Maschinenstrahlzeit am ESR

Messungen an der stochastischen Kühlung
bei unterschiedlichen Strahlenergien
10.07.2025

Claudius Peschke

25. Juli 2025

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Einleitung

Das stochastische Kühlsystem im **E**xperimentier**s**peicher**r**ing ESR ist momentan für ein festes β ausgelegt. Sowohl in den Superelektroden der Pick-ups und Kicker als auch in der Signalverarbeitung gibt es feste Verzögerungsleitungen. Die Signalverarbeitung wäre mit vertretbaren Aufwand auf ein variables β aufzurüsten, sie Superelektroden jedoch nicht. Wahrscheinlich sind die Superelektroden aber für einen gewissen β-Bereich geeignet.

Bei der Maschinenstrahlzeit am 10.07.2025 sollte die Shuntimpedanz eines einzelnen Pick-up-Moduls gegen die Teilchengeschwindigkeit β·c und die Frequenz gemessen werden. Um auch die Phase zu erfassen sollte mit einem Pick-up- und einem Kicker-Modul die **B**eam **T**ransfer **F**unction bei unterschiedlichen Teilchengeschwindigkeiten gemessen werden.

Diese Messungen sollten ursprünglich schon während der Maschinenstrahlzeit vom 1. bis 6. März 2024 durchgeführt werden. Leider war jedoch nicht genug Zeit mit Strahl verfügbar. Am Ende der Strahlzeit 2025 gab es kurzfristig die Möglichkeit die Messungen nachzuholen.

Bei einem Experiment vor der Maschinenstrahlzeit wurde ein Primärstrahl aus ^{238}U 92+ Ionen mit 300 MeV/u in den ESR injiziert, mit dem Elektronenkühler gekühlt und mit den HF-Kavitäten abgebremst für die Ejektion Richtung Cryring. Dieser Ablauf wurde leicht modifiziert auch für die Maschinenstrahlzeit genutzt. Der Strahl wurde auf die gewünschte Energie abgebremst oder beschleunigt. Dann wurden die HF-Kavitäten abgeschaltet und bei eingeschaltetem Elektronenkühler mit dem coasting beam gemessen.

2 Versuchsaufbau

Zur Messung der Shuntimpedanzen werden die Anzahl der Ionen, die Umlauffrequenz und Schottky-Spektren um Harmonischen der Umlauffrequenzen benötigt. Die Anzahl der Ionen wird von einem fest im ESR verbauten Strahlstrommonitor geliefert. Zur Erfassung der Schottky-Spektren und einer genauen Messung der Umlauffrequenz wurde ein zusätzlicher Spektrumanalysator an der Signalverarbeitungsplatte am Pick-up im Quadrupol angeschlossen.

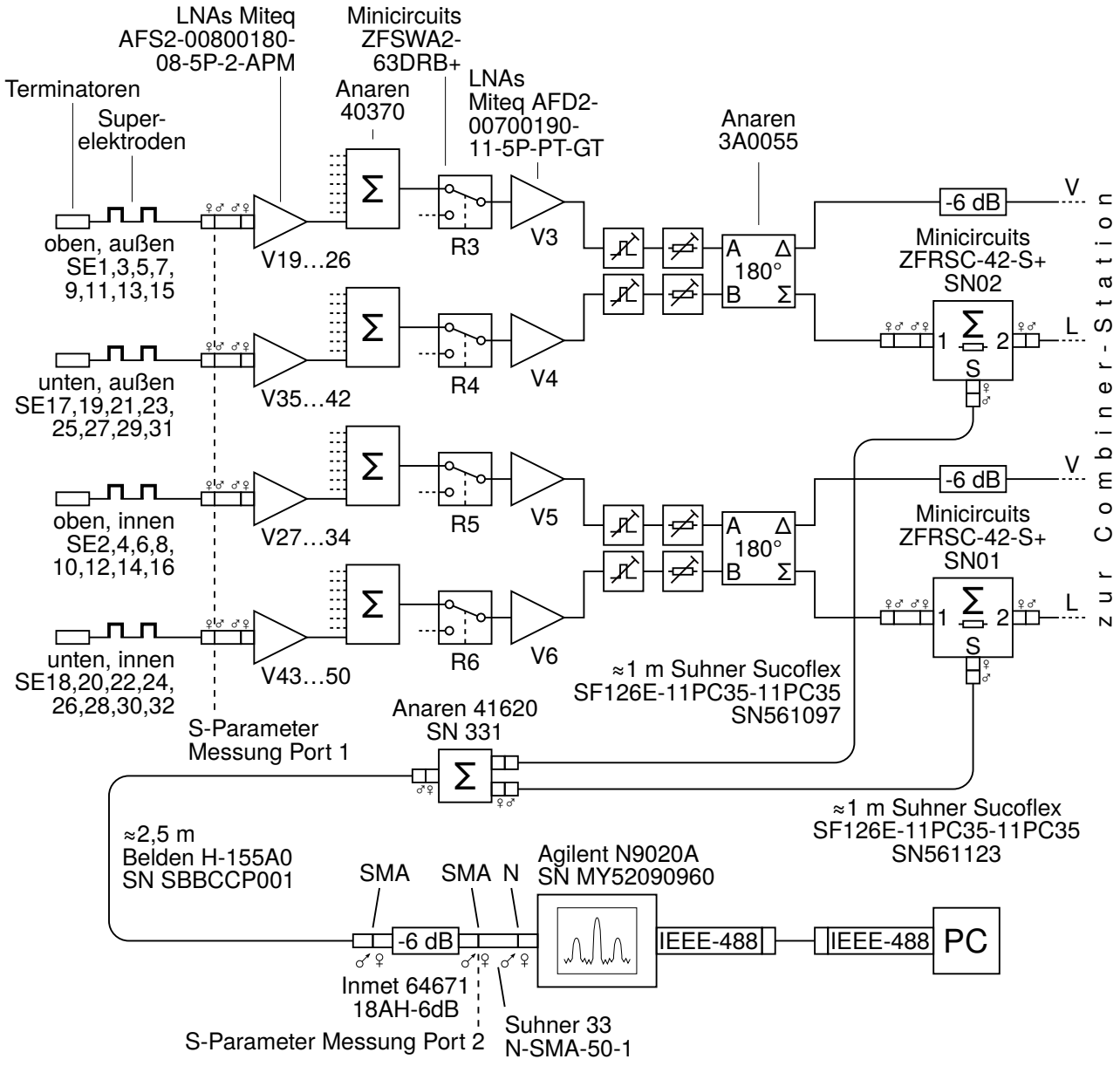


Abbildung 2.1: Versuchsaufbau an Quadrupol Pick-up

Wie in Abbildung 2.1 gezeigt, wurden die Summensignale der inneren und der äußeren Superelektroden abgegriffen, addiert und an den Spektrumanalysator geleitet. Dazu wurde auf jeder Seite ein 6 dB Abschwächer durch einen resistiven Leistungsteiler ersetzt. Der Leistungsteiler hat etwa die

gleiche Abschwächung und eine etwa 120 ps längere Signallaufzeit. Das normale Kühlsystem bleibt also funktionsfähig. Auf einem Laborwagen vor dem Quadrupol wurde ein Spektrumanalysator aufgestellt. Ein angeschlossener Computer übernahm die Datenerfassung und wurde per RDP aus dem HKR ferngesteuert.

3 Messungen

Die Messungen am ESR fanden am 10.07.2025 statt. Alle Messungen wurden mit einem ^{238}U 92+ Primärstrahl durchgeführt. Bei dem hoch geladenen Ionen verursacht der Elektronenkühler durch Elektroneneinfang relativ schnelle Strahlverluste. Der Elektronenkühler wurde daher auf einen möglichst kleinen Strom eingestellt. Die Messungen erfolgten bei den fünf unterschiedlichen Strahlenergien 200, 250, 300, 350 und 400 Mev/u. Der injizierte Strahl wurde jeweils auf die gewünschte Energie abgebremst oder beschleunigt. Bei gespeichertem Strahl wurde jeweils die horizontale Strahllage im Quadrupol Pick-up eingestellt, die Spektren eines Pick-up Moduls gemessen und eine BTF gemessen.

3.1 Einstellung Strahllage

Für jede Energie wurde zunächst die Strahllage am Pick-up im Quadrupol korrigiert. Die Bumps E01QS2F, E02QS2F, E01MU2 und E02MU2 wurden zunächst auf 0 mm gestellt.

Das Signal vom Richtkoppler im Palmer-Zweig wurde über den Verstärker V7 auf den Spektrumanalysator im Elektronikraum geleitet. Alle Zwischenverstärker bis zu dieser Auskoppelstelle wurden zunächst eingeschaltet. Der Spektrumanalysator wurde auf folgende Parameter eingestellt:

Center	≈1,2 GHz	(auf Hauptlinie einer Harmonischen eingestellt)
Span	3 MHz	
Points	1001	
Resolution BW	106 Hz	
Filter	Gaussian, Noise	
Video BW	1 kHz	
Ref Level	-50 dBm	
Scale	10 dB/div.	
Detector	Average	
Trace	Trace Avarage	
Averages	50	
Sweep Type	FFT	

Zur Bestimmung der horizontalen Strahllage im Quadrupol Pick-up wurden wechselweise die Zwischenverstärker V13 (außen) und V10 (innen) abgeschaltet. Die Amplituden der Hauptlinie wurden verglichen. Der Pick-up Bump E01QS2F wurde so eingestellt, daß die Amplituden außen und innen gleich waren. Der Kicker Bump E02QS2F wurde jeweils auf den gleichen Wert eingestellt.

Für die fünf unterschiedlichen Energien ergaben sich folgende Einstellungen:

Energie	E01QS2F,E01QS2F	P_{auen}/P_{innen}
200 MeV/u	+48 mm	-2 dB
250 MeV/u	+42 mm	-3 dB
300 MeV/u	+33 mm	-1 dB
350 MeV/u	+25 mm	+1 dB
400 MeV/u	0 mm	0 dB

Bei den Energien 200 MeV/u und 250 MeV/u reichte der Einstellbereich nicht um die Abweichung auf ≤1 dB einzustellen.

3.2 Messung Shuntimpedanz Quadrupol Pick-up

Die Messung der Shuntimpedanzen im Quadrupol Pick-up erfolgten mit dem Versuchsaufbau in Abbildung 2.1. Vermessen wurde das Modul 4 des Pick-ups. Der Elektronenkühler war eingeschaltet und die HF-Kavitäten aus. Bei jeder Energie wurden im Bereich 0,5 GHz bis 2 GHz Spektren um jede fünfte Hauptlinie n · fRev aufgenommen.

Die Steuerung des Spektrumanalysators erfolgte durch das Igor Pro Modul "`Messprogramm.ipf`". Alle Aktionen und die gemessenen Daten sind in der Igor Pro Experimentdatei "`Messung QuPU Spektren/2025-07-10 Messung.pxp`" abgespeichert. Zunächst wurden die vier LNAs des Moduls 4 und die Zwischenverstärker V3…6 eingeschaltet. Mit dem Funktionsaufruf **`eExpFindFRev(1e9, 1.5e9, `**f_s**`)`** (siehe Unterabschnitt 6.2.2) die genaue Umlauffrequenz aus Spektren von Harmonischen im Bereich 1…1,5 GHz ermittelt. Für f_s wurde jeweils ein Schätzwert für die Umlauffrequenz angegeben. Die gemessenen Umlauffrequenzen betrugen:

Energie	fRev
200 MeV/u	1,571206 GHz
250 MeV/u	1,701713 GHz
300 MeV/u	1,808579 GHz
350 MeV/u	1,898853 GHz
400 MeV/u	1,976044 GHz

Die Messung der Spektren erfolgte jeweils mit dem Funktionsaufruf **`eExpMsSpecs("mod4_…", 0.5e9, 2.8e9, 200e3, fRev, 5, `**N_{Start}**`)`** (siehe Unterabschnitt 6.2.1). Für N_{Start} wurde die Anzahl der Ionen, gemessen mit dem Strahlstrommonitor GE02DT_ML angegeben. Die Anzahl der Ionen am Ende jeder Meßreihe wurde ebenfalls in der Experimentdatei vermerkt.

Der Spektrumanalysator wird bei der Messung mit folgenden Parametern betrieben:

Span	200 kHz
Points	401
Pre Amp	LOW
Resolution BW	10,6 Hz
Filter	Gaussian, Noise
Video BW	100 Hz
Ref Level	-50 dBm
Detector	Average
Trace	Clear Write
Sweep Type	FFT

Für die Anzahl der Ionen wurden folgende Werte abgelesen:

Energie	N_{Start}	N_{End}
200 MeV/u	$9,6 \cdot 10^6$	$7,4 \cdot 10^6$
250 MeV/u	$17,7 \cdot 10^6$	$15,4 \cdot 10^6$
300 MeV/u	$15,4 \cdot 10^6$	$13,4 \cdot 10^6$
350 MeV/u	$3,7 \cdot 10^6$	$3,4 \cdot 10^6$
400 MeV/u	$3,2 \cdot 10^6$	$3,1 \cdot 10^6$

Für die Messung jeder Energie gibt es in der Experimentdatei einen Datenordner:

Energie	Datenordner
200 MeV/u	mod4_200MeV
250 MeV/u	mod4_250MeV_V2
300 MeV/u	mod4_300MeV
350 MeV/u	mod4_350MeV
400 MeV/u	mod4_400MeV

In jedem Datenordner sind unter den Namen `rawDBm`*nnn* die Spektren der $5 \cdot nnn$-ten Harmonischen oberhalb der Startfrequenz abgelegt. In einem Unterordner `parameter` sind die Meßparameter des Spektrumanalysators abgespeichert:

Variable	Wert	Einheit
`fRev`	Umlauffrequenz	Hz
`fStart`	minimale Startfrequenz der Meßreihe	Hz
`fStop`	maximale Endfrequenz der Meßreihe	Hz
`nStep`	Mittenfrequenzabstand der Spektren	fRev
`fSpan`	Frequenzbreite der einzelnen Spektren	Hz
`deltaF`	Punktabstand in einzelnen Spektren	Hz
`rBW`	Auflösungsbandbreite	Hz
`vBW`	Videobandbreite	Hz
`refLvl`	Referenzleistung	dBm
`nIons`	Anzahl der Ionen beim Start der Meßreihe	
`nIonsEnd`	Anzahl der Ionen am Ende der Meßreihe	

3.3 Messung BTF Palmer-Zweig

Die Messung der **B**eam **T**ransfer **F**unction im Palmer-Zweig erfolgte mit Modul 4 des Quadrupol Pick-ups. Am Quadrupol Kicker können die Module nur paarweise eingeschaltet werden. Gemessen wurde mit den Modulen 3 und 4. Für die BTF-Messung wurde der Zwischenverstärker V13 ausgeschaltet, das Relais R25 auf Palmer geschaltet, und die Leistungsverstärker LV13, 14, Leistungsverstärker LV21 und 22 eingeschaltet. Der Abschwächer A4 war auf 0 dB eingestellt und der Phasenschieber PH1 auf 0°.

Die BTF-Messung erfolgte mit dem Spektrum- und Netzwerkanalysatoren im Elektronikraum unter Kontrolle der Web-Applikation „ESR Stochastic Cooling System“. Zunächst wurde mit „Measure → Rev. Freqency“ erneut die Umlauffrequenz gemessen. Mit „Measure → BTF Calibrate“ wurde der Netzwerkanalysator auf das Transferrelais R11 kalibriert und mit „Measure → BTF Measure“ wurde die eigentliche Messung durchgeführt. Die Meßparameter und Ergebnisse befinden sich in Unterordnern des Ordners `esrsc01/log/2025-07-10`. Folgende Daten gehören zusammen:

Energie	Igor Folder	Rev. Freqency	BTF Calibrate	BTF Measure
200 MeV/u	mod4_200MeV	18-25-56_findFRev	18-27-17_btfCal	18-28-39_btfMeas
250 MeV/u	mod4_250MeV_V2	17-53-18_findFRev	17-55-29_btfCal	17-56-58_btfMeas
300 MeV/u	mod4_300MeV	15-50-18_findFRev	15-50-59_btfCal	15-59-13_btfMeas
350 MeV/u	mod4_350MeV	19-11-53_findFRev	19-12-47_btfCal	19-14-11_btfMeas
400 MeV/u	mod4_400MeV	19-48-24_findFRev	19-49-26_btfCal	19-50-54_btfMeas

Die Messung der Umlauffrequenz erfolgte mit folgenden Parametern:

Start Frequency	1,0 GHz
Stop Frequency	1,5 GHz
Final Span:	200 kHz
Reference Level	-50 dBm
Smooth Window Coarse	11
Smooth Window Final	1001

Es ergaben sich die Umlauffrequenzen in der folgenden Tabelle. In der letzten Spalte sind zum Vergleich die gemessenen Umlauffrequenzen von der Shuntimpedanz-Messung in Abschnitt 3.2 eingetragen. Die Ergebnisse sind gut reproduzierbar. Die maximale Abweichung beträgt 23 Hz beziehungsweise 15 ppm.

Energie	f_{rev}	$f_{rev,RsMs}$
200 MeV/u	1,571183 GHz	(1,571206 GHz)
250 MeV/u	1,701709 GHz	(1,701713 GHz)
300 MeV/u	1,808580 GHz	(1,808579 GHz)
350 MeV/u	1,898851 GHz	(1,898853 GHz)
400 MeV/u	1,976045 GHz	(1,976044 GHz)

Die Kalibration des Netzwerkanalysators legt auch die Meßpunkte bei der BTF Messung fest. Es wurde für jede Harmonische („Harmonics Step Size“ = 1) im Frequenzbereich 0,5 GHz („Min. Start Frequency“) bis 2,0 GHz („Max. Start Frequency“) kalibriert. Bei der Kalibration für 300 MeV/u wurde jedoch fälschlicherweise der Default-Wert 1,0...1,5 GHz verwendet.

Die eigentliche BTF-Messung erfolgte mit einer ZF-Bandbreite („IF Bandwidth“) von 100 Hz und einer Quellenleistung („Source Power“) von -20 dBm.

4 Auswertung

Bei den Shuntimpedanz-Messungen wurden Schottky-Spektren von einem einzelnen Pick-up Modul gemessen. Für die Auswertung wird zusätzlich der Frequenzgang der Signalverarbeitungskette vom der Superelektrode zum Spektrumanalysator benötigt. Diese Messungen konnten erst nach der Strahlzeit durchgeführt werden.

4.1 Messung Transmission Superelektrode zu Spektrumanalysator

Die Messungen der Frequenzgänge der Signalverarbeitungskette von den Superelektroden zum Spektrumanalysator erfolgte mit einem Netzwertanalysator Rohde & Schwarz ZNB8 direkt im ESR-Cave. Dazu wurde der Meßaufbau in Abbildung 2.1 an den gestrichelt markierten Bezugsebenen aufgetrennt. Der Netzwerkanalysator wurde mit folgenden Einstellung betrieben:

Start	0,5 GHz
Stop	2,0 GHz
Points	1501
IFBW	10 Hz
Power	-54 dBm bei Messung, -10 dBm bei Kalibration
Calibration	THRU mit Rohde & Schwarz ZV-Z32, SN 100620

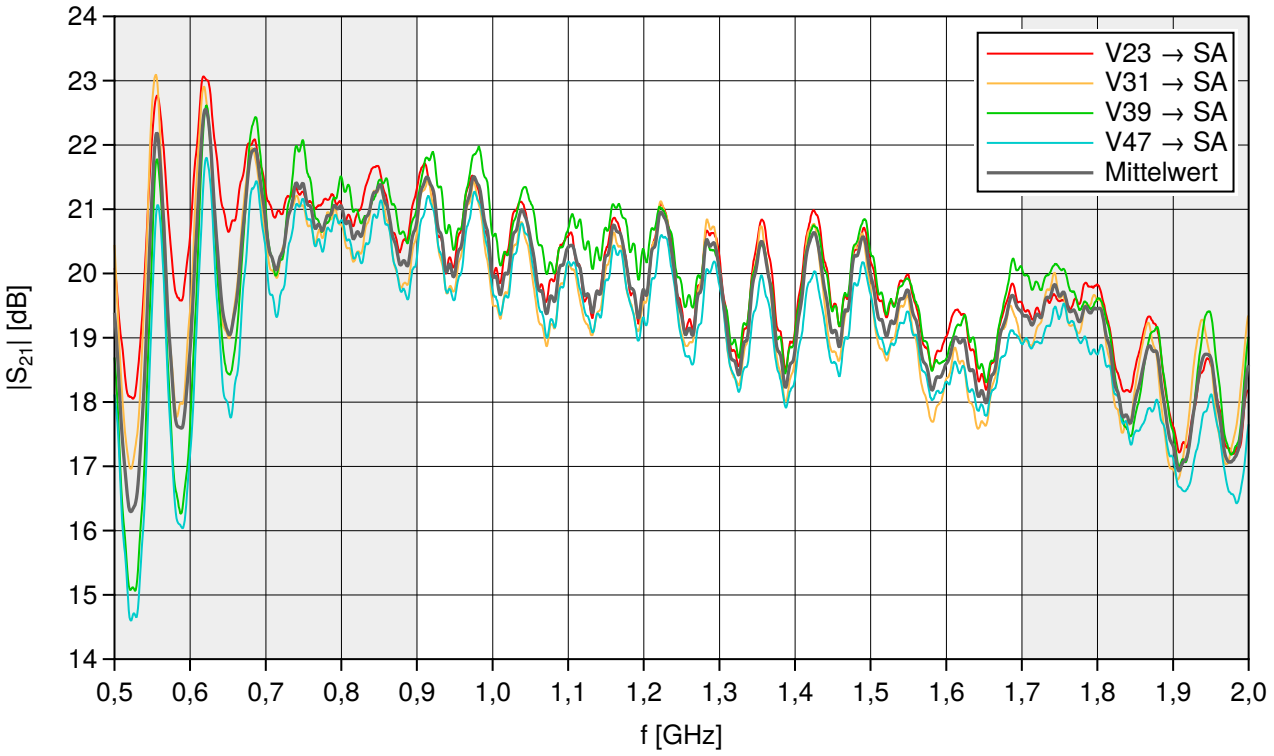


Abbildung 4.1: Transmissionen Eingänge LNAs (V23, V31, V39, V47) zum Spektrumanalysator

Die Abbildung 4.1 zeigt die Transmissionen vom Eingang jedes LNAs des Moduls 4 zum Spektrumanalysator. Die nicht zugänglichen Leitungen von der Vakuumdurchführung zu den LNAs

sind nicht in den Messungen enthalten. Die grau hinterlegten Frequenzbereiche liegen außerhalb des Nennfrequenzbereichs der stochastischen Kühlung.

Bei 1,2 GHz sind die folgenden Verstärkungen und Abschwächungen zu erwarten. Die LNAs (Miteq AFS2-00800180-08-5P-2-APM) haben im Mittel eine Verstärkung von 28 dB. Die zweiten Verstärker (Miteq AFD2-00700190-11-5P-PT-GT) haben im Mittel 22 dB Verstärkung. Zusätzlich gehen die Signale durch einen 8:1-Combiner (Anaren 40370, -9,5 dB), einen Signalschalter (Minicircuits ZFSWA2-63DRB+, -1,1 dB) einen 180°-Hybrid (Anaren 3A0055, -3,1 dB), einen resistiven Leistungsteiler (Minicircuits ZFRSC-42-S+, -6,0 dB), Eine Leitung (Suhner Sucoflex SF126E-11PC35-11PC35, -0,3 dB), einen reaktiven Leistungsteiler (Anaren 41620, -3,2 dB), noch eine Leitung (≈2,5 m Belden H-155A0, -0,8 dB) und einen 6 dB Abschwächer. Die Transmissionen sollten also bei +20,0 dB liegen. Für den Mittelwert stimmt das auch. Es zeigt sich jedoch eine unerwartet hohe Welligkeit von etwa $\pm$1 dB und ein erwarteter Abfall mit steigender Frequenz. Die Welligkeit war auch in den vorangegangenen Messungen der Spektren zu sehen.

4.2 Berechnung Shuntimpedanzen aus Schottky-Spektren

Die gemessenen Schottky-Spektren setzen sich aus zwei Komponenten zusammen und sind durch die Signalverarbeitungskette verfälscht. Abbildung 4.2 zeigt den Signalfluss. Das Signal am Ausgang jeder Superelektrode setzt sich zusammen Schottky-Spektrum und thermischem Rauschen.

Die Leistung einer Linie im Schottky-Spektrum ist abhängig von der Anzahl der Ionen N im Strahl, der Ladung eines Ions q_i, der Umlauffrequenz f_{rev} und der gesuchten Shuntimpedanz Z_{PU}. Da bei den Messungen der Elektronenkühler eingeschaltet war ist es ausreichend die Hauptlinie zu betrachten. Die Nebenlinien sind so klein, daß sie vernachlässigt werden können.

Die spektrale Rauschleistungsdichte dP_{noise}/df ist gleich der Boltzmann-Konstante multipliziert mit der Rauschtemperatur T_{noise}. Der erste rauscharme Vorverstärker (LNA) liefert noch eine Beitrag dazu. Seine Verstärkung ist groß genug, daß die weitere Signalverarbeitungskette praktisch keine Rolle mehr spielt. Ihr Einfluß wurde vernachlässigt.

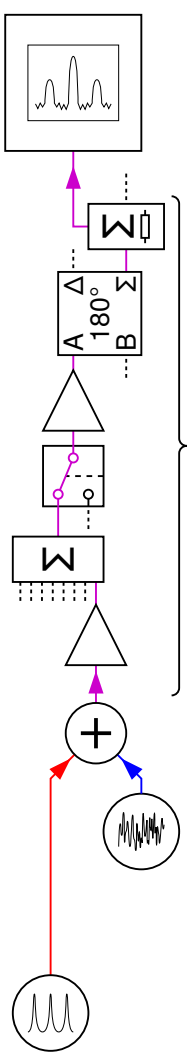
Die Überlagerung der beiden Signale geht durch den rauscharmen Vorverstärker und die weitere Signalverarbeitungskette bis zum Spektrumanalysator. Die Transmission dieses Abschnitts wurde für jede Superelektrode einzeln vermessen (siehe Abschnitt 4.1).

Am Ende der Signalverarbeitungskette wurden mit dem zusätzlichen Spektrumanalysator jeweils 152 bis 191 Einzelspektren aufgezeichnet.

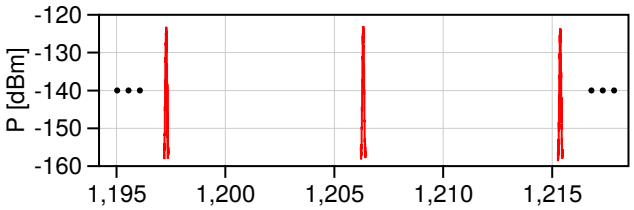
Die Auswertung der Spektren erfolgt in umgekehrter Richtung. Dazu werden zunächst die frequenzabhängigen Transmissionen der Signalverarbeitungskette aus den Spektren herausgerechnet. Die Leistungsspektren werden in spektrale Leistungsdichten umgerechnet. Der eingestellte Detektor „Average“ erfordert beim verwendeten Spektrumanalysator hierbei keinen Korrekturfaktor. Das Integral über den Bereich der Hauptlinie ergibt die Linienleistung. Aus den Randbereichen abseits der Linie wird eine spektrale Rauschleistungsdichte gemittelt. Diese Schritte erfolgen in der Funktion **`calcMsSa()`** des Auswertungsprogramms (siehe Unterabschnitt 6.3.8).

Mit der Funktion **`calcRsTn()`** des Auswertungsprogramms (siehe Unterabschnitt 6.3.9) werden hieraus schließlich die Shuntimpedanz und die Rauschtemperatur berechnet. Berechnet wird die Pick-up-Shuntimpedanz in Circuit Convention. Für die Anzahl der Ionen wird ein exponentieller Abfall während der Meßreihe angenommen.

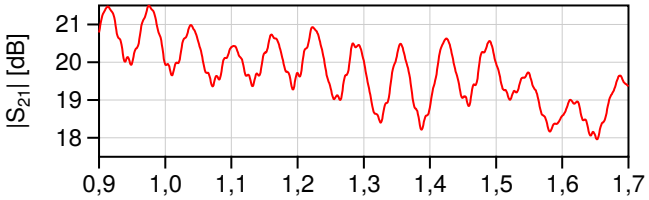
Die Transmission der Signalverarbeitungskette am Quadrupol Pick-up weist aus noch unbekannten Grund eine hohe Welligkeit auf. Da die Positionen der Maxima und Minima zwischen den Messungen nicht exakt gleich geblieben sind weisen auch die ausgewerteten Shuntimpedanzen und Rauschtemperaturen eine Restwelligkeit auf. Die Welligkeit ist der Signalverarbeitungskette und nicht den Elektroden zuzuordnen. Es erscheint daher legitim die ausgewerteten Daten numerisch etwas zu glätten. In den Diagrammen in Abbildung 4.2 und in den folgenden Abschnitten sind die geglätteten und die ungeglätteten Daten zu sehen.



Spektrumanalysator:
152…191 Einzelspektren um Harmonische der Umlauffrequenz für jede Energie gemessen

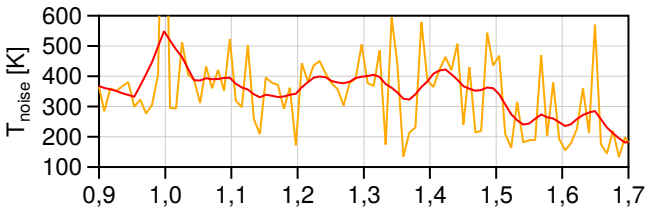


Signalverarbeitung von Superelektrode zum Spektrumanalysator:
Transmission über Frequenz gemessen (1501 Punkte)



spektrale Rauschleistungsdichte von Terminator und Superelektrode:

$$\frac{dP_{noise}}{df} = k_b \cdot T_{noise}$$



Leistung der Linie im Schottky-Spektrum:

$$P_{Schottky} = 2 \cdot N \cdot (q_i \cdot f_{rev})^2 \cdot Z_{PU}$$

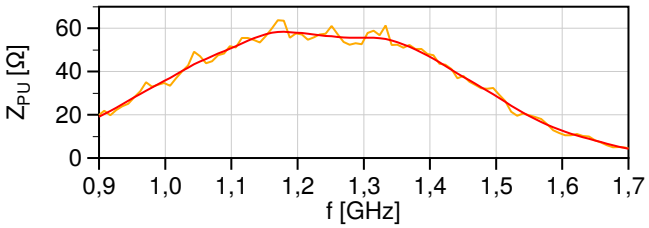


Abbildung 4.2: Auswertung der Schottky-Spektren

4.3 Auswertung Shuntimpedanzen Pick-up Quadrupol

Die Abbildungen 4.3 und 4.4 zeigen die ungeglätteten und geglätteten gemessenen Shuntimpedanzen für fünf verschiedenen Energien. Die grau hinterlegten Frequenzbereiche liegen außerhalb des Nennfrequenzbereichs der stochastischen Kühlung.

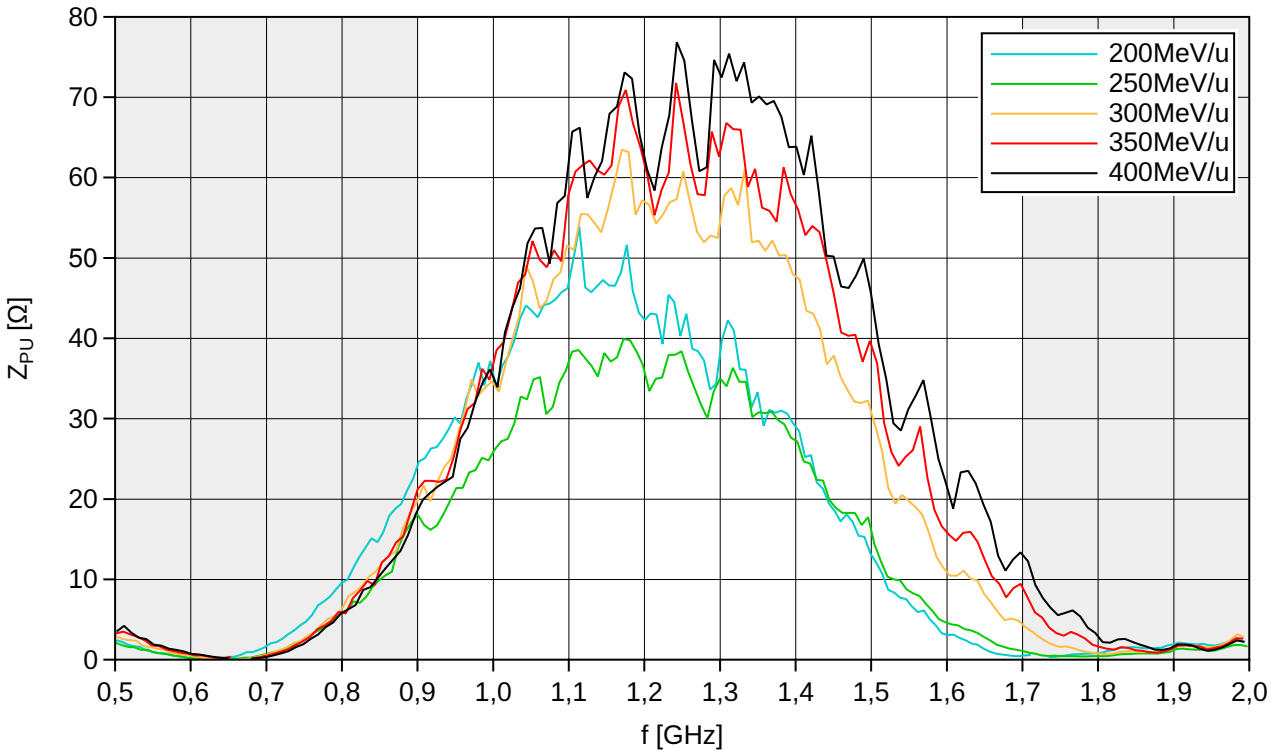


Abbildung 4.3: ungeglättete Shuntimpedanzen

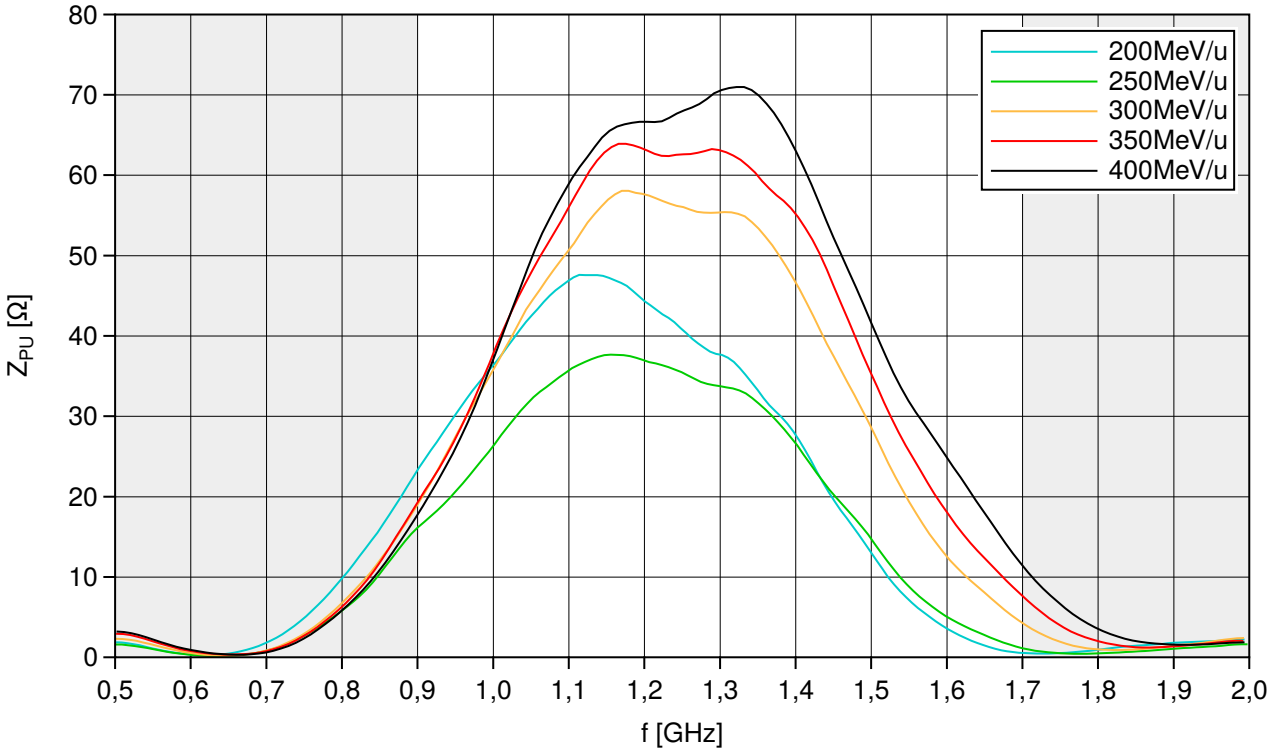


Abbildung 4.4: geglättete Shuntimpedanzen

Da das Summensignal ausgewertet wurde handelt es sich um longitudinale Shuntimpedanzen. Diese ermöglicht einen Vergleich der Superelektroden bei verschiedenen Energien. Für die Palmer-Kühlung und die vertikale Kühlung sind transversale Shuntimpedanzen maßgeblich, die hier nicht gemessen wurden, aber über das Panofsky-Wenzel-Theorem mit diesen verknüpft sind.

Die Superelektroden sind wahrscheinlich für 400 MeV/u bis 500 MeV/u dimensioniert. Wie zu erwarten war nimmt die Shuntimpedanz kleineren Energien ab. Für höhere Frequenzen ist der Abfall stärker. Die nutzbare Bandbreite nimmt also ab. Bei der 200 MeV/u Messung wird die Bandbreite kleiner, die Shuntimpedanz im Maximum ist aber unerwartet höher.

4.4 Auswertung Rauschtemperaturen

Aus den Randbereichen der Schottky-Spektren abseits der Harmonischen wird jeweils eine Rauschtemperatur berechnet. Die Rauschtemperaturen waren eigentlich nicht von besonderem Interesse. Sie eignen sich aber gut um die Messungen und die Auswertung der Shuntimpedanz auf Plausibilität zu testen. Die Abbildungen 4.5 und 4.6 zeigen die ungeglätteten und geglätteten gemessenen Rauschtemperaturen für alle Energien. Die grau hinterlegten Frequenzbereiche liegen außerhalb des Nennfrequenzbereichs der stochastischen Kühlung.

Als Rauschtemperaturen wird hier die Temperatur eines idealen Abschlußwiderstands bezeichnet, der am Eingang eines rauschfreien Verstärkers hängt, der die Verstärkung unseres jeweiligen LNAs hat. Der Verstärker liefert dann die gleiche spektrale Rauschleistungsdichte.

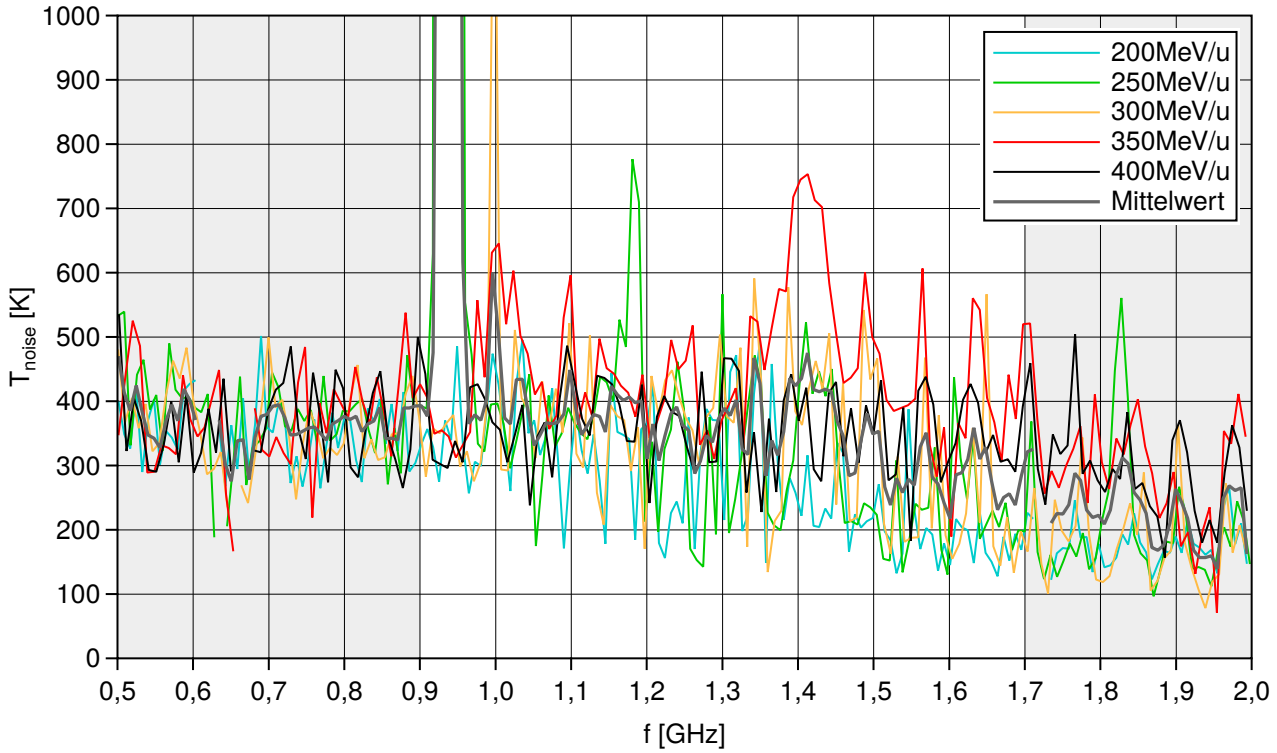


Abbildung 4.5: ungeglättete Rauschtemperaturen

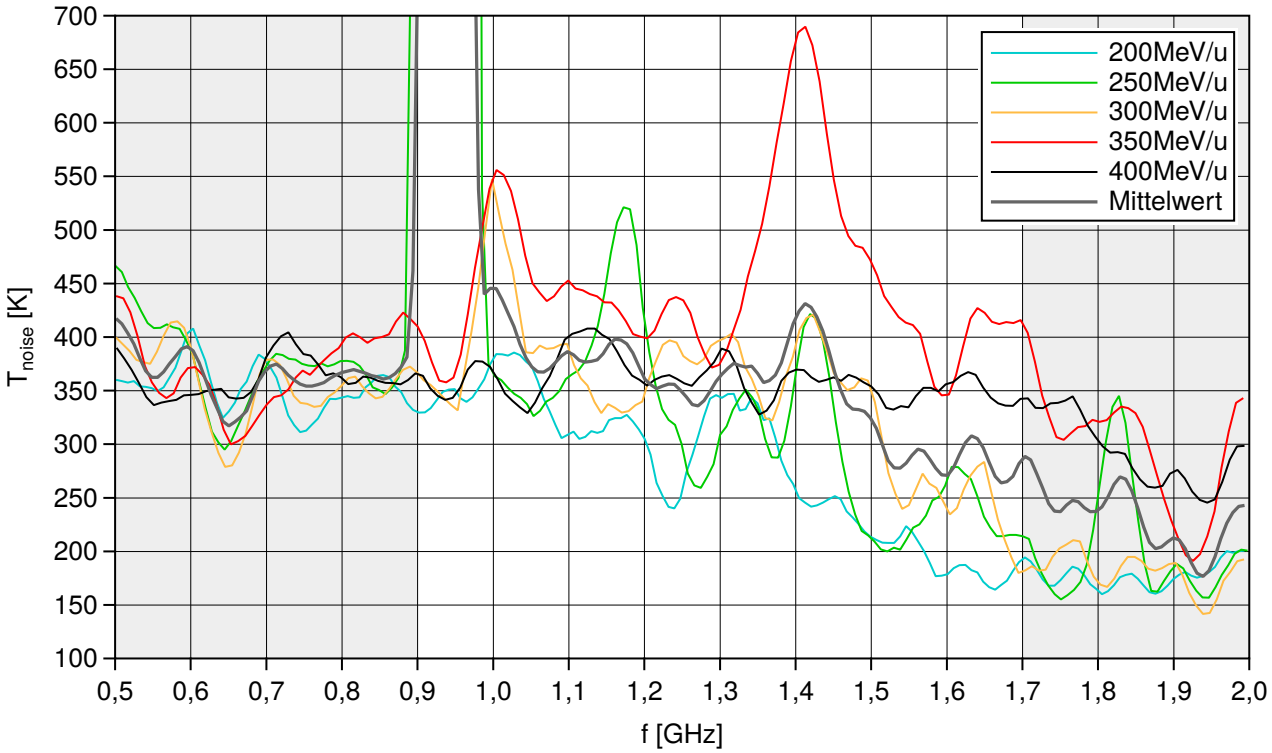


Abbildung 4.6: geglättete Rauschtemperaturen

Die Terminatoren, Superelektroden und alle Leitungen sind auf Umgebungstemperatur von etwa 298 K. Die rauscharmen Vorverstärker haben eine spezifizierte Rauschzahl von 0,7 dB. Das entspricht einer Zusatztemperatur von 51 K. Im Idealfall wären also alle Rauschtemperaturen konstant 349 K. Tatsächlich sind die Superelektroden, Leitungen und LNAs natürlich nicht perfekt angepasst. Die Werte können einerseits durch Verluste vor dem LNA höher liegen. Wenn der LNA durch Reflexionen seinen eigenen Eingang „sieht“ können sie auch niedriger liegen. Der Eingang eines LNAs kann eine niedrigere Rauschtemperatur aufweisen als seine Umgebungstemperatur. Durch die Leitungslängen wird der Verlauf frequenzabhängig. Die Spitze um 934 MHz bei der 250 MeV/u Messung, die auch auf den Mittelwert durchschlägt ist entweder ein statistischer Ausreisser oder wahrscheinlicher eine Einstrahlung aus einem GSM-900 Netz. Die meisten Meßwerte liegen um die idealen 349 K. Die Messungen sind damit plausibel.

4.5 Auswertung BTF Palmer-Zweig

Die Messung der Shuntimpedanzen liefert keine Information über den Phasengang. Es könnte sein, daß die Superelektroden niedrigeren Energien zwar genügend Signal- zu Rauschabstand liefern, aber der Phasengang die stochastische Kühlung unmöglich macht. Eine **B**eam **T**ransfer **F**unction Messung liefert die Phaseninformation.

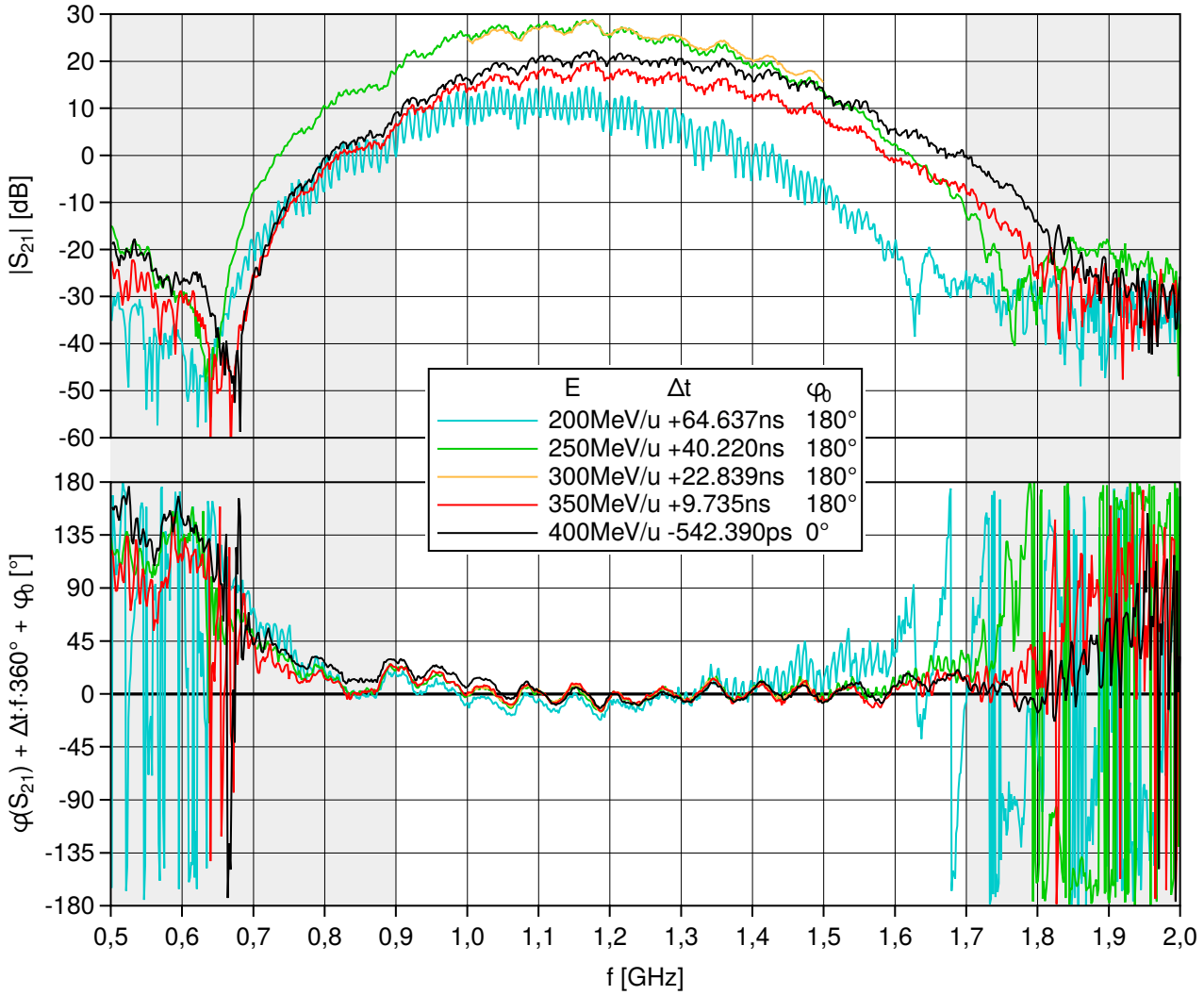


Abbildung 4.7: Rohdaten der BTF-Messung

Gemessen wurde im Palmer-Zweig mit Modul 4 des Quadrupol Pick-ups und den Modulen 3 und 4 des Quadrupol Kickers (siehe Abschnitt 3.3). Abbildung 4.7 zeigt die Transmission der BTF Messung für alle Energien. Die grau hinterlegten Frequenzbereiche liegen außerhalb des Nennfrequenzbereichs der stochastischen Kühlung. Die obere Hälfte zeigt den Betrag und die untere Hälfte den nicht Frequenz-linearen Anteil der Phase. Der Frequenz-lineare Anteil ist als Verzögerungszeit Δt angegeben. Ein konstanter Phasenanteil von 180° wurde gegebenenfalls auch getrennt als φ_0 angegeben. Die Phase φ_0 sollte eigentlich immer 180° sein wenn mit den inneren Pick-up Elektroden gemessen wird (Zwischenverstärker V13 ausgeschaltet). Bei der Messung bei 400 MeV/u ist das nicht der Fall. Vermutlich wurde versehentlich außen statt innen gemessen (Zwischenverstärker V10 ausgeschaltet).

Die Module des Kickers sind jeweils mit Verzögerungsleitungen fester Länge angeschlossen. Für niedrigere Energien stimmt damit die Signalverzögerungszeit von Modul zu Modul nicht mehr. Dies verursacht Betrags- und Phasenfehler, die nichts mit den eigentlichen Superelektroden zu tun haben. Dieser Einfluß muss also herausgerechnet werden. Um den kombinierten Einfluß der beiden Module

auf den Strahl zu berechnen kann wegen der Reziprozität den Kicker als Pick-up betrachten. Wenn zwei Pick-ups, die im Abstand Δz angebracht sind jeweils eine Spannung U_0 liefern, dann ist die über die zwei unterschiedlich langen Leitungen geführte und kombinierte Ausgangsspannung U_r:

$$U_r = \frac{U_0}{\sqrt{2}} \cdot \frac{\sin(\Delta\varphi_{err})}{\sin\left(\frac{\Delta\varphi_{err}}{2}\right)} \cdot e^{\frac{i\cdot\Delta\varphi_{err}}{2}} \tag{4.1}$$

mit dem Phasenfehler

$$\Delta\varphi_{err} = \frac{\omega \cdot \Delta z}{c} \cdot \left(\frac{1}{\beta} - \frac{1}{\beta_{design}}\right). \tag{4.2}$$

Hierbei ist β_{design} der Geschwindigkeitsfaktor für den die Leitungen dimensioniert wurden und

$$\beta = \sqrt{1 - \frac{1}{\left(\frac{E_{kin}}{m_0 \cdot c^2 + 1}\right)^2}} \tag{4.3}$$

der Geschwindigkeitsfaktor zur Energie E_{kin} eines Ions mit der Ruhemasse m_0.

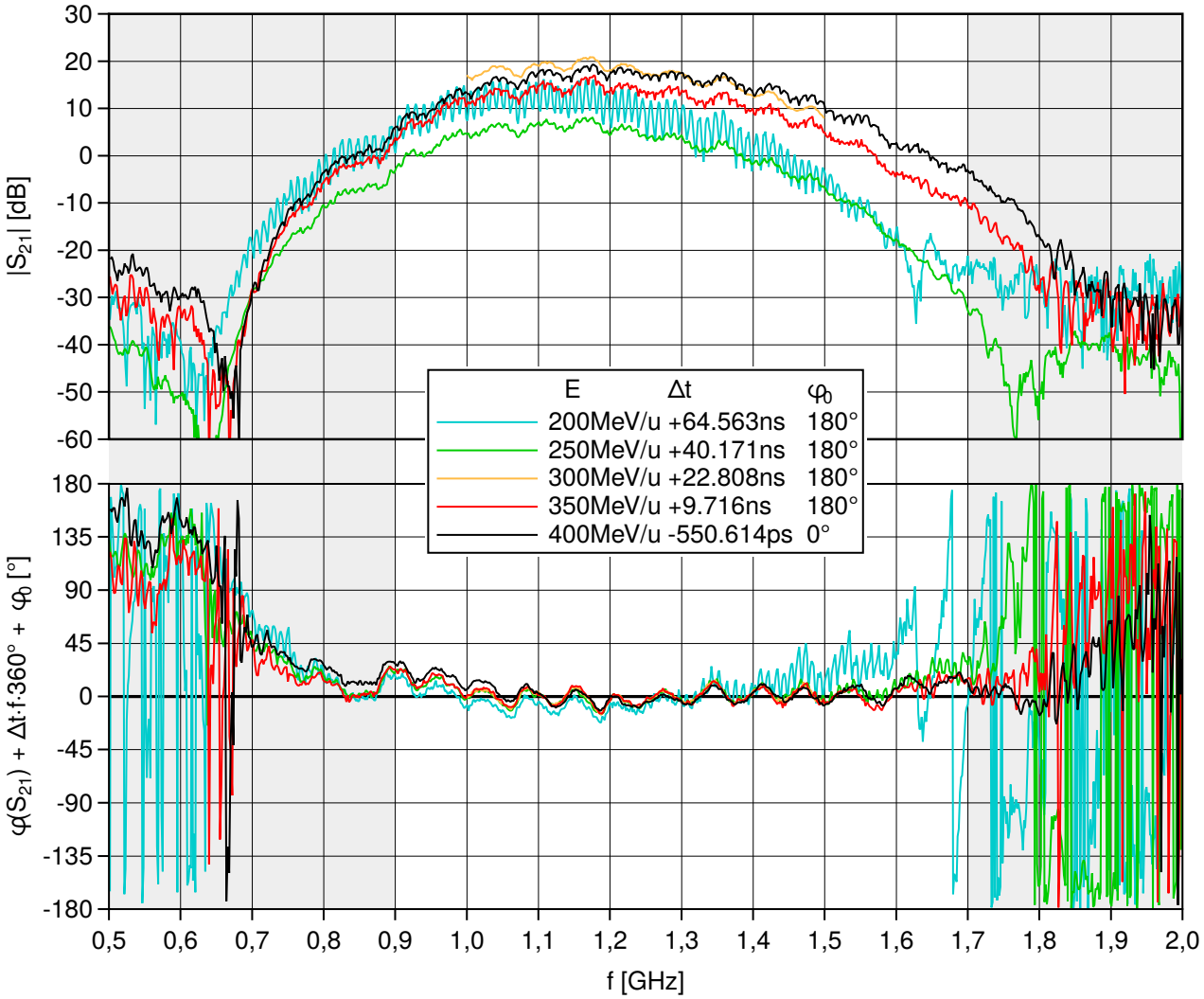


Abbildung 4.8: korrigierte BTF-Messung

Neben dem komplexen Verhältnis U_0/U_r wurde auch noch die Anzahl der Ionen auf die Anzahl der Ionen bei der 400 MeV/u Messung normiert. Hierbei wurde angenommen, daß sich die Teilchenzahl nach der Messung der Spektren mit der gleichen Basis exponentiell weiter verringert. Es wurde

nur der Startzeitpunkt der BTF-Messung berücksichtigt. Der Abfall während der BTF-Messung wurde vernachlässigt, da sie nur wenige Sekunden braucht. Abbildung 4.8 zeigt die korrigierte Transmission der BTF Messung für alle Energien.

Die Beträge der BTF-Messung rücken durch die Normierung der Teilchenzahl näher aneinander. Auf allen Messungen sieht man eine Welligkeit mit großer Periodenlänge und eine mit viel kleinerer Periodenlänge. Die große Periodenlänge ist wahrscheinlich auf Reflexionen in der Signalverarbeitung zurück zu führen. Wenn man den Netzwerkanalysator weiter laufen lässt sieht man, daß sie über längere Zeit konstant bleibt. Die Welligkeit mit kleinerer Periodenlänge verändert sich dagegen von Durchlauf zu Durchlauf. Hier liegt die Ursache wohl an anderer Stelle im ESR. Möglicherweise war die HF-Kavität nicht vollständig ausgeschaltet oder der Tune ist in der Nähe einer Resonanz. In Abbildung 4.9 wurde die Welligkeit kleinerer Periodenlänge geglättet. Abbildung 4.10 ist nur der Nennfrequenzbereich der stochastischen Kühlung dargestellt.

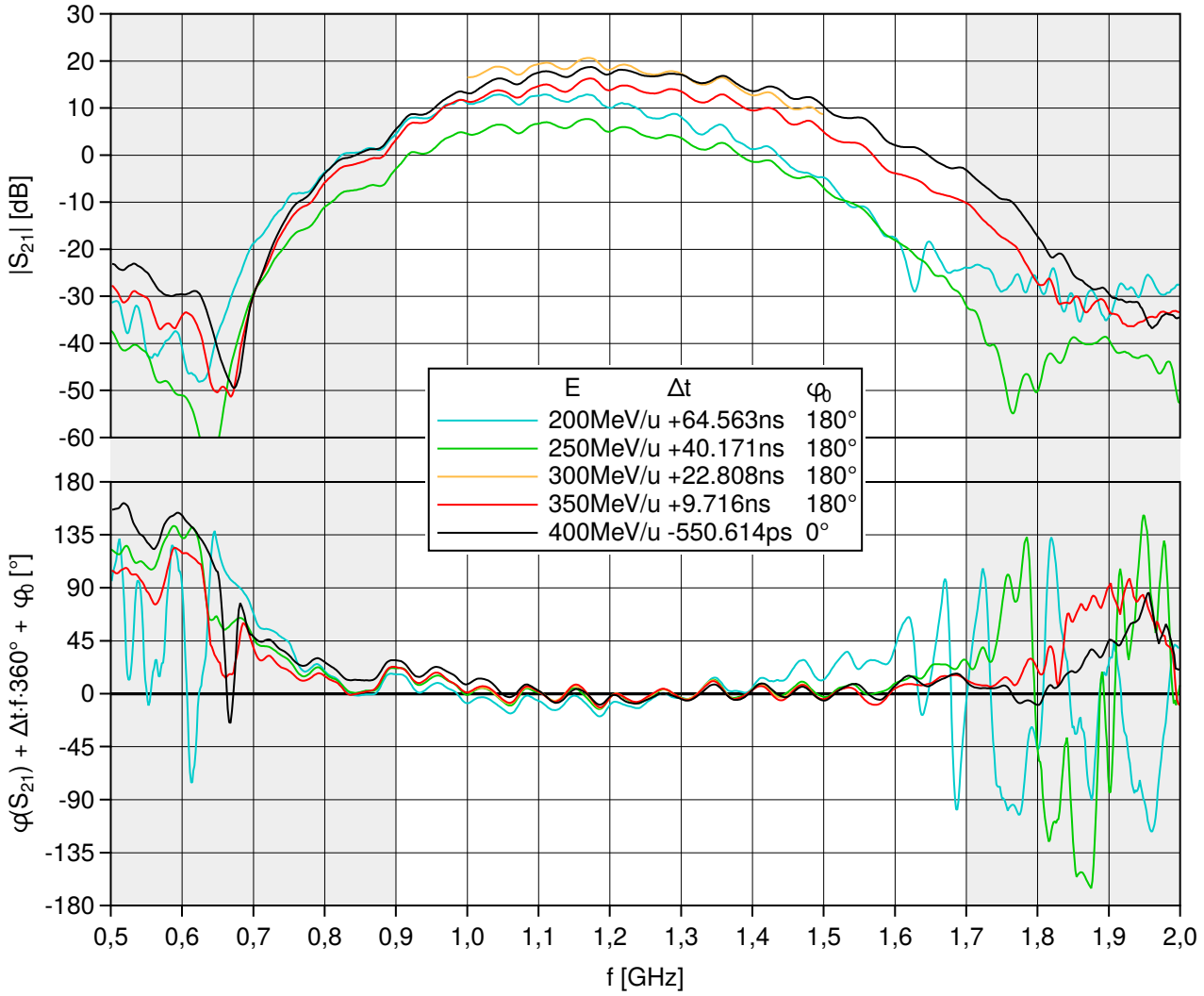


Abbildung 4.9: geglättete korrigierte BTF-Messung

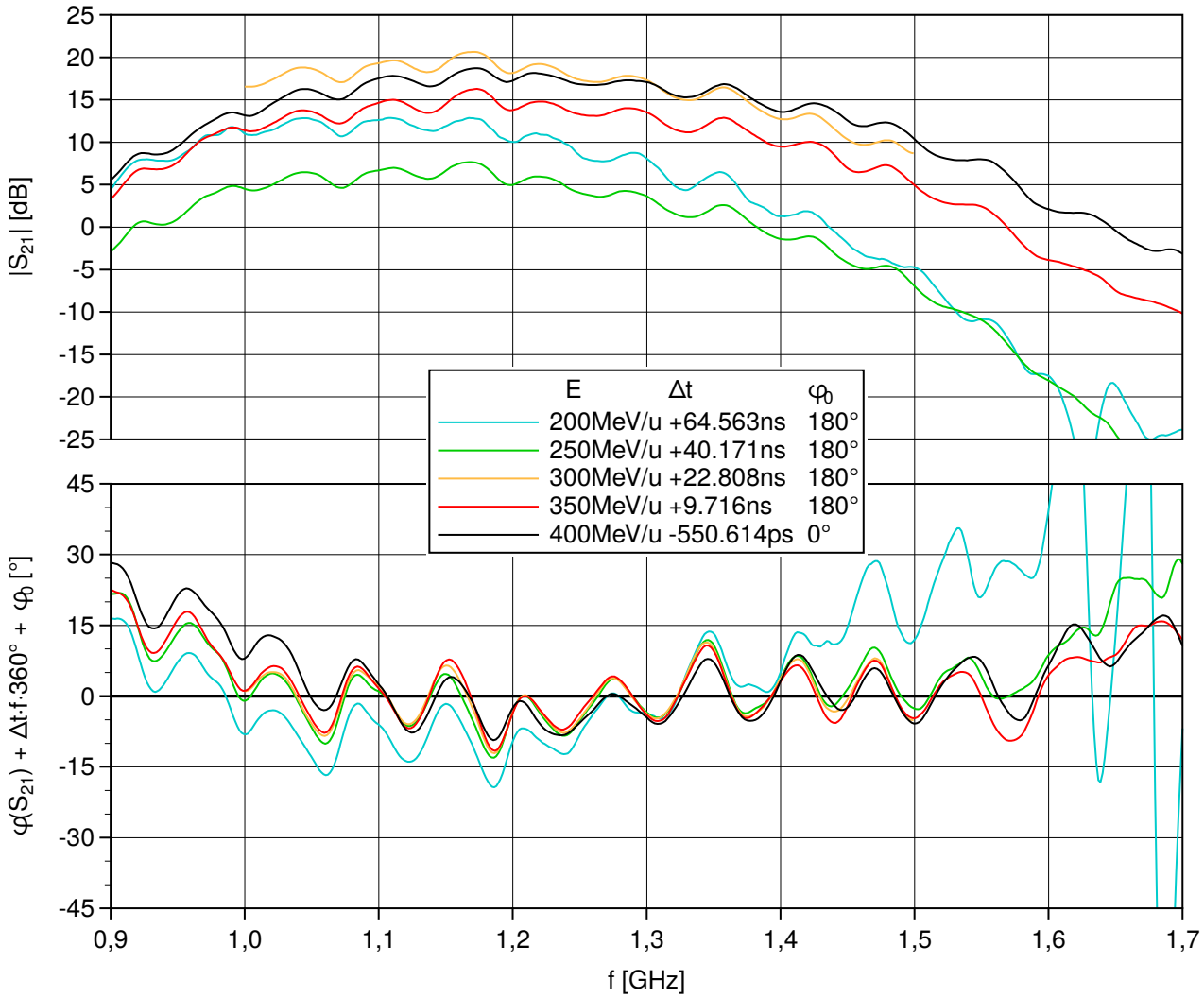


Abbildung 4.10: Nennfrequenzbereich der geglätteten korrigierten BTF-Messung

Bei allen Energien ≥250 MeV/u bleibt die Phase weitgehend gleich. Bei 200 MeV/u läuft sie bei höheren Frequenzen weg und zeigt sogar Sprünge. Die Beträge verhalten sich zueinander nicht ganz wie erwartet. Möglicherweise ist die Annahme des exponentiellen Abfalls der Teilchenzahl über die Zeit nicht vollständig. Die Zeit vom Ende der Aufnahme der Spektren bis zur BTF-Messung war sehr unterschiedlich lang.

5 Zusammenfassung und Ausblick

Bei der Maschinenstrahlzeit konnte für fünf Strahlenergien von 200 MeV/u bis 400 MeV/u die Shuntimpedanz über die Frequenz und die BTF über die Frequenz gemessen werden. Die Messungen zeigen, daß die verbauten Superelektroden bis herunter zu 250 MeV/u noch für eine stochastische Kühlung verwendbar wären. Sie haben dann noch etwa die halbe Shuntimpedanz, die nutzbare Bandbreite nimmt leicht ab und der Phasengang ist noch gut. Für hoch geladene Ionen ist das sicher ausreichend, für niedrig geladene Ionen könnte das Signal- zu Rauschverhältnis zu gering werden. Bei der 200 MeV/u Messung war die Shuntimpedanz scheinbar wieder etwas größer, die Bandbreite ist jedoch deutlich geringer und der Phasengang ist schlecht. Hier könnte es schon zu Heizeffekten bei höheren Frequenzen kommen. Die 200 MeV/u Messung ist allerdings etwas mit Vorsicht zu betrachten solange nicht geklärt ist woher die Welligkeit mit kleiner Periodenlänge herrührt.

Der Nennfrequenzbereich der stochastische Kühlung am ESR wird mit 0,9 GHz bis 1,7 GHz angegeben. Allerdings zeigen die Meßdaten bei 400 MeV/u, daß die Shuntimpedanz bei 0,9 GHz schon um über 12 dB und bei 1,7 GHz um über 15 dB abgefallen ist. Bei der BTF-Messung, die auch die Leistungsverstärker mit einbezieht sind es etwa 13 dB und 21 dB. Damit tragen die Ränder praktisch nicht mehr zur Kühlung bei. Realistischer wäre es, ein -6 dB-Band von 1,0 GHz bis 1,5 GHz anzugeben.

Die Welligkeit der Signalverarbeitung am Quadrupol Pick-up ist auffällig hoch. Hier sollte die Ursache gefunden und wenn möglich beseitigt werden.

Die Messung der Umlauffrequenz, die BTF-Kalibration und die BTF-Messung mit dem neuen Programm, bestehend aus eine Web-Applikation mit Python-Backend funktioniert gut. Zwei Details sind noch zu ändern. Zum einen sollte die BTF-Kalibration und Messung am Ende die Quellenleistung verringern um Srahlverluste zu minimieren. Zum anderen sollte beim Resultat der BFT-Messung bei der Angabe „... too long“ oder „... too short“ angegeben derden auf welches Diagramm (0°/180°) sich die Angabe bezieht.

Um die stochastische Kühlung am ESR tatsächlich für andere Strahlenergien als 400 MeV/u zu verwenden wären einige Umbauten und Erweiterungen an der Signalverarbeitung erforderlich. An allen Pick-ups und Kickern müssten die jeweils 4·7 festen Verzögerungsleitungen durch schaltbare oder variable Leitungen ersetzt werden. Am CR sind schaltbarer 8:1-Leistungsteiler für zwei β-Werte vorgesehen (Velocity Switch BETA1). Die Signallaufzeit der vier Kühl-Signalpfade müsste in einem wesentlich größeren Bereich verstellbar werden. Die bisherigen mechanisch verfahrbaren Verzögerungsleitungen („Posaunen“) müssten gegen eine elektronische Lösung ersetzt werden. Wenn ein mitlaufen der stochastischen Kühlung beim Abbremsen möglich sein soll müssten zusätzlich alle verstellbaren Komponenten Rampen fahren können. Das ist zum Beispiel mit den momentanen Abschwächern, auch bei anderer Ansteuerung nicht möglich. Sie koppeln beim Umschalten Spannungsspitzen in das HF-Signal ein. Für den CR wurde ein geeigneter Abschwächer (Phase Constant Attenuator ATTEN1) entwickelt.

6 Igor Pro Programme

Die Programme zur Messung an der Quadrupol Pick-up-Station und zur Auswertung sind aus historischen Gründen in Igor Pro Script geschrieben. Beim Programm Igor Pro werden alle Daten, Scripte und offene Fenster in einer komprimierten Experimentdatei (`*.pxp`) gespeichert. Sie enthalten auch eine Historie der aufgerufenen Funktionen mit Zeitstempeln.

6.1 Shuntimpedanz-Messung

Die Experimentdatei `2025-07-10 Messung.pxp` mit den Messungen der Shuntimpedanzen befindet sich im Verzeichnis `Messung QuPU Spektren`. Der Quelltext in der Experiment-Datei enthält neben ein paar Steueranweisungen nur ein include für das eigentliche Meßprogramm und zwei Vorlagen als Kommentare. Er sieht folgendermaßen aus:

```
#pragma TextEncoding = "UTF-8"
#pragma rtGlobals=3

#include ":Messprogramm"

eExpInit()

// eExpFindFRev(1e9, 1.5e9, 1.97e6)
// eExpMsSPecs("mod4_###Mev", 0.5e9, 2.0e9, 200e3, fRev, 5, ###e7)
```

6.2 Meßprogramm

Das eigentliche Meßprogramm `Messprogramm.ipf` liegt ebenfalls im Ordner `Messung QuPU Spektren` und ist in Unterabschnitt 6.2.3 aufgelistet.

Es enthält zwei Steueranweisungen und ein include des Moduls `Agilent_MXA_1.05.ipf` (siehe Unterabschnitt 6.4.1) zur Ansteuerung des Spektrumanalysators samt Initialisierung. Dahinter folgen einige Parameter, die den Messungen gemeinsam waren.
Es gibt zwei Funktionen für die Messungen am Pick-up:

Name	Aufgabe	Beschreibung
`eExpMsSpecs()`	misst eine Serie von Spektren	Unterabschnitt 6.2.1
`eExpFindFRev()`	misst die Umlauffrequenz	Unterabschnitt 6.2.2

6.2.1 Funktion eExpMsSpecs()

Die Funktion **`eExpMsSpecs()`** nimmt eine Reihe von Spektren um die Harmonischen der angegebenen Umlauffrequenz `fRev` auf. Es werden nur Harmonische im Bereich `fStart` bis `fStop` gemessen. Der Parameter `nStep` gibt den Abstand zwischen den Mittenfrequenzen der Spektren in Einheiten von `fRev` an. Der Parameter `fSpan` ist die Frequenzbreite der einzelnen Spektren. Der Parameter `nIons` ist die Anzahl der Ionen beim Start der Meßreihe. Die Spektren werden in einem Datenordner abgelegt. Der Parameter `folder` ist dessen Name ohne `root:`. Die einzelnen Spektren bekommen fortlaufend nummerierte Namen `rawDBm`*nnn*. Die Auflösungsbandbreite, Videobandbreite und Referenzleistung werden durch die Konstanten am Anfang des Meßprogramms

`Messprogramm.ipf` bestimmt. Der Filtertyp wird auf Gaussian, Die Bandbreitenangabe auf Noise und der Detektor wird auf Average eingestellt. Der Vorverstärker wird eingeschaltet. Die Messung erfolgt im FFT-Modus.

6.2.2 Funktion eExpFindFRev()

Die Funktion **`eExpFindFRev()`** versucht im Bereich `fStart` bis `fStop` die genaue Umlauffrequenz zu ermitteln. Die Funktion geht dazu Schrittweise vor. Als erstes wird ein breites Spektrum am Anfang des Frequenzbereichs aufgenommen und geglättet. Das Maximum ist die erste betrachtete Harmonische. Als nächstes wird mit einem breitem Spektrum die nächste Harmonische bestimmt. Hierbei wird bei der Suche zunächst die angegebene Umlauffrequenz angenommen. Aus den beiden Harmonischen wird eine genauere Umlauffrequenz bestimmt. Die zweite Messung wird dann mit jeweils verdoppeltem Harmonischen-Abstand so lange wiederholt bis die Endfrequenz erreicht ist. Zum Schluss werden um die erste und letzte Harmonische mit kleiner Frequenzbreite und Auflösungsbandbreite die genauen Frequenzen bestimmt und daraus die Umlauffrequenz berechnet. Das Ergebnis wird ausgegeben und zusätzlich unter `root:fRev` abgelegt.

6.2.3 Quelltext Messprogramm.ipf

Die Datei `Messprogramm.ipf` im Ordner `Messung QuPU Spektren` enthält das eigentliche Meßprogramm.

```
#pragma TextEncoding = "UTF-8"
#pragma rtGlobals=3

#include "Agilent_MXA_1.05"

function eExpInit()
    mxaInit()
end

// -----------------
// Konstanten
// -----------------

                                        // gemeinsame Meßparameter:
constant eExpDeltaF = 500           // Punktabstand [Hz]
constant eExpRBW = 10.6          // Auflösungsbandbreite [Hz]
constant eExpVBW = 100              // Video-Bandbreite [Hz]
constant eExpRefLvl= -50            // Referenz-Level [dBm]

strConstant eExpGpibSA = "GPIB0::14::INSTR"     // Spektrumanalysator Agilent MXA N9020A

// ---------------------
// Meßfunktionen
// ---------------------

// eExpMsSpecs() nimmt eine Reihe von Spektren um die Harmonischen der Umlauffrequenz auf.
// Gemessen wird um jede nStep-te Vielfache von fRev, die innerhalb von fStart bis fStop liegt.
// Eingang: folder: Name der Data Folders ohne "root:"
//              fStart: Startfrequenz [Hz]
//              fStop:      Endfrequenz [Hz]
//              fSpan:      Frequenzbreite der einzelnen Spektren [Hz]
//              fRev:       Umlauffrequenz [Hz]
//              nStep:      Abstand zwischen den Mittelfrequenzen der Spektren [fRev]
//              nIons:      Anzahl der Ionen
function eExpMsSpecs(folder, fStart, fStop, fSpan, fRev, nStep, nIons)
    string folder
    variable fStart, fStop, fSpan, fRev, nStep, nIons

    printf "eExpMsSPecs(\"%s\", fStart=%.6f GHz, fStop=%.6f GHz, fSpan=%.3W1PHz, fRev=%.8f MHz, nStep=%d, nIons=%.1e)\r", folder, fStart/ 1e9, fStop / 1e9, fSpan, fRev / 1e6, nStep, nIons
    // Spektrumanalysator öffnen
    print "  Spektrumanalysator öffnen"; doUpdate
```

```
42      mxaOpen(eExpGpibSA)
43      mxaSetPref()
44      // neuen Folder erzeugen
45      killDataFolder /Z $("root:" + folder)
46      string oldDF = getDataFolder(1)
47      newDataFolder /S $("root:" + folder)
48      // Parameter ablegen
49      newDataFolder parameter
50      variable /G :parameter:fStart = fStart
51      variable /G :parameter:fStop = fStop
52      variable /G :parameter:fSpan = fSpan
53      variable /G :parameter:fRev = fRev
54      variable /G :parameter:nStep = nStep
55      variable /G :parameter:deltaF = eExpDeltaF
56      variable /G :parameter:rBW = eExpRBW
57      variable /G :parameter:vBW = eExpVBW
58      variable /G :parameter:refLvl = eExpRefLvl
59      variable /G :parameter:nIons = nIons
60      // für alle Frequenzen
61      string wName
62      variable msNum = 0, nDots = 0, fMeas
63      for (fMeas = ceil(fStart / fRev) * fRev; fMeas <= fStop; fMeas += nStep * fRev)
64          // Wave für Spektrum erzeugen
65          variable fSL = fMeas - fSpan/2                  // Startfrequenz [Hz]
66          variable fSH = fMeas + fSpan/2                  // Stopfrequenz [Hz]
67          variable sPts = round((fSH - fSL) / eExpDeltaF) + 1 // Anzahl der Punkte
68          sprintf wName, "root:%s:rawDBm%03d", folder, msNum
69          make /D /N=(sPts) $wName
70          setScale /I x fSL, fSH, "Hz", $wName
71          setScale d 0, 0, "dBm", $wName
72          wave dBm =  $wName
73          // Spektrum aufnehmen
74          if (msNum == 0)
75              printf "␣␣erstes␣Spektrum␣aufnehmen:␣%.6f␣...␣%.6f␣GHz,␣%d␣Punkte,␣
                    Auflösungsbandbreite␣%.1W1PHz\r", fSL / 1e9, fSH / 1e9, sPts, eExpRBW;
                    doUpdate
76          else
77              if (nDots == 0)
78                  printf "␣␣"
79              endif
80              printf "."
81              if (nDots++ >= 70)
82                  printf "\r"
83                  nDots = 0
84              endif
85              doUpdate
86              mxaAlert()
87          endif
88          mxaSweep(dBm, eExpRefLvl, eExpRBW, VidBW=eExpVBW, filtType="GaussNoise", detType="
                AVERAGE", fFTWidth=7.99e6, preAmp="LOW")
89          msNum++
90      endFor
91     if (nDots != 0)
92          printf "\r"
93      endif
94      // Spektrumanalysator schließen
95      print "␣␣Spektrumanalysator␣schließen"; doUpdate
96      mxaClose()
97      // fertig
98      variable secs = dateTime
99      printf "␣␣fertig,␣%s,␣%s\r", replaceString("/", secs2date(secs, -1)[0,9], "."), secs2time
            (secs, 3)
100
101     setDataFolder oldDF
102 end
103
104
105 // eExpFindFRev() versucht im Bereich fStart bis fStop die Umlauffrequenz genau zu ermitteln.

106 // Eingang: fStart:     Startfrequenz [Hz]
107 //              fStop:          Endfrequenz [Hz]
108 //              typFRev:        geschätzte Umlauffrequenz
109 // Ausgang: root:fRev:  Umlauffrequenz [Hz]
110 function eExpFindFRev(fStart, fStop, typFRev)
111     variable fStart, fStop, typFRev
112
```

```
113     printf "eExpFindFRev(fStart=%.6f␣GHz,␣fStop=%.6f␣GHz)\r", fStart/ 1e9, fStop / 1e9
114     //Parameter
115     variable deltaF = 2e3       // Punktabstand für erste Suche [Hz]
116     // Spektrumanalysator öffnen
117     print "␣␣Spektrumanalysator␣öffnen"; doUpdate
118     mxaOpen(eExpGpibSA)
119     mxaSetPref()
120     // temporären Folder erzeugen
121     string oldDF = getDataFolder(1)
122     newDataFolder /O /S root:temp
123     // 1. Spektrum aufnehmen
124     variable badSNR = 0
125     variable fSL = fStart                                   // Startfrequenz
126     variable fSH = fStart + 1.25 * typFRev              // Stopfrequenz
127     variable sPts = round((fSH - fSL) / deltaF) + 1 // Anzahl der Punkte
128     variable rBW = deltaF / 10                          // Auflösungsbandbreite
129     variable smthCoarse = 11                                // Smooth-Faktor für grobe
            Messungen
130     variable smthFine = 1001                                // Smooth-Faktor für feine
            Messungen
131     variable minSnrCoarse = 12                          // minimales SNR für grobe Messungen
            [dB]
132     variable minSnrFine = 8                             // minimales SNR für feine Messungen
            [dB]
133 //  kalt: variable fineSpan = 100e3                     // Meßbreite für verfeinerte Messung
        [Hz]
134 // heiß:    variable fineSpan = 1.75e6                 // Meßbreite für verfeinerte Messung
        [Hz]
135     variable fineSpan = 500e3                               // Meßbreite für verfeinerte
            Messung [Hz]
136     make /O /D /N=(sPts) wF0
137     setScale /I x fSL, fSH, "Hz", wF0
138     setScale d 0, 0, "dBm", wF0
139     wave wF0
140     printf "␣␣Spektrum␣aufnehmen:␣%.6f␣...␣%.6f␣GHz,␣%d␣Punkte,␣Auflösungsbandbreite␣%.0W1PHz
            :␣", fSL / 1e9, fSH / 1e9, sPts, rBW; doUpdate
141     mxaSweep(wF0, eExpRefLvl, rBW, detType="AVERAGE", preAmp="LOW")
142     // glätten (Box Averaging)
143     duplicate /O wF0, wF0s
144     smooth /B smthCoarse, wF0s
145     // Maximum suchen
146     waveStats /Q wF0s
147     variable snr = V_max - V_avg
148     if (snr < minSnrCoarse)
149         printf "SNR␣schlecht:␣"
150         badSNR = 1
151     endif
152     variable fF0 = V_maxloc
153     printf "f0␣=␣%.9f␣GHz\r", fF0 / 1e9
154     // weitere Maxima finden
155    variable fRev = typFRev
156    variable lastRound = 0
157    variable peakNum = 1
158    do
159         // nächstes Maximum suchen
160         fSL = fF0 + peakNum * fRev - 0.75 * fRev / 2    // Startfrequenz
161         fSH = fF0 + peakNum * fRev + 0.75 * fRev / 2    // Stopfrequenz
162         sPts = round((fSH - fSL) / deltaF) + 1      // Anzahl der Punkte
163         make /O /D /N=(sPts) wF1
164         setScale /I x fSL, fSH, "Hz", wF1
165         setScale d 0, 0, "dBm", wF1
166         wave wF1
167         printf "␣␣Spektrum␣aufnehmen:␣%.6f␣...␣%.6f␣GHz,␣%d␣Punkte,␣Auflösungsbandbreite␣%.0
                W1PHz:␣", fSL / 1e9, fSH / 1e9, sPts, rBW; doUpdate
168         mxaSweep(wF1, eExpRefLvl, rBW, detType="AVERAGE", fFTWidth=7.99e6, preAmp="LOW")
169         // glätten (Box Averaging)
170         duplicate /O wF1, wF1s
171         smooth /B smthCoarse, wF1s
172         // Maximum suchen
173         waveStats /Q wF1s
174         snr = V_max - V_avg
175         if (snr < minSnrCoarse)
176             printf "SNR␣schlecht:␣"
177             badSNR = 1
178         endif
179         variable fF1 = V_maxloc
```

```
180             fRev = (fF1 - fF0) / peakNum
181             printf "fRev = %.6f MHz\r", fRev / 1e6
182             // nächster Schritt
183             if (lastRound)
184                 break
185             endif
186             peakNum *= 2
187             if (fF0 + peakNum * fRev + 0.75 * fRev / 2 > fStop)
188                 peakNum = floor((fStop - fF0 - 0.75 * fRev / 2) / fRev)
189                 lastRound = 1
190             endif
191         while (1)
192         // linke Linie genauer messen
193         fSL = fF0 - fineSpan / 2    // Startfrequenz
194         fSH = fF0 + fineSpan / 2    // Stopfrequenz
195         sPts = 10001            // Anzahl der Punkte
196         rBW = 10                // Auflösungsbandbreite
197         make /O /D /N=(sPts) wF0a
198         setScale /I x fSL, fSH, "Hz", wF0a
199         setScale d 0, 0, "dBm", wF0a
200         wave wF0a
201         printf "  Spektrum aufnehmen: %.6f ... %.6f GHz, %d Punkte, Auflösungsbandbreite %.0W1PHz
                :", fSL / 1e9, fSH / 1e9, sPts, rBW; doUpdate
202         mxaSweep(wF0a, eExpRefLvl, rBW, detType="AVERAGE", fFTWidth=7.99e6, preAmp="LOW")
203         // glätten (Box Averaging)
204         duplicate /O wF0a, wF0aS
205         smooth /B smthFine, wF0aS
206         // Maximum suchen
207         waveStats /Q wF0aS
208         snr = V_max - V_min
209         if (snr < minSnrFine)
210             printf "SNR schlecht: "
211             badSNR = 1
212         endif
213         variable fF0a = V_maxloc
214         printf "f0 = %.9f GHz\r", fF0a / 1e9
215         // rechte Linie genauer messen
216         fSL = fF1 - fineSpan / 2    // Startfrequenz
217         fSH = fF1 + fineSpan / 2    // Stopfrequenz
218         make /O /D /N=(sPts) wF1a
219         setScale /I x fSL, fSH, "Hz", wF1a
220         setScale d 0, 0, "dBm", wF1a
221         wave wF1a
222         printf "  Spektrum aufnehmen: %.6f ... %.6f GHz, %d Punkte, Auflösungsbandbreite %.0W1PHz
                :", fSL / 1e9, fSH / 1e9, sPts, rBW; doUpdate
223         mxaSweep(wF1a, eExpRefLvl, rBW, detType="AVERAGE", fFTWidth=7.99e6, preAmp="LOW")
224         // glätten (Box Averaging)
225         duplicate /O wF1a, wF1aS
226         smooth /B smthFine, wF1aS
227         // Maximum suchen
228         waveStats /Q wF1aS
229         snr = V_max - V_min
230         if (snr < minSnrFine)
231             printf "SNR schlecht: "
232             badSNR = 1
233         endif
234         variable fF1a = V_maxloc
235         printf "f1 = %.9f GHz\r", fF1a / 1e9
236         // Spektrumanalysator schließen
237         print "  Spektrumanalysator schließen"; doUpdate
238         mxaClose()
239         // Umlauffrequenz aufgeben
240         variable /G root:fRev = (fF1a - fF0a) / peakNum
241         nVar gFRev = root:fRev
242         if (badSNR)
243             printf "  SNR schlecht: "
244         else
245             printf "  "
246         endif
247         printf "Umlauffrequenz fRev = %.6f MHz\r", gFRev / 1e6
248         // fertig
249         variable secs = dateTime
250         printf "  fertig, %s, %s\r", replaceString("/", secs2date(secs, -1)[0,9], "."), secs2time
                (secs, 3)
251
252         setDataFolder oldDF
```

```
253 end
```

6.3 Auswertungsprogramm

Das Auswertungsprogramm liest die Daten aus den Experimentdateien und den Touchstone-Dateien der Web-Applikation, führt Berechnungen durch und erzeugt daraus Diagramme. Der Quelltext in der Experiment-Datei enthält neben ein paar Steueranweisungen nur ein include für das eigentliche Auswertungsprogramm. Er sieht folgendermaßen aus:

```
1 %#pragma TextEncoding = "UTF-8"
2 %#pragma rtGlobals=3
3
4 %#include ":Auswertung"
```

Das eigentliche Auswertungsprogramm `Auswertung.ipf` liegt im Ordner `Auswertung` und ist in Unterabschnitt 6.3.15 aufgelistet. Das Programm ist in sechs Abschnitte unterteilt. Der Abschnitt „Definitionen“ enthält die Dateisystempfade zu den Messungen, die Ionensorte, Frequenzbereiche, die Nummern der LNAs um Quadrupol Pick-up und Daten der Kicker-Elektroden. Zusätzlich enthält sie die Funktion **`setFldrNames()`**, die, wie in Abschnitt 3.3 aufgelistet, die Zuordnung der Namen der Messungen zu den Energien definiert.

6.3.1 Auswertungsprogramm-Abschnitt „Funktionen zum laden der Meßdaten“

In diesem Abschnitt des Auswertungsprogramms sind Funktionen zum laden der Meßdaten:

Name	Aufgabe	Beschreibung
`loadSPar()`	S-Par. Superelektrode → Spektrumanalysator laden	Unterabschnitt 6.3.2
`loadMsSa()`	Messungen des Spektrumanalysators laden	Unterabschnitt 6.3.3
`loadBTF()`	BTF Messungen laden	Unterabschnitt 6.3.4

6.3.2 Funktion loadSPar()

Die Funktion **`loadSPar()`** läd die S-Parameter-Messungen Superelektrode zu Spektrumanalysator aus den Experimentdateien aus den Verzeichnis `Messung QuPU S-Parameter`. Die Messungen von jeder Superelektrode zum Spektrumanalysator werden in den Datenordner `root:sPar` gespeichert. Der Wave-Name ist jeweils Name des LNAs, der an der Superelektrode hängt. Zum Beispiel enthält die Daten-Wave `root:sPar:V23` die komplexe Transmission (S_{21}) vom Eingang des LNAs V23 zum Spektrumanalysator. Aus den komplexen Transmissionen werden zusätzlich der Betrag linear (`...mag`) und als dB (`...dB`) und die Phase (`...pha`) berechnet.

6.3.3 Funktion loadMsSa()

Die Funktion **`loadMsSa()`** läd die Spektren der Messungen des Pick-up Moduls bei allen Energien aus der Experimentdatei `Messung QuPU Spektren/2025-07-10 Messung.pxp`. Für jede Energie wird ein einzelner Datenordner angelegt. Im Namen des Datenordners ist Energie in MeV kodiert. Zum Beispiel enthält der Datenordner `root:sa400` die Spektren bei 400 MeV/u. Die Spektren sind nach dem Schema `rawDBm`*nnn* ab 000 durchnummeriert. Zusätzlich werden in dem Datenordner noch die Meßparameter des Spektrumanalysators (Datenordner `parameter`) gespeichert. Darin wird auch die Anzahl der Ionen beim Start (Variable `nIons`) und am Ende (Variable `nIonsEnd`) der Meßreihe abgelegt.

6.3.4 Funktion loadBTF()

Die Funktion **`loadBTF()`** läd die BTF-Messungen bei allen Energien. Die BTF-Messungen werden von der Web-Applikation als S-Parameter im Touchstone-Format abgelegt. Alle Dateien befinden sich in Unterverzeichnissen von `esrsc01/log/2025-07-10`. Die Verzeichnisnamen werden in der Funktion **`setFldrNames()`** definiert. Für jede Energie wird ein einzelner Datenordner angelegt. Im Namen des Datenordners ist Energie in MeV kodiert. Zum Beispiel enthält der Datenordner `root:btf400` die BTF-Messung bei 400 MeV/u. Die Touchstone-Dateien `S21.s2p` enthalten, Format-bedingt neben den BTF-Daten S_{21} auch noch die anderen drei S-Parameter. Diese enthalten keine Daten und werden verworfen.

6.3.5 Auswertungsprogramm-Abschnitt „Funktionen zur Auswertung“

In diesem Abschnitt des Auswertungsprogramms sind Funktionen zur Auswertung der geladenen Meßdaten.

Name	Aufgabe	Beschreibung
`phaCorrLin0180()`	Berechnet nicht-lineare Phase	Unterabschnitt 6.3.6
`calcSPar()`	Berechnet Mittelwert der S-Parameter	Unterabschnitt 6.3.7
`calcMsSa()`	Ber. Spektrumanalysator-Messungen	Unterabschnitt 6.3.8
`calcRsTn()`	Ber. Shuntimpedanz, Rauschtemperatur	Unterabschnitt 6.3.9
`calcTnMean()`	Ber. Mittelwert Rauschtemperatur	Unterabschnitt 6.3.10
`calcBTF()`	Ber. BTF Messungen für ein Kicker-Modul	Unterabschnitt 6.3.11

6.3.6 Funktion phaCorrLin0180()

Die Funktion **`phaCorrLin0180()`** korrigiert Phasensprünge ($|\Delta\varphi| > 180°$) durch Addition ganzer Vielfacher von 360° und beseitigt den Phasen-linearen und konstanten Anteil der gesamten Wave durch Subtraktion einer affin linearen Funktion mit kleinster Fehlerquadratsumme im angegebenen Frequenzbereich. Als konstanter Anteil wird nur 0° oder 180° verwendet.

Die Funktion beseitigt zuerst die Phasensprünge mittels der Funktion **`phaCorr()`**. Dann wird eine affin-lineare Regression berechnet. Die berechnete beliebige Konstante wird auf Vielfache von 180° gerundet. Unter Berücksichtigung der gerundeten Konstante wird als nächstes eine lineare Regression berechnet. Die lineare Funktion wird von der Phase abgezogen. Der Linearitätsfaktor und die Konstante Modulo 360° werden zurückgegeben.

6.3.7 Funktion calcSPar()

Die Funktion **`calcSPar()`** berechnet den Mittelwert des Betrags der Transmissionen von den vier Superelektroden zum Spektrumanalysator. Das Ergebnis wird unter `root:sPar:avgMag` und `root:sPar:avgDB` abgelegt.

6.3.8 Funktion calcMsSa()

Die Funktion **`calcMsSa()`** berechnet zunächst aus allen Spektren aller Energien `rawDBm`*nnn* die korrigierten Spektren `corrDBm`*nnn*. Dazu wird die mittlere Transmission der Signalverarbeitung von der Superelektrode zum Spektrumanalysator herausgerechnet. Für jedes korrigierte Spektrum wird noch die spektrale Leistungsdichte `pDens`*nnn* berechnet.

Aus den Leistungsdichte-Spektren bei unterschiedlichen Frequenzen wird die Linienleistung über die Frequenz linear (`power`) und in dBm (`powerDBm`) berechnet. Zur Bestimmung der Linienleistung wird nach dem Trapez-Verfahren ein Bereich um das Maximum integriert. Die Integrationsbereich umfasst die dreifache Halbwertbreite. Aus dem Mittelwert des Rauschens an den Rändern der Spektren wird eine mittlere Rauschleistungsdichte (`noiseDens`) berechnet. Hierzu wird um die Linie die vierfache Halbwertbreite ausgespart.

6.3.9 Funktion calcRsTn()

Die Funktion **`calcRsTn()`** berechnet für alle Energien aus in der Funktion **`calcMsSa()`** (Unterabschnitt 6.3.8) berechneten Linienleistungen (`power`) die Shuntimpedanz (`ZPU`) über die Frequenz und die mittlere Rauschtemperatur (`TN`). Die Berechnung der Pick-up-Shuntimpedanz in Circuit Convention erfolgt nach der Formel

$$\mathrm{ZPU} = \frac{\mathrm{power}}{2 \cdot \mathrm{N} \cdot (\mathrm{qi} \cdot \mathrm{fRev})^2}. \tag{6.1}$$

Die Ladung eines ^{238}U 92+ Ions qi ist 92 mal die Elementarladung qe. Die Umlauffrequenz fRev wird aus den Parametern der einzelnen Messung gelesen. Für die Teilchenzahl wird ein exponentieller Abfall während der Meßreihe angenommen:

$$\mathrm{N} = \mathrm{nIons} \cdot \left(\frac{\mathrm{nIonsEnd}}{\mathrm{nIons}} \right)^{\frac{\mathrm{sNum}}{\mathrm{numS}-1}}. \tag{6.2}$$

Die Anzahl der Ionen nIons beim Start und nIonsEnd an Ende der Meßreihe werden aus den Parametern der einzelnen Messung gelesen. Der Index sNum ist die Nummer des Spektrums von 0 bis zur Anzahl der Spektren numS − 1. Von der Shuntimpedanz (`ZPU`) wird noch durch zwei Runden gleitender Mittelwert mit Fensterbreite 6 eine geglättete Shuntimpedanz (`ZPUSmooth`) berechnet.

Zur Berechnung der Rauschtemperatur `TN` wird die mittlere Rauschleistungsdichte `noiseDens` aus der Funktion **`calcMsSa()`** (Unterabschnitt 6.3.8) durch die Boltzmann-Konstante kb geteilt. Hiervon wird noch durch zwei Runden gleitender Mittelwert mit Fensterbreite 4 eine geglättete Rauschtemperatur (`TNSmooth`) berechnet.

6.3.10 Funktion calcTnMean()

Die Funktion **`calcTnMean()`** berechnet den Mittelwert der Rauschtemperatur aus den Messungen bei allen Energien. Er wird unter `root:tnMean:TN` abgelegt. Hierbei wird berücksichtigt, daß es nicht bei jeder Energie für jeden Frequenzpunkt Meßwerte gibt. Wenn die vierfache Halbwertbreite des jeweiligen Leistungsdichte-Spektrums bis über den gemessenen Frequenzbereich geht, dann kann keine Rauschtemperatur `TN` berechnet werden. Der Wert ist dann mit NaN (`numtype(T) == 2`) gekennzeichnet und wird ignoriert.

6.3.11 Funktion calcBTF()

Die Funktion **`calcBTF()`** berechnet für die BTF Messungen aller Energien die Transmission für nur ein aktives Kicker-Modul, korrigiert um die Teilchenzahl. Vorher wird noch für die spätere Darstellung eine Phasenkorrektur für Rohdaten mit der Funktion **`phaCorrLin0180()`** (sie Unterabschnitt 6.3.6) durchgeführt.

Als nächstes wird die Anzahl der Teilchen N beim Start der BTF-Messung extrapoliert. Der Abfall während der BTF-Messung wirde vernachlässigt. Hierbei wird angenommen, daß sich die Teilchenzahl nach den Meßreihen der Spektren mit der gleichen Basis exponentiell weiter verringert:

$$\mathrm{N} = \mathrm{N}_{\mathrm{SaEnd}} \cdot \left(\frac{\mathrm{N}_{\mathrm{SaEnd}}}{\mathrm{N}_{\mathrm{SaStart}}} \right)^{\frac{\mathrm{T}_{\mathrm{BtfStart}} - \mathrm{T}_{\mathrm{SaEnd}}}{\mathrm{T}_{\mathrm{SaEnd}} - \mathrm{T}_{\mathrm{SaStart}}}}. \tag{6.3}$$

Hierbei ist $T_{SaStart}$ der Startzeitpunkt und T_{SaEnd} der Endzeitpunkt der Meßreihen der Spektren, $N_{SaStart}$ und N_{SaEnd} sind die zugehörigen Teilchenzahlen. $T_{BtfStart}$ ist der Startzeitpunkt der BTF-Messung. Mit der Teilchenzahl N werden alle anderen Messungen auf die Teilchenzahl der 400 MeV/u Messung normiert.

Die Umrechnung von zwei gemessenen Kicker-Modulen auf ein einzelnes Modul geschieht mit den Gleichungen 4.1 bis 4.3 aus Abschnitt 4.5. Die umgerechneten Daten werden als `S21corr` abgelegt. Hieraus werden zusätzlich der Betrag linear (...`mag`) und als dB (...`dB`) und die Phase (...`pha`) berechnet.

Für die spätere Darstellung als Diagramm wird noch eine Phasenkorrektur mit der Funktion **`phaCorrLin0180()`** (sie Unterabschnitt 6.3.6) durchgeführt. Vom Betrag in dB `S21corrDB` und der nicht-linearen Phase `S21corrNLPha` werden noch die geglätteten Waves `S21corrSmoothDB` und `S21corrSmoothNLPha` erzeugt. Hierzu werden zwei Runden gleitender Mittelwert mit Fensterbreite 6 angewandt.

6.3.12 Auswertungsprogramm-Abschnitt „Funktionen zur Darstellung“

In diesem Abschnitt des Auswertungsprogramms sind Funktionen zur Darstellung der Meßdaten als Diagramme.

Name	Aufgabe
`drawFreqRange()`	Hilfsfunktion zeichnet äußere Frequenzbereiche
`plotTransElSa()`	Diagramm Transmission Elektroden → Spektrumanalysator
`plotRsTN()`	Diagramm Shuntimpedanz oder Rauschtemperatur
`plotBTF()`	Diagramm BTF-Messungen
`plotAnSchottky()`	Diagramm Schottky Spektren für Abbildung 4.2
`plotAnElSa()`	Diagramm Superelektrode → Spektrumanalysator für Abbildung 4.2
`plotAnTN()`	Diagramm Rauschtemperatur für Abbildung 4.2
`plotAnRs()`	Diagramm Shuntimpedanz für Abbildung 4.2

Die drei Funktionen **`plotTransElSa()`**, **`plotRsTN()`** und **`plotBTF()`** können jeweils mehrere unterschiedliche Diagramme erzeugen. Der Parameter `detail` steuert, ob nur der Nennfrequenzbereich oder alles dargestellt werden soll und `smth`, ob geglättete Daten verwendet werden sollen. Der Parameter `noise` bei **`plotRsTN()`** wählt Rauschtemperatur oder Shuntimpedanz aus, `corr` bei **`plotBTF()`** wählt korrigierte oder Rohdaten.

6.3.13 Auswertungsprogramm-Abschnitt „Funktionen zum Export der Diagramme“

In diesem Abschnitt des Auswertungsprogramms sind Funktionen die jeweils alle Diagramme des jeweiligen Typs darstellen und sie als SVG-Dateien exportieren.

Name	Aufgabe
`saveTransElSa()`	Diagramme Transmission Elektroden → Spektrumanalysator
`saveRsTN()`	Diagramme Shuntimpedanz und Rauschtemperatur
`saveBTF()`	Diagramme BTF Messungen
`saveAnal()`	kleine Diagramme für Abbildung 4.2

6.3.14 Auswertungsprogramm-Abschnitt „Hauptprogramm“

Dieser Abschnitt enthält nur die Funktion **doIt()**. Sie löscht zunächst alle alten Daten in der Experimentdatei des Auswertungsprogramm. Dann lädt sie alle Meßdaten, wertet sie aus, stellt sie dar und exportiert die Diagramme.

6.3.15 Quelltext Auswertung.ipf

Die Datei `Auswertung.ipf` im Ordner `Auswertung` enthält das eigentliche Auswertungsprogramm.

```
#pragma TextEncoding = "UTF-8"
#pragma rtGlobals=3

#include "msUtils_1.06"
#include "graphUtils_1.01"
#include "touchstone_1.06"

//--------------
// Definitionen
//--------------

// Pfade zu Messungen
strConstant dirMsSPar = "::Messung␣QuPU␣S-Parameter"
strConstant fileMsSa = "::Messung␣QuPU␣Spektren:2025-07-10␣Messung.pxp"
strConstant dirMsBFT = "::esrsc01:log:2025-07-10"

// Ionen 238U92+
constant un = 238
constant qn = 92

// Frequenzbereich
constant fLM = 0.5e9    // Startfrequenz Messung
constant fL = 0.9e9     // Startfrequenz Kühlung
constant fLP = 1.0e9    // Startfrequenz Phasenkorrektur
constant fHP = 1.5e9    // Endfrequenz Phasenkorrektur
constant fH = 1.7e9     // Endfrequenz Kühlung
constant fHM = 2.0e9    // Endfrequenz Messung

// LNA Nummern
constant lnaFirst = 23
constant lnaInc = 8
constant lnaLast = 47

// Kicker-Elektroden
constant dz = 0.108             // Abstand zwischen Superelektroden [m]
constant EnDesign = 450e6       // Design-Energie der Verzögerungsleitungen [eV/u]

strConstant stdFont = "Arial"

// setFldrNames() definiert die Zuordnung der Messungs-Namen zu den Energien
function setFldrNames()
    make /O /D root:energies    = { 200e6,                  250e6,                  300e6,
                        350e6,                  400e6                   }
    make /O /T root:srcFldrs    = { "mod4_200MeV",          "mod4_250MeV_V2",   "mod4_300MeV"
        ,       "mod4_350MeV",      "mod4_400MeV"       }
    make /O /T root:btfMeass    = { "18-28-39_btfMeas", "17-56-58_btfMeas", "15-59-13_btfMeas
        ", "19-14-11_btfMeas", "19-50-54_btfMeas"  }
end

//-----------------------------------
// Funktionen zum laden der Meßdaten
//-----------------------------------

// loadSPar() läd S-Parameter Superelektrode zu Spektrumalanysator
function loadSPar()
    printf "loadSPar():"; doUpdate
    killDataFolder /Z root:sPar
    newDataFolder /S root:sPar
    variable lnaNum
    for (lnaNum = lnaFirst; lnaNum <= lnaLast; lnaNum += lnaInc)
        // Name des LNAs
        string lnaName
```

```
60          sprintf lnaName, "V%d", lnaNum
61          printf "␣%s", lnaName; doUpdate
62          // Dateiname
63          string fName
64          sprintf fName, "%s:V%d-SA.pxp", dirMsSPar, lnaNum
65          // S-Parameter laden und benennen
66          loadData /Q /S="sMatrix" /J="S21" /P=home fName
67          rename S21 $lnaName
68          // abgeleitete Daten berechnen
69          magPha(lnaName)
70          dBPha(lnaName)
71      endfor
72      printf ".\r"; doUpdate
73      setDataFolder root:
74  end
75
76  // loadMsSa() läd Messungen des Spektrumanalysators
77  function loadMsSa()
78      printf "loadMsSa():"; doUpdate
79      // für alle Energien
80      wave energies = root:energies
81      wave /T srcFldrs = root:srcFldrs
82      string dstFldr
83      variable eIdx
84      for (eIdx = 0; eIdx < numpnts(energies); eIdx++)
85          printf "␣%.0W0PeV/u", energies[eIdx]; doUpdate
86          sprintf dstFldr, "root:sa%03d", energies[eIdx] / 1e6
87          killDataFolder /Z $dstFldr
88          newDataFolder /S $dstFldr
89          loadData /Q /S=srcFldrs[eIdx] /R /P=home fileMsSa
90      endfor
91      printf ".\r"; doUpdate
92      setDataFolder root:
93  end
94
95  // loadBTF() läd BTF Messungen
96  function loadBTF()
97      printf "loadBTF():"; doUpdate
98      // für alle Energien
99      wave energies = root:energies
100     wave /T btfMeass = root:btfMeass
101     string srcFile, dstFldr, transName
102     variable eIdx
103     for (eIdx = 0; eIdx < numpnts(energies); eIdx++)
104         printf "␣%.0W0PeV/u", energies[eIdx]; doUpdate
105         sprintf srcFile, "%s:%s:S21.s2p", dirMsBFT, btfMeass[eIdx]
106         sprintf dstFldr, "root:btf%03d", energies[eIdx] / 1e6
107         killDataFolder /Z $dstFldr
108         tsRead(srcFile, dstFldr, 0)
109         setDataFolder $dstFldr
110         wave f, S11, S12, S22
111         killWaves f, S11, S12, S22
112         nVar ports, zLine
113         killVariables ports, zLine
114         sVar comment, filePath, partName
115         killStrings comment, filePath, partName
116     endfor
117     printf ".\r"; doUpdate
118     setDataFolder root:
119 end
120
121 //---------------------------
122 // Funktionen zur Auswertung
123 //---------------------------
124
125 // phaCorrLin0180() korrigiert Phasensprünge (|dphi| > 180°) in der 1D-Wave phi durch
        Addition ganzer
126 // Vielfacher von 360° und beseitigt den phasenlinearen und konstanten Anteil durch
        Subtraktion einer
127 // affin linearen Funktion mit kleinster Fehlerquadratsumme im Bereich x1 bis x2. Als
        konstanter Anteil
128 // wird nur 0° oder 180° verwendet.
129 // Eingang: phi:    Referenz auf die zu korrigierende Wave
130 //              x1:     linker Punkt für Korrekturfunktion (optional, default erster Punkt)
131 //              x2:     rechter Punkt für Korrekturfunktion (optional, default letzter Punkt)
132 //  Ausgang:    pLin:       Steigung der Korrekturfunktion (optional)
```

```
133 //              pConst: Offset der Korrekturfunktion (optional)
134 function phaCorrLin0180(phi, [x1, x2, pLin, pConst])
135     wave phi
136     variable x1, x2, &pLin, &pConst
137 
138     // Phasensprünge beseitigen
139     phaCorr(phi)
140     // nur Punkte >= x1 verwenden
141     variable p1
142     if (paramIsDefault(x1))
143         p1 = 0
144     else
145         p1 = ceil((x1 - leftx(phi)) / deltax(phi))
146     endif
147     x1 = pnt2x(phi, p1)
148     // nur Punkte <= x2 verwenden
149     variable p2
150     if (paramIsDefault(x2))
151         p2 = numpnts(phi) - 1
152     else
153         p2 = floor((x2 - leftx(phi)) / deltax(phi))
154     endif
155     x2 = pnt2x(phi, p2)
156     // mindestens 2 Punkte notwendig
157     variable lin = 0, const = 0
158     variable points = p2 - p1 + 1
159     if (points >= 2)
160         // affin-lineare Regression
161             variable sx = points * (x1 + x2) / 2
162             variable sy = sum(phi, x1, x2)
163             duplicate /O phi, root:tmpPhaCorrLin0180
164             wave tmp = root:tmpPhaCorrLin0180
165             tmp[] = x * x
166             variable sxx = sum(tmp, x1, x2)
167             tmp[] = x * phi
168             variable sxy = sum(tmp, x1, x2)
169             lin = (sxy - sx * sy / points) / (sxx - sx * sx / points)
170             const = sy / points - lin * sx / points
171             // Offset: Vielfache von 180°
172             const = 180.0 * round(const / 180.0)
173             // lineare Regression mit berücksichtigtem Offset
174             tmp[] = x * (phi - const)
175             sxy = sum(tmp, x1, x2)
176             lin = sxy / sxx
177             phi[] -= lin * x + const
178             const = abs(mod(const, 360))
179             killWaves tmp
180     endif
181     // Parameter zurückgeben
182     if (!paramIsDefault(pLin))
183         pLin = lin
184     endif
185     if (!paramIsDefault(pConst))
186         pConst = const
187     endif
188 end
189 
190 // calcSPar() berechnet den Mittelwert des Betrags der Transmission Superelektrode zu
        Spektrumalanysator
191 function calcSPar()
192     printf "calcSPar()\r"; doUpdate
193     setDataFolder root:sPar
194     variable lnaNum
195     for (lnaNum = lnaFirst; lnaNum <= lnaLast; lnaNum += lnaInc)
196         // Name des LNAs
197         string wName
198         sprintf wName, "V%dmag", lnaNum
199         if (lnaNum == lnaFirst)
200             // ersten Transmission kopieren
201             duplicate /O $wName, avgMag
202             wave avgMag = avgMag
203         else
204             // weitere Transmission addieren
205             wave S21 = $wName
206             avgMag += S21
207         endif
```

```
208     endfor
209     avgMag /= 4
210     // dB berechnen
211     sprintf wName, "V%ddB", lnaFirst
212     wave dB = $wName
213     duplicate /O dB, avgDB
214     avgDB[] = 20 * log(avgMag)
215     setDataFolder root:
216 end
217
218 // calcMsSa() berechnet für die Messungen bei allen Energien
219 // die Leistungsdichte, Linienleistung und Rauschleistungsdichte
220 function calcMsSa()
221     printf "calcMsSa():"; doUpdate
222     // für alle Energien
223     wave energies = root:energies
224     string fldr
225     variable eIdx
226     for (eIdx = 0; eIdx < numpnts(energies); eIdx++)
227         printf "␣%.0W0PeV/u", energies[eIdx]; doUpdate
228         sprintf fldr, "root:sa%03d", energies[eIdx] / 1e6
229         setDataFolder $fldr
230         // Transmission Superelektrode zu Spektrumalanysator
231         wave transModSaDB = root:sPar:avgDB
232         // Parameter der Meßreihe
233         nVar fStart = :parameter:fStart
234         nVar fStop = :parameter:fStop
235         nVar fRev = :parameter:fRev
236         nVar nStep = :parameter:nStep
237         nVar rBW = :parameter:rBW
238         // gemessene Frequenzen
239         variable msFStart = ceil(fStart / fRev) * fRev
240         variable msDeltaF = nStep * fRev
241         variable nFreqs = floor((fStop - fStart) / msDeltaF) + 1
242         // Wave für Linienleistung und Rauschleistungsdichte über Frequenz anlegen
243         make /O /D /N=(nFreqs) power = nan, powerDBm = nan, noiseDens = nan, noiseDensDBmHz =
                nan
244         setScale /P x msFStart, msDeltaF, "Hz", power, powerDBm, noiseDens, noiseDensDBmHz
245         setScale d 0, 0, "W", power
246         setScale d 0, 0, "dBm", powerDBm
247         setScale d 0, 0, "W/Hz", noiseDens
248         setScale d 0, 0, "dBm/Hz", noiseDensDBmHz
249         // für alle Frequenzen
250         variable fIdx
251         for (fIdx = 0; fIdx < nFreqs; fIdx++)
252             // Rohdaten
253             string rawDBmName
254             sprintf rawDBmName, "rawDBm%03d", fIdx
255             wave rawDBm = $rawDBmName
256             // Transmission Superelektrode zu Spektrumalanysator herausrechnen
257             string corrDBmName
258             sprintf corrDBmName, "corrDBm%03d", fIdx
259             duplicate /O rawDBm, $corrDBmName
260             wave corrDBm = $corrDBmName
261             corrDBm[] -= transModSaDB(x)
262             // spektrale Leistungsdichte berechnen
263             string pDensName
264             sprintf pDensName, "pDens%03d", fIdx
265             duplicate /O corrDBm, $pDensName
266             wave pDens = $pDensName
267             pDens = 10 ^ (corrDBm / 10) * 1e-3 / rbw
268             setScale d 0, 0, "W/Hz", pDens
269             // Linie finden
270             duplicate /O pDens, root:pDensSmooth
271             wave pDensSmooth = root:pDensSmooth
272             smooth /F 8, pDensSmooth
273             waveStats /Q pDensSmooth
274             variable peakIdx = V_maxRowLoc
275             // Halbwertbreite der Linie bestimmen
276             variable leftIdx
277             for (leftIdx = peakIdx - 1; leftIdx >= 0; leftIdx--)
278                 if (pDensSmooth[leftIdx] < pDensSmooth[peakIdx] / 2)
279                     break
280                 endif
281             endfor
282             variable maxIdx = numpnts(pDensSmooth) - 1
```

```
283             variable rightIdx
284             for (rightIdx = peakIdx + 1; rightIdx <= maxIdx; rightIdx++)
285                 if (pDensSmooth[rightIdx] < pDensSmooth[peakIdx] / 2)
286                     break
287                 endif
288             endfor
289             if (leftIdx >= 0 && rightIdx < numpnts(pDensSmooth))
290                 // Bereich gefunden: für Linie maximal 3-fache Halbwertbreite auswählen
291                 variable leftLineIdx = max(peakIdx - 3 * (peakIdx - leftIdx), 0)
292                 variable rightLineIdx = min(peakIdx + 3 * (rightIdx - peakIdx), maxIdx)
293                 // für Rauschen 4-fache Halbwertbreite aussparen
294                 variable leftNoiseIdx = max(peakIdx - 4 * (peakIdx - leftIdx), 0)
295                 variable rightNoiseIdx = min(peakIdx + 4 * (rightIdx - peakIdx), maxIdx)
296                 // Integration Linie nach Trapez-Verfahren → Linienleistung über Frequenz,
                        linear und dBm
297                 power[fIdx] = area(pDens, pnt2x(pDens, leftIdx), pnt2x(pDens, rightIdx))
298                 powerDBm[fIdx] = 10 * log(1000 * power[fIdx])
299                 // Mittelwert Rauschen → mittlere Rauschleistungsdichte
300                 // pDens ist Kombination von vier unkorrelierten Signalen
301                 noiseDens[fIdx] = 0
302                 if (leftNoiseIdx > 0)
303                     noiseDens[fIdx] = mean(pDens, 0, leftNoiseIdx) / 4
304                 endif
305                 if (rightIdx < maxIdx)
306                     noiseDens[fIdx] = (noiseDens[fIdx] + mean(pDens, rightIdx, maxIdx) / 4) /
                            2
307                 endif
308                 if (noiseDens[fIdx] > 0)
309                     noiseDensDBmHz[fIdx] = 10 * log(1000 * noiseDens[fIdx])
310                 else
311                     noiseDens[fIdx] = nan
312                 endif
313             endif
314         endfor
315     endfor
316     killWaves root:pDensSmooth
317     printf ".\r"; doUpdate
318
319     setDataFolder root:
320 end
321
322 // calcRsTn() berechnet die Shuntimpedanzen und Rauschtemperaturen
323 function calcRsTn()
324     printf "calcRsTn():"; doUpdate
325     // Ladung der Ionen
326     variable qi = qn * qe
327     // für alle Energien
328     wave energies = root:energies
329     string fldr
330     variable eIdx
331     for (eIdx = 0; eIdx < numpnts(energies); eIdx++)
332         printf "␣%.0W0PeV/u", energies[eIdx]; doUpdate
333         sprintf fldr, "root:sa%03d", energies[eIdx] / 1e6
334         setDataFolder $fldr
335         // Parameter der Meßreihe
336         nVar fRev = :parameter:fRev
337         nVar nIons = :parameter:nIons
338         nVar nIonsEnd = :parameter:nIonsEnd
339         wave power
340         // Ergebnis-Waves erzeugen
341         make /O /D /N=(numpnts(power)) ZPU, ZPUSmooth, TN, TNSmooth
342         setScale /P x leftx(power), deltax(power), "Hz", ZPU, ZPUSmooth, TN, TNSmooth
343         setScale d 0, 0, "Ω", ZPU, ZPUSmooth
344         setScale d 0, 0, "K", TN, TNSmooth
345         // Rauschtemperatur berechnen
346         wave noiseDens
347         TN = noiseDens / kb
348         waveStats /Q TN
349         variable /G TNAvg = V_avg
350         TNSmooth = TN
351         smooth /B=2 4, TNSMooth
352         // PU-Shunt-Impedanz berechnen, circuit convention
353         // Teilchenzahl: exponentieller Abfall angenommen:
354         // N = nIons * (nIonsEnd / nIons) ^ (p / (numpnts(power) - 1))
355         variable nB = (nIonsEnd / nIons) ^ (1 / (numpnts(power) - 1))
356         ZPU[] = power / (2 * nIons * nB ^ p * (qi * fRev) ^ 2)
```

```
357         ZPUSmooth = ZPU
358         smooth /B=2 6, ZPUSmooth
359     endfor
360     printf ".\r"; doUpdate
361 
362     setDataFolder root:
363 end
364 
365 // calcTnMean() berechnet den Mittelwert der Rauschtemperatur aus allen Energien
366 function calcTnMean()
367     newDataFolder /O root:tnMean
368     printf "calcRsTn():"; doUpdate
369     // für alle Energien
370     wave energies = root:energies
371     string wName
372     variable eIdx, fIdx, f, T
373     for (eIdx = 0; eIdx < numpnts(energies); eIdx++)
374         printf "␣%.0W0PeV/u", energies[eIdx]; doUpdate
375         sprintf wName, "root:sa%03d:TN", energies[eIdx] / 1e6
376         wave TN = $wName
377         if (eIdx == 0)
378             duplicate /O TN, root:tnMean:TN
379             wave meanTN = root:tnMean:TN
380             make /O /I /N=(numpnts(meanTN)) root:tnMean:N
381             wave N = root:tnMean:N
382             N[] = numtype(meanTN) == 0
383         else
384             for (fIdx = 0; fIdx < numpnts(meanTN); fIdx++)
385                 f = pnt2x(meanTN, fIdx)
386                 if (f >= leftx(TN) && f <= rightx(TN) - deltax(TN))
387                     T = TN(f)
388                     if (numtype(T) == 0)
389                         meanTN[fIdx] += T
390                         N[fIdx] += 1
391                     endif
392                 endif
393             endfor
394         endif
395     endfor
396     meanTN[] /= N
397     waveStats /Q meanTN
398     variable /G root:tnMean:TNAvg = V_avg
399     duplicate /O meanTN, root:tnMean:TNSmooth
400     wave meanTNSmooth = root:tnMean:TNSmooth
401     smooth /B=2 4, meanTNSmooth
402     printf ".\r"; doUpdate
403 end
404 
405 // calcBTF() berechnet für die BTF Messungen bei allen Energien
406 // S-Parameter für ein Kicker-Modul, korrigiert um die Teilchenzahl
407 function calcBTF()
408     printf "calcBTF():"; doUpdate
409     // für alle Energien
410     wave energies = root:energies
411     wave /T btfMeass = root:btfMeass
412     variable nIons400
413     string saFldr, btfFldr, btfMeas
414     variable eIdx
415     for (eIdx = numpnts(energies) - 1; eIdx >= 0; eIdx--)
416         printf "␣%.0W0PeV/u", energies[eIdx]; doUpdate
417         sprintf saFldr, "root:sa%03d", energies[eIdx] / 1e6
418         sprintf btfFldr, "root:btf%03d", energies[eIdx] / 1e6
419         setDataFolder $btfFldr
420 
421         // abgeleitete Daten für Rohdaten berechnen
422         magPha("S21")
423         dBPha("S21")
424 
425         // Phasenkorrektur für Rohdaten
426         wave pha = S21Pha
427         duplicate pha, $(btfFldr + ":S21nLPha")
428         wave nLPha = $(btfFldr + ":S21nLPha")
429         variable lin, const
430         phaCorrLin0180(nLPha, x1=fLP, x2=fHP, pLin=lin, pConst=const)
431         variable /G S21phaLin = lin
432         variable /G S21phaConst = const
```

```
433         // Phasensprünge an Enden wieder herstellen wegen Darstellung
434         nLPha[] = mod(nLPha, 360)
435         nLPha[] = nLPha > 180 ? nLPha - 360 : (nLPha < -180 ? nLPha + 360 : nLPha)
436
437         // Teilchenzahl extrapolieren: Startzeit BTF
438         btfMeas = btfMeass[eIdx]
439         variable btfH, btfM, btfS
440         sscanf btfMeas, "%d-%d-%d_", btfH, btfM, btfS
441         variable btfStart = 3600 * btfH + 60 * btfM + btfS
442         // Zeit der ersten Spektrumanalysator Messung
443         string saWName
444         sprintf saWName, "%s:rawDBm%03d", saFldr, 0
445         wave sStart = $saWName
446         variable saStartDT = modDate(sStart)
447         variable saY, saM, saD
448         sscanf secs2Date(saStartDT, -2), "%d-%d-%d", saY, saM, saD
449         variable saStartD = date2secs(saY, saM, saD)
450         variable saStart = saStartDT - saStartD
451         // Zeit der letzten Spektrumanalysator Messung
452         nVar fStart = $(saFldr + ":parameter:fStart")
453         nVar fStop = $(saFldr + ":parameter:fStop")
454         nVar fRev = $(saFldr + ":parameter:fRev")
455         nVar nStep = $(saFldr + ":parameter:nStep")
456         variable nFreqs = floor((fStop - fStart) / (nStep * fRev)) + 1
457         sprintf saWName, "%s:rawDBm%03d", saFldr, nFreqs - 1
458         wave sStop = $saWName
459         variable saStopDT = modDate(sStop)
460         variable saStop = saStopDT - saStartD
461         // Teilchenzahl: exponentieller Abfall bis zum Start der Messung angenommen,
462         // während der kurzen Meßzeit konstant
463         nVar saIonsStart = $(saFldr + ":parameter:nIons")
464         nVar saIonsStop = $(saFldr + ":parameter:nIonsEnd")
465         variable /G nIons
466         nIons = saIonsStop * ((saIonsStop / saIonsStart) ^ ((btfStart - saStop) / (saStop -
                saStart)))
467         // Teilchenzahl korrigieren
468         wave /C S21
469         duplicate /O S21, $(btfFldr + ":S21corr")
470         wave /C corr = $(btfFldr + ":S21corr")
471         if (energies[eIdx] == 400e6)
472             nIons400 = nIons
473         else
474             corr = S21 * nIons400 / nIons
475         endif
476
477         // Umrechnung von zwei auf ein Kicker-Modul: Teilchengeschwindigkeit
478         variable EKin = un * energies[eIdx] * qe
479         variable m0 = qn * mp + (un - qn) * mn
480         variable /G beta
481         beta = sqrt(1 - 1 / (EKin / (m0 * c0 ^ 2) + 1) ^ 2)
482         // Phasenfehler zweite Superelektrode
483         variable betaDesign = sqrt(1 - 1 / (un * EnDesign * qe / (m0 * c0 ^ 2) + 1) ^ 2)
484         make /O /D /N=(numpnts(S21)) phiErr
485         setScale /P x leftx(S21), deltax(S21), "Hz", phiErr
486         phiErr[] = 2 * pi * x * dz / c0 * (1 / beta - 1 / betaDesign)
487         // relative Spannung der Kombination
488         make /O /C /D /N=(numpnts(S21)) Ur
489         setScale /P x leftx(S21), deltax(S21), "Hz", Ur
490         Ur = sin(phiErr) / (sqrt(2) * sin(phiErr / 2)) * exp(cmplx(0, phiErr / 2))
491         magPha("Ur")
492         rename Urmag, UrMag
493         rename Urpha, UrPha
494         // herausrechnen
495         corr /= Ur
496
497         // abgeleitete Daten für korrigierte Daten berechnen
498         magPha("S21corr")
499         dBPha("S21corr")
500         rename S21corrmag, S21corrMag
501         rename S21corrdB, S21corrDB
502         rename S21corrpha, S21corrPha
503
504         // Phasenkorrektur für korrigierte Daten
505         wave pha = S21corrPha
506         duplicate pha, $(btfFldr + ":S21corrNLPha")
507         wave nLPha = $(btfFldr + ":S21corrNLPha")
```

```
508         phaCorrLin0180(nLPha, x1=fLP, x2=fHP, pLin=lin, pConst=const)
509         variable /G S21corrPhaLin = lin
510         variable /G S21corrPhaConst = const
511         // Phasensprünge an Enden wieder herstellen wegen Darstellung
512         nLPha[] = mod(nLPha, 360)
513         nLPha[] = nLPha > 180 ? nLPha - 360 : (nLPha < -180 ? nLPha + 360 : nLPha)
514
515         // geglättete Waves für dB und nLPha
516         wave dB = S21corrDB
517         duplicate /O dB, $(btfFldr + ":S21corrSmoothDB")
518         wave smthdB = $(btfFldr + ":S21corrSmoothDB")
519         smooth /B=2 6, smthdB
520         wave nLPha = S21corrNLPha
521         duplicate /O nLPha, $(btfFldr + ":S21corrSmoothNLPha")
522         wave smthNLPha = $(btfFldr + ":S21corrSmoothNLPha")
523         smooth /B=2 6, smthNLPha
524     endfor
525     printf ".\r"; doUpdate
526     setDataFolder root:
527 end
528
529 //-----------------------------
530 // Funktionen zur Darstellung
531 //-----------------------------
532
533 // drawFreqRange() zeichnet die Markierung der äußeren Frequenzbereiche ein
534 function drawFreqRange()
535     setDrawLayer userBack
536     setDrawEnv xcoord=bottom, ycoord=prel, linethick=0, fillfgc=(0xEE00, 0xEE00, 0xEE00)
537     drawRect fLM, 0, fL, 1
538     setDrawEnv xcoord=bottom, ycoord=prel, linethick=0, fillfgc=(0xEE00, 0xEE00, 0xEE00)
539     drawRect fH, 0, fHM, 1
540 end
541
542 // plotTransElSa() erstellt ein Diagramm
543 // Transmission Superelektroden zum Spektrumanalysator
544 // Eingang: detail: Frequenzbereich der Kühlung (1) oder alles (0) darstellen
545 function plotTransElSa(detail)
546     variable detail
547
548     printf "plotTransElSa(%d)\r", detail; doUpdate
549     // Parameter
550     variable x0 = 10, y0 = 42
551     string graphWin = "trElSaGraph"
552     if (detail)
553         y0 += 80
554         graphWin += "Detail"
555     endif
556     // Diagramm
557     doWindow /K $graphWin
558     display /W=(x0, y0, x0 + 480.75, y0 + 282) /N=$graphWin /K=1 as "Transmission␣
            Superelektrode␣Spektrumanalysator"
559     modifyGraph margin(left)=42, margin(bottom)=34, margin(top)=8, margin(right)=14
560     modifyGraph width=425.197, height=240.945
561     // Linienstile
562     wave /I color = root:color
563     wave lWidth = root:lWidth
564     wave lStyle = root:lStyle
565     // Graphen für alle LNAs
566     string legTxt = ""
567     string wName, tName, lbl
568     variable lnaNum, tIdx, cIdx
569     for (tIdx = 0; tIdx <= (lnaLast - lnaFirst) / lnaInc + 1; tIdx++)
570         if (tIdx == 4)
571             // Mittelwert
572             wName = "root:sPar:avgDB"
573             tName = "avg"
574             sprintf lbl, "\\s(%s)␣Mittelwert", tName
575             cIdx = 20
576         else
577             // LNA
578             lnaNum = lnaFirst + lnaInc * tIdx
579             sprintf wName, "root:sPar:V%ddB", lnaNum
580             sprintf tName, "v%d", lnaNum
581             sprintf lbl, "\\s(%s)␣V%d␣→␣SA\r", tName, lnaNum
582             cIdx = tIdx + 1
```

```
583         endif
584         wave s21dB = $wName
585         appendToGraph s21dB/TN=$tName
586         modifyGraph rgb($tName) = (color[0][cIdx], color[1][cIdx], color[2][cIdx])
587         modifyGraph lSize($tName) = lWidth[cIdx], lStyle($tName) = lStyle[cIdx]
588         legTxt += lbl
589     endfor
590     reorderTraces v23, {avg}
591     legend /N=trElSwLeg /J /D=0.75 /H=40 /A=RT /X=1 /Y=2 legTxt
592     // Stil
593     modifyGraph axThick=0.75, standoff=0, mirror=2
594     modifyGraph grid=2, gridStyle=5, gridHair=3, gridRGB=(0, 0, 0)
595     modifyGraph gFont=$stdFont, gfSize=10, highTrip=100, useComma=1
596     modifyGraph btLen=4, btThick=0.75, stLen=2, stThick=0.25, ttLen=3, ttThick=0.25, ftLen=3,
            ftThick=0.2
597     modifyGraph lblPosMode=1, lblPos=0
598     modifyGraph manTick(left)={0, 1, 0, 0}, manMinor(left)={0, 50}
599     modifyGraph manTick(bottom)={0, 0.1, 9, 1}, manMinor(bottom)={0, 50}
600     // Achsen
601     if (detail)
602         setAxis bottom fL, fH
603     else
604         setAxis /A /N=0 bottom
605     endif
606     setAxis /A=2 /N=1 left
607     label left "|S\\B21\\M|␣[\\U]"
608     label bottom "f␣[\\U]"
609     // äußereer Frequenzbereiche
610     if (!detail)
611         drawFreqRange()
612     endif
613 end
614
615 // plotRsTN() erstellt ein Diagramm aller Shuntimpedanzen oder
616 //            Rauschtemperaturen
617 // Eingang: noise:      Rauschtemperatur (1) oder Shuntimpedanz (0)
618 //              smth:       geglättete (1) oder ungeglättete (0) Werte
619 //              detail: Frequenzbereich der Kühlung (1) oder alles (0) darstellen
620 function plotRsTN(noise, smth, detail)
621     variable noise, smth, detail
622
623     printf "plotRsTN(noise=%d,␣smth=%d,␣detail=%d)\r", noise, smth, detail; doUpdate
624     // Parameter
625     variable x0 = 310, y0 = 42
626     string wName, graphWin, title
627     if (noise)
628         wName = "TN"
629         graphWin = "tn"
630         title = "Rauschtemperatur"
631         x0 += 300
632     else
633         wName = "ZPU"
634         graphWin = "rs"
635         title = "Shuntimpedanz"
636     endif
637     if (smth)
638         wName += "Smooth"
639         graphWin += "Smth"
640         title += "␣Smooth"
641         y0 += 80
642     endif
643     if (detail)
644         graphWin += "Det"
645         title += "␣Detail"
646         y0 += 40
647     endif
648     graphWin += "Graph"
649     // Diagramm
650     doWindow /K $graphWin
651     display /W=(x0, y0, x0 + 480.75, y0 + 282) /N=$graphWin /K=1 as title
652     modifyGraph margin(left)=42, margin(bottom)=34, margin(top)=8, margin(right)=14
653     modifyGraph width=425.197, height=240.945
654     // Linienstile
655     wave /I color = root:color
656     wave lWidth = root:lWidth
657     wave lStyle = root:lStyle
```

```
658     // Graphen für alle Energien
659     string lbl, legTxt = ""
660     wave energies = root:energies
661     string fldr, valName, tName
662     variable eIdx, cIdx, nEnergies = numpnts(energies)
663     for (eIdx = 0; eIdx <= nEnergies; eIdx++)
664         if (eIdx == nEnergies)
665             if (!noise)
666                 break
667             endif
668             sprintf valName, "root:tnMean:%s", wName
669             tName = "tMean"
670             sprintf lbl, "\r\\s(%s) Mittelwert", tName
671             cIdx = 20
672         else
673             sprintf valName, "root:sa%03d:%s", energies[eIdx] / 1e6, wName
674             sprintf tName, "t%03d", energies[eIdx] / 1e6
675             sprintf lbl, "\r\\s(%s) %.0W0PeV/u", tName, energies[eIdx]
676             cIdx = nEnergies - eIdx - 1
677         endif
678         wave val = $valName
679         appendToGraph val/TN=$tName
680         modifyGraph rgb($tName) = (color[0][cIdx], color[1][cIdx], color[2][cIdx])
681         modifyGraph lSize($tName) = lWidth[cIdx], lStyle($tName) = lStyle[cIdx]
682         legTxt += lbl
683     endfor
684     legend /N=trRsTnLeg /J /D=0.75 /H=40 /A=RT /X=1 /Y=2 legTxt[1,inf]
685     // Stil
686     modifyGraph axThick=0.75, standoff=0, mirror=2
687     modifyGraph grid=2, gridStyle=5, gridHair=3, gridRGB=(0, 0, 0)
688     modifyGraph gFont=$stdFont, gfSize=10, highTrip=100, useComma=1
689     modifyGraph btLen=4, btThick=0.75, stLen=2, stThick=0.25, ttLen=3, ttThick=0.25, ftLen=3,
          ftThick=0.2
690     modifyGraph lblPosMode=1, lblPos=0
691     modifyGraph manTick(bottom)={0, 0.1, 9, 1}, manMinor(bottom)={0, 50}
692     // Achsen
693     if (noise)
694         if (smth)
695             setAxis left 100, 700
696             modifyGraph manTick(left)={0, 50, 0, 0}, manMinor(left)={0, 50}
697         else
698             modifyGraph highTrip(left)=10000
699             setAxis left 0, 1000
700             modifyGraph manTick(left)={0, 100, 0, 0}, manMinor(left)={0, 50}
701         endif
702     else
703         setAxis /A /N=1 /E=1 left
704         modifyGraph manTick(left)={0, 10, 0, 0}, manMinor(left)={0, 50}
705     endif
706     if (detail)
707         setAxis bottom fL, fH
708     else
709         setAxis bottom fLM, fHM
710     endif
711     if (noise)
712         label left "T\\Bnoise\\M [\\U]"
713     else
714         label left "Z\\BPU\\M [\\U]"
715     endif
716     label bottom "f [\\U]"
717     // äußereer Frequenzbereiche
718     if (!detail)
719         drawFreqRange()
720     endif
721 end
722
723 // plotBTF() erstellt ein Diagramm aller BTF-Messungen
724 // Eingang: corr:       korrigierte Daten (1) oder Rohdaten (0)
725 //              smth:       geglättete (1) oder ungeglättete (0) Werte
726 //              detail: Frequenzbereich der Kühlung (1) oder alles (0) darstellen
727 function plotBTF(corr, smth, detail)
728     variable corr, smth, detail
729
730     printf "plotBTF(corr=%d, smth=%d, detail=%d)\r", corr, smth, detail; doUpdate
731     // Parameter
732     variable x0 = 310, y0 = 282
```

```
733     string wName = "S21", vName = "S21", graphWin = "btf", title = "BTF␣Messung"
734     if (corr)
735         wName += "corr"
736         vName += "corr"
737         graphWin += "Corr"
738         title += "␣korrigiert"
739         x0 += 200
740     endif
741     if (smth)
742         wName += "Smooth"
743         graphWin += "Smth"
744         title += "␣Smooth"
745         y0 += 100
746     endif
747     if (detail)
748         graphWin += "Det"
749         title += "␣Detail"
750         y0 += 50
751     endif
752     graphWin += "Graph"
753     // Diagramm
754     doWindow /K $graphWin
755     display /W=(x0, y0, x0 + 480.75, y0 + 395.25) /N=$graphWin /K=1 as title
756     modifyGraph margin(left)=42, margin(bottom)=34, margin(top)=8, margin(right)=14
757     modifyGraph width=425.197, height=354.331
758     // Linienstile
759     wave /I color = root:color
760     wave lWidth = root:lWidth
761     wave lStyle = root:lStyle
762     // Graphen für alle Energien
763     string lbl, legTxt = "\tE\Δtt\φt\\B0\\M"
764     wave energies = root:energies
765     string fldr, basePath, tDBName, tPhaName
766     variable eIdx, cIdx, nEnergies = numpnts(energies)
767     for (eIdx = 0; eIdx < nEnergies; eIdx++)
768         sprintf basePath, "root:btf%03d:", energies[eIdx] / 1e6
769         if (corr || smth)
770             wave dB = $(basePath + wName + "DB"),  pha = $(basePath + wName + "NLPha")
771         else
772             wave dB = $(basePath + wName + "dB"),  pha = $(basePath + wName + "nLPha")
773         endif
774         sprintf tDBName, "t%03ddB", energies[eIdx] / 1e6
775         sprintf tPhaName, "t%03dpha", energies[eIdx] / 1e6
776         appendToGraph dB/TN=$tDBName
777         appendToGraph /L=phase pha/TN=$tPhaName
778         cIdx = nEnergies - eIdx - 1
779         modifyGraph rgb($tDBName) = (color[0][cIdx], color[1][cIdx], color[2][cIdx])
780         modifyGraph lSize($tDBName) = lWidth[cIdx], lStyle($tDBName) = lStyle[cIdx]
781         modifyGraph rgb($tPhaName) = (color[0][cIdx], color[1][cIdx], color[2][cIdx])
782         modifyGraph lSize($tPhaName) = lWidth[cIdx], lStyle($tPhaName) = lStyle[cIdx]
783         nVar lin = $(basePath + vName + "PhaLin")
784         nVar const = $(basePath + vName + "PhaConst")
785         sprintf lbl, "\r\\s(%s)␣%.0W0PeV/u\t%+.3W0Ps\t%.0f°", tDBName, energies[eIdx], lin /
                360, const
786         legTxt += lbl
787     endfor
788     legend /N=trBTFLeg /J /D=0.75 /H=40 /A=MC /X=0 /Y=0 /T={52, 93, 148} legTxt
789     // Stil
790     modifyGraph axThick=0.75, standoff=0, mirror=2, freePos=0
791     modifyGraph zero(phase)=1, zeroThick(phase)=1.25
792     modifyGraph grid=2, gridStyle=5, gridHair=3, gridRGB=(0, 0, 0)
793     modifyGraph gFont=$stdFont, gfSize=10, highTrip=100, useComma=1
794     modifyGraph btLen=4, btThick=0.75, stLen=2, stThick=0.25, ttLen=3, ttThick=0.25, ftLen=3,
          ftThick=0.2
795     modifyGraph lblPosMode=1, lblPos=0
796     modifyGraph manTick(bottom)={0, 0.1, 9, 1}, manMinor(bottom)={0, 50}
797     variable ySplit = 0.5
798     modifyGraph axisEnab(phase)={0, ySplit-0.025}, axisEnab(left)={ySplit+0.025, 1}
799     // Achsen
800     if (detail)
801         if (corr)
802             setAxis left -25, 25
803             modifyGraph manTick(left)={0, 5, 0, 0}, manMinor(left)={0, 50}
804         else
805             setAxis left -30, 30
806             modifyGraph manTick(left)={0, 10, 0, 0}, manMinor(left)={0, 50}
```

```
807         endif
808         setAxis bottom fL, fH
809         setAxis phase -45, 45
810         modifyGraph manTick(phase)={0, 15, 0, 0}, manMinor(phase)={2, 50}
811     else
812         setAxis left -60, 30
813         setAxis bottom fLM, fHM
814         setAxis phase -180, 180
815         modifyGraph manTick(left)={0, 10, 0, 0}, manMinor(left)={0, 50}
816         modifyGraph manTick(phase)={0, 45, 0, 0}, manMinor(phase)={0, 50}
817     endif
818     if (detail)
819     endif
820     label left "|S\\B21\\M|␣[\\U]"
821     label phase "φ(S\\B21\\M)␣+␣Δt·f·360°␣+␣φ\\B0\\M␣[\\U]"
822     label bottom "f␣[\\U]"
823     // Linien zwischen oben und unten
824     setDrawLayer userFront
825     setDrawEnv linethick= 0.75
826     drawLine 0, 1 - ySplit + 0.025, 1, 1 - ySplit + 0.025
827     drawLine 0, 1 - ySplit - 0.025, 1, 1 - ySplit - 0.025
828     // äußereer Frequenzbereiche
829     if (!detail)
830         drawFreqRange()
831     endif
832 end
833
834 // plotAnSchottky() erstellt kleines Diagramm Schottky Spektren
835 function plotAnSchottky()
836     // Diagramm
837     string graphWin = "analSchottkyGraph"
838     doWindow /K $graphWin
839     display /W=(37, 50, 273.25, 136.25) /N=$graphWin /K=1 as "Schottky␣Spektren"
840     modifyGraph margin(left)=36, margin(bottom)=28, margin(top)=5, margin(right)=8
841     modifyGraph width=198.425, height=56.6929
842     // Graphen
843     wave rawDBm0 = root:sa300:corrDBm077
844     wave rawDBm1 = root:sa300:corrDBm078
845     wave rawDBm2 = root:sa300:corrDBm079
846     appendToGraph rawDBm0, rawDBm1, rawDBm2
847     // Stil
848     modifyGraph axThick=0.75, standoff=0, mirror=2, lsize=0.75
849     modifyGraph grid=2, gridStyle=5, gridHair=1, gridRGB=(0xCCCC, 0xCCCC, 0xCCCC)
850     modifyGraph gFont=$stdFont, gfSize=9, highTrip=100, useComma=1
851     modifyGraph btLen=4, btThick=0.75, stLen=2, stThick=0.25, ttLen=3, ttThick=0.25, ftLen=3,
          ftThick=0.2
852     modifyGraph lblPosMode=1, lblPos=0
853     // Achsen
854     setAxis left -160, -120
855     setAxis bottom 1.1942e9, 1.2185e9
856     modifyGraph manTick(left)={0, 10, 0, 0}, manMinor(left)={0, 5}
857     label left "P␣[\\U]"
858     label bottom "f␣[\\U]"
859 end
860
861 // plotAnElSa() erstellt kleines Diagramm Transmission Superelektrode Spektrumanalysator
862 function plotAnElSa()
863     // Diagramm
864     string graphWin = "analElSaGraph"
865     doWindow /K $graphWin
866     display /W=(37, 170, 273.25, 256.25) /N=$graphWin /K=1 as "Transmission␣Superelektrode␣
          Spektrumanalysator"
867     modifyGraph margin(left)=36, margin(bottom)=28, margin(top)=5, margin(right)=8
868     modifyGraph width=198.425, height=56.6929
869     // Graphen
870     wave avgDB = root:sPar:avgDB
871     appendToGraph avgDB
872     // Stil
873     modifyGraph axThick=0.75, standoff=0, mirror=2, lsize=0.75
874     modifyGraph grid=2, gridStyle=5, gridHair=1, gridRGB=(0xCCCC, 0xCCCC, 0xCCCC)
875     modifyGraph gFont=$stdFont, gfSize=9, highTrip=100, useComma=1
876     modifyGraph btLen=4, btThick=0.75, stLen=2, stThick=0.25, ttLen=3, ttThick=0.25, ftLen=3,
          ftThick=0.2
877     modifyGraph lblPosMode=1, lblPos=0
878     // Achsen
879     setAxis left 17.5, 21.5
```

```
880     setAxis bottom fL, fH
881     modifyGraph manTick(left)={0, 1, 0, 0}, manMinor(left)={0, 50}
882     modifyGraph manTick(bottom)={0, 0.1, 9, 1},manMinor(bottom)={0, 50}
883     label left "|S\\B21\\M|␣[\\U]"
884     label bottom "f␣[\\U]"
885 end
886 
887 // plotAnTN() erstellt kleines Diagramm Rauschtemperatur
888 function plotAnTN()
889     // Diagramm
890     string graphWin = "analTNGraph"
891     doWindow /K $graphWin
892     display /W=(37, 290, 273.25, 376.25) /N=$graphWin /K=1 as "Rauschtemperatur"
893     modifyGraph margin(left)=36, margin(bottom)=28, margin(top)=5, margin(right)=8
894     modifyGraph width=198.425, height=56.6929
895     // Graphen
896     wave TN = root:sa300:TN
897     wave TNSMooth = root:sa300:TNSmooth
898     appendToGraph TN, TNSMooth
899     modifyGraph rgb(TN)=(0xFFFF, 0xAAAA, 0)
900     // Stil
901     modifyGraph axThick=0.75, standoff=0, mirror=2, lsize=0.75
902     modifyGraph grid=2, gridStyle=5, gridHair=1, gridRGB=(0xCCCC, 0xCCCC, 0xCCCC)
903     modifyGraph gFont=$stdFont, gfSize=9, highTrip=100, useComma=1
904     modifyGraph btLen=4, btThick=0.75, stLen=2, stThick=0.25, ttLen=3, ttThick=0.25, ftLen=3,
            ftThick=0.2
905     modifyGraph lblPosMode=1, lblPos=0
906     // Achsen
907     setAxis left 100, 600
908     setAxis bottom fL, fH
909     modifyGraph manTick(bottom)={0, 0.1, 9, 1}, manMinor(bottom)={0, 50}
910     label left "T\\Bnoise\\M␣[\\U]"
911     label bottom "f␣[\\U]"
912 end
913 
914 // plotAnRs() erstellt kleines Diagramm Shuntimpedanz
915 function plotAnRs()
916     // Diagramm
917     string graphWin = "analRsGraph"
918     doWindow /K $graphWin
919     display /W=(37, 410, 273.25, 496.25) /N=$graphWin /K=1 as "Shuntimpedanz"
920     modifyGraph margin(left)=36, margin(bottom)=28, margin(top)=5, margin(right)=8
921     modifyGraph width=198.425, height=56.6929
922     // Graphen
923     wave ZPU = root:sa300:ZPU
924     wave ZPUSmooth = root:sa300:ZPUSmooth
925     appendToGraph ZPU, ZPUSmooth
926     modifyGraph rgb(ZPU)=(0xFFFF, 0xAAAA, 0)
927     // Stil
928     modifyGraph axThick=0.75, standoff=0, mirror=2, lsize=0.75
929     modifyGraph grid=2, gridStyle=5, gridHair=1, gridRGB=(0xCCCC, 0xCCCC, 0xCCCC)
930     modifyGraph gFont=$stdFont, gfSize=9, highTrip=100, useComma=1
931     modifyGraph btLen=4, btThick=0.75, stLen=2, stThick=0.25, ttLen=3, ttThick=0.25, ftLen=3,
            ftThick=0.2
932     modifyGraph lblPosMode=1, lblPos=0
933     // Achsen
934     setAxis left 0, 70
935     setAxis bottom fL, fH
936     modifyGraph manTick(bottom)={0, 0.1, 9, 1}, manMinor(bottom)={0, 50}
937     label left "Z\\BPU\\M␣[\\U]"
938     label bottom "f␣[\\U]"
939 end
940 
941 //-------------------------------------
942 // Funktionen zum Export der Diagramme
943 //-------------------------------------
944 
945 // saveTransElSa() erzeugt das Diagramm Transmission Superelektrode
946 // zu Spektrumanalysator und exportiert es als SVG-Dateien
947 function saveTransElSa()
948     variable smth
949     for (smth = 0; smth <= 1; smth++)
950         PlotTransElSa(smth)
951         savePICT /O /E=-9 /P=home
952     endfor
953 end
```

```
954
955 // saveRsTN() erzeugt alle Diagramme Shuntimpedanz und Rauschtemperatur
956 // und exportiert sie als SVG-Dateien
957 function saveRsTN()
958     variable noise, smth, detail
959     for (noise = 0; noise <= 1; noise++)
960         for (smth = 0; smth <= 1; smth++)
961             for (detail = 0; detail <= 1; detail++)
962                 plotRsTN(noise, smth, detail)
963                 savePICT /O /E=-9 /P=home
964             endfor
965         endfor
966     endfor
967 end
968
969 // saveBTF() erzeugt alle Diagramme von BTF Messungen
970 // und exportiert sie als SVG-Dateien
971 function saveBTF()
972     variable corr, smth, detail
973     for (corr = 0; corr <= 1; corr++)
974         for (smth = 0; smth <= 1; smth++)
975             if (corr || !smth)
976                 for (detail = 0; detail <= 1; detail++)
977                     plotBTF(corr, smth, detail)
978                     savePICT /O /E=-9 /P=home
979                 endfor
980             endif
981         endfor
982     endfor
983 end
984
985 // saveAnal() erzeugt alle kleinen Diagramme für
986 // Zeichnung Auswertung und exportiert sie als SVG-Dateien
987 function saveAnal()
988     plotAnSchottky()
989     savePICT /O /E=-9 /P=home
990     plotAnTN()
991     savePICT /O /E=-9 /P=home
992     plotAnRs()
993     savePICT /O /E=-9 /P=home
994     plotAnElSa()
995     savePICT /O /E=-9 /P=home
996 end
997
998 //---------------
999 // Hauptprogramm
1000 //---------------
1001
1002 // doIt() führt alle Schritte zur Auswertung durch
1003 function doIt()
1004     closeAllWins(windowTypes=7)
1005     KillDataFolder root:
1006     setFldrNames()
1007     setLineStyles()
1008     // Transmission Superelektrode zu Spektrumanalysator
1009     loadSPar()      // laden
1010     calcSPar()      // Mittelwert berechnen
1011     saveTransElSa() // alle Diagramme erzeugen und abspeichern
1012     // Messungen Spektrumanalysator
1013     loadMsSa()      // laden
1014     calcMsSa()      // Linienleistung und Rauschleistungsdichte berechnen
1015     // Shuntimpedanzen und Rauschtemperaturen
1016     calcRsTn()      // berechnen
1017     calcTnMean()        // Mittelwert Rauschtemperatur berechnen
1018     saveRsTN()      // alle Diagramme erzeugen und abspeichern
1019     // BTF Messungen
1020     loadBTF()           // laden
1021     calcBTF()           // Teilchenzahl-korrigierte S-Par für 1 Kicker-Modul berechnen
1022     saveBTF()           // alle Diagramme erzeugen und abspeichern
1023     // Diagramme für Zeichnung Auswertung
1024     saveAnal()
1025 end
```

6.4 Include-Dateien

Das Meßprogramm und das Auswertungsprogramm verwenden einige Include-Dateien mit Meß- und Hilfsfunktionen. Diese werden im Folgenden kurz vorgestellt.

6.4.1 Quelltext Agilent_MXA_1.05.ipf

Die Datei `Quelltext Agilent_MXA_1.05.ipf` in den gemeinsamen Igor Pro includes enthält Funktionen zur Ansteuerung des Spektrumanalysators Agilent MXA N9020A oder Keysight MXA N9020B.

```
//-------------------------------------------------------------------------------
// Modul zur Ansteuerung des Spektrum Analysers Agilent MXA N9020A
// Initialisierung bei Programmstart: mxaInit()
//
// (c) 2016,2021,2025 Claudius Peschke
// Gesellschaft für Schwerionenforschung mbH
//
// Version 1.00:    erste Version
// Version 1.01:    Update auf Igor Pro 7
// Version 1.02:    devIO_1.02 verwenden
// Version 1.03:    devIO_1.03 verwenden
// Version 1.04:    mxaSweep(): Parameter VidBW eingeführt
// Version 1.05:    mxaSweep(): Parameter filtType, swpTypeRule, fFTWidth und preAmp eingeführt
//-------------------------------------------------------------------------------

#pragma TextEncoding = "UTF-8"
#pragma rtGlobals=3
#include "devIO_1.03"

//-----------------------
// Hilfsfunktionen
//-----------------------

// mxaInit() wird beim laden des Moduls aufgerufen und initialisiert alle nötigen Variablen
function mxaInit()
    ioInit()
    mxaChkFolder(1)
end

// mxaChkFolder() erzeugt, falls nicht vorhanden den Ordner "root:mxa" mit allen notwendigen Variablen.
// Falls bereits vorhanden, werden deren Werte auf Gültigkeit geprüft und ggf. korrigiert.
// Eingang: init:   Auch bereits vorhandene Variablen mit Default-Werten initialisieren (1) oder nicht (0)
function mxaChkFolder(init)
    variable init

    string oldDF = getDataFolder(1)
    string ioFolder = "root:mxa"
    variable old = dataFolderExists(ioFolder)
    newDataFolder /O /S $ioFolder   // Ordner und Inhalt erzeugen, falls nicht vorhanden
    variable /G ioDev               //  Geräte-ID des Leistungmessers oder 0 für geschlossen
    variable /G initFlag            //  Flag für Spektrum Analyser ist seit Pgm.-Start initialisiert worden
    if (init || !old)
        ioDev = 0
        initFlag = 0
    endif
    setDataFolder oldDF
end

// ------------------------------
// Steuerungsfunktionen
//-------------------------------

function mxaOpen(ioPath)
    string ioPath

    mxaChkFolder(0)
    string oldDF = getDataFolder(1)
```

```
58      string ioFolder = "root:mxa"
59      setDataFolder ioFolder
60
61      if (ioDebug)
62          printf "mxaOpen(\"%s\")\r", ioPath
63      endif
64      nVar ioDev, initFlag
65      if (ioDev == 0)
66          ioOpen(ioFolder, ioPath, "Spektrum␣Analyser␣Agilent␣MXA")
67          string msgBuf = ioIDN(ioFolder)
68          string s1, s2
69          sscanf msgBuf, "%[^,],%[^,]", s1, s2
70          if (cmpStr(s2[0], "␣") == 0)
71              s2 = s2[1,inf]
72          endif
73          if (cmpStr(s1, "Agilent␣Technologies") != 0 || cmpStr(s2, "N9020A") != 0)
74              ioClose(ioFolder)
75              abort "mxaOpen():␣Der␣Spektrum␣Analyser␣Agilent␣MXA␣antwortet␣nicht."
76          endif
77      endif
78      if (!initFlag)
79          mxaSetPref()
80          initFlag = 1
81      endif
82
83      setDataFolder oldDF
84  end
85
86  // mxaClose() schließt die Schnittstelle
87  function mxaClose()
88      string ioFolder = "root:mxa"
89      if (ioDebug)
90          print "mxaClose()"
91      endif
92      ioClose(ioFolder)
93  end
94
95  // mxaSetPref() stellt die Factory defaults ein
96  function mxaSetPref()
97      string ioFolder = "root:mxa"
98      ioChkOpen(ioFolder, "mxaSetPref")
99      ioSetTmo(ioFolder, 10)
100     ioPutStr(ioFolder, "*CLS")
101     ioPutStr(ioFolder, "*RST")
102     mxaAlert()
103 end
104
105 // mxaAlert() überprüft den Status des Spektrum Analyser. Im Fehlerfall wird das Programm mit
        einer
106 // Fehlermeldung abgebrochen.
107 function mxaAlert()
108     mxaAlertIgnore(0)
109 end
110
111 // mxaAlertIgnore() überprüft den Status des Spektrum Analyser. Im Fehlerfall wird das
        Programm
112 // mit einer Fehlermeldung abgebrochen. Fehler mit der Nummer ignNo werden ignoriert
113 // Eingang: ignNo: zu ignorierende Fehlernummer oder 0
114 // Rückgabe: Fehlernummer
115 function mxaAlertIgnore(ignNo)
116     variable ignNo
117
118     string ioFolder = "root:mxa"
119     variable errNo
120     string errTxt
121     ioPutStr(ioFolder, "SYST:ERR?")
122     string msg = ioGetStr(ioFolder, 1)
123     sscanf msg, "␣%d␣,\"%[^\"]\"", errNo, errTxt
124     if (errNo != 0)
125         ioPutStr(ioFolder, "*CLS")
126         if (errNo != ignNo)
127             mxaClose()
128             abort "Der␣Spektrum␣Analyser␣Agilent␣MXA␣meldet␣den␣Fehler:\r" + replaceString(";
                    ", errTxt, "\r")
129         endif
130     endif
```

```
131     return errNo
132 end
133 
134 //------------------------------
135 // Ausgabefunktionen
136 //------------------------------
137 
138 
139 //-----------------------------------------
140 // Eingabe- und Meßfunktionen
141 //-----------------------------------------
142 
143 // mxaGetWave() empfängt eine Wave vom Spektrum Analyser. Die Anzahl der Punkte wird durch die Länge der
144 // Wave festgelegt.
145 // data: Referenz auf die Wave
146 function mxaGetWave(data)
147     wave data
148 
149     string msg = ""
150     variable eot = 0
151     variable val
152     variable i = 0, j = 0, k
153     do
154         if (strlen(msg) < 50 && !eot)
155             msg += ioGetNStr("root:mxa", 100, 1, eot)
156         endif
157         k = strsearch(msg, ",", 0)
158         if (k == -1)
159             k = strlen(msg)
160         endif
161         val = str2num(msg[0, k - 1])
162         if (j)
163             data[i] = val
164             i += 1
165         else
166             if (abs(val - pnt2x(data, i)) / val > 1e-5)
167                 abort "mxaGetWave():␣falsche␣Frequenz␣gelesen"
168             endif
169         endif
170         j = 1 - j
171         msg = msg[k + 1,Inf]
172     while (strlen(msg) > 0 || eot == 0)
173 end
174 
175 // mxaSweep() führt eine Messung im Spektrum Analyser Mode durch. Der Frequenzbereich und
176 // die Anzahl der Punkte wird durch die Wave festgelegt.
177 // Eingang: dBm:            Wave für Meßwerte [dBm]
178 //              RefLvl:     Referenz-Level [dBm]
179 //              ResBW:          Auflösungsbandbreite [Hz]
180 //              VidBW:          Video-Bandbreite [Hz] (0 = auto, optional, default auto)
181 //              nAvgs:          Anzahl der Averages (optional, default 1)
182 //              filtType:       Filtertyp ("Gauss-3dB", "Gauss-6dB", "GaussNoise", "GaussImpulse" oder "Flattop")
183 //                              (optional, default "Gauss-3dB")
184 //              detType:        Detektortyp ("NORMAL", "AVERAGE", "SAMPLE", "POSITIVE", "NEGATIVE",
185 //                              "QPEAK", "EAVERAGE" oder "RAVERAGE") (optional, default "NORMAL")
186 //              swpTypeRule: Auswahlregel Sweep Typ ("AUTO", ""SPEED", "DRANGE") (optional, default AUTO)
187 //              swpType:        Sweep Typ ("AUTO", "SWEEP" oder "FFT") (optional, default "AUTO")
188 //              fFTWidth:       maximale FFT-Bandbreite [Hz] (optional, default: auto)
189 //                              (4.01e3, 28.81e3, 167.4e3, 411.9e3, 7.99e6, 10e6, 25e6, 0: auto)
190 //              preAmp:     Vorverstärker ein ("FULL"), bis 3,6 GHz ein ("LOW") oder aus ("OFF")
191 
192 
193 function mxaSweep(dBm, RefLvl, ResBW, [VidBW, nAvgs, filtType, detType, swpTypeRule, swpType, fFTWidth, preAmp])
194     wave &dBm
195     variable RefLvl, ResBW, VidBW, nAvgs, fFTWidth
196     string filtType, detType, swpTypeRule, swpType, preAmp
197 
```

```
198     string oldDF = getDataFolder(1)
199     string ioFolder = "root:mxa"
200     setDataFolder ioFolder
201
202     ioChkOpen(ioFolder, "mxaSweep")
203     // Frequenzbereich und Anzahl der Punkte auslesen und überprüfen
204     variable fStart = leftx(dBm)
205     variable fStop = pnt2x(dBm, numpnts(dBm)-1)
206     variable nPts = numpnts(dBm)
207     if (fStart < 10 || fStart > 13.6e9)
208         setDataFolder oldDF
209         abort "mxaSweep():␣ungültige␣Startfrequenz"
210     elseif (fStop < 10 || fStop > 13.6e9)
211         setDataFolder oldDF
212         abort "mxaSweep():␣ungültige␣Stopfrequenz"
213     elseif (fStart > fStop)
214         setDataFolder oldDF
215         abort "mxaSweep():␣Die␣Stopfrequenz␣ist␣größer␣als␣die␣Startfrequenz."
216     elseif (nPts < 1 || nPts > 40001)
217         setDataFolder oldDF
218         abort "mxaSweep():␣ungültige␣Anzahl␣von␣Punkten"
219     endif
220     // Referenz-Level überprüfen
221     if (RefLvl < -170 || RefLvl > 30)
222         setDataFolder oldDF
223         abort "mxaSweep():␣ungültiger␣Referenz␣Level"
224     endif
225     // Auflösungsbandbreite überprüfen
226     if (ResBW < 1 || ResBW > 8e6)
227         setDataFolder oldDF
228         abort "mxaSweep():␣ungültige␣Auflösungsbandbreite"
229     endif
230     // Video-Bandbreite überprüfen
231     if (paramIsDefault(VidBW))
232         VidBW = 0
233     elseif ((VidBW < 1 || VidBW > 50e6) && VidBW != 0)
234         setDataFolder oldDF
235         abort "mxaSweep():␣ungültige␣Video-Bandbreite"
236     endif
237     // Anzahl der Averages überprüfen
238     if (paramIsDefault(nAvgs))
239         nAvgs = 1
240     elseif (nAvgs < 1 || nAvgs > 999)
241         setDataFolder oldDF
242         abort "mxaSweep():␣ungültige␣Anzahl␣von␣Mittelungen"
243     endif
244     string filtTypes = "Gauss-3dB;Gauss-6dB;GaussNoise;GaussImpulse;Flattop;"
245     string filtKeys = "DB3;DB6;NOIS;IMP;;"
246     variable filtNum
247     if (paramIsDefault(filtType))
248         filtNum = 0
249     else
250         filtNum = whichListItem(filtType, filtTypes)
251         if (filtNum < 0)
252             setDataFolder oldDF
253             abort "mxaSweep():␣ungültiger␣Filtertyp"
254         endif
255     endif
256     if (paramIsDefault(detType))
257         detType = "NORMAL"
258     elseif (whichListItem(detType, "NORMAL;AVERAGE;SAMPLE;POSITIVE;NEGATIVE;QPEAK;EAVERAGE;
        RAVERAGE;") < 0)
259         setDataFolder oldDF
260         abort "mxaSweep():␣ungültiger␣Detektortyp"
261     endif
262     if (paramIsDefault(swpTypeRule))
263         swpTypeRule = "AUTO"
264     elseif (whichListItem(swpTypeRule, "AUTO;SPEED;DRANGE;") < 0)
265         setDataFolder oldDF
266         abort "mxaSweep():␣ungültiger␣Sweep␣Typ␣Regel"
267     endif
268     if (paramIsDefault(swpType))
269         swpType = "AUTO"
270     elseif (whichListItem(swpType, "AUTO;SWEEP;FFT;") < 0)
271         setDataFolder oldDF
272         abort "mxaSweep():␣ungültiger␣Sweep␣Typ"
```

```
273     endif
274     if (paramIsDefault(fFTWidth))
275         fFTWidth = 0
276     else
277         string fFTWidthVals = "0;4.01e3;28.81e3;167.4e3;411.9e3;7.99e6;10e6;25e6;"
278         variable nFFTWidthVals = itemsInList(fFTWidthVals)
279         variable i
280         for (i = 0; i < nFFTWidthVals; i++)
281             if (fFTWidth == str2num(stringFromList(i, fFTWidthVals)))
282                 break
283             endif
284         endfor
285         if (i >= nFFTWidthVals)
286             abort "mxaSweep():␣ungültige␣maximale␣FFT-Bandbreite"
287         endif
288     endif
289     if (paramIsDefault(preAmp))
290         swpType = "OFF"
291     elseif (whichListItem(preAmp, "FULL;LOW;OFF;") < 0)
292         setDataFolder oldDF
293         abort "mxaSweep():␣ungültige␣Vorverstärker␣Einstellung"
294     endif
295     // Spektrum Analyser anhalten
296     ioSetTmo(ioFolder, 10)
297     ioPutStr(ioFolder, "INIT:CONT␣OFF")
298     ioPutStr(ioFolder, "*OPC?"); ioGetStr(ioFolder, 1)
299     // Parameter setzen
300     string msg
301     sprintf msg, "SENS:FREQ:STAR␣%f␣HZ", fStart; ioPutStr(ioFolder, msg)
302     sprintf msg, "SENS:FREQ:STOP␣%f␣HZ", fStop; ioPutStr(ioFolder, msg)
303     sprintf msg, "SENS:SWE:POIN␣%d", nPts; ioPutStr(ioFolder, msg)
304     sprintf msg, "DISP:WIND1:TRAC:Y:SCAL:RLEV␣%f", RefLvl; ioPutStr(ioFolder, msg)
305     string filtKey = stringFromList(filtNum, filtKeys)
306    if (strLen(filtKey))
307         ioPutStr(ioFolder, "SENS:BWID:SHAP␣GAUS")
308         ioPutStr(ioFolder, "SENS:BWID:TYPE␣" + filtKey)
309     else
310         ioPutStr(ioFolder, "SENS:BWID:SHAP␣FLAT")
311     endif
312     sprintf msg, "SENS:BWID:RES␣%f␣HZ", ResBW; ioPutStr(ioFolder, msg)
313     if (VidBW == 0)
314         ioPutStr(ioFolder, "SENS:BWID:VID:AUTO␣ON")
315     else
316         ioPutStr(ioFolder, "SENS:BWID:VID:AUTO␣OFF")
317         sprintf msg, "SENS:BWID:VID␣%f␣HZ", VidBW; ioPutStr(ioFolder, msg)
318     endif
319     sprintf msg, "SENS:DET:TRAC1␣%s", detType; ioPutStr(ioFolder, msg)
320     if (cmpStr(swpTypeRule, "AUTO") == 0)
321         ioPutStr(ioFolder, "SENS:SWE:TYPE:AUTO:RUL:AUTO:STAT␣ON")
322     else
323         ioPutStr(ioFolder, "SENS:SWE:TYPE:AUTO:RUL:AUTO:STAT␣OFF")
324         ioPutStr(ioFolder, "SENS:SWE:TYPE:AUTO:RUL␣" + swpTypeRule)
325     endif
326     if (cmpStr(swpType, "AUTO") == 0)
327         ioPutStr(ioFolder, "SENS:SWE:TYPE:AUTO␣ON")
328     else
329         ioPutStr(ioFolder, "SENS:SWE:TYPE:AUTO␣OFF")
330         sprintf msg, "SENS:SWE:TYPE␣%s", swpType; ioPutStr(ioFolder, msg)
331     endif
332     if (fFTWidth)
333         ioPutStr(ioFolder, "SENS:SWE:FFT:WIDT:AUTO␣OFF")
334         sprintf msg, "SENS:SWE:FFT:WIDT␣%f␣HZ", fFTWidth; ioPutStr(ioFolder, msg)
335     else
336         ioPutStr(ioFolder, "SENS:SWE:FFT:WIDT:AUTO␣ON")
337     endif
338     if (nAvgs < 1.5)
339         nAvgs = 1
340         ioPutStr(ioFolder, "TRAC1:TYPE␣WRIT")
341     else
342         nAvgs = round(nAvgs)
343         sprintf msg, "SENS:AVER:COUN␣%d", nAvgs; ioPutStr(ioFolder, msg)
344         ioPutStr(ioFolder, "TRAC1:TYPE␣AVER")
345     endif
346     if (cmpStr(preAmp, "OFF") == 0)
347         ioPutStr(ioFolder, "SENS:POW:RF:GAIN:STAT␣OFF")
348     else
```

```
349         ioPutStr(ioFolder, "SENS:POW:RF:GAIN:STAT␣ON")
350         ioPutStr(ioFolder, "SENS:POW:RF:GAIN:BAND␣" + preAmp)
351     endif
352     mxaAlert()
353     // Timeout setzen
354     ioPutStr(ioFolder, "SENS:SWE:TIME?")
355     variable tmo = ioGetFlt(ioFolder)
356     ioSetTmo(ioFolder, 30 + 1.1 * nAvgs * tmo)
357     mxaAlert()
358     // Messung durchführen
359     ioPutStr(ioFolder, "INIT:IMM")
360     ioPutStr(ioFolder, "*OPC?"); ioGetStr(ioFolder, 1)
361     mxaAlert()
362     // Ergebnis auslesen
363     ioSetTmo(ioFolder, 10)
364     ioPutStr(ioFolder, "FORM:TRAC:DATA␣ASC")
365     ioPutStr(ioFolder, "FETCH:SAN1?")
366     mxaGetWave(dBm)
367
368     setDataFolder oldDF
369     mxaAlert()
370 end
```

6.4.2 Quelltext msUtils_1.06.ipf

Die Datei `msUtils_1.06.ipf` in den gemeinsamen Igor Pro includes enthält physikalische Konstanten und Hilfsfunktionen zur Auswertung von Messungen. Neben den Konstanten werden die Funktionen **`phaCorr()`**, **`magPha()`** und **`dBPha()`** bei der Auswertung verwendet.

```
1  //-------------------------------------------------------------------------------------
2  // Modul mit diversen Hilfsfunktionen für Meßprogramme
3  //
4  // (c) 2006, 2007,2016 Claudius Peschke, Gesellschaft für Schwerionenforschung mbH
5  //
6  // Version 1.00:    erste Version
7  // Version 1.02:    Funktion phaCorr4D() implementiert
8  // Version 1.03:    Funktionen magPha(), dBPha(),
9  //                  SMxMagPha(), SMxDbPha() implementiert
10 // Version 1.04:    Funktionen phaCorrLinConstRange() und
11 //                  phaCorrLinConstRange() implementiert
12 // Version 1.05:    Update auf Igor Pro 7
13 // Version 1.06:    weitere Konstanten hinzugefügt
14 //-------------------------------------------------------------------------------------
15
16 #pragma TextEncoding = "UTF-8"
17 #pragma rtGlobals=1
18
19 //------------------
20 // Konstanten
21 //------------------
22
23 constant c0 = 2.99792458e8          // Vakuumlichtgeschwindigkeit [m/s]
24 constant mu0 = 1.2566370614e8          // Vakuumpermeabilitätskonstante [N/A^2]
25 constant eps0 = 8.854187871e-12     // Vakuumdielektrizitätskonstante [(As)^2/N/m^2]
26 constant kB = 1.380658e-23          // Boltzmann-Konstante [J/K]
27 constant qe = 1.602176634e-19           // Elementarladung [C]
28 constant me = 9.1093897e-31         // Elektronenmasse [kg]
29 constant mp = 1.6726231e-27         // Protonenmasse [kg]
30 constant mn = 1.6749286e-27         // Neutronenmasse [kg]
31
32 constant minPrim = 0, maxPrim = 30  // min/max IEEE-488 Primäradresse
33
34 //-----------------------------------
35 // exportierte Funktionen
36 //-----------------------------------
37
38 // clip(var, minVal, maxVal) liefert min(minVal, max(maxVal, var))
39 Function clip(var, minVal, maxVal)
40     Variable var, minVal, maxVal
41     if (var < minVal)
42         var = minVal
43     elseif (var > maxVal)
44         var = maxVal
```

```
45      endif
46      return var
47  End
48
49  // cInterp() liefert einen interpolierten Wert der komplexen wave ywave an der Stelle x. Die
        Interpolation
50  // erfolgt linear in Betrag und Phase. Phasensprünge (|dphi|>pi) werden berücksichtigt. Bei X
        -Werten
51  // außerhalb des Bereichs wird der erste bzw. letzte Wert von ywave zurückgegeben
52  // (konstante Extrapolation).
53  function /C cInterp(ywave, x)
54      wave /C ywave
55      variable x
56
57      if (x <= leftx(ywave))
58          return ywave[0]
59      elseif (x >= pnt2x(xwave, numpnts(xwave)-1))
60          return ywave[inf]
61      else
62          variable pos = (x - leftx(ywave))/deltax(ywave)
63          variable i = floor(pos), j = i+1
64          // NaNs überspringen
65          if (numtype(ywave[i]) != 0)
66              do
67                  i -= 1
68              while (numtype(ywave[i]) && i >= 0)
69          endif
70          if (numtype(ywave[j]) != 0)
71              do
72                  j += 1
73              while (numtype(ywave[j]) != 0 && j < numpnts(ywave))
74          endif
75          if (i <= -1 && j >= numpnts(ywave))
76              return NaN
77          elseif (i <= -1)
78              return ywave[j]
79          elseif (j >= numpnts(ywave))
80              return ywave[i]
81          endif
82          variable f = (pos-i)/(j-i)
83          //### nan?
84          variable magL = cabs(ywave[i]), magR = cabs(ywave[j])
85          variable phaL = imag(r2polar(ywave[i])), phaR = imag(r2polar(ywave[j]))
86          if (phaR-phaL > pi)
87              phaR -= 2*pi
88          elseif (phaR-phaL < -pi)
89              phaR += 2*pi
90          endif
91          return p2rect(cmplx((1-f)*magL+f*magR, (1-f)*phaL+f*phaR))
92      endif
93  end
94
95  // cInterpXY() liefert einen interpolierten Wert der komplexen wave ywave an der Stelle x.
        Die korospondierenden X-Werte
96  // zu ywave werden in der streng monotonen wave xwave übergeben. Die Interpolation erfolgt
        linear in Betrag und Phase.
97  // Phasensprünge (|dphi|>pi) werden berücksichtigt. Bei X-Werten außerhalb des, durch xwave
        gegebenen Bereichs wird
98  // der erste bzw. letzte Wert von ywave zurückgegeben (konstante Extrapolation).
99  function /C cInterpXY(ywave, xwave, x)
100     wave /C ywave
101     wave xwave
102     variable x
103
104     if (x <= xwave[0])
105         return ywave[0]
106     elseif (x >= xwave[inf])
107         return ywave[inf]
108     else
109         // Exakte Position in x-Wave finden
110         findLevel /Q /P xwave, x
111         variable i = floor(V_LevelX), j = i+1
112         // NaNs überspringen
113         if (numtype(xwave[i]) != 0 || numtype(ywave[i]) != 0)
114             do
115                 i -= 1
```

```
            while ((numtype(xwave[i]) != 0 || numtype(ywave[i]) != 0) && i >= 0)
        endif
        if (numtype(xwave[j]) != 0 || numtype(ywave[j]) != 0)
            do
                j += 1
            while ((numtype(xwave[j]) != 0 || numtype(ywave[j]) != 0) && j < numpnts(ywave))
        endif
        if (i <= -1 && j >= numpnts(ywave))
            return NaN
        elseif (i <= -1)
            return ywave[j]
        elseif (j >= numpnts(ywave))
            return ywave[i]
        endif
        variable f = (V_LevelX-i)/(j-i)
        // Interpolieren
        variable magL = cabs(ywave[i]), magR = cabs(ywave[j])
        variable phaL = imag(r2polar(ywave[i])), phaR = imag(r2polar(ywave[j]))
        if (phaR-phaL > pi)
            phaR -= 2*pi
        elseif (phaR-phaL < -pi)
            phaR += 2*pi
        endif
        return p2rect(cmplx((1-f)*magL+f*magR, (1-f)*phaL+f*phaR))
    endif
end

// phaCorr(phi) korrigiert Phasensprünge (|dphi| > 180°) in der 1D-Wave phi durch Addition ganzer Vielfacher von 360°
function phaCorr(phi)
    wave phi

    variable left, right, a, phaOfs
    phaOfs = 0
    a = 0; left = -1
    for (right = 0; right < numpnts(phi); right += 1)
        if (numType(phi[right]) == 0)
            if (left != -1)
                phi[right] += phaOfs
                if(phi[right] > phi[left]+a*(right-left)+180)
                    phaOfs -= 360
                    phi[right] -= 360
                elseif(phi[right] < phi[left]+a*(right-left)-180)
                    phaOfs += 360
                    phi[right] += 360
                endif
                a = (a+(phi[right]-phi[left])/(right-left))/2
            endif
            left = right
        endif
    endfor
end

// phaCorrLin(phi) korrigiert Phasensprünge (|dphi| > 180°) in der 1D-Wave phi durch Addition ganzer Vielfacher von 360°
// und beseitigt den phasenlinearen Anteil durch Subtraktion einer linearen Funktion mit kleinster Fehlerquadratsumme
function phaCorrLin(phi)
    wave phi

    phaCorr(phi)
    curveFit /Q line phi
    phi -= K1*x
    return K1
end

// phaCorrLinConst(phi) korrigiert Phasensprünge (|dphi| > 180°) in der 1D-Wave phi durch Addition ganzer Vielfacher von 360° und beseitigt
// den phasenlinearen und konstanten Anteil durch Subtraktion einer affin linearen Funktion mit kleinster Fehlerquadratsumme
function phaCorrLinConst(phi)
    wave phi

    phaCorr(phi)
    curveFit /Q line phi
    phi -= K1*x+K0
```

```
187     return K0
188 end
189
190 // phaCorrLinRange(phi, x1, x2) korrigiert Phasensprünge (|dphi| > 180°) in der 1D-Wave phi
        durch Addition ganzer Vielfacher
191 // von 360° und beseitigt den phasenlinearen Anteil durch Subtraktion einer linearen Funktion
         mit kleinster Fehlerquadratsumme
192 // im Bereich x1 bis x2.
193 function phaCorrLinRange(phi, x1, x2)
194     wave phi
195     variable x1, x2
196
197     phaCorr(phi)
198     curveFit /Q line phi(x1, x2)
199     phi -= K1*x
200     return K1
201 end
202
203 // phaCorrLinConstRange(phi, x1, x2) korrigiert Phasensprünge (|dphi| > 180°) in der 1D-Wave
        phi durch Addition ganzer Vielfacher
204 // von 360° und beseitigt den phasenlinearen und konstanten Anteil durch Subtraktion einer
        affin linearen Funktion mit
205 // kleinster Fehlerquadratsumme im Bereich x1 bis x2.
206 function phaCorrLinConstRange(phi, x1, x2)
207     wave phi
208     variable x1, x2
209
210     phaCorr(phi)
211     curveFit /Q line phi(x1, x2)
212     phi -= K1*x+K0
213     return K0
214 end
215
216 // phaCorr4D(phi) korrigiert Phasensprünge (|dphi| > 180°) in der  maximal vierdimensionalen
        Wave phi durch Addition
217 // ganzer Vielfacher von 360°:
218 // In der äußeren Schleife wird der Vektor aus den [0][0][0]-Ecken jedes Quaders aus dem
        Hyper-Quader korrigiert.
219 // Nach jedem korrigierten Eckpunkt wird in der zweiten Schleife die [0][0]-Ecke jeder Fläche
         darin korrigiert.
220 // Die dritte Schleife korrigiert analog dazu das [0]-Ende des Vektors und die vierte
        Schleife die Punkte den Vektors.
221 function phaCorr4D(phi)
222     wave phi
223
224     variable ofsX, ofsY, ofsZ, ofsT
225     variable iX, iY, iZ, iT
226     ofsT = 0
227     // Korrektur über alle [0][0][0]-Ecken jedes Unterquaders des Hyperquaders
228     for (iT = 0; iT < max(dimSize(phi, 3), 1); iT += 1)
229         if (iT > 0)
230             phi[0][0][0][iT]+= ofsT
231             if (phi[0][0][0][iT] > phi[0][0][0][iT-1] + 180)
232                 ofsT -= 360
233                 phi[0][0][0][iT] -= 360
234             elseif(phi[0][0][0][iT] < phi[0][0][0][iT-1] - 180)
235                 ofsT += 360
236                 phi[0][0][0][iT] += 360
237             endif
238         endif
239         // Korrektur über alle [0][0]-Ecken jeder Unterfläche des Quaders
240         ofsZ = ofsT
241         for (iZ = 0; iZ < max(dimSize(phi, 2), 1); iZ += 1)
242             if (iZ > 0)
243                 phi[0][0][iZ][iT]+= ofsZ
244                 if (phi[0][0][iZ][iT] > phi[0][0][iZ-1][iT] + 180)
245                     ofsZ -= 360
246                     phi[0][0][iZ][iT] -= 360
247                 elseif(phi[0][0][iZ][iT] < phi[0][0][iZ-1][iT] - 180)
248                     ofsZ += 360
249                     phi[0][0][iZ][iT] += 360
250                 endif
251             endif
252             // Korrektur über alle [0]-Enden jedes Untervektors des Quaders
253             ofsY = ofsZ
254             for (iY = 0; iY < max(dimSize(phi, 1), 1); iY += 1)
```

```
                if (iY > 0)
                    phi[0][iY][iZ][iT]+= ofsY
                    if (phi[0][iY][iZ][iT] > phi[0][iY-1][iZ][iT] + 180)
                        ofsY -= 360
                        phi[0][iY][iZ][iT] -= 360
                    elseif(phi[0][iY][iZ][iT] < phi[0][iY-1][iZ][iT] - 180)
                        ofsY += 360
                        phi[0][iY][iZ][iT] += 360
                    endif
                endif
                // Korrektur über alle Unterpunkte des Vektors
                ofsX = ofsY
                for (iX = 1; iX < max(dimSize(phi, 0), 1); iX += 1)
                    phi[iX][iY][iZ][iT] += ofsX
                    if (phi[iX][iY][iZ][iT] > phi[iX-1][iY][iZ][iT] + 180)
                        ofsX -= 360
                        phi[iX][iY][iZ][iT] -= 360
                    elseif(phi[iX][iY][iZ][iT] < phi[iX-1][iY][iZ][iT] - 180)
                        ofsX += 360
                        phi[iX][iY][iZ][iT] += 360
                    endif
                endfor
            endfor
        endfor
    endfor
end

// cSqrt(cwave) zieht vor Ort die Quadratwurzel einer komplexen Wave und versucht
// dabei immer auf dem gleichen Lösungszweig zu bleiben
function cSqrt(cwave)
    wave /C cwave
    variable i, pha, phaL, ofs
    pha = imag(r2polar(cwave[0]))
    ofs = 0
    for (i = 1; i < numpnts(cwave); i += 1)
        phaL = pha
        pha = ofs+imag(r2polar(cwave[i]))
        if(pha > phaL+pi)
            ofs -= 2*pi
            pha -= 2*pi
        elseif(pha < phaL-pi)
            ofs += 2*pi
            pha += 2*pi
        endif
        cwave[i] = p2rect(cmplx(sqrt(cabs(cwave[i])), pha/2))
    endfor
end

// magPha(wName) berechnet zu einer Wave mit Namen wName den
// Betrag wName+"mag" und die Phase wName+"pha"
function magPha(wName)
    string wName

    wave /C cplx = $wName
    make /O /D /N=(dimSize(cplx, 0), dimSize(cplx, 1), dimSize(cplx, 2), dimSize(cplx, 3)) $(
        wName+"mag"),  $(wName+"pha")
    wave mag = $(wName+"mag"),  pha = $(wName+"pha")
    setScale /P x dimOffset(cplx, 0), dimDelta(cplx, 0), waveUnits(cplx, 0), mag, pha
    setScale /P y dimOffset(cplx, 1), dimDelta(cplx, 1), waveUnits(cplx, 1), mag, pha
    setScale /P z dimOffset(cplx, 2), dimDelta(cplx, 2), waveUnits(cplx, 2), mag, pha
    setScale /P t dimOffset(cplx, 3), dimDelta(cplx, 3), waveUnits(cplx, 3), mag, pha
    setScale d 0, 0, "", mag
    setScale d 0, 0, "°", pha
    mag = cabs(cplx)
    pha = 180/pi * imag(r2polar(cplx))
    if(dimSize(cplx, 1) == 0)
        phaCorr(pha)
    else
        phaCorr4D(pha)
    endif
end

// dBPha(wName) berechnet zu einer Wave mit Namen wName den
// Betrag wName+"dB" in dB und die Phase wName+"pha"
function dBPha(wName)
    string wName
```

```
330
331     wave /C cplx = $wName
332     make /O /D /N=(dimSize(cplx, 0), dimSize(cplx, 1), dimSize(cplx, 2), dimSize(cplx, 3)) $(
            wName+"dB"),  $(wName+"pha")
333     wave dB = $(wName+"dB"),  pha = $(wName+"pha")
334     setScale /P x dimOffset(cplx, 0), dimDelta(cplx, 0), waveUnits(cplx, 0), dB, pha
335     setScale /P y dimOffset(cplx, 1), dimDelta(cplx, 1), waveUnits(cplx, 1), dB, pha
336     setScale /P z dimOffset(cplx, 2), dimDelta(cplx, 2), waveUnits(cplx, 2), dB, pha
337     setScale /P t dimOffset(cplx, 3), dimDelta(cplx, 3), waveUnits(cplx, 3), dB, pha
338     setScale d 0, 0, "dB", dB
339     setScale d 0, 0, "°", pha
340     dB = 20*log(cabs(cplx))
341     pha = 180/pi * imag(r2polar(cplx))
342     if(dimSize(cplx, 1) == 0)
343         phaCorr(pha)
344     else
345         phaCorr4D(pha)
346     endif
347 end
348
349 // SMxMagPha(folder, n) berechnet zu den S-Parametern S11..Snn die
350 // Beträge Smnmag und die Phasen Smnpha
351 function SMxMagPha(folder, n)
352     string folder   // Name des Folders
353     variable n  // Anzahl der Ports
354
355     string oldDF = getDataFolder(1)
356     setDataFolder $folder
357
358     string wName
359     variable i, j
360     for (i = 1; i <= n; i += 1)
361         for (j = 1; j <= n; j += 1)
362             sprintf wName, "S%1d%1d", i, j
363             magPha(wName)
364         endfor
365     endfor
366
367     setDataFolder oldDF
368 end
369
370 // SMxDbPha(folder, n) berechnet zu den S-Parametern S11..Snn die
371 // Beträge SmndB in dB und die Phasen Smnpha
372 function SMxDbPha(folder, n)
373     string folder   // Name des Folders
374     variable n  // Anzahl der Ports
375
376     string oldDF = getDataFolder(1)
377     setDataFolder $folder
378
379     string wName
380     variable i, j
381     for (i = 1; i <= n; i += 1)
382         for (j = 1; j <= n; j += 1)
383             sprintf wName, "S%1d%1d", i, j
384             dBPha(wName)
385         endfor
386     endfor
387
388     setDataFolder oldDF
389 end
```

6.4.3 Quelltext graphUtils_1.01.ipf

Die Datei `graphUtils_1.01.ipf` in den gemeinsamen Igor Pro includes enthält einige Hilfsfunktionen zur graphischen Darstellung von Messungen. Hieraus wird die Funktionen **`setLineStyles()`** zur Definition von einheitlichen Linien-Farben und -Stilen verwendet. Die Funktionen **`closeAllWins()`** schließt alle Fenster eines Typs, in unserem Fall alle Diagramme.

```
//-------------------------------------------------------------------------------------
// Modul zur mit diversen Hilfsfunktionen für den Grafik-Export
//
// (c) 2022,2023 Claudius Peschke, Gesellschaft für Schwerionenforschung mbH
//
// Version 1.00:    erste Version
// Version 1.01:    neue Funktion vw250svg2ps()
//-------------------------------------------------------------------------------------

#pragma TextEncoding = "UTF-8"
#pragma rtGlobals=3

// PDF Ausgabe A4 Hochformat
constant pgA4Width  = 21.0  // Seitenbreite [cm]
constant pgA4Height = 29.7  // Seitenhöhe [cm]
constant pgA4BrdL   = 2.0       // linker Rand [cm]
constant pgA4BrdT   = 1.0       // oberer Rand [cm]
constant pgA4BrdR   = 1.0       // rechter Rand [cm]
constant pgA4BrdB   = 1.0       // unterer Rand [cm]

// String für einige Sonderzeichen aus Font "TeXGyreHeros" (Workaround für CMAP-Problem)
strConstant charLGamma  = "\\F'TeXGyreHerosΓ'\\F'default'"   // großes Gamma Γ()
strConstant charLDelta  = "\\F'TeXGyreHerosΔ'\\F'default'"   // großes delta Δ()
strConstant charSPhi    = "\\F'TeXGyreHerosφ'\\F'default'"   // kleines phi φ()
strConstant charLSigma  = "\\F'TeXGyreHerosΣ'\\F'default'"   // großes Sigma Σ()
strConstant charLOmega  = "\\F'TeXGyreHerosΩ'\\F'default'"   // großes Omega Ω()
strConstant charApprox  = "\\F'TeXGyreHeros≈'\\F'default'"   // ungefähr gleich ≈()
strConstant charEllips  = "\\F'TeXGyreHeros…'\\F'default'"   // drei Punkte …()

// setLineStyles() definiert eine gemeinsame Farb- und Stilskala für Graphen
// Die Stile sollten auf dem Bildschirm und Drucker gut unterscheidbar sein.
// Es werden 4·baseColors Farb/Breite/Linienstil-Kombinationen erzeugt.
// Eingang: baseColors: Anzahl der Grundfarben bis zur Wiederholung mit anderem Linienstil
//                              (optional, default: 12)
// Ausgang: color…[02][i]:   Farbe R (0), G (1), B (2) des Stils i
//              lWidth[i]:      Linienbreite des Stils i [pt]
//              lStyle[i]:      Linienstil des Stils i (siehe "modifyGraph lsize…=")
function setLineStyles([baseColors])
    variable baseColors

    // alle möglichen Grundfarben setzen
    make /O /I /N=(3, 12) root:baseColor
    wave /I baseColor = root:baseColor
    baseColor[][00] = {0x00, 0x00, 0x00}    // schwarz
    baseColor[][01] = {0xFF, 0x00, 0x00}    // rot
    baseColor[][02] = {0xFF, 0xBB, 0x44}    // orange
    baseColor[][03] = {0x00, 0xCC, 0x00}    // grün
    baseColor[][04] = {0x00, 0xCC, 0xCC}    // türkis
    baseColor[][05] = {0x00, 0x00, 0xFF}    // blau
    baseColor[][06] = {0xFF, 0x00, 0xCC}    // violett
    baseColor[][08] = {0x66, 0x66, 0x66}    // mittelgrau
    baseColor[][07] = {0xBB, 0x88, 0x33}    // dunkelbraun
    baseColor[][09] = {0xBB, 0xBB, 0xBB}    // hellgrau
    baseColor[][10] = {0xFF, 0xDD, 0x88}    // hellbraun
    baseColor[][11] = {0xDD, 0xDD, 0x33}    // gelb
    baseColor *= 0x100
    // Stile erzeugen
    if (paramIsDefault(baseColors))
        baseColors = 12
    endif
    variable nStyles = 4 * baseColors
    make /O /I /N=(3, nStyles) root:color
    wave /I color = root:color
    make /O /D /N=(nStyles) root:lWidth
    wave /I lWidth = root:lWidth
    make /O /I /N=(nStyles) root:lStyle
    wave lStyle = root:lStyle
    // erste Farbrunde: durchgezogen, 0,75pt
```

```
    variable l = 0, u = baseColors - 1
    color[][l,u] = baseColor[p][q]
    lWidth[l,u] = 0.75
    lStyle[l,u] = 0
    // zweite Farbrunde: durchgezogen, 1,25pt
    l = u + 1; u += baseColors
    color[][l,u] = baseColor[p][q - l]
    lWidth[l,u] = 1.25
    lStyle[l,u] = 0
    // dritte Farbrunde: gestrichelt, 1,25pt
    l = u + 1; u += baseColors
    color[][l,u] = baseColor[p][q - l]
    lWidth[l,u] = 1.25
    lStyle[l,u] = 3
    // vierte Farbrunde: Strich/Punkt, 1,25pt
    l = u + 1; u += baseColors
    color[][l,u] = baseColor[p][q - l]
    lWidth[l,u] = 1.25
    lStyle[l,u] = 5
    // Grundfarben löschen
    killWaves baseColor
end

// closeWins() schließt alle Fenster die in einer semikolon-separierten Liste angegeben werden
// Eingang: list: semikolon-separierten Liste der Fensternamen
function closeWins(list)
    string list
    variable i
    for (i = itemsInList(list) - 1; i >= 0; i -= 1)
        doWindow /K $(stringFromList(i, list))
    endfor
end

// closeAllWins() schließt alle Fenster bestimmter Typen
// Eingang: windowTypes:    Bit-Feld für Fenster-Typen
//                              (optional, siehe WinList(), default: Graphs und Layouts)
function closeAllWins([windowTypes])
    variable windowTypes

    if (paramIsDefault(windowTypes))
        windowTypes = 5
    endif
    closeWins(winList("*", ";", "WIN:" + num2istr(windowTypes)))
end

// epsExportA4() exportiert das aktuelle Fenster als DIN A4 ESP-Datei ohne Fonts
// Eingang: fName: Datainame
function epsExportA4(fName)
    string fName

    savePICT /O /E=-3 /EF=1 /M /W=(0, 0, pgA4Width, pgA4Height) /P=home as fName
end

// svg2ps() konvertiert eine SVG-Datei in eine Postscript-Datei
// Eingang: srcName:    Name der Quelldatei
//              dstName:    Name der Zieldatei
function svg2ps(srcName, dstName)
    string srcName, dstName

    // Inkscape kann nicht per UNC zugreifen
    pathInfo home
    if (cmpStr(S_path[1], ":") != 0)
        abort "svg2ps():␣Das␣Verzeichnis␣mit␣dem␣Experiment␣muß␣über␣einen␣Laufwerksbuchstaben␣zugreifbar␣sein."
    endif
    string basePath = S_path[0,1] + "\\" + replaceString(":", S_path[2,inf], "\\")
    // 1. Versuch Version >= 1.00: Binary im bin-Verzeichnis
    string inkPath = "C:\\Program␣Files\\Inkscape\\bin"
    string inkExe = "inkscape.exe"
    string inkOpts = "--export-area-page␣--export-type=ps␣--export-filename=\"" + basePath + dstName + "\"␣--export-ps-level=3"
    newPath /O /Q /Z inkSym, inkPath
    if (!V_flag)
        if (findListItem(inkExe, indexedFile(inkSym, -1, ".exe")) < 0)
            V_flag = -1 // nicht gefunden
```

```
142         endif
143     endif
144     if (V_flag)
145         // 2. Versuch Version < 1.00: Binary im Basis-Verzeichnis
146         inkPath = "C:\\Program␣Files\\Inkscape"
147         inkExe = "inkscape.exe"
148         inkOpts = "--export-area-page␣--export-ps=\"" + basePath + dstName + "\"␣--export-ps-
                level=3"
149         newPath /O /Q /Z inkSym, inkPath
150         if (!V_flag)
151             if (findListItem(inkExe, indexedFile(inkSym, -1, ".exe")) < 0)
152                 V_flag = -1 // nicht gefunden
153             endif
154         endif
155     endif
156     if (V_flag)
157         abort "svg2ps():␣Das␣Programm␣Inkscape␣konnte␣nicht␣gefunden␣werden."
158     endif
159     string cmd = "\"" + inkPath + "\\" + inkExe + "\"" + inkOpts + "␣\"" + srcName + "\""
160     executeScriptText cmd
161 end
```

6.4.4 Quelltext touchstone_1.06.ipf

Die Datei `touchstone_1.06.ipf` in den gemeinsamen Igor Pro includes enthält Funktionen zum lesen und schreiben von Touchstone-Dateien. Die Funktionen **`tsRead()`** wird verwendet um die BTF-Messungen der Web-Applikation zu laden.

```
1  //-------------------------------------------------------------------------------------
2  // Modul zum lesen und schreiben von S-Parameter in Touchstone-Dateien
3  //
4  // Initialisierung bei Programmstart: smxInit()
5  //
6  // (c) 2006, Claudius Peschke, Gesellschaft für Schwerionenforschung mbH
7  //
8  // Version 1.00:    erste Version
9  // Version 1.01:    Skalierungsfehler in tsRead() beseitigt
10 // Version 1.02:    Tabulator statt Leerzeichen in tsRead() erlauben
11 // Version 1.03:    tsWrite() kann Dateien mit unvollständigem S-Parametersatz schreiben
12 // Version 1.04:    Update auf Igor Pro 7
13 // Version 1.05:    maximal 4 Parameter in eine Zeile ausgeben
14 // Version 1.06:    Fehlerkorrektor bei UTF8-Strings
15 //-------------------------------------------------------------------------------------
16
17 #pragma TextEncoding = "UTF-8"
18 #pragma rtGlobals=3
19
20 // tsReadFloat() ist eine Hilfsfunktion von tsRead(), die eine Gleitkommazahl liest
21 function tsReadFloat(filePath, refNum, line, msg)
22     string  filePath
23     variable    refNum
24     string  &line
25     string  msg
26
27     variable endPos = strsearch(line, "␣", 0)
28     if (endPos < 1)
29         endPos = strsearch(line, "!", 0)
30         if (endPos < 1)
31             endPos = strlen(line)
32         endif
33     endif
34     string fStr = line[0, endPos-1]
35     line = line[endPos, inf]
36     do
37         line = line[1, inf]
38     while (cmpStr(line[0], "␣") == 0)
39     variable fVal
40     string s
41     sscanf fStr, "%g%1s", fVal, s
42     if (v_flag != 1)
43         abort msg+"␣\""+fStr+"\"␣in␣der␣Datei␣\""+filePath+"\"␣ist␣ungültig."
44     endif
45     return fVal
```

```
46  end
47
48  // tsRead() liest S-Parameter aus einer Touchstone-Datei in einen Data-Folder ein
49  // Der Folder enthält danach folgende Variablen, * = optional:
50  // nvar ports:              Anzahl der Ports (1..9)
51  // wave f:                  Frequenzen [Hz]
52  // wave /C Snm:             komplexe S-Parameter Waves mit m=1..ports und n=1..ports
53  //                              wenn f äquidistant ist wird Smn skaliert, sonst ist leftx=0
        und deltax=1
54  // nvar zLine:              Bezugsimpedanz [Ohm]
55  // svar filePath:           Pfad/Name der SnP-Datei
56  // svar partName*:          Name des Bauteils, wird aus der ersten Kommentar-Zeile gelesen
57  // svar comment*:           Kommentar, weitere Kommentarzeilen
58  // Eingang: path:           Pfad/Name der SnP-Datei relativ zur Experiment-Datei oder absolut
        oder ""
59  //              folder:     Pfad des Data-Folders (z.B. "root:sMatrix")
60  //              lileDialog: Flag für File-Dialog anzeigen (1) oder nicht anzeigen (0)
61  //                              wenn path == "" wird er auf jeden Fall angezeigt
62  // Rückgabe:    Datei geladen (0) oder Abbruch (1)
63  function tsRead(path, folder, fileDialog)
64      string  path, folder
65      variable    fileDialog
66
67      string oldDF = getDataFolder(1)
68      killDataFolder /Z $folder
69      newDataFolder /S $folder
70      // Datei öffnen mit einer Zeile look-ahead
71      string fileType = selectString(stringmatch(IgorInfo(2), "Macintosh*"), ".s?p", "TEXT")
72      variable refNum
73      do
74          string pStr = path[strLen(path)-4,inf]
75          variable valid = cmpStr(pStr[0,1], ".s") == 0 && cmpStr(pStr[2], "0") >= 0 && cmpStr(
                pStr[2], "9") <= 0 && cmpStr(pStr[3], "p") == 0
76          if (!fileDialog && !valid)
77              doAlert 0, "Die␣Dateierweiterung␣\""+pStr+"\"␣ist␣ungültig.␣Touchstone-Dateien␣
                    müssen␣mit␣.s1p␣bis␣.s9p␣enden."
78          endif
79          if (fileDialog || !valid)
80              open /D /R /C="R*ch" /T=(fileType) /M="Touchstone-Datei␣laden␣..." /P=home refNum
                     as path
81              if (strLen(s_fileName) == 0)
82                  return 1
83              endif
84              path = s_fileName
85              fileDialog = 0
86          endif
87      while (!valid)
88      string /G filePath = path
89      variable /G ports = str2num(pStr[2])
90      open /R /C="R*ch" /T=(fileType) /P=home refNum as filePath
91      string line
92      fReadLine refNum, line
93      line = replaceString("\t", line, "␣")
94      // Bauteilename lesen
95      do
96          if (cmpStr(line[0], "!") == 0)
97              do
98                  line = line[1,inf]
99              while (cmpStr(line[0], "␣") == 0)
100             string /G partName = line[0,strLen(line)-2]
101             fReadLine refNum, line
102             line = replaceString("\t", line, "␣")
103             break
104         elseif (cmpStr(line[0], "␣") == 0)
105             line = line[1,inf]
106         elseif (cmpStr(line[0], "\r") == 0)
107             fReadLine refNum, line
108             line = replaceString("\t", line, "␣")
109         else
110             break
111         endif
112     while (1)
113     // Kommentar lesen
114     string lclComment = ""
115     do
116         if (cmpStr(line[0], "!") == 0)
```

```
            do
                line = line[1,inf]
            while (cmpStr(line[0], "␣") == 0)
            lclComment += line
            fReadLine refNum, line
            line = replaceString("\t", line, "␣")
        elseif (cmpStr(line[0], "␣") == 0)
            line = line[1,inf]
        elseif (cmpStr(line[0], "\r") == 0)
            fReadLine refNum, line
            line = replaceString("\t", line, "␣")
            break
        else
            break
        endif
    while (1)
    if (strLen(lclComment) > 0)
        string /G comment = lclComment[0,strLen(lclComment)-2]
    endif
    // Optionszeile lesen
    string optLine = ""
    do
        if (cmpStr(line[0], "#") == 0)
            do
                line = line[1,inf]
            while (cmpStr(line[0], "␣") == 0)
            optLine = "#"+line[0,strLen(line)-2]
            fReadLine refNum, line
            line = replaceString("\t", line, "␣")
            break
        elseif (cmpStr(line[0], "␣") == 0)
            line = line[1,inf]
        elseif (cmpStr(line[0], "\r") == 0)
            fReadLine refNum, line
            line = replaceString("\t", line, "␣")
        elseif (cmpStr(line[0], "!") == 0)
            fReadLine refNum, line
            line = replaceString("\t", line, "␣")
        else
            break
        endif
    while (1)
    if (strLen(optLine) == 0)
        abort "Die␣Datei␣\""+filePath+"\"␣enthält␣keine␣Optionszeile␣(#...)."
    endif
    optLine = optLine[1,inf]
    // Einheit der Frequenz auswerten
    string unitStr = lowerStr(optLine[0,2])
    variable scale
    if (cmpstr(unitStr[0,1], "hz") == 0)
        scale = 1
        optLine = optLine[2,inf]
    else
        scale = 1000 ^ (whichListItem(unitStr, "khz;mhz;ghz;") + 1)
        if (scale == 1)
            abort "Die␣Einheit␣\""+unitStr+"\"␣der␣Frequenz␣in␣der␣Datei␣\""+filePath+"\"␣ist
                ␣ungültig."
        endif
        optLine = optLine[3,inf]
    endif
    do
        if (cmpStr(optLine[0], "␣") == 0)
            optLine = optLine[1,inf]
        else
            break
        endif
    while (1)
    // Parametertyp auswerten
    string parType = lowerStr(optLine[0])
    do
        optLine = optLine[1,inf]
    while (cmpStr(optLine[0], "␣") == 0)
    if (cmpStr(parType, "s") != 0)
        abort "Der␣Parametertyp␣\""+parType+"\"␣in␣der␣Datei␣\""+filePath+"\"␣wird␣nicht␣
            unterstützt."
    endif
```

```
    // Parameterformat auswerten
    string parFmtStr = lowerStr(optLine[0,1])
    optLine = optLine[2,inf]
    do
        if (cmpStr(optLine[0], "␣") == 0)
            optLine = optLine[1,inf]
        else
            break
        endif
    while (1)
    variable parFmt = whichListItem(parFmtStr, "db;ma;ri;")
    if (parFmt == -1)
        abort "Das␣Parameterformat␣\""+parFmtStr+"\"␣in␣der␣Datei␣\""+filePath+"\"␣ist␣
            ungültig."
    endif
    // Impedanz auswerten
    string zStr = lowerStr(optLine[0,inf])
    variable i = strsearch(zStr, "!", 0)
    if (i >= 0)
        zStr = zStr[0,i-1]
    endif
    variable z
    string s
    sscanf zStr+"$", "r␣%g␣%1s", z, s
    if (v_flag != 2 || cmpStr(s, "$") != 0)
        abort "Die␣Impedanz␣\""+zStr+"\"␣in␣der␣Datei␣\""+filePath+"\"␣ist␣ungültig."
    endif
    variable /G zLine = z
    // S-Parameter lesen
    variable allocStep = 20, xAlloc = 0, xSize = 0
    make /D /N=0 f
    variable m, n
    for (m = 1; m <= ports; m += 1)
        for (n = 1; n <= ports; n += 1)
            make /C /D /N=0 $("S"+num2str(m)+num2str(n))
        endfor
    endfor
    variable val1, val2
    variable /C cVal
    do
        if (cmpStr(line[0], "!") == 0)
            fReadLine refNum, line
            line = replaceString("\t", line, "␣")
        elseif (cmpStr(line[0], "␣") == 0)
            line = line[1,inf]
        elseif (cmpStr(line[0], "\r") == 0)
            fReadLine refNum, line
            line = replaceString("\t", line, "␣")
        elseif (strLen(line) == 0)
            break
        else
            val1 = tsReadFloat(filePath, refNum, line, "Die␣Frequenz")
            xSize += 1
            if (xSize > xAlloc)
                xAlloc += allocStep
                redimension /N=(xAlloc) f
                for (m = 1; m <= ports; m += 1)
                    for (n = 1; n <= ports; n += 1)
                        redimension /N=(xAlloc) $("S"+num2str(m)+num2str(n))
                    endfor
                endfor
            endif
            f[xSize-1] = scale*val1
            if (ports <= 2)
                for (n = 1; n <= ports; n += 1)
                    for (m = 1; m <= ports; m += 1)
                        val1 = tsReadFloat(filePath, refNum, line, "Der␣S-Parameter")
                        val2 = tsReadFloat(filePath, refNum, line, "Der␣S-Parameter")
                        switch (parFmt)
                            case 0: // DB
                                cVal = p2rect(cmplx(10^(val1/20), pi/180*val2))
                                break
                            case 1: // MA
                                cVal = p2rect(cmplx(val1, pi/180*val2))
                                break
                            case 2: // RI
```

```
                                cVal = cmplx(val1, val2)
                            endswitch
                            wave /C Smn = $("S"+num2str(m)+num2str(n))
                            Smn[xSize-1] = cVal
                        endfor
                    endfor
                    fReadLine refNum, line
                    line = replaceString("\t", line, " ")
                else
                    for (m = 1; m <= ports; m += 1)
                        for (n = 1; n <= ports; n += 1)
                            if (n >= 5 && mod(n, 4) == 1)
                                if (grepString(line, "^ *$"))
                                    fReadLine refNum, line
                                    line = trimString(replaceString("\t", line, " "))
                                endif
                            endif
                            val1 = tsReadFloat(filePath, refNum, line, "Der S-Parameter")
                            val2 = tsReadFloat(filePath, refNum, line, "Der S-Parameter")
                            switch (parFmt)
                                case 0: // DB
                                    cVal = p2rect(cmplx(10^(val1/20), pi/180*val2))
                                    break
                                case 1: // MA
                                    cVal = p2rect(cmplx(val1, pi/180*val2))
                                    break
                                case 2: // RI
                                    cVal = cmplx(val1, val2)
                            endswitch
                            wave /C Smn = $("S"+num2str(m)+num2str(n))
                            Smn[xSize-1] = cVal
                        endfor
                        fReadLine refNum, line
                        line = replaceString("\t", line, " ")
                        do
                            if (cmpStr(line[0], "!") == 0)
                                fReadLine refNum, line
                                line = replaceString("\t", line, " ")
                            elseif (cmpStr(line[0], " ") == 0)
                                line = line[1,inf]
                            elseif (cmpStr(line[0], "\r") == 0)
                                fReadLine refNum, line
                                line = replaceString("\t", line, " ")
                            else
                                break
                            endif
                        while(1)
                    endfor
                endif
            endif
        while (1)
        // prüfen, ob f äquidistant
        variable dMean = (f[xSize-1] - f[0]) / (xSize-1), dPnt
        variable eqFlag = 1
        for (i = 0; i < xSize-1; i += 1)
            dPnt = f[i+1] - f[i]
            if (dPnt > 1.0001*dMean || dPnt < 0.9999*dMean)
                eqFlag = 0
                break
            endif
        endfor
        // Waves beschneiden und skalieren
        redimension /N=(xSize) f
        setScale d 0, 0, "Hz", f
        for (m = 1; m <= ports; m += 1)
            for (n = 1; n <= ports; n += 1)
                redimension /N=(xSize) $("S"+num2str(m)+num2str(n))
                if (eqFlag)
                    setScale /I x f[0], f[xSize-1], "Hz", $("S"+num2str(m)+num2str(n))
                endif
            endfor
        endfor
        // Datei schließen
        close refNum

        setDataFolder oldDF
```

```
342     return 0
343 end
344 
345 // tsWrite() schreibt S-Parameter aus einem Data-Folder in eine Touchstone-Datei
346 // Wenn eine f-Wave vorhanden ist, werden die Frequenzen daraus verwendet. Falls nicht,
347 // wird davon ausgegangen, daß die Frequenzabstände äquidistant sind.
348 // Der Folder muß folgende Variablen enthalten, * = optional:
349 // nvar ports:              Anzahl der Ports (1..9)
350 // wave /C Snm:             komplexe S-Parameter Waves mit m=1..ports und n=1..ports
351 //                              nicht vorhandene Waves werden als -200 dB, 0° ausgegeben
352 // wave f*:                 Frequenzen [Hz]
353 // nvar zLine*:             Bezugsimpedanz [Ohm] (default = 50 Ohm)
354 // svar filePath*:          Pfad/Name der SnP-Datei, falls nicht vorhanden wird nachgefragt
355 // svar partName*:          Name des Bauteils, wird in die erste Kommentar-Zeile geschrieben
356 // svar comment*:           Kommentar, weitere Kommentarzeilen
357 // Eingang: folder:     Pfad des Data-Folders (z.B. "root:sMatrix")
358 //              lileDialog: Flag für File-Dialog anzeigen (1) oder nicht anzeigen (0)
359 //                              wenn filePath nicht existiert wird er auf jeden Fall
    angezeigt
360 // Rückgabe:    Datei abgespeichert (0) oder Abbruch (1)
361 function tsWrite(folder, fileDialog)
362     string  folder
363     variable    fileDialog
364 
365     string oldDF = getDataFolder(1)
366     setDataFolder $folder
367     nvar ports
368     svar /Z filePath
369     svar /Z partName
370     svar /Z comment
371     nvar /Z zLine
372 
373     // Bauteilename bereinigen
374     string part = "DUT"
375     variable i
376     string char
377     if (svar_exists(partName))
378         if (strlen(partName) > 0)
379             part = ""
380             for (i = 0; i < strLen(partName); i += 1)
381                 char = partName[i]
382                 if (char2num(char) < 0)
383                     char = comment[i,i+1]
384                 endif
385                 strswitch(char)
386                     case "ä":
387                         part += "ae"
388                         break
389                     case "ö":
390                         part += "oe"
391                         break
392                     case "ü":
393                         part += "ue"
394                         break
395                     case "ß":
396                         part += "ss"
397                         break
398                     case "Ä":
399                         part += "Ae"
400                         break
401                     case "Ö":
402                         part += "Oe"
403                         break
404                     case "Ü":
405                         part += "Ue"
406                         break
407                     case "␣":
408                     case "-":
409                     case "_":
410                         part += char
411                         break
412                     case "/":
413                     case "\\":
414                     case ",":
415                     case ";":
416                     case ".":
```

```
                    case ":":
                    case "=":
                        part += "-"
                        break
                    case "\t":
                        part += "␣"
                        break
                    default:
                        if (cmpStr(char, "0") >= 0 && cmpStr(char, "9") <= 0)
                            part += char
                        elseif (cmpStr(char, "a") >= 0 && cmpStr(char, "z") <= 0)
                            part += char
                        elseif (cmpStr(char, "A") >= 0 && cmpStr(char, "Z") <= 0)
                            part += char
                        endif
                endswitch
            endfor
        endif
    endif
    // Pfadnamen erzeugen
    if (!svar_exists(filePath))
        string /G filePath
        fileDialog = 1
        pathInfo home
        sprintf filePath, "%s%s.s%1up", s_path, part, ports
    endif
    // Datei öffnen
    string fileType
    if (stringmatch(IgorInfo(2), "Macintosh*"))
        fileType = "TEXT"
    else
        sprintf fileType, ".s%1up", ports
    endif
    variable refNum
    if (fileDialog)
        open /D /C="R*ch" /T=(fileType) /M="Touchstone-Datei␣speichern␣als␣..." /P=home
            refNum as filePath
        if (strLen(s_fileName) == 0)
            return 1
        endif
        filePath = s_fileName
    endif
    open /C="R*ch" /T=(fileType) /P=home refNum as filePath
    // Bauteilename ausgeben
    if (svar_exists(partName))
        if (strlen(partName) > 0)
            fprintf refNum, "!␣%s\r\n", part
        endif
    endif
    // Kommentar ausgeben
    variable x = 0
    if (svar_exists(comment))
        for (i = 0; strlen(comment[i]) != 0; i += 1)
            char = comment[i]
            if (char2num(char) < 0)
                char = comment[i,i+1]
            endif
            strswitch(char)
                case "\r":
                    if (cmpStr(comment[i+1], "\n") == 0)
                        do
                            i += 1
                        while (cmpStr(comment[i+1], "\n") == 0)
                    endif
                    fprintf refNum, "\r\n"
                    x = 0
                    break
                case "\n":
                    if (cmpStr(comment[i+1], "\r") == 0)
                        do
                            i += 1
                        while (cmpStr(comment[i+1], "\r") == 0)
                    endif
                    fprintf refNum, "\r\n"
                    x = 0
                    break
```

```
                default:
                    strswitch(char)
                        case "\t":
                            char = padString("", 4 - mod(x, 4), 32)
                            break
                        case "ä":
                            char = "ae"
                            break
                        case "ö":
                            char = "oe"
                            break
                        case "ü":
                            char = "ue"
                            break
                        case "ß":
                            char = "ss"
                            break
                        case "Ä":
                            char = "Ae"
                            break
                        case "Ö":
                            char = "Oe"
                            break
                        case "Ü":
                            char = "Ue"
                            break
                    endswitch
                    if (char2num(char) >= 32 && char2num(char) <= 126)
                        if (x == 0)
                            fprintf refNum, "!␣"
                        endif
                        fprintf refNum, "%s", char
                        x += strLen(char)
                    endif
            endswitch
        endfor
        if (x != 0)
            fprintf refNum, "\r\n"
        endif
    endif
    // Frequenzbereich überprüfen
    variable fStart = nan, fDelta = nan, fStop = nan, numPoints = nan
    string noPar = ""
    variable m, n
    wave /Z f
    if (waveExists(f))
        waveStats /Q /Z f
        fStart = v_min
        fStop = v_max
    else
        for (m = 1; m <= ports; m += 1)
            for (n = 1; n <= ports; n += 1)
                string wName = "S"+num2str(m)+num2str(n)
                wave /Z S = $wName
                if (waveExists(S))
                    variable wFStart = leftx(S), wFStop = rightx(S) - deltax(S)
                    variable wNumPoints = numpnts(S)
                    if ((numtype(fStart) != 2 && fStart != wFStart) || (numtype(fStop) != 2
                        && fStop != wFStop) || (numtype(numPoints) != 2 && numPoints !=
                        wNumPoints))
                        close refNum
                        setDataFolder oldDF
                        abort "tsWrite():␣Der␣Parameter␣S"+num2str(m)+num2str(n)+"␣hat␣eine␣
                            abweichende␣Skalierung"
                    endif
                    fStart = wFStart; fDelta = deltax(S); fStop = wFStop
                    numPoints = wNumPoints
                else
                    noPar += wName + ";"
                endif
            endfor
        endfor
    endif
    variable unit
    if (numtype(fStop) == 2)
        unit = 0
```

```
565     else
566         unit = min(max(floor(log(fStop) / 3), 0), 3)
567     endif
568     variable scale = 1000 ^ -unit
569     // nicht vorhandene S-Parameter anmerken
570     for (m = 0; m < itemsInList(noPar); m += 1)
571         string fmt
572         if (m == 0)
573             fmt = "!␣The␣parameters␣%s"
574         elseif (m == itemsInList(noPar) - 1)
575             fmt = "␣and␣%s␣are␣not␣measured.\r\n"
576         elseif (mod(m + 3, 15) == 0)
577             fmt = ",\r\n!␣%s"
578         else
579             fmt = ",␣%s"
580         endif
581         fprintf refNum, fmt, stringFromList(m, noPar)
582     endfor
583     // Header ausgeben
584     variable z = 50
585     if (nvar_exists(zLine))
586         z = zLine
587     endif
588     fprintf refNum, "#␣%s␣S␣DB␣R␣%g\r\n\r\n", stringFromList(unit, "HZ;KHZ;MHZ;GHZ;"), z
589     // S-Parameter ausgeben
590     fprintf refNum, "!␣freq[%s]", stringFromList(unit, "Hz␣;kHz;MHz;GHz;")
591     if (ports == 1)
592         fprintf refNum, "␣␣S11[dB]␣S11[deg]\r\n"
593     elseif (ports == 2)
594         fprintf refNum, "␣␣S11[dB]␣S11[deg]␣S21[dB]␣S21[deg]␣S12[dB]␣S12[deg]␣S22[dB]␣S22[deg
            ]\r\n"
595     else
596         fprintf refNum, "␣"
597         for (m = 1; m <= ports; m += 1)
598             if (m >= 5 && mod(m, 4) == 1)
599                 fprintf refNum, "\r\n!␣␣␣␣␣␣␣␣␣␣␣"
600             endif
601             fprintf refNum, "␣Sm%1u[dB]␣Sm%1u[deg]", m, m
602         endfor
603         fprintf refNum, "\r\n"
604     endif
605     for (i = 0; i < numPoints; i += 1)
606         fprintf refNum, "%11.9f", scale * (waveExists(f) ? f[i] : fStart + i * fDelta)
607         for (m = 1; m <= ports; m += 1)
608             if (ports >= 3 && m > 1)
609                 fprintf refNum, "\r\n␣␣␣␣␣␣␣␣␣␣␣"
610             endif
611             for (n = 1; n <= ports; n += 1)
612                 if (ports >= 3)
613                     wave /Z S = $("S"+num2str(m)+num2str(n))
614                 else
615                     wave /Z S = $("S"+num2str(n)+num2str(m))
616                 endif
617                 if (n >= 5 && mod(n, 4) == 1)
618                     fprintf refNum, "\r\n␣␣␣␣␣␣␣␣␣␣␣"
619                 endif
620                 if (waveExists(S))
621                     fprintf refNum, "␣%8.3f␣%7.2f", 20*log(real(r2polar(S[i]))), 180/pi*imag(
                        r2polar(S[i]))
622                 else
623                     fprintf refNum, "␣%8.3f␣%7.2f", -200, 0
624                 endif
625             endfor
626         endfor
627         fprintf refNum, "\r\n"
628     endfor
629     // Datei schließen
630     close refNum
631
632     setDataFolder oldDF
633     return 0
634 end
```

6.4.5 Quelltext devIO_1.03.ipf

Die Datei `devIO_1.03.ipf` in den gemeinsamen Igor Pro includes enthält Funktionen zur Kapselung der Meßgeräte Ein-/Ausgaben. Sie wird von dem Modul `Agilent_MXA_1.05.ipf` benötigt.

```
1  //-------------------------------------------------------------------------
2  // Modul zur Kapselung der Meßgeräte Ein-/Ausgaben
3  // Initialisierung bei Programmstart: ioInit()
4  //
5  // (c) 2016,2021 Claudius Peschke,
6  // Gesellschaft für Schwerionenforschung mbH
7  //
8  // Version 1.00:    erste Version für VISA
9  // Version 1.01:    Update auf Igor Pro 7
10 // Version 1.02:    ioGetStr() und ioGetNStr()
11 //                  um optionalen Parameter tmoOk erweitert
12 // Version 1.03:    ingorieren von Übertragungsfehlern ermöglichen
13 //-------------------------------------------------------------------------
14
15 #pragma TextEncoding = "UTF-8"
16 #pragma rtGlobals=3
17
18 //------------------
19 // Konstanten
20 //------------------
21
22 constant ioDebug = 0
23
24 //------------------------
25 // Hilfsfunktionen
26 //------------------------
27
28 // ioInit() wird beim laden des Moduls aufgerufen und initialisiert alle nötigen Variablen
29 // Die Funktion darf mehrfach aufgerufen werden.
30 function ioInit()
31     ioChkFolder(1)
32 end
33
34 // ioChkFolder() erzeugt, falls nicht vorhanden den Ordner "root:io" mit allen notwendigen
       Variablen.
35 // Falls bereits vorhanden, werden deren Werte auf Gültigkeit geprüft und ggf. korrigiert.
36 // Eingang: init:   Auch bereits vorhandene Variablen mit Default-Werten initialisieren (1)
       oder nicht (0)
37 function ioChkFolder(init)
38     variable init
39
40     string oldDF = getDataFolder(1)
41     variable old = dataFolderExists("root:io")
42     newDataFolder /O /S root:io // Ordner und Inhalt erzeugen, falls nicht vorhanden
43     variable /G viRM                    //  VISA Session des Rescource Managers oder 0 für
           geschlossen
44     string /G instList              //  Semikolon-sepatierte Liste der VISA Instrumente
45     if (init || !old)
46         viRM = 0
47         instList = ""
48     endif
49     setDataFolder oldDF
50 end
51
52 // ioGetName() liefert einen Namen für ein Gerät für Fehlermeldungen
53 // Eingang: ioFolder:   Pfad des Data Folders mit den Variablen ioDev und optional ioName und
       ioPath
54 function /S ioGetName(ioFolder)
55     string ioFolder
56
57     string viName
58     if (exists(ioFolder + ":ioName") == 2)
59         sVar ioName = $(ioFolder + ":ioName")
60         viName = ioName
61     elseif (exists(ioFolder + ":ioPath") == 2)
62         sVar ioPath = $(ioFolder + ":ioPath")
63         viName = "Gerät␣" + ioPath
64     else
65         viName = "Gerät"
```

```
66      endif
67      return viName
68  end
69
70  // ioQErr() bricht das Programm mit einer Fehlermeldung ab, falls der VISA-Status einen
71  // Fehler anzeigt. Die Session und viRM werden geschlossen, falls sie geöffnet waren.
72  // Eingang: ioFolder:   Pfad des Data Folders mit den Variablen ioDev und optional ioName und
        ioPatht oder "" für viRM
73  //              viStat: VISA-Status
74  //              preStr: String, der der VISA-Meldung vorangestellt wird oder "" für "VISA
        error"
75  function ioQErr(ioFolder, viStat, preStr)
76      string ioFolder, preStr
77      variable viStat
78
79      if (viStat < VI_SUCCESS)
80          string errStr, msg
81          if (strlen(ioFolder) && exists(ioFolder + ":ioDev") == 2)
82              nVar viObj = $(ioFolder + ":ioDev")
83              viStatusDesc(viObj, viStat, errStr)
84              if (viObj)
85                  viClose(viObj)
86                  viObj = 0
87              endif
88          elseif (exists("root:io:viRM") == 2)
89              nVar viRM = root:nrp2:viRM
90              viStatusDesc(viRM, viStat, errStr)
91          endif
92          if (exists("root:io:viRM") == 2)
93              nVar viRM = root:nrp2:viRM
94              if (viRM)
95                  viClose(viRM)
96                  viRM = 0
97              endif
98          endif
99          if (strLen(preStr) == 0)
100             preStr = "VISA␣error"
101         endif
102         setDataFolder "root:"
103         sprintf msg, "%s:\r%s\r", preStr, errStr
104         abort msg
105     endif
106 end
107
108 // ioOpenRM() öffnet die Resource Manager Session, falls sie geschlossen war
109 function ioOpenRM()
110     ioChkFolder(0)
111     nVar viRM = root:io:viRM
112     variable viTmpRM, viStat
113     if (viRM == 0)
114         viStat = viOpenDefaultRM(viTmpRM)
115         variable dummy = 0
116         ioQErr("", viStat, "VISA␣error␣opening␣resource␣manager")
117         viRM = viTmpRM
118     endif
119 end
120
121 // ioChkOpen() prüft, ob das Gerät geöffnet ist. Falls nicht wird eine Fehlermeldung
        ausgegeben.
122 // Eingang: ioFolder:   Pfad des Data Folders für die Variablen ioPath, ioDev und optional
        ioName
123 //              fName:  Name der aufrufenden Funktion ohne Klammern
124 function ioChkOpen(ioFolder, fName)
125     string ioFolder, fName
126
127     variable dev = 0
128     if (exists(ioFolder + ":ioDev") == 2)
129         nVar ioDev = $(ioFolder + ":ioDev")
130         dev = ioDev
131     endif
132     if (dev == 0)
133         string msg
134         sprintf msg, fName + "():␣%s␣ist␣nicht␣geöffnet", ioGetName(ioFolder)
135         abort msg
136     endif
137 end
```

```
138
139 // -------------------------------
140 // Steuerungsfunktionen
141 //-------------------------------
142
143 // ioOpen() öffnet ein Gerät
144 // Eingang: ioFolder:   Pfad des Data Folders für die Variablen ioPath, ioDev und optional
        ioName
145 //              ioPath: VISA Pfad des Geräts
146 //              ioName: Name des Geräts für Meldungen oder ""
147 function ioOpen(ioFolder, ioPath, ioName)
148     string ioFolder, ioPath, ioName
149
150     ioChkFolder(0)
151     string oldDF = getDataFolder(1)
152     setDataFolder root:io
153     nVar viRM
154
155     if (ioDebug)
156         printf "ioOpen(\"%s\",␣\"%s\")\r", ioFolder, ioPath
157     endif
158     string /G $(ioFolder + ":ioPath")
159     sVar gPath = $(ioFolder + ":ioPath")
160     gPath = ioPath
161     if (strlen(ioName))
162         string /G $(ioFolder + ":ioName")
163         sVar gName = $(ioFolder + ":ioName")
164         gName = ioName
165     endif
166     if (exists(ioFolder + ":ioDev") != 2)
167         variable /G $(ioFolder + ":ioDev") = 0
168     endif
169     nVar ioDev = $(ioFolder + ":ioDev")
170     if (ioDev == 0)
171         ioOpenRM()
172         variable viTmpDev
173         variable viStat = viOpen(viRM, ioPath, 0, 0, viTmpDev)
174         ioQErr("", viStat, "ioOpen():␣Fehler␣beim␣Öffnen␣von␣" + ioGetName(ioFolder))
175         ioDev = viTmpDev
176     endif
177
178     setDataFolder oldDF
179 end
180
181 // ioClose() schließt die Schnittstelle
182 // Eingang: ioFolder: Pfad des Data Folders mit den Variablen ioDev und optional ioName und
        ioPath
183 function ioClose(ioFolder)
184     string ioFolder
185
186     if (ioDebug)
187         printf "ioClose(\"%s\")\r", ioFolder
188     endif
189     ioChkOpen(ioFolder, "ioClose")
190     nVar ioDev = $(ioFolder + ":ioDev")
191     viClose(ioDev)
192     ioDev = 0
193 end
194
195 // ioSetTmo() setzt den Timeout für die Schnittstelle
196 // Eingang: ioFolder:   Pfad des Data Folders mit den Variablen ioDev und optional ioName und
         ioPath
197 //              secs:   gewünschter Timeout in s
198 function ioSetTmo(ioFolder, secs)
199     string ioFolder
200     variable secs
201
202     if (ioDebug)
203         printf "ioSetTmo(\"%s\",␣%f)\r", ioFolder, secs
204     endif
205     ioChkOpen(ioFolder, "ioSetTmo")
206     nVar ioDev = $(ioFolder + ":ioDev")
207     variable viStat = viSetAttribute(ioDev, VI_ATTR_TMO_VALUE, 1000 * secs)
208     ioQErr(ioFolder, viStat, "")
209 end
210
```

```
211 // ioSetEnd() setzt den Endzeichen-Modus
212 // Eingang: ioFolder:   Pfad des Data Folders mit den Variablen ioDev und optional ioName und
        ioPath
213 //              term:       ASCII-Wert des Endzeichens
214 //              eoi:        bei IEEE-488 EOI mit letztem Zeichen senden (1=ja, 0=nein)
215 //              outchar:    term-Zeichen nach letztem Byte senden (1=ja, 0=nein)
216 //              inchar: bei term-Zeichen Eingabe beenden (1=ja, 0=nein)
217 function ioSetEnd(ioFolder, term, eoi, outchar, inchar)
218     string ioFolder
219     variable term, eoi, outchar, inchar
220
221     if (ioDebug)
222         printf "ioSetTmo(\"%s\",␣%f,␣%f,␣%f,␣%f)\r", ioFolder, term, eoi, outchar, inchar
223     endif
224     switch (eoi)
225         case 0:
226             eoi = VI_FALSE
227             break
228         case 1:
229             eoi = VI_TRUE
230             break
231         default:
232             abort "ioSetEnd():␣ungültiger␣Parameter␣eoi:␣" + num2str(eoi)
233     endswitch
234     switch (outchar)
235         case 0:
236             outchar = VI_ASRL_END_NONE
237             break
238         case 1:
239             outchar = VI_ASRL_END_TERMCHAR
240             break
241         default:
242             abort "ioSetEnd():␣ungültiger␣Parameter␣outchar:␣" + num2str(outchar)
243     endswitch
244     switch (inchar)
245         case 0:
246             inchar = VI_ASRL_END_NONE
247             break
248         case 1:
249             inchar = VI_ASRL_END_TERMCHAR
250             break
251         default:
252             abort "ioSetEnd():␣ungültiger␣Parameter␣inchar:␣" + num2str(inchar)
253     endswitch
254     ioChkOpen(ioFolder, "ioSetEnd")
255     nVar ioDev = $(ioFolder + ":ioDev")
256     variable viStat
257     viStat = viSetAttribute(ioDev, VI_ATTR_TERMCHAR, term)
258     ioQErr(ioFolder, viStat, "")
259     viStat = viSetAttribute(ioDev, VI_ATTR_SEND_END_EN, eoi)
260     ioQErr(ioFolder, viStat, "")
261     viStat = viSetAttribute(ioDev, VI_ATTR_TERMCHAR_EN, (inchar || outchar) ? VI_TRUE :
            VI_FALSE)
262     ioQErr(ioFolder, viStat, "")
263     viStat = viSetAttribute(ioDev, VI_ATTR_ASRL_END_OUT, outchar)
264     ioQErr(ioFolder, viStat, "")
265     viStat = viSetAttribute(ioDev, VI_ATTR_ASRL_END_IN, inchar)
266     ioQErr(ioFolder, viStat, "")
267 end
268
269 // ioSetSerPar() setzt die Parameter einer seriellen Schnittstelle
270 // Eingang: ioFolder:   Pfad des Data Folders mit den Variablen ioDev und optional ioName und
        ioPath
271 //              baud:       Baudrate
272 //              bits        Anzahl der Datenbits (5-8)
273 //              parity: Paritätsbit (0=keines, 1=ungerade, 2=gerade, 3=mark, 4=space)
274 //              stop:       Anzahl der Stopbits (1, 1.5 oder 2)
275 //              flow:       Flußkontrolle (0=keine, 1=XON/XOFF, 2=RTS/CTS, 3=DTR/DSR)
276 function ioSetSerPar(ioFolder, baud, bits, parity, stop, flow)
277     string ioFolder
278     variable baud, bits, parity, stop, flow
279
280     if (ioDebug)
281         printf "ioSetSerPar(\"%s\",␣%f,␣%f,␣%f,␣%f,␣%f)\r", ioFolder, baud, bits, parity,
                stop, flow
282     endif
```

```
283     if (baud <= 0 || baud > 1e9)
284         abort "ioSetSerPar():␣ungültige␣Parameter␣Baudrate:␣" + num2str(baud) + "␣baud"
285     endif
286     if (bits < 5 || bits > 8)
287         abort "ioSetSerPar():␣ungültige␣Parameter␣Anzahl␣Datenbits:␣" + num2str(bits)
288     endif
289     switch (parity)
290         case 0:
291             parity = VI_ASRL_PAR_NONE
292             break
293         case 1:
294             parity = VI_ASRL_PAR_ODD
295             break
296         case 2:
297             parity = VI_ASRL_PAR_EVEN
298             break
299         case 3:
300             parity = VI_ASRL_PAR_MARK
301             break
302         case 4:
303             parity = VI_ASRL_PAR_SPACE
304             break
305         default:
306             abort "ioSetSerPar():␣ungültiger␣Parameter␣Parität:␣" + num2str(parity)
307     endswitch
308     switch (10 * stop)
309         case 10:
310             stop = VI_ASRL_STOP_ONE
311             break
312         case 15:
313             stop = VI_ASRL_STOP_ONE5
314             break
315         case 20:
316             stop = VI_ASRL_STOP_TWO
317             break
318         default:
319             abort "ioSetSerPar():␣ungültige␣Parameter␣Anzahl␣Stopbits:␣" + num2str(stop)
320     endswitch
321     switch (flow)
322         case 0:
323             flow = VI_ASRL_FLOW_NONE
324             break
325         case 1:
326             flow = VI_ASRL_FLOW_XON_XOFF
327             break
328         case 2:
329             flow = VI_ASRL_FLOW_RTS_CTS
330             break
331         case 3:
332             flow = VI_ASRL_FLOW_DTR_DSR
333             break
334         default:
335             abort "ioSetSerPar():␣ungültige␣Parameter␣Flußkontrolle:␣" + num2str(flow)
336     endswitch
337     ioChkOpen(ioFolder, "ioSetSerPar")
338     nVar ioDev = $(ioFolder + ":ioDev")
339     variable viStat
340     viStat = viSetAttribute(ioDev, VI_ATTR_ASRL_BAUD, baud)
341     ioQErr(ioFolder, viStat, "")
342     viStat = viSetAttribute(ioDev, VI_ATTR_ASRL_DATA_BITS, bits)
343     ioQErr(ioFolder, viStat, "")
344     viStat = viSetAttribute(ioDev, VI_ATTR_ASRL_PARITY, parity)
345     ioQErr(ioFolder, viStat, "")
346     viStat = viSetAttribute(ioDev, VI_ATTR_ASRL_STOP_BITS, stop)
347     ioQErr(ioFolder, viStat, "")
348     viStat = viSetAttribute(ioDev, VI_ATTR_ASRL_FLOW_CNTRL, flow)
349     ioQErr(ioFolder, viStat, "")
350 end
351
352 // ioInstList() holt von VISA die Liste der definierten Instrumente als
353 // Semikolon-sepatierte Liste und legt sie unter root:io:instList ab
354 function ioInstList()
355     ioChkFolder(0)
356     ioOpenRM()
357     nVar viRM = root:io:viRM
358     // erstes Instrument holen
```

```
359     string devList = ""
360     string viRsrc
361     variable viList, nDevs
362     variable viStat = viFindRsrc(viRM, "(GPIB|TCPIP|ASRL)[0-9]?*:INSTR", viList, nDevs,
            viRsrc)
363     if (viStat < VI_SUCCESS)
364         if (viStat != VI_ERROR_RSRC_NFOUND)
365             ioQErr("", viStat, "VISA error while searching for instruments")
366         endif
367     else
368         variable i
369         for (i = 1; i <= nDevs; i += 1)
370             devList += viRsrc + ";"
371             // nächstes Instrument holen
372             if ( i < nDevs)
373                 viStat = viFindNext(viList, viRsrc)
374                 ioQErr("", viStat, "VISA error while searching for instruments")
375             endif
376         endfor
377         viClose(viList)
378     endif
379     sVar instList = root:io:instList
380     instList = devList
381 end
382
383 //-----------------------------
384 // Ausgabefunktionen
385 //-----------------------------
386
387 // ioPutStr() sendet einen String
388 // Eingang: ioFolder:   Pfad des Data Folders mit den Variablen ioDev und optional ioName und
        ioPath
389 //              str:        zu sendender String
390 //              stopOnError:    bei Übertragungsfehlern abbrechen (1) oder
391 //                              Fehler zurückgeben (0) (optional, default 1)
392 // Rückgabe: OK (0) oder VISA-Fehlernummer (<0) oder unvollständig gesendet (1)
393 function ioPutStr(ioFolder, str, [stopOnError])
394     string ioFolder
395     string str
396     variable stopOnError
397
398     if (ioDebug)
399         printf "ioPutStr(\"%s\", \"%s\")\r", ioFolder, str[0,999]
400     endif
401     if (paramIsDefault(stopOnError))
402         stopOnError = 1
403     endif
404     ioChkOpen(ioFolder, "ioPutStr")
405     nVar ioDev = $(ioFolder + ":ioDev")
406     variable written, len = strLen(str)
407     variable viStat = viWrite(ioDev, str, len, written)
408     if (stopOnError)
409         ioQErr(ioFolder, viStat, "ioPutStr(): Fehler beim Schreiben an " + ioGetName(ioFolder
                ))
410     elseif (viStat < VI_SUCCESS)
411         return viStat
412     endif
413     if (written < len)
414         if (stopOnError)
415             abort "ioPutStr(): Unvollständige Ausgabe beim Schreiben an " + ioGetName(
                    ioFolder)
416         else
417             return 1
418         endif
419     endif
420     return 0
421 end
422
423 // ioPutMsg() sendet eine Anzahl von Bytes aus einer Integer-Wave
424 // Eingang: ioFolder:   Pfad des Data Folders mit den Variablen ioDev und optional ioName und
        ioPath
425 //              buf:            zu sendende Wave
426 //              nBytes:     Anzahl der Bytes
427 //              stopOnError:    bei Übertragungsfehlern abbrechen (1) oder
428 //                              Fehler zurückgeben (0) (optional, default 1)
429 // Rückgabe: OK (0) oder VISA-Fehlernummer (<0) oder unvollständig gesendet (1)
```

```
430 function ioPutMsg(ioFolder, buf, nBytes, [stopOnError])
431     string ioFolder
432     wave /U /B buf
433     variable nBytes, stopOnError
434 
435     if (ioDebug)
436         printf "ioPutMsg(\"%s\",␣%d):␣", ioFolder, nBytes
437         variable n
438         for (n = 0; n < nBytes; n += 1)
439             printf "%02X␣", buf[n]
440         endfor
441         printf "\r"
442     endif
443     if (paramIsDefault(stopOnError))
444         stopOnError = 1
445     endif
446     ioChkOpen(ioFolder, "ioPutMsg")
447     string bufStr = ""
448     variable i
449     for (i = 0; i < nBytes; i += 1)
450         bufStr += num2char(buf[i], 1)
451     endfor
452     nVar ioDev = $(ioFolder + ":ioDev")
453     variable written
454     variable viStat = viWrite(ioDev, bufStr, nBytes, written)
455     if (stopOnError)
456         ioQErr(ioFolder, viStat, "ioPutMsg():␣Fehler␣beim␣Schreiben␣an␣" + ioGetName(ioFolder
            ))
457     elseif (viStat < VI_SUCCESS)
458         return viStat
459     endif
460     if (written != nBytes)
461         if (stopOnError)
462             abort "ioPutMsg():␣Unvollständige␣Ausgabe␣beim␣Schreiben␣an␣" + ioGetName(
                ioFolder)
463         else
464             return 1
465         endif
466     endif
467     return 0
468 end
469 
470 //------------------------------
471 // Eingabefunktionen
472 //------------------------------
473 
474 // ioGetStr() empfängt einen String
475 // Eingang: ioFolder:   Pfad des Data Folders mit den Variablen ioDev und optional ioName und
      ioPath
476 //              cut:        Flag für CR, LF und NUL am Ende abschneiden (1) oder nicht (0)
477 //              tmoOk:      Behandlung von Timeout oder Fehler: 0=Fehler, 1=abbrechen (
    optional, default: 0)
478 // Rückgabe: String
479 function /S ioGetStr(ioFolder, cut, [tmoOk])
480     string ioFolder
481     variable cut, tmoOk
482 
483     if (ioDebug)
484         printf "ioGetStr(\"%s\",␣%d)\r", ioFolder, cut
485     endif
486     ioChkOpen(ioFolder, "ioGetStr")
487     if(paramIsDefault(tmoOk))
488         tmoOk = 0
489     endif
490     nVar ioDev = $(ioFolder + ":ioDev")
491     variable eot
492     string msg = ""
493     do
494         msg += ioGetNStr(ioFolder, 255, cut, eot, tmoOk=tmoOk)
495     while (!eot)
496     return msg
497 end
498 
499 // ioGetNStr() empfängt einen String mit einer maximalen Länge
500 // Eingang: ioFolder:   Pfad des Data Folders mit den Variablen ioDev und optional ioName und
      ioPath
```

```
501 //              maxlen: maximale Anzahl zu lesender Zeichen
502 //              cut:        Flag für CR, LF und NUL am Ende abschneiden (1) oder nicht (0)
503 //              eot:        wird gesetzt wenn das letzte Zeichen gelesen wurde (EOS oder EOI)
504 //              tmoOk:      Behandlung von Timeout oder Fehler: 0=Fehler, 1=abbrechen (
        optional, default: 0)
505 // Rückgabe: String
506 function /S ioGetNStr(ioFolder, maxlen, cut, eot, [tmoOk])
507     string ioFolder
508     variable maxlen, cut, &eot, tmoOk
509
510     if (ioDebug)
511         printf "ioGetNStr(\"%s\",␣maxlen=%d,␣cut=%d,␣eot)", ioFolder, maxlen, cut
512     endif
513     ioChkOpen(ioFolder, "ioGetStr")
514     nVar ioDev = $(ioFolder + ":ioDev")
515     if(paramIsDefault(tmoOk))
516         tmoOk = 0
517     endif
518     string msgBuf
519     variable nRead
520     variable viStat = viRead(ioDev, msgBuf, maxlen, nRead)
521     variable c = char2num(msgBuf[strlen(msgBuf) - 1])
522     eot = (c == 10 || c == 13 || nRead < maxlen || (nRead == maxlen && viStat !=
        VI_SUCCESS_MAX_CNT))
523     if (tmoOk && viStat < VI_SUCCESS)
524         if (nRead == 0)
525             msgBuf = ""
526         endif
527         eot = 1
528     else
529         ioQErr(ioFolder, viStat, "ioGetNStr():␣Fehler␣beim␣Lesen␣von␣" + ioGetName(ioFolder))
530     endif
531     if (cut)
532         variable i = strlen(msgBuf) - 1
533         do
534             c = char2num(msgBuf[i])
535             if (c != 0 && c != 10 && c != 13)
536                 break
537             endif
538             i -= 1
539         while (i >= 0)
540         msgBuf = msgBuf[0,i]
541     endif
542     if (ioDebug)
543         printf "␣->␣eot=%d,␣return=\"%s\"\r", eot, msgBuf[0,999]
544     endif
545     return msgBuf
546 end
547
548 // ioGetFlt() empfängt eine Gleitkommazahl
549 // Eingang: ioFolder:   Pfad des Data Folders mit den Variablen ioDev und optional ioName und
      ioPath
550 //              tmoOk:      Behandlung von Timeout oder Fehler: 0=Fehler, 1=abbrechen (
        optional, default: 0)
551 // Rückgabe:    Gleitkommazahl
552 function ioGetFlt(ioFolder, [tmoOk])
553     string ioFolder
554     variable tmoOk
555
556     if(paramIsDefault(tmoOk))
557         tmoOk = 0
558     endif
559     return str2num(ioGetStr(ioFolder, 1, tmoOk=tmoOk))
560 end
561
562 // ioIDN() sendet "*IDN?" an das Gerät und empfängt die Antwort.
563 // Eingang: ioFolder:   Pfad des Data Folders mit den Variablen ioDev und optional ioName und
      ioPath
564 // Rückgabe:    Identifikationsstring oder "" für keine Antwort innerhalb 300 ms
565 function /S ioIDN(ioFolder)
566     string ioFolder
567
568     ioChkOpen(ioFolder, "ioIDN")
569     nVar ioDev = $(ioFolder + ":ioDev")
570     variable viStat, cnt
571     viStat = viSetAttribute(ioDev, VI_ATTR_TMO_VALUE, 300)
```

```
572     ioQErr(ioFolder, viStat, "ioIDN():␣VISA␣error")
573     string msgBuf = ""
574     viStat = viWrite(ioDev, "*IDN?\r", 6, cnt)
575     if (viStat >= VI_SUCCESS)
576         viStat = viRead(ioDev, msgBuf, 80, cnt)
577         if (viStat < VI_SUCCESS)
578             msgBuf = ""
579         endif
580     endif
581     return msgBuf
582 end
583 
584 // ioTryIDN() sendet "*IDN?" an ein Gerät und empfängt die Antwort. Das Gerät muß nicht
        geöffnet sein.
585 // Eingang: ioPath: VISA Pfad des Geräts
586 // Rückgabe:    Identifikationsstring oder "" für keine Antwort innerhalb 300 ms
587 function /S ioTryIDN(ioPath)
588     string ioPath
589 
590     ioOpenRM()
591     string msgBuf = ""
592     nVar viRM = root:io:viRM
593     variable viTmpDev
594     variable viStat = viOpen(viRM, ioPath, 0, 0, viTmpDev)
595     if (viStat >= VI_SUCCESS)
596         viGpibSendIFC(viTmpDev)
597         viStat = viSetAttribute(viTmpDev, VI_ATTR_TMO_VALUE, 300)
598         ioQErr("", viStat, "ioTryIDN():␣VISA␣error")
599         variable cnt
600         viStat = viWrite(viTmpDev, "*IDN?\r", 6, cnt)
601         if (viStat >= VI_SUCCESS)
602             viStat = viRead(viTmpDev, msgBuf, 80, cnt)
603             if (viStat < VI_SUCCESS)
604                 msgBuf = ""
605             endif
606         endif
607         viStat = viClose(viTmpDev)
608         ioQErr("", viStat, "ioTryIDN():␣VISA␣error")
609     endif
610     return msgBuf
611 end
612 
613 // ioGetMsg() empfängt einen String mit einer bestimmten Länge
614 // Eingang: ioFolder:   Pfad des Data Folders mit den Variablen ioDev und optional ioName und
        ioPath
615 //              buf:        zu empfangende Wave
616 //              offset: Index der ersten abgelegten Bytes in der Wave
617 //              nBytes: Anzahl der Bytes
618 //              tmoOk:      Behandlung von Timeout oder Fehler: 0=Fehler, 1=abbrechen (
        optional, default: 0)
619 // Rückgabe: OK (0) oder VISA-Fehlernummer (<0) oder unvollständig empfangen (1)
620 function ioGetMsg(ioFolder, buf, offset, nBytes, [tmoOk])
621     string ioFolder
622     wave /U /B buf
623     variable offset, nBytes, tmoOk
624 
625     if (ioDebug)
626         printf "ioGetMsg(\"%s\",␣buf,␣%d)\r", ioFolder, nBytes
627     endif
628     if(paramIsDefault(tmoOk))
629         tmoOk = 0
630     endif
631     ioChkOpen(ioFolder, "ioGetMsg")
632     nVar ioDev = $(ioFolder + ":ioDev")
633     string bufStr
634     variable nRead
635     variable viStat = viRead(ioDev, bufStr, nBytes, nRead)
636 
637     if (tmoOk && viStat < VI_SUCCESS)
638         return viStat
639     else
640         ioQErr(ioFolder, viStat, "ioGetMsg():␣Fehler␣beim␣Lesen␣von␣" + ioGetName(ioFolder))
641     endif
642     variable i
643     for (i = 0; i < nRead; i++)
644         buf[i + offset] = char2num(bufStr[i]) & 0xFF
```

```
645         endfor
646         if (nRead != nBytes)
647             if (tmoOk)
648                 return 1
649             else
650                 viClose(ioDev)
651                 ioDev = 0
652                 abort "ioGetMsg():␣Unvollständige␣Nachricht␣beim␣Lesen␣von␣" + ioGetName(ioFolder
                        )
653             endif
654         endif
655         if (ioDebug)
656             printf "␣->␣"
657             variable n
658             for (n = 0; n < nBytes; n += 1)
659                 printf "%02X␣", buf[n + offset]
660             endfor
661             printf "\r"
662         endif
663         return 0
664 end
```

7 Blockschaltbilder stochastisches Kühlsystem

Die folgenden Seiten zeigen den Aufbau des stochastischen Kühlsystems des ESR zum Zeitpunkt des Maschinenexperiments als Blockschaltbilder.

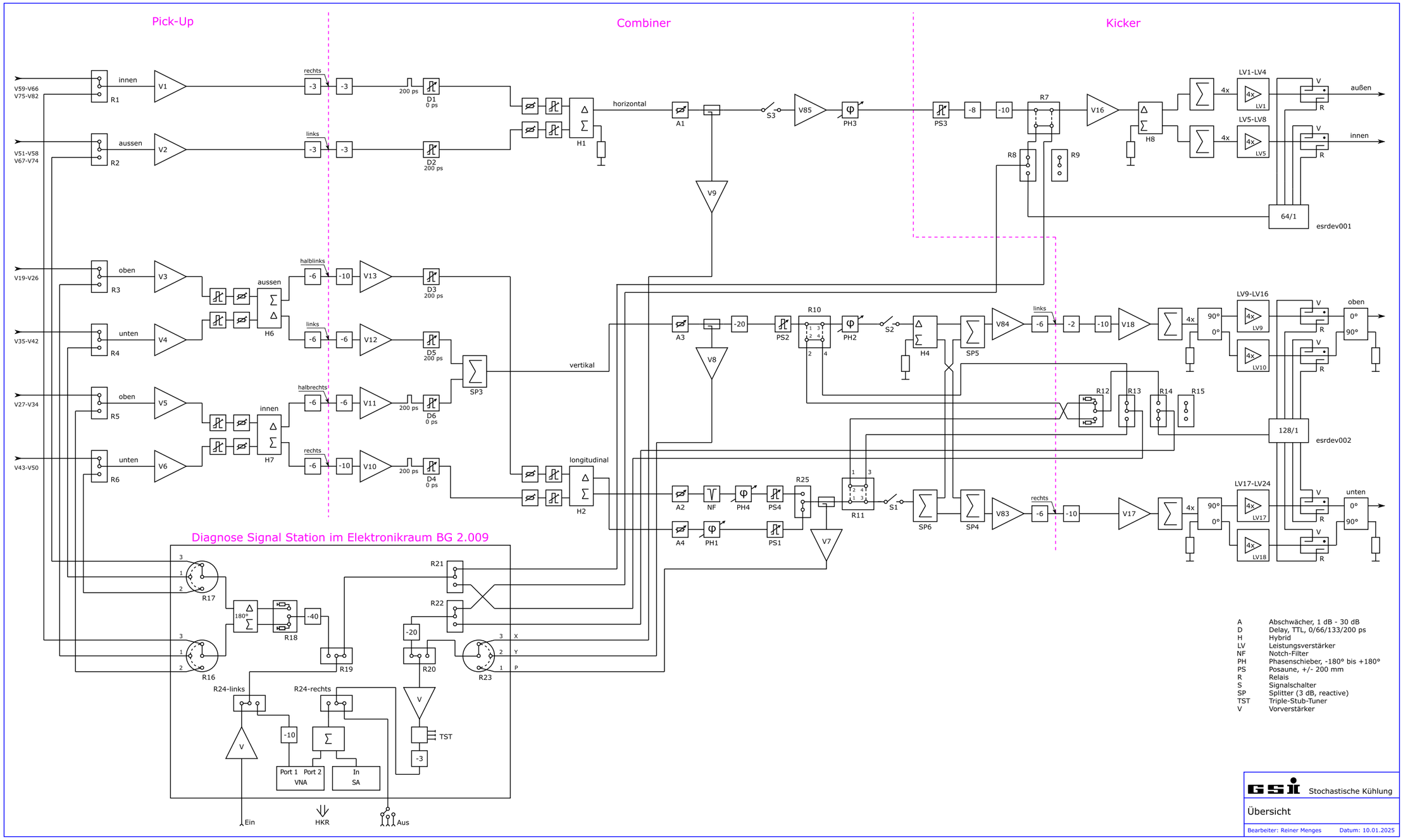


Abbildung 7.1: Blockschaltbild Übersicht

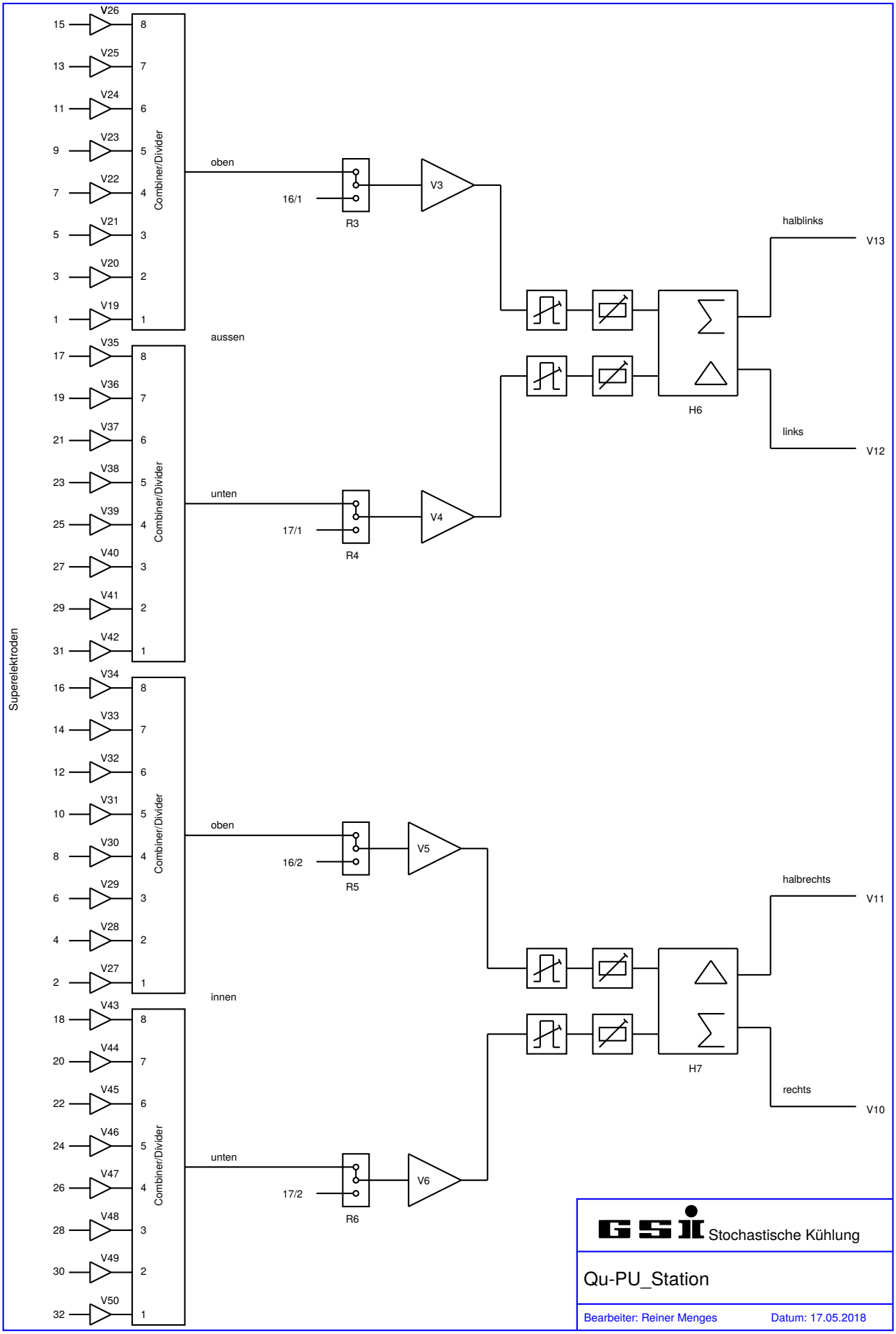


Abbildung 7.2: Blockschaltbild Quadrupol Pick-up-Station

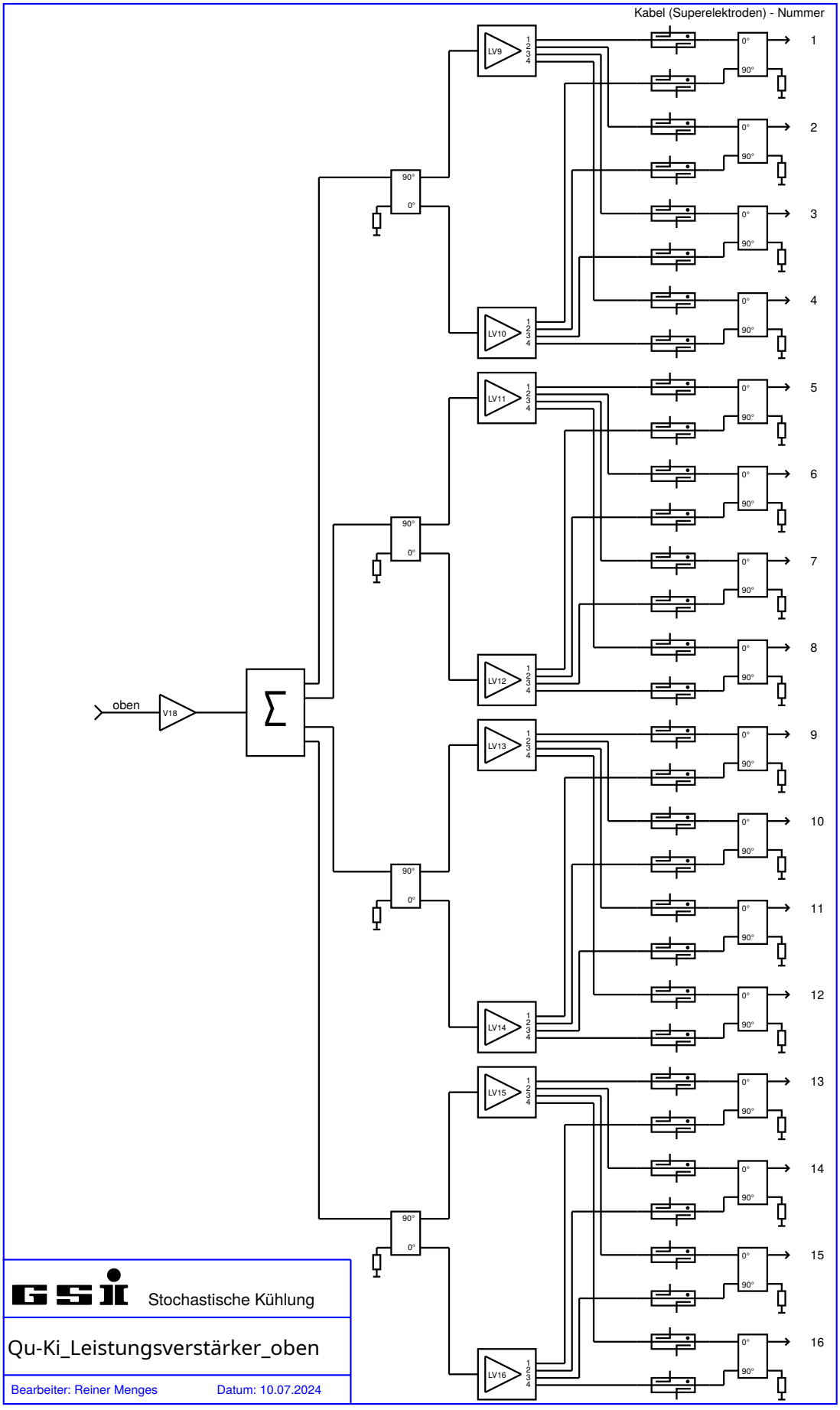


Abbildung 7.3: Blockschaltbild Quadrupol Kicker-Station, obere Elektroden

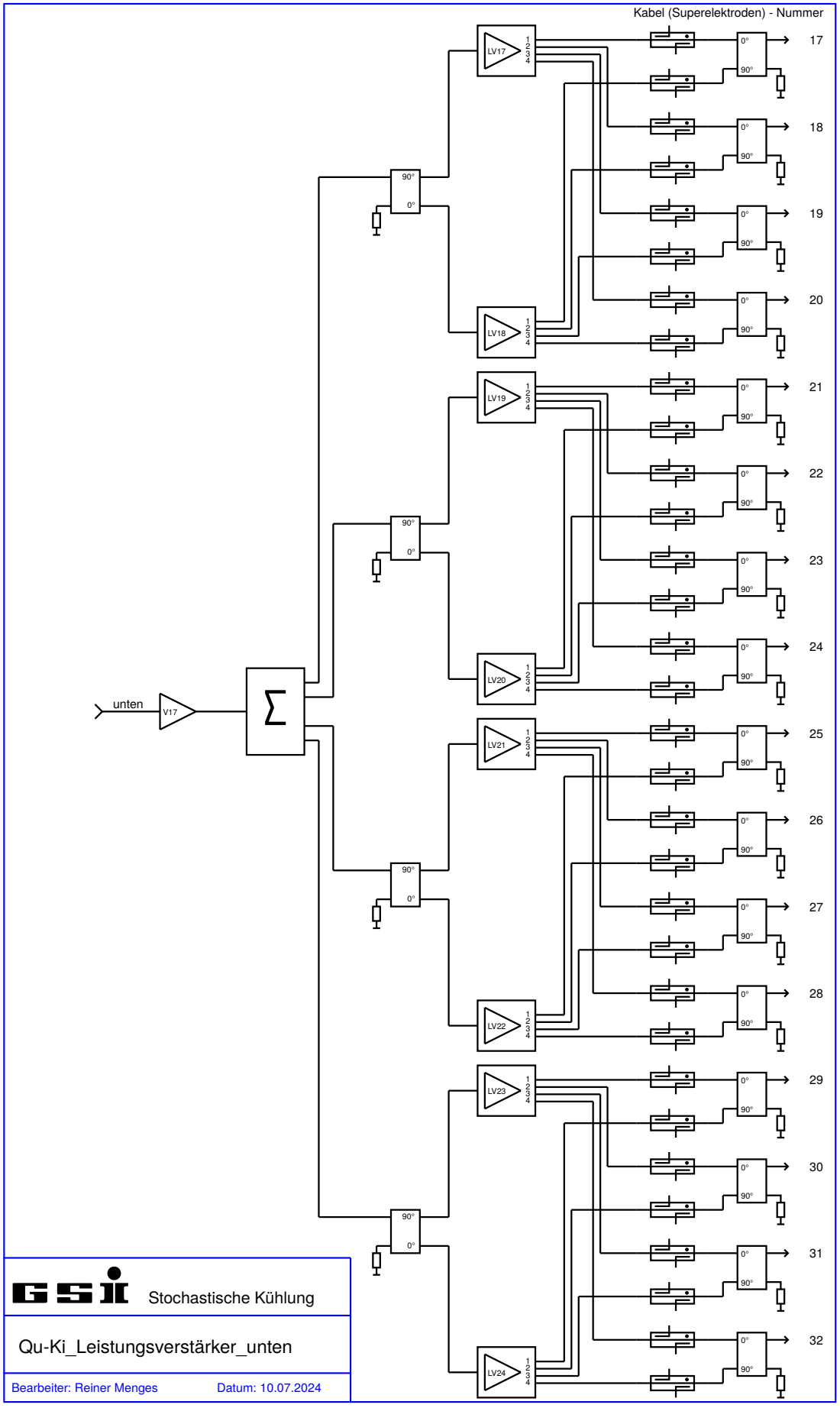


Abbildung 7.4: Blockschaltbild Quadrupol Kicker-Station, untere Elektroden