Diese Dissertation haben begutachtet:

Ao.Univ.-Prof. Dipl.-Ing. Dr.techn. Günter Fafilek

Ao.Univ.-Prof. Dipl.-Ing. Dr.techn. Thomas Grechenig

TECHNISCHE UNIVERSITÄT WIEN
Vienna University of Technology

DISSERTATION

Analyse, Modellierung und Management komplexer Messvorgänge in der Chemie

ausgeführt zum Zwecke der Erlangung des akademischen Grades eines Doktors der technischen Wissenschaften unter der Leitung von

Ao.Univ.-Prof. Dipl.-Ing. Dr.techn. Günter Fafilek
Institut für Chemische Technologien und Analytik (E164)

eingereicht an der

Technischen Universität Wien
Fakultät für Technische Chemie

von

Dipl.-Ing. Markus Demetz
Matr.-Nr. 9904146
markus@demetz.eu

Wien, am

Kurzfassung

Messdaten sind vielfach die Grundlage zur Erforschung empirischer Zusammenhänge und zur Gewinnung naturwissenschaftlicher Erkenntnisse. Die Abbildung von physikalischen Größen auf numerische Werte übernimmt ein Messgerät oder eine gesamte Messeinrichtung. Philosophisch betrachtet ist die Messung ein Hilfsmittel für den Prozess der Wahrnehmung. Dabei wird zwischen *sensorischer Wahrnehmung* und *rationaler Wahrnehmung* unterschieden. Bei einer Messung erhält das Subjekt durch *sensorische Wahrnehmung* ein Bild über die reale Welt bzw. über die **Eigenschaften** einer realen **Umgebung**, während die *rationale Wahrnehmung* die Analyse, Synthese, Auswertung usw. dieses Bildes betrifft. Die Eigenschaften einer Messumgebung werden durch ihre **Zustände** ausgedrückt und bilden hiermit den Vorrat für den Wahrnehmungsprozess. Die Abbildung des Zustands einer Eigenschaft auf einen numerischen Wert ist die *Messung des Zustands* bzw. der *Wert der Eigenschaft*.

Mit zunehmender Komplexität der messtechnischen Fragestellung ist es schwierig, dem Subjekt ein unverzerrtes und gut korreliertes Bild der realen (Mess-) Umgebung zu präsentieren. Dabei spielt die korrekte Abbildung der Relationen zwischen den Eigenschaften eine wichtige Rolle. Das Zusammenspiel unterschiedlicher Messgeräte ist aus Kompatibilitätsgründen oft nicht möglich oder gestaltet das Messsystem träge, was zu erheblichen Verzerrungen im Gesamtergebnis führt. Die *rationale Wahrnehmung* wird folglich von einer Unsicherheit beeinflusst und bildet vor allem bei *indirekten Messungen* eine kritische Angelegenheit. Am Beispiel eines *Scanning Electrochemical Microscopes (SECM)* ist dies deutlich zu erkennen. Der Strom einer Mikroelektrode ist sowohl ein Maß für die Oberflächenaktivität eines Substrates als auch ein Maß für den Abstand zum Substrat selbst. Die dadurch erforderliche korrelierte Messung bzw. Regelung des Abstands im Submikrometerbereich stellt eine schwierige messtechnische Aufgabe dar.

Vorliegende Arbeit präsentiert theoretische und softwaretechnische Aspekte, um komplexe Datenerfassungs- und Steuerprozesse durchzuführen, ohne dabei wichtige Information zu verlieren, damit ein möglichst unverzerrtes Bild der Messumgebung aufgenommen werden kann.

Abstract

Measurement data is the basis for the investigation of empirical relations and for the winning of natural scientific knowledge. The mapping from physical dimensions to numerical values is done by a measurement device or a whole measurement system. In terms of philosophy, the measurement is a medium for the cognition process. There exist two forms of congnition: *sensory cognition* and *rational cognition*. By *sensory cognition* the subject gains an image of the surrounding **reality**, while *rational cognition* concerns the analysis, synthesis etc. of the image acquired. The reality consists of a series of **qualities** which are expressed by its **states**. The mapping of a state to a numerical value is the *measurement of the state* or the *value of the quality* respectively.

The presentation of an unbiased image, showing the proper correlations between qualities, is a big challenge when complexity of the metrological question increases. Because of incompatibilities, the interaction between different measurement devices is often not possible or leads to big timing troubles, which is the cause of distortions in the image produced. *Rational cognition*, in this case, becomes influenced by an uncertainty, which is critical; especially in *indirect measurements*. This can be seen at the example of a *Scanning Electrochemical Microscope (SECM)*, where the current is a measure of the surface reactivity of a substrate as well as a measure of the distance to the substrate itself. Hence, the distance measurement in the submicrometer range is a difficult metrological task.

This work introduces theoretical and softwaretechnological aspects, to perform data capture and control processes, without losing important information and therefore to produce an undistorted image of reality.

Eidesstattliche Erklärung

Ich erkläre an Eides statt, daß ich die vorliegende Arbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen nicht benützt und die den benutzten Quellen wörtlich oder inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.

Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungsbehörde vorgelegt und auch noch nicht veröffentlicht.

Wien, im Juni 2015

Dipl.-Ing. Markus Demetz

Danksagung

Mein Dank gilt allererstens an *Ao.Univ.-Prof. Dipl.-Ing. Dr.techn. Günter Fafilek*, der mir die Möglichkeit gegeben hat, diese Arbeit unter seiner Leitung durchzuführen und stets bereit war, mir mit Ratschlägen und Ideen Beihilfe zu leisten.

Weiteres bedanke ich mich bei meinen Eltern, *Gitti* und *Gustav*, sowie bei meiner Großmutter *Lisi-Oma*, die mich über die gesamte Studiendauer tatkräftig unterstützt haben.

Ich bedanke mich auch bei allen Mitarbeitern für Ideen, konstruktive Gespräche und die kollegiale Zusammenarbeit.

Inhaltsverzeichnis

1 Einleitung

1.1 Übersicht

Messdaten sind vielfach die Grundlage zur Erforschung empirischer Zusammenhänge und zur Gewinnung naturwissenschaftlicher Erkenntnisse. Damit physikalische Vorgänge in der Forschung analysiert werden können, ist es notwendig, diese durch einen aussagekräftigen Wert darzustellen. Das Ergebnis ist ein Produkt zwischen Zahlenwert und Einheit und bildet die Grundlage zur Verarbeitung der gewonnenen Information. Die Abbildung von physikalischen Größen auf numerische Werte übernimmt ein Messgerät oder eine gesamte *Messeinrichtung*, die wie folgt definiert ist [1]:

> *„Gesamtheit aller Messgeräte und zusätzlicher Einrichtungen zur Erzielung eines Messergebnisses.“*

Die Bedienung von modernen Messgeräten erfordert, auf Grund der Vielfalt an Funktionen, fundierte Kenntnisse über deren Einstellungsmöglichkeiten. Die meisten Messinstrumente stellen eine Schnittstelle bereit, welche die Möglichkeit bietet, Einstellungen am Gerät – indirekt – über einen Computer vorzunehmen. Dazu wird vom Fabrikant gewöhnlicherweise die benötigte Software mitgeliefert oder zum Download bereitgestellt. Die Datenübertragung zum Computer findet über ein Bussystem statt, welches das Gerät unterstützt. Beispiele für solche Bussysteme sind der *General Purpose Interface Bus (GPIB)*, welcher vor allem in den 80er Jahren die Standardanbindung von Messinstrumenten an den Computer war, oder *Industry Standard Architecture (ISA)*. Moderne Messgeräte verfügen meistens über einen *Universal-Serial-Bus (USB)*-Anschluss. Auch *Peripheral Component Interconnect (PCI)* sowie *RS-232* bzw. *RS-485* sind Beispiele von Verbindungen, die zum Einsatz kommen. Schnittstellen, die über das *TCP/IP*-Protokoll arbeiten, werden von einigen Geräten ebenfalls unterstützt.

1.2 Problemstellung

Durch die zunehmende Komplexität der messtechnischen Fragestellung wachsen auch die Anforderungen an das System, sodass nur selten ein Gerät im Alleingang die gesamte Aufgabe übernimmt. Meistens ist eine Einrichtung gefordert, in welcher Apparaturen unterschiedlichster Art zum Erzielen des gewünschten Ergebnisses beitragen. Bussysteme haben sich im Laufe der Jahre weiterentwickelt und verändert, sodass andersartige Geräte im System oft über verschiedene Schnittstellen angesprochen werden müssen, wenn man diese softwaremäßig steuern will. Das automatisierte Zusammenspiel von verschiedenen Messgeräten ist daher aus Kompatibilitätsgründen oft nicht möglich und muss durch einen manuellen Eingriff unterstützt werden. Eine Intervention des Experimentators ermöglicht zwar die Durchführung einer komplexen Messaufgabe, jedoch sind auf Grund der dadurch entstehenden Trägheit des Gesamtablaufs erhebliche Verzerrungen im Ergebnis zu erwarten. Messergebnisse müssen manuell von einem Gerät auf das andere übertragen werden, falls die gewünschten Einstellungen eines Instrumentes vom Ergebnis eines anderen abhängen. Es ist daher wünschenswert, dass der Eingriff in den Messablauf automatisiert erfolgt, um solche Verzerrungen weitestgehend zu reduzieren. Ein automatisierter Ablauf von mehreren Geräten erfordert jedoch ein koordiniertes Miteinander, damit unerwünschte Effekte, die im Extremfall sogar zu mechanischen Beschädigungen im System führen, vermieden werden können.

Ein bekanntes Softwaresystem, welches den Ansatz verfolgt, verschiedene Geräte zentralisiert zu steuern, ist LabVIEW von NATIONAL INSTRUMENTS. Dieses Programmpaket erlaubt die grafische Darstellung von Mess- und Steuerinstrumenten durch das modulare Einbinden sogenannter *Virtual Instruments (VIs)*. Die Module für diese virtuellen Instrumente müssen vom Gerätehersteller mitgeliefert bzw. selbst entwickelt werden und stellen eine Abstraktionsschicht zwischen Messgerät und Software dar. Mit Hilfe von Kontrollstrukturen und diversen Anweisungen können Messabläufe erstellt bzw. koordiniert werden, in denen unterschiedliche Geräte zum Einsatz kommen. Dem Benutzer wird eine Vielfalt an Möglichkeiten und Funktionen geboten, jedoch ist dieser mit einer hohen Lernkurve konfrontiert. Die grafische Programmierung von Messabläufen kann, je nach Anwendungsfall, schnell zu unüberschaubaren Konstrukten führen und ist für den Experimentator eine große Herausforderung, wenn dieser keinen sicheren Umgang mit dem System besitzt.

1.3 Zielsetzung und Vorgehensweise

Die Messung ist ein Hilfsmittel für den Prozess der Wahrnehmung. Damit ein solcher Wahrnehmungsprozess stattfinden kann, müssen zunächst physikalische Vorgänge durch aussagekräftige Daten dargestellt werden. Ein Messgerät übernimmt dabei die Aufgabe einer *sensorischen Wahrnehmung* von Eigenschaften einer realen Umgebung. Die dadurch erhaltene Information stellt die Ausgangsbasis für eine *rationale Wahrnehmung* dar.

Ausgehend von Grundlagen der Theorie der Messung wird versucht, die Zusammenhänge von physikalischen Eigenschaften einer Messumgebung zu formalisieren. Es werden dabei sowohl theoretische als auch technische Aspekte ausgearbeitet, die zur Erzeugung einer konsistenten Abbildung der Messumgebung notwendig sind. Klassische Messaufgaben in der Elektrochemie reichen von einfachen Abläufen, wie z. B. das periodische Einlesen eines gewissen Potentials, über komplexere Aufgaben, wie das zusätzliche Schalten von Messkanälen, bis hin zu sehr komplexen Messverfahren, wie z. B. *Scanning Electrochemical Microscopy (SECM)*, bei welchen eine ausgereifte Struktur mit koordiniertem Ablauf unentbehrlich ist.

Mit Hilfe von softwaretechnischen Aspekten wird die Entwicklung eines Systems angestrebt, mit welchem die theoretisch ausgearbeiteten Anforderungen umgesetzt werden können. Inkompatibilitäten sollen durch geeignete Abstraktionsmechanismen umgangen werden und ein Messablauf soll weitestgehend automatisiert stattfinden. Durch das gewählte Maß an Abstraktion soll ein guter Kompromiss zwischen Anwendbarkeit, Flexibilität und Erweiterbarkeit des Systems gefunden werden.

1.4 State-of-the-art

Messinstrumente sind in ihrer Bedienung oft sehr kompliziert. Die Möglichkeit, das Instrument softwaremäßig zu steuern, ist daher nicht nur erwünscht, sondern heutzutage schon fast eine Voraussetzung, um am Markt konkurrenzfähig zu sein [2]. Beim Kauf eines Mess- bzw. Steuerinstrumentes ist daher fast immer eine Software samt der dazugehörigen Treiber im Lieferumfang enthalten. Ebenfalls dabei sind Programmierschnittstellen (APIs[1]) für verschiedene Programmiersprachen, mit denen das Gerät direkt angesprochen werden kann und dem Programmierer ermöglichen, eigene Funktionen und Programme für das Gerät zu entwickeln. Da

[1] Application Programming Interface (Schnittstelle zur Anwendungsprogrammierung).

die meisten Programme sehr gerätespezifisch entwickelt werden und nur ein Gerät bzw. höchstens eine Gerätefamilie bedienen können, ist die Programmierung von eigenen Funktionen ein erster Ansatz, einen flexiblen Messablauf zu erstellen. Mit Hilfe der APIs ist es also möglich, ein Programm zu entwickeln, in welchem mehrere Geräte involviert sind. Programmierschnittstellen werden meistens für die Sprache C bzw. C++ angeboten. Auch wenn C/C++ mittlerweile etwas in die Jahre gekommen ist, hat diese Sprache ihre natürliche Nische in der Entwicklung anspruchsvoller Systemanwendungen gefunden, die sie noch viele Jahre behalten wird [3]. Neuere Sprachen wie z. B. Java, C# oder Ruby/Rails sind im Vergleich zu C++ weniger leistungsstark, wenn es um die Programmierung von hardwarenahen Prozessen geht.

Die Entwicklung maßgeschneideter Software ist eine sehr aufwendige und repetitive Arbeit, sodass der Wunsch nach Abstraktion und Wiederverwendbarkeit von bereits programmierten Softwaremodulen groß ist. LabVIEW [4] ist eine mächtige Software, ist aber auf Grund der Komplexität einiger messtechnischen Fragestellungen nicht immer eine gute Wahl. Aus diesem Grund wurden bereits einige Messversuche mit einer Eigenentwicklung [5] durchgeführt. Weniger bekannt sind andere Systeme wie z. B. die Taskbeschreibungssprache TDL [6], welche eine C++-Erweiterung ist, oder CINT [7], ein C++-Interpreter, welcher im ROOT [8] Paket enthalten ist. Eine weitere jüngst erschienene Software ist NOVA [9] von METHROHM AUTOLAB. Mit diesem proprietären Softwaresystem versucht der Hersteller von Potentiostaten und Galvanostaten den modernen Anforderungen von Messsystemen entgegenzukommen.

2 Bussysteme

> Wenn du eine weise Antwort verlangst, musst du vernünftig fragen.
>
> *(Johann Wolfgang von Goethe)*

2.1 Übersicht

Bussysteme dienen zum Austausch von Information. Diese kann in Form von physikalischen oder logischen Größen dargestellt werden. Der Austausch bzw. die Übertragung selbst findet entweder analog oder digital statt. In diesem Kapitel werden einige Bussysteme bzw. Hardwareschnittstellen beschrieben, welche die Messeinrichtung mit einem Computer verbinden. Historisch gesehen ist der *General Purpose Interface Bus* von großer Bedeutung, daher wird dieser ausführlich behandelt. Weitere Systeme wie *ISA*, *PCI*, *USB* oder *FireWire* werden übersichtsmäßig beschrieben. Die Verwendung von Feldbussen findet vorwiegend in Anfertigungen statt, die sich selten ändern – dies würde unseren Anforderungen jedoch widersprechen. Jede kleinste Änderung im System wäre dadurch mit erheblichen Kosten verbunden.

2.2 General Purpose Interface Bus (GPIB)

2.2.1 Geschichte

Unter Messgeräten sehr stark verbreitet ist der *General Purpose Interface Bus (GPIB)*. Dieser wurde Ende der 60er Jahre von HEWLETT-PACKARD (HP), einem Hersteller (zu dieser Zeit) von Test- und Messgeräten, entwickelt und unter dem Namen *HP Interface Bus (HP-IB)* veröffentlicht. Wenig später trägt dieser Bus auch die Namen *General Purpose Interface Bus*, *GPIB*, *IEEE-488* oder *General Purpose Instrumentation Bus*.

Ziel war es, Mess- und Kontrollgeräte einfacher und über eine einheitliche Schnittstelle miteinander zu verbinden. Verschiedene Hersteller kopierten den *HP-IB* und nannten ihn *General Purpose Interface Bus* bzw. *GPIB*, sodass dieser ziemlich schnell zu einer Art Standard wurde. Dank seiner, für damalige Verhältnisse, hohen Übertragungsrate von 1 Megabyte pro Sekunde, konnte der *GPIB* schnell an Popularität gewinnen und wurde vom INSTITUTE OF ELECTRICAL AND ELECTRONICS ENGINEERS als *IEEE Standard Digital Interface for Programmable Instrumentation* standardisiert. Der Bus ist heute weltweit hauptsächlich unter drei verschiedenen Namen bekannt:

- *Hewlett-Packard Interface Bus (HP-IB)*
- *General Purpose Interface Bus (GPIB)*
- *IEEE-488 Bus*

Der ursprüngliche Standard des Busses enthielt keine Richtlinien bezüglich einer bevorzugten Syntax, sodass weiterhin an der Spezifikation gearbeitet werden musste, um die Systemkompatibilität zu verbessern. Dies führte zu einem erweiterten Standard namens *IEEE-488.2*. Der alte Bus wurde durch diesen jedoch nicht ersetzt, sondern lediglich umbenannt und trägt ab nun den Namen *IEEE-488.1*.

Der *IEEE-488.2*-Standard wurde in folgenden Punkten erweitert:

- Format- und Syntaxkonvention
- Geräteunabhängige Befehle
- Fehlercodes

Jedes Gerät, welches dem *IEEE-488.2*-Standard entsprechen wollte, musste nun über eine Klasse von Befehlen verfügen, welche unter dem Namen *Standard Commands for Programmable Instrumentation (SCPI)* benannt sind. Nur in diesem Fall ist die Kompatibilität zum *IEEE-488.2*-Bus gewährleistet.

2.2.2 Design

Der *General Purpose Interface Bus* ist ein externer, paralleler 8-Bit-Datenbus, an welchem bis zu 15 Geräte gleichzeitig angeschlossen werden können. Adressierungslimits können direkt mit Hilfe von Bus-Expandern oder indirekt mit einem Isolator oder Extender umgangen werden. Ältere Geräte, welche diesen Bustyp

unterstützen, verfügen über mechanische Schalter, um ihre Adresse einzustellen. Bei moderneren Geräten wird diese digital über das *Frontpanel* eingestellt. Manche Geräte verfügen über 2 *GPIB*-Ports, wobei einer für ASCII-Befehle und Daten verwendet wird, der andere für *High Speed Binary Output (DUMP)*. Die Adresse für ASCII I/O wird auch *major address* bezeichnet und ist immer eine gerade Adresse. Die unmittelbar darauffolgende Adresse wird *minor address* bezeichnet und ist natürlich ungerade. Theoretisch können über den *GPIB* 30 Geräte adressiert werden; in der Praxis ist jedoch nur die Adressierung von 15 Geräten möglich. Sobald ein Computer an den *GPIB* angeschlossen wird, übernimmt dieser die Rolle des *Controllers* und sorgt für einen koordinierten Datenaustausch zwischen den angeschlossenen Geräten.
Abbildung 2.1 zeigt einen weiblichen Steckverbinder eines *IEEE-488*-Busses; dieser verfügt insgesamt über 24 Pins und erlaubt durch *weiblich/männlich*-Verbindungen eine anreihbare Kombination von mehreren Geräten.

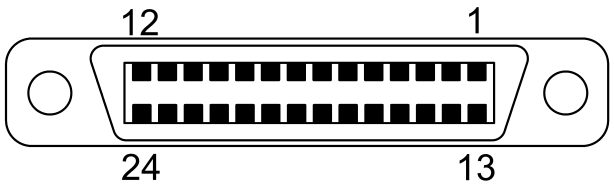


Abbildung 2.1 – Weiblicher Steckverbinder für den *General Purpose Interface Bus*. *Quelle: Wikipedia*

Die Kabellänge sollte in Summe 20 Meter nicht überschreiten und die Anzahl der Steckverbinder sollte sich auf 4 beschränken, da die Funktionalität sonst nicht mehr gewährleistet ist. Zu jedem Zeitpunkt darf nur ein Gerät Daten senden, nämlich der *active talker*. Alle anderen am Bus angeschlossenen Geräte, welche die gesendeten Daten empfangen, werden als *active listeners* bezeichnet. Um die Busgeschwindigkeit nicht unnötigerweise zu beschränken, werden jene Geräte, welche die vom *active talker* gesendeten Daten nicht benötigen, vom *Controller* angewiesen, nicht zu «hören» [10][1]. Das langsamste Gerät bestimmt die Gesamtgeschwindigkeit im System.

2.2.3 Interface Signale

Das *IEEE-488*-Bussystem verfügt über insgesamt 24 Leitungen, davon 16 Signalleitungen sowie 8 *Ground*-Leitungen.

[1] Übersetzt aus: In order to optimize bus speed, the controller instructs all other devices to unlisten.

Tabelle 2.1 – Übersicht der Pin-Belegung vom *General Purpose Interface Bus*.

Pin	Funktion	Gruppe
1–4	DIO 1–4 (Data Input-Output)	Data
5	EOI (End Or Identify)	Controller
6	DAV (DAta Valid)	Transfer
7	NRFD (Not Ready For Data)	Transfer
8	NDAC (No Data ACcepted)	Transfer
9	IFC (InterFace Clear)	Controller
10	SRQ (Service ReQuest)	Controller
11	ATN (ATteNtion)	Controller
12	SHIELD	Common
13–16	DIO 5–8	Data
17	REN (Remote ENable)	Controller
18–24	GND (GrouND)	Common

Die Signalleitungen werden in Datenleitungen (8), Handshake-Leitungen (3) und Bus-Management-Leitungen (5) eingeteilt.

Datenleitungen: Die Datenleitungen (Pins DIO1 bis DIO8) übertragen Adresse, Kontrollinformation und Daten. Der Bus arbeitet mit negativer Logik und TTL[2].

Handshake-Leitungen: Die Handshake-Leitungen (DAV, NRFD und NDAC) sorgen für eine reibungslose und kollisionsfreie Datenübertragung auf den Datenleitungen. Die Handshake-Leitungen arbeiten unabhängig von der Übertragungsgeschwindigkeit, da der Bus die Geschwindigkeit des langsamsten Gerätes übernimmt.

DAV (DAta Valid): Der Sender bestimmt über den Zustand der DAV Leitung. Wird diese Leitung auf *low* gesetzt, so wird allen «hörenden» Geräten mitgeteilt, dass gültige Daten auf den Leitungen liegen.

NRFD (Not Ready For Data): Mit diesem Signal teilen die «hörenden» Geräte mit, dass sie noch nicht zur Datenaufnahme bereit sind.

NDAC (Not Data ACcepted): Mit diesem Signal teilen die «hörenden» Geräte mit, dass die Daten auf den Datenleitungen DIO1–DIO8 noch nicht bearbeitet bzw. aufgenommen wurden.

[2] Transistor-Transistor-Logic.

Bus-Management-Leitungen: 5 Leitungen kontrollieren die Bus-Aktivitäten.

ATN (ATteNtion): Dieses Signal teilt mit, dass ein *Command Byte* am Bus vorhanden ist (z. B. eine Adresse).

IFC (InterFace Clear): Der *System-Controller* kann den Bus zurücksetzen und aktiver Controller werden.

REN (REmote Enable): Alle Teilnehmer im Bus gehen in den *Remote-Modus* wenn dieses Signal gesetzt wird.

EOI (End Or Identify): Das EOI Signal wird gesetzt, sobald das Ende einer Nachricht erreicht wird bzw. das letzte Byte gesendet wurde.

SRQ (Service ReQuest): Über diese Leitung können Geräte dem *Controller* ein Service anfragen. Der *Controller* sucht mit Hilfe von *Polling* das zu bedienende Gerät und verrichtet die vorgesehene Operation (ähnlich wie bei einem *Hardware Interrupt* [11]).

Die 8 verbleibenden Leitungen sind *Ground*-Leitungen.

Der Ablauf einer Handshake-Sequenz wird in Abbildung 2.2 dargestellt und funktioniert wie folgt:

1. Der Sender oder der *Controller* will Daten auf den Bus anlegen und setzt das DAV-Signal auf *high*. Damit werden die Teilnehmer im Bussystem informiert, dass die Daten nicht mehr gültig sind. Bevor irgendwelche Daten geschrieben werden, muss jedoch gewährleistet sein, dass alle Geräte bereit sind, neue Daten zu empfangen.
2. Jeder Empfänger, der bereit ist Daten aufzunehmen, setzt das NRFD-Signal auf *high*. Erst wenn alle Empfänger das Signal auf *high* gesetzt haben geht die NRFD-Leitung auf *high*.
3. Nun wird das NRFD-Signal wieder auf *low* gesetzt, da die Empfänger mit der Datenaufnahme beschäftigt sind. Wenn ein Empfänger fertig ist, dann gibt er das NDAC-Signal frei. Erst wenn alle Geräte das NDAC-Signal freigegeben haben, geht die Leitung wieder auf *high*.

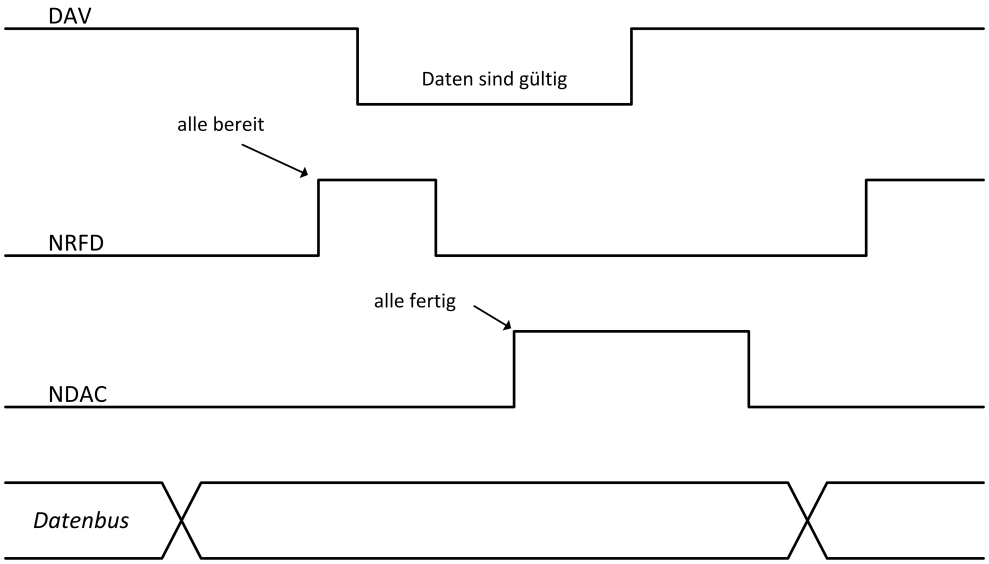


Abbildung 2.2 – Handshake-Sequenz für die Übertragung von einem Byte über den *General Purpose Interface Bus*.

2.2.4 Das High-Speed GPIB Handshake Protokoll

Das *High-Speed GPIB Handshake Protokoll (HS488)* wurde 2003 von NATIONAL INSTRUMENTS entwickelt, um eine höhere Datenübertragungsrate als die bisherige von 1 Megabyte pro Sekunde zu erhalten. Dieser patentierte Mechanismus erfordert, dass alle am System teilhabenden Geräte dem *HS488 Protokoll* konform sind. Durch den verbesserten Handshake-Mechanismus sind Datenübertragungsraten von bis zu 8 Megabyte pro Sekunde möglich; die tatsächliche Geschwindigkeit wird jedoch vom langsamsten Gerät im Bussystem bestimmt.

Sollte ein Gerät das Protokoll nicht unterstützen, so benützt der Bus automatisch den *IEEE-488.1*-Standard, um Rückwärtskompatibilität zu gewährleisten [12].

2.2.5 Ein Ersatz für GPIB

Der *General Purpose Interface Bus* ist seit mehr als 30 Jahren die *de-facto* Standardschnittstelle für die Verbindung von Messinstrumenten mit dem Computer. Viele Anwender sind jedoch der Meinung, dass der mittlerweile in die Jahre gekommene *IEEE-488.1*-Standard ersetzt werden sollte. Vor allem der Preis[3] sowie der Wunsch nach modernen I/O Schnittstellen, welche standardmässig im einem Computersystem eingebaut sind, lassen nach einem Ersatz rufen [13].

[3] Der Preis für ein *GPIB*-System (*Controller* und Kabel) beläuft sich um die 500$.

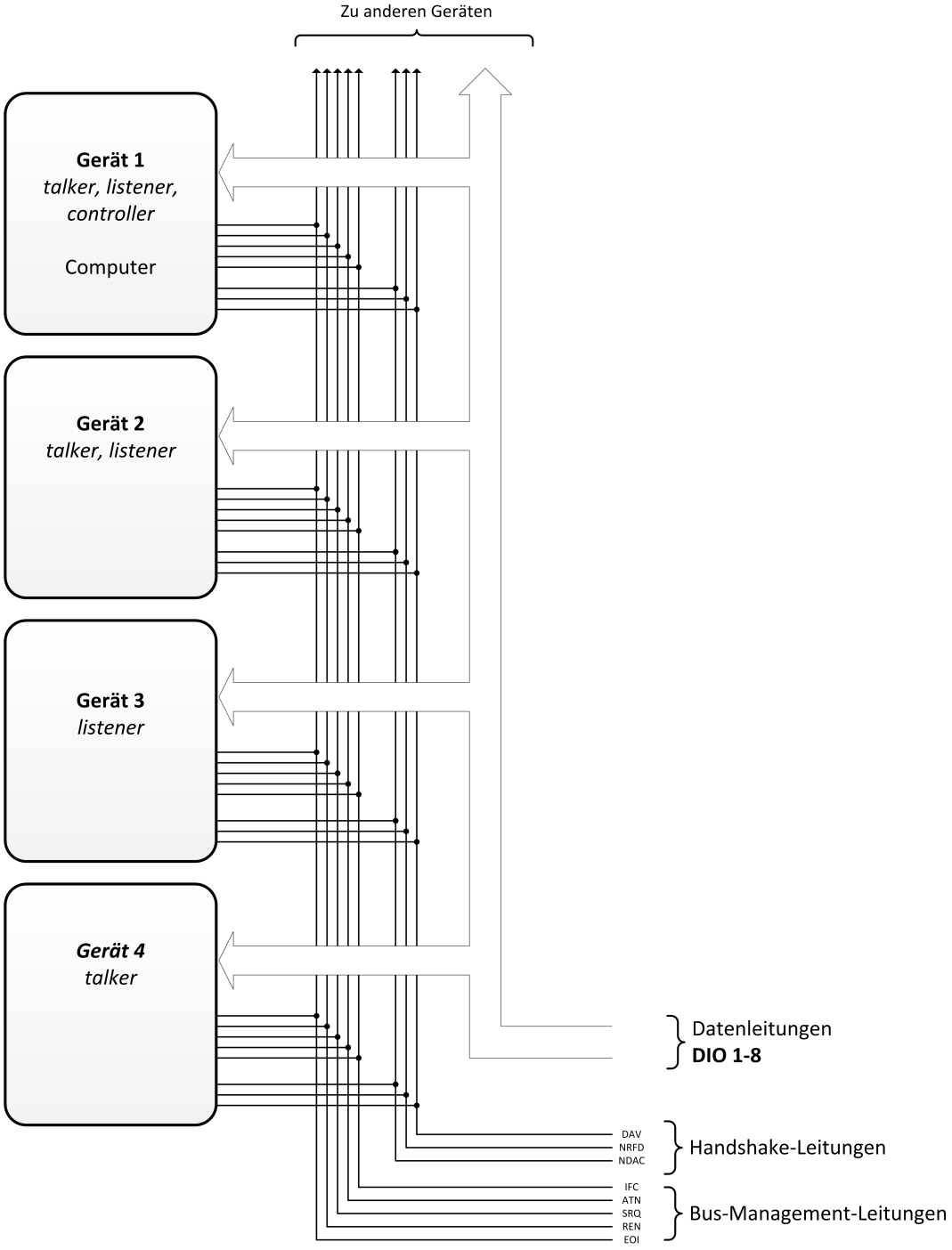


Abbildung 2.3 – Beispielkonfiguration von Geräten mit dem *General Purpose Interface Bus*.

Folgende Standards haben sich in den letzten Jahren an Computersystemen etabliert:

- *Industry Standard Architecture (ISA)*
- *Peripheral Component Interconnect (PCI)*
- *Universal Serial Bus (USB)*
- *IEEE 1394 (FireWire)*
- *Netzwerk I/O (TCP/IP)*

2.2.6 Vorteile und Nachteile von GPIB

Vorteile:

- *IEEE-488* ist eine ausgereifte und stark verbreitete Schnittstelle.
- Es existiert eine große Anzahl an Software, die in den ganzen Jahren für Geräte mit *GPIB*-Schnittstellen geschrieben wurde.
- *GPIB* verfügt über Leitungen für spezielle Funktionen (z. B. SQR – *Service Request*; einem *Hardware Interrupt* gleichwertig [11]).

Nachteile:

- Die Kabellänge ist auf 2 Meter begrenzt.
- Die maximale Anzahl an angeschlossenen Geräten (inklusive *Controller*) beträgt 15.
- Die Busgeschwindigkeit ist max. 1.5 Megabyte pro Sekunde (ausgenommen beim *HS488 Protokoll*).
- Kabel sind sehr teuer.
- Die Kosten belaufen sich auf ca. 500$ pro System (*Controller* und Kabel).

2.3 Industry Standard Architecture (ISA)

Der *Industry Standard Architecture* Bus, welcher in der Praxis fast immer nur *ISA* genannt wird, ist ein Anfang der 80er Jahre durch ein IBM-Projekt entstandenes Bussystem. Dank seiner offenen Architektur durch IBM konnte sich dieser Standard besser durchsetzen als vergleichsweise bessere Bussysteme, welche zur gleichen Zeit entstanden sind. Der *ISA*-Bus entwickelte sich zusammen mit dem PC zum Standard und konnte sich fast zwei Jahrzehnte behaupten. Dank der robusteren Ausführung bzw. Entwicklung von *PC/104*, ein Derivat von *ISA*, konnte der Bus auch ohne Mutterplatine verwendet und somit gut im *Embedded*-Bereich eingesetzt werden.

Für neuere PC-Systeme war der 16-Bit und 8 MHz-schnelle Bus jedoch zu langsam, sodass die Massenversorger INTEL und MICROSOFT aus u. a. folgenden Gründen zur Entfernung des *ISA* Busses aufgerufen haben [14]:

- Der Bus musste durch manuelles Setzen von Jumpern konfiguriert werden und Hardwarekonflikte wurden nicht verlässlich vom System gelöst.
- Trotz des Versuchs, automatische Konfiguration zu ermöglichen, hat der *Plug-and-Play* Mechanismus in vielen Fällen versagt.
- Für einen modern ausgerüsteten PC würden die zur Konfiguration notwendigen *Interrupts* nicht reichen.

Auf Grund des langjährigen Standards ist der *ISA*-Bus heutzutage noch immer im Einsatz. Vor allem im Industrie- und Militärbereich, wo die Anschaffung von Geräten spezieller Ausführung mit hohen Kosten verbunden war, wird dieser Bustyp noch benötigt. Es werden daher noch immer extra angefertigte Mutterplatinen am Markt angeboten, welche über diese Schnittstelle verfügen, die in herkömmlichen PCs bereits Ende der 90er Jahre verschwunden ist.

Auch im *Embedded*-Bereich ist dieser Bus noch vorhanden, da die Geschwindigkeit für viele Anforderungen ausreichend ist. Die Konfiguration wird nur einmal zu Beginn von Experten durchgeführt und nachher vom Konsumenten nicht mehr verändert. Die niedrigere Leistungsaufnahme eines mit *ISA* ausgestatteten Systems ist ein weiterer Grund, weshalb dieser Bus für gewisse Aufgaben noch zum Einsatz kommt.

2.4 Peripheral Component Interconnect (PCI)

Als INTEL Anfang der 90er Jahre einen Nachfolger für den *ISA*-Bus auf den Markt brachte, etablierte sich *Peripheral Component Interconnect* ziemlich bald zu einem Standard für Hochgeschwindigkeitsbusse in PC Systemen. Das *PCI*-Bussystem existiert in einer 32-Bit und in einer 64-Bit-Ausführung mit jeweils einer Taktgeschwindigkeit von 33 bzw. 66 MHz. Es können 10 Geräte am Bus teilnehmen und die maximale Datenübertragungsrate beträgt je nach Ausführung 132 MByte/s (33 MHz x 4 Bytes) bzw. 528 MByte/s (66 MHz x 8 Bytes) [15].

Ein Überschreiten des Limits von 10 Geräten kann mit Hilfe von *PCI-PCI-Bridges* erreicht werden, in welchen ein sogenannter *Arbiter* den Zugriff auf das Bussystem dirigiert um Konflikte und Kollisionen zu vermeiden. Durch eine eigene Taktleitung ist es zudem möglich, mit niedrigeren Übertragungsraten zu arbeiten, um Strom einzusparen. Auch wenn eine *Peer-to-Peer* Kommunikation möglich ist, arbeitet das System meistens im *Master-Slave*-Modus.

Dank der fehlenden Notwendigkeit einer Rückwärtskompatibilität[4] war INTEL in der Lage, viele Probleme des *ISA*-Busses zu vermeiden:

- Durch ein festes Adressierungsschema namens *PCI Configuration Space* kann das System die vorhandenen *PCI*-Geräte analysieren und die notwendigen Resourcen bestimmen. Angeschlossene *PCI*-Geräte werden während des Bootvorgangs gescannt und die benötigten Resourcen werden automatisch zugewiesen. Dies ermöglicht eine automatische Konfiguration und die Notwendigkeit von manueller Einstellung durch das Setzen von Jumpern wird beseitigt.

- Durch gemeinsame Leitungen werden weniger der wertvollen Interrupts benötigt.

Durch die starke Verbreitung und dem preiswerten Zugang am PC-Markt konnte sich das *Peripheral-Component-Interconnect*-System lange Zeit behaupten. Seit 2005 wird *PCI* jedoch schrittweise durch das schnellere *PCI-Express*-System ersetzt.

[4] *PCI* wurde im Vergleich zu *ISA* von Grund auf neu entwickelt und war nicht an eine Rückwärtskompatibilität gebunden.

2.5 Universal Serial Bus (USB)

Die Entwicklung von *Universal Serial Bus (USB)* begann Mitte der 90er Jahre durch führende Hersteller wie INTEL, MICROSOFT usw. und hatte zum Ziel, den Konflikt zwischen einer ständig wachsenden Anzahl von periphären Geräten sowie der limitierten Anzahl von *ISA*- und *PCI*-Slots in Mainboards, zu lösen [16]. *USB* gilt als Nachfolger der *RS-232*-Schnittstelle, welche seit der Einführung des moderneren Bussystems nach und nach verdrängt und auf Motherboards immer seltener wurde. Spezielle *USB-to-Serial-Adapter* ermöglichen es, die *RS-232*-Schnittstelle weiterhin zu verwenden. Vor allem in technischen Einrichtungen im industriellen Umfeld ist diese Schnittstelle teilweise noch immer vorhanden. Der *Universal Serial Bus* benötigt einen *Host-Controller (Master)*, welcher die im System angeschlossenen Geräte koordiniert. Die theoretische Anzahl von anschließbaren Geräten *(Slave-Clients)* ist 127[5].

Im Vergleich zu traditionellen Schnittstellen verfügt *USB* über folgende Besonderheiten:

- Hohe Übertragungsrate
- *Plug-and-play* sowie *Hot-plugging*[6]
- Ressourcenschonend
- Bessere Zuverlässigkeit durch eine starke Fehlerkorrektur der übertragenen Daten

Das relativ simple architekturelle Design erlaubt eine preisgünstige Herstellung dieses Bussystems, ist aber auch für die relativ hohen Interruptzeiten von bis zu 1 ms verantwortlich. Die Ursprungsversion *USB 1.0* erlaubte eine maximale Datenübertragungsrate von 12 MBit/s, welche sich mit der Verabschiedung der Nachfolgespezifikationen – *USB 2.0* sowie *USB 3.0* – stetig verbesserte.

Weiteres verfügt der *Universal Serial Bus* über sogenannte Geräteklassen, welche die Verwendung eines periphären Gerätes ermöglichen, ohne einen bestimmten Treiber installieren zu müssen. Erst für die Verwendung von speziellen Funktionen des angeschlossenen Gerätes muss der dazugehörige Treiber verwendet werden. Einige Geräte, wie z. B. *USB-Sticks*, wurden sogar erst durch dieses Bussystem geboren.

[5] Die Adressierung der Geräte erfolgt durch 7 Bit.
[6] *USB* erlaubt das Hinzufügen von Geräten ohne den Computer neu starten zu müssen.

2.6 FireWire (IEEE 1394)

Die Entstehung des *IEEE-1394*-Busses, welcher auch als *FireWire* oder *i.Link* bekannt ist, geht ursprünglich auf APPLE zurück; ein Standard wurde aber erst ein Jahrzehnt später (1995) verabschiedet. Lange Zeit konnte der *General Purpose Interface Bus* die Anforderungen in Test- und Messsystem erfüllen, sodass sich die Mess- und Computerindustrie divergierend entwickelten [17]. Die Entwicklung des *FireWire*-Busses wurde vor allem durch die Bedürfnisse der Multimedia-Industrie vorangetrieben und findet, auf Grund seiner hohen Datenübertragungsrate, vorwiegend in der Unterhaltungselektronik Verwendung.

Mit der Zeit wurde jedoch das neue Bussystem auch für die Test- und Messindustrie interessant, sodass in manchen modernen Geräten auch eine *IEEE-1394*-konforme Schnittstelle zu finden ist. Dieser serielle Datenbus wurde zwar für eine einfachere Verwendung und leichtere Verbindung konzipiert, jedoch sollte im Gegensatz zu *USB* eine hohe Performance stets im Vordergrund stehen. Aus diesem Grund konnte *FireWire* preislich mit dem *Universal Serial Bus* nicht mithalten, wodurch die Verbreitung gehemmt wurde. Im Gegensatz zu *USB* basiert der *IEEE-1394*-Bus auf einer *Peer-to-Peer*-Architektur und benötigt daher keinen *Host-Controller*; es ist eine direkte Verbindung zwischen einzelnen Geräten möglich. Die Übertragungsrate dieses Bussystems lag in seiner Erstversion schon bei ca. 12 MB/s und wurde in regelmäßigen Abständen verbessert.

2.7 Gegenüberstellung

General Purpose Interface Bus

Der *General Purpose Interface Bus* ist ein weit verbreitetes und sehr robustes Bussystem, welches auch heute noch von einer großen Auswahl an Geräten unterstützt wird. Die lange «Lebensdauer» von (Mess-) Geräten sowie die Entstehung von modernen Bussystemen führte zur Entwicklung von verschiedenen Konvertern wie z. B. *USB-zu-GPIB*, sodass der *IEEE-488*-Bus nach wie vor in vielen Systemen Verwendung findet. Der große Nachteil: sein Preis.

Industry Standard Architecture

Industry Standard Architecture wurde mittlerweile komplett aus den PCs eliminiert und findet sich höchstens noch in speziell angefertigten *Industrial Mainboards* oder

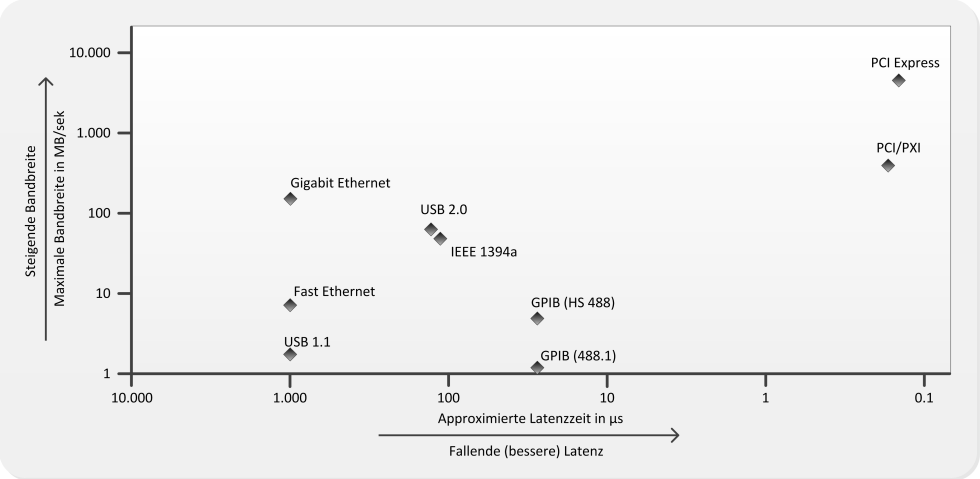


Abbildung 2.4 – Latenzzeiten und Bandbreite verschiedener Bussysteme.
Quelle der Daten: [18]

in älteren Systemen, in welchen Geschwindigkeit kein ausschlaggebendes Kriterium ist, wieder. Der *ISA*-Bus wurde durch den *PCI*-Bus ersetzt, kann aber mit Hilfe einer *PCI-ISA-Bridge* weiterhin verwendet werden.

Peripheral Component Interconnect

Peripheral Component Interconnect ist der *de-facto* Nachfolger von *ISA* und verfügt sowohl über geringe Latenzwerte als auch über hohe Datenübertragungsraten. PCs verfügen jedoch meistens über wenige *PCI*-Schnittstellen, sodass die anzuschließenden Geräte in ihrer Anzahl limitiert sind. Systeme, in denen Latenzzeiten einen kritischen Faktor darstellen, werden ohnehin keine (oder nur wenige) parallelläufigen Tasks ausführen; die Knappheit von *PCI-Slots* ist daher in kritischen Messsystemen eher unbegründet.

Universal Serial Bus

Der *Universal Serial Bus* ist weit verbreitet und jeder moderne PC verfügt über einen *USB-Host-Controller*. Es sind zwar hohe Datenübertragungsraten möglich, jedoch ist dieser Bus wegen seiner im Vergleich hohen Latenzwerte – Interruptzeiten von bis zu 1 ms [17] – nicht für jedes System die beste Wahl. Für die hohen Latenzwerte ist das relativ simple architekturelle Design verantwortlich, welches eine kostengünstige Herstellung von *USB* ermöglicht. Immer mehr Geräte verwenden den *Universal Serial Bus* zur Datenübertragung an den Computer. Die Anschaffung des teueren *GPIB*-Busses bleibt damit erspart und auch die leicht höheren Latenzzeiten sind in den meisten Fällen vernachlässigbar. Mit der Einführung von *USB 2.0* und *USB 3.0* konnte die Datenübertragung wesentlich verbessert werden.

FireWire

Technisch gesehen ist *FireWire* ein starkes Bussystem und dem *Universal Serial Bus* in vielen Bereichen überlegen. Oft wurde in der Vergangenheit gezeigt, dass technische Überlegenheit keine Garantie für eine starke Verbreitung bzw. Durchsetzung am Markt ist. Preisdruck, ursprüngliche Lizenzgebühren sowie ungünstiges *Timing* lässt das Duell zwischen den beiden Bussystemen derzeit zugunsten des kostengünstigeren Kontrahenten *USB* entscheiden.

3 Problemstellung und Analyse

> Messen, was messbar ist –
> messbar machen, was nicht
> messbar ist!
>
> *(Galileo Galilei)*

3.1 Übersicht

Das vorherige Kapitel zeigt und beschreibt die Entstehung sowie die Evolution einiger der wichtigsten Schnittstellen bzw. Bussysteme der letzten 30–40 Jahre. Lange Zeit galt der *General Purpose Interface Bus* als der *de-facto* Standard für die Verbindung von Mess- und Steuergeräten mit dem Computer. Nach und nach erschienen neue Systeme und neue Standards, um eine bessere Performance, bessere Kompatibilität usw. zu erreichen.

Der Wunsch, Systeme zu standardisieren bzw. zu vereinheitlichen, konnte sich nur bedingt erfüllen. Nach der Einführung neuer Standards ist es nur eine Frage der Zeit, bis neue Komponenten am Markt verfügbar sind, die nicht oder nur zum Teil auf diesen Standards basieren. Durch die Weiterentwicklung von Geräten kann sich ein Standard im Nachhinein als Schwachstelle erweisen und die optimale Performance ist nicht mehr erreichbar. Forschung ist jedoch sehr stark von der Leistung und den Fähigkeiten neuer Systeme abhängig, sodass es notwendig ist, neue Systeme ehestmöglich zum Einsatz zu bringen. Um eine optimale und zielbringende Performance in einem System zu erlangen, sollte der Gesamtablauf gut abgestimmt werden. Dieses Kapitel beschäftigt sich mit den theoretischen und technischen Aspekten, um ein an die Messaufgabe bestens angepasstes System zu entwickeln.

3.2 Theoretische Analyse

Die Grundaufgabe eines Messsystems ist es, physikalische Grössen durch einen aussagekräftigen Wert darzustellen. Philosophisch betrachtet ist die Messung ein Hilfsmittel für den Prozess der Wahrnehmung [19, S.1–2]. Dabei wird zwischen *sensorischer Wahrnehmung* und *rationaler Wahrnehmung* unterschieden. Bei einer Messung erhält das Subjekt durch *sensorische Wahrnehmung* ein Bild über die reale Welt bzw. über die **Eigenschaften** einer **realen Umgebung**, während die *rationale Wahrnehmung* die Analyse, Synthese, Auswertung usw. dieses Bildes betrifft. Die Eigenschaften einer Messumgebung werden durch ihre **Zustände** ausgedrückt und bilden hiermit den Vorrat für den Wahrnehmungsprozess. Die Abbildung des Zustands einer Eigenschaft ist die *Messung des Zustands* bzw. der *Wert der Eigenschaft*. Damit das Bild über die reale Welt nicht durcheinandergebracht wird, unterliegt der Abbildungsprozess einigen Bedingungen, welche Gegenstand der Theorie der Messung sind.

3.2.1 Relationale Systeme

Ein relationales System beschreibt Beziehungen zwischen Objekten. Mengentheoretisch ausgedrückt ist eine k-dimensionale Relation R auf einer Menge von Objekten A eine Menge von Tupel, welche eine Teilmenge des kartesischen Produktes $A^k = A \times A \times ... \times A$ bildet. Sowohl Funktionen als auch Operationen auf Objekten einer Menge A können als Relation aufgefasst werden. Eine Funktion $f : A \rightarrow A$ ist eine Menge von geordneten Paaren $< a, f(a) >$ und daher eine Teilmenge von A^2. Eine (binäre) Operation ist ebenfalls eine Menge von geordneten Paaren $< a_1, a_2 >$ mit $a_1, a_2 \in A$. Ist $\circ$ eine binäre Operation auf einer Menge A und gilt $a_1 \circ a_2 = a_3$ mit $a_1, a_2, a_3 \in A$, so ist das Tupel $< a_1, a_2, a_3 >$ ein Element der ternären Relation $R_\circ$. Generell wird eine k-äre Operation zu einer $k+1$-ären Relation.

Definition 3.2.1 (Relationales System) *Ein relationales System ist definiert als ein geordnetes Tupel* $\boldsymbol{R} =< A; R_1, R_2, ..., R_n >$ *wobei:*

- *A eine nichtleere Menge von Objekten ist,*
- $R_1, ..., R_n$ *k_i-äre Relationen auf Objekten der Menge A mit* $1 \leq i \leq n$ *und* $k_i \in \mathbb{N}$ *sind.*

Betrachtet man eine Menge von Objekten die miteinander in Beziehung stehen, so geht man davon aus, dass diesen Objekten Eigenschaften zu Grunde liegen. Bei

den Beziehungen solcher Eigenschaften handelt es sich um ein **Empirisches Relativ**. Betrachtet man hingegen Beziehungen zwischen einer Menge von Zahlen, so handelt es sich um ein **Numerisches Relativ**. Man unterscheidet daher zwischen *Eigenschaften* (empirisch) und *Messergebnissen* (numerisch). Möchte man eine Korrespondenz zwischen diesen beiden Relativen formalisieren, so sollte eine gegebene Abbildung von Objekten auf numerische Werte «strukturerhaltend» sein. Bleiben die Beziehungen der Objekte zwischen den Relativen untereinander erhalten, so ist die Abbildungsfunktion homomorph und man spricht von einer Skala.

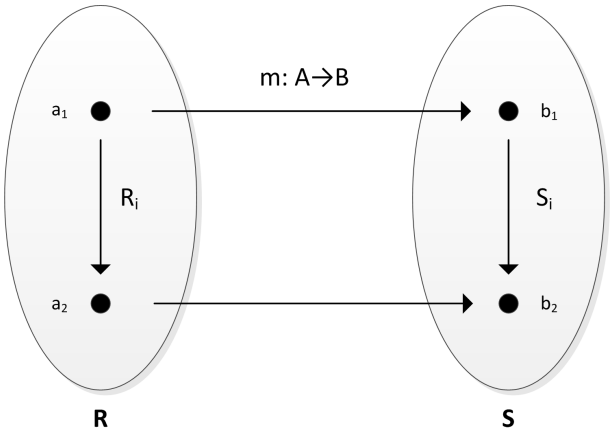


Abbildung 3.1 – Grundmodell einer Messung mit Beibehaltung der Relationen.

Definition 3.2.2 (Skala) *Sei* $\boldsymbol{R} =< A; R_1, R_2, ..., R_n >$ *ein empirisches relationales System und* $\boldsymbol{S} =< \mathrm{B}; S_1, S_2, ..., S_n >$ *ein numerisches relationales System. Weiteres sei* k_i *die Dimension von* R_i *und* S_i *für* $1 \leq i \leq n$*. Die Abbildung* $m : A \rightarrow \mathrm{B}$ *von Objekten aus der Menge A auf numerische Werte B ist das Maß der Objekte aus A. Das Tripel* $< \boldsymbol{R}, \boldsymbol{S}, m >$ *ist eine Skala wenn die Relationen erhalten bleiben und folgendes gilt:*

$$R_i(a_1, ..., a_{k_i}) \Leftrightarrow S_i(m(a_1), ..., m(a_{k_i})) \; \forall a_j \in A \text{ mit } 1 \leq j \leq k_i, \; 1 \leq i \leq n \quad (3.1)$$

Abbildung 3.1 stellt den Sachverhalt anhand grafischer Elemente [20] für die formale Modellierung von Messprozessen methodisch dar.

3.2.2 Schlüsselpunkte der Theorie der Messung

Die Schlüsselpunkte der Theorie der Messung sind [21]:

1) Die Messung ist eine Abbildung von empirischen Objekten auf numerische Werte.

2) Empirische Relationen zwischen den Objekten bleiben nach der Abbildung erhalten.

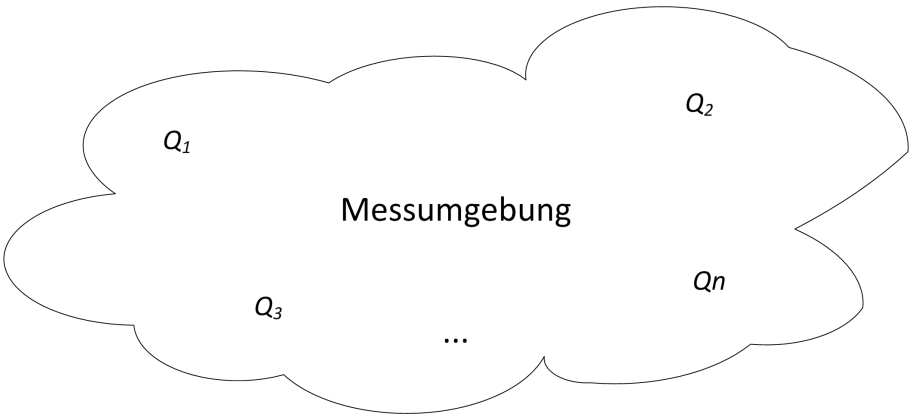


Abbildung 3.2 – Messumgebung mit verschiedenen Eigenschaften Q_i.

ad 1) Es sei M eine Messumgebung mit den Eigenschaften $Q_1 \ldots Q_n$ sowie A_i die Menge der Zustände für die Eigenschaft Q_i. Weiteres sei B_i eine Menge von numerischen Werten für das Ausmaß der Eigenschaft Q_i. Die Abbildung $A_i \rightarrow B_i$ ist also eine Funktion $b = f(a)$, die jeden Zustand $a \in A_i$ auf einen Wert $b \in B_i$ abbildet und stellt somit das Ausmaß der Eigenschaft Q_i bzw. die Intensität ihres Zustandes numerisch dar.

ad 2) Wir definieren ein relationales System auf einer Menge von Zuständen A_i einer Eigenschaft Q_i durch $\boldsymbol{R_i} = \langle A_i; R_{i_1}, \ldots, R_{i_n} \rangle$ wobei:

- A_i eine nichtleere Menge von Zuständen einer Eigenschaft Q_i ist,
- $R_{i_1}, \ldots, R_{i_n}$ k_{i_j}-äre Relationen auf Zustände aus A_i für $1 \leq j \leq n$ sind.

Eine binäre Operation $\circ$ auf A_i wird dann als ternäre Relation aufgefasst. Ist $a_1 \circ a_2 = a_3$, dann ist das Tupel (a_1, a_2, a_3) ein Element aus $R_{i,\circ}$ mit $a_1, a_2, a_3 \in A_i$.

Die Operanden des empirschen relationalen Systems sind die unterschiedlichen Zustände einer Eigenschaft Q_i in der Messumgebung. Beispielsweise kann die (binäre) Operation *ist wärmer* auf zwei verschiedene Zustände einer Temperatur angewendet werden. Ein numerisches relationales System hat (meistens) reelle Zahlenwerte als Operanden. Die empirische Operation *ist wärmer* wird durch die numerische Relation $>$ ausgedrückt. Die Temperatur ist ein klassisches Beispiel in der Physik für eine nicht-additive Größe, erfüllt aber die Bedingungen einer Ordnungsrelation. Die Abbildung ist daher homomorph und die Struktur der Relationen bleibt nach der Abbildung erhalten. Es sei $b = f(a)$ die Abbildung eines Zustandes $a \in A$ der Tempera-

tur auf einen Wert $b \in B$ sowie R_* die empirische Relation *ist wärmer* und $S_>$ die numerische Relation $>$, so gilt folgender Zusammenhang:

$$R_*(a_1, a_2) \Leftrightarrow S_>(f(a_1), f(a_2)) \;\; \forall a_i, a_j \in A \tag{3.2}$$

Generell muss folgendes gelten:

$$b = f(a) \tag{3.3}$$

$$a^* = \varphi(b) = f^{-1}(b) \tag{3.4}$$

$$a^* \equiv a \quad \forall a \in A \tag{3.5}$$

3.2.3 Arten der Messung

Ursprünglich wurden von CAMPBELL[1] zwei Arten von Messungen unterschieden. Seine Beobachtung war, dass die Messung mancher Eigenschaften von der Messung zwei oder mehrerer anderen Eigenschaften abhängt. Dies führte zum Begriff der *abgeleiteten Messung*. Die Messung einer Eigenschaft, welche von keinen weiteren Eigenschaften abhängt, bezeichnete er als *fundamental*.

Nach ELLIS war diese Unterscheidung nicht vollständig: es gibt Eigenschaften, die nicht fundamental messbar sind und nur von einer einzigen anderen Eigenschaft abhängen [22, S.53–54]. Für die Temperaturmessung ist es z. B. nur notwendig, Druck, Volumen oder den elektrischen Widerstand (oder eine andere thermometrische Eigenschaft) zu messen. Diese Feststellung führte zur Erweiterung des Systems von CAMPBELL durch Umbenennung der zwei Arten von Messung in *Direkte Messung* und *Indirekte Messung*. Die *abgeleitete Messung* ist nun eine Spezialform einer *indirekten Messung*. Für Messungen, die dem Beispiel der Temperaturmessung entsprechen, wurde der Begriff *Assoziative Messung* eingeführt.

Zusammengefasst ergeben sich daraus folgende Unterscheidungen, welche im weiteren als Grundlage für die Formalisierung unserer Modelle dienen:

[1] NORMAN ROBERT CAMPBELL (1880–1949), englischer Physiker und Wissenschaftsphilosoph.

Direkte Messung: Grundlegendes Gebot einer *direkten Messung* ist die Messbarkeit einer Eigenschaft, ohne auf andere Eigenschaften rückgreifen zu müssen.

Indirekte Messung: Bei einer *indirekten Messung* hingegen wird die Eigenschaft nicht direkt gemessen; der gewünschte Messwert wird durch eine *direkte Messung* von anderen Eigenschaften, sowie eventuellen weiteren Berechnungen auf deren Ergebnisse, erhalten. Dabei werden zusätzlich zwei Unterscheidungen getroffen.

Abgeleitete Messung: Eine abgeleitete Messung erhält man durch mindestens zwei direkte Messwerte und durch die Kombination (unter gegebenen empirischen Gesetzen) dieser. Zum Beispiel ist die Dichte eine abgeleitete Messung aus Masse und Volumen. Für alle Objekte des gleichen Materials ist das Verhältnis von Masse zu Volumen eine Konstante ρ. Daraus folgt, dass nach Dichte geordnete Objekte auch nach dem Verhältnis ρ geordnet sind [23].

Assoziative Messung: Eine assoziative Messung betrifft die Messung mindestens einer Eigenschaft sowie einen bekannten Zusammenhang zwischen der gemessenen und der gewünschten Eigenschaft. Die Temperaturmessung über den Widerstand ist ein Beispiel für eine gewünschte bzw. assoziative Eigenschaft.

Der Begriff der abgeleiteten Messung wurde ursprünglich von CAMPBELL eingeführt [24, S.53]. Der Author selbst sowie weitere Theoretiker zweifeln über die Sinnhaftigkeit dieser Art von Messung und diskutieren, ob diese überhaupt eine Messung sei [25]. Die Klassifizierung einer abgeleiteten Messung sorgt daher immer wieder für Kontroversen. Oftmals ist es frei wählbar, ob eine Messung direkt oder indirekt (abgeleitet) ist. Die berechnete Dichte aus Masse und Volumen kann z. B. auch direkt gemessen werden. Während bei einer direkten Messung eine Abbildung von einem empirischen Relativ auf ein numerisches Relativ erfolgt, so wird bei einer abgeleiteten Messung versucht, ausgehend vom numerischen Relativ, ein sinnergebendes empirisches Relativ zu bestimmen, welches einen homomorphen Zusammenhang aufweist. Es würde z. B. bei einer Temperatur keinen Sinn ergeben zu sagen: $2° + 3° = 5°$.

In der Wissenschaft sollte die Feststellung einer fundamentalen Skala einer Eigenschaft, mit eigenständiger Bedeutung und empirischem Gesetz, einer abgelei-

teten Skala bevorzugt werden [24, S.32]. In einer komplexen Messaufgabe gilt es festzulegen, was und wie gemessen wird. Dies hängt von der konkreten Fragestellung sowie von den zur Verfügung stehenden Apparaturen ab. In der Arbeit von SUPPES und ZINNES [26], welche vor den Arbeiten von ELLIS [22] veröffentlicht wurde, wird neben der fundamentalen und abgeleiteten Messung, der Begriff *Pointer Measurement* erwähnt [26, S.20]. Mit dieser Art von Messung ist das direkte Ablesen eines Wertes von einem Messinstrument gemeint, welches zuvor so eingestellt wurde, dass die Werte der Abbildung einer fundamentalen oder abgeleiteten Messung entsprechen. In Anlehnung an [27] unterteilen wir die Eigenschaften einer Messumgebung in *Meta-Eigenschaften* und *Komplexe-Eigenschaften*. Dies ermöglicht eine pragmatische, wenn auch nicht immer formal korrekte, Trennung von direkter und indirekter Messung.

Definition 3.2.3 (Meta-Eigenschaft:) *Eine Meta-Eigenschaft ist eine grundlegende Eigenschaft, die nicht in kleinere Elemente geteilt und direkt gemessen werden kann.*

Definition 3.2.4 (Komplexe-Eigenschaft:) *Eine Komplexe-Eigenschaft ist eine zusammengesetzte Eigenschaft, die nicht direkt gemessen werden kann, aber in eine Menge von Meta-Eigenschaften zerlegbar ist.*

Wird eine *Meta-Eigenschaft* gemessen, so sprechen wir von einer *direkten Messung*; im Falle von einer *Komplexen-Eigenschaft*, von einer *indirekten Messung*.

Die Abbildungen 3.3 und 3.4 verdeutlichen das Prinzip der abgeleiteten sowie der assoziativen Messung anhand der grafischen Elemente für die formale Modellierung von Messprozessen [20]. Durch die Kombination von numerischen Werten, welche aus direkten Messungen erhalten werden, gelangt man zu einer abgeleiteten Messung. Durch eine Abbildung der assoziativen Eigenschaft auf numerische Werte, sowie den bekannten Zusammenhang zwischen assoziativer und gewünschter Eigenschaft, erhält man eine gleichwertige *direkte Messung*.

3.2.4 Korrelation von Eigenschaften

Nur selten bilden die Eigenschaften einer Messumgebung eine eigenständige und geschlossene Einheit. Wie am Beispiel der assoziativen Messung erkennbar ist, besteht zwischen manchen Eigenschaften eine Korrelation, die ausgenutzt werden kann, um auf andere Eigenschaften zurückzugreifen. Der Zusammenhang von Eigenschaften ist zwar bekannt, jedoch wird dieser von weiteren Einflüssen im System bestimmt. Es ist daher notwendig, komplexerere Phänomene einer Messumgebung mit mehreren Eigenschaften so abzubilden, dass eine *rationale Wahrnehmung*

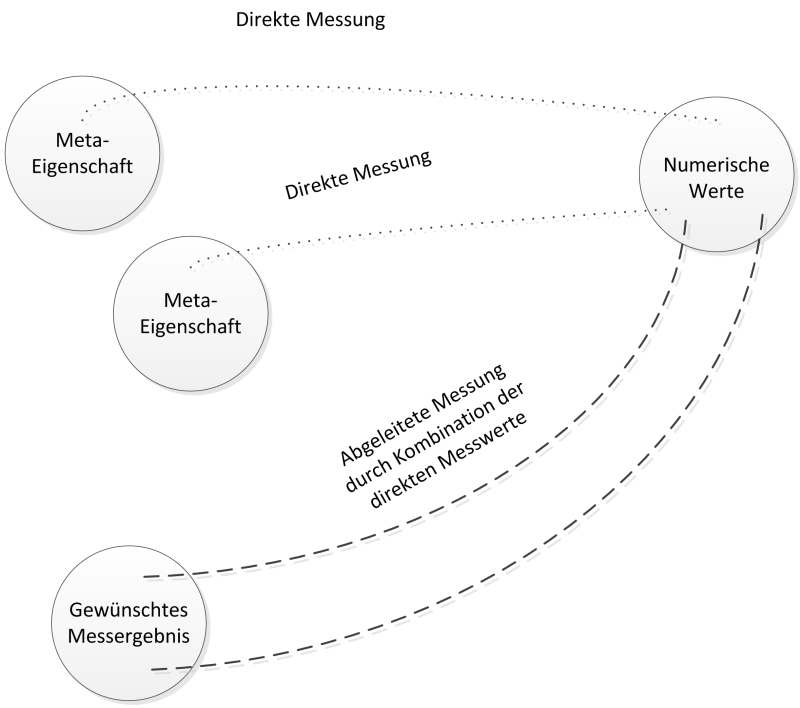


Abbildung 3.3 – Grafisches Modell einer indirekten, abgeleiteten Messung.

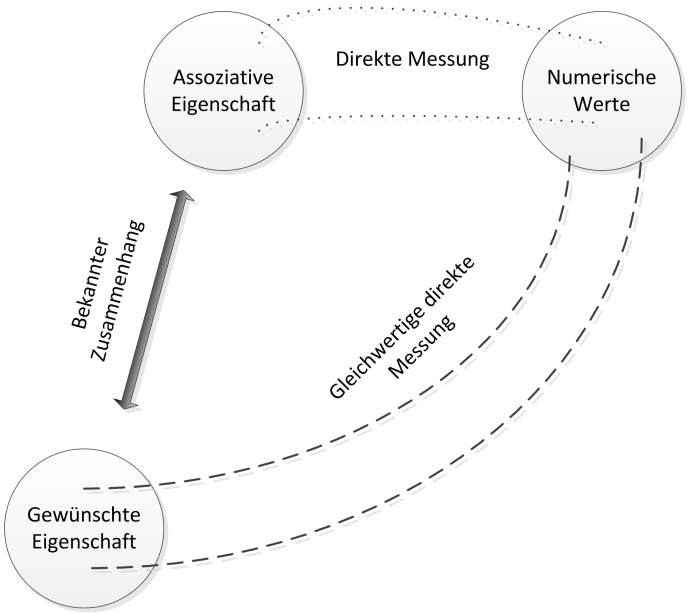


Abbildung 3.4 – Grafisches Modell einer indirekten, assoziativen Messung.

so wenig wie möglich verzerrt wird. Dies bedarf eines guten Überblicks über das Gesamtsystem sowie über den Zusammenhang zwischen dessen unterschiedlichen Eigenschaften. Die Abhängigkeiten sind natürlich von Fall zu Fall verschieden und es ist Aufgabe des Subjekts, das Maß der Korrelation auf Basis einer wissenschaftlichen Beurteilung sowie anhand der konkreten Fragestellung festzustellen und zu berücksichtigen.

Beispiel: Es seien Temperatur, Leitfähigkeit und Konzentration die Eigenschaften Q einer Messumgebung M. Die Leitfähigkeit σ ist eine Funktion über die Temperatur T und über die Konzentration c: $\sigma = f(T, c)$.

Formal gesehen ist die Leitfähigkeit eine *Komplexe-Eigenschaft*, da diese in die beiden *Meta-Eigenschaften* Temperatur Q_T und Konzentration Q_c zerlegbar ist. Die Zustände der *Meta-Eigenschaften* dienen als Grundlage zur Berechnung des indirekten (abgeleiteten) Messwertes σ. Die Messung entspricht einer **abgeleiteten Messung** wie in Abbildung 3.3 dargestellt ist: man erhält den gewünschten Wert durch die Kombination der beiden *Meta-Eigenschaften*.

Leitfähigkeit von Elektrolytlösungen

Die Leitfähigkeit von Elektrolytlösungen beruht auf der Bewegung von Ladungsträgern in Form von Ionen im elektrischen Feld. Der Stromfluss I zwischen 2 Elektroden ist bei vorgegebener Zusammensetzung und konstanter Temperatur proprotional zur Spannung welche über den Ionenleiter abfällt:

$$I = G \cdot U \text{ bzw. } U = R \cdot I \text{ mit } R = \frac{1}{G} \tag{3.6}$$

Der Proportionalitätsfaktor G stellt dabei den Leitwert dar. Aus dieser Gleichung folgt, dass der Widerstand einer Elektrolytlösung ein *Ohm*'scher Widerstand ist. Die gesuchte Leitfähigkeit σ ergibt sich letztendlich durch Berücksichtigung der Geometrie der Zelle bzw. dem Geometriefaktor K:

$$\sigma = \frac{1}{R} \cdot \frac{l}{A} = \frac{I}{U} \cdot K \tag{3.7}$$

Da der Gesamtwiderstand beim Anlegen einer Gleichspannung wegen der Elektrodenwiderstände eine nichtlineare Kennlinie aufweist, wird dieser mit Hilfe einer Wechselspannung $A \cdot \sin(\omega t)$ gemessen [28, S.13].

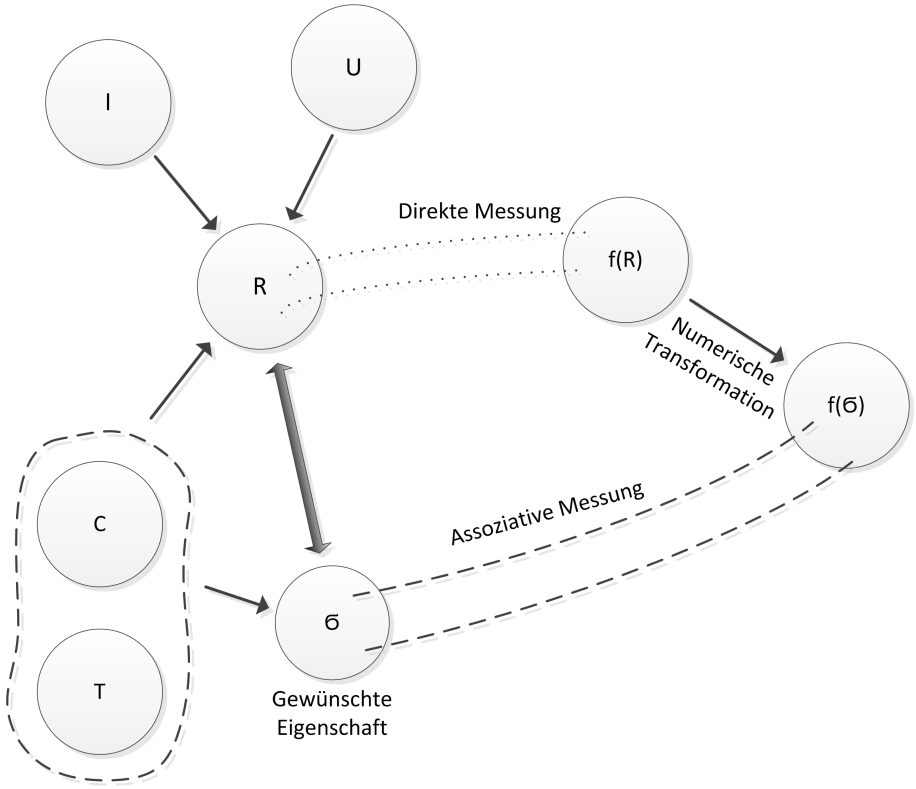


Abbildung 3.5 – Formale Darstellung der Leitfähigkeitsmessung einer Elektrolytlösung über den bekannten Zusammenhang mit dem Widerstand.

In der Praxis wird die Leitfähigkeit jedoch durch den bekannten Zusammenhang über den Widerstand R gemessen[2], was wiederum dem Modell einer **assoziativen Messung** entspricht (Abbildung 3.4). Die bevorzugte Wahl einer assoziativen Messung mancher Eigenschaften gegenüber einer direkten (fundamentalen) Messung begründet sich hauptsächlich in der Einfachheit der Durchführung [22, S.100]. Die Widerstandsmessung selbst setzt eine Spannung oder einen Strom voraus damit ein bekannter Zusammenhang hergestellt werden kann. Die Leitfähigkeit ergibt sich letztendlich durch die numerische Transformation $\sigma = \frac{1}{R} \cdot \frac{l}{A}$, wobei die Fläche A und der Abstand l während der Messung als konstant vorausgesetzt wird. Der Zusammenhang wird in Abbildung 3.5 grafisch dargestellt.

Der Einfluss von Temperatur T und Konzentration c ist allein auf Basis dieser Messung nicht bekannt und muss anhand weiterer Sachkenntnis wissenschaftlich ausgearbeitet werden. Mögliche chemische Abläufe führen dazu, dass sich die Konzentration einer Lösung ändert; z. B. eine Fällungsreaktion.

[2] Siehe Kasten.

3.2.5 Messunsicherheit

Die meisten Abbildungen von Zustände auf numerische Werte entsprechen nicht exakt dem realen Sachverhalt der gemessenen Eigenschaft. Der erhaltene Wert weicht im Regelfall bis auf eine endliche Differenz vom *wahren Wert* ab. Der Grund für diese Abweichung sind Messfehler, welche *systematischer* oder *zufälliger* Natur sein können und zu einer Unsicherheit im Messergebnis führen. Unsicherheit bedeutet Zweifel bzw. Bedenken über den gemessenen Wert und ist für die *rationale Wahrnehmung* ein ausschlaggebender Faktor, der berücksichtigt werden muss.

Zufällige Fehler können in der Regel nicht vermieden werden, da ihre Entstehung verschiedene Ursachen haben kann, die sich meistens nicht beeinflussen lassen. Mit Hilfe von Mittelwertbildung kann diese Art von Fehler sehr gut abgeschätzt werden. Systematische Fehler können hingegen mit Hilfe von wissenschaftlichem *Know-How* sowie einer guten Kenntnis über die Messumgebung mehr oder weniger gut ermittelt und bis zu einem gewissen Grad ausgebessert werden.

Nach der *Guide to the expression of Uncertainty in Measurements* kann die *Standardunsicherheit u* auf zwei verschiedene Arten festgelegt werden [29]:

i) statistisch, als Schätzwert der Standardabweichung des Mittelwertes einer Menge von unabhängigen Beobachtungen

ii) auf Basis einer wissenschaftlichen Beurteilung mit Hilfe der verfügbaren, relevanten Information; z. B. bisherige Messdaten, Kenntnis über das Verhalten der Eigenschaften von relevanten Materialien usw.

In komplexen Messaufgaben ist es schwierig, mehrere Messreihen mit gleicher Ausgangsbasis durchzuführen da sich – speziell in der Chemie – die Messumgebung schnell ändern kann. Eine statistische Schätzung der Unsicherheit ist daher weniger geeignet und auch nicht Ziel unserer Analyse. Eine genaue Darstellung der Realität ist mit «nackten» Zahlen oft nicht möglich. Die empirischen Zusammenhänge in der Messumgebung sind daher für die *rationale Wahrnehmung* von großer Bedeutung, um ein aufschlussreiches Bild zu erhalten.

Ungewissheit wird durch *indirekte Messung* eine kritische Angelegenheit [30]. Sei Y eine indirekte Messung, welche durch N Werte $B_1, B_2 \ldots, B_n$ bestimmt wird, so ist

$$Y = f(B_1, B_2, \ldots, B_n) \tag{3.8}$$

die Gleichung für den Erhalt eines indirekten Messwertes y aus den direkten Messwerten $b_1, b_2, \ldots, b_n$:

$$y = f(b_1, b_2, \ldots, b_n) \tag{3.9}$$

Wird der Messwert y, wie am Beispiel der Leitfähigkeit, direkt gemessen, so müssen die Werte $b_1 \cdots b_n$ empirisch geschätzt werden. Eine numerische Verknüpfung ist anhand einer Messung nicht möglich, jedoch kann die Kenntnis über die empirischen Zusammenhänge eine *rationale Wahrnehmung* positiv beeinflussen. Wird der Messwert y, wie formal vorgesehen, durch die Berechnung der direkten Messwerte $b_1 \cdots b_n$ erhalten, so ist es wichtig, dass die Messung der einzelnen Eigenschaften in zeitlicher Korrelation zueinander erfolgt.

3.3 Technische Analyse

Während es eine Aufgabe des Experimentators ist, die theoretischen Zusammenhänge eines Messsystems bestmöglich zu verstehen und zu berücksichtigen, so ist es auf der technischen Seite notwendig, die Voraussetzungen dafür zu schaffen, dass ein konsistentes und gut korreliertes Bild der Messumgebung dargestellt wird. Dabei sind einige technische Aspekte zu beachten, welche in diesem Abschnitt beschrieben werden.

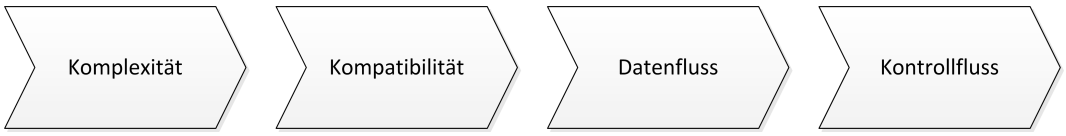


Abbildung 3.6 – Technische Aspekte zur Erzeugung eines konsistenten Datenbildes.

3.3.1 Komplexität

Speziell in der Elektrochemie erfordern Messaufgaben meistens das Zusammenlegen von mehreren Operationen um ein gewünschtes Ziel zu erreichen. Eine logische Sequenz von Operationen, welche auf die Messumgebung einwirkt und zu Zustandsveränderungen führt, wird als Messmethode bezeichnet. Bedingt durch die messtechnische Fragestellung sowie durch den Aufbau des Systems und dessen technischer Ausrüstung, ist es nicht immer möglich, den Operationsablauf im

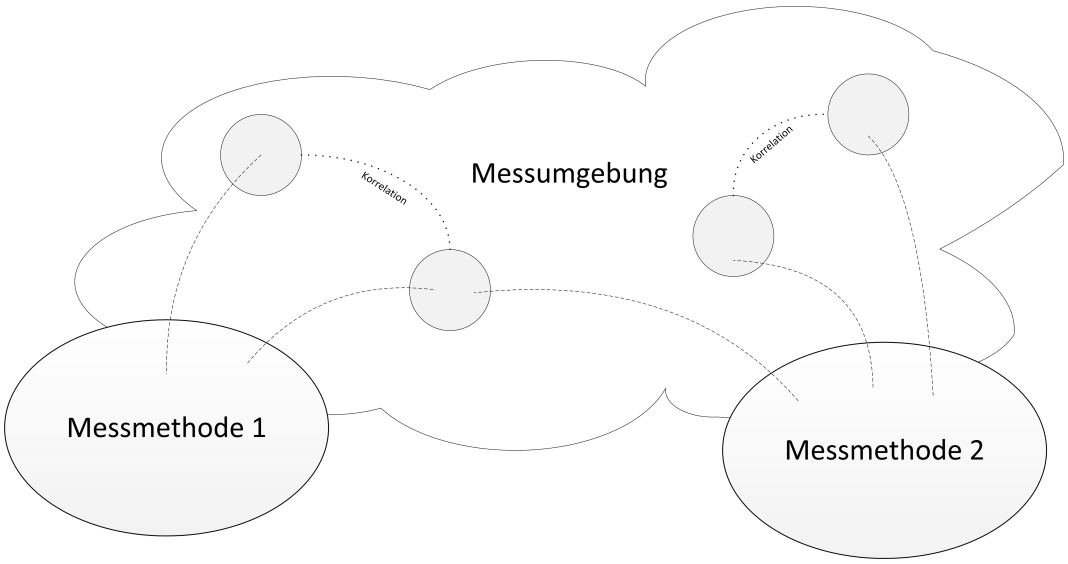


Abbildung 3.7 – Komplexe Messumgebung durch paralleles Einwirken verschiedener Messmethoden.

System sequentiell zu gestalten. In vielen Situationen ist es notwendig, mehrere Operationen parallel durchzuführen, da diese von unterschiedlichen Apparaturen gesteuert werden. Beispielsweise möchte man den Oxidationsprozess durch (kontrolliertes) Einblasen von Sauerstoff in einer Elektrolytlösung fördern, während man eine Korrosionsmessung durchführt. Es entsteht eine Parallelität von Abläufen, dessen Operationen **gleichzeitig** auf die Eigenschaften einer Messumgebung einwirken. Abbildung 3.7 stellt den Sachverhalt grafisch dar. Durch den parallelen Ablauf von Operationen entsteht ein komplett neues System, dessen Gesamtverhalten sich nicht sehr einfach beschreiben lässt.

Nicht nur das gezielte bzw. gewollte Einwirken auf die Messungebung, sondern auch die Messung einer Eigenschaft selbst, kann zu Zuständsveränderungen im Gesamtsystem führen. Als Beispiel betrachte man die Temperaturmessung eines Körpers durch einen anderen Körper, bei welcher es zu einem Wärmefluss zwischen diesen beiden kommen kann und dadurch die Temperatur beider Körper beeinflusst wird. Zudem können Eigenschaften, wie im vorigen Abschnitt erwähnt, auch untereinander in Beziehung stehen. Leitfähigkeit und Temperatur sind ein Beispiel dafür, dessen Korrelation zusätzlich durch die chemische Substanz bestimmt wird. Ebenfalls kann die Löslichkeit eines Stoffes und damit die Konzentration von der Temperatur abhängen. Das Ausmaß bzw. die Art und Weise, in welcher sich gewisse Eigenschaften gegenseitig beeinflussen, ist sehr verschieden und muss vom Experimentator bestmöglich abgeschätzt werden.

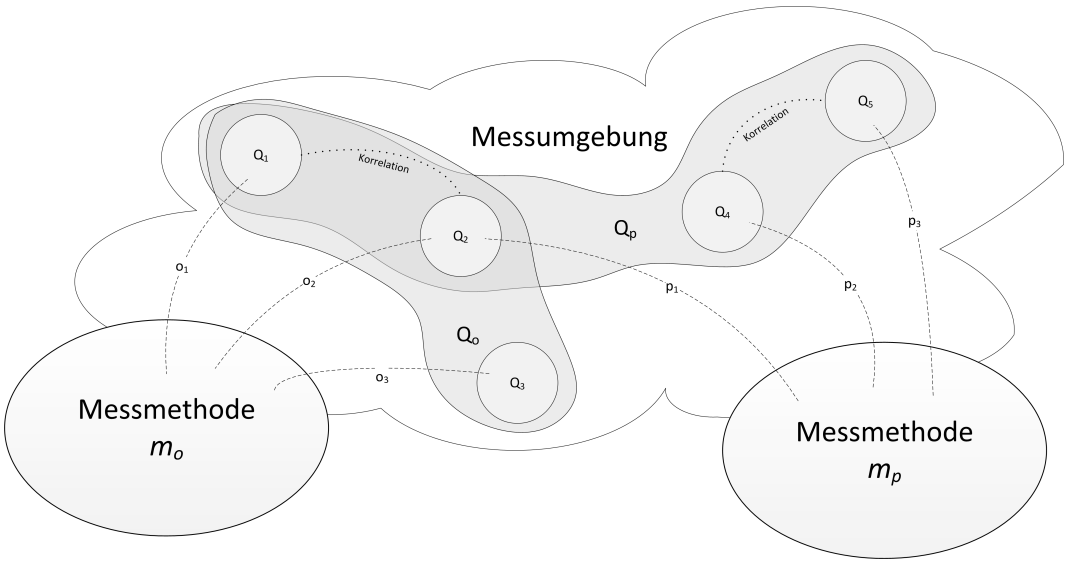


Abbildung 3.8 – Formale Darstellung der Einflüsse von Operationen auf korrelierte Eigenschaften der Messumgebung.

Es seien $m_o = \{o_1; o_2; \ldots; o_n\}$ und $m_p = \{p_1; p_2; \ldots; p_m\}$ zwei verschiedene Messmethoden auf einer gemeinsamen Messumgebung M mit den sequentiellen Operationen o_i und p_j für $i, j \in \mathbb{N}$. Weiteres seien $M_o = m_o(M)$ und $M_p = m_p(M)$ die empirischen Zustände nach der Anwendung der Messmethoden m_o bzw. m_p auf die gemeinsame Messumgebung M. Die Mengen der Eigenschaften, welche durch die Messmethoden m_o bzw. m_p direkt oder indirekt beeinflusst werden, bezeichnen wir mit Q_o bzw. Q_p. Es gilt folgender Sachverhalt:

$$Q_o \cap Q_p = \emptyset \Longrightarrow \rho(m_o, m_p) = 0 \tag{3.10}$$

Ist die Schnittmenge der von m_o bzw. m_p beeinflussten Eigenschaften leer, so ist die Korrelation der beiden Messmethoden gleich 0. Das Zusammenlegen der beiden Messmethoden hätte keine Bedeutung. Ist die Korrelation der beiden Messmethoden $\rho(m_o, m_p) > 0$, was in der Regel der Fall bzw. gewünscht ist, so wird das Bild der Messumgebung durch die Komplexität des Systems verzerrt. Eine Auswertung des Bildes ist alleine auf Basis der abgebildeten Eigenschaften nicht vernünftig durchführbar. In Abbildung 3.8 wird der Sachverhalt dargestellt: Messmethode m_o beeinflusst die Eigenschaften $Q_o = \{Q_1, Q_2, Q_3\}$; die Messmethode m_p hat direkten Einfluss auf die Eigenschaften Q_2, Q_4 und Q_5. Da die beiden Eigenschaften Q_1 und Q_2 in Korrelation zueinander stehen, wird Q_1 von m_p indirekt beeinflusst und ist ebenfalls in der Menge $Q_p = \{Q_1, Q_2, Q_4, Q_5\}$ der von m_p beeinflussten Eigenschaften enthalten.

Die Verzerrung des Bildes hängt also nicht nur vom parallelen Ablauf der Messmethoden ab, sondern auch in welchem Zusammenhang die von den Messmethoden beeinflussten Eigenschaften zueinander stehen. Es liegt also im Ermessen des Subjekts, das Ausmaß der Komplexität sowie die Zusammenhänge im System und deren Auswirkungen, anhand von wissenschaftlicher Beurteilung abzuschätzen. Dabei ist es notwendig, verschiedene Messmethoden genauestens zu analysieren und zu beobachten, damit die Ursachen für die Verzerrung identifiziert werden können. Es ist jedoch schwierig und mühsam, den Ablauf von verschiedenen Messmethoden gesamtheitlich zu verstehen, wenn diese nicht zentral gesteuert werden. Wir gelangen daher zur nächsten technischen Herausforderung, *Kompatibilität*.

3.3.2 Kompatibilität

Durch die Komplexität der messtechnischen Fragestellung entstehen Messsysteme, in denen nur selten ein Gerät im Alleingang die gesamte Aufgabe übernimmt. Messmethoden lassen sich auf Grund unterschiedlicher technischer Ausführung der verwendeten Instrumente nicht ohne Weiteres verknüpfen. Das automatisierte Zusammenspiel ist daher nicht möglich und muss vom Experimentator per Hand übernommen werden. Die Notwendigkeit einer manuellen Intervention verzögert den gesamten Ablauf und es entsteht eine Trägheit im System, sodass erhebliche Verzerrungen im Gesamtergebnis zu erwarten sind. Beispielsweise verändert sich die chemische Zusammensetzung und Temperatur einer Lösung durch eine ablaufende Reaktion. Die Verzerrung von Messdaten ist in vielen Fällen schwer zu vermeiden, jedoch sollte sie durch ein gut abgestimmtes Zusammenspiel sowie einer guten Analyse des Systems auf ein minimales Ausmaß reduziert werden.

Sei $\Delta M = |M_{t_1} - M_{t_2}|$ die empirische Veränderung einer Messumgebung in einem Intervall $[t_1, t_2]$. Je nach spezifischer Fragestellung ist es beispielsweise sinnvoll, die Veränderung eines Teils der Messumgebung ΔM^* mit $M^* \subset M$, möglichst klein bzw. auf einen konstanten Wert zu halten, um gemessene Daten in $\overline{M^*} = M \setminus M^*$ so wenig wie möglich zu verzerren bzw. Verzerrungen rückrechnen zu können. Die Eigenschaften aus M^*, welche mit Eigenschaften aus $\overline{M^*}$ in Korrelation stehen, sind über eventuelle Verzerrungen im Gesamtbild verantwortlich.

In Abbildung 3.9 stehen alle Eigenschaften aus M^* direkt oder indirekt in Korrelation mit den Eigenschaften aus $\overline{M^*}$. Ändern sich die Eigenschaften in M^* durch eine ablaufende chemische Reaktion, so werden möglicherweise auch Eigenschaften in $\overline{M^*}$ beeinflusst und die Messergebnisse mit größer werdender Messdauer

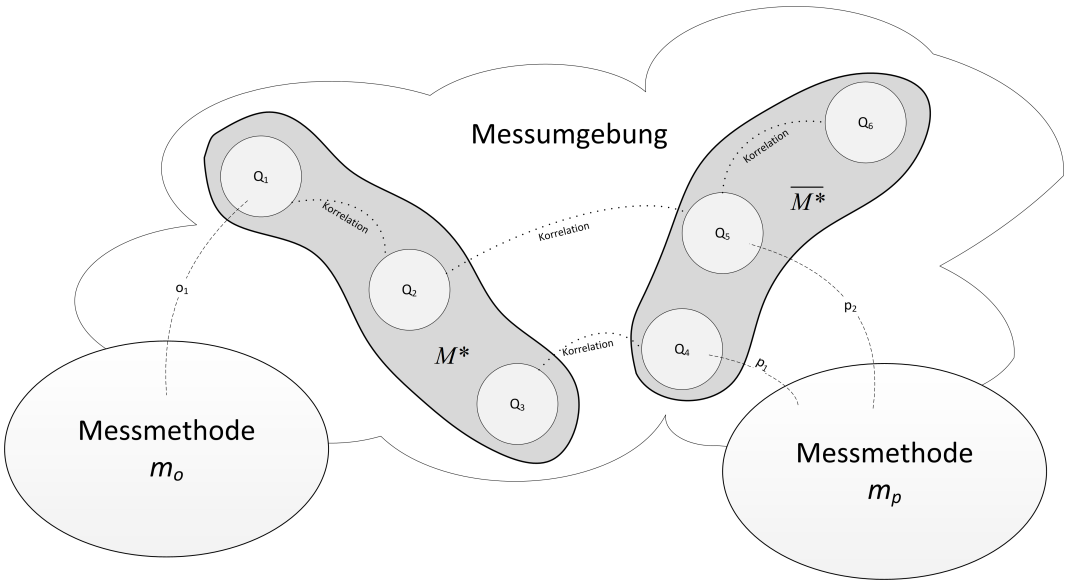


Abbildung 3.9 – Trennung der Eigenschaften in 2 Komplementärmengen zur Reduktion von Verzerrungen im Messergebnis.

verzerrt. Die Notwendigkeit eines manuellen Eingriffs in den Messablauf dehnt das Intervall $[t_1, t_2]$ stark aus wodurch ΔM^* wiederum beeinflusst wird. In anderen Fällen wäre es hilfreich zu erfahren, wie sich ein gewisses Intervall auf ΔM^* oder sogar auf ΔM selbst auswirkt, um z. B. einen Faktor zu bestimmen, welcher in weiterer Folge für Berechnungen hilfreich sein kann, um das gemessene Bild zu «entzerren». Sind die Messmethoden inkompatibel zueinander, so ist es schwierig, dieses Intervall flexibel anzupassen.

Die Komponenten einer Messeinrichtung basieren oft auf unterschiedlichen Systemplattformen. Während ein Teilsystem eine gewisse Aufgabe durchführt, hat das andere Teilsystem keine Auskunft darüber, welche Messparameter vom anderen System gesetzt bzw. verändert werden. Dem Anwender unterliegt die Aufgabe, dies genauestens zu protokollieren und den Messablauf gezielt zu steuern, um eine spätere Datenauswertung zu ermöglichen. Der Experimentator ist damit beauftragt, die Nebenläufigkeit der beiden Messmethoden zu bestimmen. Wir gelangen zum nächsten Aspekt, dem *Datenfluss*.

3.3.3 Datenfluss

Ein Datenfluss beschreibt die Möglichkeit, Information zwischen zwei Komponenten über eine Datenstruktur auszutauschen. In Kapitel 2 wurden bereits die gängigsten Schnittstellen beschrieben, mit denen es möglich ist, einen Datenfluss zwischen Messeinrichtung und Computer herzustellen. Dadurch können Einstellungen

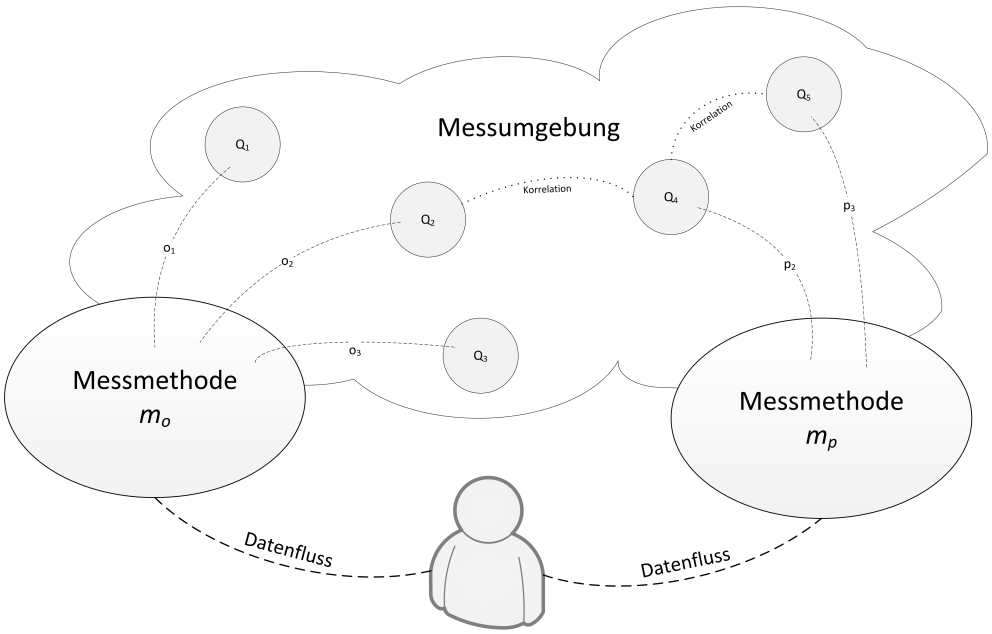


Abbildung 3.10 – Manuelle Herstellung des Datenflusses über den Experimentator.

am Gerät, indirekt, mit Hilfe einer Software, vorgenommen werden. Eine dafür geeignete Software wird vom Hersteller meistens mitgeliefert oder zum Download bereitgestellt. In komplexen Messaufgaben übernimmt nur selten ein Gerät im Alleingang die gesamte Aufgabe. Messergebnisse müssen manuell von einem Gerät auf das andere übertragen werden, falls die gewünschten Einstellungen eines Instruments vom Ergebnis eines anderen abhängen.

In der *Software Architektur* wird durch einen Datenfluss die automatische Neuberechnung von Werten durch eine Funktion ausgelöst, sobald sich ein oder mehrere Werte ändern, von denenen diese Funktion abhängig ist. Möchte man verschiedene Messmethoden miteinander verbinden, so beschreibt der Datenfluss den kausalen Zusammenhang zwischen diesen. Dadurch kann die Parallelität bzw. Nebenläufigkeit bestimmt werden. Besteht zwischen zwei Operationen, welche von unterschiedlichen und inkompatiblen Geräten durchgeführt werden, ein kausale Abhängigkeit, so muss der Datenfluss durch eine manuelle Intervention des Experimentators hergestellt werden. In Abbildung 3.10 stehen die Eigenschaften Q_2 und Q_4 in Korrelation zueinander und werden von verschiedenen Systemen erfasst. Die Operation p_2 der Messmethode m_p könnte zur Berechnung das Ergebnis der Operation o_2 aus m_o benötigen. Die manuelle Übertragung eines Parameters ist zeitaufwendig und fehleranfällig. Weiteres könnte das Ergebnis einer Operation o_x aus m_o Entscheidung darüber treffen, ob eine Operation p_y aus m_p ausgeführt wird oder nicht. Wir gelangen daher zum nächsten Thema, dem *Kontrollfluss*.

3.3.4 Kontrollfluss

Durch einen Kontrollfluss kann die zeitliche Abfolge von Befehlen bzw. Operationen einer Messmethode gesteuert werden. Kontrollstrukturen erlauben es, vom sequentiellen Ablauf der Operationen abzuweichen. Zudem können Messergebnisse verwendet werden, um Kontrollflussabhängigkeiten einzuführen und die Operationen anhand des Zustandes der Messumgebung zu steuern. Je nach Zustand von Eigenschaften in der Messumgebung werden gewisse Operationen ausgeführt oder nicht. Das Zusammenlegen mehrerer Messmethoden zu einem komplexen Messsystem gewinnt erst dann an Bedeutung, wenn diese koordiniert miteinander eingesetzt werden können. Verschiedene Komponenten im System haben Zugriff auf die selbe Messumgebung bzw. auf einen Teil davon. Ein unkoordinierter bzw. unkontrollierter Ablauf eines komplexen Messsystems kann zu unerwünschten Effekten führen, die bestenfalls zu leichten Verzerrungen der Messdaten führen, im Extremfall aber auch mechanische Beschädigungen verursachen. Als Beispiel sei die Abstandskontrolle in einem *SECM*-System erwähnt, bei deren inkorrekter Handhabung die Mikroelektrode beschädigt werden kann. Der Kontrollfluss wird mit zunehmender Anzahl der Komponenten im System schwieriger zu durchschauen; ein manueller Eingriff kann sehr bald eine ernste Hürde werden.

Ist ein kompatibler Datenfluss zwischen den relevanten Komponenten im System hergestellt, so kann eine Software den Kontrollfluss übernehmen. Die Kontrollkomponente steuert den zeitlichen und logischen Ablauf der einzelnen Gerätefunktionen im System, sodass diese sich nicht schadwirkend auf die Messumgebung ausüben. Es entstehen neue Relationen zwischen den einzelnen Komponenten, sodass die messtechnische Fragestellung durch viele Möglichkeiten ausgeweitet werden kann.

3.4 Lösungskriterien

3.4.1 Anwendbarkeit

Kommerzielle Software unterliegt während der Entwicklung einer langen Anforderungsanalyse. Unter Miteinbeziehung von potentiellen Anwendern kann der Entwicklungsprozess schon frühzeitig auf die Anforderung an die Software Rücksicht nehmen. Es folgt daraus, dass die Software sehr genau an die konkrete Problem-

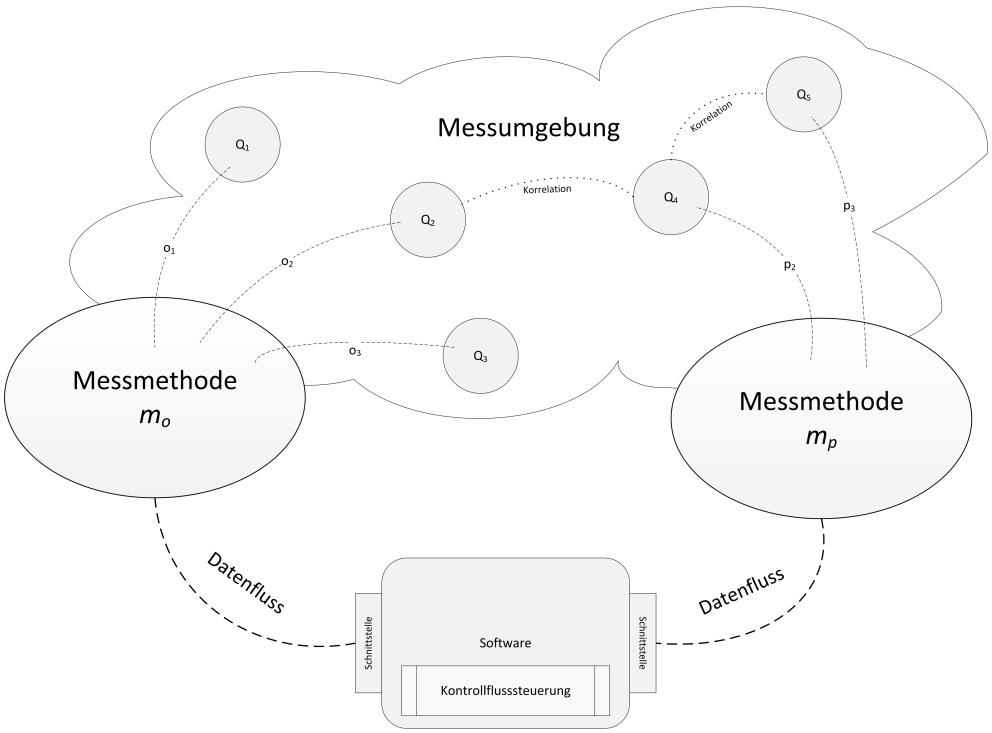


Abbildung 3.11 – Kontrollflusssteuerung und Herstellung eines kompatiblen Datenflusses über eine Softwareimplementierung.

bzw. Aufgabenstellung angepasst wird, um eine bestmögliche Anwendbarkeit zu erzielen. Die INTERNATIONALE ORGANISATION FÜR NORMIERUNG definiert *Usability* wie folgt [31]:

> *Usability eines Produktes ist das Ausmaß, in dem es von einem bestimmten Benutzer verwendet werden kann, um bestimmte Ziele in einem bestimmten Kontext effektiv, effizient und zufriedenstellend zu erreichen.*

Das Problem bei komplexen Messsystemen, die zum Teil auf selbstentwickelten Geräten basieren, ist, dass man im Vorfeld die Anwendung nicht immer klar definieren kann. Es ist daher kaum möglich, für sich ständig verändernde Aufgaben, eine Software mit ausgereiftem *Usability*-Konzept vorzulegen. Zum einen ist ein gewisses Maß an Generizität notwendig, um für verschiedene Problemstellungen gerüstet zu sein; zum anderen sind die Ziele und der Kontext vor Beginn von Messversuchen oft unscharf definiert.
Ein generisches Messsystem kann nicht zielgerichtet für die Lösung eines konkreten Problems entwickelt werden. Es muss damit gerechnet werden, dass sich das Messsystem oder einzelne Komponenten ändern, erweitert oder ausgetauscht werden. Es ist also ersichtlich, dass die Software nicht einen bestmöglichen *Usability*-Faktor gewährleisten kann. Wichtig ist aber, dass der Anwender ohne größeren Lernaufwand einen Gesamtüberblick über die Funktionsweise des Systems erhält.

3.4.2 Flexibilität

Die vielfältigen Zielsetzungen von Experimenten bedingen den Einsatz unterschiedlichster Messsysteme. Herkömmliche Systeme sind daher in der Regel für ganz bestimmte Aufgaben konstruiert [32, S.233–234]. Konsequenterweise werden komplexe Abläufe mit klar definierter Fragestellung meistens durch maßgeschneiderte Software durchgeführt. Am Beispiel der *Scanning Electrochemical Microscopy* sind kommerzielle Lösungen am Markt erhältlich, die genau für das Zusammenspiel einer bestimmten Hardware bzw. Hardwareeinstellung abgestimmt ist. Die Hard- und Software wird als Gesamtpaket angeboten und funktioniert nur in dieser Kombination. Hierbei handelt es sich um ein statisches System, welches weder auf Softwareebene noch auf Hardwareebene erweitert werden kann.
Im Falle von wissenschaftlichen Laborversuchen ist es jedoch von großer Bedeutung, messtechnisches Neuland betreten zu können, um neue Erkenntnisse und Methoden zu erforschen. Hardwaretechnische Änderungen sind durch statische Softwarelösungen nicht möglich. Bei kommerziellen Lösungen ist es notwendig, auf Updates zu warten, sobald gröbere Systemänderungen vorgenommen werden. Bestenfalls ist ein modularer Austausch möglich, jedoch ist man vom Zulieferer der Software abhängig, was in der Regel mit einer störenden Wartezeit verbunden ist. Diese Tatsache ist ein Mitgrund für den Aufbau eines eigenen *Scanning-Electrochemical-Microscopy*-Systems, auf welches in Abschnitt 6.4 eingegangen wird. Es werden zum Teil selbstentwickelte Geräte verwendet, die nicht am Markt verfügbar sind; der *Nanobistat*-Potentiostat[3] ist ein Beispiel dafür.
Veränderungen am Messsystem und an der messtechnischen Fragestellung müssen unproblematisch vorgenommen werden können, um ein gewisses Maß an Flexibilität zu gewährleisten. Durch ein vernünftiges Maß an Generizität sollen manche Elemente im System parametrisiert werden können. Diese Parametrisierung betrifft vor allem die Gerätefunktionen und die Kontrollflusssteuerung, damit diese in Abhängigkeit von der konkreten Fragestellung angepasst werden können und dadurch ein automatisierter Messablauf möglich ist.

3.4.3 Erweiterbarkeit

Ständig erscheinen neue Geräte am Markt, die eine bessere Effizienz, eine bessere Bedienbarkeit und mehr Flexibilität versprechen. Auch die Software, mit welcher diese Geräte gesteuert wird, spielt dabei eine immer wichtigere Rolle. Die Firma METROHM AUTOLAB hat erst kürzlich eine Software namens `NOVA` für ihre Geräte

[3] Der *Nanobistat* ist ein speziell für die Nanoelektrochemie entwickelter Niedrigstrom-Bipotentiostat auf welchen im Abschnitt 6.4 näher eingegangen wird.

zur Verfügung gestellt, die auf einem komplett überarbeiteten Konzept aufgebaut ist und sich laut eigenen Angaben grundlegend von den Vorgängerversionen unterscheidet [33, S.7]. Durch die wachsende Anzahl von Geräten hat der Hersteller von Potentiostaten und Galvanostaten erkannt, dass der Aufwand zu groß ist, bestehende Software mit jeder Neuerscheinung eines Gerätes neu bzw. umschreiben zu müssen.

Durch die Verfügbarkeit neuer Geräte ist es naheliegend, ältere Komponenten im System durch neuere auszutauschen. Im Falle von maßgeschneiderter Software würde dies ein Umschreiben der relevanten Programmteile erfordern. Es wäre jedoch besser, die Funktionalität eines neuen Gerätes modular in das System einbinden zu können, sodass dieses sofort verwendet werden kann.

4 Abstraktionskonzepte

> Das Ganze ist mehr als die Summe seiner Teile.
>
> *(Aristoteles)*

4.1 Übersicht

Unter Abstraktion versteht man das bewusste Weglassen bzw. Abziehen *(abs-trahere)* von Einzelheiten und Details, um eine einfachere bzw. allgemeinere Sicht auf das Beobachtete zu erhalten. In der heutigen Zeit wäre es kaum mehr möglich, große und komplexe Systeme ohne Verwendung von Abstraktion zu verstehen. Die ersten Abstraktionsprozesse fanden schon in der Steinzeit statt, als der Mensch begann Höhlenwände zu bemalen. Lange Zeit war Abstraktion ein unbewusster Denkprozess, welcher spätestens seit der Erfindung des Computers überlegt eingesetzt wird.

4.2 Merkmale von Abstraktion

Während eines stattfindenden Abstraktionsprozesses muss darauf geachtet werden, dass das Absehen von Einzelheiten und das Weglassen von wichtigen Details nicht dazu führt, das System nicht mehr ausreichend zu verstehen. Eine geschickt angewandte Abstraktion auf ein Messsystem weist folgende Merkmale auf, welche in Abbildung 4.1 zusammenfassend dargestellt werden:

a) Abstraktion ermöglicht das **Erkennen von Zusammenhängen** zwischen verschiedenen Komponenten in komplexen Systemen, ohne deren Details sichtbar zu machen. Die Zusammenhänge von unterschiedlichen Komponenten können von großer Bedeutung sein, da diese in vielen Fällen direkte oder indirekte Auswirkungen auf das Messergebnis haben. Beispielsweise ist es notwendig, den Zustand einer Komponente A zu kennen, um den Output einer anderen Komponente B richtig zu interpretieren. Strommessdaten eines *SECM*-Systems (siehe Abschnitt 6.4.3) könnten ohne Kenntnis über die genaue Position der Elektrode nicht richtig ausgewertet werden.

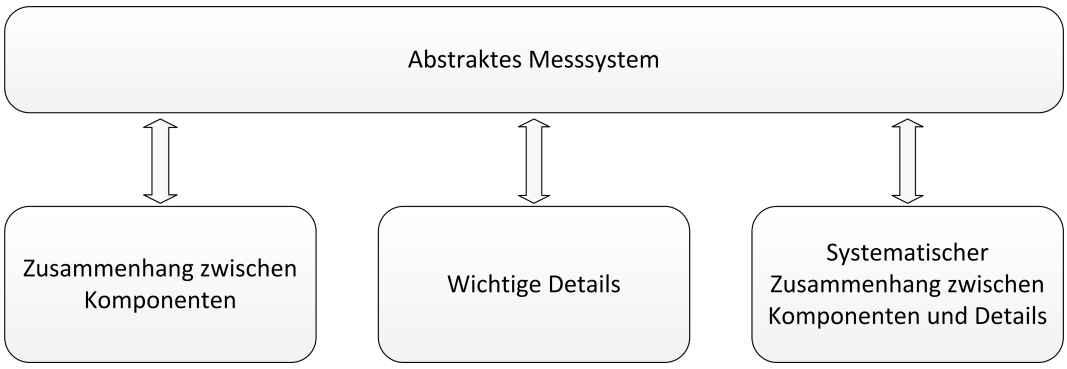


Abbildung 4.1 – Merkmale von Abstraktion in einem Messsystem.

b) Abstraktion ermöglicht das **Hervorheben eines Details** durch Verallgemeinerung des Rests. Kleine Veränderungen im Messsystem bzw. in der Messumgebung haben oft große Auswirkungen auf das Messverhalten, während sich andere Veränderungen im System kaum bemerkbar machen. Die systemrelevanten Details müssen daher dem Anwender auf jeden Fall sichtbar gemacht werden.

c) Durch Abstraktion ist der **systematische Zusammenhang** zwischen den unterschiedlichen Komponten im System und den wichtigen Details leichter zu erkennen. In komplexen Messsystemen müssen die Zusammenhänge jederzeit erkennbar sein, damit:

   1) Korrelationen zwischen Eigenschaften korrekt auf die numerischen Werte abgebildet werden, *und*

   2) die gesammelten Daten auf Grund der Übersicht über die Messumgebung auch richtig interpretiert werden können.

4.3 Abstraktion durch Middleware

Die meisten Programme für Messaufgaben kommunizieren direkt mit dem Treiber für das Messgerät bzw. mit dem Treiber für das Bussystem, an dem das Gerät angeschlossen ist. Die Implementierung des Treibers sorgt für die korrekte Übertragung bzw. Kommunikation mit dem Messgerät über den Bus. Schnittstellen und Protokolle sind jedoch sehr verschieden, sofern es sich nicht um eng verwandte Geräte handelt. Es wäre hilfreich, das heterogene System, als einheitliches, koherentes Gebilde betrachten zu können. Ein erster Ansatz ist das Hinzufügen einer Schicht

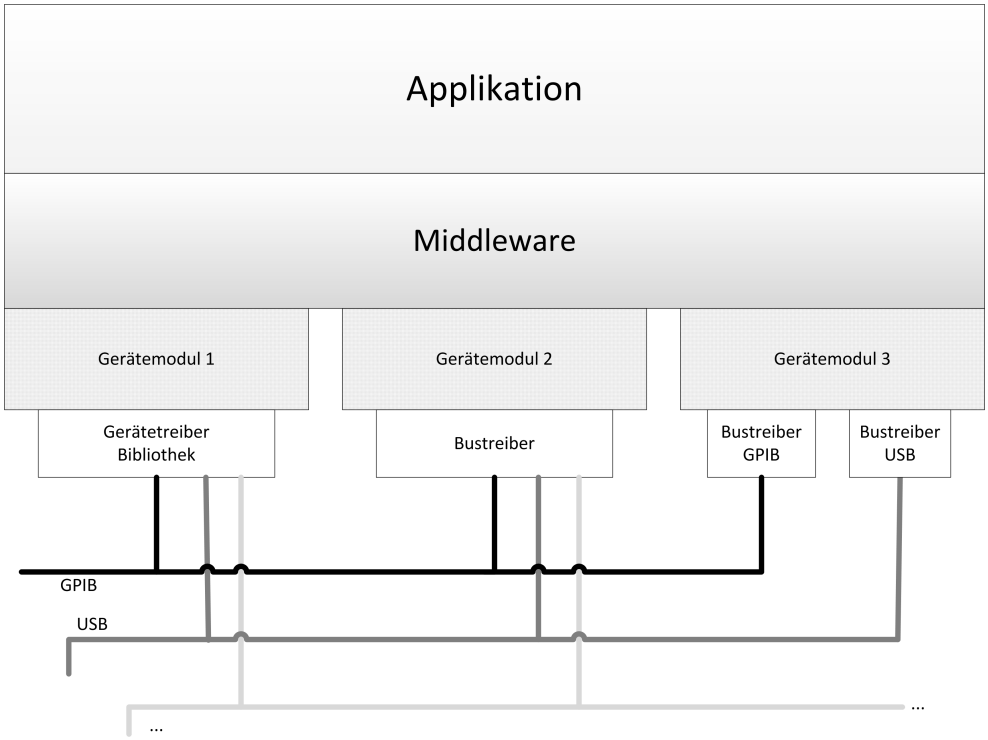


Abbildung 4.2 – Middleware zwischen Applikation und *low-level*-Gerätelogik.

zwischen Applikation und den darunterliegenden Komponenten, um einen breiteren Grad an Abstraktion zu ermöglichen. Diese Schicht trennt die Applikation von den gerätespezifischen *low-level*-Funktionen, welche eng mit der Programmlogik verflochten sind. Sie wird als **Middleware** bezeichnet und ist in Abbildung 4.2 samt der Zusammenhänge grafisch dargestellt.

Der Applikation bleibt verborgen, wie und über welchen Bus mit einem Gerät kommuniziert wird. Auch wenn es möglich ist, von der Applikation, direkt auf Gerätemodule zuzugreifen, sollte ausschließlich über die *Middleware* interagiert werden, um das Abstraktionskonzept nicht zu brechen. Durch eine einheitliche Schnittstelle der *Middleware* können dem System neue Komponenten hinzugefügt werden, ohne die Applikation verändern zu müssen. Würde zum Beispiel die Applikation das Gerät direkt über einen *GPIB*-Treiber steuern, so müsste die Software umgeschrieben werden, falls stattdessen das gleiche Gerät über eine *USB*-Schnittstelle angesprochen werden soll.

4.4 Abstraktionsmechanismen

4.4.1 Bibliotheken

Große und komplexe Programme bestehen meistens aus mehreren Teilprogrammen, von denen jedes für spezielle Aufgaben verantwortlich ist. Diese Unterprogramme bilden in der Regel keine unabhängige Einheit, sondern stellen Hilfsmodule bereit, welche von anderen Komponenten im System verwendet werden können. Damit der Zugang zu solchen Modulen vereinfacht wird, werden diese (thematisch kategorisiert) in sogenannte *Software-Bibliotheken* verpackt. Teile der Applikation, welche repetitive Aufgaben ausführen, werden in Bibliotheken ausgelagert, um eine Wiederverwendbarkeit und ein besseres Verständnis über den Aufbau des Gesamtsystems zu ermöglichen. Softwarebibliotheken können in 3 Gruppen eingeteilt werden, je nachdem ob diese in Form von Quellcode oder in Form von statisch bzw. dynamisch vorkompiliertem Code vorliegen.

Quellcode-Bibliotheken: Bei Quellcode-Bibliotheken handelt es sich um reinen Programmcode von Werten, Deklarationen, Klassen und Funktionen, der noch nicht kompiliert wurde. Der Quellcode wird zusammen mit dem eigenen Programm übersetzt.

Statische Bibliotheken: Bei dieser Art von Bibliotheken handelt es sich um bereits vorkompilierte Programmstücke. Diese werden nach dem eigentlichen Kompiliervorgang des eigenen Programms hinzugefügt. Die Software wird also um die Größe der statischen Bibliothek erweitert.

Dynamische Bibliotheken: Dynamische-Bibliotheken werden erst zur Laufzeit dazugebunden und vermeiden daher das Größerwerden des Programms. Die Nutzung kann von mehreren Programmen aus gleichzeitig erfolgen und sie werden nur bei Bedarf geladen.

Der Aufbau von einer Bibliothek selbst besteht in der Regel aus einer Schnittstelle und einer Implementierung. Das Ziel dieser logischen Trennung ist ein besseres Verständnis bzw. eine bessere Aufteilung zwischen dem was die Bibliothek macht und wie sie es umsetzt.

Schnittstellen: Schnittstellen beschreiben **was** eine Komponente tut und sollten möglichst selten verändert werden, damit die Kompatibilität zu anderen Komponenten bei einer Weiterentwicklung stets aufrecht erhalten bleibt. Sie ermöglichen eine abstrakte Sicht auf eine Komponente, indem von der eigentlichen Implementierung abgesehen wird.

Implementierung: Die Funktionalität einer Bibliothek ist hinter der Schnittstelle verborgen. Die Implementierung beschreibt also **wie** die Arbeit getan wird und ist für die Umsetzung des «**Was**» der Schnittstelle verantwortlich. Implementierungen verändern sich meist in kürzeren Abständen, ohne jedoch Auswirkungen auf die Schnittstelle zu haben. Dies ermöglicht Fehlerkorrekturen und Programmoptimierungen durch einen modularen Austausch vorzunehmen, ohne dabei Kompatibilitätsverluste in Kauf nehmen zu müssen.

Linken der Bibliotheken

Der Prozess, dem Programm eine Bibliothek hinzuzufügen, heißt, sofern die Bibliothek nicht direkt als Quellcode vorliegt und vor dem Kompilieren eingefügt wird, *Linken* oder *Binden*. Dieses Verfahren ist für die modulare Softwareentwicklung von großer Bedeutung [34] und kann zur Zeit der *Kompilierung*, des *Ladens* oder der *Ausführung* des Programms stattfinden. Geschieht der Vorgang während der Kompilierung, so spricht man von *statischem Linken*; in den beiden anderen Fällen handelt es sich um *dynamisches Linken*.

Statisches Linken: Statisches Linken fügt dem Programm alle Bibliothekskomponenten, die das Programm verwendet, statisch hinzu. Das Produkt ist eine einzige ausführbare Datei.

Dynamisches Linken: Beim dynamischen Linken werden die Bibliotheken bzw. die davon verwendeten Funktionen erst bei der Ausführung des Programms geladen. Das Programm enthält die notwendige Information, um auf Funktionen der dynamischen Bibliothek zugreifen zu können.

Dynamisches Linken kann entweder beim Start oder während der Laufzeit eines Programms stattfinden; man unterscheidet daher zusätzlich zwischen *Load-Time Dynamic Linking* und *Run-Time Dynamic Linking*.

Load-Time Dynamic Linking: Die Applikation verwendet Funktionen aus Bibliotheken, als wären diese lokal. Durch eine Importbibliothek wird dem Compiler mitgeteilt, wie der *Linker* die dynamische Bibliothek laden und die benötigten Funktionen aufrufen kann.

Run-Time Dynamic Linking: Die Applikation lädt die Funktion einer dynamischen Bibliothek während der Laufzeit, indem die Funktion durch einen symbolischen Namen aufgelöst wird. In diesem Falle ist keine Import Bibliothek notwendig.

Folgende Kriterien sind bei der Wahl zwischen *Load-Time Dynamic Linking* und *Run-Time Dynamic Linking* zu berücksichtigen:

Startzeit der Applikation

Falls die Startdauer der Applikation von sehr kurzer Zeit sein soll, so ist *Run-Time Dynamic Linking* die richtige Wahl.

Einfachheit

Beim *Load-Time Dynamic Linking* übernimmt der Compiler mit Hilfe einer Importbibliothek die notwendige Arbeit, um die exportierten Funktionen einer Bibliothek in das Programm einzubinden. Der Aufruf einer Funktion gleicht somit dem Aufruf einer lokalen Funktion.

Applikationslogik

Will man gewisse Module je nach Bedarf laden, so ist auf *Run-Time Dynamic Linking* zurückzugreifen.

Viele glauben, dass *statisches Linken* Vorteile hat. Dies ist jedoch nicht der Fall und wird es auch niemals sein [35]:

i) Durch gemeinsames Benutzen von Bibliotheken wird der Speicherplatz effizienter genutzt. Man denke an eine `printf(...)`-Funktion, die von fast jeder Applikation verwendet wird.

ii) Eine Fehlerbehebung oder das Schließen einer Sicherheitslücke kann durch einfaches Ersetzen der Komponente getätigt werden. Im Falle von *statischem Linken* müsste jedes Programm, welches mit der betroffenen Bibliothek erstellt wurde, neu kompiliert werden.

iii) Fixe Adressierung durch *statisches Linken* stellt einen guten Angriffsboden für Hacker dar.

4.4.2 Entwurfsmuster

Entwurfsmuster sind Elemente objektorientierter Software, welche immerwiederkehrende Probleme beschreiben und einen möglichen Lösungsweg dafür bereitstellen. Werden die relevanten Objekte passend abstrahiert, so kann diese Lösung beliebig oft angewendet werden [36].

Die grundlegenden Elemente eines Entwurfsmusters sind:

- Mustername
- Problemabschnitt
- Lösungsabschnitt
- Konsequenzenabschnitt

Der *Mustername* beschreibt in ein oder zwei Worten das Entwurfsproblem bzw. seine Lösungen. Durch diese Benennung ist es möglich, eine neue Abstraktionsebene zu schaffen. Es ergibt sich daraus, dass Entwurfsmuster nicht nur zur Lösung immer wiederkehrender Probleme dienen, sondern auch wesentlich zum Gesamtverständnis eines Systems und somit zur Abstraktion seiner Komponenten beitragen. Der *Problemabschnitt* beschreibt, für welche Problemstellungen das Muster geeignet ist. Vom *Lösungsabschnitt* wird eine Schablone für die konkrete Implementierung einer möglichen Lösung bereitgestellt. Konsequenzen sowie Vor- und Nachteile der Anwendung des Musters werden im *Konsequenzenabschnitt* erläutert.

Funktionsabstraktion

Jede Funktion eines Messgerätes kann entweder durch den jeweiligen Treiber und durch die dazugehörige Programmierschnittstelle oder durch direktes Schreiben eines Befehls auf den Datenbus aufgerufen werden. Eine eventuelle Busantwort liefert einen Rückgabewert der am Gerät ausgeführten Funktion oder einen Statuscode.
Messgeräte werden einerseits durch unterschiedliche Bussysteme miteinander bzw. mit dem Computer verbunden, andererseits ist die Syntax der gesendeten Busbefehle sowie die Einstellung zur korrekten Busübertragung in vielen Fällen verschieden. Um zu einer einheitlichen Darstellung von Funktionen eines Messgerätes zu gelangen ist es notwendig, dem System bzw. dem Anwender diese Fallunterscheidungen zu verbergen. Ein möglicher Abstraktionsweg ist es, jede Funktion als Objekt zu kapseln und ein einheitliches *Interface* zur Verfügung zu stellen.

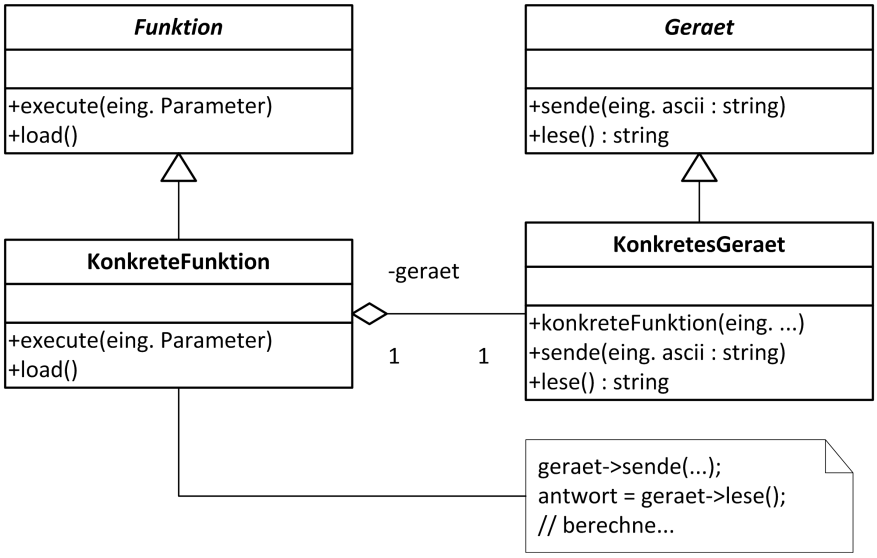


Abbildung 4.3 – Funktionsabstraktion durch Verwendung des *Befehl*smusters.

Das Entwurfsmuster *Befehl* (*Command-Pattern*) [36] bietet diese Möglichkeit an. Jede Funktion eines Gerätes wird in ein *Befehl*sobjekt gekapselt und erhält einen eindeutigen Namen. Der Name einer Funktion sowie sämtliche Funktionsparameter und Rückgabewerte können durch eine einheitliche Schnittstelle abgefragt werden. Die konkrete Implementierung des *Befehls* sowie die Kommunikation zwischen Programm und Gerät bleibt für den Anwender transparent. Auch die Komplexität des *Befehls* ist für den Anwender nicht ersichtlich. Es kann sich einerseits lediglich um den Aufruf einer Funktion am Gerät handeln oder aber auch um einen komplexeren Ablauf mehrerer Befehle, der dem Anwender als einzelne Funktion dargestellt wird. In Abbildung 4.3 ist der Sachverhalt dargestellt. Die konkrete Implementierung einer Funktion hat Zugriff auf das entsprechende Gerät und führt die notwendigen Operationen durch. Die Komplexität der konkreten Implementierung einer Funktion ist nicht ersichtlich. Die einheitliche Schnittstelle wird von den Oberklassen *Funktion* und *Geraet* bereitgestellt.

Dieser Ansatz findet sich auch in bekannten Systemen wie z. B. dem Webframework `Struts`[1] [37] wieder. In der Web-Welt wird eine *Befehl*sklasse meistens *Action* genannt und ist für die *Business*-Logik im *Request-Response* Zyklus verantwortlich. Ein konkreter Befehl wird mit Hilfe von *Action-Mapping*-Information ausgewählt und ausgeführt. Es hat sich jedoch gezeigt, dass es umständlich und ineffizient ist, für jede Funktion bzw. für jedes kleine Programmfragment eine eigene *Acti-*

[1] Hier ist die ursprüngliche Version gemeint welche lange Zeit als *de-facto* Standard für die Entwicklung von Java-Webapplikationen galt. Die Entwicklung von Struts in der Version 1 wurde im Jahr 2008 offiziell eingestellt.

on-Klasse zu erzeugen. Die Weiterentwicklung von `Struts` bzw. Verschmelzung mit `Webwork`, einem weiteren Framework für Webanwendungen, welches ebenfalls eingestellt wurde, führte zu `Struts 2` [38]. Die Verschmelzung dieser beiden Frameworks verwendet diesbezüglich einen praktischeren Ansatz, bei welchem eine *Action* nicht mehr einem einzigen *Befehl* gleichzusetzen ist. Mit Hilfe von *Reflexion* und *Introspektion* können beliebige Funktionen eines Objektes aufgerufen werden. Abstraktion ist nicht mehr strengstens an das Objekt gebunden, was jedoch einen sorgfältigeren Umgang im Design erfordert.

4.4.3 Reflexion

Reflexion ist ein Mechanismus, um während der Laufzeit eines Programms auf Information über die innere Struktur zugreifen zu können und diese zu verwenden, um das Verhalten des Programms zu beeinflussen. Diese Information beinhaltet zum Beispiel den Namen von Funktionen und von Klassen oder die Parametertypen bzw. den Rückgabetyp einer Funktion. In vielen traditionellen Programmiersprachen ist dieser Mechanismus nicht bzw. nur eingeschränkt verfügbar, da keine *Meta-Information* nach einer Kompilierung in Maschinencode erhalten bleibt. Erst modernere Programmiersprachen wie z. B. Java oder C# verfügen über einen ausgereiften Reflexionsmechanismus. Java ist jedoch nicht für die hardwarenahe Programmierung konzipiert worden und daher für die Steuerung von Messgeräten ungeeignet.

Bei der Kompilierung von einem C++-Programm geht die Information über seine innere Struktur (*Meta-Information*) verloren. In vielen Fällen wäre es äußerst hilfreich, während der Ausführung des Programmes, Kenntnis über die Struktur der Funktionen und Datentypen zu haben [39]. Den Beibehalt von *Meta-Information* über das Programm macht den Programmcode natürlich größer, was sich auch auf die Laufzeit auswirkt. In vielen Fällen, vor allem in der Messtechnik, hat Flexibilität der Messroutine einen weitaus größeren Nutzen als eine optimale Laufzeit [7]. Die Abfrage von Information über die innere Struktur eines Programms wird als *Introspektion* bezeichnet. *Reflexion* stellt die architekturelle Technik dar, um *Introspektion* zu ermöglichen [40]. Der Name *Reflexion* rührt daher, dass durch *Meta-Information* die Programmstruktur dem Programm selbst reflektiert wird. Wie schon erwähnt, geht in C++ während des Kompiliervorgangs sämtliche Information über die Programmstruktur verloren. Lediglich Information für den *Virtual Function Mechanismus* für polymorphe Objekte sowie Typidentifikation während

der Laufzeit (RTTI[2]) bleibt erhalten. Folgende Eigenschaften sind notwendig, um ein C++-Programm über die innere Programmstruktur abfragen zu können [41]:

Reifikation: Vergegenständlichung der inneren Struktur des Programms durch *Meta-Information*, z. B. *Meta-Objekte*

Introspektion: Die Möglichkeit, Information über die innere Struktur des Programms abzufragen bzw. zu modifizieren

Existierende *Meta-Object Frameworks* wie *System Object Model (SOM)* von IBM oder *Common Object Model* von MICROSOFT unterstützen die Implementierung von *Instrospektion*. Diese Frameworks erfordern entweder:

1. Objekte stehen in Untertyprelation zu einem Wurzelobjekt, welches die *Meta-Information* implementiert, *oder*
2. die Objekte sind direkte Instanzen von *Meta-Objekten*.

In [42] wird ein Ansatz zur Implementierung von *Meta-Objekten* nach dem Paradigma Nr. 2 für C++ gezeigt. Dieser Ansatz wurde verwendet, um Gerätemodule und Funktionen zu *reifizieren* und dem Task++-System, welches in Kapitel 5 beschrieben wird, verständlich zu machen.

Obige Mechanismen stellen jedoch ein Problem dar, wenn man existierende Objekte, welche ohne Zuhilfenahme eines Frameworks erstellt wurden, einbinden möchte. Ebenfalls ein Problem hat man mit geschlossenen Softwarebibliotheken, bei welchen der Quellcode nicht zugänglich ist. In [41] wird gezeigt, wie Introspektion implementiert werden kann, ohne den eigentlichen Programmcode zu verändern. In Kapitel 5 wird ein weiterer Ansatz gezeigt, um Bibliotheken, die nicht in offenem Quellcode vorliegen, in ein reflektiertes Objekt zu *wrappen*.

4.4.4 Interpretation

C++ ist eine Programmiersprache, die vom Compiler in Maschinensprache übersetzt wird. Die Generierung von Maschinencode kann je nach Programmgröße einige Zeit in Anspruch nehmen. Für manche Aufgaben ist der Prozess Kompilierung/-Linken/Ausführen/Debuggen unproduktiv und speziell für Programmieranfänger bzw. Nicht-Programmierexperten ungeeignet. Diese Tatsache führte zur Entwicklung verschiedener C/C++ Interpreter [43],[44] und [7], welche vor allem eine

[2] Runtime Type Information.

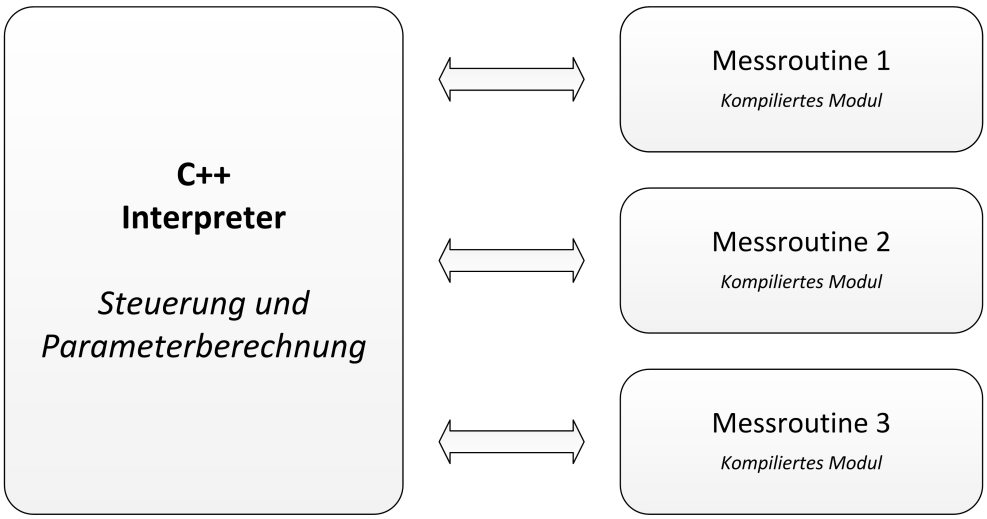


Abbildung 4.4 – Steuerung von kompilierten Messroutinen über den C++-Interpreter.

einfachere Verwendung zum Ziel haben. Sowohl der Ch C/C++ Interpreter [43] als auch der *Command Line InterPreter for a subset of C++ (CLIP)* [44] folgen dem Ziel, Programmieranfängern besser zu verstehende Fehlermeldungen zu liefern. Vor allem zu Beginn der Lernphase ist es laut den Entwicklern von CLIP von essentieller Bedeutung, dass die Programmierwerkzeuge so einfach wie möglich zu bedienen sind. Während diese beiden Systeme vor allem zu didaktischen Zwecken eingesetzt werden, wird CINT [7] hauptsächlich für messtechnische Angelegenheiten am CERN verwendet.

Während Labor- bzw. Messversuchen ist es sehr oft notwendig, kleine Änderungen am Messablauf vorzunehmen. Ein ständiges (re-)kompilieren des Programms ist daher mühsam und bremst den gesamten Messversuch. Interpreter hingegen führen das Programm sofort aus. Die Programmausführung ist in der Regel allerdings bis zu 10 mal langsamer, als wie wenn der Programmcode direkt in Maschinencode vorliegt. Ein Interpreter ist daher die optimale Wahl, wenn sich Programmcode sehr oft ändert, da Flexibilität einer optimalen Laufzeit vorausgeht. Wie schon erwähnt wurde, hat C und C++ eine natürliche Nische in der Entwicklung von hardwarenahen Anwendungen gefunden. Zwar gibt es einige Inkompatibilitäten zwischen C und C++, jedoch hat der Schöpfer von C++ – BJARNE STROUSTRUP – ausdrücklich empfohlen, diese auf ein Minimum zu halten, um Interoperatibilität zu maximieren [45]. Ein gut geschriebener C-Code sollte daher, bis auf wenige Ausnahmen, jeden C++-Compiler passieren.

Mit dem C++-Interpreter CINT hat MASAHARU GOTO [7] ein Werkzeug entwickelt, welches einen Kompromiss zwischen den oben beschriebenen Problemen

findet. MASAHARU GOTO arbeitet seit vielen Jahren für AGILENT TECHNOLOGIES, einem führenden Hersteller von Messgeräten, und kennt daher bestens die programmiertechnischen Anforderungen für komplexe Messsysteme. Der C++-Interpreter ist ein wesentlicher Bestandteil von ROOT [8], einem Datenanalyseframework, welches am CERN entwickelt wird. ROOT ersetzt die ältere, in Fortran geschriebene Software, da diese die beim *Large Hadron Collider* entstehende Datenmenge nicht bewältigen würde.

Die wichtigsten Merkmale des C++-Interpreters sind:

- Das System interpretiert eine portable Skriptsprache, welche 85–95 % des C++-Standards unterstützt.
- Interpretierter Code kann mit Maschinencode gemischt werden, indem eine vorkompilierte Bibliothek eingebunden wird.
- Die Verwendung von C++ wird vereinfacht und kann somit auch von Nicht-Programmierexperten zielführend eingesetzt werden.

Durch die Möglichkeit, interpretierbaren Programmcode mit kompilierten Modulen zu verbinden, können Messroutinen erstellt und durch Kontrollstrukturen sowie weiteren Anweisungen, gesteuert und parametrisiert werden.

Die Enwickler von CLIP sehen die Verwendung von CINT jedoch wenig intuitiv [44] und die Integration eigener Module als schwierig, sodass diese sich entschlossen haben, einen eigenen Interpreter zu entwickeln.

4.4.5 Beschreibung von Tasks

Dynamische Messumgebungen erfordern eine effiziente und präzise Koordination zwischen sämtlichen Routinen. Konventionelle Programmiersprachen führen mit zunehmender Komplexität des Messsystems zu stark nicht-linearem Code, welcher oft sehr schwer zu verstehen bzw. zu debuggen und warten ist [6]. Grund dafür sind die vielen Fallunterscheidungen im Programmablauf. Durch eine geeignete Abstraktion durch Tasks, welche kleine Messaufgaben bzw. Aktionen im System darstellen, ist eine verständlichere Sicht auf den komplexen Messablauf möglich.

Im Gegensatz zum C++-Interpreter ist die *Task Description Language* TDL [6] eine C++-Erweiterung, welche es ermöglicht, Tasks auf einer abstrakten Ebene

im Programmcode zu beschreiben. Dieser Programmcode wird vom TDL Compiler in kompilierbarem C++-Code übersetzt. Die Taskbeschreibungssprache TDL ist zur Koordination von Aktionen in Roboter-Systemen entwickelt worden. Ein komplexes Messsystem kann mit einem Roboter insofern verglichen werden, dass die Messumgebung der Umgebung des Roboters entspricht und der Messablauf bzw. die Aktionen des Roboters von seiner Umgebung abhängen.

Taskbäume

Der Programmablauf wird in TDL durch einen Taskbaum dargestellt. Dieser ermöglicht eine strukturierte Sicht über das Gesamtsystem und hebt Abhängigkeiten zwischen einzelnen Tasks hervor. Jeder Knoten im Baum entspricht einem Task, welcher im wesentlichen ein ausführbares und parametrisierbares Stück Programmcode ist. Durch die Darstellung der Tasks als Baumstruktur werden zum einen Eltern-Kind-Beziehungen sichtbar, zum anderen wird der zeitliche Ablauf der Tasks sowie deren Zusammenhänge dargestellt. Durch eine Momentaufnahme der Messumgebung können im aktuellen Task Entscheidungen über den weiteren Ablauf getroffen und Messparameter dementsprechend gesetzt werden. Ein Taskbaum besteht im wesentlichen aus 2 Typen von Knoten:

Kommandoknoten: Kommandoknoten (*commands*) beschreiben das Verhalten bzw. die Aktionen im System und sind üblicherweise die Blattknoten im Baum.

Zielknoten: Zielknoten (*goals*) stellen Tasks höherer Ordnung dar und gruppieren weitere Knoten, welche bei Bedarf expandiert werden können. Die Knoten dieser Art beschreiben einen gewünschten Zustand und stellen somit ein Teilziel im Gesamtablauf dar. Von den darunterliegenden Aktionen im System wird abstrahiert.

Die Abbildungen 4.5 und 4.6 erläutern das Prinzip anhand des typischen Ablaufs in einem *Scanning-Electrochemical-Microscopy*-System (siehe Abschnitt 6.4). Jeder Knoten im Taskbaum kann sich in einem von 4 Zuständen befinden. Diese sind *disabled, enabled, active* und *completed*. Als *handling* eines einzelnen Knotens wird der aktuelle Zustand bezeichnet:

$$handling(n : node) \in \{disabled, enabled, active, completed\}$$

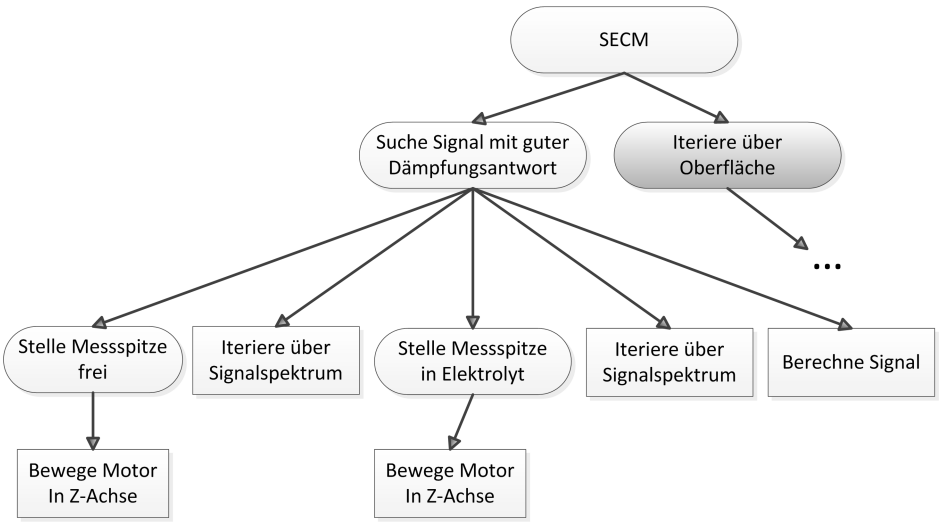


Abbildung 4.5 – Darstellung einer vereinfachten Taskstruktur für den *Frequency-Scan*-Ablauf in einem *Scanning-Electrochemical-Microscopy*-System.

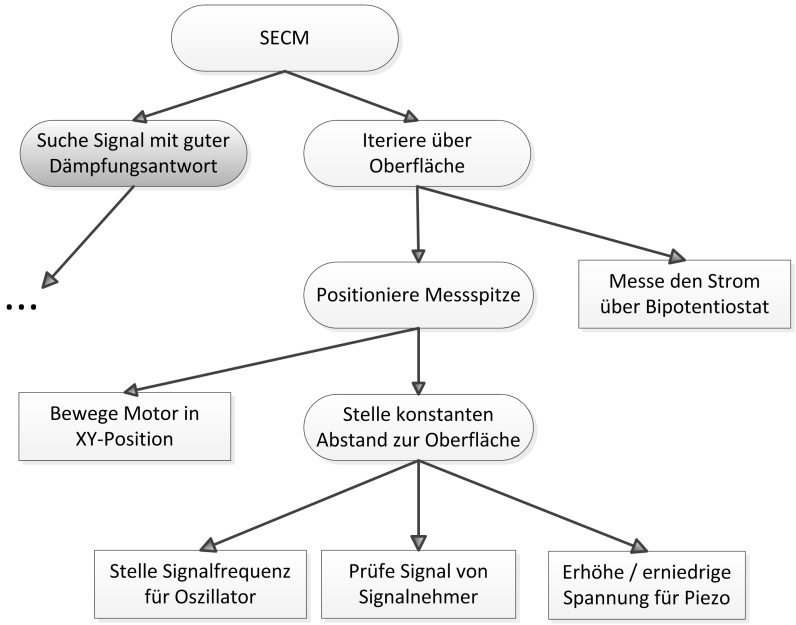


Abbildung 4.6 – Darstellung einer vereinfachten Taskstruktur für den *Surface-Scan*-Ablauf in einem *Scanning-Electrochemical-Microscopy*-System.

Der Gesamtzustand eines Knotens samt seiner Unterknoten wird als *expansion* bezeichnet; der Zustand von Kommandoknoten (*commands*) als *execution*:

$$expansion(n : node) \in \{disabled, enabled, active, completed\}$$
$$execution(n : node) \in \{disabled, enabled, active, completed\}$$

Diese Notation ermöglicht eine intuitive Sicht auf die verschiedenen Zustände der einzelnen Knoten. Ein Unterbaum wird *expandiert (expanded)*, indem alle seine Zielknoten (goals) *gehandelt (handling)* werden. Befindet sich ein Zielknoten im Zustand *completed* ($expansion(n : node) = completed$), so impliziert dies, dass die Zustände von *handling* all seiner Unterknoten ebenfalls *completed* sind. Sei $Tree(N)$ die Menge aller Unterknoten eines Knotens N und $goal(n)$ das Prädikat für einen Zielknoten, so kann diese Beziehung formal dargestellt werden durch:

$$\forall(N : goal)\, expansion(N, completed) \Rightarrow$$
$$[\forall(n \in Tree(N))\, goal(n) \Rightarrow handling(n, completed)]$$

Die Tasks für den *Frequency Scan* in Abbildung 4.5 haben eine (von links nach rechts gesehen) lineare Abhängigkeit. Der Task *Iteriere über Oberfläche* befindet sich im Zustand *disabled*, solange keine geeignete Frequenz für die *Shear-Force*-Kontrolle (siehe Abschnitt 6.4.3) gefunden wurde. Die Unterknoten für den Zielknoten *Suche Signal mit guter Dämpfungsantwort* sind ebenfalls linar abhängig. Sobald der Vorgang der Signalsuche beendet ist, kann der Zustand des Knotens *Iteriere über Oberfläche* aktiviert werden (*enabled*). Der Knoten wartet in diesem Zustand, falls notwendige Resourcen im Moment nicht zur Verfügung stehen. Beispielsweise könnte ein Gerät im System ausgeschaltet sein oder nicht reagieren. Läuft das System jedoch einwandfrei, so wechselt der Zustand des Knotens in *active* und bleibt solange in diesem Zustand, bis alle seine Unterknoten in den Zustand *completed* wechseln.
Die Tasks im Unterbaum von *Iteriere über Oberfläche* in Abbildung 4.6 müssen nicht notwendigerweise in linearer Abhängigkeit zueinander stehen. Der Task *Stelle konstanten Abstand zur Oberfläche* kann sofort aktiviert werden. Während der Motor die XY-Position anfährt, stellt dieser Task sicher, dass die Messspitze nicht an einer unebenen Stelle der Oberfläche anstößt bzw. hängenbleibt und im schlimmsten Fall sogar beschädigt wird. Auch der Task *Messe den Strom über Bipotentiostat* kann im Falle einer ortskontinuierlichen Auflösung sofort aktiviert werden. Im Falle einer ortsdiskreten (Raster-) Aufnahme wird die Strommessung nur dann aktiviert, wenn die XY-Position erreicht ist.

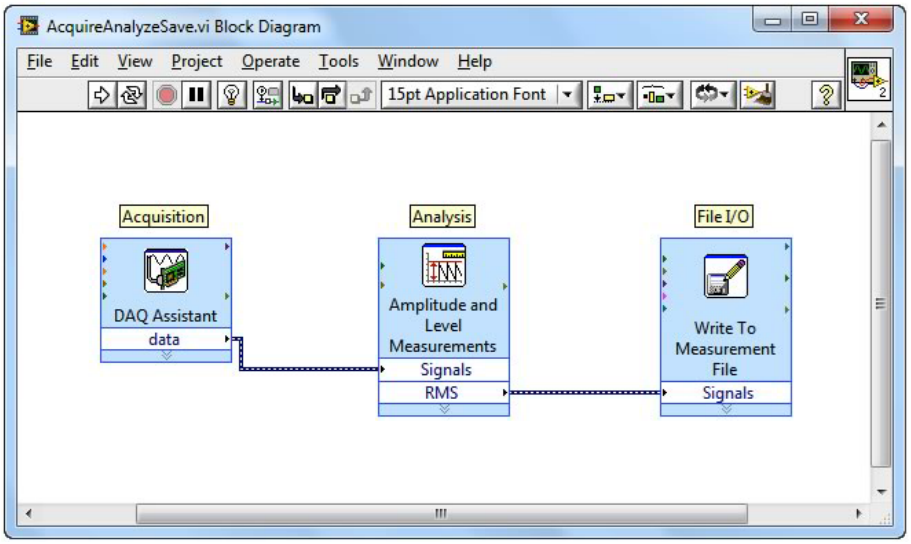


Abbildung 4.7 – Darstellung eines Blockdiagramms in LabVIEW.
Quelle: Vorteile der grafischen Programmierung mit NI LabVIEW

4.5 Softwarelösungen

4.5.1 LabVIEW

LabVIEW steht für «Laboratory Virtual Instrumentation Engineering Workbench» und ist ein von NATIONAL INSTRUMENTS vertriebenes Softwaresystem, welches eine visuelle Programmierumgebung zur Steuerung und Regelung von Geräten in unterschiedlichen Anwendungsbereichen bietet.
Das LabVIEW Projekt hat seine Ursprünge im April 1983 und wurde im Oktober 1986 in der Version 1.0 für Macintosh fertig gestellt. Das Herz von LabVIEW ist eine visuelle Programmiersprache namens G, dessen Ziel es war, die Mess- und Automatisierungstechnik zu revolutionieren. Wissenschaftlern und Ingenieuren soll eine Möglichkeit geboten werden, Messgeräte durch *Interfacing* in das System einbinden und steuern zu können. Dank der grafischen Programmiermöglichkeit konnte sich die Software sowohl bei Anfängern als auch bei Experten etablieren und gewann somit immer mehr an Popularität. Es hat jedoch bis 1992 gedauert, bis LabVIEW auch für andere Systemplattformen wie Microsoft Windows zur Verfügung gestellt wurde. Durch die grafische Darstellung von *Virtuellen Instrumenten* sowie Kontrollanweisungen auf diesen, wird eine abstrakte Sicht auf den Messablauf ermöglicht. Der Programmierer benötigt keine oder nur wenige Kenntnisse über systemnahe Programmierung, da LabVIEW die grafische Darstellung in ausführbarem Programmcode übersetzt. Um den Überblick des Gesamtablaufs nicht zu verlieren, können einzelne Teile der Programmstruktur in eine Komponente mit wohldefinierter Schnittstelle gekapselt werden.

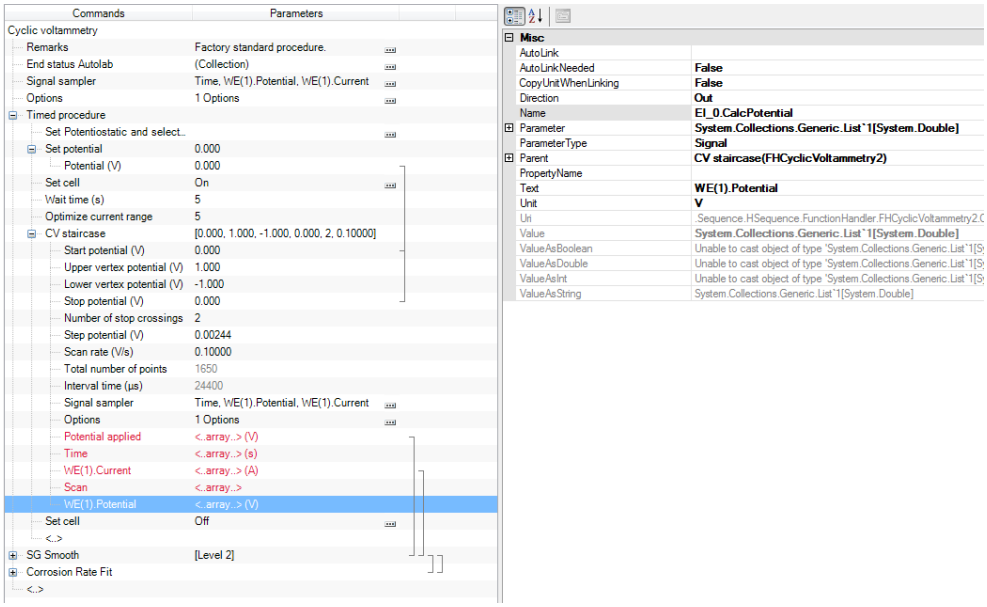


Abbildung 4.8 – Prozedur für eine Zyklovoltammetrie in NOVA von METROHM AUTOLAB. *Quelle: [46]*

4.5.2 NOVA

NOVA ist eine jüngst erschienene Software zur Steuerung von elektrochemischen Messgeräten von METROHM AUTOLAB. Dieses Softwarepaket baut auf das .NET Framework auf und verspricht bessere Flexibilität zur Steuerung von Potentiostaten und Galvanostaten, als ihre bisherigen Lösungen. Laut eigenen Angaben unterscheidet sich NOVA sehr stark von den meisten Softwarepaketen für die Elektrochemie [33, S.7].

Der methodenbasierte Ansatz der Vorgängerversionen von NOVA hatte den Nachteil, dass die sogenannten *Use-cases* nicht verändert werden konnten. Es waren fix definierte Messroutinen vorgegeben; neue Routinen waren erst nach einem Software-Update verfügbar. Mit der jungen Version von NOVA verfolgt der Hersteller von Potentiostaten und Galvanostaten einen neuartigen Ansatz, mit welchem der Benutzer Zugriff auf sämtliche Objekte hat, die den *low-level*-Funktionen der Geräte entsprechen. Der Anwender hat dadurch die Möglichkeit, aus einer großen Auswahl an Funktionen, einfache und komplexe Messroutinen selbst zusammenzubauen. Das Einbinden von externen Geräten ist nur eingeschränkt möglich, wenn es sich dabei nicht um um hauseigene Instrumente handelt. Zwar unterstützt das System den Anschluss an *RS-232* oder *Digital Input/Output*, jedoch ist für die Verwendung von Geräten mit *GPIB*- oder *PCI*-Schnittstellen im Tutorial für die Einbindung externer Geräte [47] keine Information vorhanden.

4.5.3 Task++

Task++ ist ein selbstentwickeltes Softwaresystem, welches durch die in diesem Kapitel beschriebenen Abstraktionsmechanismen einen Kompromiss zwischen Anwendbarkeit, Flexibilität und Erweiterbarkeit in Messaufgaben zu finden versucht. Das Softwaresystem wird im nächsten Kapitel ausführlich beschrieben.

5 Task++

Ich kenne keinen sicheren Weg zum Erfolg, aber einen sicheren Weg zum Misserfolg: Es allen Recht machen zu wollen.

(Platon)

5.1 Übersicht

Task++ ist ein selbstentwickeltes Softwaresystem zur Steuerung von komplexen Messabläufen. Durch die Analyse von sämtlichen messtechnischen Fragestellungen ist ein breites Spektrum an Anforderungen entstanden, denen diese Eigenentwicklung entgegenkommen soll. Die theoretischen und technischen Aspekte aus Kapitel 3 dienen dazu, die Messabläufe so zu modellieren, dass diese ein bestmögliches Resultat liefern. Weniger komplexe Abläufe oder nahezu triviale Aufgaben stehen ebenfalls oft an der Tagesordnung, sodass der Aufwand zu groß wäre, für jede Angelegenheit eine eigene Software zu entwickeln. Durch die im vorhergehenden Kapitel beschriebenen Abstraktionsmechanismen wird versucht, einen Kompromiss zwischen **Anwendbarkeit**, **Flexibilität** und **Erweiterbarkeit** einzugehen.

Anwendbarkeit: Üblicherweise durchlaufen zugeschnittene Softwarelösungen eine lange *Usability*-Studie und werden unter Miteinbeziehung von potentiellen Anwendern erstellt, bevor sie auf den Markt gelangen. Maßgeschneiderte Software hat jedoch den Nachteil, dass sie eben maßgeschneidert ist. Bei größeren Abweichungen der messtechnischen Fragestellung kann der Kragen schnell einmal zu eng werden und die Software ist nicht mehr zielführend einsetzbar. Der Anwender sollte daher die Software bedienen können, auch wenn diese nicht zielgerichtet für eine bestimmte Aufgabe erstellt wurde.

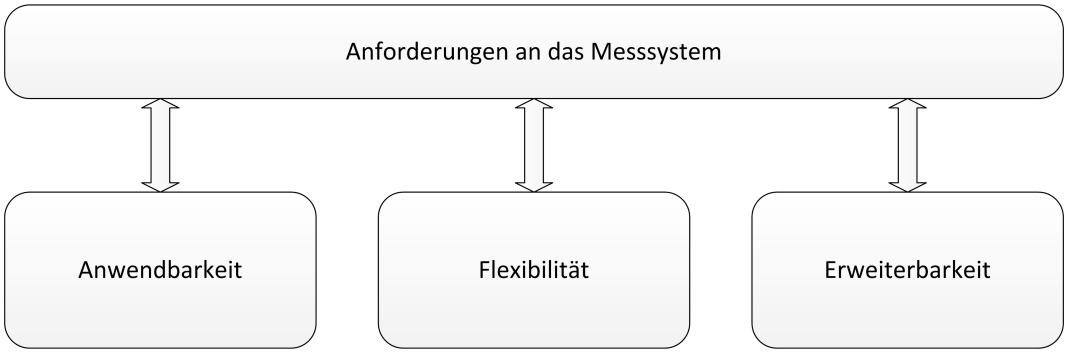


Abbildung 5.1 – Anforderungen an ein komplexes Messsystem.

Flexibilität: Bei stärker abweichenden Anforderungen ist eine speziell an die konkrete Aufgabe angepasste Software nicht mehr geeignet. Es ist erwünscht, dass die Software mit verschiedensten Problemstellungen umgehen kann, ohne dass sie jedesmal neu geschrieben werden muss. Durch Generischhalten der sich oft ändernden Messparameter kann ein gewisses Maß an Flexibilität erreicht werden.

Erweiterbarkeit: Im Laufe der Zeit unterliegen Messsysteme Veränderungen. Komponenten im System werden durch neuere oder passendere ausgetauscht oder es werden zusätzliche Module eingefügt. Bei maßgeschneiderter Software bedeutet dies eine komplette Neuerstellung des Programms; auch `LabVIEW` wird diesbezüglich in die Kritik genommen. Task++ versucht durch modulares Einbinden von neuen Komponenten dieses Problem zu umgehen.

Philosophie von Task++ ist es, das gesamte Messsystem, ähnlich wie bei der *Task Description Language* (siehe Abschnitt 4.4.5), in Tasks zu zerlegen. Ein Task ist ein kleines, parametrisierbares Programmfragment, welches mit einem oder auch mehreren Messgeräten kommuniziert, um eine gewisse Aktion im System auszuführen bzw. einen gewünschten Zustand herzustellen.

Der Umfang eines Tasks ist **gross wie nötig**, um eine sinnvolle Aufgabe durchzuführen und **klein genug**, um eine brauchbare Wiederverwendung zu ermöglichen. Die tatsächliche Größe hängt natürlich von der gewünschten Aufgabe sowie von der gewünschten Flexibilität des Tasks ab. Grundelemente der Tasks sind abstrak-

te Gerätebefehle[1], die das Messgerät anweisen, einen gewissen Operationsablauf durchzuführen sowie Steuerbefehle (*If-Statements*, Schleifen usw.), um einen Kontrollfluss im Programmablauf zu ermöglichen. Mehrere Tasks können durch einen übergeordneten Task zusammengefasst und durch geeignete Parametrisierung der notwendigen Details an verschiedenste Situationen angepasst werden.

5.2 Verwendung der Abstraktionskonzepte

5.2.1 Modulares Binden

Beim Erwerb eines Messgerätes erhält man üblicherweise eine Menge an Zusatzmaterial, welches im Lieferumfang enthalten ist oder vom Hersteller zum Download bereitgestellt wird. Die softwaremäßige Steuerung von Geräten ist heutzutage schon fast eine Voraussetzung, um am Markt konkurrenzfähig zu sein [2]. Die vom Hersteller angebotene Software ist jedoch nicht immer für die konkrete Messaufgabe geeignet, sodass durch die Bereitstellung von Treibern und Modulen dem Anwender zusätzliche Flexibilität ermöglicht wird. Mit Hilfe von Programmierschnittstellen und Protokollbeschreibungen hat der Anwender die Möglichkeit, den Messablauf durch selbst entwickelte Programme zu steuern. Die eigenen Programme können in Module verpackt und als Bibliothek in weiterer Folge wiederverwendet werden. Die Einbindung dieser Bibliotheken in ein anderes System kann entweder *statisch* oder *dynamisch* erfolgen (siehe Abschnitt 4.4.1).

Statisches Binden

Im Falle von *statischer Bindung* einer Bibliothek, ist diese ein fest verankerter Teil des Gesamtsystems. Die Erweiterung des Systems durch ein neues Gerät erfordert die Rekompilierung des Programms. Dies ist jedoch unerwünscht, da eine Systemerweiterung mit einer störenden Verzögerung verbunden ist. Eine Lösung dieses Problems erreicht man durch *dynamisches Binden*.

Dynamisches Binden

Mit Hilfe von *dynamischem Binden* können die Bibliotheken zur Kommunikation eines Gerätes zur Laufzeit in das Task++-System eingebunden werden. Sowohl

[1] Eine Zusammenfassung von konkreten physikalischen Befehlen die auf den Datenbus geschrieben werden.

beim statischen als auch beim dynamischen Binden müssen die Einstiegspunkte einer Funktion bekannt sein, da der Programmcode bereits in kompilierter Version vorliegt. Im Falle einer statischen Bindung sowie einer dynamischen Bindung zur Ladezeit (*Load-Time Dynamic Binding*) erhält der Compiler die notwendige Information, um die Bibliothek richtig hinzuzufügen. Im Falle einer dynamischen Bindung während der Ausführungszeit (*Run-Time Dynamic Binding*) muss der Programmierer selbst dem Programm diese Information zur Verfügung stellen. Funktionen verschiedener Bibliotheken haben jedoch verschiedene Eintrittspunkte und verschiedene Signaturen. Es ist daher eine Zwischenstufe notwendig, um Softwarebibliotheken von Drittanbietern dynamisch einbinden zu können, ohne das gesamte Programm neu kompilieren zu müssen. Jedes Gerätemodul wird hierfür in ein weiteres Objekt gekapselt, welches dem System eine einheitliche Schnittstelle auf die Gerätefunktionen zur Verfügung stellt. Die folgenden Abschnitte beschreiben, wie dieser Mechanismus umgesetzt wird.

5.2.2 Funktionsabstraktion

Um der Applikation einen einheitlichen Zugriff auf *low-level*-Funktionen zu ermöglichen, werden diese durch eine *Middleware*-Schicht von den darüber liegenden Komponenten getrennt. Dadurch werden Inkompatibilitäten umgangen und von der konkreten Implementierung einer Funktion wird abstrahiert. Als Beispiel sei folgendes Programmfragment in C++ gegeben:

Listing 5.1 – Abstraktion von Rückgabewert und Eingangsparameter einer Funktion.

```
1 Value measure_current_dc(Function *f) {
2   stringstream ss;
3   int ch = f->getValue(0).get<int>();
4   ss << "meas:curr:dc? (@" << ch << ")";
5   send(ss.str().c_str());
6   return Value(RealValue<string>(read()));
7 }
```

Jede Funktion eines einzubindenden Gerätemoduls erhält den selben Eingangsparameter- und Rückgabetyp. Der übergebene Parameter `Function *f` ist ein Zeiger auf ein Funktionsobjekt (*Meta-Objekt*) und erlaubt Zugriff auf die von der Implementierung benötigten Werte. Der Rückgabewert `Value` repräsentiert einen abstrakten Wert, der auch `void` darstellen kann. In Zeile 3 in Listing 5.1 wird der benötigte Parameter über den Funktionszeiger[2] ausgelesen und der Variablen `ch` zugeordnet; es wird ein *Integer*-Wert erwartet. Zeile 4 enthält den ASCII-Befehl welcher dem Ge-

[2] Eine Variable welche auf die Speicheradresse einer Funktion zeigt [48].

rät geschickt wird, um den Gleichstrom eines Kanals zu messen. Schließlich wird der Kanal dem ASCII-Befehl hinzugefügt welcher in Zeile 5 über die `send`-Funktion auf den Bus geschrieben wird. Die `read`-Funktion in Zeile 6 liest die Busantwort, welche als Konstruktorparameter für das `RealValue<string>`-Objekt dient. Dieses wird wiederum in ein einheitliches `Value`-Objekt verpackt und zurückgegeben.

5.2.3 Reflexion

Der Name der im letzten Abschnitt beschriebenen Funktion (`measure_current_dc`) geht in C++ nach einer Kompilierung verloren, da der Maschinencode nur mehr seine Speicheradresse benötigt, um das Programm auszuführen. Ziel ist es jedoch, die Funktion zu einem späteren Zeitpunkt über die *Middleware*-Schicht hinaus über ihren Namen aufzufinden und aufzurufen. Dasselbe gilt auch für Geräteobjekte und deren Attribute. Hierzu muss die Struktur der Geräte- und Funktionsobjekte auf eine *Meta*-Ebene gebracht und dem System vergegenständlicht werden.
Durch die Erzeugung von *Meta-Objekten* kann während der Laufzeit auf diese Information zugegriffen werden. Mit Hilfe von Makros können in Task++ *Meta-Objekte* erstellt werden. Der Code in Listing 5.2 zeigt die Einbindung der Funktion aus Listing 5.1 für das Gerät *Agilent 34970A*.

Listing 5.2 – Erzeugung eines *Meta-Objekts* mit Hilfe der bereitgestellten Makros.

```
1 BEGIN_DEFINITION("agilent34970a",
2                  "Agilent 34970A Data Aqcuisition / Switch Unit")
3   FUNCTION("measure_current_dc", measure_current_dc,
4            TASKPP_STRING)
5     NAME("Measure Current DC")
6     DESCRIPTION("Measures the direct current on a channel"
7                 "channel_nr given as parameter")
8     ATTRIBUTE("channel_nr", TASKPP_INT)
9 END_DEFINITION
```

`BEGIN_DEFINITION` leitet die Erstellung eines *Meta-Objektes* für ein Gerät ein. Der erste Parameter des Makrobefehls ist ein eindeutiger Name für das Gerät und stellt in Task++ den Typ des Gerätes dar. Der zweite Parameter ist der vollständige Name des Gerätes, der dem Benutzer angezeigt wird. Durch `END_DEFINITION` wird die Definition eines Gerätes abgeschlossen.
Funktionsdefinitionen werden mit dem Makrobefehl `FUNCTION` eingeleitet, welcher 3 Parameter erhält. Der erste Parameter ist eine Zeichenkette und stellt den Funktionsnamen dar, mit dessen Hilfe in weiterer Folge die Funktion im System gefunden bzw. angezeigt wird. Der zweite Parameter ist der Funktionszeiger, durch den die Funktion mit Hilfe eines *call-back*-Mechanismus aufgerufen werden kann. Der dritte Parameter stellt letztendlich den Rückgabewert der Funktion dar.

Die beiden Makros NAME und DESCRIPTION sind optional und erhalten jeweils eine Zeichenkonstante mit dem Namen der Funktion sowie eine Beschreibung, die dem Benutzer angezeigt wird und zum besseren Verständnis dienen soll. Nun müssen noch die Parameter der Funktion beschrieben werden. Dies geschieht durch das Makro ATTRIBUTE, welches einen Namen als Zeichenkette sowie einen von den unterstützten Typen erhält. Der Typ eines Parameters kann ein ganzzahliger Wert (TASKPP_INT), eine Gleitkommazahl mit doppelter Genauigkeit (TASKPP_DOUBLE) oder eine Zeichenkette (TASKPP_STRING) sein. Ein Komplettbeispiel für die Erstellung eines Gerätemoduls findet sich im Anhang A.2.

5.2.4 Interpretation

Die soeben eingebundene Funktion kann jetzt in der Task++-Skriptsprache verwendet werden. Listing 5.3 zeigt die Syntax für den Aufruf einer Funktion am *Agilent 34970A*, dessen Rückgabewert auf der Konsole ausgegeben wird.

Listing 5.3 – Aufruf einer Gerätefunktion mit Task++.

```
1 device $ag("agilent34970a");
2 $result = $ag.measure_current_dc(101);
3 print("Ergebnis: " + $result);
```

In Zeile 1 wird durch das Keyword `device` eine Instanz des Gerätes vom Typ *agilent34970a* erstellt. Nun kann in weiterer Folge über den Variablennamen `$ag` auf dieses Gerät bzw. auf dessen Funktionen zugegriffen werden. In Zeile 2 wird die Funktion aufgerufen und als Parameter wird die Kanalnummer `101` übergeben. Der Rückgabewert wird der Variablen `$result` zugewiesen, welche in Zeile 3 über die `print`-Funktion auf der Konsole ausgegeben wird.

Ein Messablauf besteht in der Regel aus einer Folge von mehreren, kleineren und überschaubaren Tasks. Es ist nun eine weitere Schicht notwendig, welche den Ablauf dieser Tasks steuert, damit die Parameter in Abhängigkeit von der Messumgebung angepasst werden können bzw. der Messvorgang beendet wird, sobald gewisse Kriterien erfüllt sind. Beispielsweise wäre es denkbar, einen Task solange zu wiederholen, bis der Messwert eines Gerätes unter einem bestimmten Wert liegt. Das Zustandsdiagramm in Abbildung 5.2 verdeutlicht das Prinzip. Listing 5.4 zeigt eine mögliche Implementierung; die Syntax von Task++ wird im Abschnitt 5.4 näher beschrieben.

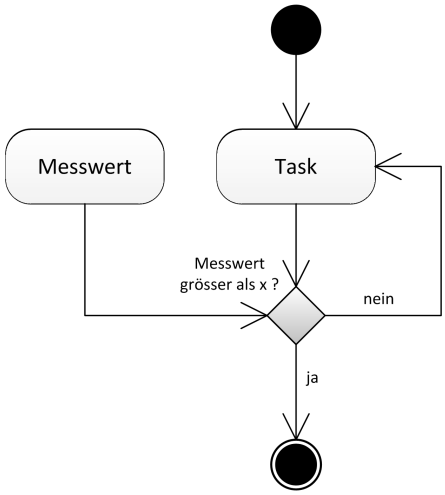


Abbildung 5.2 – Zustandsdiagramm für die Ablaufkontrolle eines Tasks.

Listing 5.4 – Kontrollflusssteuerung eines Tasks.

```
1 device $ag("agilent34970a");
2 $x = 0.5;
3 do {
4   // fuehre Task aus...
5   // ...
6   $v = $ag.read_voltage(104);
7 } while ($v <= $x);
```

5.3 Architektur und Implementierung

Das Herzstück der Architektur vom Task++-System ist eine *Virtuelle Maschine*, welche eine Softwareschicht zwischen Hardware und Applikation bildet. Durch Verwendung der Abstraktionskonzepte aus Kapitel 4 werden Inkompatibilitäten umgangen, wodurch die Einsatzflexibilität der darunter liegenden Hardware sowie die Interoperabilität dieser mit der darüber liegenden Software gesteigert wird. Eine grobe Darstellung dieses Ansatzes wird in Abbildung 5.3 gezeigt. *Meta-Objekte* kapseln die gerätespezifischen Funktionen, sodass diese über ihren Namen aufgefunden und angesprochen werden können. Dazu werden sie vom Task++-System bei einer *Registry*[3] eingetragen. Darüber liegende Schichten wie z. B. das *Graphical User Interface (GUI)*, haben Zugriff auf die *Registry* und auf das Task++-System. Der Anwender hat nun die Möglichkeit, durch Auswahl der geeigneten Funktionen, einen Messablauf zusammenzubauen, welcher in weiterer Folge von der virtuellen Maschine abgearbeitet wird.

[3] A well-known object that other objects can use to find common objects and services [49].

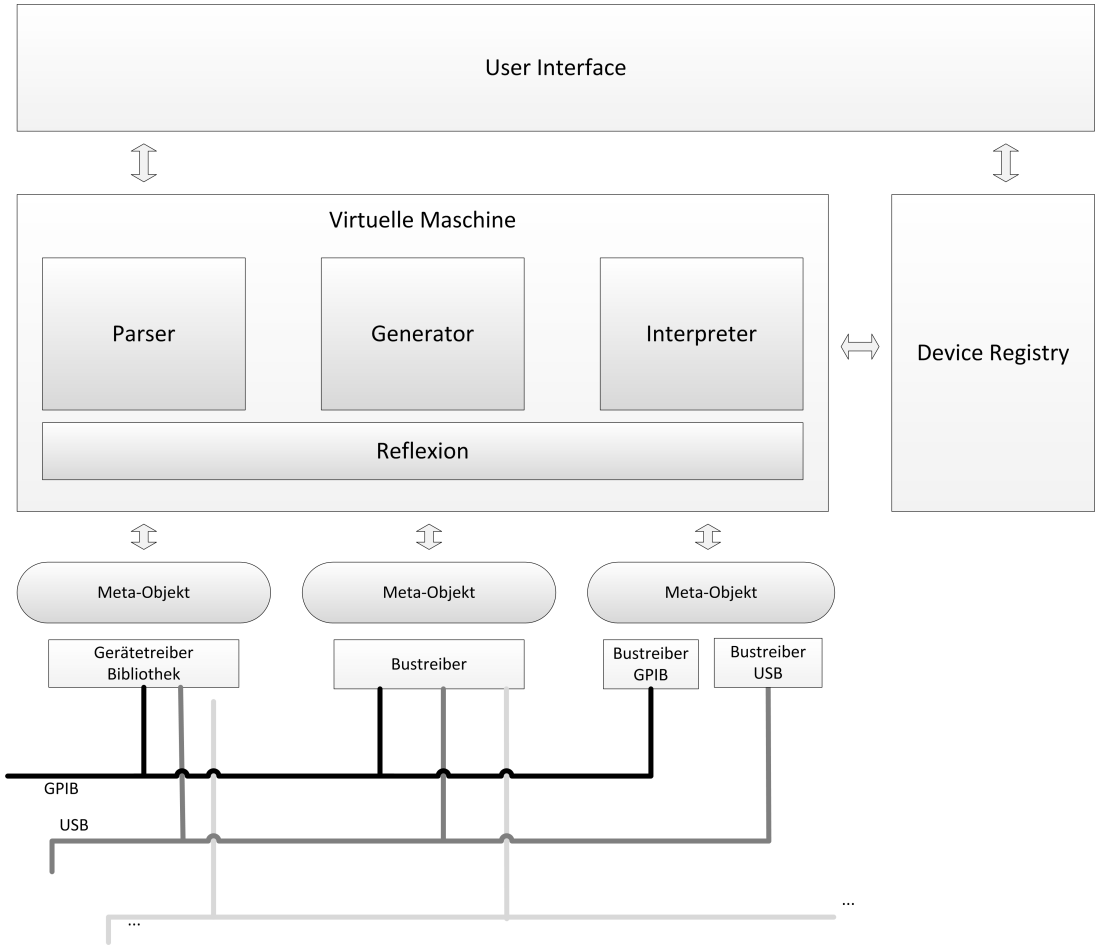


Abbildung 5.3 – Architektur des Task++-Grundsystems.

Auf Grund der vielfältigen Einsatzbereiche von virtuellen Maschinen gibt es kaum einheitliche Konzepte über deren Architektur [50]. Für die Ziele dieser Arbeit wurde daher hauptsächlich Wert auf Einfachheit gelegt. Im Wesentlichen besteht die Task++-*Virtuelle-Maschine* aus 3 Teilen: einem Parser für den Aufbau eines *Abstrakten Syntaxbaumes*, einem Generator für die Erzeugung von Maschinencode sowie einem Interpreter, der den erzeugten Code ausführt.

5.3.1 Parser

Der Parser ist Bestandteil der *Virtuellen Maschine* vom Task++-System und erhält als Eingabe eine Darstellung des Messablaufs in Form einer dynamisch typisierten Skriptsprache. Der Parser wurde mit Hilfe von `Spirit` [51], einer `Boost`-Bibliothek [52], erstellt. `Spirit` ist ein *Parsergenerator-Framework*, welches vollständig in C++ geschrieben ist und sich somit sehr gut in ein in C++ entwickeltes Programm einbinden lässt. Durch Operatorenüberladung kann eine Syntax verwendet werden, die der *Erweiterten Backus Naur Form* sehr ähnlich ist. Das Ergebnis des Parsers

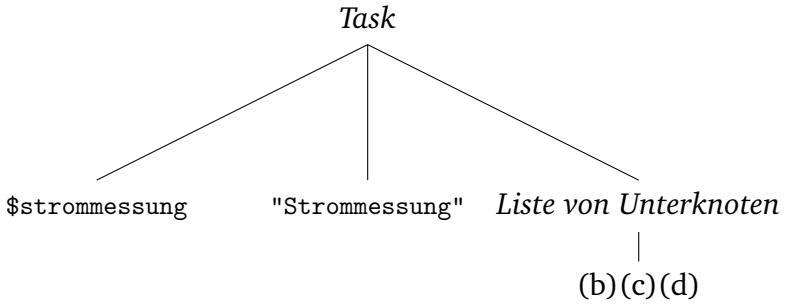
ist eine Darstellung des gesamten Messprogramms als *Abstrakter Syntaxbaum*, welcher in weiterer Folge von einem Codegenerator in eine für die Task++-*Virtuelle-Maschine* ausführbare Repräsentation umgewandelt wird. Folgend sei nochmals das Beispiel aus Listing 5.3 in erweiterter Form gegeben:

Listing 5.5 – Task für die Strommessung am Kanal 101 eines *Agilent-34970A*-Gerätes.

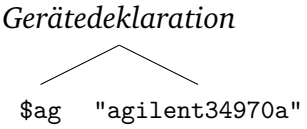
```
1 task $strommessung("Strommessung") {
2   device $ag("agilent34970a");
3   $result = $ag.measure_current_dc(101);
4   print("Ergebnis: " + $result);
5 }
```

Die Ausgabe des Parsers ist ein *Abstrakter Syntaxbaum*, der wie folgt aussieht:

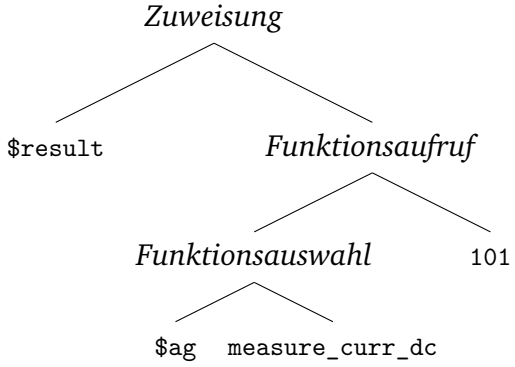
(5.3.1) (a)



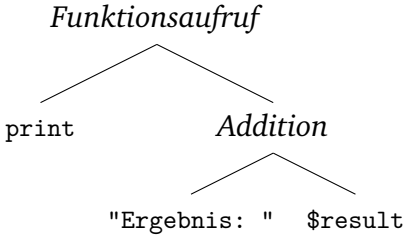
(b)



(c)



(d)



Der Baum in (a) stellt die Struktur eines Tasks dar. Der Task kann über die Variable `$strommessung` angesprochen werden, hat einen Namen sowie eine Liste von Unterknoten; diese bilden die Unterbäume aus (b) (c) und (d). In Baum (b) wird ein Gerät des Typs `agilent34970a` instanziiert und dieses über die Variable `$ag` zugänglich gemacht. Baum (c) beschreibt die Zuweisung des Rückgabewertes einer Gerätefunktion an die Variable `$result`. Die Funktion wird über die Instanz `$ag`, welche mit dem Gerätemodul verknüpft ist, aufgefunden und mit dem Parameterwert `101` aufgerufen. In (d) wird die Zeichenkonstante `"Ergebnis: "` mit dem Wert der Variablen `$result` addiert und als Parameter der `print`-Funktion übergeben, welche den Wert auf die Konsole schreibt.

Listing 5.6 – Struktur eines Knotens des *Abstrakten Syntaxbaumes*.

```
struct ast_node {
  typedef boost::variant<
        , boost::recursive_wrapper<ast_list_t>
        , boost::recursive_wrapper<stmt_node>
        , boost::recursive_wrapper<id_node>
        , boost::recursive_wrapper<constant_node>
        , boost::recursive_wrapper<binary_op>
        , boost::recursive_wrapper<unary_op>
        , boost::recursive_wrapper<ast_node>
      >
  type;

  ast_node() : expr(nil()) {}

  template <typename Expr>
  ast_node(Expr const& expr)
    : expr(expr) {}

  type expr;
};
```

Die Knoten im *Abstrakten Syntaxbaum* werden mit Hilfe einer varianten Struktur `boost::variant` (siehe Listing 5.6) dargestellt. Diese wird von der `Boost Library` [52] bereitgestellt und erlaubt die Aufnahme von Werten unterschiedlichen Datentyps. Die Typen der verschiedenen Knoten (siehe Tabelle 5.1) werden als Templateparameter übergeben und zusätzlich von `boost:recursive_wrapper<>` eingeschlossen. Dieser Mechanismus ist Notwendig, um zirkuläre Abhängigkeiten aufzulösen. Der vom Parser aufgebaute Syntaxbaum wird dem Generator übergeben, welcher für die Erzeugung einer für die Task++-*Virtuelle-Maschine* ausführbaren Darstellung zuständig ist.

Tabelle 5.1 – Typen der Knoten im *Abstrakten Syntaxbaum* des Task++-Systems.

Typ	Beschreibung
`ast_list_t`	Dieser Knotentyp kann eine weitere Liste von (Unter-) Knoten aufnehmen.
`stmt_node`	Ein *Statement*-Knoten stellt eine Anweisung dar (*If-Statement*, *For*-Schleife usw.).
`constant_node`	Konstanten (Zeichenketten und numerische Werte) werden in diesem Knotentyp gespeichert.
`id_node`	Dieser Knoten beinhaltet den Namen einer Funktion oder einer Variablen.
`binary_op`	Dieser Knoten stellt eine binäre Operation dar (Addition, Division usw.).
`unary_op`	Dieser Knoten stellt eine unäre Operation dar (z. B. Negation).
`ast_node`	Generischer Knoten, welcher die anderen Knotentypen aufnehmen kann.

5.3.2 Generator

Der Generator erhält als Eingabe den *Abstrakten Syntaxbaum* des Messprogramms und hat als Aufgabe die Erzeugung von Maschinencode, welcher von dem im nachfolgenden Abschnitt beschriebenen Interpreter ausgeführt wird.
Mit Hilfe eines *Besucher*objektes *(Visitor Object)* [36] kann der Generator jeden Knoten besuchen und den Baum nach Bedarf mehrere Male durchlaufen, um entsprechenden Maschinencode zu erzeugen. Damit eine Typüberprüfung zur Komilierzeit möglich ist, leitet das *Besucher*objekt von `boost::static_visitor<>` ab. Es muss also für jeden Typ im Knoten ein *Besucher*mechanismus implementiert sein, damit die Prüfung erfolgreich bestanden wird.

Listing 5.7 – Codegenerierung für das Laden einer Konstante vom Speicher.

```
1  #define CODE(x) taskpp.code.push_back(x);
2  #define VISIT(x)  boost::apply_visitor(*this, x);
3  #define CODEPOS() taskpp.code.size();
4
5  void operator()(constant_node const& expr) const {
6    address_t addr = taskpp.mem->add(expr);
7    CODE(op_load)
8    CODE(addr)
9  }
10
11 void operator()(ast_node const& ast) const {
12   VISIT(ast.expr);
13 }
```

Tabelle 5.2 – Instruktionsset der *Virtuellen Maschine* von Task++.

Arithmetik	Logische Operatoren	Speicherzugriff	Programmfluss
op_add	op_or	op_store	op_jmpz
op_mul	op_and	op_load	op_jmp
op_div	op_ge	op_assign	op_dot
op_sub	op_eq	op_addr	op_call
op_mod	op_ne		op_halt
op_dec	op_gt		
op_inc	op_lt		
	op_le		

Listing 5.7 zeigt einen Ausschnitt des *Besucher*objektes für die Typen `constant_node` und `ast_node`. In Zeile 6 wird die Konstante `expr` in den Speicher gelegt und ihre Speicheradresse zurückgegeben. In Zeile 7 und 8 wird Code erzeugt, welcher den Speicherinhalt auf den Stack legt. In Zeile 12 wird der *Besucher*mechanismus auf seinen Unterknoten aufgerufen. Der Einfachheit halber werden 3 Makros (`CODE`, `VISIT`, `CODEPOS`) definiert, die selbsterklärend sind.

5.3.3 Interpreter

Der vom Generator produzierte Programmcode wird von einer stackbasierten *Virtuellen Maschine* entgegengenommen und abgearbeitet. Durch eine dynamische Typisierung sowie durch die Verwendung von varianten Stackelementen ist es möglich, mit einem Set von 24 Instruktionen auszukommen. Diese sind wie in Tabelle 5.2 dargestellt in 4 Gruppen aufgeteilt.

Arithmetik: Die arithmetischen Instruktionen nehmen 2 Elemente vom Stack, führen die Operation durch und legen das Ergebnis wieder auf den Stack zurück. Die Instruktion ist typenunabhängig, da die Stackelemente vom varianten Typ `mem_item` sind. Ein *Memory-Item* ist definiert durch

```
boost::variant<constant_node, address, objectPtr>
```

und kann somit eine Konstante, eine Adresse oder ein Zeiger auf ein Objekt sein.

Logik: Die logischen Operatoren nehmen 2 Elemente vom Stack und legen einen Wahrheitswert auf den Stack zurück. Dabei ist logisch *falsch* als 0 definiert; alles ungleich 0 ist logisch *wahr*.

Speicherzugriff: Die beiden Instruktionen `op_store` und `op_load` speichern bzw. laden ein Element vom Stack in den Speicher bzw. vom Speicher auf den Stack. Die Speicheradresse befindet sich im Programmcode unmittelbar nach dem Befehlscode der Instruktion. Die Instruktion `op_addr` legt eine Adresse auf den Stack. Die Instruktion `op_assign` speichert das oberste Stackelement (TOS[4]) in den Speicher mit der Adresse, welche sich als zweites Element auf den Stack befindet (TOS−1).

Programmfluss: Die Instruktionen `op_jmpz` und `op_jmp` stellen einen bedingten bzw. unbedingten Sprung im Programmcode dar. Sprungziel ist die (relative) Adresse, welche sich im Programmcode unmittelbar nach dem Befehlscode der Instruktion befindet. Im Falle von `op_jmpz` wird nur dann an die Programmstelle gesprungen, wenn das oberste Stackelement gleich 0 ist.
Die Instruktion `op_dot` ist der Selektor einer Funktion und erwartet einen Funktionsnamen als oberstes Stackelement sowie die Speicheradresse eines Objektes als darunterliegendes Element. Das Objekt wird für den Funktionsaufruf vorbereitet. Durch die Instruktion `op_call` wird die Funktion aufgerufen. Das oberste Stackelement gibt die Anzahl der Parameter an, welche sich auf den Stack über dem Objekt befinden, welches die Funktion beinhaltet. Die Instruktion `op_halt` beendet das Programm.

Abbildung 5.4 zeigt den Maschinencode für das Programm aus Listing 5.5 sowie den Speicherzustand nach der Abarbeitung von `op_load` am Programmzähler. Die zwei obersten Stackelemente sind ein Zeiger auf ein Objekt (TOS-1) sowie ein Zeiger auf eine Zeichenkette, welche die Funktion des Objektes (in diesem Fall eine Gerätefunktion) darstellt. Die darauffolgende Instruktion ist `op_dot`, welche auf die obersten zwei Stackelemente operiert und das Objekt für den Funktionsaufruf vorbereitet. Abbildung 5.5 zeigt den Speicherzustand kurz vor dem Funktionsaufruf durch die Instruktion `op_call`. Das oberste Stackelement (TOS) enthält die Anzahl der darunterliegenden Parameter, die über dem Objektzeiger liegen. In diesem Fall wird die Funktion mit genau einem Parameter (`101`) aufgerufen. Der Rückgabewert des Funktionsaufrufes wird auf den Stack gelegt und befindet sich genau über der Speicheradresse, die auf den Speicherbereich für die Variable `$result` zeigt. Durch `op_assign` wird der Rückgabewert der Funktion in den Speicherbereich der Variablen geschrieben und der Stack ist wieder leer.

[4] Top of stack.

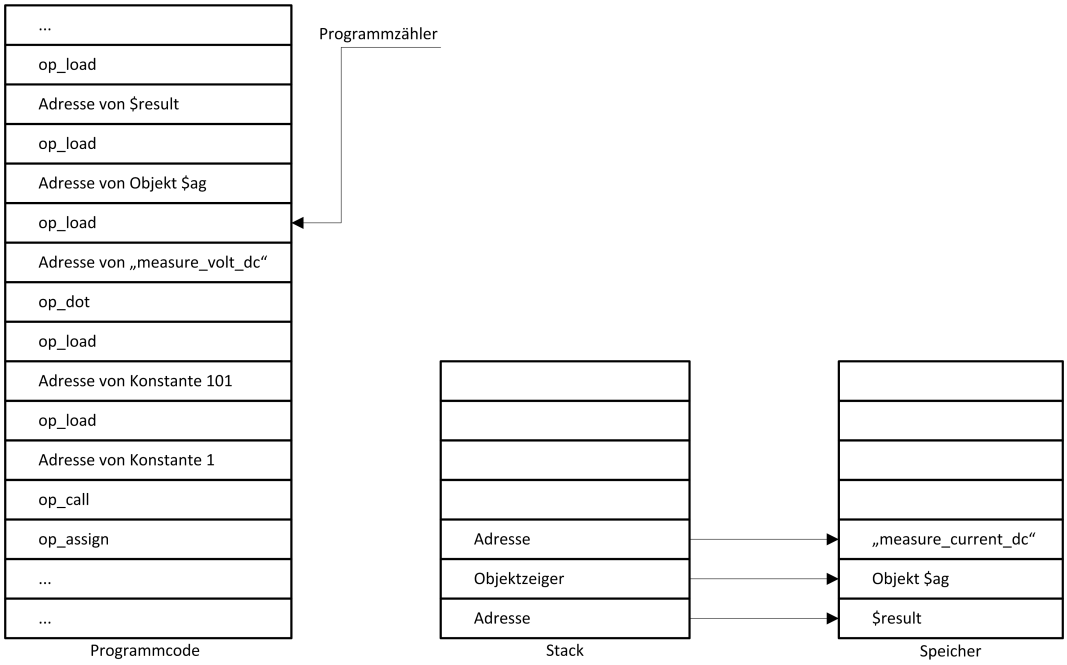


Abbildung 5.4 – Speicherzustand für das Programm aus Listing 5.5 nach der Ausführung des Befehlscodes `op_load`.

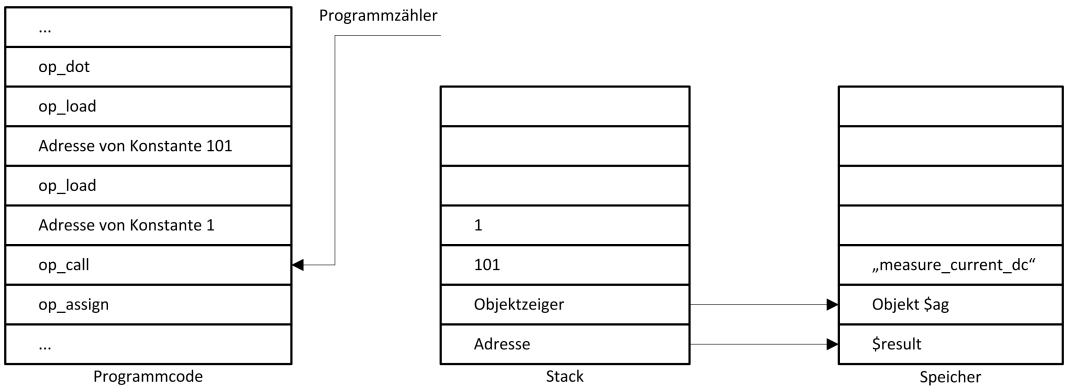


Abbildung 5.5 – Speicherzustand für das Programm aus Listing 5.5 vor der Ausführung des Befehlscodes `op_call`.

Listing 5.8 – Aufruf einer Gerätefunktion über den Reflexionsmechanismus.

```
1  Registry *reg = Registry::Instance();
2  Object *dev = reg->getObject(name);
3  Function *f = dev->getFunction(funcName, attributeCount);
4  const FuncDef *def = static_cast<const FuncDef*>
5                       (f->instanceOf());
6  FuncDef::AttributeIterator it = def->attrBegin();
7
8  while (it != def->attrEnd()) {
9    arg = boost::get<constant_node>(args[i].expr);
10   attr = (*it);
11   typeId = attr->getType().getType();
12
13   switch(typeId) {
14     case Type::intT:
15       // try to convert mem_item to an integer
16       intArg = boost::apply_visitor(c_int_visitor(), arg.expr);
17       f->setValue(attr->getName(), RealValue<int>(intArg));
18       break;
19     // other cases
20   }
21 }
22 Value retVal = f->call();
```

Listing 5.8 zeigt die Implementierung für den reflektiven Aufruf einer Funktion inklusive Umwandlung des Parameters in einen *Integer*-Wert über den *Besucher*mechanismus. In Zeile 1 wird auf eine *Singleton* Instanz der *Registry* zugegriffen, mit deren Hilfe in Zeile 2 das Objekt (bzw. Gerät) durch Angabe dessen Namen gefunden werden kann. In Zeile 3 wird versucht, die Funktion über das Objekt durch Angabe von Funktionsnamen und Anzahl der Parameter aufzufinden. In Zeile 4 wird über die Funktion das *Meta-Objekt* abgefragt, welches Auskunft über die formalen Parameter und deren Typen gibt. Die *While*-Schleife ab Zeile 8 iteriert über die formalen Parameter und in Zeile 16 wird versucht, das Argument in den zu erwarteten Typ (falls notwendig) umzuwandeln. Hier wird nur die Umwandlung in einen *Integer*-Wert dargestellt. Falls die Umwandlung geklappt hat, so wird in Zeile 17 der Wert eines Attributes durch Angabe seines Namens in der Funktion gesetzt. Sind alle Parameter gesetzt, so wird in Zeile 22 die Funktion aufgerufen.

Dieser Mechanismus verzögert die Typüberprüfung des Inhalts von Variablen solange, bis die Umwandlung erfolgen muss. Das Argument des Aufrufs der Funktion

```
$ag.measure_curr_dc(101);
```

könnte also genauso als Zeichenkette der Form `"101"` oder als Gleitkommazahl `101.00` angegeben werden, da beides in einen ganzzahligen Wert konvertiert werden kann. Da die meisten Geräte eine Zeichenkette als Antwort liefen, welche in weiterer Folge als Parameter für eine Funktion eines anderen Gerätes dienen kann,

muss man sich nicht um die Umwandlung des Typs kümmern. Angenommen man wollte den Ausgang eines Kanals auf einen Wert setzen, der von einem anderen Kanal gelesen und als Zeichenkette zurückgegeben wurde, so kann man folgendes schreiben:

```
$volt = $ag.measure_curr_dc(101);
$ag.source_volt_dc(104, $volt);
```

bzw.

```
$ag.source_volt_dc(104, $ag.measure_curr_dc(101));
```

5.4 Verwendung von Task++

Das Task++-System kann auf zwei verschiedene Arten verwendet werden. Zum einen ist es als reiner Interpreter einsetzbar, indem es ein Programm in Form einer Skriptsprache als Eingabe erhält und verarbeitet. Zum anderen ist es möglich, mit Hilfe einer Bedienoberfläche – auf visueller Basis – Messroutinen zusammenzustellen und auszuführen. Die Verwendung von Task++ als Interpreter bzw. die textuelle Programmierung von Messroutinen wird im folgenden Abschnitt beschrieben. Die Bedienoberfläche für die Komposition von Messabläufen wird automatisch gestartet, wenn das das System kein Programm als Eingabe erhält und wird im darauffolgenden Abschnitt vorgestellt.

5.4.1 Textbasierte Programmierung

Ein textbasiertes Task++-Programm besteht aus zwei Teilen:

- Programmcode zur Generierung von GUI (Eingabe-) Elementen,
- Programmcode zur Steuerung des Messablaufs.

Der Teil für die Generierung von Eingabeelementen ist nicht zwingend erforderlich, dient aber für eine bequeme Parametrisierung des Messablaufs. Zur besseren Strukturierung und Identifizierung eines Tasks werden die beiden Teile von einer Taskdeklaration umschlossen. Diese besteht aus einem einleitenden Keyword `task`, einer Variable, die den Task identifiziert sowie einer Zeichenkette, die den Task kurz beschreibt:

Listing 5.9 – Grundstruktur eines Tasks.

```
task $einTask("Eine kurze Beschreibung") {
  // Generierung von Eingabeelementen ...
  // Steuerung des Messablaufs ...
}
```

Für eine bessere Verständlichkeit wird zuerst die Erstellung von Programmcode zur Steuerung des Messablaufs beschrieben.

Steuerung des Messablaufs

Die Syntax von Task++ ähnelt jener von C und PHP. Variablen werden wie in PHP durch ein vorangestelltes $ Zeichen identifiziert und sind dynamisch typisiert. Erst wenn der Inhalt einer Variable von einer (Geräte-) Funktion gebraucht wird, erfolgt eine Prüfung, ob der Wert in den für die Gerätefunktion notwendigen Typ umgewandelt werden kann (siehe Listing 5.8).

Ein wesentliches Merkmal von Task++ ist die Deklaration von Geräte-Objekten. Hierbei wird das Gerätemodul samt seiner Funktionen an die Variable gebunden. Durch diesen Mechanismus ist es möglich, durch eine einheitliche Syntax, auf Funktionen unterschiedlicher Geräte zuzugreifen. Überlicherweise kommen nur selten mehrere Messgeräte des gleichen Typs in einem System zum Einsatz. Ist dies der Fall, so müssen mehrere Module mit unterschiedlichem Namen exportiert werden, da der Prototyp des Task++-Systems nicht mit verschiedenen Geräteinstanzen des gleichen Typs umgehen kann. Für modulinterne *Member*-Variablen wurde kein Reflexionsmechanimus bereitgestellt; dies würde die Erstellung von Modulen erheblich verkomplizieren.

```
device $ag("agilent34970a");
```

Mit diesem *Statement* wird ein Gerät vom Typ `agilent34970a` instanziiert, auf welches in weiterer Folge im Programmcode durch den angegebenen Variablennamen `$ag` zugegriffen werden kann. Gerätefunktionen des instanziierten Typs können durch Anhängen des Funktionsnamens, getrennt von einem Punkt, aufgerufen werden. Zur Kontrollflusssteuerung des Programms dienen Schleifen und Anweisungen, wie man sie aus gewöhnlichen prozeduralen Programmiersprachen kennt. Will man beispielsweise alle ungeraden Kanäle einer Multiplexerkarte des Geräts vom Typ `agilent34970a` schließen, so bietet sich folgendes Skript an:

Listing 5.10 – Schließen aller ungeraden Kanäle einer Multiplexer-Karte des *Agilent-34970A*-Gerätes.

```
task $mux_close_odd("Schliesse ungerade Kanaele auf MUX") {
  device $ag("agilent34970a");
  for ($i = 201; $i <= 220; $i++) {
    if ($i % 2 == 0) {
      $ag.route_close($i);
    }
    else {
      $ag.route_open($i);
    }
  }
}
```

Voraussetzung hierfür ist natürlich, dass das Gerät *Agilent 34970A* und die Funktionen `route_close` sowie `route_open` zuvor dem System bekannt gemacht wurden. Unter der Annahme, dass sich dieses Programm in der Datei `programm.tpp` befindet, kann es wie folgt aufgerufen werden:

```
task++.exe programm.tpp
```

Die Referenz von Task++ befindet sich im Anhang A.4.

Generierung von GUI (Eingabe-) Elementen

Zur Parametrisierung der Tasks bietet Task++ die Möglichkeit an, Eingabemasken zu beschreiben, die bei der Ausführung bzw. beim Aufruf des Programms automatisch generiert werden. Für die Darstellung von grafischen Oberflächen, samt der beschriebenen Eingabeelemente, wird das *Qt-Framework* [53] verwendet. Die Deklaration einer Eingabemaske wird durch das Keyword `gui` eingeleitet und in geschwungenen Klammern – `{` und `}` – eingeschlossen.
Eingabefelder werden durch das Keyword `parameter` deklariert und können mit Hilfe der Keywords `group` und `tab` gruppiert bzw. getrennt werden, um eine bessere Darstellung zu ermöglichen. Die Keywords `group` und `tab` werden, analog zum `gui` Keyword, von einer öffnenden `{` Klammer, weiteren Elementen, und einer schließenden `}` Klammer umgeben. Diese können zwar beliebig geschachtelt werden, jedoch ist hier auf einen sinnvollen Einsatz zu achten.
Innerhalb jedes der eben beschriebenen Keywords können Eingabefelder für Parameter deklariert werden. Die Deklaration eines Parametereingabefeldes geschieht wie folgt:

```
parameter($var, "Parametername", Defaultwert) is TYPE;
```

Dieses Programmfragment erzeugt ein einfaches Eingabefeld mit der Beschriftung «Parametername» und setzt einen angegebenen Defaultwert. Der Wert des Eingabefeldes kann in späterer Folge mit Hilfe der angegebenen Variable `$var` referenziert bzw. verwendet werden. `TYPE` steht in diesem Fall für den Typ der eingegeben werden soll bzw. erwartet wird. Dies unterstützt eine vorzeitige Prüfung der Eingaben. Wird als `TYPE` z. B. `int` angegeben und eine Zeichenkette `"abc"` eingegeben, so wird der Parameter nicht angenommen, sondern eine Fehlermeldung ausgegeben. `TYPE` kann einen der Werte aus `int`, `double` und `string` annehmen, wobei hierfür die Angabe zusätzlicher Eingabeschranken möglich ist. Will man z. B. den Bereich eines ganzzahligen Wertes auf 0 bis 100 einschränken, so ergibt sich folgende Deklarationsmöglichkeit:

```
parameter($var, "Prozentangabe", 50) is int[0,100];
```

Werte außerhalb der zulässigen Grenze werden nicht angenommen und die Voreinstellung liegt bei 50. Wird hingegen die Schranke auf einen `string` Typ angegeben, so bezieht sich der Wert auf die Länge der eingegebenen Zeichenkette.

Das Eingabeelement kann ebenfalls geändert werden, um dem Benutzer die Bedienung zu erleichtern. Die Prozenteingabe aus vorigem Beispiel könnte ebenfalls durch einen Schieberegler (`slider`) gesteuert werden. Dadurch erweitert sich die Parameterdeklaration folgendermaßen:

```
parameter($var, "Prozentangabe", 50) is int[0,100] as slider;
```

Diese Deklaration erzeugt einen Schieberegler mit den Werten 0 und 100 an seinen Extremitäten. Wird dem Keyword `slider` noch ein Argument in der Form `slider(5)` angehängt, so wird die Schrittweite bzw. Granularität auf 5 gesetzt.

Will man jedoch nur die Auswahl bestimmter Prozentangaben ermöglichen, so ist dies über eine *Dropdown*-Liste möglich, welche nur die entsprechenden Werte als *Label-Value*-Paar enthält:

```
parameter($var4, "Prozent 4", 40)
  is string {"10"=>10, "40"=>40, "80"=>80, "100"=>100}
  as combo;
```

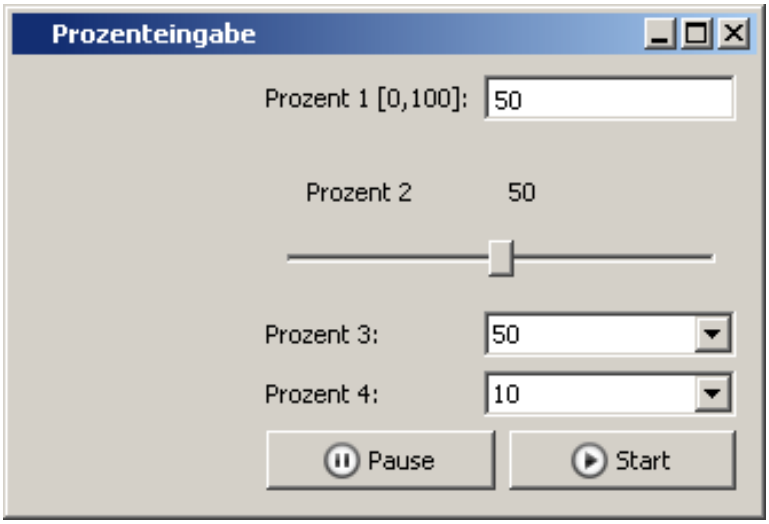


Abbildung 5.6 – Automatisch generierte Eingabemaske für Prozentwerte durch die Deklarationen aus Listing 5.11.

Zusammengefasst sind für die Prozenteingabe beispielsweise folgende Möglichkeiten gegeben:

Listing 5.11 – Deklaration verschiedener Eingabefelder für Prozentwerte.

```
task $prozente("Prozenteingabe") {
  gui {
    parameter($var1, "Prozent 1", 50) is int[0,100];
    parameter($var2, "Prozent 2", 50) is int[0,100] as slider(5);
    parameter($var3, "Prozent 3", 50) is int[0,100] as combo;
    parameter($var4, "Prozent 4", 40)
      is string {"10"=>10, "40"=>40, "80"=>80, "100"=>100}
      as combo;
  }
}
```

Abbildung 5.6 zeigt die von Listing 5.11 erzeugte Eingabemaske.

Task++-Systemfunktionen

Zusätzlich zu den Gerätemodulen, welche die physikalischen (Mess-) Geräte repräsentieren, enthält Task++ ein virtuelles Gerät, welches dem Programmierer noch einige Systemfunktionen zur Verfügung stellt. Diese können je nach Bedarf beliebig erweitert werden und sind in Tabelle 5.3 aufgelistet.

5.4.2 Visuelle Programmierung

Für die visuelle Programmierung steht ein *User-Interface* zur Verfügung, welches automatisch gestartet wird, wenn dem System kein Programm als Parameter übergeben wird. Dadurch hat der Anwender die Möglichkeit – auf visueller Basis –

Tabelle 5.3 – Task++-Systemfunktionen.

Funktion	Beschreibung
`wait($millisekunden)`	Unterbricht den Programmfluss und wartet eine Anzahl an übergebenen Millisekunden.
`print($daten)`	Schreibt eine Variable, einen Text oder einen Ausdruck auf die Konsole.
`fprint($dateiname, $daten)`	Schreibt eine Variable, einen Text oder einen Ausdruck in eine Datei. Der Dateiname wird als erster Parameter übergeben. Existiert bereits eine Datei mit dem angegebenen Namen, so werden die Daten am Ende dieser Datei angehängt.
`now()`	Liefert das Datum und die aktuelle Uhrzeit im Format `DD.MM.YYYY HH:MM:SS`.
`timestamp()`	Liefert einen Zeitstempel im Format `YYYYMMDDHHMMSS`.

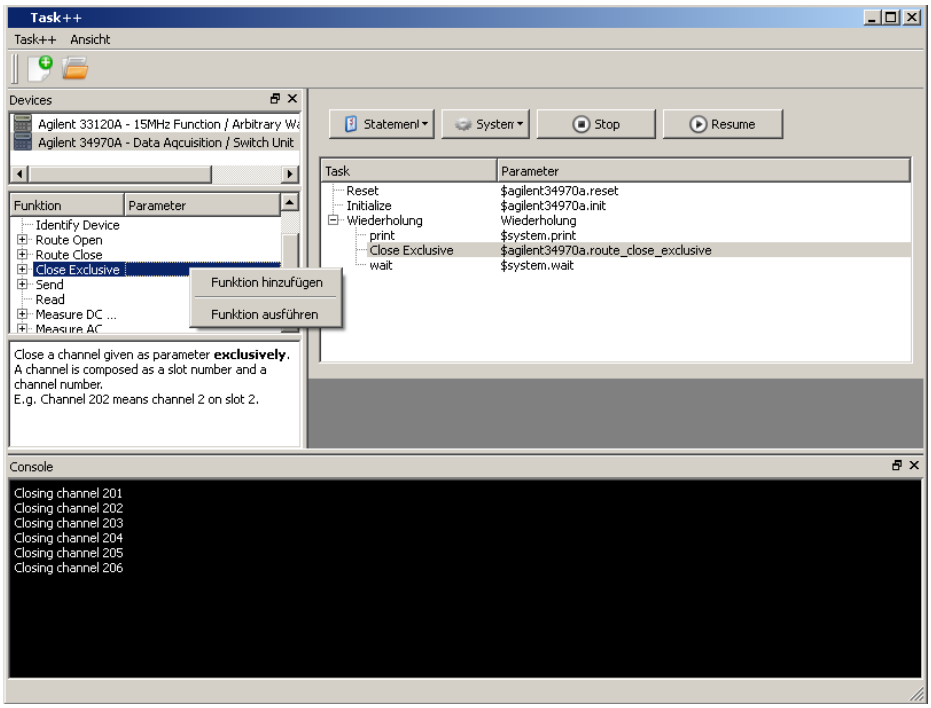


Abbildung 5.7 – Benutzeroberfläche für die Komposition von Messroutinen.

einfache Messaufgaben zusammenzustellen und auszuführen. Die Benutzeroberfläche zeigt die verfügbaren Geräte sowie deren Funktionen samt einer Beschreibung an, so wie in Abbildung 5.7 zu sehen ist. Das erste Fenster in der linken Spalte zeigt die Geräte, welche im System gerade verfügbar sind (jeweils eines der AGILENT TECHNOLOGIES Instrumente *33120A* und *34970A*). Durch Auswahl eines dieser Geräte wird im darunter liegenden Fenster eine Liste der vom ausgewählten Gerät bereitgestellten Funktionen angezeigt. Das wiederum darunterliegende Fenster enthält eine Funktionsbeschreibung, die angezeigt wird, sobald eine Gerätefunktion ausgewählt ist. Diese Information wird von den *Meta-Objekten* der Geräte bereitgestellt, welche von der Benutzerschnittstelle über die *Registry* ausgelesen werden. Das Hauptfenster (rechts) enthält den eigentlichen Messablauf, welcher als verschachtelte Baumstruktur dargestellt wird. Mittels *Drag-and-Drop* können sämtliche Gerätefunktionen in das Hauptfenster eingefügt und verschoben werden, sodass die Struktur dem gewünschten Messablauf entspricht. In manchen Situationen ist es praktisch, eine Gerätefunktion direkt auszuführen, ohne sie dafür in den Programmablauf einfügen zu müssen. Über das Kontextmenü, welches sich durch einen Rechtsklick mit der Maustaste auf die entsprechende Gerätefunktion öffnet, hat man diese Möglichkeit. Eventuell benötigte Parameter müssen vor Ausführung der Funktion gesetzt werden. Diese werden durch einen Klick auf das +-Symbol angezeigt. Kontrollflussanweisungen wie `if`, `for` und `while` können über den *Statement-Button*, welcher sich über dem Hauptfenster befindet, ausgewählt und eingefügt werden. Der *System-Button* enthält die Funktionen des virtuellen Gerätes, welche u. a. für die Ausgabe auf die Konsole (unterstes Fenster) bzw. in eine Datei verantwortlich sind. Auch das virtuelle Gerät kann somit ständig um Funktionen erweitert werden, ohne in den Kern des Systems eingreifen zu müssen.

Durch die visuelle Programmierung über die Benutzeroberfläche können noch nicht alle Möglichkeiten ausgeschöpft werden, die durch manuelle Programmierung möglich sind.

5.5 Gegenüberstellung

5.5.1 LabVIEW

Einer der wichtigsten Bestandteile von `LabVIEW` ist ein Compiler für die grafische Datenflusssprache G, mit welcher ein intuitives Verständnis auf die zugrundeliegende Hardware möglich ist [54]. Task++ erlaubt sowohl eine textuelle als auch eine visuelle Programmierung durch Strukturierung eines Baummodells mit Hilfe

von Tasks sowie Kontrollanweisungen auf diesen. Dadurch können kleine Messroutinen erstellt und zu einem späteren Zeitpunkt als Task wiederverwendet werden. Einen Task könnte man in gewisser Hinsicht mit einem *Virtuellen Instrument* in LabVIEW vergleichen. Ein *VI* enthält entweder einen Datenfluss in G oder weitere sub-*VI*s. Ein Task kann zu einer Einheit zusammengefasst und in weiterer Folge als parametrisierbares Ganzes wiederverwendet werden. Die Verschachtelung von Tasks samt Verknüpfung von Parametern wurde im Zeitrahmen dieser Arbeit nicht in voller Funktion implementiert.

5.5.2 NOVA

Ähnlich wie in Task++, lassen sich in NOVA komplexe Prozeduren durch die Komposition von Grundfunktionen und Basiselementen zusammenstellen. Frühere Versionen erlaubten lediglich das Verändern von Parametern vorgegebener Messroutinen. Die Struktur selbst konnte nicht verändert werden, sodass man auf Updates warten musste, bis neue Routinen im System verfügbar waren [33, S.7–9].

Werden in einem Setup ausschließlich Geräte von METROHM AUTOLAB verwendet, so ist NOVA die optimale Wahl. Die Einbindung von externen Geräten in NOVA ist nur eingeschränkt möglich. Mit einem SDK[5] ist es möglich, Geräte dieses Herstellers über das .NET Framework in Programmen wie LabVIEW einzubinden. Über die Programmierung von eigenen Modulen und deren Verwendung in NOVA ist im Tutorial für die Einbindung externer Geräte [47] keine Information verfügbar. Womöglich ist dies vom Hersteller auch nicht gewollt. Die Ähnlichkeit zu Task++ ist markant.

[5] Software Development Kit.

6 Experimente

> Alles sollte so einfach wie möglich gemacht werden aber nicht einfacher.
>
> *(Albert Einstein)*

6.1 Messung eines Arrays von Mikroelektroden

6.1.1 Übersicht

Im Rahmen der Dissertation von BERNHARD EGGER [55] wurde ein Array von 120 Doppelelektroden und 120 Ag/AgCl-Referenzelektroden entworfen. Es soll möglich sein, wesentliche Korrosionsparameter wie Leitfähigkeit, pH-Wert, Sauerstoff- und Chloridkonzentration zu messen. Mit konventionellen elektrochemischen Methoden werden nur Durchschnittswerte gemessen – ortsaufgelöste Information über den Korrosionsprozess ist dadurch nicht verfügbar. Vor allem Spaltkorrosion kann mit den herkömmlichen Methoden nicht lokalisiert werden. Auf Grund der Bauweise wie z. B. in Flanschverbindungen von Rohren oder in Autos können diese Spalte nicht vermieden werden. Eine Korrosion kann erst nach vollständiger Durchlöcherung beobachtet bzw. erkannt werden. Für die ortsaufgelöste Korrosionsmessung existieren bereits Techniken wie *Scanning Electrochemical Microscopy (SECM)* oder *Scanning Reference Electrode*. Diese Techniken sind jedoch für Korrosionstests in Testkammern (Salzsprühtests oder VDA[1] Tests) ungeeignet. Die Notwendigkeit einer anderen Technik führte zur Entwicklung eines Arrays von Mikroelektroden (siehe Abbildung 6.1), welches der korrosiven Atmosphäre standhalten würde. Das Multielektroden-Array besteht aus 120 Mikrozellen, welche in einem Abstand von 5 mm voneinander getrennt sind. Jede dieser Zellen besteht aus einer Ag/AgCl-Referenzelektrode mit einem Durchmesser von 250 µm sowie einer Zwillingselektrode von jeweils 50 µm breiten Platin-Scheiben, welche in Borosilikatglas versiegelt sind. Durch diese Vorgehensweise soll es möglich sein, eine bessere Messung von Spaltkorrosion durchzuführen.

[1] Verband der Deutschen Automobilindustrie.

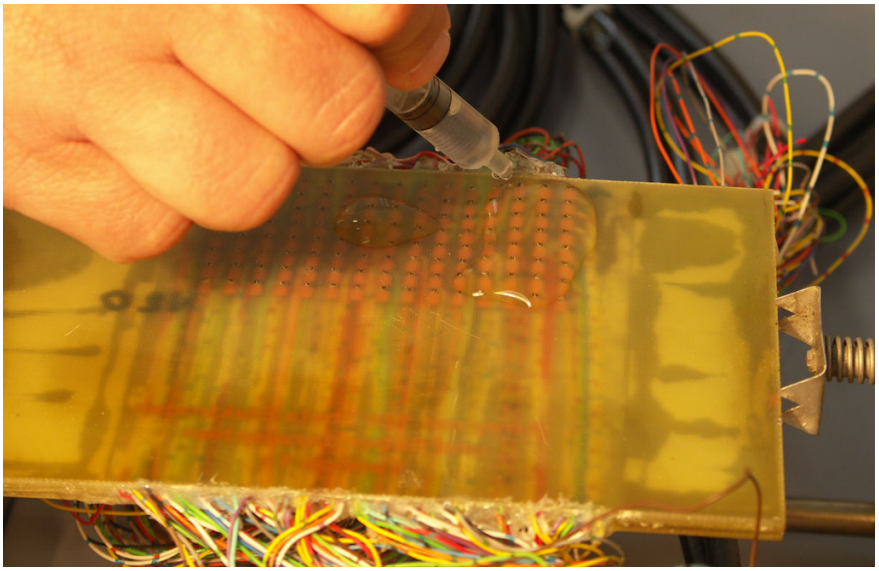


Abbildung 6.1 – Anfertigung eines Arrays von Mikroelektroden.

Die für den Korrosionsprozess relevanten Parameter sind:

- Leitfähigkeit
- pH-Wert
- Sauerstoffkonzentration
- Chloridkonzentration

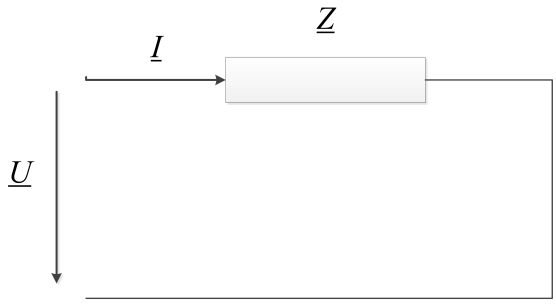
Mit diesem Array von Mikroelektroden können verschiedene Messungen wie z. B. elektrochemische Impedanzspektroskopie (EIS) oder Zyklovoltammetrie durchgeführt werden.

Impedanzspektroskopie

Bei der EIS werden die elektrischen Eigenschaften eines chemischen Systems untersucht. Die zu untersuchende Probe befindet sich dabei zwischen zwei Elektroden an denen ein sinusförmiges Spannungssignal angelegt wird. Der Frequenzbereich liegt dabei üblicherweise zwischen 10^{-4} und 10^{8} Hz. Die angelegte Wechselspannung lässt sich mit der Formel

$$U(t) = U_0 \sin(\omega t + \phi_U) \tag{6.1}$$

beschreiben, wobei U_0 die Amplitude, ω die Kreisfrequenz und ϕ_U den Phasenwinkel festlegt.



$$\underline{Z}(j\omega) = \frac{\underline{U}(j\omega)}{\underline{I}(j\omega)}$$

Abbildung 6.2 – Setup für eine Impedanzmessung.

Die Antwort des Systems ist ein Strom mit der gleichen Frequenz, dessen Amplitude I_0 und Phase ϕ_I vom betrachteten System abhängt.

$$I(t) = I_0 \sin(\omega t + \phi_I) \tag{6.2}$$

Die Impedanz $\underline{Z}$ ergibt sich aus dem Verhältnis der angelegten Spannung $\underline{U}$ zu dem gemessenen Strom $\underline{I}$ (Abbildung 6.2).

Zyklovoltammetrie

Durch eine Zyklovoltammetrie (auch Dreieckspannungsmethode genannt) ist es möglich, ein Bild über die elektrochemischen Abläufe an einer Elektrode in einer Lösung zu erhalten. Dabei werden dreiecksförmige oder sägezahnförmige Spannungsrampen erzeugt und an der Arbeitselektrode angelegt. In wässrigen Lösungen werden die Grenzpotentiale auf die Sauerstoff- bzw. Wasserstoffentwicklung eingestellt. Die Bezugselektrode hilft dem Potentiostaten, die Spannung an der Arbeitselektrode genau zu bestimmen. Die Stromstärke wird in Abhängigkeit zum angelegten Potential gemessen; z. B. sind Strommaxima im Strom-Spannungsverlauf von Interesse. Abbildung 6.3 zeigt den typischen Potentialverlauf einer Zyklovoltammetrie mit konstanter Änderung über die Zeit.

6.1.2 Problematik

Für diese Messaufgabe kommen mehrere Geräte zum Einsatz. Diese sind zwar über den selben Bus (*GPIB*) miteinander verbunden, können jedoch nicht miteinander kommunizieren. Es ist also ein manueller Eingriff notwendig, um einen sequenziellen und koordinierten Ablauf der Messaufgabe zu realisieren. Die Kanalauswahl muss mit Hilfe eines Multiplexers manuell vorgenommen werden, was einige Se-

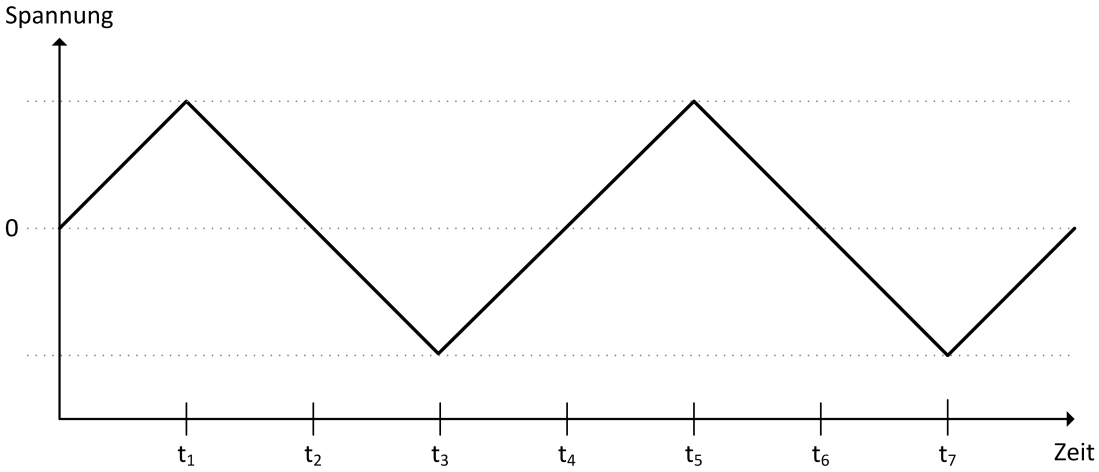


Abbildung 6.3 – Typischer Potentialverlauf einer Zyklovoltammetrie.

kunden in Anspruch nimmt. Durch natürliche Veränderungen, denen die Messumgebung ausgesetzt ist, wird das Ergebnis wegen der Trägheit im Gesamtablauf verzerrt. Beispielsweise verändert sich die Leitfähigkeit einer eventuell vorhandenen Elektrolytlösung durch Verdunsten des Wassers. Ein Messdurchgang dauert mit diesem Verfahren mehrere Stunden und muss permanent überwacht werden. Es wäre daher wünschenswert, einen automatisierten und synchronen Messablauf zu erzielen. Eine speziell für dieses Bedürfnis programmierte Software würde das Problem sofort in den Griff bekommen. Auch hier ist jedoch wieder ein hohes Maß an Flexibilität bzw. Parametrisierungsmöglichkeit notwendig. Zum einen müssen sämtliche Parameter für die Impedanzmessung am Messgerät einstellbar sein, zum anderen ist es auch notwendig, die Funktionalität eines Multiplexers flexibel steuern zu können. Man denke an den Ausfall einer Mikroelektrode im Array oder an den Austausch eines *Plug-In*-Moduls in einem Gerät. Es ist also von großer Bedeutung, jeden konfigurierbaren Parameter **aller Geräte**, aus dem **selben Programm** verändern zu können. Beispielsweise will man die Spannungsrampen der Zyklovoltammetrie in Abhängigkeit der Anzahl der Elektroden einstellen, um ein gewisses Zeitfenster einzuhalten.

6.1.3 Systemaufbau

Die Mikroelektroden (Abbildung 6.1) wurden mit einem digitalen Multimeter der Firma KEITHLEY INSTRUMENTS vom Typ *2750 Multimeter / Switch System* – Abbildung 6.4(a) – miteinander verbunden. Mit Hilfe von *Plug-In*-Modulen kann die Funktionalität des Gerätes erweitert werden. Das *Keithley 2750* wurde als Multiplexer verwendet, dessen Eingänge direkt mit den Mikroelektroden verbunden sind.

(a) (b)

Abbildung 6.4 – (a) Messgerät *Keithley 2750 – Multimeter / Switch System*.
(b) Messgerät *HP4192A – LF Impedance Analyzer*.
Quelle (a): `www.keithley.com`
Quelle (b): `www.used-line.com`

Abbildung 6.5 – Messgerät *Solartron 1286 – Potentiostat / Galvanostat.*

Für die **Impedanzmessung** wurden die Ausgänge des Multiplexers an das Impedanzmessgerät *HP4192A* – Abbildung 6.4(b) – von AGILENT TECHNOLOGIES bzw. HEWLETT-PACKARD[2] verbunden; für die **Zyklovoltammetrie** an das *Solartron 1286 – Potentiostat / Galvanostat* (Abbildung 6.5).

Alle Geräte werden am *General Purpose Interface Bus* angeschlossen. Die Schnittstellenbezeichnung des *HP4192A* Gerätes ist noch *HP-IB*, welche der Vorgänger der ursprünglichen *GPIB*-Version war (siehe Abschnitt 2.2.1) und daher mit dieser kompatibel ist.

6.1.4 Steuerung

Am Beispiel der Zyklovoltammetrie wird hier die Steuerung durch Task++ erläutert. Analog zur Impedanzmessung wird jede Elektrode im Array für eine Messung herangezogen. Die Auswahl der Elektroden erfolgt über den Multiplexer im *Keithley 2750*, dessen Kanäle mit dem *Solartron 1286* verbunden sind.

[2] AGILENT TECHNOLOGIES entstand 1999 als Ableger aus einigen Abteilungen von HEWLETT-PACKARD. Die Messgeräte-Sparte wurde 2014 ausgegliedert und heisst seither KEYSIGHT TECHNOLOGIES.

Listing 6.1 – Programmcode für die Generierung der Eingabemaske.

```
tab("Setup Ramp Sweep") {
  group("Rampe A") {
    parameter($va, "V[A]", 0) is double[-14.5, +14.5];
    parameter($ta, "T[A] ms", 0) is int[0, 60000];
  }
  group("Rampe B") {
    parameter($vb, "V[B]", 0) is double[-14.5, +14.5];
    parameter($tb, "T[B] ms", 0) is int[0, 60000];
  }
  group("Rampe C") {
    parameter($vc, "V[C]", 0) is double[-14.5, +14.5];
    parameter($tc, "T[C] ms", 0) is int[0, 60000];
  }
  group("Rampe D") {
    parameter($vd, "V[d]", 0) is double[-14.5, +14.5];
    parameter($td, "T[D] ms", 0) is int[0, 60000];
  }
  group("Delay & Segments") {
    parameter($solDelay, "Delay in ms", 1000) is int[0, 60000];
    parameter($segments, "Segments", 4) is int[2, 8];
  }
}
```

Listing 6.2 – Programmcode für die Ablaufsteuerung der Zyklovoltammetrie.

```
// Get devices
device $mux("keythley2750");
device $sol("solartron1286");

// Reset devices
$mux.reset();
$sol.reset();

// Setup Sweep
$sol.send("VA" + $va);
$sol.send("VB" + $vb);
$sol.send("VC" + $vc);
$sol.send("VD" + $vd);

$sol.send("TA" + $ta);
$sol.send("TB" + $tb);
$sol.send("TC" + $tc);
$sol.send("TD" + $td);

$sol.send("DL" + $solDelay);
$sol.send("SM" + $segments);

$filename = "zyklovoltammetrie_" + now();

for($i=1; $i<=$elektroden; $i+=1) {
  $mux.setChannel($i);
  wait($muxDelay);
  $data = $sol.cv();
  fprint($filename, $data);
}
```

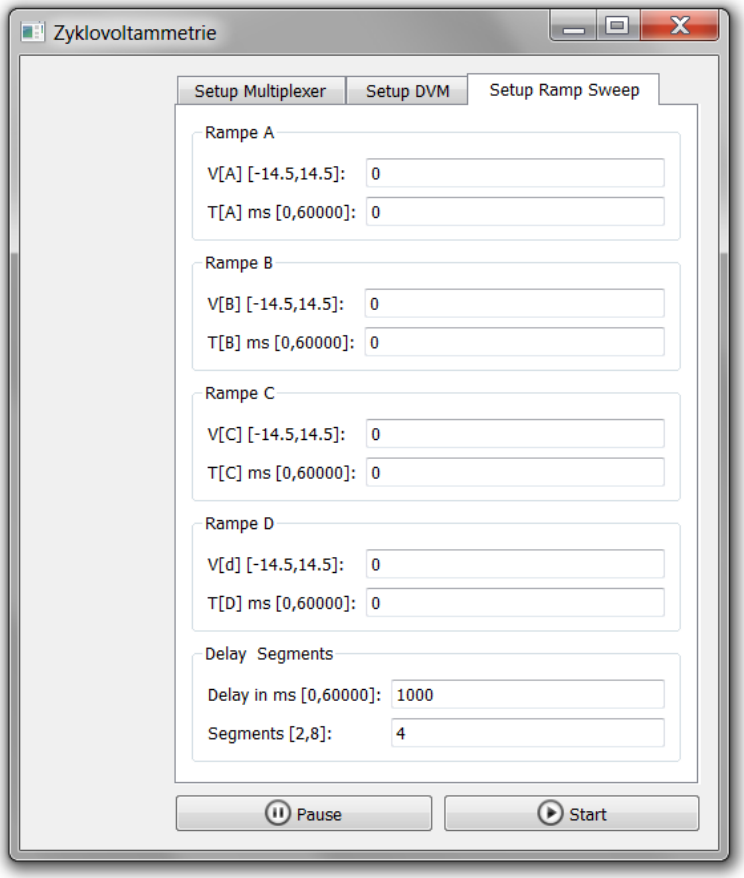
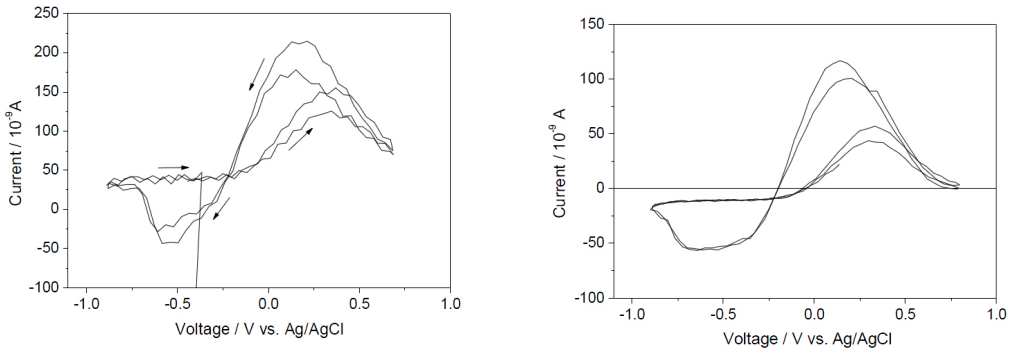


Abbildung 6.6 – Generierte Eingabemaske für die Parametriesierung der Zyklovoltammetrie.

Der erste Teil des Programms (Listing 6.1) beschreibt den Teil der Benutzerschnittstelle für die Eingabe der Parameter zur Konfiguration der Spannungsrampen. Die Eingabemaske wird vom Task++-System automatisch generiert und sieht wie in Abbildung 6.6 dargestellt aus.

Listing 6.2 zeigt den eigentlichen Programmablauf. In den Zeilen 2 und 3 werden zunächst die benötigten Geräte initialisiert. Beide Geräte sind am selben Bus (*GPIB*) angeschlossen und jeweils einer eindeutigen Kanalnummer zugeordnet. Jedem Gerät kann über eine Konfigurationsdatei eine Kanalnummer zugewiesen werden. Diese muss dann im Programmcode nicht mehr explizit angegeben werden, falls sie bereits der Einstellung am Gerät entspricht (siehe Anhang A.1). In den Zeilen 10–18 wird dem Potentiostaten die Rampenkonfiguration geschickt; in den Zeilen 20 und 21 die Delayeinstellung sowie die Anzahl der Segmente. Die Werte der Variablen werden, wie in Abschnitt 5.4 beschrieben, von der Benutzerschnittstelle übernommen. In Zeile 25 startet die Schleife über die Anzahl der Elektroden im Array. Zunächst wird der Multiplexer dazu veranlasst, den Kanal auf den aktu-



(a) (b)

Abbildung 6.7 – Zyklovoltammogramm von Zelle 88 am Tag 3 (links) und am Tag 5. *Quelle: [56]*

ellen Schleifenzähler zu setzen. Anschließend wird für eine gewisse «Einschwingungszeit» gewartet, bevor in Zeile 28 die Zyklovoltammetrie durchgeführt wird. Da die Zyklovoltammetrie einen etwas komplexeren Messvorgang darstellt, wurde diese in eine separate Funktion ausgelagert. Das Kommando `$sol.cv()` ruft die ausgelagerte Funktion auf, welche keine zusätzlichen Parameter benötigt, da der Potentiostat bereits entsprechend konfiguriert wurde. Der Rückgabewert wird der Variablen `$data` zugewiesen und in Zeile 29 mit der `fprint`-Funktion in eine Datei geschrieben. Der Dateiname wird in Zeile 23 festgelegt und mit Hilfe der *System*-Funktion `now()` mit einem Zeitstempel versehen.

6.1.5 Ergebnisse

Durch den automatisierten Ablauf mit Hilfe von Task++ konnte ein Messablauf in wesentlich kürzerer Zeit erledigt werden. Die natürlichen Veränderungen der Messumgebung wurden dadurch auf ein minimales Ausmaß reduziert. Das Resultat war ein zusammenhängendes und gut korreliertes Bild der chemischen Eigenschaften des Arrays. Die Ergebnisse dieses Experiments wurden in *Investigation of the corrosion mechanism in flanged joints of car bodies* [56] veröffentlicht. Abbildung 6.7 zeigt das aufgenommene Zyklovoltammogramm von Zelle 88 am Tag 3 – 6.7(a) – sowie am Tag 5 – 6.7(b). Die laterale Auflösung von 5 mm war für das Projekt ausreichend. Für eine bessere Auflösung des Arrays wäre eine andere Herstellungstechnik wie z. B. jene die für *Integrierte Schaltungen* verwendet wird, notwendig gewesen.

6.2 Simultanmessung elektrochemischer Zellen

6.2.1 Übersicht

Im Zuge eines Forschungsprojektes unter industrieller Beteiligung namhafter Partner wie VOESTALPINE, HENKEL KGAA und AUDI AG sollte das Korrosionsverhalten im Mischbau näher untersucht werden. Ziel war ein Verständnis der grundlegenden Korrosionsmechanismen von elektrolytisch verzinkten Stahl-Aluminium-Mischbauverbindungen, wie sie im Fahrzeugbau jüngst eingesetzt werden.

Galvanische Korrosion

Wird eine Elektrode in eine Elektrolytlösung eingetaucht, so entsteht auf Grund von Oxidations- bzw. Reduktionsreaktionen ein elektrochemisches Potential. Eine Kombination von Elektroden aus verschiedenen Materialien in einem gemeinsamen Eletrolyten wird als galvanisches Element bezeichnet. Da jedes Metall eine andere Fähigkeit besitzt, Elektronen abzugeben, bildet sich eine Potentialdifferenz zwischen diesen beiden Polen. Werden die Elektroden mit einem Elektronenleiter sowie einem Ionenleiter (Elektrolyt) miteinander verbunden, so wird der Kreis geschlossen und es fließt Strom. Dadurch ist es möglich, chemische Energie in elektrische Energie umzuwandeln. Der durch diesen Stromkreis ausgelöste Korrosionsvorgang wird als *Galvanische Korrosion* bezeichnet.

Messung der galvanischen Korrosion

Die Messung der galvanischen Korrosion erfolgt in einer Doppelzelle, wie in Abbildung 6.8 dargestellt ist. Damit sich die Elektrodenreaktionen nicht gegenseitig beeinflussen ist der Kathodenraum vom Anodenraum durch eine Fritte voneinander getrennt. Die Elektrode, an welcher Elektronen abgegeben werden, ist die Anode. An dieser werden durch Oxidation Elektronen freigegeben und wandern über den Leiter zur Kathode. Anionen wandern nun zur positiv geladenen Anode. Kationen wandern zur negativ geladenen Kathode, an welcher Reduktionsreaktionen stattfinden können.
Unter atmosphärischen Bedingungen sowie in neutralen Lösungen findet vorwiegend Sauerstoffkorrosion statt [28, S.242]. Zudem hängt die Korrosion von Metallen zusätzlich vom pH-Wert der gegenwärtigen Elektrolytlösung ab. Es wird daher von beiden Seiten (Kathoden- bzw. Anodenraum) reiner Sauerstoff bzw. Luft ein-

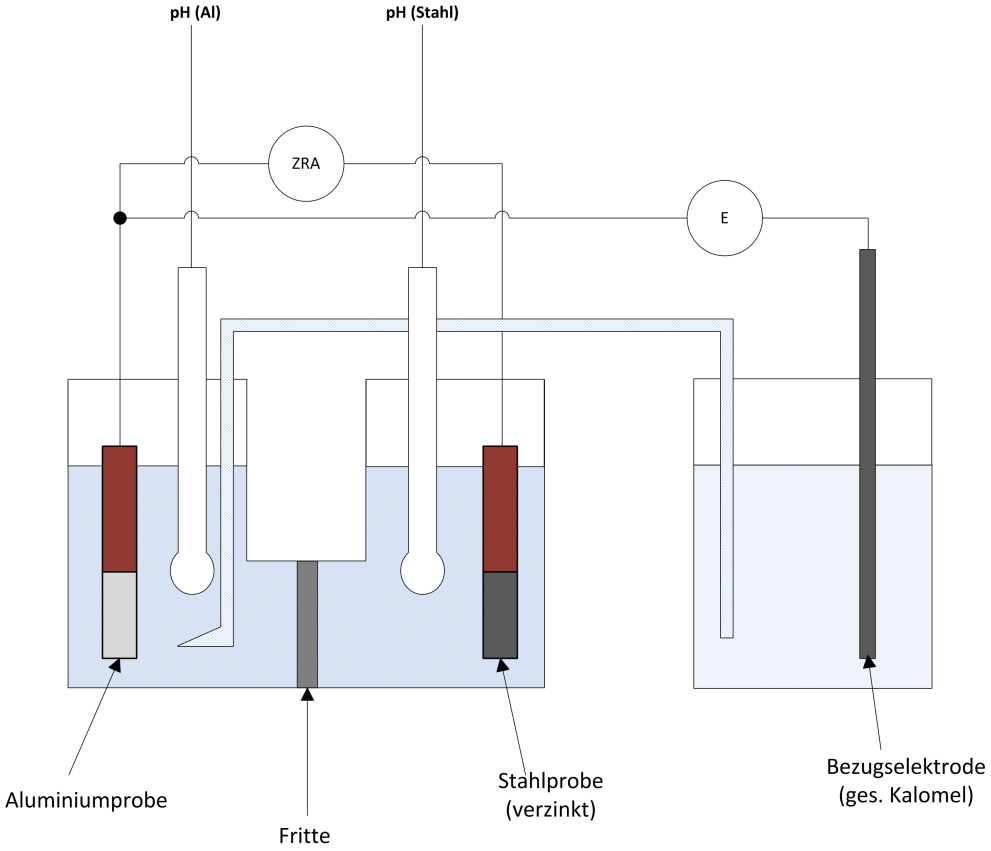


Abbildung 6.8 – Schematische Darstellung einer elektrochemischen Doppelzelle.

geblasen. Durch Oxidation der Zink-Schicht wandern Elektronen über den Leiter ab und es bilden sich positiv geladene Zinkionen (Teilreaktion 6.3). Die Gegenreaktion bzw. Teilreaktion an der Kathode ist die Sauerstoffreduktion und es werden Hydroxidionen OH^- gebildet (6.4).

$$Zn \rightarrow Zn^{2+} + 2e^- \tag{6.3}$$

$$2H_2O + O_2 + 4e^- \rightarrow 4OH^- \tag{6.4}$$

Die Gesamtreaktion der Doppelzelle lautet also:

$$Zn + O_2 + 2H_2O \rightarrow 2Zn^{2+} + 4OH^- \tag{6.5}$$

Der Stromfluss wird über ein *Zero Resistance Amperemeter* (ZRA in Abbildung 6.8) gemessen und stellt das Ausmaß für die galvanische Korrosion dar. Der innenwiderstandsfreie Strom-Spannungswandler verhindert einen Einfluss auf den Versuchsverlauf durch einen ungewollten Spannungsabfall zwischen Anode und Kathode. Das Mischpotential bzw. Korrosionspotential wird in Bezug auf die gesät-

tigte Kalomelelektrode (+0,2412 V gegen SHE[3]) gemessen. Je höher dieses Potential ist, desto schneller verläuft die Korrosion [57]. Der verwendete Elektrolyt ist eine 3 % $NaCl$ Lösung. Da es sich bei $NaCl$ um eine Ionenbindung handelt, dissoziiert diese Verbindung im Wasser in Na^+ und Cl^- Ionen, die den elektrischen Strom transportieren [58, S.147], indem sie je nach Ladung zur Kathode bzw. zur Anode wandern.

6.2.2 Problematik

Man möchte durch diesen Messversuch das Korrosionsverhalten von mehreren Beschichtungen gleichzeitig bzw. simultan untersuchen. Einmal werden die beiden Proben in einer Doppelzelle ohne gegenseitiger Beeinflussung, ein anderes mal in einer Spaltzelle mit gegenseitiger Beeinflussung der Elektroden gemessen. Zum einen ist es möglich, die stattfindenden Reaktionen mit Hilfe der Doppelzelle genauer zu analysieren. Die Untersuchungen in der Doppelzelle können das reale Korrosionsverhalten jedoch nur unzureichend beschreiben, sodass die Proben zusätzlich als galvanisches Paar – simultan – in einer Spaltzelle untersucht werden müssen. Die Simultanmessung von mehreren dieser Zellen erfordert eine komplexe Verdrahtung, da der galvanische Kontakt zwischen den einzelnen Paarungen stets aufrecht erhalten bleiben muss. Das korrekte Zusammenspiel zwischen Schaltung und Messung der einzelnen Kanäle kann in Programmen wie `LabVIEW` sehr schnell zu unüberschaubaren Konstrukten führen, in denen man sich bald nicht mehr auskennt. Spätestens bei nachträglichen Änderungen in der Verdrahtung ist man mit den Nachteilen der *Visuellen Programmierung* konfrontiert, da man sich leicht «verstrickt».

6.2.3 Systemaufbau

Der Messablauf erfolgt mit dem *Agilent 34970A* Multifunktionsgerät sowie einem *pH-Meter*. Das *Agilent 34970A* verfügt über 3 Slots, in welche verschiedene *Plug-In*-Module eingeführt werden können. Für dieses Messsystem werden 2 Multiplexer (Schalt-) Karten des Typs *HP 34901A* verwendet und so verkabelt, dass die erste Karte die Messfunktionen und die zweite Karte die Schaltfunktionen übernimmt. Die Verdrahtung des Messsystems von zwei Doppelzellen und zwei Spaltzellen mit den Geräten und permanentem galvanischen Kontakt wird in Abbildung 6.9 gezeigt.

[3] Standard-Wasserstoffelektrode.

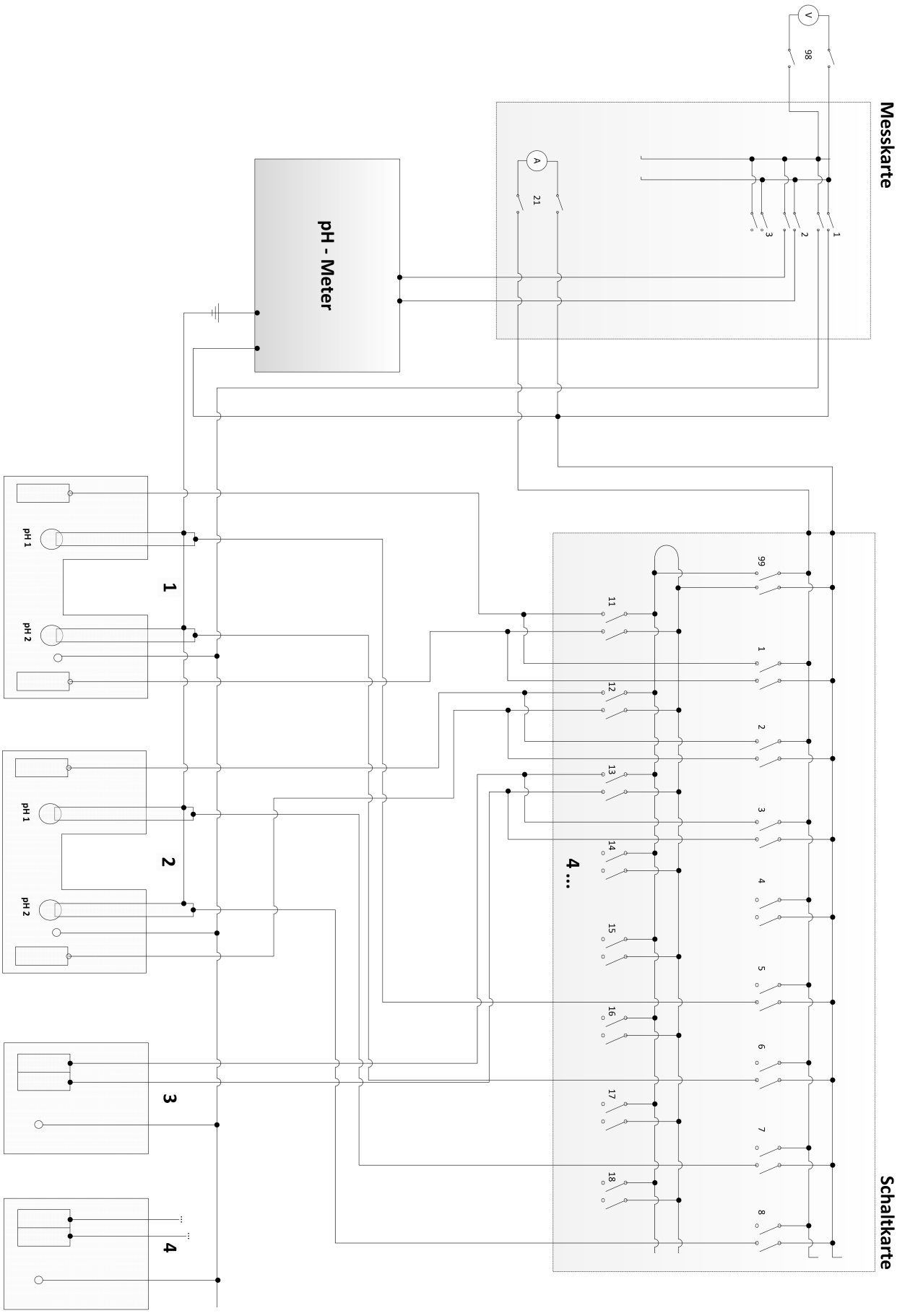


Abbildung 6.9 – Schaltplan für die Messung mehrerer Korrosionszellen mit optionaler pH-Messung und ununterbrochenem galvanischen Kontakt.

(a) (b)

Abbildung 6.10 – Messgerät *Agilent 34970A – Data Acquisition / Switch Unit Family*. *Quelle: Wikimedia Commons*

6.2.4 Steuerung

Für die Simultanmessung elektrochemischer Zellen wurde mit Hilfe von Task++ ein Messprogramm geschrieben, welches im Vergleich zur Komplexität der Schaltung relativ wenige Codezeilen benötigt. Die Eingabemaske für sämtliche Parameter (Abbildung 6.11) wird dynamisch durch das Skript generiert. Zur besseren Gruppierung wird die Eingabe der Parameter in 3 Tabs aufgeteilt. Das folgende Programmfragment erzeugt den Tab «Kanaleinstellungen».

Listing 6.3 – Deklaration der Eingabefelder für den Tab «Kanaleinstellungen».

```
1  gui {
2    tab("Kanaleinstellungen") {
3      group("Messkarte") {
4        parameter($slotMeasurementCard, "Slot der Messkarte", 1) is
             ↪  int[1,3] as combo;
5        parameter($channelVolt, "Kanalnummer der Potentialmessung",
             ↪  1) is int[1,20] as combo;
6        parameter($channelPHVolt, "Kanalnummer der PH -
             ↪ Potentialmessung", 2) is int[1,20] as combo;
7        parameter($channelCurrent, "Kanalnummer der Strommessung",
             ↪ 21) is int[21,22] as combo;
8      }
9      group("Schaltkarte") {
10       parameter($slotSwitchCard, "Slot der Schaltkarte", 2) is
             ↪ int[1,3] as combo;
11     }
12   }
13 }
```

Durch das Keyword `tab` werden die Kindelemente in einem Tab gruppiert, wie in Abbildung 6.11 zu sehen ist. Das `group` Keyword gruppiert die Eingabeelemente, indem sie durch eine Umrandung optisch voneinander getrennt werden. Für die-

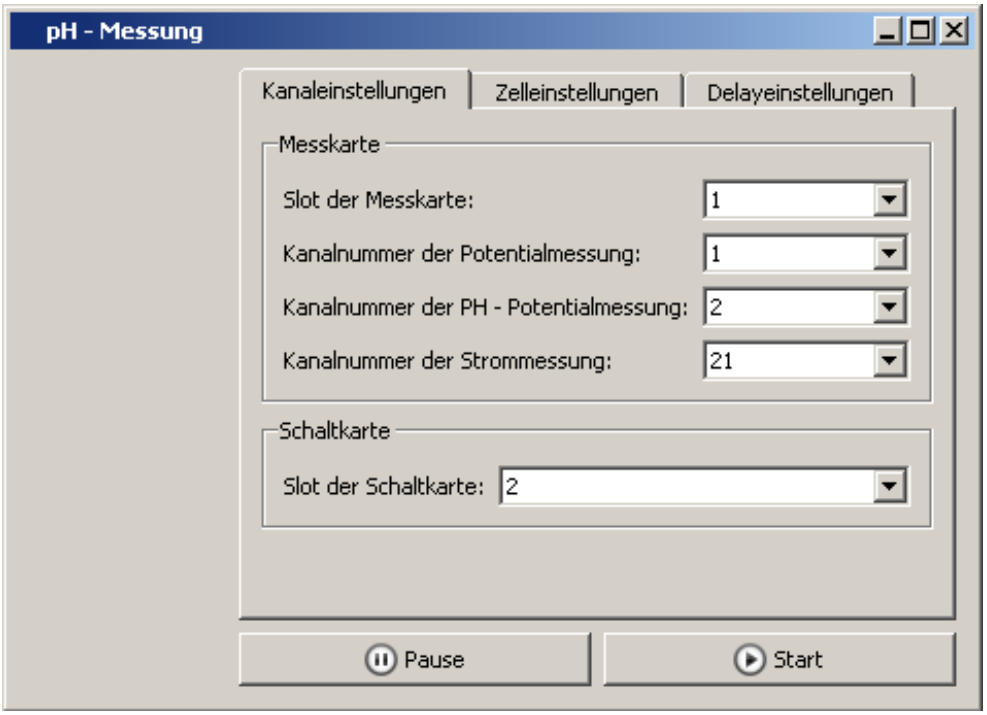


Abbildung 6.11 – Automatisch generierte Eingabemaske für die Simultanmessung von elektrochemischen Zellen.

se Messaufgabe werden 2 Steckkarten benötigt. Eine Karte für die Schaltung der Kanäle und eine weitere Karte für die Strom- und Potentialmessung. Die beiden Eingabefelder *Slot der Messkarte* und *Slot der Schaltkarte* erlauben eine Eingabe eines *Integer*-Wertes zwischen 1 und 3 (`is int[1,3]`) und wird über eine *Dropdown-Box* (`as combo`) ausgewählt. Die Kanalnummern für die beiden Potentialmessungen sowie für die Strommessung werden ebenfalls auf diese weise, jedoch mit einer anderen Intervallangabe, deklariert.
Die Eingaben werden in den angegebenen Variablen gespeichert und können daraufhin im Task++-Skript verwendet werden. Die Eingabe für das Feld *Slot der Schaltkarte* wird der Variablen `$slotSwitchCard` zugewiesen und kann in weiterer Folge im Programmcode verwendet werden.

Listing 6.4 – Initialisierung der Kanäle einer Schaltkarte im Slot 1 des *Agilent-34970A*-Gerätes.

```
1 device $daq("agilent34970a");
2 // Initialization part
3 $daq.reset();
4 for ($i = 1; $i <= 10; $i++) {
5   $chOpen = $slotSwitchCard * 100 + $i;
6   $chClose = $slotSwitchCard * 100 + $i + 10;
7   $daq.route_open($chOpen);
8   $daq.route_close($chClose);
9 }
```

Listing 6.4 zeigt die Verwendung des Eingabewertes in der Initialisierung. Kanäle 1–10 werden geöffnet und Kanale 11–20 werden geschlossen. Die Kanalnummer ist eine dreistellige Zahl, wobei die erste Ziffer den *Slot* bezeichnet und die weiteren den Kanal selbst. Das gesamte Task++-Programm für dieses Experiment ist im Anhang A.5 zu finden.

6.2.5 Ergebnisse

Abbildung 6.12 zeigt den beobachteten Verlauf der galvanischen Korrosion der Materialpaarung verzinkter Stahl und AA6016 in der Doppelzelle. Wie in der Arbeit von DAVID JEROLITSCH [59] erläutert wurde, lassen sich anhand der gesammelten Daten 4 bedeutende Phasen erkennen.

Phase I: In Phase I beginnt die Korrosion von Zink, welche durch einen hohen galvanischen Anteil charakterisiert ist. Das Mischpotential wird ebenfalls durch die Zink-Elektrode bestimmt. Der Anstieg des pH-Wertes lässt eine zunehmende Eigenkorrosion des Zinks erkennen.

Phase II: In Phase II findet eine Passivierung des Aluminiums an der Kathode durch eine vorkommende Sauerstoffreduktion statt. Eine weitere Sauerstoffreduktion wird dadurch gehemmt und der galvanische Strom sinkt. An der Zink-Elektrode findet daher vorwiegend Eigenkorrosion statt und das Aluminium ist kathodisch geschützt.

Phase III: Nach und nach verlassen Zinkionen die Anode und es entstehen partiell freiwerdende Stahloberflächen. Dies führt zu einem Anstieg des Mischpotentials bis zum Lochkorrosionspotential des Alumiuniums.

Phase IV: In Phase IV kehrt die Stromrichtung um. Der freigelegte Stahl wird zur Kathode und die Aluminium-Elektrode zur Anode. Das Mischpotential bleibt nahezu konstant und der Strom steigt stetig an. Die Stahl-Elektrode wird nun vom Aluminium kathodisch geschützt, wodurch an dieser keine Korrosion mehr stattfindet.

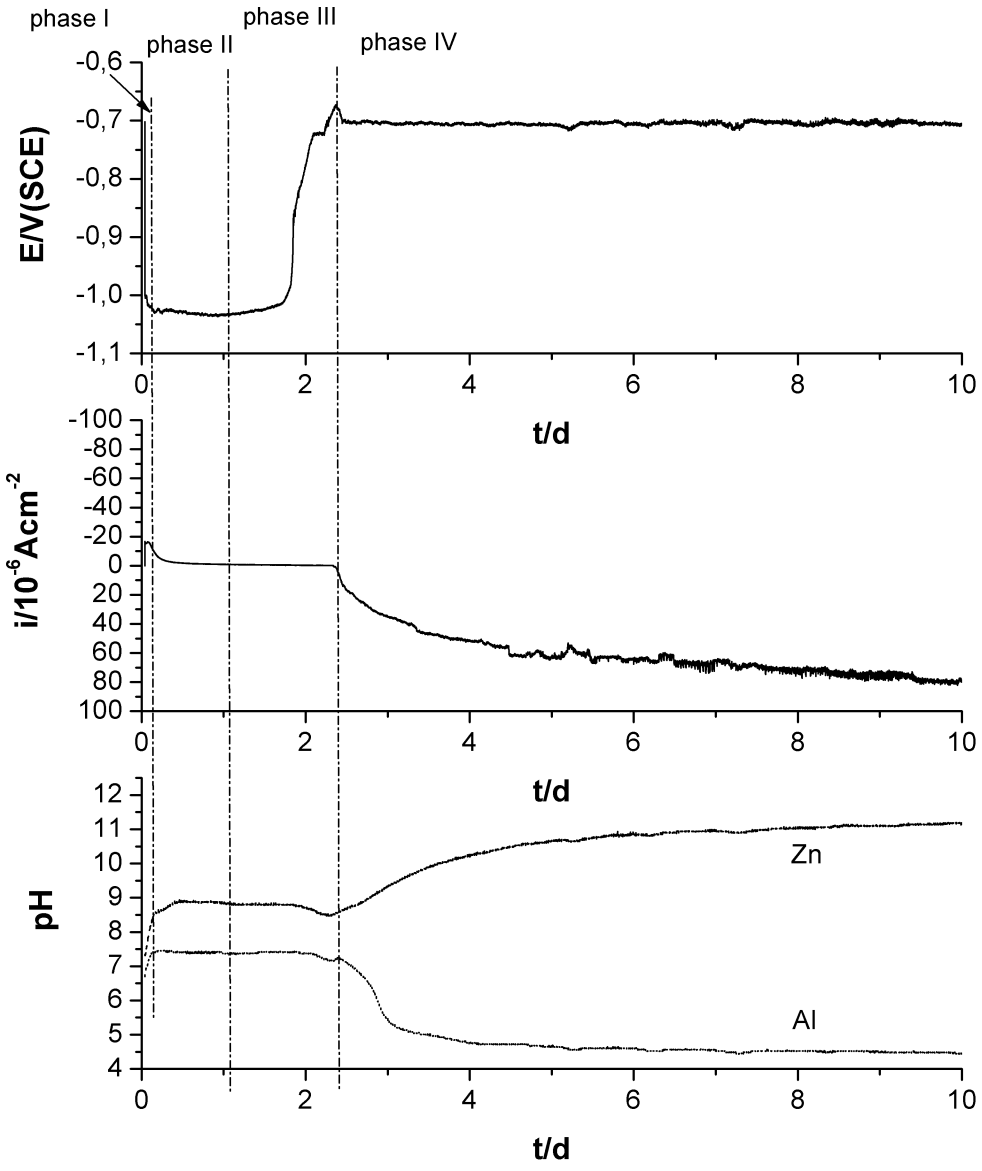


Abbildung 6.12 – Strom-, Potential- und pH-Wertverlauf der Korrosion von AA6016 und galvanisiertem Stahl mit einer Zinkschicht von 7.5 µm.

Eine Phase V mit einer Korrosion des Stahls würde nach Auflösung des Aluminiums eintreten. Dies wurde in diesem Experiment jedoch nicht untersucht.

Durch das Integral der gesamten Korrosionsstromkurve (bis zum Beginn der Aluminiumkorrosion) lässt sich mit Hilfe der *Faraday'schen Gesetze* der galvanische Anteil der Korrosion berechnen. Laut dem ersten *Faraday'schen Gesetz* ist die durch die Elektrode geflossene Ladung $Q = I \cdot t$ proportional zur Stoffmenge n bzw. zur Masse m, die während der Elektrolyse abgeschieden wird. Dabei stellt I die konstante Stromstärke und t die Elektrolysezeit dar. Die benötigte Ladung, um eine Stoffmenge n eines Ions mit der Wertigkeit z abzuscheiden, ist $Q = n \cdot z \cdot F$, wobei F die Faradaykonstante ist.

Mit der Definition der Molmasse $M = \frac{m}{n}$ ergibt sich:

$$Q = \frac{m \cdot z \cdot F}{M} \tag{6.6}$$

Durch Umstellung erhält man die Gleichung zur Berechnung der Masse m, welche durch die Elektrolyse abgeschieden wird:

$$m = \frac{M \cdot Q}{z \cdot F} \tag{6.7}$$

Anhand dieser Zusammenhänge berechnet sich für das Tripel Aluminium-Zink-Stahl ein galvanischer Korrosionsanteil von ca. 2.5 %. Die restlichen 97.5 % sind Eigenkorrosion.

Ein Anstieg des pH-Wertes deutet auf eine zunehmende Eigenkorrosion. In Phase I beträgt die mittlere Änderungsrate bis zu $1\,pH/h$.

$$\left.\frac{\Delta pH}{\Delta t}\right|_{max} = \frac{1\,pH}{h} \tag{6.8}$$

Möchte man den pH-Wert in Korrelation mit dem gemessenen Stromfluss bringen, so ist durch die Messverzögerung eine Verzerrung im Ergebnis zu erwarten. Bei einer automatisiert stattfindenden Messung von mehreren Zellen kann ein Intervall von 10 Sekunden eingehalten werden.

Die Verzerrung der Korrelation ist akzeptabel:

$$\frac{1\,pH}{h} \cdot 10\,s = \frac{1}{360} \approx 0.0028\,pH \tag{6.9}$$

6.3 Korrosionsmessung im Fahrzeug

6.3.1 Übersicht

Korrosionsschutz an Fahrzeugen stellt für den Automobilbau eine besondere Herausforderung dar. Durch die Mobilität fallen an einem Fahrzeug die unterschiedlichsten Korrosionsursachen zusammen. Man denke an Auftausalzen auf der Fahrbahn während der kalten Jahreszeit oder an starker Sonneneinstrahlung in feuchten Regionen im Hochsommer. Wegen der Verwendung verschiedenster Materialien ist im Fahrzeugbau ein einheitlicher Korrosionsschutz von großem Interesse. Das Augenmerk liegt hauptsächlich auf der passenden Materialauswahl sowie geeigneten Beschichtungen der Materialoberflächen. Die Verwendung von verzinktem Stahl oder Aluminum hat sich bei den meisten Automobilherstellern als Grundlage für den Korrosionsschutz etabliert.

Mit Hilfe von elektrochemischen Messungen ist es möglich, einen Überblick über die Korrosionseigenschaften der verwendeten Materialien zu erhalten. Vor allem zyklovoltammetrische Diagramme geben Auskunft über die Wechselwirkungen zwischen Strom und Potential eines zu untersuchenden Materials und somit ein Bild über dessen korrosives Verhalten. Durch diesen Informationsgewinn ist es möglich, das Korrosionsverhalten abzuschätzen und die Einsatzgebiete zu bestimmen.

6.3.2 Problematik

Daten, welche aus Untersuchungsmethoden im Labor gewonnen werden, haben den Nachteil, dass das resultierende Bild nur sehr eingeschränkt der Wirklichkeit entspricht. Praxisnahe Prozesse wie z. B. Auftrocknung von Elektrolyt können bei Labormessungen nur eingeschränkt simuliert werden.
Die Durchführung von Messungen im Fahrzeug erfordert eine robuste Messeinrichtung, welche großen Erschütterungen standhalten muss. Zum Einsatz kommen teuere Messgeräte, welche über einen längeren Zeitraum, ausfallsicher und zuverlässig, Messdaten aufnehmen müssen. Ein auch nur kurzzeitiger Ausfall des Messsystems könnte zum Verlust von wichtigen Messdaten führen. Kurzzeitig stattfindende Ereignisse können sich auf den weiteren Korrosionsverlauf auswirken und sind daher von relevanter Bedeutung. Auch die Energieversorgung der eher stromhungrigen Messeinrichtung muss über den gesamten Messzyklus gesichert sein.

Für dieses Experiment ist ein koordinierter Ablauf nötig, welcher mit Hilfe von einem eingebauten Laptop bewerkstelligt wird. Software von den Herstellern der

(a) (b)

Abbildung 6.13 – Korrosionsrelevante Messstellen im Audi Q7. *Quelle: [59]*

Messgeräte steht für diesen Spezialfall erwartungsgemäß nicht zur Verfügung. Der gesamte Messablauf sowie dessen Parameter müssen während des Betriebes zu jeder Zeit verändert bzw. angepasst werden können.

6.3.3 Systemaufbau

Für die realitätsnahe Messung des Korrosionsverhaltens von Falzverbindungen wurde ein Werksdienstwagen des Typs *Audi Q7* der AUDI AG verwendet. Dabei wurden 4 korrosionsrelevante Stellen am Messfahrzeug identifiziert [59] und mit Modellprüfkörpern ausgestattet, dessen Produktionsprozess für die Verwendung in elektrochemischen Messungen angepasst wurde. Diese standardisierten Probenkörper der AUDI AG stellen realitätsnahe Nachbildungen von Falzen dar, wie sie typischerweise in der Automobilindustrie Verwendung finden.

Abbildung 6.13 zeigt die korrosionsrelevanten Messstellen am Fahrzeug:

Messstelle 1: Vor dem Kühler

Messstelle 2: Seitlicher Wasserablauf an der Frontklappe

Messstelle 3: Frontklappe scheibenseitig

Messstelle 4: Unterer Falzbereich an der Heckklappe

An jede dieser Stellen wurden die Probenkörper angebracht und mit 4-poligen Kabeln verbunden. Die Messung erfolgte mit dem *Agilent 34970A*, welches mit einer 220-V-Spannung versorgt werden musste. Während des Fahrbetriebes wurde die Stromversorgung über die Batterie und mit Hilfe der Lichtmaschine gesichert. Eine

Abbildung 6.14 – Messeinrichtung in einem Audi Q7. *Quelle: [59]*

12 V / 100-Ah-Pufferbatterie sorgt für eine Überbrückung von bis zu 24 Stunden bis das abgestellte Fahrzeug von einem 220-V-Stromnetz versorgt werden kann. Abbildung 6.14 zeigt den Einbau des Messsystems samt Messgerät, Laptop, Pufferbatterie, Ladegerät und Spannungswandler.

Diese unhandliche Messeinrichtung führte zu weiteren Überlegungen und letztendlich zur Entwicklung eines eigenen Niedrigstrommessgerätes, dessen Einsatz speziell für solche herausfordernde Umgebungen bestimmt ist. Dabei sind Robustheit und niedriger Energieverbrauch die wesentlichen Anforderungen, denen diese Eigenentwicklung entgegen kommen muss. Der Kern des Gerätes basiert auf einen Mikrocontroller, welcher eine Peripherie mit mehreren Komponenten ansteuert. Messdaten werden auf einer SD-Karte gespeichert, bevor das System bis zum nächsten Messzyklus in den Energiesparmodus wechselt. Der Aufbau und die Funktionsweise dieser Eigenentwicklung wird im Anhang B.1 näher erläutert.

6.3.4 Steuerung

Die Steuerung der Messeinrichtung ist jener für die Simultanmessung elektrochemischer Zellen sehr ähnlich, daher wird an dieser Stelle auf eine erneute Darstellung der Ablaufsteuerung durch Task++ verzichtet und stattdessen auf Abschnitt 6.2.4 verwiesen.

6.3.5 Ergebnisse

Die gesammelten Daten der Messung im Mobilbetrieb stellen an sich schon eine neuartige Grundlage für die Korrosionsanalyse dar. Das Ergebnis der 5-monatigen Korrosionsaufzeichnung von 8 Werkstoffproben erlaubt einen attraktiven Einblick in die realen Abläufe der Geschehnisse. Durch einen Abgleich mit Messergebnissen aus dem Labor, können die Daten evaluiert und die Messmethode verbessert werden. Die Ergebnisse dieses Experiments werden in der Dissertation von DAVID JEROLITSCH ausführlich behandelt [59, S.126–130] und diskutiert [59, S.176–178].

6.4 Scanning Electrochemical Microscopy

6.4.1 Übersicht

Dieses Kapitel beschäftigt sich mit dem Aufbau und der Steuerung eines *Scanning-Electrochemical-Microscopy (SECM)*-Systems, das dazu dient, die Topografie und die chemische Reaktivität von Substraten zu untersuchen. Das Kernstück dieses Verfahrens ist eine Ultramikroelektrode (UME) mit einem Durchmesser a in der Größenordnung von einigen nm bis hin zu 25 µm.
Mit Hilfe solcher Mikroelektroden werden Ströme gemessen, während diese sich in einer Lösung und nahe eines Substrats befinden. Dabei handelt es sich meistens um feste Oberflächen wie Glas, Metall, biologisches Material usw., welche die elektrochemische Reaktion der Mikroelektrode bzw. Messspitze anregen. Dies ermöglicht es, Information über die Beschaffenheit des Substrats zu gewinnen. Die Bewegung der Mikroelektrode erfolgt durch hochauflösende Positioniersysteme wie piezoelektrische Aktuatoren und *Stepper*- bzw. geregelte Stellmotoren. Die Richtungen X und Y stellen die Ebene dar; die Z-Richtung den Abstand zum Substrat.

6.4.2 Problematik

Ein *SECM*-System erfordert eine komplexe Ablaufsteuerung. In einigen Fällen ist es möglich auf freie Software oder kommerzielle Lösungen zurückzugreifen. In anderen Fällen ist es notwendig, Programme zur Bedienung des Systems selbst zu entwickeln. Wie in diesem Kapitel noch verdeutlicht wird, ist vor allem die **Positionierung der Messspitze** sowie deren **Signalaufnahme** eine Aufgabe, bei welcher man auf Eigenlösungen angewiesen ist [60, S.34]. Sobald unterschiedliche Softwarelösungen für die Steuerung des Systems eingesetzt werden, ist es schwierig, die Messdaten in einer einheitlichen Form zu erfassen. Während eine Software für die

Positionierung der Mikroelektrode verantwortlich ist, hat die Software für die Messung der Ströme keine Auskunft über deren Position. Folgende Richtlinien sollten von einer Software zur Steuerung in einem *SECM*-System eingehalten werden:

- Bewegungen der Mikroelektroden sollten in Echtzeit überwacht werden können, um Beschädigungen an der Messspitze bzw. am Substrat zu vermeiden.
- Daten sollten sofort gespeichert werden, damit nach einem eventuellen Computercrash oder einer Fehlbedienung eine Wiederherstellung möglich ist.
- Daten sollten in Korrelation zu Messzeit, Hardwareeinstellungen, Scangeschwindigkeit usw. stehen.

Im Wesentlichen kann der Ablauf zur Steuerung eines *Scanning Electrochemical Microscopes*, wie in Abbildung 6.15 dargestellt, in 4 Bereiche unterteilt werden:

Bewegung der Messspitze: Die Mikroelektrode muss sich in alle 3 Richtungen bewegen können und ihre Position muss zu jeder Zeit bekannt sein. Die Positionierung erfolgt über Präzisions-Linearaktoren sowie lineare Piezoversteller.

Signalaufnahme: Die Signalaufnahme geschieht über einen hochauflösenden Potentiostaten, welcher die gemessenen Werte in digitalisierter Form bereitstellt.

Hilfsroutinen: Durch Hilfsroutinen kann das Programm erweitert werden. Diese können eine Voltammetrie durchführen oder den Abstand der Mikroelektrode zum Substrat steuern.

Darstellung und Analyse: Während eines Oberflächenscans ist es sinnvoll, Feedback über den Ablauf zu erhalten. Eine einfache grafische Darstellung ist ausreichend. Sobald der komplette Datensatz aufgenommen wurde, können Datenanalyseprogramme verwendet werden, um eine bestmögliche Auswertung und Darstellung zu erreichen.

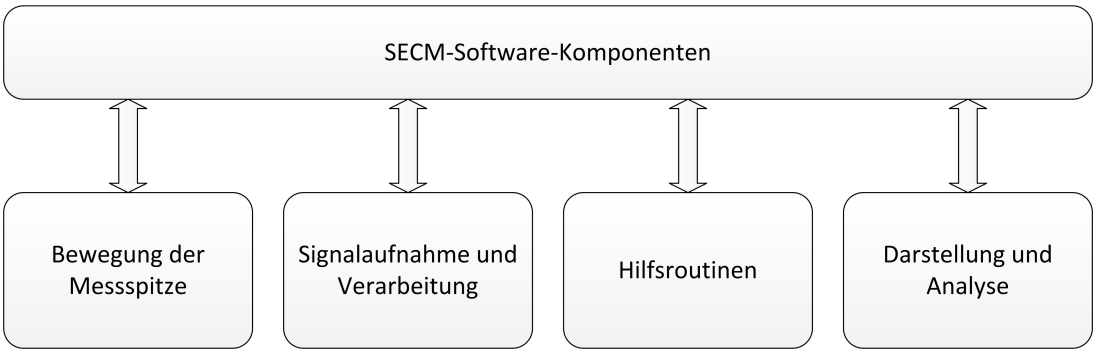


Abbildung 6.15 – Software-Komponenten für die Steuerung eines *Scanning-Electrochemical-Microscopy*-Systems.

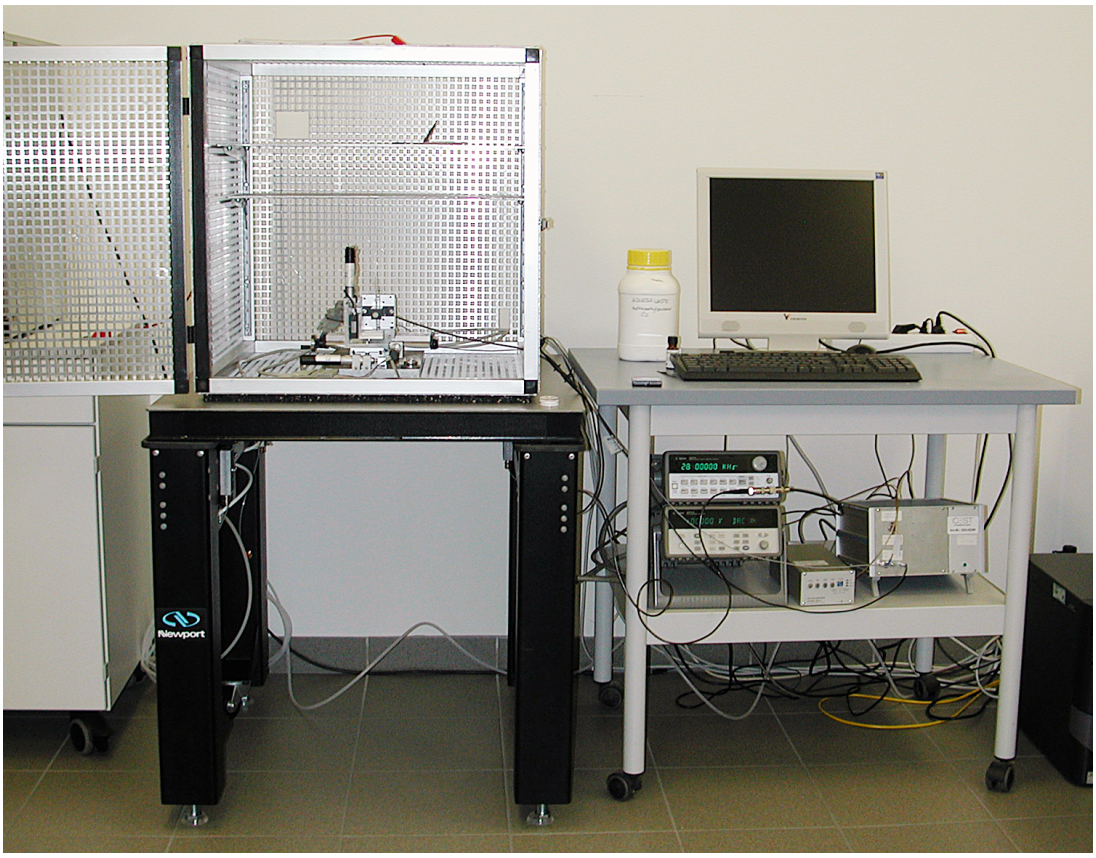


Abbildung 6.16 – Das gesamte *Scanning-Electrochemical-Microscopy*-System. Die Positioniergeräte befinden sich in einem Faraday'schen Käfig auf einem schwingungsdämpfenden Tisch.

6.4.3 Systemaufbau

Stellmotoren

Die Positionierung des zu untersuchenden Substrats erfolgt durch hochauflösende Linearaktoren des Typs *M-227.10* von PHYSIKINSTRUMENTE. Diese befinden sich auf einem schwingungsdämpfenden Stelltisch und werden direkt vom Computer über ein *PCI-Interface* gesteuert. Das Messobjekt kann um einen Stellweg von maximal 10 mm in die X- sowie in die Y-Richtung bewegt werden. Der Linearaktor für die Positionierung in der Z-Ebene ist für die «grobe» Höhenverstellung der Elektrode verantwortlich. Die kleinste Schrittweite des Aktors beträgt 50 nm und ist für die Auflösung der Probenoberfläche ausreichend. Für eine exakte Höheneinstellung ist diese Auflösung jedoch zu niedrig, sodass die Feinjustierung in der Z-Richtung zusätzlich mit Hilfe von einem piezobasierten Positioniersystem übernommen wird.

Piezopositioniersystem

Das piezobasierte Positioniersystem ist für die präzise Bewegung der Mikroelektrode verantwortlich. Durch ein angelegtes Spannungssignal wird ein piezoelektrisches Material verformt, wodurch die Messspitze im Nanobereich positioniert werden kann. Der verwendete Aktor des Typs *P-611.1S* ist ebenfalls von PHYSIKINSTRUMENTE und erlaubt eine maximale Weglänge von 100 µm mit einer minimalen Schrittweite von 2 nm. Der Aktor dieses Typs ist eigentlich für die Bewegung in der X-Ebene gedacht, wurde jedoch durch geeignete Adaption für die Feineinstellung der Z-Position verwendet.

Shear-Force

Voraussetzung für ein möglichst unverzerrtes Oberflächenbild ist ein konstanter Abstand zwischen Elektrode und Substrat. Abstandsbedingte Stromänderungen können in der Regel von reaktivitätsbedingten Stromänderungen nicht unterschieden werden [61]. Daher muss während des gesamten Scans eine konstante Entfernung zur Oberfläche eingehalten werden. Durch dieses Verfahren ist es neben der Strommessung zusätzlich möglich, ein topografisches Bild des Substrats zu erhalten, indem für jeden Punkt in der Ebene die Z-Position ausgewertet wird.
Eine bewährte Vorgehensweise für die Abstandskontrolle ist die *Shear-Force*-Methode. Hierzu wird die Spitze der UME[4] durch einen Signalgeber in eine mechanische Resonanz zwischen üblicherweise 10 und 50 kHz gesetzt.
Durch die *Shear-Force* zwischen Spitze und Probenoberfläche wird die Schwingung

[4] Ultra-Mikro-Elektrode.

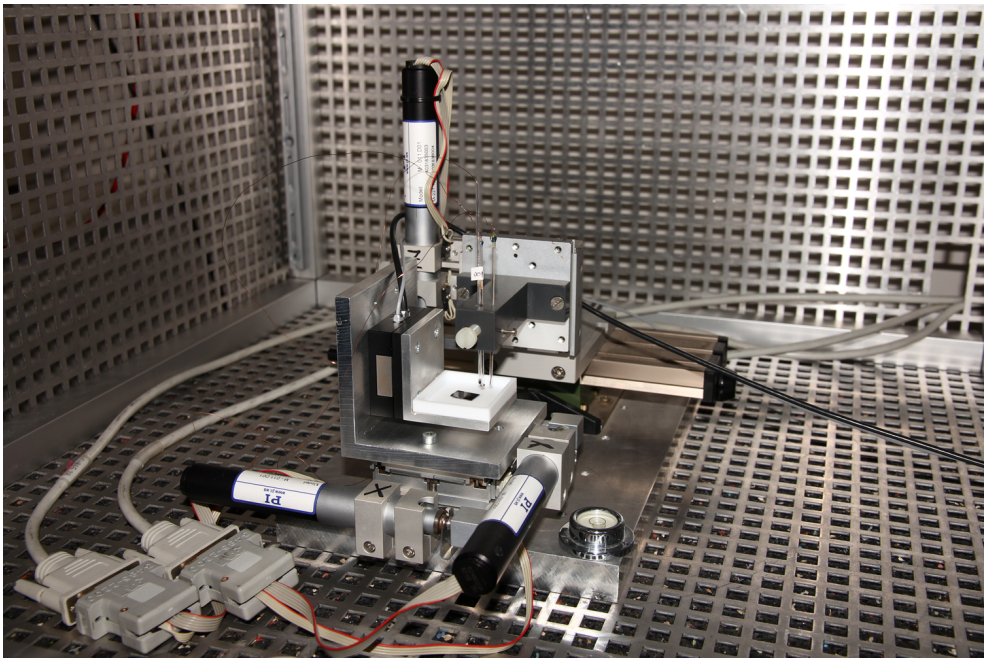


Abbildung 6.17 – Das Positioniersystem mit den Stellmotoren und dem *Piezo-Pusher* im Faraday'schen Käfig.

der Elektrode gedämpft, was zu einer Änderung der Resonanzfrequenz führt. Als Signalgeber und -nehmer kommen Piezoschallwandler zum Einsatz, die auf den Glasmantel der Elektrode aufgeklebt werden. Der Signalgeber wird an einen Frequenzgenerator von AGILENT TECHNOLOGIES des Typs *33120A – 15 MHz Function Generator / Arbitrary Waveform Generator* angeschlossen. Der Signalnehmer empfängt das mechanisch übertragene Signal und führt dieses in einen *Lock-in*-Verstärker weiter. In Abbildung 6.18 werden die Komponenten für die *Shear-Force*-Methode schematisch dargestellt.

Lock-in Verstärker

Der *Lock-in*-Verstärker erhält als Eingang ein Referenzsignal sowie ein moduliertes Messsignal. Das Referenzsignal kommt vom *Agilent 33120A* und setzt die Elektrode mit Hilfe des Piezoschallwandlers in eine mechanische Schwingung. In der Nähe des Substrats wird diese Schwingung in ihrer Amplitude und Phase verändert. Das von einem weiteren Schallwandler aufgenommene veränderte Signal wird als moduliertes Messsignal in den Verstärker geführt. Der *Phasenverschieber* im *Lock-in*-Verstärker verschiebt das Referenzsignal, sodass die beiden Signale wieder phasengleich sind. Diese werden anschließend durch einen *Multiplizierer* geschickt. Ein *Tiefpass / Integrator* mittelt das dadurch entstandene Signal über die *Lock-in*-Zeit T und liefert eine Gleichspannung als Ausgang (siehe Formel 6.10).

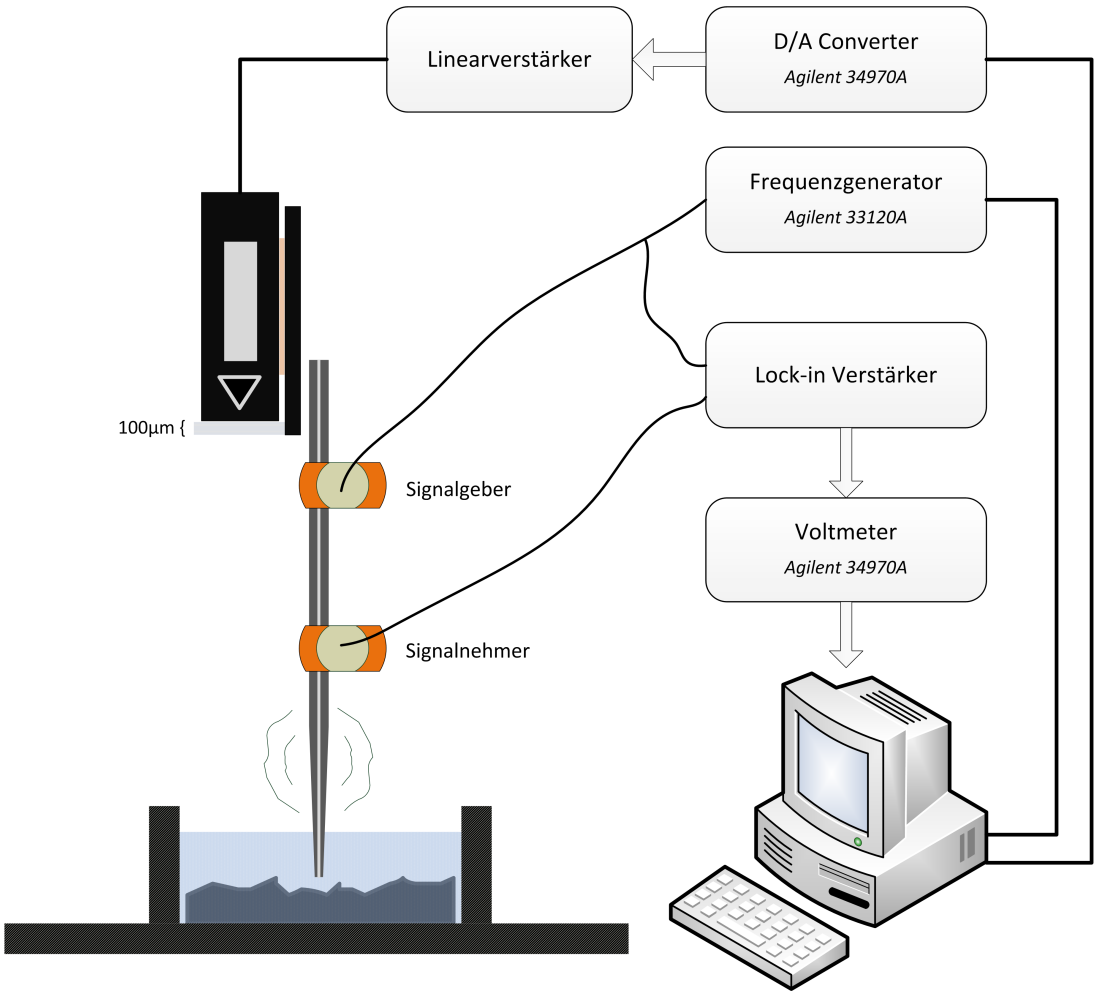


Abbildung 6.18 – Schematische Darstellung der *Shear-Force*-Methode.

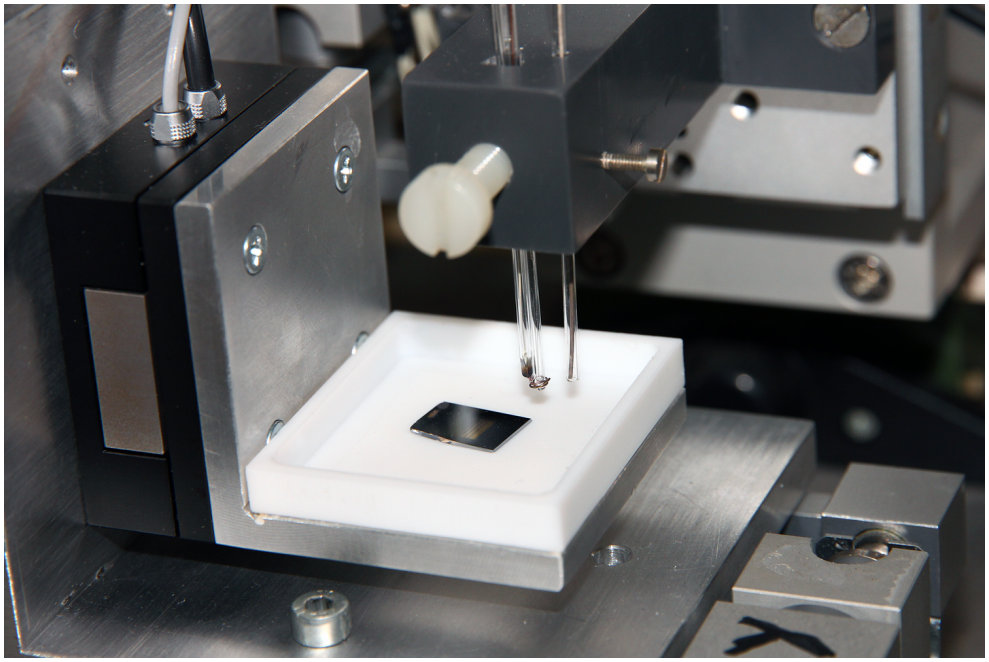


Abbildung 6.19 – Nahaufnahme der Mikroelektrode mit dem Substrat.

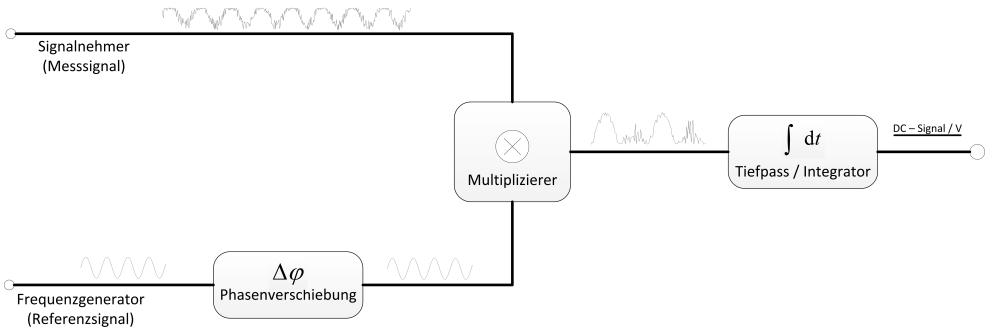


Abbildung 6.20 – Schematische Darstellung der *Lock-in*-Funktionalität.

$$U_{out}(t) = 1/T \int_{t-T}^{t} \sin[2\pi f_{ref} \cdot s + \Delta\varphi] U_{in}(s)\,\mathrm{d}s \tag{6.10}$$

Die Funktionsweise der *Lock-in*-Funktionalität ist in Abbildung 6.20 (ohne Ausgang für $\Delta\varphi$) schematisch dargestellt.

Bipotentiostat

Für die Stromaufnahme an der Mikroelektrode wird ein *Niedrig-Strom*-Bipotentiostat verwendet, welcher von GÁBOR MÉSZÁROS und TAMÁS PAJKOSSY speziell für die Nanoelektrochemie entwickelt wurde. Der *Nanobistat* verfügt über Anschlüsse für 2 Arbeitselektroden, *WE1* und *WE2*, eine Gegenelektrode *C* sowie eine Referenzelektrode *R*.
Für die Kommunikation mit *Nanobistat* wurde zusammen mit GÁBOR MÉSZÁROS eine Treiberbibliothek geschrieben, um das Gerät in das Task++-System einbinden zu können. Alternativ ermöglicht eine von MÉSZÁROS und PAJKOSSY geschriebene Software die volle Ausschöpfung der Funktionalität dieses hochempfindlichen Messgerätes. Der *Nanobistat*-Potentiostat wird über eine *USB*-Schnittstelle angesprochen. Die *USB*-Signale werden von einem Optokoppler, wie in Abbildung 6.21 dargestellt, in optische Signale umgewandelt, um Interferenzen zu vermeiden.

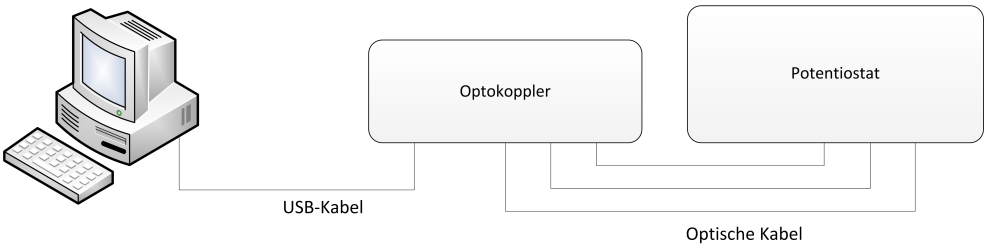


Abbildung 6.21 – Vermeidung von Interferenzen durch Umwandlung der *USB*-Signale mit Hilfe von einem Optokoppler.

6.4.4 Steuerung

Die Steuerung eines *SECM*-Ablaufs kann grob in 2 Phasen aufgeteilt werden:

Phase 1: In Phase 1 wird mit Hilfe eines *Frequency Scans* versucht, eine geeignete Frequenz für die *Shear-Force*-Methode zur Abstandskontrolle zu finden.

Phase 2: In Phase 2 wird mit konstantem Abstand der Mikroelektrode über die Oberfläche iteriert und ein elektrochemisches Bild aufgenommen.

Frequency Scan

Nachdem die Piezo-Schallwandler an der Mikroelektrode angebracht wurden, möchte man wissen, in welchem Frequenzbereich sich die Dämpfung (nahe des Substrats) am meisten auf das Messsignal auswirkt. Dies ist natürlich immer verschieden und muss daher für jede Elektrode und vor jeder Durchführung separat vorgenommen werden. Dazu wird ein Frequenzspektrum zwischen üblicherweise 10 und 50 kHz erstellt. Das Spektrum zeigt für jede Frequenz das DC-Signal des *Lock-in*-Verstärkers der Amplitude und der Phase. Zuerst wird die Messspitze frei positioniert und ein Frequenzspektrum erstellt. Für das zweite Spektrum wird die Messspitze in die Lösung eingetaucht, jedoch noch nicht sehr nahe dem Substrat.

Ein Task für die Erzeugung zweier Frequenzspektren könnte wie in Listing 6.5 aussehen.

Listing 6.5 – Programm für die Erstellung der Frequenzspektren.

```
task $frequencyscan("Frequency Scan") {
  gui {
    tab("Frequency Scan Setup") {
      group("Positionseinstellungen") {
        parameter($positionOut, "Position Messspitze (aus) [mm]",
            ↪  0) is int;
        parameter($positionIn, "Position Messspitze (in) [mm]",
            ↪ 50) is int;
      }
      group("Frequenzeinstellungen") {
        parameter($freqStart, "Startfrequenz [Hz]", 10) is int;
        parameter($freqEnd, "Stopfrequenz [Hz]", 50000) is int;
        parameter($amplitude, "Amplitude [Volt]", 2.5) is double
            ↪ [0, 5];
        parameter($offset, "Offset [Volt]", 0) is double[0,2.5];
      }
    }
    tab("Multimeter / IO") {
      group("Multimeter Einstellungen") {
        parameter($slotNumber, "Slot der Messkarte", 1) is int
            ↪ [1,3] as combo;
        parameter($channelAmp, "Kanal der Amplitudenmessung", 1)
            ↪ is int[1,20] as combo;
```

```
19          parameter($channelPhase, "Kanal der Phasenmessung", 2) is
               ↪  int[1, 20] as combo;
20          parameter($delay, "Wartezeit vor Messung [ms]", 10) is
               ↪ int[0, 100];
21        }
22        group("I/O Einstellungen") {
23          parameter($filenameOut, "Dateiname Spektrum (aus)", "freq
               ↪ -out.txt") is string;
24          parameter($filenameIn, "Dateiname Spektrum (in)", "freq-
               ↪ in.txt") is string;
25        }
26      }
27    }
28
29    // Get devices
30    device $c843("c843");
31    device $fg("agilent33120a");
32    device $daq("agilent34970a");
33
34    $chMeasAmp = $slotNumber * 100 + $channelAmp;
35    $chMeasPhase = $slotNumber * 100 + $channelPhase;
36
37    // Frequenzspektrum fuer freistehende Spitze
38    $c843.setZ($positionOut);
39    for ($freq = $freqStart; $freq <= $freqEnd; $freq += 10) {
40      $fg.applySin($freq, $amplitude, $offset);
41      wait($delay);
42      $resultAmp = $daq.measure_current_dc($chMeasAmp);
43      $resultPhase = $daq.measure_current_dc($chMeasPhase);
44      printf($filenameOut, $freq + ", " + $resultAmp + ", " +
           ↪ $resultPhase);
45    }
46
47    // Frequenspektrum fuer eingetauchte Spitze
48    $c843.setZ($positionIn);
49    for ($freq = $freqStart; $freq <= $freqEnd; $freq += 10) {
50      $fg.applySin($freq, $amplitude, $offset);
51      wait($delay);
52      $resultAmp = $daq.measure_current_dc($chMeasAmp);
53      $resultPhase = $daq.measure_current_dc($chMeasPhase);
54      printf($filenameIn, $freq + ", " + $resultAmp + ", " +
           ↪ $resultPhase);
55    }
56
57    $c843.setZ($positionOut);
58  }
```

Abstandskontrolle

Nach der Wahl einer geeigneten Frequenz kann nun mit Hilfe der *Shear-Force*-Methode eine Annäherung der Elektrode an das Substrat vorgenommen werden. Die Spitze wird dazu mit freiem Auge sehr nahe an der Probenoberfläche in Position gebracht und dann in kleinen Schritten dem Substrat angenähert. Sobald die Distanz zwischen Substrat und Messspitze nur noch wenige Dutzend Nanome-

ter beträgt wird die Schwingung der Spitze sehr stark gedämpft. Dies macht sich durch einen starken Abfall der Amplitude des DC-Signals vom *Lock-in*-Verstärker bemerkbar [61]. Bei einer gut gewählten Frequenz können sich Signaländerungen von durchaus 30 % ergeben.

Oberflächenscan

Um ein Oberflächenbild der elektrochemischen Reaktivität zu erstellen, wird die Mikroelektrode rasterförmig über das Substrat geführt. Die *Shear-Force*-Komponente sorgt währenddessen für eine konstante Entfernung zum Objekt, damit die Messung nicht durch abstandsinduzierte Stromänderungen verzerrt wird. Auch eine eventuelle Beschädigung der Mikroelektrode wird dadurch vermieden, wenn die Abstandskontrolle vor jeder Neupositionierung der Mikroelektrode durchgeführt wird. Zudem ist es möglich, ein topografisches Oberflächenbild zu erstellen, indem zu jedem Punkt in der Ebene die Z-Position ausgewertet wird.

Durch den sichergestellten konstanten Abstand zur Oberfläche ist gewährleistet, dass der gemessene Strom an der Mikroelektrode das Bild der Oberflächenaktivität des Substrates zeigt und über das gesamte Raster unverzerrt bleibt.

6.4.5 Ergebnisse

Abbildung 6.22 zeigt das Ergebnis eines *Frequency Scans* einer Mikroelektrode: 2 Frequenzspektren (jeweils mit Betrag und Phase) bis 50 kHz mit einer freistehenden sowie einer eingetauchten Messspitze. Mit Hilfe dieser Information kann herausgefunden werden, in welchem Frequenzbereich bzw. mit welcher Frequenz ein aussagekräftiger Wert für die Dämpfung der schwingenden Spitze entsteht. Hierfür kann man zwischen Betrag oder Phase wählen. Für die Auswahl steht in der Regel kein Algorithmus zur Verfügung, mit einem geschulten Auge ist eine geeignete Frequenz jedoch auffindbar [62].

Betrachtet man das Differenzspektrum in Abbildung 6.23, so lassen sich im Bereich zwischen 18 kHz und 26 kHz interessante Frequenzen erkennen, welche als mögliche Werte herangezogen werden können. Auch im niedrigen Frequenzbereich zwischen 0 und 5 kHz zeigen sich Frequenzen mit einer hohen Empfindlichkeit. Dieser Bereich ist erfahrungsgemäß jedoch eher ungeeignet. Als Ursache könnte der proportionale Zusammenhang zwischen Frequenz und Dämpfung eine Rolle spielen.

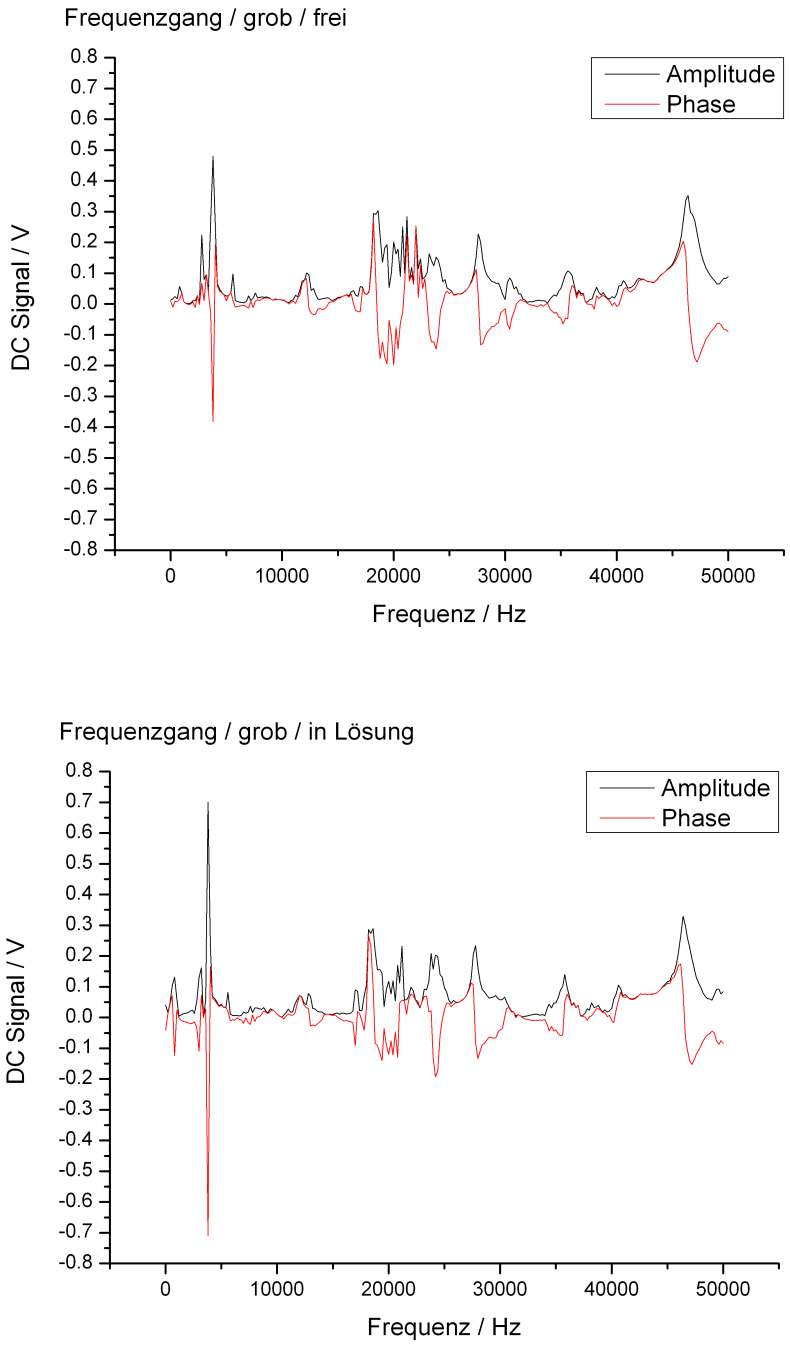


Abbildung 6.22 – Frequenzspektren der schwingenden Messspitze ausserhalb (oben) und innerhalb einer Lösung.

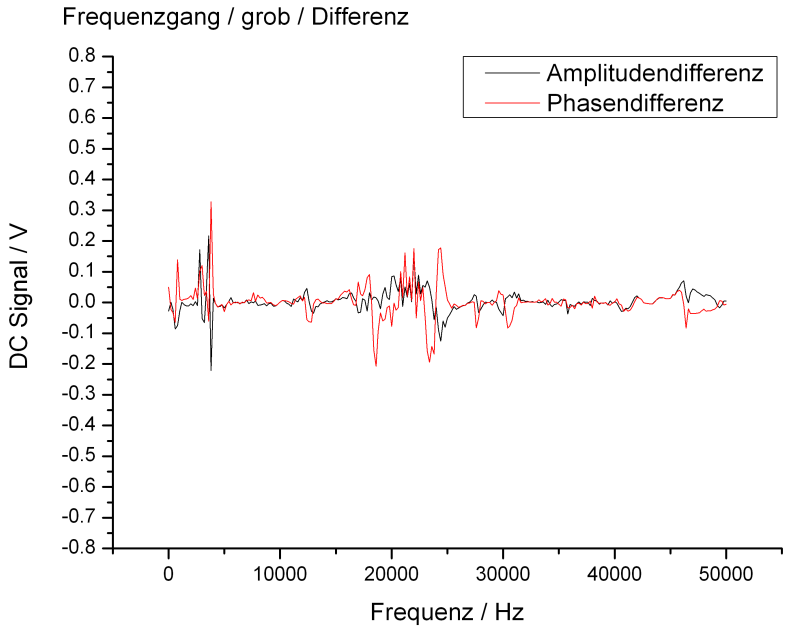


Abbildung 6.23 – Differenzspektrum der schwingenden Messspitze in- und ausserhalb einer Lösung.

Nachdem eine geeignete Frequenz gefunden wurde, kann diese als Parameter für die Abstandskontrolle verwendet werden. Abbildung 6.24 zeigt den gemessenen Signalverlauf des *Lock-In*-Verstärkers bei einer gewählten Frequenz von 24.2 kHz. Am Beispiel dieses Signalverlaufs wäre ein gemessener Wert von 1.1 V eine gute Ausgangsbasis, um einen konstanten Abstand zu erhalten. Wird die Messspitze auf eine neue Rasterposition gestellt, so muss zunächst das *Piezo-Drive*-Signal auf einen neuen Wert gestellt werden, sodass der *Lock-In*-Verstärker den Referenzwert liefert. Die Werte der am Raster angelegten Spannungen für den *Piezo-Drive* können zusätzlich dazu verwendet werden, um ein Oberflächenprofil zu erstellen.

Die verwendete Hardware für die Abstandskontrolle war zu ungenau und hätte durch präzisere und wesentlich teurere Positioniersysteme ersetzt werden müssen. Aus budgetären Gründen wurde beschlossen, die Weiterentwicklung des *SECM*-Systems einzustellen.

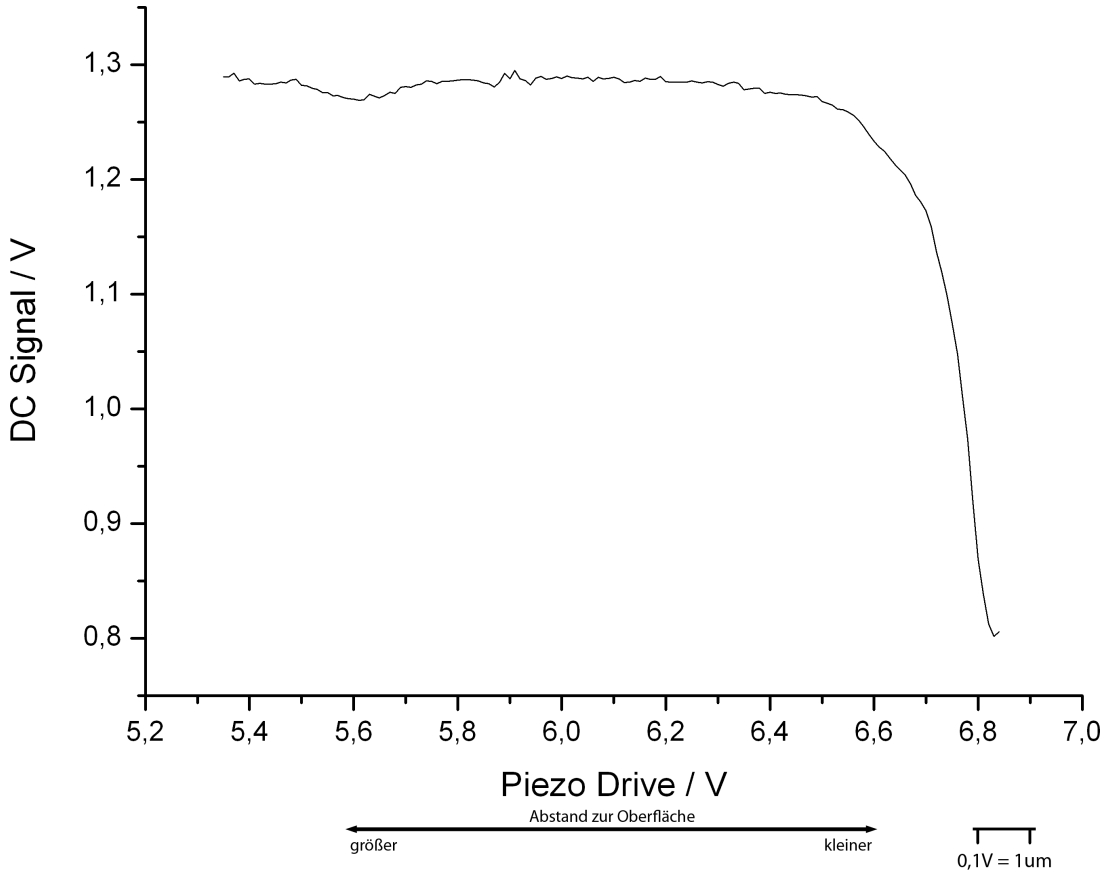


Abbildung 6.24 – Signalverlauf während einer schrittweisen Näherung zum Substrat. *Shear-Force*-Frequenz: 24.2 kHz

Abbildung 6.25 – Schematische Darstellung des verwendeten Testbandes.

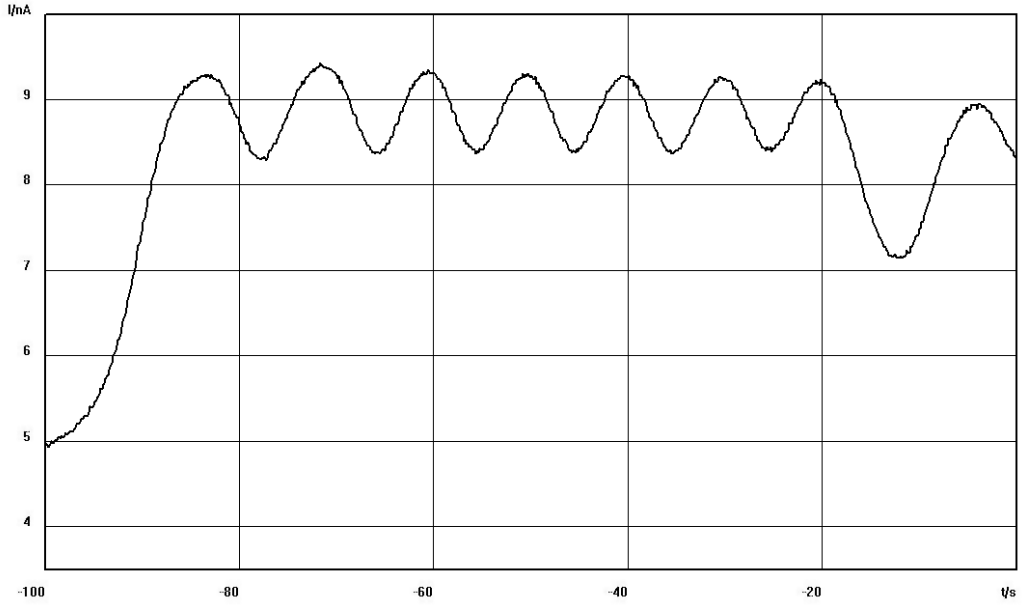


Abbildung 6.26 – Stromverlauf des *Nanobistat*-Potentiostats während eines linearen *Scans* über das Testband.

7 Zusammenfassung

Messaufgaben in der Elektrochemie sind auf Grund der zu untersuchenden Eigenschaften oft sehr komplex. In Kapitel 2 wurden die gängigsten Schnittstellen beschrieben, welche für die Steuerung von Messinstrumenten im Laufe der Jahre zum Einsatz gekommen sind. Der *General Purpose Interface Bus* war lange Zeit der *defacto* Standard und wird auch heute noch auf Grund seiner Robustheit sowie der langen «Lebensdauer» von Messgeräten verwendet. Sein hoher Preis bewegt die Hersteller jedoch zur Verwendung von moderneren und vor allem billigeren Bussystemen wie z. B. dem *Universal Serial Bus (USB)*. Andere Systeme setzen auf die *Peripheral-Component-Interconnect (PCI)*-Schnittstelle, um bessere Latenzzeiten erzielen zu können.

Der Aufbau von Messabläufen gestaltet sich auf Grund der mangelnden Interoperatibilität grundsätzlich schwierig. Einerseits sind die Protokolle zur Kommunikation mit dem Computer unterschiedlich, andererseits werden die im Messsystem vorhandenen Apparaturen über unterschiedliche Schnittstellen angesprochen. Dies stellt sowohl eine hardwaretechnische als auch eine softwaretechnische Hürde dar. In Kapitel 3 wurde versucht, die technischen Aspekte zu formalisieren, um einen Messablauf zu modellieren, dessen Ergebnisse bestmöglich verwertet werden können. Die Theorie der Messung spricht in diesem Zusammenhang über *sensorische* und *rationale Wahrnehmung*. Diese philosophisch betrachteten Begriffe für den Wahrnehmungsprozess wurden anhand von theoretischen Grundlagen formalisiert. Die korrekte und korrelierte *sensorische Wahrnehmung* ist eine grundlegende Vorausetzung, um eine vernünftige *rationale Wahrnehmung* zu ermöglichen. Der Zusammenhang zwischen unterschiedlichen Eigenschaften einer Messumgebung sowie die Messung auf diesen, wurde mit Hilfe von grafischen Elementen für die Formalisierung von Messprozessen methodisch dargestellt.

Abstraktion ist ein nahezu unentbehrliches Werkzeug, wenn es darum geht, komplexe Systeme überschaubar zu halten. Die Überschaubarkeit ist wiederum eine notwendige Voraussetzung, um komplexe Messabläufe korrekt zu koordinieren. In Kapitel 4 wurden architekturelle Elemente und softwaretechnische Aspekte erläu-

tert, die als Wegbereiter für das selbstentwickelte Softwaresystem Task++ dienen. Durch die Notwendigkeit, systemnahe Routinen zu erstellen, ist die Entscheidung auf C++ bei der Wahl der Programmiersprache gefallen. Da C++ für optimale Performance ausgerichtet ist, hat diese Sprache eine natürliche Nische in der Programmierung systemnaher Aufgaben gefunden. Ein Grund für die gute Performance ist das Fehlen von *Meta-Information*. Diese wird während der Laufzeit, im Gegensatz zu anderen Programmiersprachen wie z. B. Java oder C#, nicht gebraucht. Das Fehlen dieser Information ermöglicht jedoch keine Introspektion, was jedoch ein wichtiges Abstraktionswerkzeug ist. Die fehlende *Meta-Information* muss dem System durch Reflexion verfügbar gemacht werden. Der C++-Interpreter folgt dieser Idee, bei welcher Performanceeinbußen bewusst in Kauf genommen wird. Die Begründung liegt in der Tatsache, dass Flexibilität in Messaufgaben einen größeren Nutzen hat, als eine optimale Laufzeit. Mit Hilfe der *Task Description Language* wird eine Methode dargestellt, Messabläufe durch Tasks zu beschreiben. Dieser Mechanismus erlaubt eine weitere nützliche Abstraktionsmöglichkeit, um Messabläufe überschaubar zu halten und gezielt zu koordinieren.

Mit Hilfe der gewonnenen Erkenntnisse wurde ein Softwaresystem entwickelt, welches mit den speziell im Bereich der Elektrochemie gegebenen Anforderungen gut umgehen kann. Anwendbarkeit, Flexibilität und Erweiterbarkeit sind die Hauptziele, die durch Verwendung der Abstraktionskonzepte aus Kapitel 4 erreicht werden sollen. Durch Task++ ist es möglich, einen Messablauf, der in Form einer einfachen Skriptsprache dargestellt ist, einzulesen und auszuführen. Durch Deklaration von Eingabeelementen und deren Verknüpfung mit Variablen kann der Messablauf komfortabel über eine dynamisch erstellte Benutzeroberfläche parametrisiert werden. Die Verschachtelung von Tasks sowie die taskübergreifende Verknüpfung von Variablen wurde im Zeitrahmen dieser Arbeit nicht implementiert. Auch das *Graphical User Interface* stellt nur den Prototyp einer Implementierung dar und ist in keinster Weise mit anderen Systemen wie `NOVA` zu vergleichen.

Die Anwendung von Task++ wird anhand einiger Experimente in Kapitel 6 erläutert. Die Messung eines Arrays von Mikroelektroden stellt ein klassisches Beispiel einer alltäglichen Messaufgabe in der Elektrochemie dar, bei welcher verschiedene Apparaturen zum Einsatz kommen. Vor allem der Zeitgewinn durch das automatisierte Zusammenspiel der inkompatiblen Geräte führte bei diesem Experiment zu guten Ergebnissen. Bei der Simultanmessung von elektrochemischen Zellen ist hingegen vor allem die komplexe Verdrahtung ein Grund dafür, ein flexibles Messsystem einzusetzen. Solche Anforderungen führen in Programmen wie `LabVIEW`

leicht zu unüberschaubaren Konstrukten. Die Durchführung von Messungen im Fahrzeug ist ein Thema für sich. Obwohl die Grundaufgabe eng verwandt mit der Simultanmessung von elektrochemischen Zellen ist, gestaltet sich die Durchführung der mobilen Variante aus hardwaretechnischer Sicht etwas schwierig. Die stromhungrige Messeinrichtung muss permanent mit Energie versorgt werden und die sensiblen Geräte sind starken Erschütterungen ausgesetzt. Dieses Experiment führte zu weiteren Überlegungen und zur Entwicklung eines Messgerätes mit geringer Leistungsaufnahme, welches im Anhang B.1 beschrieben ist. Der Einsatz von Task++ zur Steuerung eines selbst entwickelten *SECM*-Systems konnte die in dieser Arbeit gestellten Anforderungen erfüllen. Die verwendete Hardware war für die Abstandskontrolle allerdings zu ungenau und hätte durch teurere Komponenten ersetzt werden müssen. Daher wurde beschlossen, die Eigenentwicklung eines *SECM*-Systems einzustellen.

A Task++-Referenz

A.1 Verwendung und Konfiguration

Der Kern des Task++-Systems besteht aus einer ausführbaren Datei `task++.exe` sowie 5 weiteren dynamischen Bibliotheksdateien, welche jeweils einen bestimmten Bereich implementieren. Die Aufgabe der Dateien wird in Tabelle A.1 kurz erläutert; Abbildung A.1 zeigt die Datei- und Verzeichnisstruktur des Systems.

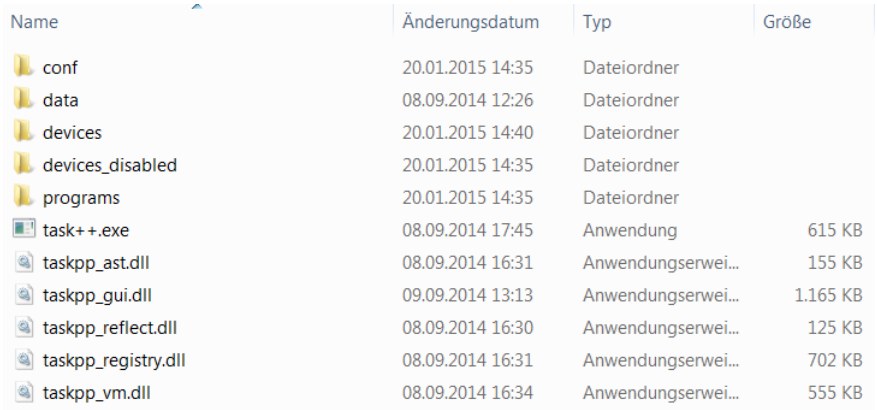


Abbildung A.1 – Datei- und Verzeichnisstruktur des Task++-Systems.

Die Verzeichnisse `devices` und `devices_disabled` beinhalten die Gerätemodule in aktivem bzw. inaktivem Zustand. Das `conf` Verzeichnis enthält eine Initialisierungsdatei namens `config.ini`, mit dessen Hilfe gewisse Voreinstellungen an den Gerätemodulen vorgenommen werden können. Listing A.1 zeigt eine mögliche Konfigurationsdatei, mit deren Hilfe dem Task++-System mitgeteilt wird, dass die Geräte *Agilent 34970A* und *Agilent 33120A* auf dem Kanal 9 bzw. 10 im Bussystem angeschlossen sind.

Listing A.1 – Konfigurationsdatei `config.ini` für Voreinstellungen an den Geräten.

```
[agilent34970a]
channel=9

[agilent33120a]
channel=10
```

Tabelle A.1 – Beschreibung der Dateien des Task++-Systems.

Dateiname	Beschreibung
`task++.exe`	Ausführbare Datei des Task++-Systems. Wird kein Programm als Parameter übergeben, so wird die Bedienoberfläche für die Komposition von Messroutinen gestartet.
`taskpp_ast.dll`	Enthält die Implementierung für den Aufbau des *Abstrakten Syntaxbaumes*.
`taskpp_gui.dll`	Enthält die Implementierung für die Generierung von Eingabeelementen.
`taskpp_reflect.dll`	Stellt den Reflexionsmechanismus bereit.
`taskpp_registry.dll`	Verwaltet die dynamisch geladenen Gerätemodule und stellt *Meta-Information* zur Verfügung.
`taskpp_vm.dll`	Enthält die Implementierung der *Virtuellen Maschine* und ist für die Ausführung des Programms zuständig.

Der Gerätename wird, wie vom INI-Format vorgegeben, in eckigen Klammern eingeschlossen. Als Schlüssel im Schlüssel-Werte-Paar kann jeder Name verwendet werden, für welchen es eine entsprechende *Setter*-Funktion im Modul gibt. Im erwähnten Beispiel enthält also jedes der beiden Gerätemodule eine Funktion mit dem Namen `setChannel`, um den Buskanal einzustellen, auf welchen das entsprechende Gerät kommuniziert. Der Bustreiber, welcher vom spezifischen Gerätemodul verwendet wird, muss zum Task++-System separat installiert werden. Das `data`-Verzeichnis enthält Dateien, welche mit der `printf`-Funktion erstellt werden bzw. in welche geschrieben wird. Das `programs`-Verzeichnis enthält die Programme, welche auf textueller oder visueller Basis erstellt wurden.

A.2 Erstellung eines Gerätemoduls

Die Erstellung eines Gerätemoduls ist nur in C bzw. C++ möglich. Dabei muss die vom Task++-System bereitgestellte *Header-Bibliothek* `taskpp_reflect.h` eingebunden werden. Diese Bibliothek stellt die Makrobefehle bereit, welche im Abschnitt 5.2.3 vorgestellt wurden. In Listing A.3 wird ein Gerätemodul mit dem Namen `agilent34970a` erstellt, welches über den *General Purpose Interface Bus* kommuniziert. Dazu muss auch die *Header-Datei* `ieee-c.h` der Treiberbibliothek inkludiert werden. Der Bustreiber muss in diesem Fall separat zum Task++-System installiert werden, da es sich um eine *dynamische Bindung* der Bibliothek handelt. Die Funktion `send(const char *cmd)` in Zeile 12 wird nur modulintern verwendet. Die beiden

Funktionen `measure_current_dc` und `setChannel` werden mit Hilfe der bereitgestellten Makros so aufbereitet, dass sie vom Task++-System verwendet werden können und haben, wie in Abschnitt 5.2.2 beschrieben, den selben Eingangsparameter- und Rückgabetyp.

Listing A.2 – Erstellung eines Moduls für das Gerät *Agilent 34970A* mit Hilfe der vom Task++-System bereitgestellten Bibliothek für die Erzeugung von *Meta-Objekten*.

```
#include <string>
#include <sstream>
#include <iostream>
#include "agilent34970a.h"
#include "ieee-c.h"
#include "taskpp_reflect.h"

using namespace taskpp::reflect;

int CHANNEL = 8;

void send(const char *cmd) {
    unsigned int status;
    ieee488_send (CHANNEL, cmd, &status);
    if (status != 0) throw DeviceError("Device not responding!");
}

Value measure_current_dc(Function *f) {
    stringstream ss;
    int ch = f->getValue(0).get<int>();
    ss << "meas:curr:dc? (@" << ch << ")";
    send(ss.str().c_str());
    return Value(RealValue<string>(read()));
}

Value setChannel(Function *f) {
  CHANNEL = f->getValue(0).get<int>();
  return Value();
}


BEGIN_DEFINITION("agilent34970a", "Agilent 34970A - Data
   ↪ Aqcuisition / Switch Unit")

  FUNCTION("setChannel", setChannel, TASKPP_VOID)
    NAME("Set channel")
    DESCRIPTION("Set the GPIB channel which this Device is
       ↪ listening to")
    ATTRIBUTE("channel_nr", TASKPP_INT)

  FUNCTION("measure_current_dc", measure_current_dc,
     ↪ TASKPP_STRING)
    NAME("Measure DC current")
    ATTRIBUTE("channel number", TASKPP_INT)

END_DEFINITION
```

A.3 Syntax

A.3.1 Variablen

Variablen bestehen aus einer alphanumerischen Zeichenkette ohne führender Ziffer und einem vorangestellten `$`-Zeichen. Es gibt keine Typen und sie werden vom System bei der ersten Verwendung initialisiert.

```
$eineVariable = 0;
$_eineVariable = 1;
$eine123Variable = 2;
```

A.3.2 Kontrollstrukturen

Als Kontrollstrukturen stehen zum einen bedingte Anweisungen mit Verzweigungsmöglichkeit, zum anderen 3 verschiedene Schleifentypen zur Verfügung. Für eine bedingte Anweisung bzw. *If-Statement* wird das Schlüsselwort `if` verwendet. Darauf folgt ein Ausdruck in runden Klammern, welcher zu einem Wahrheitswert evaluiert. Die bedingten Anweisungen werden nachstehend von geschwungenen Klammern – `{` und `}` – eingeschlossen. Optional kann mit den Schlüsselwörtern `elseif` und `else` ein (bzw. mehrere) alternativer Anweisungsblock angegeben werden.

```
if($messwert > 10) {
  // bedingte Anweisungen
} else {
  // alternative Anweisungen
}

if ($messwert > 20) {
  // ...
} elseif($messwert > 10) {
  // ...
}
```

Als Schleifen kann die *For*-Schleife, die *While*-Schleife sowie die *Do-While*-Schleife verwendet werden.

```
for($i = 0; $i < 10; $i++) {
  // ...
}

while($i < 10) {
  // ...
}

do {
  // ...
} while($i < 10);
```

A.3.3 Gerätedeklaration

Ein Gerät wird durch Angabe des Schlüsselwortes `device`, einer Variablen sowie einer in runden Klammern eingeschlossene Zeichenkette, die das Gerät identifiziert (siehe Anhang A.2), deklariert. Das Deklarations*statement* wird mit einem Semicolon abgeschlossen.

```
device $geraet("GeraeteId");
```

A.3.4 Geräte- und Systemfunktionen

Der Aufruf einer Gerätefunktion erfolgt durch Angabe des Funktionsnamens, gefolgt von einer optionalen Liste von Argumenten, welche in runden Klammern eingeschlossen wird. Zur Identifikation des Gerätes wird die Variable, welche auf das Gerät zeigt, gefolgt von einem Punkt, vorangestellt.

```
device $geraet("agilent34970a");
$geraet.setChannel(7);
$result = $geraet.measure_current_dc(101);
```

Systemfunktionen werden ohne vorangestellten Qualifizierer aufgerufen:

```
wait(5000);
$val = $geraet.measure_current_dc(101);
print("Gemessener Gleichstrom: " + $val);
```

A.4 Grammatik

A.4.1 Grammatik für die Taskstruktur

⟨*task*⟩ → `task $ (` ⟨*quoted_string*⟩ `) {` ⟨*gui*⟩ ⟨*program*⟩ `}`

⟨*gui*⟩ → siehe Grammatik in Abschnitt A.4.2

⟨*program*⟩ → siehe Grammatik in Abschnitt A.4.3

A.4.2 Grammatik für die Deklaration von Eingabeelementen

⟨*gui*⟩ → `gui {` ⟨*gui_elements*⟩ `}`
| ϵ

⟨*gui_elements*⟩ → ⟨*gui_element*⟩
| ⟨*parameter_decl*⟩
| ⟨*gui_elements*⟩

⟨*gui_element*⟩ → ⟨*element_type*⟩ `(` ⟨*quoted_string*⟩ `)` `{` ⟨*gui_elements*⟩ `}`

⟨*element_type*⟩ → `tab`
| `group`

⟨*parameter_decl*⟩ → `parameter` `(` ⟨*variable_identifier*⟩ `,` ⟨*quoted_string*⟩ `,` ⟨*default_value*⟩ `)` ⟨*type_decl*⟩ ⟨*element_decl*⟩

⟨*type_decl*⟩ → `is` ⟨*type*⟩ ⟨*map*⟩
| `is` ⟨*type*⟩ ⟨*range*⟩
| `is` ⟨*type*⟩

⟨*element_decl*⟩ → `as` ⟨*widget*⟩ ⟨*widget_params*⟩
| ϵ

⟨*widget*⟩ → `textfield`
| `slider`
| `combo`

⟨*widget_params*⟩ → `(` ⟨*constant_list*⟩ `)`
| ϵ

⟨*constant_list*⟩ → ⟨*constant*⟩ `,` ⟨*constant_list*⟩
| ⟨*constant*⟩

⟨*type*⟩ → ⟨*numeric_type*⟩
| `string`

⟨*numeric_type*⟩ → `int`
| `double`

⟨*range*⟩ → `[` ⟨*constant*⟩ `,` ⟨*constant*⟩ `]`

⟨*map*⟩ → `{` ⟨*map_item_list*⟩ `}`

⟨*map_item_list*⟩ → ⟨*map_item*⟩
| ⟨*map_item*⟩ `,` ⟨*map_item_list*⟩

⟨*map_item*⟩ → ⟨*quoted_string*⟩ `=>` ⟨*constant*⟩

⟨*default_value*⟩ → ⟨*constant*⟩
| ⟨*quoted_string*⟩

⟨*quoted_string*⟩ → siehe Regel in Grammatik in Abschnitt A.4.3

⟨*variable_identifier*⟩ → siehe Regel in Grammatik in Abschnitt A.4.3

⟨*constant*⟩ → siehe Regel in Grammatik in Abschnitt A.4.3

A.4.3 Grammatik für die Task-Programmierung

⟨*program*⟩ → ⟨*stmt_list*⟩

⟨*stmt_list*⟩ → ⟨*stmt*⟩ ⟨*stmt_list*⟩
| ϵ

⟨*stmt*⟩ → ⟨*device_decl*⟩
| ⟨*selection_stmt*⟩
| ⟨*iteration_stmt*⟩
| ⟨*control_stmt*⟩
| ⟨*expression_stmt*⟩

⟨*device_decl*⟩ → `device` ⟨*variable_identifier*⟩ `(` ⟨*quoted_string*⟩ `)`

⟨*selection_stmt*⟩ → `if (` ⟨*expression*⟩ `) {` ⟨*stmt_list*⟩ `}` ⟨*else_stmt*⟩

⟨*else_stmt*⟩ → `elseif (` ⟨*expression*⟩ `) {` ⟨*stmt_list*⟩ `}` ⟨*else_stmt*⟩
| `else {` ⟨*stmt_list*⟩ `}`
| ϵ

⟨*iteration_stmt*⟩ → `while (` ⟨*expression*⟩ `) {` ⟨*stmt_list*⟩ `}`
| `do {` ⟨*stmt_list*⟩ `} while (` ⟨*expression*⟩ `) ;`
| `for (` ⟨*expression_stmt*⟩ `,` ⟨*expression_stmt*⟩ `,` ⟨*expression*⟩ `) {` ⟨*stmt_list*⟩ `}`

⟨*control_stmt*⟩ → `continue`
| `halt`
| `break`

⟨*expression_stmt*⟩ → ⟨*expression*⟩ `;`
| ϵ

⟨*expression*⟩ → ⟨*assignment_expression*⟩
| ⟨*logical_or_expression*⟩

⟨*assignment_expression*⟩ → ⟨*unary_expression*⟩ ⟨*assignment_operator*⟩ ⟨*logical_or_expression*⟩

⟨*logical_or_expression*⟩ → ⟨*logical_and_expression*⟩
| ⟨*logical_and_expression*⟩ `||` ⟨*logical_or_expression*⟩

⟨*logical_and_expression*⟩ → ⟨*equality_expression*⟩
| ⟨*equality_expression*⟩ `&&` ⟨*logical_and_expression*⟩

⟨*equality_expression*⟩ → ⟨*relational_expression*⟩
| ⟨*relational_expression*⟩ ⟨*equality_operator*⟩ ⟨*equality_expression*⟩
| ⟨*relational_expression*⟩ ⟨*relation_operator*⟩ ⟨*equality_expression*⟩

⟨*relational_expression*⟩ → ⟨*additive_expression*⟩
| ⟨*additive_expression*⟩ ⟨*relation_operator*⟩ ⟨*relational_expression*⟩

⟨*additive_expression*⟩ → ⟨*multiplicative_expression*⟩
| ⟨*multiplicative_expression*⟩ ⟨*additive_operator*⟩
⟨*additive_expression*⟩

⟨*multiplicative_expression*⟩ → ⟨*unary_expression*⟩
| ⟨*unary_expression*⟩ ⟨*multiplicative_operator*⟩
⟨*multiplicative_expression*⟩

⟨*unary_expression*⟩ → ⟨*primary_expression*⟩
| ⟨*primary_expression*⟩ (⟨*argument_expression_list*⟩)
| ⟨*primary_expression*⟩ . ⟨*identifier*⟩
| ⟨*primary_expression*⟩ ⟨*inc_dec_operator*⟩

⟨*argument_expression_list*⟩ → ⟨*logical_or_expression*⟩
| ⟨*logical_or_expression*⟩ , ⟨*argument_expression_list*⟩

⟨*primary_expression*⟩ → ⟨*function_identifier*⟩
| ⟨*variable_identifier*⟩
| ⟨*constant*⟩
| (⟨*expression*⟩)

⟨*function_identifier*⟩ → ⟨*identifier*⟩

⟨*variable_identifier*⟩ → $ ⟨*identifier*⟩

⟨*identifier*⟩ → ⟨*alpha*⟩ ⟨*identifier_numeric*⟩
| _ ⟨*identifier_numeric*⟩

⟨*identifier_numeric*⟩ → ⟨*alphanum*⟩ ⟨*identifier_numeric*⟩
| _ ⟨*identifier_numeric*⟩
| ϵ

⟨*constant*⟩ → ⟨*double*⟩
| ⟨*integer*⟩
| ⟨*quoted_string*⟩

⟨*quoted_string*⟩ → " ⟨*string*⟩ "

⟨*assignment_operator*⟩ → =
| *=
| /=
| +=
| -=
| %=

⟨*equality_operator*⟩ → ==
| !=

⟨*relation_operator*⟩ → <=
| >=
| <
| >

⟨*additive_operator*⟩ → +
| -

⟨*multiplicative_operator*⟩ → *
| /
| %

⟨*inc_dec_operator*⟩ → ++
| --

⟨*alpha*⟩ → a-z
| A-Z

⟨*alphanum*⟩ → ⟨*alpha*⟩
| 0-9

⟨*string*⟩ → `*(lit("\\"")|(char_ - '"'))`[1]

⟨*double*⟩ → `boost::spirit::double_`[2]

⟨*integer*⟩ → `boost::spirit::int_`[2]

[1] C++ Code für das Parsen einer (in Anführungszeichen eingeschlossenen) Zeichenkette in `Boost::Spirit` [51].
[2] Interner Parser von `Boost::Spirit` [51].

A.5 Programmbeispiel: Simultanmessung elektrochemischer Zellen

Listing A.3 – Programm für die Simultanmessung elektrochemischer Zellen.

```
task $ph("pH - Messung") {

  gui {
    tab("Delays") {
      group("Strom und Potential") {
      parameter($delayBeforeCurrent, "Delay vor Strommessung",
          ↪ 1000) is int[100,10000] as slider;
      parameter($delayBeforeVolt, "Delay vor Potentialmessung",
          ↪ 1000) is int[100,10000] as slider;
      }
      group("pH Messung") {
      parameter($delayBeforeVoltPH, "Delay vor Leer-pH-
          ↪ Potentialmessung in Sekunden", 10) is int[1,240] as
          ↪ slider;
      parameter($delayBetweenPH, "Delay zwischen Leermessung und
          ↪ pH-Messung in Sekunden", 6) is int[1,240] as slider;
      }
      parameter($delayProZyklus, "Delay vor jedem Zyklus in
          ↪ Sekunden",5) is int[1,300] as slider;
    }

    tab("Kanaleinstellungen") {
      group("Messkarte") {
        parameter($slotMeasurementCard, "Slot der Messkarte", 1)
            ↪ is int[1,3] as combo;
        parameter($channelVolt, "Kanalnummer der Potentialmessung
            ↪ ", 1) is int[1,20] as combo;
        parameter($channelPHVolt, "Kanalnummer der PH -
            ↪ Potentialmessung", 2) is int[1,20] as combo;
        parameter($channelCurrent, "Kanalnummer der Strommessung"
            ↪ , 21) is int[21,22] as combo;
      }
      group("Schaltkarte") {
        parameter($slotSwitchCard, "Slot der Schaltkarte", 2) is
            ↪ int[1,3] as combo;
      }

    }
    tab("Zelleinstellungen") {
      parameter($cells, "Anzahl der Zellen", 4) is int[1,4] as
          ↪ combo;
      parameter($pHCells, "Anzahl der pH-Zellen", 2) is int[1,4]
          ↪ as combo;
      parameter($chCellStart, "Startkanal des Zellpotentials", 1)
          ↪  is int[1,10] as combo;
      parameter($chPHStart, "Startkanal des pH-Potentials", 5) is
          ↪  int[1,10] as combo;
    }

  }

  device $daq("agilent34970a");
```

```
38
39  // Initialization part
40  $daq.reset();
41  $daq.route_open(299);
42  $filename = "pHMessung-"+timestamp()+".txt";
43  $header = "";
44  for ($i = 0; $i < $cells; $i++) {
45    $z = ($i+1);
46    if ($z > 1) $header += ", ";
47    $header += "Strom #" + $z + ", Potential #" + $z;
48    // also perform pH Measurement
49    if ($i < $pHCells) {
50      for ($j = 0; $j < 2; $j++) {
51        $header += ", pH" + ($j + 1) + " #" + $z;
52      }
53    }
54  }
55  fprint($filename, $header);
56  $dataline = "";
57  // \\ Initialization part
58
59  while(1) {
60
61  $dataline = "";
62  $dataline += timestamp();
63  print(now());
64  for ($i = 0; $i < $cells; $i++) {
65    for ($ch = 1; $ch <= 10; $ch++) {
66      $chOpen = $slotSwitchCard * 100 + $ch;
67      $chClose = $slotSwitchCard * 100 + $ch + 10;
68      // Make before break!
69      $daq.route_close($chClose);
70      $daq.route_open($chOpen);
71
72    }
73    print("+------------------------------------------------+");
74    print("|              Messung fuer Zelle #" + ($i + 1) + "
       ↪             |");
75    print("+------------------------------------------------+");
76    $chMeas = $slotSwitchCard * 100 + $i+1;
77    $chSwitch = $slotSwitchCard * 100 + $i + 11;
78
79    $tCurr = $slotMeasurementCard * 100 + $channelCurrent;
80    $tVolt = $slotMeasurementCard * 100 + $channelVolt;
81    $tVoltPH = $slotMeasurementCard * 100 + $channelPHVolt;
82    print("Close channel " + $chMeas);
83    $daq.route_close($chMeas);
84    print("Open channel " + $chSwitch);
85    $daq.route_open($chSwitch);
86
87    print("Strommessung auf Kanal " + $channelCurrent + " in slot
       ↪  #" + $slotMeasurementCard + "(@" + $tCurr + ")");
88    wait($delayBeforeCurrent);
89    $curr = $daq.measure_current_dc($tCurr);
90    $dataline += ", " + $curr;
91    print("Potentialmessung auf Kanal " + $channelVolt + " in
       ↪ slot #" + $slotMeasurementCard + "(@" + $tVolt + ")");
92    wait($delayBeforeVolt);
93    $volt = $daq.measure_voltage_dc($tVolt);
```

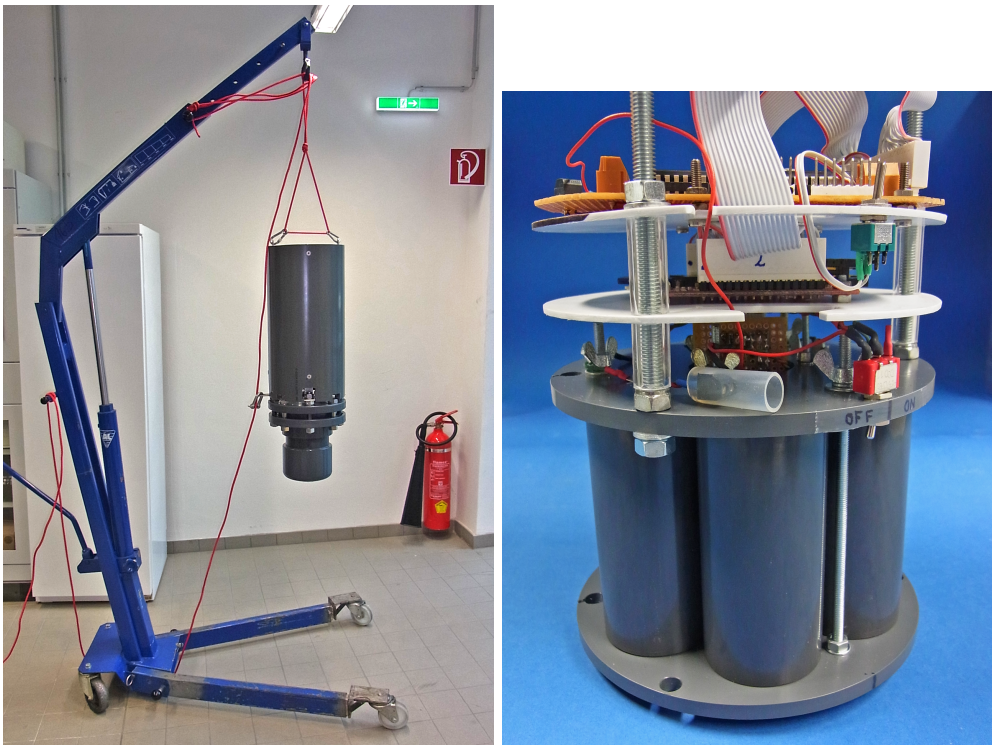
```
94      $dataline += ", " + $volt;
95      print("Close channel " + $chSwitch);
96      $daq.route_close($chSwitch);
97      print("Open channel " + $chMeas);
98      $daq.route_open($chMeas);
99
100     // also perform pH Measurement
101     if ($i < $pHCells) {
102       for ($j = 0; $j < 2; $j++) {
103         $chMeas = $slotSwitchCard * 100 + $chPHStart + $i + $j;
104
105         print("Close channel " + $chMeas);
106         $daq.route_close($chMeas);
107         print("pH Potentialmessung #" + ($j+1) + " auf Kanal " +
              ↪ $channelPHVolt + " in slot #" +
              ↪ $slotMeasurementCard + "(@" + $tVoltPH + ")");
108         wait($delayBeforeVoltPH*1000);
109         $volt = $daq.measure_voltage_dc($tVoltPH);
110         wait($delayBetweenPH*1000);
111
112         $volt = $daq.measure_voltage_dc($tVoltPH);
113         $dataline += ", " + $volt;
114         print("Open channel " + $chMeas);
115         $daq.route_open($chMeas);
116       }
117     }
118     print("");
119   }
120   fprint($filename, $dataline);
121   wait($delayProZyklus*1000);
122   print("");
123   print("");
124   }
125 }
```

B Datalogging und Auswertung

B.1 DataLogger

In vielen Situationen ist der Einsatz von großen Apparaturen schwierig oder nahezu unmöglich. Für die Korrosionsmessung im Fahrzeug aus Abschnitt 6.3 wurde durch hohen Aufwand die Messeinrichtung adaptiert, sodass diese den herausfordernden Bedingungen standhalten würde. Vor allem die Gewährleistung der Energieversorgung ist in Umgebungen wie dieser eine delikate Aufgabe. Während man im Fahrzeug die Möglichkeit hat, die Stromzufuhr mit Hilfe der Autobatterie, der Lichtmaschine oder ein nahe dem Fahrzeug befindlichem Stromnetz, aufrecht zu erhalten, so gibt es auch Situationen, in denen nur reiner Batteriebetrieb möglich ist. Die Verwendung eines Messgerätes samt Laptop zur Steuerung und Aufbewahrung von Messdaten ist in vielen Umgebungen unvorstellbar. In Zusammenarbeit mit Professor PAUL LINHARDT wurde ein Messgerät entwickelt, welches speziell auf Robustheit und niedrigen Stromverbrauch ausgerichtet ist.
Eine angestrebte Verwendung ist eine mit diesem System ausgestattete Boje, welche über einen längeren Zeitraum, in einem Gewässer, das Korrosionsverhalten von Werkstoffproben für die Wasserkrafttechnologie aufzeichnen kann. Angeregt wird dieses Experiment durch die Beobachtung, dass es trotz bedachter Materialauswahl, in manchen Fällen, zu Korrosionsschäden an Turbinenkomponenten kommt, die nicht unmittelbar erklärbar erscheinen. Untersuchungen an der TECHNISCHEN UNIVERSITÄT WIEN haben gezeigt, dass diese Schäden auf die Besiedlung von Mikroorganismen (Bakterien, Pilze, Algen usw.) zurückzuführen sind. Durch diesen sogenannten Biofilm kann – unter bestimmten Voraussetzungen – die Beständigkeit des Metalls durch hohe Oxidationskräfte überfordert werden und es kommt zu Korrosion. Das Phänomen dieser biochemischen Vorgänge wird als *Ennoblement* bezeichnet und kann indirekt als elektrochemisches Potential gemessen werden. Abbildung B.1(a) zeigt die MICRA-Boje[1] an einem Kran aufgehängt und zur Auslagerung ins Gewässer bereit. Das Messsystem befindet sich im wasserdichten Kopf der Boje und ist in Abbildung B.1(b) dargestellt.

[1] MICRA-Buoy (Microbially Influenced Risk Assessment Buoy).



(a) (b)

Abbildung B.1 – Die Boje ist an einem Kran aufgehängt und für die Auslagerung ins Gewässer bereit (a). Die Elektronik für den Messablauf befindet sich im wasserdichten Kopf von MICRA-Buoy (b).

Der Kern des Gerätes basiert auf einen Mikrocontroller, welcher eine Peripherie mit mehreren Komponenten ansteuert bzw. mit dieser kommuniziert. Der Grundaufbau des Systems wird in Abbildung B.2 dargestellt und besteht aus folgenden Elementen:

Mikrocontroller: Das Kernstück des Systems ist ein 8-Bit-Mikrocontroller des Typs *PIC18F67J50* von MICROCHIP TECHNOLOGY, welcher mit den notwendigen Modulen ausgestattet ist, um mit der Peripherie zu kommunizieren bzw. diese zu steuern.

A/D Konverter: Zwar beinhaltet der Mikrocontroller ein *ADC*-Modul, jedoch ist dieses mit einer 10-Bit-Genauigkeit für eine hochauflösende Messung von Korrosionsströmen nicht geeignet.
Der verwendete Analog/Digital-Konverter des Typs *AD7712AN* verfügt über eine Samplingtiefe von 24 Bit und kann somit auch Ströme im Nanobereich erfassen.

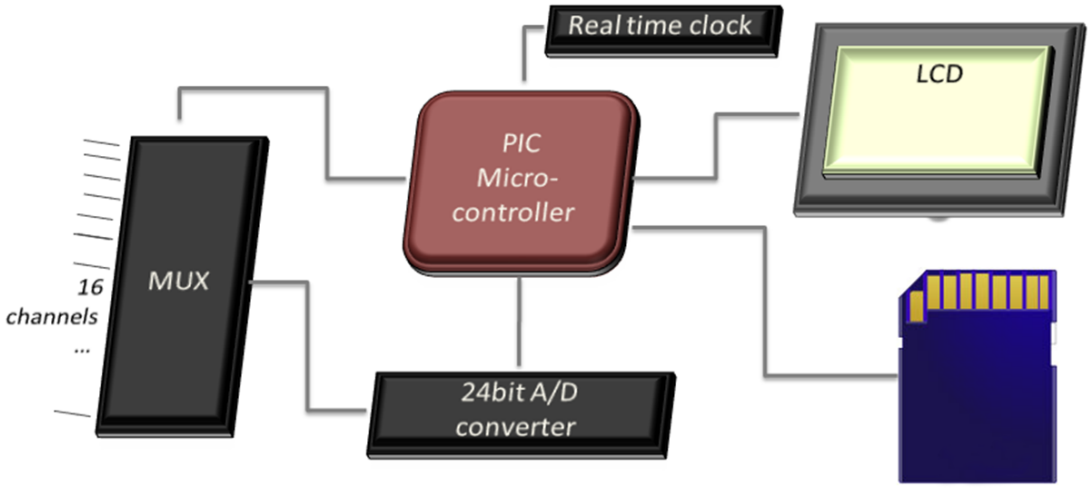


Abbildung B.2 – Systemaufbau des Datenloggers für die Langzeitmessung von Korrosionsströmen.

Multiplexer: Über einen 16-Kanal-Multiplexer des Typs *ADG506A* gelangt das Messsignal zum Analog/Digital-Konverter. Über 4 I/O-Pins des Mikrocontrollers wird einer der 16 Eingänge adressiert und ausgewählt.

SD Slot: Über das *Serial-Peripheral-Interface (SPI)*-Modul werden die gemessenen Daten auf einer SD-Karte gespeichert, bevor das System bis zum nächsten Messzyklus in den Energiesparmodus wechselt.

LC-Display: Ein LC-Display des Typs *Nokia3310*, welches ebenfalls mit dem *Serial Peripheral Interface* Modul des Mikrocontrollers kommuniziert, zeigt den aktuellen Status des Systems an.

Echtzeituhr: Mit Hilfe einer Echtzeituhr des Typs *DS1306* können die Datensätze mit einem exakten Zeitstempel versehen werden.

Für die MICRA-Boje-Variante des Messsystems wurde auf die Verwendung des LC-Displays und der Echtzeituhr verzichtet. Datensätze werden nur mit einem relativen Zeitstempel versehen, für welchen das interne *Timer*-Modul vom Mikrocontroller ausreichend ist. Über eine Konfigurationsdatei auf der SD-Karte können Einstellungen – Anzahl der zu messenden Kanäle, Wartedauer zwischen zwei Messzyklen usw. – vorgenommen werden.

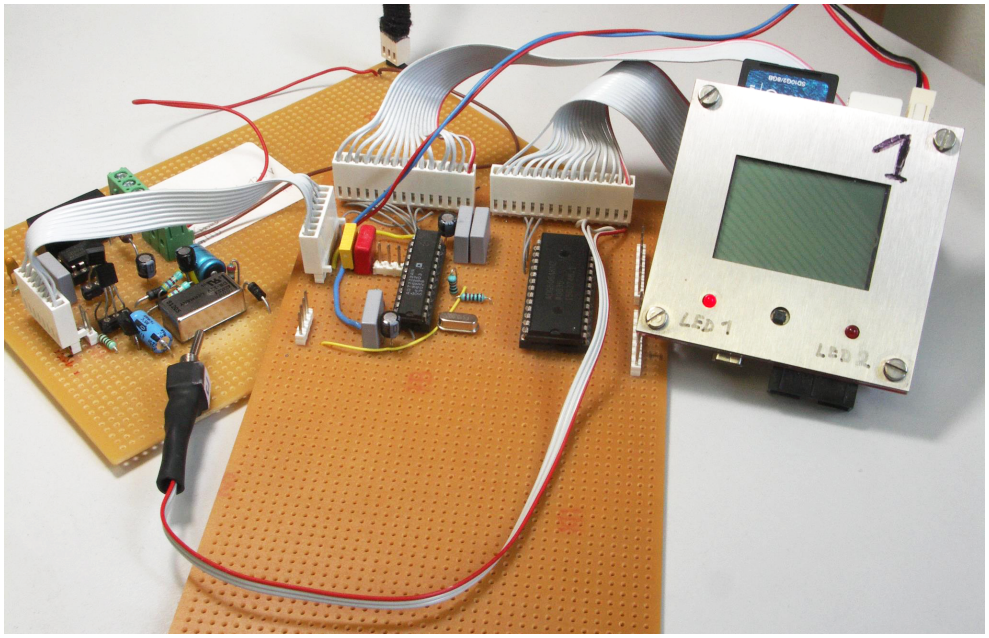


Abbildung B.3 – Die Entwicklung des Prototyps vom `DataLogger`.

B.2 DataAnalyzer

Bei elektrochemischen Messungen sammelt sich mit der Zeit eine große Menge an Daten an. Je nach Anwendungsfall sind Messungen über einen längeren oder kürzeren Zeitraum erforderlich, um das Geschehen an den gemessenen Proben gut zu verstehen. Auch während eines langen Zeitfensters ist eine detaillierte Auflösung nötig, um eventuelle, plötzlich eintretende Ereignisse gut analysieren und von Artefakten unterscheiden zu können.

Die Auswertung von großen Datensätzen ist oft eine mühsame Angelegenheit. Der Umgang mit Auswertungssoftware wie `Origin` oder `Microsoft Excel` erfordert etwas Routine und die Daten müssen zunächst aufbearbeitet werden, um den Sachverhalt korrekt darzustellen. In Zusammenhang mit dem Experiment der Korrosionsmessung im Fahrzeug aus Abschnitt 6.3 wurde ein Tool mit dem Namen `DataAnalyzer` entwickelt, mit welchem ein intuitiver Ablauf von Importieren von Daten, Darstellung und Auswertung möglich ist. Abbildung B.4 zeigt das Programmfenster inklusive Darstellung der Daten von zwei Messstellen im ausgewählten Zeitraum samt der berechneten Korrosionskennwerte.
Die Verwendung der Software ist verhältnismäßig intuitiv. Zu Beginn wird eine neue Messung erstellt, worauf ein erster Datensatz importiert werden muss. Aus der ersten Datenmenge wird die Anzahl der Messkanäle sowie deren Beschriftung ermittelt. Danach können zu jedem Zeitpunkt weitere Messdaten hinzugefügt wer-

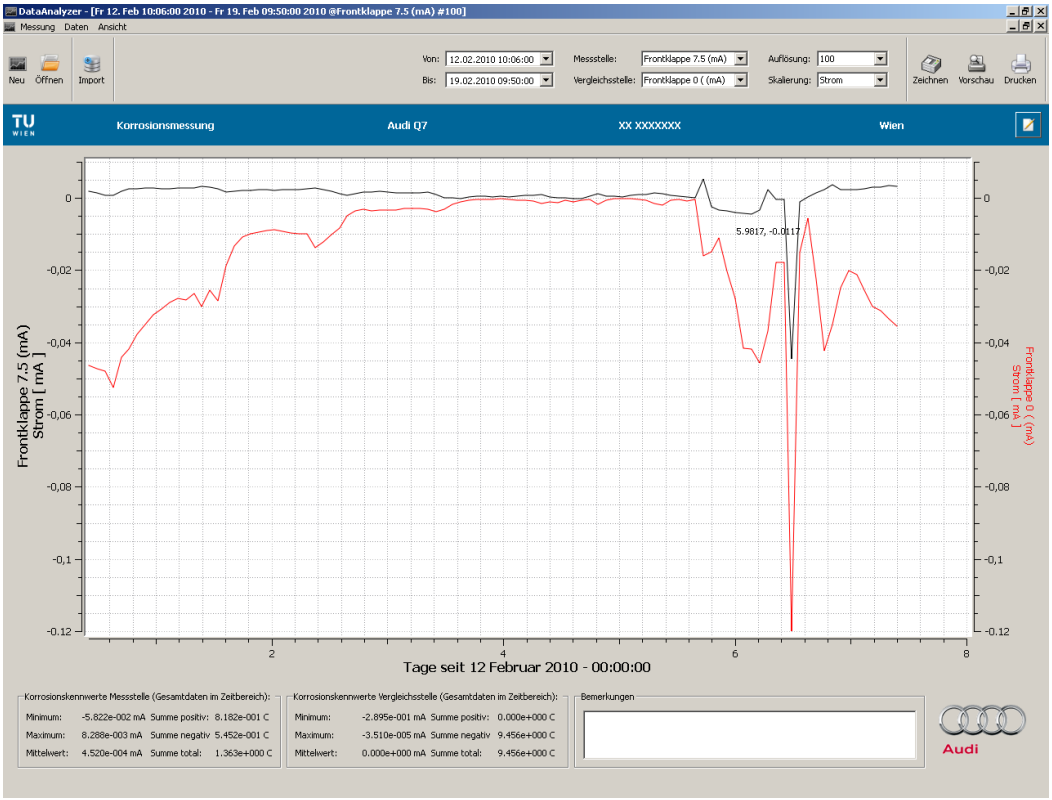
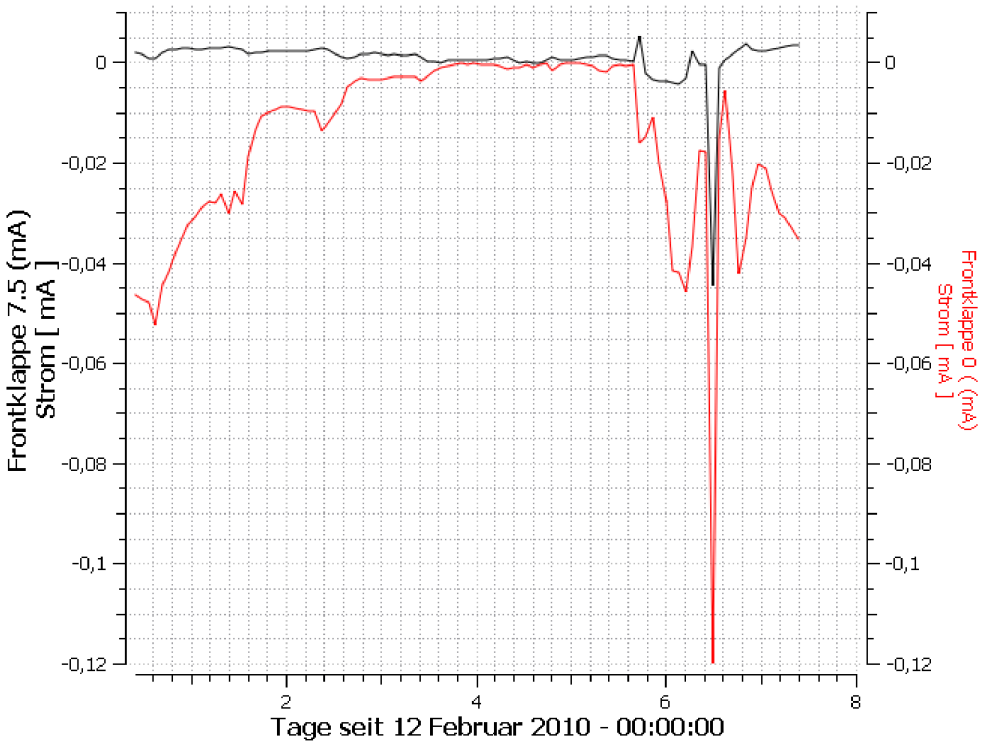


Abbildung B.4 – Datenanalysesoftware `DataAnalyzer` für die Auswertung von Korrosionsströmen.

den, wobei die Anazahl der Kanäle in späterer Folge konstant bleiben muss. Zur Anzeige kann entweder eine Messreihe alleine ausgewählt oder mit einer zweiten Messreihe verglichen werden. Die Kombination aus ausgewähltem Zeitraum und Auflösung bestimmt die Auswertungsdauer der Messdaten. Will man sich nur einen groben Überblick verschaffen, so ist es möglich, die Wartezeit zur Berechnung der Korrosionskennwerte durch Auswahl einer niedrigeren Auflösung zu verkürzen. Zum Schluss kann für die Archivierung ein Dokument, wie in Abbildung B.5 dargestellt ist, generiert und ausgedruckt werden.

Korrosionsmonitoring

Messung:	Korrosionsmessung	**Auflösung:**	100
Messstelle:	Frontklappe 7.5 (mA)	**Skalierung:**	Strom (mA)
Vergleichsstelle:	Frontklappe 0 ((mA)		
Zeitraum:	12 Februar 2010 - 10:06:00 - 19 Februar 2010 - 09:50:00		
Bemerkungen:			



Korrosionsmessdaten (Frontklappe 7.5 (mA))

berechnet aus den vollständigen Originaldaten im ausgewählten Zeitraum

Minimum:	-5.822e-002 mA	**Summe positiv:**	8.182e-001 C
Maximium:	8.288e-003 mA	**Summe negativ:**	5.452e-001 C
Mittelwert:	4.520e-004 mA	**Summe gesamt**	1.363e+000 C

Korrosionsmessdaten (Frontklappe 0 ((mA))

berechnet aus den vollständigen Originaldaten im ausgewählten Zeitraum

Minimum:	-2.895e-001 mA	**Summe positiv:**	0.000e+000 C
Maximium:	-3.510e-005 mA	**Summe negativ:**	9.456e+000 C
Mittelwert:	0.000e+000 mA	**Summe gesamt**	9.456e+000 C

Abbildung B.5 – Von `DataAnalyzer` generiertes Dokument.

Tabellenverzeichnis

Abbildungsverzeichnis

Quellcodeverzeichnis

Literaturverzeichnis

[1] Norm DIN 1319-1 1995-01. *Grundlagen der Meßtechnik*

[2] MARUYAMA, Tasuya ; YAMADA, Tsutomu: Sharing IO Devices using Hardware Virtualization Method for Component-Based Industrial Controllers. In: *Emerging Technologies and Factory Automation, 2008. ETFA 2008. IEEE International Conference on*, IEEE, 15–18 Sept. 2008, S. 705–708

[3] MEYERS, Scott ; MERKLE, Bernhard: Die Zukunft von C++. In: *iX* 09 (2006), S. 118–119

[4] *LabVIEW – Grafisches Programmiersystem von Native Instruments*. `http://www.ni.com`

[5] DEMETZ, Markus: *Software-modules for an electrochemical measurement system*, Institut für Chemische Technologien und Analytik, Technische Universität Wien, Diplomarbeit, 2007

[6] SIMMONS, R. ; APFELBAUM, D.: A task description language for robot control. In: *Intelligent Robots and Systems, 1998. Proceedings., 1998 IEEE/RSJ International Conference on* Bd. 3, 1998, S. 1931–1937 vol.3

[7] *CINT: The C++ Interpreter*. `http://root.cern.ch/root/Cint.html`

[8] *ROOT | A Data Analysis Framework*. `http://root.cern.ch`

[9] *NOVA – Advanced Electrochemical Software – Proprietäres Softwaresystem von Metrohm Autolab*. `http://www.metrohm-autolab.com/`

[10] KEITHLEY: *Model 2750 Multimeter/Switch System User's Manual*, 2003

[11] RYLAND, J.: Can LXI replace GPIB? In: *Autotestcon, 2005. IEEE*, 2005, S. 739–743

[12] *The HS488 Protocol*. NI Developer Zone. `http://www.ni.com/white-paper/4283/en/`. Version: 2006

[13] PURCELL, A.: The search for a GPIB replacement. In: *AUTOTESTCON '99. IEEE Systems Readiness Technology Conference, 1999. IEEE*, 1999. – ISSN 1080–7725, S. 169–177

[14] BROWN, M.F.: PC/104-ISA to PCI. In: *Wescon/98*, 1998. – ISSN 1095–791X, S. 210–215

[15] CHAME, A.: PCI bus in high speed I/O systems applications. In: *Aerospace Conference, 1998 IEEE* Bd. 4, 1998. – ISSN 1095–323X, S. 505–514 vol.4

[16] BI, Bo ; SUN, Shuying ; WANG, Chunping: Design of Data Acquisition Equipment Based on USB. In: *Electronic Measurement and Instruments, 2007. ICEMI '07. 8th International Conference on*, 2007, S. 1–866–1–869

[17] ATCHISON, L.: Firewire: standard computer industry interconnect for test measurement. In: *Aerospace and Electronic Systems Magazine, IEEE* 14 (1999), Aug, Nr. 8, S. 21–24. http://dx.doi.org/10.1109/62.784045. – DOI 10.1109/62.784045. – ISSN 0885–8985

[18] MCCARTHY, A.B. ; FAYA, Peng: Comparing GPIB, LAN/LXI, PCI/PXI Measurement Performance in Hybrid Systems. In: *Autotestcon, 2006 IEEE*, 2006. – ISSN 1088–7725, S. 122–128

[19] PIOTROWSKI, Janus: *Theory of Physical and Technical Measurement*. 1992

[20] CROPLEY, D.H.: A graphical modelling method for the representational form of measurement theory. In: *Instrumentation and Measurement Technology Conference, 1997. IMTC/97. Proceedings. Sensing, Processing, Networking., IEEE* Bd. 2, 1997. – ISSN 1091–5281, S. 1338–1343 vol.2

[21] WIERMAN, M.J. ; TASTLE, W.J.: Measurement theory and subsethood. In: *Fuzzy Information Processing Society (NAFIPS), 2010 Annual Meeting of the North American*, 2010, S. 1–5

[22] ELLIS, BRIAN: *BASIC CONCEPTS OF MEASUREMENT*. CAMBRIDGE UNIVERSITY PRESS, 1966

[23] FINKELSTEIN, Ludwik: Fundamental concepts of measurement. In: *Acta IMEKO* 3, no. 1, article 4 (2014), May

[24] PFANZAGL, J.: *THEORY OF MEASUREMENT*. Physica-Verlag, 1968

[25] CAUSEY, Robert L.: Derived Measurement, Dimensions, and Dimensional Analysis. In: *Philosophy of Science* 36 (1969), Sep., Nr. 3, S. 252–270

[26] SUPPES, Patrick ; ZINNES, Joseph L.: Basic Measurement Theory. In: LUCE, Duncan R. (Hrsg.) ; BUSH, Robert R. (Hrsg.) ; GALANTER, Eugene (Hrsg.): *Handbook of Mathematical Psychology* Bd. I, John Wiley & Sons, Inc., 1963

[27] WANG, Yingxu: The measurement theory for software engineering. In: *Electrical and Computer Engineering, 2003. IEEE CCECE 2003. Canadian Conference on* Bd. 2, 2003. – ISSN 0840–7789, S. 1321–1324 vol.2

[28] HARMANN, Carl H. ; VIELSTICH, Wolf: *Elektrochemie*. Wiley-VCH, 2001

[29] BIPM ; IEC ; IFCC ; ILAC ; IUPAC, ISO ; IUPAP ; OIML: *Evaluation of measurement data – Guide to the expression of uncertainty in measurement*. 2008. – First edition

[30] BONDAVALLI, A. ; CECCARELLI, A. ; FALAI, L. ; VADURSI, M.: Foundations of Measurement Theory Applied to the Evaluation of Dependability Attributes. In: *Dependable Systems and Networks, 2007. DSN '07. 37th Annual IEEE/IFIP International Conference on*, 2007, S. 522–533

[31] Norm DIN EN ISO 9241 2011-01. *Ergonomie der Mensch-System-Interaktion*

[32] WEBER, Heinrich: *Rechnergestützte Meßverfahren*. Vogel Buchverlag, 1996

[33] *NOVA User manual 1.10*

[34] PRESSER, Leon ; WHITE, John R.: *Linkers and Loaders*. University of California, 1972

[35] DREPPER, Ulrich: *Static Linking Considered Harmful*. `http://www.akkadia.org/drepper/no_static_linking.html`

[36] GAMMA, Erich ; HELM, Richard ; JOHNSON, Ralph ; VLISSIEDES, John: *Design Patterns. Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995

[37] *Struts*. The Apache Software Foundation. `https://struts.apache.org/struts1eol-announcement.html`

[38] *Struts2*. The Apache Software Foundation. `http://struts.apache.org`

[39] BROWN, Walter E. ; CANAL, Philippe u. a.: *A Case for Reflection*. 2005

[40] DOUCET, F. ; SHUKLA, S. ; GUPTA, R.: Introspection in system-level language frameworks: meta-level vs. integrated. In: *Design, Automation and Test in Europe Conference and Exhibition, 2003*, 2003. – ISSN 1530–1591, S. 382–387

[41] TYNG-RUEY, Chuang ; KUO, Y.S. ; CHIEN-MIN, Wang: Non-intrusive object introspection in C++: architecture and application. In: *Software Engineering, 1998. Proceedings of the 1998 International Conference on*, 1998. – ISSN 0270–5257, S. 312–321

[42] VOLLMANN, Detlef: *Metaclasses and Reflection in C++*. `http://www.vollmann.ch/pubs/meta/meta/meta.html`. Version: 2000

[43] CHENG, Harry H.: Ch: A C/C++ Interpreter for Script Computing. In: *C/C++ Users Journal* 24 (2006), January, Nr. 1, S. 6–12

[44] LUOMA, Harri ; LAHTINEN, Essi ; JARVINEN, Hannu-Matti: CLIP, a Command Line InterPreter for a subset of C++. In: LISTER, Raymond (Hrsg.) ; SIMON (Hrsg.): *Seventh Baltic Sea Conference on Computing Education Research (Koli Calling 2007)* Bd. 88. Koli National Park, Finland : ACS, 2007 (CRPIT), S. 199–202

[45] STROUSTRUP, Bjarne: *Sibling Rivalry: C and C++*. The C/C++ Users Journal, 7, 8, 9 2002

[46] *Autolab SDK 1.11 – User Manual*

[47] *NOVA External Devices Tutorial*

[48] HAENDEL, Lars: *The Function Pointer Tutorials*. `http://www.newty.de/fpt/index.html`. Version: 2005

[49] FOWLER, Martin: *Patterns of enterprise application architecture*. Addison-Wesley, 2008

[50] SMITH, J.E. ; NAIR, R.: The architecture of virtual machines. In: *Computer* 38 (2005), May, Nr. 5, S. 32–38. `http://dx.doi.org/10.1109/MC.2005.173`. – DOI 10.1109/MC.2005.173. – ISSN 0018–9162

[51] *Boost Spirit Parser Framework*. `http://boost-spirit.com/home/`

[52] *Boost C++ Libraries*. `http://www.boost.org/`

[53] *Qt – C++-Klassenbibliothek für die plattformübergreifende Programmierung von grafischen Benutzeroberflächen.* `http://qt-project.org`

[54] ANDRADE, H.A. ; KOVNER, S.: Software synthesis from dataflow models for G and LabVIEW/sup TM/. In: *Signals, Systems amp; Computers, 1998. Conference Record of the Thirty-Second Asilomar Conference on* Bd. 2, 1998. – ISSN 1058–6393, S. 1705–1709 vol.2

[55] EGGER, Bernhard: *Material characterization with microelectrodes*, Institut für Chemische Technologien und Analytik, Technische Universität Wien, Diss., 2007

[56] STELLNBERGER, K. ; BRAIDT, R. ; LUCKENEDER, G. ; OGLE, K. ; DOSTAT, L. ; KÖNIG, S. ; JANSSEN, S. ; BEIER, F. ; FAFILEK, G.: Investigation of the corrosion mechanism in flanged joints of car bodies. 2009. – Forschungsbericht

[57] FESSMANN, Jürgen ; ORTH, Helmut: *Angewandte Chemie und Umwelttechnik für Ingenieure: Handbuch für Studium und betriebliche Praxis*. Hüthig Jehle Rehm, 2002

[58] HOLLEMAN ; WIBERG: *Lehrbuch der anorganischen Chemie*. 1964

[59] JEROLITSCH, David: *Verbesserung der Korrosionsbeständigkeit von Mischbauverbindungen an Fahrzeuganbauteilen*, Institut für Chemische Technologien und Analytik, Technische Universität Wien, Diss., 2011

[60] WIPF, David O. ; FAN, Fu-Ren F. u. a. ; BARD, Allen J. (Hrsg.) ; MIRKIN, Michael V. (Hrsg.) u. a.: *Scanning Electrochemical Microscopy*. Marcel Dekker, Inc., 2001

[61] SCHULTE, Albert ; SCHUHMANN, Wolfgang: *Constant-Distance Mode Scanning Electrochemical Microscopy*

[62] *Persönliche Mitteilung von Prof. Dr. Gunther Wittstock.*

LEBENSLAUF

PERSONENDATEN

Name:	Markus Demetz, Dipl.-Ing.
Geburtsdatum:	17. Mai 1979 in Tulln an der Donau
Staatsangehörigkeit:	Italien
Email:	markus@demetz.eu

BERUFLICHE LAUFBAHN

Juni 2013 - Mai 2014
AIT Austrian Institute of Technology GmbH

Softwareentwickler
Implementierung von Werkzeugen im Bereich der Testfallgenerierung. High-Performance Modell Exploration.

April 2008 - Februar 2012
CEST GmbH

Wissenschaftlicher Mitarbeiter
Wissenschaftlicher Mitarbeiter im Bereich der Messablaufsteuerung.

Juni 2006 - März 2008
Technische Universität Wien

Projekt Assistent
Projektassistent am Institut für Chemische Technologien und Analytik, Bereich Elektrochemie. Programmierung von Modulen für elektrochemische Messaufgaben.

April 2007 - Dezember 2014
Selbstständiger Unternehmer

Selbständiger Unternehmer
Selbstständiger Unternehmer im Gewerbe „Dienstleistungen in der automatischen Datenverarbeitung und Informationstechnik“.

AUSBILDUNG

2008 - 2015
Doktoratsstudium

Technische Universität Wien
Doktoratsstudium an der Technischen Universität Wien am Institut für Chemische Technologien und Analytik. Thema: Analyse, Modellierung und Management komplexer Messvorgänge in der Chemie.

2000 - 2007
Diplomstudium Informatik

Technische Universität Wien
Diplomstudium Informatik an der Technischen Universität Wien.

1999 - 2000
Magisterstudium Wirtschaftsinformatik

Universität Wien
Magisterstudium Wirtschaftsinformatik an der Universität Wien. Nicht abgeschlossen.

1994 - 1999
Handelsoberschule „Raetia“

Handelsoberschule „Raetia“ St.Ulrich
Handelsschule mit Maturaabschluss und paritätischem Lehrgang in den Fächern Italienisch und Deutsch.

SPRACHKENNTNISSE

Deutsch	· Muttersprache	Italienisch	· Zweitsprache
Englisch	· Gut in Wort und Schrift	Spanisch	· Grundkenntnisse

Wien, am