den Puffer, der die Binärdaten enthält
i_wCount Anzahl der zu sendenden Bytes

Beispiel: `NrpSendBinaryBlock ("CALibration:DATA", caldata_ptr, 1234);`
Der Befehl *CALibration:DATA* wird zum Übertragen eines 1234 Bytes langen Kalibrierdatensatzes in den Flash-Speicher des Messkopfes genutzt. Der Zeiger *i_pBuf* zeigt auf den Anfang des Speicherbereiches, in dem die Kalibrierdaten stehen.

NrpSendCommand

Mit *NrpSendCommand* werden Befehle als ASCII-Strings an den Messkopf gesendet. Danach wartet die Funktion, bis entweder der Messkopf die Ausführung des Befehls bestätigt oder die Wartezeit die angegebene Timeout-Zeit überschreitet. Die Funktion liefert als Rückgabewert 1, wenn der Befehl erfolgreich ausgeführt wurde, und 0, wenn wenn dies nicht der Fall ist.

Wird für *i_wTimeoutMs* der Wert 0 übergeben, so liefert die Funktion sofort den Rückgabewert 1, ohne auf eine Bestätigung vom Messkopf zu warten.

Eine Übersicht über die verfügbaren Befehle bietet Abschnitt 6 dieses Benutzerhandbuchs.

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

Parameter: *i_pcCommand* Zeiger auf den zu sendenden Befehl (nullterminierter String)

Beispiel: `NrpSendCommand ("*IDN?");`
Der Abfragebefehl **IDN?* wird an den Messkopf gesendet.

Funktionen zum Abholen von Messergebnissen und Daten

NrpDataAvailable

NrpDataAvailable liefert 1, wenn Daten vom Messkopf eingetroffen sind, und 0, wenn dies nicht der Fall ist und wenn alle internen Puffer leer sind. Wird mit einer Callback-Funktion auf das Vorhandensein von Daten reagiert, so ist die gleichzeitige Verwendung von *NrpDataAvailable* nicht sinnvoll.

Prototyp: `long NrpDataAvailable (void);`

NrpGetData

NrpGetData ermittelt den Typ des abzuholenden Ergebnisses (siehe Tabelle 5-2) sowie die Gruppennummer und die Parameternummer, wenn das Ergebnis ein Parameter ist. Die Gruppennummer ist die laufende Nummer einer Gruppe von zusammen gehörenden Parametern bzw. Befehlen. Von verschiedenen Messkopftypen werden in Abhängigkeit von deren Eigenschaften unterschiedliche Parameter-/Befehlsgruppen implementiert. Die Parameternummer ist die laufende Nummer eines Parameters/Befehls innerhalb seiner Gruppe. Bei Messergebnissen sind Gruppen- und Parameternummer 0.

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

Parameter:

o_pDataType	Zeiger auf die *LONG*-Variable, die den numerischen Wert des Datentyps aufnehmen soll
o_pGroupNr	Zeiger auf die *LONG*-Variable, die die Gruppennummer aufnehmen soll
o_pParamNr	Zeiger auf die *LONG*-Variable, die die Parameternummer aufnehmen soll

Tabelle 5-2 Bedeutung der Datentypen

Datentyp (enum-Konstante)	**num. Wert**	**abzuholende Daten**
DATA_BINARYBLOCK	0	Eine binäre Bytefolge kann mit *NrpGetBinaryResult* abgeholt werden (Kalibrierdatensatz des Messkopfes bei Verwendung des Befehls *CALibration:DATA?*).
DATA_BITFIELDLIMIT	1	Drei Bitfeld-Werte können mit *NrpGetBitfieldResult* abgeholt werden. Der erste Wert ist die Voreinstellung, der zweite Wert ist die Bitmaske zur Darstellung aller zulässigen Zustände, der dritte Wert ist ungenutzt und mit 0 belegt.
DATA_BITFIELDPARAM	2	Drei Bitfeld-Werte zur Darstellung von diskreten Zuständen (z. B. *OFF*, *ON*, *ONCE*) können mit *NrpGetBitfieldResult* abgeholt werden. Der erste Wert ist ein Parameterwert, der zweite und der dritte Wert sind ungenutzt und mit 0 belegt.
DATA_BITFIELDFEATURE	3	Drei Bitfeld-Werte zur Darstellung von implementierten Befehlen können mit *NrpGetBitfieldResult* abgeholt werden. Der erste Wert enthält die Information (für jeden innerhalb einer Befehlsgruppe vorhandenen Befehl oder Parameter ist ein Bit gesetzt, beginnend bei Bit 0 für Parameter-Nr. 1), der zweite und der dritte Wert sind ungenutzt und mit 0 belegt.
DATA_FLOATARRAY	4	Ein Array von *FLOAT*-Werten (Leistungswerte) und ein Array von *LONG*-Werten (Indizes) können mit *NrpGetFloatArray* abgeholt werden.
DATA_FLOATLIMIT	5	Drei *FLOAT*-Werte (Voreinstellung, oberer Grenzwert und unterer Grenzwert) können mit *NrpGetFloatResult* abgeholt werden.

Datentyp (enum-Konstante)	**num. Wert**	**abzuholende Daten**
DATA_FLOATPARAM	6	Drei *FLOAT*-Werte können mit *NrpGetFloatResult* abgeholt werden. Der erste Wert ist ein Parameterwert, der zweite und der dritte Wert sind unbenutzt und mit 0.0 belegt.
DATA_FLOATRESULT	7	Drei *FLOAT*-Werte (Leistungswert und zwei sekundäre Messwerte) können mit *NrpGetFloatResult* abgeholt werden.
DATA_LONGLIMIT	8	Drei *LONG*-Werte (Voreinstellung, oberer Grenzwert und unterer Grenzwert) können mit *NrpGetLongResult* abgeholt werden.
DATA_LONGPARAM	9	Drei *LONG*-Werte können mit *NrpGetLongResult* abgeholt werden. Der erste Wert ist ein Parameterwert, der zweite und der dritte Wert sind ungenutzt und mit 0 belegt.
DATA_STRING	10	Ein String kann mit *NrpGetStringResult* abgeholt werden.

Tabelle 5-3 Bedeutung der Gruppen- und Parameternummern

Gruppen-nummer	**Befehlsgruppe**	**Parameter-nummer**	**Parameter oder Befehl**
1	CALibration	1	CALibration:DATA
		2	CALibration:ZERO:AUTO
		3	CALibration:DATA:LENGth
2	INPut (z. Z. reserviert)		
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

Gruppen-nummer	Befehlsgruppe	Parameter-nummer	Parameter oder Befehl
		13	SENSe:RANGE:AUTO
		14	SENSe:RANGE:CLEVel
		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	1	SENSe:POWer:BURSt:DTOLerance

Gruppen-nummer	Befehlsgruppe	Parameter-nummer	Parameter oder Befehl
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
		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
		14	SENSe:TRACe:MPWidth?
8	SYSTem	1	SYSTem:INFO
		2	SYSTem:INITialize
		3	SYSTem:TRANsaction:BEGin
		4	SYSTem:TRANsaction:END
		5	SYSTem:MINPower

Gruppen-nummer	Befehlsgruppe	Parameter-nummer	Parameter oder Befehl
9	Triggersystem-Befehle	1	ABORt
		2	INITiate:CONTinuous
		3	INITiate:IMMediate
		4	reserviert
		5	reserviert
		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
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

Gruppen-nummer	Befehlsgruppe	Parameter-nummer	Parameter oder Befehl
		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
		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 liest sämtliche Antwort-Datenblöcke, die seit dem letzten Aufruf der Funktion vom Messkopf gesendet wurden, aus dem Binärpuffer aus und kopiert sie in einen bereitgestellten Puffer. Die ausgelesenen Bytes werden aus dem Binärpuffer gelöscht. Die ältesten Antwort-Datenblöcke werden zuerst ausgelesen. Die Funktion liefert als Rückgabewert die Anzahl der tatsächlich aus dem Binärpuffer ausgelesenen Bytes.

Übergibt man der Funktion als Argument *o_pBuf* einen Nullpointer, so liefert die Funktion die Anzahl der im Binärpuffer befindlichen Bytes.

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

Parameter: *o_pBuf* Zeiger auf den Puffer, in dem das Ergebnis hinterlegt wird
i_wCount Puffergröße in Bytes

NrpGetBinaryResultLength

NrpGetBinaryResultLength liefert die Anzahl der im Binärpuffer befindlichen Bytes. Zwar kann auch die Funktion *NrpGetBinaryResult* diese Aufgabe erfüllen, jedoch muss man dafür einen Nullpointer übergeben, was z. B. in Visual-Basic-Projekten Probleme bereitet.

Prototyp: `long NrpGetBinaryResultLength (void);`

NrpGetBitfieldResult

Parameter, die eine bestimmte Anzahl diskreter Zustände (z. B. *OFF*, *ON* und *ONCE*) annehmen können, werden als 32 Bit breite Bitfelder codiert. Jedes Bit repräsentiert einen der möglichen Zustände. Auf diese Weise ist es möglich, dass der Messkopf bei abhängigen Parametern (analog zu Unter- und Obergrenze bei numerischen Parametern) bestimmte diskrete Zustände zulassen oder ausschließen kann.

NrpGetBitfieldResult liest den ältesten der Bitfeld-Blöcke, die seit dem letzten Aufruf der Funktion vom Messkopf gesendet wurden, aus dem Bitfeldpuffer aus. Der ausgelesene Bitfeldblock wird aus dem Bitfeldpuffer gelöscht. Die Bitfelder werden in *LONG*-Variablen übergeben.

Die Funktion liefert als Rückgabewert 1, wenn das Auslesen erfolgreich war, und 0, wenn beim Auslesen ein Fehler aufgetreten ist.

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

Parameter: *o_pdwR1* Zeiger auf die erste der drei *LONG*-Variablen, die das Ergebnis aufnehmen sollen
o_pdwR2 Zeiger auf die zweite der drei *LONG*-Variablen, die das Ergebnis aufnehmen sollen
o_pdwR3 Zeiger auf die dritte der drei *LONG*-Variablen, die das Ergebnis aufnehmen sollen

NrpGetFloatArray

Befindet sich der Messkopf im sogenannten *buffered mode, so* werden Messergebnisse nicht einzeln, sondern blockweise übertragen. Dadurch lässt sich eine besonders hohe Übertragungsgeschwindigkeit erreichen. Auch in den Modi *Timeslot* und *Scope* liefert der Messkopf seine Ergebnisse in Form von *FLOAT*-Array-Blöcken. Jedem der *FLOAT*-Werte ist innerhalb des Arrays ein fester Index zugeordnet. *NrpGetFloatArray* kopiert eine bestimmte Anzahl von *FLOAT*-Werten und deren Indizes, beginnend mit den ältesten Werten, aus dem DLL-internen Float-Array-Puffer in ein *FLOAT*- bzw. *LONG*-Array. Die

kopierten Werte werden aus dem Float-Array-Puffer gelöscht. Als Rückgabewert liefert die Funktion die Anzahl der tatsächlich kopierten *FLOAT*-Werte bzw. Indizes.

Übergibt man der Funktion als Argument *o_pFarr* einen Nullpointer, so liefert die Funktion die Anzahl der im Float-Array-Puffer befindlichen *FLOAT*- und Index-Werte.

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

Parameter: *o_pFarr* Zeiger auf das *FLOAT*-Array, das die *FLOAT*-Werte aufnehmen soll
o_pIarr Zeiger auf das *LONG*-Array, das die Indizes der *FLOAT*-Werte aufnehmen soll
i_wCount Länge der Arrays

NrpGetFloatArrayLength

NrpGetFloatArrayLength liefert die Anzahl der im Float-Array-Puffer befindlichen *FLOAT*- und Index-Werte. Zwar kann auch die Funktion *NrpGetFloatArray* diese Aufgabe erfüllen, jedoch muss man dafür einen Nullpointer übergeben, was z. B. in Visual-Basic-Projekten Probleme bereitet.

Prototyp: `long NrpGetFloatArrayLength (void);`

NrpGetFloatResult

NrpGetFloatResult kopiert die drei *FLOAT*-Werte, die in einem *FLOAT*-Block enthalten sind, in *FLOAT*-Variable. Handelt es sich bei dem *FLOAT*-Block um ein Messergebnis, so ist der erste *FLOAT*-Wert der Leistungswert, der zweite und der dritte *FLOAT*-Wert sind sekundäre Messwerte, z. B. Rauschen und Messunsicherheit. Die Funktion liefert als Rückgabewert 1, wenn das Auslesen erfolgreich war, und 0, wenn beim Auslesen ein Fehler aufgetreten ist.

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

Parameter: *o_pfR1* Zeiger auf die *FLOAT*-Variable, die den ersten *FLOAT*-Wert aufnehmen soll
o_pfR2 Zeiger auf die *FLOAT*-Variable, die den zweiten *FLOAT*-Wert aufnehmen soll
o_pfR3 Zeiger auf die *FLOAT*-Variable, die den dritten *FLOAT*-Wert aufnehmen soll

NrpGetLongResult

NrpGetLongResult kopiert die drei *LONG*-Werte, die in einem *LONG*-Block enthalten sind, in *LONG*-Variable. Die Funktion liefert als Rückgabewert 1, wenn das Auslesen erfolgreich war, und 0, wenn beim Auslesen ein Fehler aufgetreten ist.

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

Parameter: *o_pwR1* Zeiger auf die *LONG*-Variable, die den ersten *LONG*-Wert aufnehmen soll
o_pwR2 Zeiger auf die *LONG*-Variable, die den zweiten *LONG*-Wert aufnehmen soll
o_pwR3 Zeiger auf die *LONG*-Variable, die den dritten *LONG*-Wert aufnehmen soll

NrpGetStringResult

NrpGetStringResult erlaubt den Zugriff auf die bis zum Aufruf dieser Funktion eingetroffenen String-Antworten. Eine Stringantwort wird in der Regel in mehrere String-Blöcke aufgeteilt empfangen, in der DLL zusammengesetzt und im internen String-Puffer gespeichert. Die Funktion kopiert eine bestimmte Anzahl von Zeichen aus diesem internen String-Puffer in einen bereitgestellten Puffer. Die kopierten Zeichen werden aus dem internen String-Puffer gelöscht. Als Rückgabewert liefert die Funktion die Anzahl der tatsächlich gelesenen Zeichen.

Übergibt man der Funktion als Argument *o_pBuf* einen Nullpointer, so liefert die Funktion die Anzahl der im Stringpuffer befindlichen Zeichen. Dabei wird das terminierende Nullzeichen nicht mitgezählt. Soll das terminierende Nullzeichen mit ausgelesen werden, so muss folglich ein um 1 größerer Puffer übergeben werden.

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

Parameter:	*o_pBuf*	Zeiger auf den Puffer, in welchem die Antwort hinterlegt werden soll
	i_wCount	Puffergröße in Bytes

Funktionen zur Status- und Fehlererkennung

NrpEmptyAllBuffers

NrpEmptyAllBuffers löscht sämtliche DLL-internen dynamischen Puffer.

Prototyp: `void NrpEmptyAllBuffers (void);`

NrpEmptyErrorQueue

NrpEmptyErrorQueue löscht alle in der Fehler-Queue befindlichen Fehler.

Prototyp: `void NrpEmptyErrorQueue (void);`

NrpGetLastError

NrpGetLastError liest den am weitesten zurückliegenden Fehler aus der Fehler-Queue aus. Um die Fehler-Queue zu leeren, müssen entweder nacheinander alle Fehler mit *NrpGetLastError* ausgelesen werden, oder einer der Funktionen *NrpEmptyErrorQueue* oder *NrpEmptyAllBuffers* muss aufgerufen werden. Eine Zusammenstellung aller möglichen Fehlerzustände bietet Tabelle 5-4.

Prototyp: `long NrpGetLastError (void);`

Tabelle 5-4 Mögliche Messkopf-Fehlerzustände

Fehler (enum-Konstante)	**num. Wert**	**Bedeutung**
NRPERROR_NOERROR	0	Dieser Wert wird von der Funktion *NrpGetLastError* zurückgegeben, wenn kein Fehler aufgetreten ist.
NRPERROR_CALDATA_FORMAT	1	Das Format des an den Messkopf zu übertragenden Kalibrierdatensatzes entspricht nicht der Spezifikation. Entweder wird der vorhandene Kalibrierdatensatz nicht als solcher erkannt, oder er liegt in einer nicht unterstützten Version vor.
NRPERROR_OVERRANGE	2	Die Leistung des Messsignals übersteigt den Messbereich des manuell gewählten Messkanals (*bei SENSe:RANGe:AUTO OFF*).
NRPERROR_NOTINSERVICEMODE	3	Der an den Messkopf geschickte Befehl ist nur im Service-Modus verfügbar, der Messkopf befindet sich aber nicht im Service-Modus.
NRPERROR_CALZERO	4	Der Nullabgleich konnte nicht durchgeführt werden, weil die anliegende HF-Leistung zu hoch ist.
NRPERROR_TRIGGERQUEUEFULL	5	Im Modus *Burst Average*: Der Burst ist breiter als zulässig, daher wurde die Messung nach Ablauf der maximal zulässigen Burst-Dauer abgebrochen. Das so erhaltene Ergebnis kann vom korrekten Messwert abweichen.
NRPERROR_EVENTQUEUEFULL	6	Die interne Queue für Trigger-Ereignisse ist übergelaufen. Diese Fehlermeldung deutet auf einen Software-Fehler hin, da der Messkopf normalerweise bei einem drohenden Überlauf dieser Queue keine weiteren Triggerereignisse annimmt.

Fehler (enum-Konstante)	**num. Wert**	**Bedeutung**
NRPERROR_SAMPLEERROR	7	Ein oder mehrere Samples konnten nicht verarbeitet werden, weil der Messkopf z. B. während der Abarbeitung eines Bus-Triggers nicht auf den Sample-Interrupt reagieren konnte. Normalerweise führt dies nicht zu einer Verfälschung des Messergebnisses, außer im Realtime-Modus *(SENSe:TRACe:REALtime ON)*.
NRPERROR_OVERLOAD	8	Die Leistung des Messsignals übersteigt die obere Messgrenze des Messkopfes. Es besteht die Gefahr einer irreversiblen Beschädigung des Messkopfes.
NRPERROR_HARDWARE	9	Ein Hardware-Fehler ist aufgetreten. Fehler am Temperatursensor führen während des Betriebes sofort zu dieser Fehlermeldung. Wurde bei einem der Testbefehle **TST?* oder *TEST:SENSor?* ein Fehler festgestellt, so führt dies so lange zu dieser Fehlermeldung, bis ein erneuter Selbsttest keinen Fehler mehr feststellt. Fehler bei den Betriebsspannungen werden außerdem separat gemeldet (siehe *NRPERROR_VOLTAGE*).
NRPERROR_CHECKSUM	10	Die Überprüfung der Prüfsumme des zu ladenden Kalibrierdatensatzes ergab einen Fehler. Der Kalibrierdatensatz wurde nicht geladen.
NRPERROR_ILLEGALSERIAL	11	Es wurde versucht, einen Kalibrierdatensatz in den Messkopf zu laden, dessen Seriennummer nicht mit der Seriennummer des Messkopfes übereinstimmt. Der Kalibrierdatensatz wurde nicht geladen.
NRPERROR_FILTERTRUNCATED	12	Im Auto-Averaging-Modus: Die Messung wurde nach Ablauf der maximalen Messzeit (Parameter *SENSe:AVERage:COUNt:AUTO :MTIMe* oder *SENSe:TRACe:AVERage:COUNt:AUTO:MTIMe*) abgebrochen, ohne dass die vorgegebene Auflösung oder das vorgegebene Signal-Rausch-Verhältnis erreicht wurden.
NRPERROR_BURST_TOO_SHORT	13	Im Modus *Burst Average* war ein Burst zu kurz, um korrekt erkannt zu werden.
NRPERROR_GENERIC	128	Es ist ein nicht näher spezifizierter Fehler aufgetreten.
NRPERROR_OVERMAX	129	Der numerische Parameter ist größer als die zulässige Obergrenze.
NRPERROR_UNDERMIN	130	Der numerische Parameter ist kleiner als die zulässige Untergrenze.
NRPERROR_VOLTAGE	131	Mindestens eine der internen Versorgungsspannungen fehlt oder liegt außerhalb des zulässigen Bereiches.
NRPERROR_SYNTAX	132	Die Syntax des an den Messkopf geschickten Befehls ist fehlerhaft.
NRPERROR_MEMORY	133	Der zur Verfügung stehende Speicher reicht nicht aus. Diese Fehlermeldung deutet auf einen Firmware-Fehler hin.
NRPERROR_PARAMETER	134	Dieser Parameter ist unzulässig.
NRPERROR_TIMING	135	reserviert
NRPERROR_NOTIDLE	136	Der an den Messkopf geschickte Befehl steht während einer laufenden Messung nicht zur Verfügung.
NRPERROR_UNKNOWNCOMMAND	137	Der an den Messkopf geschickte Befehl konnte nicht interpretiert werden.
NRPERROR_OUTBUFFERFULL	138	Der Sendepuffer des Messkopfes ist übergelaufen, weil die vom Messkopf ausgegebenen Daten nicht abgeholt wurden.

Fehler (enum-Konstante)	num. Wert	Bedeutung
NRPERROR_FLASHPROG	139	Es ist ein Fehler beim Programmieren des Flash-Speichers aufgetreten.
NRPERROR_CALDATANOTPRESENT	140	Es ist kein gültiger Kalibrierdatensatz vorhanden.

NrpGetErrorText

NrpGetErrorText dient zum Abrufen der zum Fehlercode vom Typ *NRPERROR* passenden Fehlermeldung im Klartext. Diese wird als nicht nullterminierter String in einem bereitgestellten Puffer abgelegt. Als Rückgabewert liefert die Funktion die Anzahl der im Puffer abgelegten Zeichen.

Übergibt man der Funktion als Argument *o_pBuf* einen Nullpointer, so liefert die Funktion die Anzahl der im Stringpuffer befindlichen Zeichen. Dabei wird das terminierende Nullzeichen nicht mitgezählt. Soll das terminierende Nullzeichen mit ausgelesen werden, so muss folglich ein um 1 größerer Puffer übergeben werden.

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

Parameter:

i_wErr	Fehlercode, der beispielsweise von *NrpGetLastError* geliefert wurde
o_pBuf	Zeiger auf den Puffer, in dem der Fehlertext abgelegt werden soll
i_wCount	Puffergröße in Bytes

NrpIsAlive

NrpIsAlive liefert als Rückgabewert

- 0, wenn der als aktiv gewählte Messkopf nicht betriebsbereit ist,
- 1, wenn er betriebsbereit und im Messmodus ist,
- 2, wenn der Bootloader aktiv ist.

Prototyp: `long NrpIsAlive (void);`

NrpGetTriggerState

NrpGetTriggerState liefert als Rückgabewert den numerischen Wert des aktuellen Zustandes des Triggersystems (Tabelle 5-5). Um eine kurze Klartextbeschreibung des Triggerzustandes zu erhalten, kann die Funktion *NrpGetTriggerStateText* benutzt werden.

Prototyp: `long NrpGetTriggerState (void);`

Tabelle 5-5 Mögliche Messkopf-Triggerzustände

Triggerzustand (enum-Konstante)	num. Wert	Beschreibung
SENSORHW_STATE_IDLE	0	Es läuft keine Messung, der Messkopf ist bereit.
SENSORHW_STATE_WAIT_FOR_ARM	1	Reserviert. Der R&S NRP-Zxx hat keinen ARM-Layer.
SENSORHW_STATE_WAIT_FOR_TRIGGER	2	Eine Messung wurde gestartet, der Messkopf wartet auf ein Trigger-Ereignis.

Triggerzustand (enum-Konstante)	num. Wert	Beschreibung
SENSORHW_STATE_MEASURING	3	Eine Messung läuft.

NrpGetTriggerStateText

NrpGetTriggerStateText liefert eine kurze Klartextbeschreibung des z. B. mit *NrpGetTriggerState* ermittelten Triggerzustandes.

Übergibt man der Funktion als Argument *o_pBuf* einen Nullpointer, so liefert die Funktion die Anzahl der Zeichen im beschreibenden String. Dabei wird das terminierende Nullzeichen nicht mitgezählt. Soll das terminierende Nullzeichen mit ausgelesen werden, so muss folglich ein um 1 größerer Puffer übergeben werden.

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

Callback-Funktionen

Um die verzögerungsfreie und asynchrone Auswertung bestimmter Ereignisse durch die Anwendung zu ermöglichen, können Callback-Funktionen festgelegt werden, die beim Auftreten des jeweiligen Ereignisses durch die DLL aufgerufen werden. Die Ereignisse, auf die reagiert werden kann, umfassen:

- *CommandAccepted*: Ein zum Messkopf gesendeter Befehl wurde von diesem quittiert.
- *DataAvailable*: Daten vom Messkopf sind eingetroffen.
- *DeviceChanged*: Ein Messkopf wurde an den Host-Rechner angeschlossen oder vom Host-Rechner getrennt.
- *ErrorOccurred*: Der Messkopf hat einen Fehler gemeldet.
- *StateChanged*: Der Triggerstatus des Messkopfes hat sich geändert.

Um Callback-Funktionen zu definieren, dienen folgende Funktionen:
- *NrpSetNotifyCallbackCommandAccepted*
- *NrpSetNotifyCallbackDataAvailable*
- *NrpSetNotifyCallbackDeviceChanged*
- *NrpSetNotifyCallbackErrorOccurred*
- *NrpSetNotifyStateChanged*

Diese Funktionen sind im Abschnitt „Funktionen zum Setzen von Callback-Funktionen“ auf Seite 5.21 näher beschrieben. Der Typ *function* ist dabei wie folgt definiert:
```
typedef void (__stdcall *function) (void *arg1, void *arg2);
```

Die zu übergebende Callback-Funktion muss die folgende Form haben:
```
void __stdcall MyCallbackFunction (void *arg1, void *arg2)
{
  …
}
```

Ist die Callback-Funktion eine Klassenmethode, so muss sie als *static* vereinbart werden.

Im Falle der Ereignisse *CommandAccepted*, *DataAvailable*, *ErrorOccured* und *StateChanged* wird *arg1* mit *NULL* belegt und braucht nicht weiter ausgewertet zu werden. Bei *DeviceChanged* wird dagegen ein Zeiger auf eine Struktur des Typs *USB_DEVICECHANGED_HANDLER_RESULT* (siehe Beschreibung der Funktion *NrpSetNotifyCallbackDeviceChanged* auf Seite 5.21) übergeben, die für eine differenziertere Ereignisbehandlung ausgewertet werden kann.

Das Argument *arg2* dient zum Übergeben einer Kontextinformation (z. B. Referenz auf eine Klasse, wenn die Callback-Funktion eine Methode dieser Klasse aufrufen soll). Es wird beim Aufruf der *SetNotifyCallback…*-Funktion angegeben.

Folgendes Beispiel zeigt das Beispiel einer Callbackfunktion für das *DeviceChanged*-Ereignis, welche in Visual C++ als Klassenmethode implementiert ist. Es handelt sich dabei nicht um ein selbständig lauffähiges Programm. Das betrifft insbesondere die nicht dargestellte Klassenmethode *AddToProtocol* zur Ausgabe eines Strings z. B. in einem mehrzeiligen Edit-Control.

Definition im Headerfile:

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

Anmelden im Initialisierungsteil der Klasse:

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

Implementierung:

```
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);
}
```

Funktionen zum Setzen von Callback-Funktionen

NrpSetNotifyCallbackCommandAccepted

NrpSetNotifyCallbackCommandAccepted legt eine nutzerspezifische Funktion fest, die aufgerufen wird, wenn ein zum Messkopf gesendeter Befehl von diesem erfolgreich abgearbeitet und quittiert wurde (siehe Abschnitt „Callback-Funktionen“ auf Seite 5.19).

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

Parameter:	*f*	aufzurufende Funktion
	arg	Argument zum Übergeben einer Kontextinformation

NrpSetNotifyCallbackDataAvailable

NrpSetNotifyCallbackDataAvailable legt eine nutzerspezifische Funktion fest, die aufgerufen wird, wenn Messwerte oder andere Daten vom Messkopf zur Abholung vorliegen (siehe Abschnitt „Callback-Funktionen“ auf Seite 5.19).

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

Parameter:	*f*	aufzurufende Funktion
	arg	Argument zum Übergeben einer Kontextinformation

NrpSetNotifyCallbackDeviceChanged

NrpSetNotifyCallbackDeviceChanged legt eine nutzerspezifische Funktion fest, die aufgerufen wird, wenn ein Messkopf an den Host-Rechner angeschlossen oder vom Host-Rechner getrennt wurde (siehe Abschnitt „Callback-Funktionen“ auf Seite 5.19).

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

Parameter:	*f*	aufzurufende Funktion
	arg	Argument zum Übergeben einer Kontextinformation

NrpSetNotifyCallbackErrorOccured

NrpSetNotifyCallbackErrorOccured legt eine nutzerspezifische Funktion fest, die aufgerufen wird, wenn ein Fehler aufgetreten ist (siehe Abschnitt „Callback-Funktionen“ auf Seite 5.19).

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

Parameter:	*f*	aufzurufende Funktion
	arg	Argument zum Übergeben einer Kontextinformation

NrpSetNotifyCallbackStateChanged

NrpSetNotifyCallbackStateChanged legt eine nutzerspezifische Funktion fest, die aufgerufen wird, wenn sich der Triggerzustand des Messkopfes ändert (siehe Abschnitt „Callback-Funktionen“ auf Seite 5.19).

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

Parameter:	*f*	aufzurufende Funktion
	arg	Argument zum Übergeben einer Kontextinformation

Windows-Messages

Die DLL versendet und empfängt Windows™-Messages. Sie ist also darauf angewiesen, dass das aufrufende Programm diese weiterleitet. In der Regel ist das aufrufende Programm eine GUI-Applikation, die auf dem Windows API aufsetzt, womit diese Bedingung immer erfüllt ist. Handelt es sich jedoch um eine einfache Konsolenapplikation, so werden von dieser automatisch keine Messages weitergeleitet. Um eine Message loop in einer Konsolenapplikation nachbauen zu können, bietet die DLL die Funktion *NrpMessageLoopBody* an. Sie muss in der Hauptschleife der Konsolenapplikationen zyklisch aufgerufen werden, um die von der DLL ein- und ausgehenden Windows™-Messages weiterzuleiten.

Prototyp: `void NrpMessageLoopBody (void);`

USB-spezifische Befehle

Folgende Kommandos betreffen nur die USB-Schnittstelle und haben keinerlei Bezug zur Funktionalität des Messkopfes:

NrpOpenDriver

NrpOpenDriver stellt die Verbindung zum USB-Device-Treiber her.

Prototyp: `void NrpOpenDriver (void);`

NrpCloseDriver

NrpCloseDriver beendet die Verbindung zum USB-Device-Treiber.

Prototyp: `void NrpCloseDriver (void);`

Hinweis: Beim Schließen der DLL wird eine bestehende Verbindung zum USB-Device-Treiber automatisch abgebaut.

NrpClearDevice

NrpClearDevice sendet den *Vendor Request SET_DEVICE_CLEAR* an den Messkopf. Dieser *Vendor Request* löscht die USB-FIFOs und initialisiert das Triggersystem.

Prototyp: `void NrpClearDevice (void);`

NrpGetDeviceChangedDevName

Die Callback-Funktion, die durch das *DeviceChanged*-Ereignis aufgerufen wird, liefert als Argument die Struktur *USB_DEVICECHANGED_HANDLER_RESULT* (siehe Tabelle 8). Eines der Strukturmitglieder ist der Zeiger *pDeviceName* auf den Namen des Gerätes, das dieses Ereignis ausgelöst hat. Da in bestimmten Programmierumgebungen, wie z. B. Visual Basic, Strings nicht direkt als Zeiger übergeben werden können, wird als Alternative die Funktion *NrpGetDeviceChangedDevName* angeboten, die den Gerätenamen in einen Puffer kopiert. Die Funktion liefert als Rückgabewert die Anzahl der tatsächlich kopierten Zeichen.

Übergibt man der Funktion als Argument *o_pBuf* einen Nullpointer, so liefert die Funktion die Anzahl der Zeichen im Device-Name-String. Dabei wird das terminierende Nullzeichen nicht mitgezählt. Soll das terminierende Nullzeichen mit ausgelesen werden, so muss folglich ein um 1 größerer Puffer übergeben werden.

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

Parameter: *o_pBuf* Zeiger auf den Puffer, der den Gerätenamen aufnehmen soll
i_wCount Puffergröße in Bytes

Tabelle 5-6 Beschreibung der Struktur *USB_DEVICECHANGED_HANDLER_RESULT*

Typ	Name	Beschreibung
long	wResult	Der Wert ist 1, wenn dem Ereignis ein Gerät zugeordnet werden konnte, und 0, wenn dies nicht der Fall ist (dann ist der String, auf den pDeviceName zeigt, ein Leerstring).
long	wEventType	Kann die Werte *SENSORHW_STATE_DEVICEARRIVAL* bis *SENSORHW_STATE_DEVICEUNKNOWN* einnehmen.
long	wEventTextHandle	Text-Handle, über das die Funktion *NrpGetDeviceChangedMsgText* die Klartextbeschreibung des Ereignisses ermittelt.
char*	pEventText	Zeiger auf die Klartextbeschreibung des Ereignisses. Dieser Zeiger ist nur in einer C- oder C++-Umgebung zu verwenden. Für Visual Basic müssen die Funktionen *NrpGetDeviceChangedMsgText* und *NrpGetDeviceChangedMsgTextLength* verwendet werden.
char*	pDeviceName	Der Name des Gerätes, das das Ereignis ausgelöst hat. Dieser Zeiger ist nur in einer C- oder C++-Umgebung zu verwenden. Für Visual Basic müssen die Funktionen *NrpGetDeviceChangedDevName* und *NrpGetDeviceChangedDevNameLength* verwendet werden.

NrpGetDeviceChangedMsgText

Die Callback-Funktion, die durch das *DeviceChanged*-Ereignis aufgerufen wird, liefert als Argument die Struktur *USB_DEVICECHANGED_HANDLER_RESULT* (siehe Tabelle 5-6). Eines der Strukturmitglieder ist *wEventTextHandle*, ein *LONG*-Wert, hinter dem sich ein DLL-interner String-Listen-Index verbirgt. Mit *NrpGetDeviceChangedMsgText* lässt sich die in der String-Liste abgelegte Ereignismeldung in einen Puffer kopieren. Die Funktion liefert als Rückgabewert die Anzahl der tatsächlich kopierten Zeichen.

Übergibt man der Funktion als Argument *o_pBuf* einen Nullpointer, so liefert die Funktion die Anzahl der Zeichen in der Ereignismeldung. Dabei wird das terminierende Nullzeichen nicht mitgezählt. Soll das terminierende Nullzeichen mit ausgelesen werden, so muss folglich ein um 1 größerer Puffer übergeben werden.

Prototyp:

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

Parameter: *wEventTextHandle* String-Listen-Index
o_pBuf Zeiger auf den Puffer, der die Ereignismeldung aufnehmen soll
i_wCount Puffergröße in Bytes

NrpGetDeviceID

NrpGetDeviceID liefert die laufende Nummer des zur Zeit als aktiv konfigurierten Messkopfes. Jeder angeschlossene Messkopf erhält eine solche laufende Nummer. Die Zählung beginnt mit 0.

Prototyp: `long NrpGetDeviceID (void);`

NrpGetIDByLogicalName

Jeder Messkopf hat eine eindeutige Kombination aus Typenbezeichnung und Seriennummer, welcher beim ersten Anschluss an den USB automatisch ein logischer Name zugeordnet wird (Sensor1, Sensor2 usw.). Diese Zuordnungen sind in der Registry unter *HKEY_CURRENT_USER\Software\Rohde&Schwarz\NrpDll\Sensors* eingetragen. Sie können mit Hilfe der Funktion *NrpChannelAssignment* geändert werden. *NrpGetIDByLogicalName* liefert die laufende Nummer eines angeschlossenen Messkopfes, wenn der übergebene logische Name in der Registry gefunden wurde. Wurde der logische Name nicht in der Registry gefunden oder ist der betreffende Messkopf nicht angeschlossen, dann liefert die Funktion –1.

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

Parameter: *i_pLogicalName* logischer Name des Messkopfes als nullterminierter String

NrpGetLogicalName

Jeder Messkopf hat eine eindeutige Kombination aus Typenbezeichnung und Seriennummer, welcher beim ersten Anschluss an den USB automatisch ein logischer Name zugeordnet wird (Sensor1, Sensor2 usw.). Diese Zuordnungen sind in der Registry unter *HKEY_CURRENT_USER\Software\Rohde&Schwarz\NrpDll\Sensors* eingetragen. Sie können mit Hilfe der Funktion *NrpChannelAssignment* geändert werden. *NrpGetLogicalName* kopiert den logischen Namen des angeschlossenen Messkopfes mit der als Argument übergebenen laufenden Nummer in einen bereitgestellten Puffer. Als Rückgabewert liefert die Funktion die Anzahl der tatsächlich kopierten Zeichen.

Übergibt man der Funktion als Argument *o_pBuf* einen Nullpointer, so liefert die Funktion die Anzahl der Zeichen im logischen Namen. Dabei wird das terminierende Nullzeichen nicht mitgezählt. Soll das terminierende Nullzeichen mit ausgelesen werden, so muss folglich ein um 1 größerer Puffer übergeben werden.

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

Parameter:

	i_wID	laufende Nummer des Messkopfes
	o_pBuf	Zeiger auf den Puffer, der den logischen Namen aufnehmen soll
	i_wCount	Puffergröße in Bytes

NrpGetManufacturerCode

NrpGetManufacturerCode kopiert die Herstellerbezeichnung des angeschlossenen Messkopfes mit der als Argument übergebenen laufenden Nummer in einen bereitgestellten Puffer. Als Rückgabewert liefert die Funktion die Anzahl der tatsächlich kopierten Zeichen.

Übergibt man der Funktion als Argument *o_pBuf* einen Nullpointer, so liefert die Funktion die Anzahl der Zeichen im logischen Namen. Dabei wird das terminierende Nullzeichen nicht mitgezählt. Soll das terminierende Nullzeichen mit ausgelesen werden, so muss folglich ein um 1 größerer Puffer übergeben werden.

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

Parameter:	*i_wID*	laufende Nummer des Messkopfes
	o_pBuf	Zeiger auf den Puffer, der die Herstellerbezeichnung aufnehmen soll
	i_wCount	Puffergröße in Bytes

NrpGetNrOfDevices

NrpGetNrOfDevices liefert als Rückgabewert die Anzahl der an den USB angeschlossenen Messköpfe.

Prototyp: `long NrpGetNrOfDevices (void);`

NrpGetProductID

NrpGetProductID kopiert die Typenbezeichnung des angeschlossenen Messkopfes mit der als Argument übergebenen laufenden Nummer in einen bereitgestellten Puffer. Als Rückgabewert liefert die Funktion die Anzahl der tatsächlich kopierten Zeichen.

Übergibt man der Funktion als Argument *o_pBuf* einen Nullpointer, so liefert die Funktion die Anzahl der Zeichen in der Typenbezeichnung. Dabei wird das terminierende Nullzeichen nicht mitgezählt. Soll das terminierende Nullzeichen mit ausgelesen werden, so muss folglich ein um 1 größerer Puffer übergeben werden.

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

Parameter:	*i_wID*	laufende Nummer des Messkopfes
	o_pBuf	Zeiger auf den Puffer, der die Typenbezeichnung aufnehmen soll
	i_wCount	Puffergröße in Bytes

NrpGetSerialNumber

NrpGetSerialNumber kopiert den Seriennummernstring des angeschlossenen Messkopfes mit der als Argument übergebenen laufenden Nummer in einen bereitgestellten Puffer. Als Rückgabewert liefert die Funktion die Anzahl der tatsächlich kopierten Zeichen.

Übergibt man der Funktion als Argument *o_pBuf* einen Nullpointer, so liefert die Funktion die Anzahl der Zeichen im Seriennummernstring. Dabei wird das terminierende Nullzeichen nicht mitgezählt. Soll das terminierende Nullzeichen mit ausgelesen werden, so muss folglich ein um 1 größerer Puffer übergeben werden.

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

Parameter:	*i_wID*	laufende Nummer des Messkopfes
	o_pBuf	Zeiger auf den Puffer, der den Seriennummernstring aufnehmen soll
	i_wCount	Puffergröße in Bytes

NrpSDRGetDescriptor

NrpSDRGetDescriptor sendet den *Standard Device Request GetDescriptor* an den Messkopf. Dieser *Standard Device Request* dient zum Auslesen von Device-, Configuration- und String-Deskriptoren. Interface- und Endpoint-Deskriptoren können nicht direkt, sondern nur zusammen mit dem Configuration Descriptor ausgelesen werden. Als Rückgabewert liefert die Funktion den Fehlerstatus nach Tabelle 5-4. Der Aufbau der Deskriptoren ist in der USB-Fachliteratur detailliert beschrieben.

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

Parameter:	*o_pBuffer*	Zeiger auf den Puffer, der die Deskriptordaten aufnehmen soll
	io_wByteCount	bei Aufruf: Puffergröße in Bytes nach Aufruf: Anzahl der tatsächlich im Puffer abgelegten Bytes
	i_wRecipient	0 = Device, 1 = Interface, 2 = Endpoint, 3 = Andere
	i_wDescrType	Typ des Deskriptors (1 = Device Descriptor, 2 = Configuration Descriptor inkl. zugehörige Interface-, Klassen- und Endpoint-Deskriptoren, 3 = String Descriptor)
	i_wDescrIndex	nur bei String Descriptor: Index des Strings für Herstellerbezeichnung, Produktbezeichnung oder Seriennummer
	i_wLangID	nur bei String Descriptor: Sprachenindex (optional, 0x0409 = Englisch), sonst 0

NrpSelectDevice

NrpSelectDevice selektiert den angeschlossen Messkopf mit der als Argument übergebenen laufenden Nummer.

Prototyp: `void NrpSelectDevice (long i_wID);`

Parameter:	*i_wID*	laufende Nummer des Messkopfes

Anwendungsbeispiele

Die folgenden Beispiele demonstrieren die Funktionalität des Messkopfs. Zum Kennenlernen der Befehle eignet sich das Programmmodul **USB-Terminal** des NRP-Toolkits. Sofort, nachdem der Messkopf an den USB-Anschluss des PC angeschlossen wurde, ist er einsatzbereit.

Befehle müssen dem Messkopf mit Hilfe der Funktion *NrpSendCommand* übermittelt werden. Der Übersichtlichkeit halber werden im Folgenden nur die SCPI-Befehlsfolgen dargestellt. Es wird generell die Kurzform der Befehle verwendet. Für nähere Erläuterungen zu den einzelnen Befehlen siehe Befehlsreferenz in Abschnitt 6 dieses Benutzerhandbuchs.

Vor der Messung sollte mit dem Messkopf bei abgeschalteter Leistung ein Nullabgleich durchgeführt werden.

```
cal:zero:auto once      oder   cal:zero:auto on
```

Zusätzlich muss bei jeder Messung die HF-Trägerfrequenz (im folgenden Beispiel 500 MHz) dem Sensor mitgeteilt werden.

```
sens:freq 500e6
```

Einfache Leistungsmittelwertmessung

Um den Leistungsmittelwert zu messen, sind im einfachsten Fall nur folgende Befehle notwendig:

`*rst`	setzt den Messkopf auf die Voreinstellungen zurück
`init:imm`	startet die Messung

In allen Messmodi steht ein Mittelungsfilter zur Verfügung. Dieses Filter kann entweder mit einem manuell eingestellten oder einem vom Messkopf automatisch ermittelten Averaging-Faktor betrieben werden. Bei Nutzung der Voreinstellung ermittelt der Messkopf den Averaging-Faktor automatisch. Zur Ermittlung des Averaging-Faktors kann zwischen zwei Algorithmen gewählt werden: Beim klassischen Verfahren wird die gewünschte Auflösung vorgegeben, und der Messkopf ermittelt die Filterlänge anhand der anliegenden Leistung, wobei eine Obergrenze für die Messzeit (Parameter *SENSe:AVERage:COUNt:AUTO:MTIMe*) eingehalten wird. Mit dem neuen „Fixed Noise“-Verfahren (*SENSe:AVERage:COUNt:AUTO:TYPE NSRatio*) wird dagegen ein genau definierter Rauschanteil im Messergebnis eingehalten.

Zwei Verhaltensweisen des Mittelungsfilters werden unterschieden. Der Parameter *SENSe:AVERage:TCONtrol* kann die Werte *MOVing* und *REPeat* annehmen.

- Im Fernsteuerbetrieb ist in der Regel die Einstellung *REPeat* vorteilhaft. Hier wird der Mittelwert erst ausgegeben, wenn das Filter vollständig gefüllt ist. Im Modus *Continuous Average* wird zum Füllen des Mittelungsfilters nur ein Triggerereignis zum Start der Messung benötigt, die restlichen erforderlichen Messungen werden automatisch durchgeführt. In den Modi *Burst Average*, *Timeslot* und *Scope* ist wegen des Zeitbezugs ein Triggerereignis pro Messwert notwendig.
- Bei der Einstellung *MOVing* wird jedes Mal, wenn ein Messwert in das Filter geschoben wird, der Mittelwert gebildet und ausgegeben. Dieses Filterverhalten lässt es zu, frühzeitig eine Tendenz im angezeigten Mittelwert zu erkennen. Es kommt auch bei der Handbedienung über das Grundgerät zum Einsatz.

Beispiel für eine Konfiguration mit manuellem Averaging und *REPeating*-Verhalten:

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

Beispiel für eine Konfiguration mit Auto-Averaging mit „Fixed Noise“-Verfahren (Rauschanteil: 0,01 dB, Obergrenze für die Messzeit: 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
```

Die Einstellungen des Mittelungsfilters sind für die Modi *Continuous Average*, *Burst Average* und *Timeslot* gleich und werden beim Moduswechsel übernommen. Für den Modus *Scope* wird das Filter gesondert konfiguriert, die Befehle beginnen mit *SENSe:TRACe:AVERage: …*

Schnelle und mehrfache Leistungsmittelwertmessung

Für schnelle aufeinanderfolgende Messungen stellt der Messkopf einen Messwertpuffer mit einstellbarer Größe zur Verfügung. Erst wenn dieser Puffer komplett mit Messwerten gefüllt ist, wird sein Inhalt zum Steuergerät (Grundgerät/PC) übertragen. Durch die Übertragung in *FLOAT*-Array-Blöcken wird eine besonders zeitsparende Übertragung möglich. Folgendes Beispiel zeigt die Einstellungen für den schnellstmöglichen Betrieb:

`*rst`	setzt den Messkopf auf die Voreinstellungen zurück
`sens:freq 500e6`	HF-Trägerfrequenz einstellen (hier 500 MHz)
`sens:aver:stat off`	schaltet das Mittelungsfilter aus
`sens:pow:avg:aper 100e-6`	kürzestes Messfenster einstellen (hier 100 µs)
`sens:pow:avg:buff:size 100`	Messwertpuffer soll 100 Messwerte aufnehmen
`sens:pow:avg:buff:stat on`	Messwertepufferung einschalten
`trig:sour imm`	freilaufender Trigger
`trig:coun 100`	100 Messwerte aufnehmen
`init:imm`	Messung starten

Der Befehl `trig:count 100` sorgt dafür, dass nach einem einmaligen Messstart mit `init:imm` 100 Messungen durchgeführt werden, um den bereitgestellten Puffer zu füllen. Würde dagegen hier die Voreinstellung `trig:count 1` verwendet, müsste für dieselbe Anzahl von Messergebnissen 100 Mal nacheinander `init:imm` aufgerufen werden.

Messung eines einzelnen Bursts

Der Messkopf ist im Modus *Burst Average* in der Lage, einzelne Bursts selbständig zu erkennen und ihren Leistungsmittelwert zu messen. In diesem Messmodus wartet der Messkopf auf eine steigende Triggerflanke und misst so lange, bis er eine fallende Flanke erkennt. Sollte innerhalb von 50 ms kein Burstende erkannt werden, bricht der Messkopf die Messung ab und gibt die Fehlermeldung *NRPERROR_TRIGGERQUEUEFULL* aus.

`*rst`	setzt den Messkopf auf die Voreinstellungen zurück
`sens:func "pow:burs:avg"`	Modus *Burst Average* einstellen
`trig:lev 1e-6`	Triggerschwelle auf 1 µW einstellen
`sens:pow:burs:dtol 50e-6`	Dropout-Toleranz einstellen
`init:imm`	Messung starten

Zusätzlich können noch Bereiche zu Beginn und am Ende des Bursts ausgeschlossen werden (z. B. um einen Einfluss von flachen Signalflanken oder Überschwingern auf das Messergebnis auszuschließen). Im folgenden Beispiel werden die ersten 20 µs und die letzten 30 µs des Bursts von der Messung ausgeschlossen:

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

Messung eines Frames mit mehreren Zeitfenstern (Timeslots)

Für die Messung der Leistung in mehreren aufeinanderfolgenden Zeitfenstern (Timeslots) in einem festen Zeitrahmen stellt der Messkopf den Modus *Timeslot* zur Verfügung. Für die Messung müssen Anzahl und Dauer der Zeitfenster eingestellt werden. Getriggert wird eine Messung durch das Messsignal selbst *(TRIGger:SOURce INTernal)* oder durch ein von außen zugeführtes Triggersignal *(TRIGger:SOURce EXTernal)*. Um Messfehler durch unpräzise Triggerung zu vermeiden, ist einer externen Triggerquelle der Vorzug zu geben. Eine Triggerung über den USB *(TRIGger:SOURce BUS)* ist zwar prinzipiell möglich, wird aber aufgrund der ungenauen zeitlichen Auflösung nicht empfohlen.

Im folgenden Beispiel wird gezeigt, wie die Leistung in den acht Timeslots eines GSM-/EDGE-Signals gemessen werden kann:

`*rst`	setzt den Messkopf auf die Voreinstellung zurück
`sens:func "pow:tsl:avg"`	Modus *Timeslot* wählen
`sens:freq 500e6`	HF-Trägerfrequenz einstellen (hier 500 MHz)
`sens:aver:coun:auto off`	Auto-Averaging deaktivieren
`sens:aver:coun 8`	Averaging-Faktor 8 einstellen
`trig:sour ext`	externe Triggerquelle wählen (Hier kann auch *int* für interne Triggerung angegeben werden, dann muss zusätzlich die Triggerschwelle mit *TRIGger:LEVel* festgelegt werden.)
`trig:slop pos`	Triggerung auf steigende Flanke des Triggersignals
`sens:pow:tsl:avg:coun 8`	Anzahl der aufeinanderfolgenden Timeslots (hier 8)
`sens:pow:tsl:avg:widt 570e-6`	Breite eines Timeslots in Sekunden (hier 570 µs = Breite eines GSM-Timeslots)
`init:imm`	Messung starten

Wie im Modus *Burst* können auch im Modus *Timeslot* Start- und Endbereiche der Timeslots von der Messung ausgenommen werden. Die Einstellparameter sind dieselben wie im Modus *Burst Average*:

`sens:tim:excl:star 50e-6`	die ersten 50 µs jedes Timeslots werden ausgeblendet
`sens:tim:excl:stop 80e-6`	die letzten 80 µs jedes Timeslots werden ausgeblendet

> Die Parameter *SENSe:TIMing:EXCLude:STARt* und *-STOP* gelten sowohl für den Modus *Burst Average* als auch für den Modus *Timeslot*, d. h. die in einem der beiden Modi gemachten Einstellungen werden beim Wechsel in den jeweils anderen Modus übernommen.

Scopemodus zur Darstellung von periodischen Signalen

Um den zeitlichen Verlauf der Leistung ähnlich wie mit einem Oszilloskop darzustellen, bietet der Messkopf den Modus *Scope*. Wie im Modus *Timeslot* ist eine Triggerung durch das Messsignal selbst oder ein von außen zugeführtes Triggersignal möglich. Um Messfehler durch unpräzise Triggerung zu vermeiden, ist auch hier einer externen Triggerquelle der Vorzug zu geben.

Im Modus *Scope* stehen separate Parameter für das Mittelungsfilter zur Verfügung.

Um den Messkopf für eine *Scope*-Messung zu konfigurieren, muss angegeben werden, welche Zeit durch die *Scope*-Darstellung abgedeckt werden soll, und in wie viele Intervalle die Darstellung aufgelöst werden soll. Die kleinste zeitliche Auflösung des Messkopfes beträgt 10 µs. Es ist daher zu beachten, dass bei der Einstellung der o. g. Parameter die Intervallbreite (= Zeit / Anzahl der Intervalle) diese Zeit nicht unterschritten wird, da sonst die *Scope*-Darstellung verfälscht wird.

Das folgende Beispiel zeigt die Nutzung des Modus *Scope*, wobei der Trigger aus dem Messsignal abgeleitet wird:

`*rst`	setzt den Messkopf auf die Voreinstellung zurück
`sens:func "xtim:pow"`	Modus *Scope* wählen
`sens:freq 500e6`	HF-Trägerfrequenz einstellen (hier 500 MHz)
`sens:trac:aver:coun:auto off`	Auto-Averaging deaktivieren
`sens:trac:aver:coun 8`	Averaging-Faktor 8 einstellen
`trig:sour int`	Triggerung durch das Messsignal wählen
`trig:slop pos`	Triggerung auf steigende Flanke des Messsignals
`trig:lev 1e-6`	Triggerschwelle einstellen (hier 1 µW)
`trig:hyst 3`	Triggerhysterese einstellen (hier 3 dB, d. h. eine erneute Triggerung durch Überschreiten der Triggerschwelle wird erst möglich, nachdem der Pegel des Messsignals die Triggerschwelle um mindestens 3 dB unterschritten hat)
`sens:trac:time 20e-3`	Zeit in Sekunden, die von der Scope-Darstellung abgedeckt werden soll (hier 20 ms)
`sens:trac:poin 200`	Anzahl der Intervalle, in die die Scope-Darstellung aufgelöst werden soll (hier 200, d. h. Auflösung = 20 ms / 200 = 100 µs)
`init:imm`	Messung starten

Scopemodus zur Darstellung eines einmaligen Vorgangs (Single-Shot-Messung)

Ein einmaliges, nichtperiodisches Signal kann im Modus *Scope* erfasst werden, wenn der Parameter *SENSe:TRACe:REALtime ON* gesetzt wird. Eine Mittelung ist bei einmaligen Vorgängen natürlich nicht möglich. Die Einstellungen des Mittelungsfilters werden daher ignoriert (und auch nicht verändert). Anders als in den letzten beiden Betriebsarten ist hier eine Bustriggerung möglich, da nicht mehrere Messungen durch die Mittelung überlagert werden.

Das folgende Beispiel zeigt die Einstellungen für die Single-Shot-Messung:

`*rst`	setzt den Messkopf auf die Voreinstellung zurück
`sens:func "xtim:pow"`	Modus *Scope* wählen
`sens:freq 500e6`	HF-Trägerfrequenz einstellen (hier 500 MHz)
`sens:trac:time 20e-3`	Zeit in Sekunden, die von der Scope-Darstellung abgedeckt werden soll (hier 20 ms)
`sens:trac:poin 200`	Anzahl der Intervalle, in die die Scope-Darstellung aufgelöst werden soll (hier 200, d. h. Auflösung = 20 ms / 200 = 100 µs)
`sens:trac:real on`	Single-Shot-Modus aktivieren
`trig:sour ext`	externe Triggerquelle wählen (Hier kann auch *int* für interne Triggerung angegeben werden, dann muss zusätzlich die Triggerschwelle mit *TRIGger:LEVel* festgelegt werden.)
`trig:slop pos`	Triggerung auf steigende Flanke des Triggersignals
`init:imm`	Messung starten

Für eine busgetriggerte Single-Shot-Messung sind folgende Befehle notwendig:

`trig:sour hold`	alle Triggerereignisse außer *TRIGger:IMMediate* ignorieren
`init:imm`	Messung initialisieren, Triggersystem wartet auf Triggerereignis
`trig:imm`	Single-Shot-Messung triggern

Hier ist zu beachten, dass durch den Befehl *TRIGger:IMMediate* die Triggerverzögerungszeit (Parameter *TRIGger:DELay*) ignoriert wird.

Zum Umgang mit mehreren NRP-Messköpfen an einem PC

Die Dynamic Link Library *NrpControl.dll* gestattet es, gleichzeitig mehrere Messköpfe zu betreiben (max. 127), allerdings können die Messungen nur sequenziell (d. h. mit nicht mehr als einem Messkopf gleichzeitig) durchgeführt werden. Der Messkopf, mit dem gemessen werden soll, wird zu diesem Zweck mit der DLL-Funktion *NrpSelectDevice* ausgewählt. Danach kann er konfiguriert und die Messung gestartet werden. Alle anderen, nicht ausgewählten Messköpfe behalten währenddessen ihre zuletzt eingestellte Konfiguration, liefern aber keine Messergebnisse und können auch sonst nicht angesprochen werden.

Um alle an einen PC angeschlossenen Messköpfe eindeutig identifizieren zu können, werden ihnen logische Namen zugeordnet, die z. B. mit den Adressen in einem IEC-Bus-Messsystem verglichen werden können. Diese logischen Namen können über die DLL-Funktion *NrpChannelAssignment* angezeigt und editiert werden. Diese Funktion öffnet ein Dialogfenster, in dem alle angeschlossenen Messköpfe (auch Messköpfe, die zuvor schon einmal angeschlossen waren und z. Z. nicht mehr angeschlossen sind) mit Typenbezeichnung, Seriennummer und logischem Namen aufgelistet sind. Messköpfen wird, wenn sie erstmalig an den PC angeschlossen werden, vom USB-Treiber ein Default-Name (Sensor1, Sensor2, Sensor3, …) zugewiesen, der entweder beibehalten oder geändert werden kann.

Leider lassen sich Messköpfe nicht direkt über ihren logischen Namen ansprechen, sondern man benötigt dazu den den Messkopf kennzeichnenden Parameter *i_wID*. Dabei handelt es sich um eine fortlaufende Nummer (0, 1, 2, …), die der USB-Treiber relativ willkürlich bei jedem Einschalten des PCs und jedem Messkopfwechsel neu vergibt. Diese laufende Nummer muss deswegen in solchen Fällen stets neu ermittelt werden. Dafür steht die Funktion *NrpGetIDByLogicalName* zur Verfügung, die aus dem logischen Namen eines Messkopfes dessen aktuelle laufende Nummer ermittelt. Ist der gewünschte Messkopf nicht angeschlossen, liefert die Funktion den Wert –1.