Lehrstuhl für Umformtechnik

Masterarbeit

Implementierung eines zentralen Data Acquisition Systems und Softwarelösung zur Datensynchronisation einer Induktionsprüfanlage

Dominik Joachim Müller, BSc

Februar 2023

Eidesstattliche Erklärung

MONTANUNIVERSITÄT LEOBEN
www.unileoben.ac.at

EIDESSTATTLICHE ERKLÄRUNG

Ich erkläre an Eides statt, dass ich diese Arbeit selbständig verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt, und mich auch sonst keiner unerlaubten Hilfsmittel bedient habe.

Ich erkläre, dass ich die Richtlinien des Senats der Montanuniversität Leoben zu "Gute wissenschaftliche Praxis" gelesen, verstanden und befolgt habe.

Weiters erkläre ich, dass die elektronische und gedruckte Version der eingereichten wissenschaftlichen Abschlussarbeit formal und inhaltlich identisch sind.

Datum 13.02.2023

Unterschrift Verfasser/in
Dominik Joachim Müller

Danksagung

An dieser Stelle möchte ich allen danken, die mich im Verlauf meines Studiums und beim Verfassen dieser Arbeit unterstützt haben.

Ein großer Dank gilt meinen Betreuern von der Montanuniversität Leoben Univ.Prof. Dipl.-Ing. Dr.techn. Martin Stockinger und Dipl.-Ing. Marcel Sorger. Marcel möchte ich hier noch einmal ganz besonders danken, für seine Geduld, seine ständige Verfügbarkeit und seine Bemühtheit, die mich, trotz mancher Schwierigkeiten, zum Weitermachen animiert hat.
Danke sagen möchte ich auch den Betreuern vom Material Center Leoben, allen voran Dr. Peter Raninger und DI Petri Prevedel, die diese Arbeit überhaupt erst ermöglicht haben und allzeit für etwaige Fragen zur Verfügung gestanden sind.

Ein riesengroßer Dank gilt auch meiner Familie, allen voran meinen Eltern, die mich nicht nur im Studium, sondern in jeder meiner Lebensphasen unterstützt haben und ohne die all das nicht möglich gewesen wäre.
Auch meiner Freundin Maria, die in den letzten Monaten wohl am meisten auszuhalten hatte, möchte ich von Herzen danken. Ohne euch wäre ich heute nicht dort wo und nicht der, der ich bin. Danke!

Kurzfassung

Unter dem Begriff Industrie 4.0 versteht man die immer weiter zunehmende Anwendung von Technologien der IT über alle Ebenen der Wertschöpfungskette hinweg. Die starren Grenzen der herkömmlichen Automatisierungstechnik verschwimmen dabei zusehends. Um mit dem rasanten technischen Fortschritt im digitalen Bereich mitzuhalten und konkurrenzfähig zu bleiben, sehen sich Betriebe aktuell häufig mit der Notwendigkeit konfrontiert ihre Systeme an die Bedürfnisse dieser vierten industriellen Revolution anzupassen. Ein Weg dies zu bewerkstelligen ist das sogenannte „Retrofitting“, bei dem bestehende Anlagen durch Aufrüstung mit geeigneten Komponenten an die neuen Anforderungen angepasst werden.

Im Zuge dieser Arbeit soll der Induktionsprüfstand am Material Center Leoben einem solchen Retrofitting unterzogen werden. Dieser besteht bis dato aus mehreren proprietären Teilsystemen, die zur Durchführung einer Messung jeweils separat bedient werden müssen. Ziel ist es, mittels einer SPS als zentraler Steuereinheit, diese Komponenten in ein nicht proprietäres Gesamtsystem einzubinden, welches die Messdurchführung vereinfacht und eine einheitliche Verarbeitung aller Messdaten ermöglicht. Diese Masterarbeit stellt dabei den ersten Teil dieses Projektes dar und zielt einerseits darauf ab die benötigte Hardware aufzubauen und andererseits ein softwaretechnisches Grundgerüst zu schaffen, welches als Basis für die nachfolgenden Arbeiten dienen soll und im Rahmen dieser ergänzt und erweitert werden kann.

Abstract

The term Industry 4.0 refers to the permanently increasing use of information technology across all levels of the value chain. As a result, the rigid boundaries of conventional automation technology are becoming increasingly blurred. To keep up with the rapid development in the digital field and stay competitive, companies are faced with the need to adapt their machinery to match the requirements of this fourth industrial revolution. One way this can be accomplished is via the use of the so-called “retrofitting”, which upgrades existing systems by using suitable components, to meet the new demands.

In the course of this work the induction test stand at the Material Center Leoben is to be subjected to such a retrofitting. Up to now it consists of several proprietary subsystems, each of which must be operated separately in order to carry out a measurement. The aim is to integrate these components into one non-proprietary system using a PLC as the central control unit. This new overall–system can simplify the execution of measurements and enable uniform processing of all measurement data. This master thesis represents the first part of this project and focuses on the one hand on building up the necessary hardware and on the other hand on creating a software framework which should serve as a basis for subsequent work and may furthermore be supplemented and extended.

Inhaltsverzeichnis

Abkürzungsverzeichnis

Zeichen	Einheit	Beschreibung
R_0	$[\Omega]$	Grundwiderstand Pt100
R_{Pt}	$[\Omega]$	Widerstand Pt100
I_0	$[A]$	Messstrom Pt100
U_m	$[V]$	Spannungsabfall Pt100
T	$[K]$	zu berechnende Temperatur Pt100
α	K^{-1}	Koeffizient zur Temperaturberechnung Pt100 (linear)
b	K^{-2}	Koeffizient zur Temperaturberechnung Pt100 (quadratisch)
c	K^{-4}	Koeffizient zur Temperaturberechnung Pt100 (vierte Potenz)
$\vec{F}_m$	$[N]$	Lorentzkraft
Q	$[C]$	elektrische Ladung
$\vec{B}$	$[T]$	vektorielle magnetische Flussdichte
$\vec{v}$	$[ms^{-1}]$	Geschwindigkeit einer elektrischen Ladung im Magnetfeld
u_i	$[V]$	induzierte Spannung
N	$[-]$	Windungsanzahl Spule
Φ	$[Wb]$	magnetischer Fluss
t	$[s]$	Zeit
$\vec{E}$	$[Vm^{-1}]$	elektrisches Feld
$\vec{s}$	$[m]$	Weg
Q_i	$[J]$	induzierte Wärme
I_i	$[A]$	induzierter Strom
R_Q	$[\Omega]$	elektrischer Widerstand bei Induktionserwärmung
I_{WS}	$[A]$	Wirbelströme
$I_{Induktor}$	$[A]$	Induktorstrom
J	$[Am^{-2}]$	Stromdichte
J_0	$[Am^{-2}]$	maximale Stromdichte am Werkstückrand
δ	$[mm]$	Eindringtiefe
ρ	$[\Omega m]$	spezifischer elektrischer Widerstand
μ_r	$[-]$	relative magnetische Permeabilität
f	$[Hz]$	Frequenz des induzierten Stroms
π	$[-]$	Kreiszahl
B	$[T]$	skalare magnetische Flussdichte

H	$[Am^{-1}]$	Magnetische Feldstärke
B_s	$[T]$	Sättigungsmagnetisierung
B_r	$[T]$	Remanenz
H_c	$[Am^{-1}]$	Koerzitivfeld
I_{Sensor}	$[A]$	Sensorstrom
U_{Sensor}	$[V]$	Sensorspannung
f_{SPS_a}	$[Hz]$	Sampling Rate des RevPis (aktiver Modus)
SF_{SG_a}	$[-]$	Signalform des Signalgenerator (aktiver Modus)
f_{SG_a}	$[Hz]$	Anregungsfrequenz des Signalgenerators (aktiver Modus)
A_{SG_a}	$[V]$	Anregungsamplitude des Signalgenerators (aktiver Modus)
f_{pico_a}	$[Hz]$	Sampling Rate des Picoscopes (aktiver Modus)
t_{pico_a}	$[s]$	Dauer einer Picoscope-Messung (aktiver Modus)
N_{sampes,P_a}	$[-]$	Anzahl der zu messenden Datenpunkte pro Periode des Erregersignals (aktiver Modus)
N_{p_a}	$[-]$	Anzahl der zur Messung herangezogenen Perioden des Erregersignals (aktiver Modus)
N_{EM_a}	$[-]$	Anzahl der Einzelmessungen die mit dem Picoscope durchgeführt werden (aktiver Modus)
Δt_{EM_a}	$[s]$	Zeitabstand zwischen den Einzelmessungen des Picoscopes (aktiver Modus)
f_{SPS_p}	$[Hz]$	Sampling Rate des RevPis (passiver Modus)
SF_{SG_p}	$[-]$	Signalform des Signalgenerator (passiver Modus)
f_{SG_p}	$[Hz]$	Anregungsfrequenz des Signalgenerators (passiver Modus)
A_{SG_p}	$[V]$	Anregungsamplitude des Signalgenerators (passiver Modus)
f_{pico_p}	$[Hz]$	Sampling Rate des Picoscopes (passiver Modus)
t_{pico_p}	$[s]$	Dauer einer Picoscope-Messung (passiver Modus)
N_{sampes,P_p}	$[-]$	Anzahl der zu messenden Datenpunkte pro Periode des Erregersignals (passiver Modus)
N_{p_p}	$[-]$	Anzahl der zur Messung herangezogenen Perioden des Erregersignals (passiver Modus)
N_{EM_p}	$[-]$	Anzahl der Einzelmessungen die mit dem Picoscope durchgeführt werden (passiver Modus)
Δt_{EM_p}	$[s]$	Zeitabstand zwischen den Einzelmessungen des Picoscopes (passiver Modus)

1 Einleitung

Im Rahmen der vierten industriellen Revolution, auch als Industrie 4.0 bekannt, sehen sich viele Betriebe dazu veranlasst, ihre Strukturen dahingehend zu verändern, dass die neuen Technologien der IT im Bereich der Digitalisierung und Automatisierung, die die Industrie 4.0 mit sich bringt, eingebunden werden können. Dabei geht der Trend weg von den starren Strukturen einzelner, proprietärer Systeme, hin zu flexibleren, offeneren Systemen, die eine Kommunikation und einen Datenaustausch über alle Betriebsebenen hinweg ermöglichen. Ein wichtiger Begriff ist hierbei der des "Retrofittings“. Darunter versteht man das Aufrüsten bestehender Anlagen, um diese auf den Stand der Industrie 4.0 zu bringen.

So einem Retrofitting soll der Induktionsprüfstand am Material Center Leoben (MCL) unterzogen werden. Dieser besteht aus mehreren proprietären Einzelsystemen, die aktuell alle separat bedient werden müssen. Ziel ist nun eine zentrale Kontrolleinheit in Form einer SPS zu implementieren, die die Koordination und Steuerung der einzelnen Komponenten übernimmt und für den Messablauf und die Datenverarbeitung zuständig ist. Dieses Projekt soll im Rahmen von zumindest drei Masterarbeiten durchgeführt werden, wobei diese hier vorgestellte Arbeit den Beginn des Projektes markiert. Ziel dabei ist es die benötigte Hardware aufzubauen und ein softwaretechnisches „Framework“ zu liefern, das als Basis für die weiteren Arbeiten dienen soll und im Zuge dieser erweitert und angepasst werden kann.

Im nachfolgenden zweiten Kapitel wird der aktuelle Stand der Technik behandelt, wobei hier der Begriff der Industrie 4.0 näher betrachtet wird. Darauf folgt – in Kapitel 3 – eine Erläuterung der Grundlagen, die für diese Arbeit von Relevanz sind. Im vierten Kapitel wird der aktuelle Stand des Induktionsprüfstandes beschrieben und ein Ausblick auf den gewünschten, angedachten Endzustand gegeben. Weiters werden die Ziele für diese Arbeit noch einmal spezifiziert. Das Kernstück der Arbeit stellt Kapitel 0 dar. Hier wird sowohl auf die hardware- als auch die softwaretechnische Umsetzung der im vorhergehenden Kapitel behandelten Zielsetzungen eingegangen. Zu guter Letzt folgt eine Endbetrachtung der Arbeit, wobei dabei sowohl auf aufgetretene Probleme, als auch auf mögliche weiterführende Arbeitsschritte eingegangen wird.

2 Stand der Technik

Obwohl in der Literatur bis dato keine einheitliche Definition des Begriffes **Industrie 4.0** existiert, herrscht Konsens darüber, dass dieser die vierte industrielle Revolution und den damit einhergehenden weiter zunehmenden Einsatz von Methoden der Kommunikations- und Informationstechnologie in der Industrie beschreibt [1].

Stand bei der dritten industriellen Revolution noch die Automatisierung einzelner Fertigungssysteme mittels Technologien wie z.B. Sensoren und Aktoren im Fokus, wird im Zuge der vierten industriellen Revolution das Augenmerk vermehrt auf die Kommunikation einzelner Fertigungssysteme miteinander und das automatisierte Generieren, Verarbeiten und Auswerten von produktspezifischen Daten gelegt. Diese Vernetzung und der Zugang zu erforderlichen Informationen in Form von (Echtzeit-)Daten ermöglichen eine Optimierung der Prozesse entlang der gesamten Wertschöpfungskette. So können beispielsweise Produktentwicklungszeiten verkürzt, die Produkt– und Prozessqualität erhöht, die Fehlerdetektion optimiert und die Produktivität gesteigert werden [2,3].

Als ein wesentlicher Grundbaustein der Industrie 4.0 gilt das auf das **„Internet of Things**“(IoT) aufbauende „**Industrial Internet of Things**“(IIoT). Dabei handelt es sich um ein Netzwerk, das industrielle Anlagen, wie Maschinen, Steuer- und Regelsysteme und Informationssysteme miteinander verknüpft und diesen eine Kommunikation untereinander ermöglicht [2]. Dabei werden betriebliche Infrastrukturen sowohl auf vertikaler als auch auf horizontaler Ebene miteinander verknüpft. Daraus ergeben sich Möglichkeiten zur zentralen Überwachung und Koordinierung von einzelnen Anlagen und Prozessen und zur schnellen und dynamischen Reaktion auf etwaige gewünschte Änderungen im Prozesslauf. Weiters bietet diese Vernetzung und die kontinuierliche Auswertung von Daten die Möglichkeit nahende Probleme frühzeitig zu erkennen und so z.B. Wartungsarbeiten flexibel abzustimmen [3]. Während die prinzipiellen Anforderungen an das IoT und das IIoT ähnlich sind (niedrigpreisige und ressourcenschonende Geräte mit der Möglichkeit zur Netzwerkanbindung, Skalierbarkeit, etc.), wird beim IIoT das Augenmerk primär auf die zuverlässige und zeitgenaue Übertragung von Daten und weniger auf die Flexibilität gelegt. Auch steht beim IIoT die Kommunikation einzelner Geräte untereinander und nicht die Kommunikation mit dem Menschen im Vordergrund [4].

Ein weiterer Grundbaustein der Industrie 4.0 sind die sogenannten „**Cyber Physical Systems**“ (CPS), im Industriekontext bezeichnet als „**Cyber Physical Production Systems**“(CPPS). Diesen zugrunde liegen die "Embedded Systems“(ES). Darunter versteht man elektronische Steuereinheiten, die ohne äußeres Zutun mithilfe von Sensoren, Aktoren und softwarebasierten Anweisungen Regel-, Steuer- und Überwachungsaufgaben übernehmen. Diese Systeme sind zumeist geschlossene Regelkreise, die nicht mit

ihrer Umwelt interagieren [5]. Die CPS stellen eine Erweiterung zu den ES dar. So können laut *Pistorius* CPS definiert werden als ES, die [2]:

- durch Unterstützung von Sensoren physikalische Daten generieren und mittels Aktoren reale Vorgänge beeinflussen
- Daten sichern und verarbeiten und daraus Handlungen ableiten
- über Kommunikationsschnittstellen miteinander verbunden sind
- bereitstehende Dienste und Daten ortsunabhängig nutzen und anbieten
- und Möglichkeiten zur Kommunikation und Steuerung in Form von Mensch-Maschine-Schnittstellen („Graphical User Interface“ (GUI)) zur Verfügung stellen

CPPS stellen die Verknüpfung der virtuellen mit der physikalischen Welt dar. So wird durch den digitalen Teil eines CPPS ein realer Prozess oder ein Produkt nachgebildet und durch das Füttern mit Daten möglichst echtheitsgetreu modelliert. Dieses digitale Abbild wird in der Literatur, je nach Art der Kommunikation zwischen digitaler und physikalischer Domäne, als „**Digital Model**“(DM), „**Digital Shadow**“(DS), oder „**Digital Twin**“(DT) bezeichnet [6]. Dabei steht nicht nur die Abbildung des momentanen Prozessablaufs im Vordergrund, sondern vor allem die Modellierung von zukünftigem Verhalten, um z.B. nahende Probleme frühzeitig zu erkennen, bevor sie auftreten [7]. Den physikalischen Teil eines CPPS stellen unter anderem Sensoren, Aktoren und Prozessoren, neben weiteren mechanischen und elektrischen Komponenten und Schnittstellen zur Kommunikation, dar[2].

Kurz gesagt kann man ein CPPS als ein System von Einzelsystemen beschreiben, wobei die einzelnen Systeme untereinander in Echtzeit interagieren und ihre Aktionen situationsbedingt aufeinander abstimmen. Dabei werden alle Produktionsebenen – Maschinensteuerung, Produktionsprozesse und Logistik – eingeschlossen, was es ermöglicht auf unvorhergesehene Vorkommnisse schneller zu reagieren und Entscheidungsfindungen vereinfacht [4,8,9]. Das führt zum Entstehen von „Smart Factories“, die sich durch das Vorhandensein von CPPS auszeichnen. Weitere Schlüsseltechnologien der Smart Factories bilden dabei das „IoT“, „Big Data“ und „Artificial Intelligents“ (AI) [10].

Das Zusammenspiel aus dem IIoT und den CPS (bzw. CPPS) bildet die Basis für die Industrie 4.0 und ist in *Abb. 2-1* schematisch dargestellt.

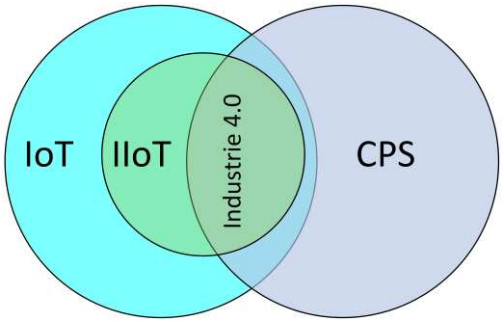


Abb. 2-1: Zusammenhang IoT, IIoT, CPS und Industrie 4.0 [4]

Durch die Einführung von CPPS ergeben sich eine Vielzahl an Chancen und Möglichkeiten für Unternehmen. Das hohe Maß an permanent verfügbaren relevanten Informationen schafft die Möglichkeit Prozesse zu optimieren und Arbeitsschritte ideal aufeinander abzustimmen und somit Zeit und Kosten zu sparen. Auch können durch die permanente automatisierte Überwachung von Maschinendaten Fehlern vorgebeugt und der Mensch in seiner Arbeit unterstützt werden. Darüber hinaus ergibt sich durch die Anpassungsfähigkeit von CPPS die Möglichkeit schnell auf Änderungen im Prozessablauf zu reagieren [2].

Die oben beschriebenen Innovationen der Industrie 4.0 bringen, wie schon erwähnt, eine große Menge an unterschiedlichen Daten mit sich, die verarbeitet und ausgewertet werden müssen, oft in Echtzeit. Die Methoden, die hierfür verwendet werden, werden in der Literatur weithin unter dem Begriff „**Big Data Analysis**“ zusammengefasst [3]. Schlüsselpunkte dabei sind die Menge an verfügbaren Daten, die Geschwindigkeit der Datengenerierung und Verarbeitung, die Vielfalt an Datenstrukturen, die Richtigkeit der Daten und der Nutzen, den man aus den analysierten Daten ziehen kann [6].

Die Grenzen der Bausteine, auf welchen die Industrie 4.0 aufbaut, sind verschwimmend und es ergibt sich eine Vielzahl an integralen und verwandten Forschungsgebieten, die in den letzten Jahren an Bedeutung gewonnen haben. „Industriell angewandte künstliche Intelligenz“, „edge computing to the edge cloud“, „5G in Fabriken“, „kollaborative Roboteranwendungen“, „eigenständige intralogistische Systeme“ und „zuverlässige Dateninfrastrukturen“ bilden die sechs großen Trends, die sich dabei für die weitere Entwicklung der Industrie 4.0 herauskristallisiert haben [11,12].

Mit den neuen Entwicklungen der Industrie 4.0 mitzuhalten ist essenziell für die Konkurrenzfähigkeit moderner Betriebe, doch sind vor allem ältere Anlagen und Maschinen häufig nicht für die neuen Technologien ausgelegt. Das klassische „**Retrofitting**“ beschreibt den Prozess des Aufrüstens bestehender Anlagen – mit z.B. weiteren Sensoren, Aktoren und Regelsystemen – um einzelne Maschinen an neue Gegebenheiten und Anforderungen anzupassen. „**Smart Retrofitting**“ schließt dabei die Anwendung von CPS mit ein [13].

3 Grundlagen

Im nachfolgenden Abschnitt werden die relevanten Grundlagen für diese Arbeit besprochen. Dabei wird zuerst der Fokus auf die Automatisierungspyramide gelegt, anhand derer Ebenen die einzelnen Teilsysteme des Automatisierungsprozesses und deren physikalische Funktionsmechanismen beschrieben werden. In weiterer Folge wird der Begriff „Retrofitting“ näher erläutert. Abschließend wird auf die Grundlagen der Induktionserwärmung und der dafür notwendigen Gerätschaften eingegangen.

3.1 Automatisierungspyramide

Um die Komplexität bei der Automatisierung von Anlagen und Prozessen in Unternehmen zu verringern und den Aufbau von Automationssystemen übersichtlich zu gestalten, werden die dafür notwendigen Technologien in aufeinander aufbauende Ebenen unterteilt. Die sich dabei ergebende hierarchische Struktur wird als „**Automatisierungspyramide**“ bezeichnet. Je nach Literaturquelle besteht diese dabei aus drei bis sieben Ebenen, da teilweise, je nach Bedarfsfall, Ebenen weggelassen oder zusammengefasst werden. Zwischen den Ebenen gibt es herkömmlicherweise wenig Schnittstellen, weshalb diese als isoliert betrachtet werden können. In *Abb. 3-1* ist die Automatisierungspyramide nach Siepmann abgebildet, die auf der Normreihe *DIN EN 62264 (ident zu IEC 62264)* basiert. Zusätzlich zu den fünf in diesem Standard festgelegten Ebenen fügt Siepmann noch eine untergeordnete Ebene, die Prozessebene, ein [14,15].

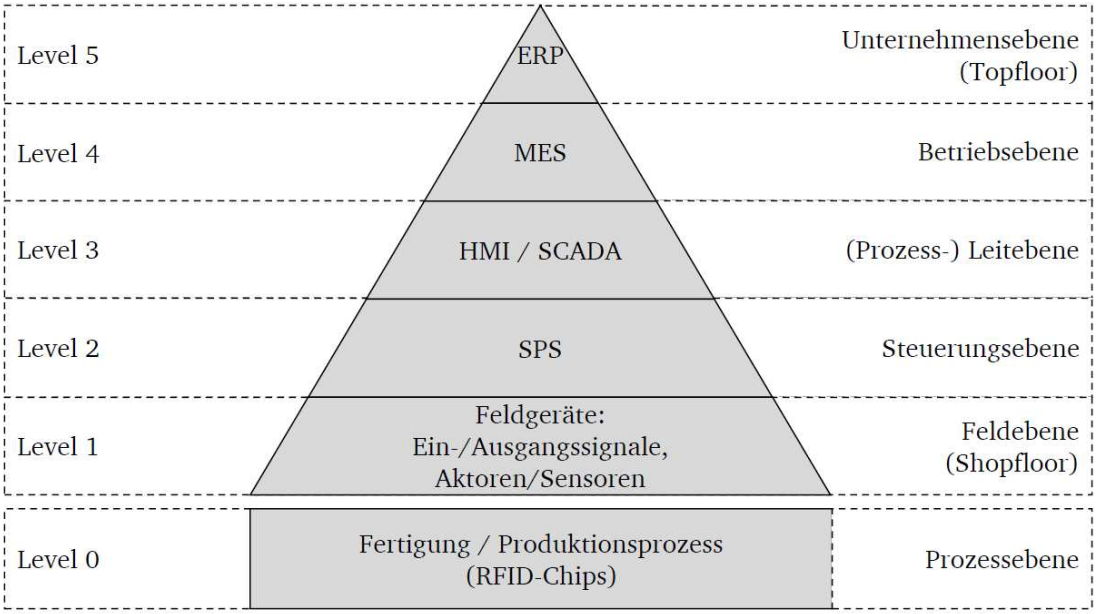


Abb. 3-1: Klassische Automatisierungspyramide nach Siepmann [15]

Die Gliederung der Pyramide stellt sich dabei wie folgt dar:

- Level 0 – Prozessebene:

  Diese Ebene beschreibt den wertschöpfenden Prozess, also den physikalischen Produktionsprozess [16].

- Level 1 – Feldebene/Shopfloor:

  Die Feldebene umfasst den Produktionsbereich und die sich dort befindlichen prozessrelevanten Sensoren und Aktoren. Informationen werden in Form von Ein- und Ausgangssignalen verarbeitet [15].

- Level 2 – Steuerungsebene:

  Eingehende Signale der Feldebene werden hier, meist mittels einer *speicherprogrammierbaren Steuerung* (SPS), verarbeitet und daraus resultierende Ausgangssignale erzeugt, die zur Steuerung von Aktoren der Feldebene dienen [15,16].

- Level 3 – (Prozess-)Leitebene:

  Diese Ebene dient dem Beobachten, Überwachen und Bedienen von Prozessen. Dabei kommen sogenannte *Supervision Control und Data Acquisition* (SCADA)–Systeme zum Einsatz, welche die aus der Feld- und Steuerungsebene kommenden Daten sammeln und es dem Bediener durch eine Mensch-Maschine-Schnittstelle ermöglichen diese zu überwachen und Bedieneingaben vorzunehmen [15].

- Level 4 – Betriebsebene:

  Diese Ebene dient der Produktionsplanung und Steuerung und bildet das Bindeglied zwischen Maschinensteuerung und Unternehmenseben. Hier kommt zumeist ein *Manufacturing Execution System* (MES) zum Einsatz, welches die Steuerung, Lenkung und Kontrolle der Produktion übernimmt. Hierbei werden Betriebs-, Maschinen-, Ressourcen- und Personaldaten erfasst und an die nächsthöhere Ebene weitergeleitet [15].

- Level 5 – Unternehmensebene / Topfloor:

  Die oberste Ebene der Automatisierungspyramide bildet ein *Enterprise Resource Planning* (ERP)-System. Hier finden die Grobplanung der Produktionsprozesse und die Ressourcenplanung statt [15,16].

Allgemein lässt sich festhalten, dass eine ebenenübergreifende Kommunikation nur zwischen benachbarten Ebenen stattfindet. Weiters nimmt der Zeithorizont der Datenverarbeitung von unten nach oben zu. Auf Feldebene werden Signale im Bereich von Millisekunden bis Sekunden verarbeitet, während auf Unternehmenseben der Zeitraum der Datenauswertung im Bereich von mehreren Tagen liegen kann. Die Informationsqualität nimmt ebenfalls von den unteren Ebenen zu den oberen Ebenen hin zu, wogegen die Informationsquantität abnimmt. Auf Feldebene werden eine Vielzahl von Daten

generiert, aus welchen dann, über die Ebenen hinweg, relevante Informationen extrahiert werden [17].

Durch die zunehmende Vernetzung, bedingt durch die fortschreitende Verbreitung von CPS und leistungsstärkerer elektronischer Geräte, die es erlauben die Aufgaben mehrerer Ebenen in zusammenfassenden Systemen zu realisieren, verschwimmen die starren Grenzen der Automatisierungspyramide zusehends [18] (siehe *Abb. 3-2*). Viel mehr wird in Zukunft ein komplexes Wertschöpfungsnetzwerk an deren Stelle treten, bei dem die Produktionssteuerung dezentral abläuft und ein Datenaustausch sowohl vertikal als auch horizontal unmittelbar über mehrere Ebenen der Automatisierungspyramide hinweg notwendig und möglich wird. Aktuell finden sich allerdings häufig noch Softwaresysteme mit proprietären Schnittstellen auf jeder Ebene der Automatisierungspyramide. Diese sind zumeist nicht miteinander vernetzt, oder erlauben aufgrund unterschiedlicher Semantik keine Datenintegration. Die Herausforderung wird sein, diese abgeschlossenen Systeme schrittweise zu öffnen, um eine Kommunikation untereinander zu ermöglichen. Der Weg führt also weg von hierarchisch strukturierten „Fertigungsinseln“ hin zu intelligent vernetzten Systemen, bei denen das Produkt selbst die Produktion steuert [14,15,17].

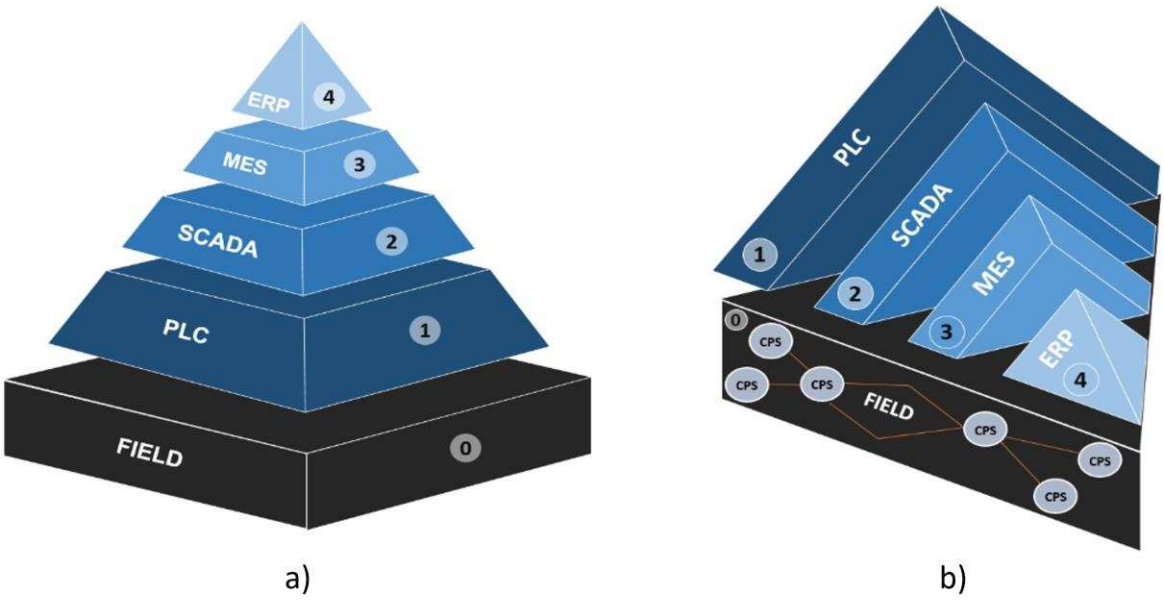


Abb. 3-2: Gegenüberstellung der Automatisierungspyramide mit a) starren Grenzen und b) aufgeweichten Strukturen [19]

3.2 Sensorik

Die Sensorik sitzt, wie die Aktorik, auf der untersten Ebene – der Feldebene – der in *Abb. 3-3* abgebildeten vereinfachten Darstellung der Automatisierungspyramide. Hierbei wurde die Prozessebene weggelassen, da diese bezogen auf die hier vorgestellte Arbeit nicht von Relevanz ist.

Abb. 3-3: Vereinfachte Automatisierungspyramide ohne Prozessebene (in Anlehnung an [15])

Allgemein stellt die Sensorik das Bindeglied zwischen nichtelektrischer Umwelt und elektrischer Messtechnik dar. Physikalische, nicht elektrische Messgrößen werden, unter Ausnützung verschiedener physikalischer und chemischer Effekte, in dazu proportionale elektrische Signale umgewandelt, die dann analog oder digital weiterverarbeitet werden können. Der Sensor selbst besteht dabei aus einem *Sensor-Element,* das zum Umwandeln der interessierenden Messgröße in ein elektrisches Messsignal dient, und einer *Auswerte-Elektronik* (Messsignalverarbeitung), die das Messsignal in einen darstellbaren Messwert umwandelt. Zusammen mit einer *Ausgabeeinheit*, die die Messwerte ausgibt, speichert oder weiterverarbeitet, bilden diese die in *Abb. 3-4* dargestellte Messkette. Typische Messwerte, die von der Messsignalverarbeitung ausgegeben werden, sind 4-20 mA oder 0-10 V. Besitzt der Sensor einen Mikrocontroller zur Vorverarbeitung und Digitalisierung des erhaltenen Messwerts, so wird dieser als „intelligenter Sensor“ oder „smart Sensor“ bezeichnet [20–22].

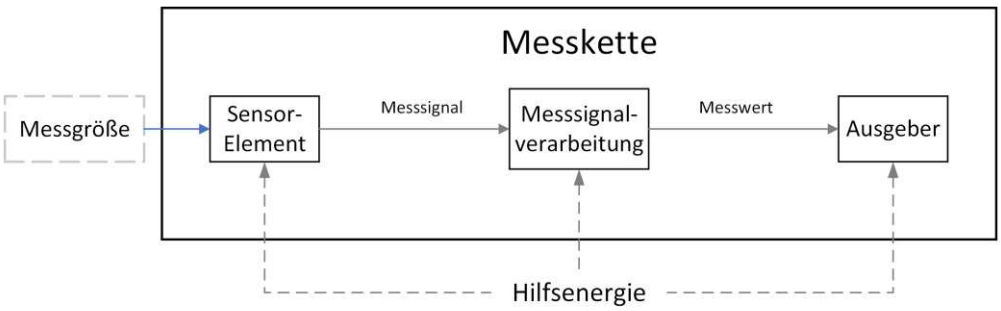


Abb. 3-4: Grundlegender Aufbau einer Messkette (in Anlehnung an [23])

Der Zusammenhang zwischen der zu messenden Größe und dem Ausgangssignal wird als Kennlinie bezeichnet. Diese ist im Idealfall linear und durch die Messglied-Empfindlichkeit, also die Steigung der Kennlinie, definiert. Die Abweichung von der idealen Kennlinie wird dabei als Linearitätsabweichung bezeichnet (siehe *Abb. 3-5*) [21].

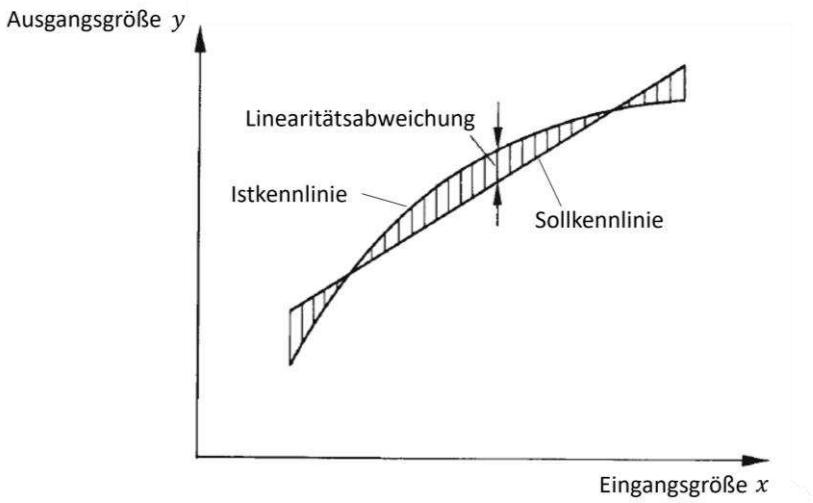


Abb. 3-5: Schematische Darstellung einer Kennlinie mit Linearitätsabweichung [21]

Zur computerunterstützten Weiterverarbeitung eines Messwertes muss dieser digitalisiert werden. Dazu wird der kontinuierliche, analoge Messwert mittels einem Analog-Digital-Converter (ADC) diskretisiert und in ein binäres, digital verarbeitbares Signal umgewandelt. Die Anzahl der dabei zur Verfügung stehenden Bits des ADCs bestimmt die erreichbare Auflösung (siehe *Kapitel 3.2.4*) [22].

3.2.1 Druckmessung

Das Prinzip der Druckmessung beruht für gewöhnlich darauf, dass durch die Verformung einer Membran die Druckdifferenz zwischen deren Vorder- und Rückseite gemessen wird. Dabei lässt sich grundlegend zwischen drei Messprinzipien – *Differenzdruckmessung, Relativdruckmessung* und *Absolutdruckmessung* – unterscheiden. Bei der Differenzdruckmessung werden an beide Seiten der Membran die zu messenden Drücke angelegt und der Unterschied der beiden gemessen. Bei der Relativdruckmessung wird eine Seite der Membran mit dem Außendruck beaufschlagt und der zu bestimmende Druck im Bezug dazu gemessen. Das führt allerdings dazu, dass Luftdruckschwankungen die Messung beeinflussen. Um äußeren Druckeinflüssen auf die Messung vorzubeugen kann man die Membranrückseite mit einem definierten Druck beaufschlagen und gegen diesen den interessierenden Druck messen. Diese Vorgehensweise wird als Absolutdruckmessung bezeichnet. Ebenfalls zur Absolutdruck-

bestimmung eigenen sich Methoden, die nicht auf dem Vergleich mit anderen Drücken beruhen, sondern bestimmte Materialeigenschaften ausnutzen, wie beispielsweise piezoelektrische oder piezoresonatorische Effekte. [20]

Die zum Druck proportionale Verformung der Membran, die zur Bestimmung des Drucks benutzt wird, kann auf verschiedene Arten bestimmt werden. Ein physikalischer Effekt, der dabei genutzt wird, ist der resistive Effekt. Dieser beruht darauf, dass sich der elektrische Widerstand eines Leiters proportional zur angelegten mechanischen Spannung und der daraus resultierenden Dehnung ändert. Diese Eigenschaft wird beispielsweise bei Dehnmessstreifen (DMS) ausgenützt. Werden nun solche DMS auf die Membran aufgebracht, führt eine Verformung der Membran zu einer Verformung der DMS und somit zu einer, dem Druck proportionalen, Impedanzänderung. Diese kann durch eine Wheatstone'sche Brückenmessschaltung, welche aus zumindest vier DMS besteht, bestimmt werden [20,24,25]. Die piezoresistive Druckmessung basiert ebenfalls auf einer spannungsinduzierten Widerstandsänderung. Hierbei wird allerdings das unter Spannung anisotrope Widerstandverhalten von Halbleitern – zumeist Silizium – ausgenützt. Dieses führt dazu, dass die Widerstandsänderung im Vergleich zu DMS um bis zu 50-mal höher ausfällt. Bei der piezoresistiven Druckmessung werden Siliziumwafer mit Fremdatomen dotiert, um örtlich gezielt die Leitfähigkeit zu beeinflussen. Diese Stellen bilden die piezoresistiven Widerstände. Der Siliziumwafer selbst fungiert hierbei als Membran und wird lokal, je nach gewünschter Druckempfindlichkeit, abgedünnt [20,26] (siehe Abb. 3-6). Auch hier kommt wiederum eine Wheatstone'sche Brückenschaltung zum Einsatz, um die Impedanzänderung der Piezoresistoren zu messen.

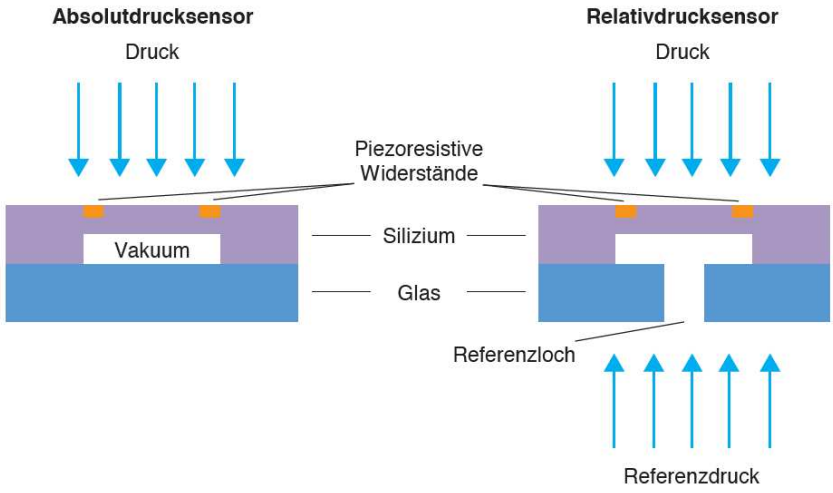


Abb. 3-6: Schematischer Aufbau eines piezoresistiven Drucksensors [26]

Weitere Methoden zur Bestimmung der Membranverformung und des dazu proportionalen Drucks sind die Messung der Änderung der Induktivität oder der Kapazität bei Verformung der Membran

durch geeignete Aufbauten. Hierbei handelt es sich um eine Positionsbestimmung der Membran, aus der sich der Druck berechnen lässt [20,27].

3.2.2 Temperaturmessung

Zur Bestimmung der Temperatur lässt sich eine Vielzahl physikalischer Prinzipien nutzen. So können temperaturabhängige Materialparameter wie Widerstand, Ausdehnung und Eigenfrequenz genutzt werden, oder auch die temperaturabhängige Strahlungsintensität und der thermoelektrische Effekt, der auf der temperaturabhängigen elektrischen Spannung zwischen zwei verschiedenen Metallen beruht [20]. In weiterer Folge wird nur näher auf die, für die Arbeit relevanten, Prinzipien des temperaturabhängigen Widerstandes und des thermoelektrischen Effekts eingegangen.

3.2.2.1 Temperaturabhängiger Widerstand – Pt100

Der Pt100 ist ein nach DIN IEC751 (bzw. DIN EN 60751) genormter Platinwiderstand, dessen Widerstandswert sich temperaturabhängig ändert. Diese Art von Widerständen werden auch als RTD (Resistance Temperature Detector)–Sensoren bezeichnet. Der Grundwiderstand R_0 des Pt100 beträgt dabei 100 Ω bei einer Temperatur von 0 °C. Die Kennlinie des Widerstandes ist in Abb. 3-7 dargestellt und beschreibt den Widerstandswert bezogen auf die Temperatur. Wie aus der Abbildung ersichtlich nimmt der Widerstand mit steigender Temperatur zu, weswegen der Pt100, wie beispielsweise auch Ni100–Widerstände, zu der Gruppe der Kaltleiter gezählt wird. Gegenteilig dazu existieren die NTC (Negative Temperature Coefficient)-Widerstände, deren Widerstandwert mit steigender Temperatur abfällt.

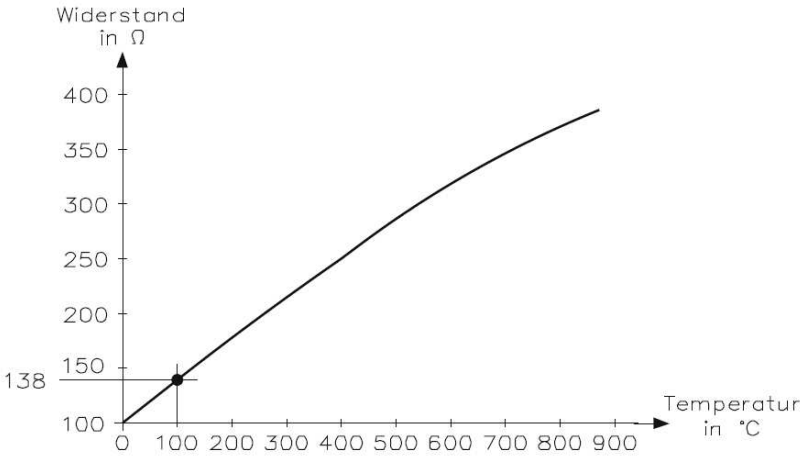


Abb. 3-7: Kennlinie des Widerstandsthermometers Pt100 [24]

Der Messbereich handelsüblicher Pt100 Widerstände liegt im Bereich von -200 °C – +550 °C, wobei mit Sonderausführungen der Messbereich, bei kurzzeitiger Anwendung, bis +700 °C erweitert werden kann [25].

Die Messung des Widerstandwertes R_{Pt} des Pt100 erfolgt, indem ein bekannter, konstanter Messstrom I_0 über den Widerstand geleitet wird und der Spannungsabfall U_m am Widerstand gemessen wird. Folglich kann der Widerstand nach dem Ohm'schen Gesetz [28]

$$R_{Pt} = \frac{U_m}{I_0} \tag{3-1}$$

bestimmt werden.

Die Messschaltung kann dabei, wie in *Abb. 3-8* ersichtlich, als 2-Draht–, 3-Draht–, oder 4-Drahtschaltung ausgeführt sein. Bei der 2-Drahtmessung wird der Widerstand der Leitungen mitgemessen, was bei größeren Leitungslängen schnell zu einem entsprechend großen Messfehler führt. Bei der 3-Drahtschaltung wird, durch das Einbringen eines weiteren Leiterdrahtes zum Anschlussdraht des Widerstandes, ein weiterer Messkreis geschaffen, über den der Widerstand der Leitung ohne Temperaturmesssensor bestimmt werden kann. Dieser Wert wird zur Kompensation des gemessenen Pt100-Widerstandswertes verwendet. Hierzu müssen allerdings Hin- und Rückleitung dieselbe Länge aufweisen. Um das Problem des Leitungswiderstandes zu umgehen, wird bei der 4-Drahtmessung der Spannungsabfall direkt an den Anschlüssen des Temperaturwiderstandes abgegriffen. Da Spannungsmessungen hochohmig durchgeführt werden, fließt dabei über die Messleitungen kein nennenswerter Strom und es kommt somit auch zu keinem, die Messung verfälschenden, Spannungsabfall [20,29].

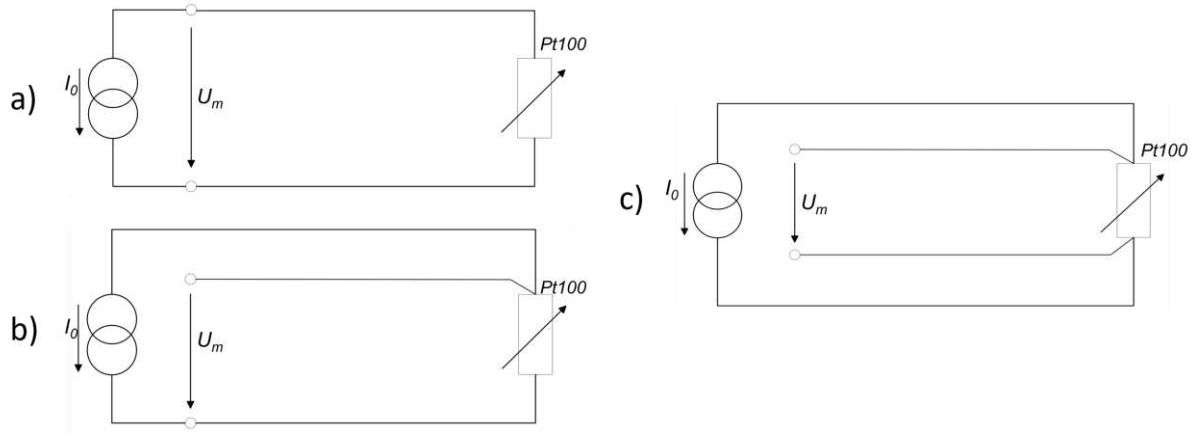


Abb. 3-8: Anschlussarten eines Pt100 Temperaturmesswiderstandes: a) 2–Drahtmessung b) 3–Drahtmessung c) 4–Drahtmessung (in Anlehnung an [20])

Der Zusammenhang des gemessenen Widerstandswertes R_{Pt} mit der Temperatur T kann durch die nachfolgenden Näherungsformeln beschrieben werden [20]. R_0 ist dabei der Grundwiderstand des Pt100 und α, b und c sind empirisch ermittelte Konstanten.

- Temperaturbereich 0 °C –100 °C:

$$R_{Pt} = R_0(1 + \alpha T) \tag{3-2}$$

- Temperaturbereich 100 °C – 700 °C:

$$R_{Pt} = R_0(1 + \alpha T + bT^2) \tag{3-3}$$

- Temperaturbereich unter 0 °C:

$$R_{Pt} = R_0(1 + \alpha T + bT^2 + c(T - 100K)T^3) \tag{3-4}$$

mit $\alpha = 3.85 \cdot 10^{-3}K^{-1}, b = -5.775 \cdot 10^{-7}K^{-2}, c = -4.183 \cdot 10^{-12}K^{-4}$

3.2.2.2 Thermoelektrischer Effekt – Thermoelement

Der Thermoelektrische Effekt, oder auch Seeback-Effekt, beschreibt die in einem elektrischen Leiter auftretende Ladungsverschiebung und die damit einhergehende elektrische Spannung, die durch einen Temperaturgradienten hervorgerufen wird. Wird ein freies Ende eines Leiters erwärmt, erhöht sich dadurch die kinetische Energie der Elektronen und es kommt zur Thermodiffusion der Elektronen hin zum kälteren Ende. Die sich einstellende unterschiedliche Elektronendichte der beiden Enden führt zu einer elektrischen Spannung, die sich proportional zum Temperaturgradienten verhält. Die Abhängigkeit der Spannung vom Temperaturgradienten ist materialspezifisch und verhält sich in weiten Bereichen linear. Um diesen Effekt, in Form eines Thermoelements, technisch nutzbar zu machen, werden zwei unterschiedliche Leitermaterialien durch Löten oder Schweißen an einer Kontaktstelle miteinander verbunden (siehe *Abb. 3-9*). Existiert nun eine Temperaturdifferenz zwischen den verbundenen Leiterenden (Messtelle) und den freien Leiterenden (Vergleichsstelle) kann an der Vergleichsstelle eine Spannung gemessen werden, die aus den unterschiedlichen thermoelektrischen Eigenschaften der beiden Materialien resultiert. Um aus dieser Spannung die Temperatur an der Messstelle berechnen zu können, muss die Temperatur an der Vergleichsstelle bekannt sein. In Laboranwendungen wird dazu häufig ein Eisbad verwendet, mit dessen Hilfe die Temperatur auf 0 °C thermostatisiert wird. In Industrieanwendungen werden Messumformer eingesetzt, die die Temperatur an der Vergleichsstelle mit Hilfe eines Widerstandthermometers bestimmen und ein der Messtemperatur proportionales Ausgangssignal in Strom- oder Spannungsform (zumeist 4-20 mA bzw. 0-10 V) liefern [20,24,25].

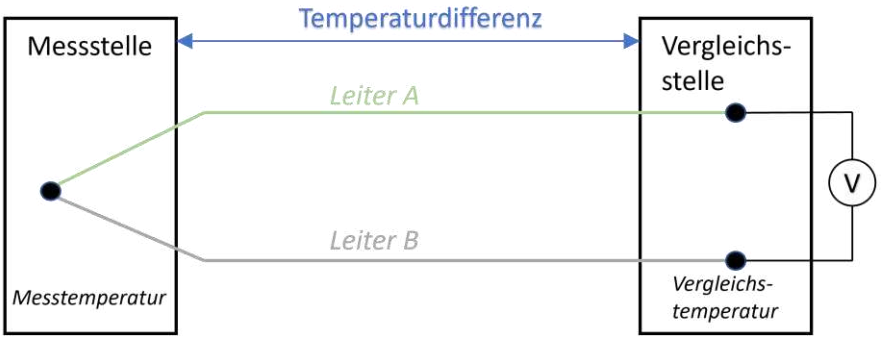


Abb. 3-9: Aufbau eines Thermoelements (in Anlehnung an [25])

In **Tabelle 1** sind einige gängige Thermoelementpaarungen mit den jeweiligen Messbereichen angegeben.

Tabelle 1: Einige gängige Thermoelementpaarungen [20,22]

Typ	Material	Temperaturbereich[°C]
K	NiCr-Ni	-180 – 1350
J	Fe-CuNi	-180 – 750
R	Pt13Rh-Pt	- -50 – 1700
N	NiCrSi-NiSi	- -270 – 1300

3.2.3 Durchflussmessung

Die Durchflussmessung dient meist der Bestimmung der Menge einer Flüssigkeit oder eines Gases, welche einen Kontrollquerschnitt pro Zeiteinheit passiert. Es wird zwischen Volumenstrommessung, bei der das Volumen des zu bestimmenden Materials eruiert wird, und Massenstrommessung, bei der die Masse die Messgröße darstellt, unterschieden. Ist die Dichte und, bei Gasen, Druck und Temperatur der zu messenden Substanz bekannt, kann zwischen Volumenstrom und Massenstrom umgerechnet werden. Es existieren eine Vielzahl an Messprinzipien zur Ermittlung des Durchflusses. Dazu zählen beispielsweise die Messung via Drossel, Verdrängungszähler oder Stauscheibe, oder auch die Messung mittels Ultraschalllaufzeit oder magnetischer Induktion. Zur Messung von gasförmigen Medien eignen sich besonders thermische Messverfahren [20,22]. Bei diesen thermischen Verfahren wird der Massenstrom eines Mediums bestimmt, indem beispielsweise zwei auf konstante Temperatur beheizte Temperatursensoren in das Medium eingebracht werden. Ein Sensor befindet sich dabei in einem strömungsfreien Bereich, um die Temperatur der ruhenden Substanz zu ermitteln, und der andere wird direkt in die Strömung eingebracht. Die infolge der Strömung vom Sensor abgeführte Wärme ist proportional zum Massenstrom des Mediums und kann aus der Differenz der benötigen Energie zum Heizen der beiden Widerstände bestimmt werden [22].

3.2.4 Analog-Digital-Converter

Da Messgrößen für gewöhnlich als analoge Signale vorliegen, für eine computerbasierte Weiterverarbeitung jedoch digitale Signale von Nöten sind, müssen diese zuerst digitalisiert werden. Dies geschieht mit Hilfe von Analog-Digital-Wandlern, oder auch Analog-Digital-Converter (ADC) genannt [25]. Das analoge, elektrische Messsignal eines Sensors wird dabei in erster Instanz durch einen geeigneten Messumformer in einen elektrischen Strom oder eine elektrische Spannung konvertiert. Die Ausgangsgrößen des Messumformers befinden sich standardmäßig im Bereich von 0–20 mA, 4–20 mA oder 0–10 V. Dieser normierte Messwert wird dann mit Hilfe eines ADCs in weiterer Folge in ein proportionales digitales Signal umgewandelt (siehe *Abb. 3-10*) [24].

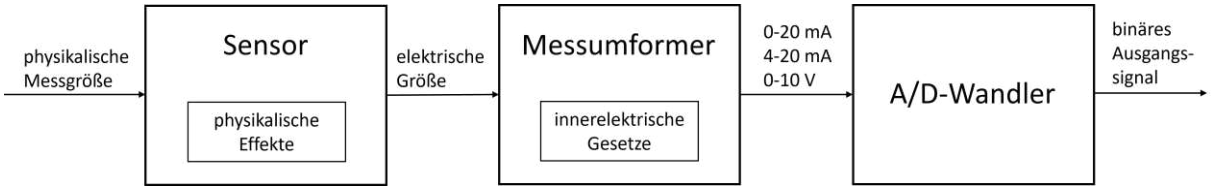


Abb. 3-10: Schematischer Ablauf der Umwandlung einer physikalischen Messgröße in ein digitales Signal (in Anlehnung an [25])

Ein analoges Signal weist laut Definition einen unendlich großen Wertevorrat auf, sprich es kann jeden beliebigen Wert zwischen Maximalwert und Minimalwert annehmen. Ein Rechner hat jedoch nur eine begrenzte Anzahl an Speicherstellen zur Verfügung, um dieses Signal darzustellen. Deshalb muss, um ein Signal digital abzubilden, der Wertebereich dieses analogen Signals in endlich viele diskrete Werte unterteilt werden. Die Anzahl der möglichen Werte, die das digitale Signal annehmen kann, ist dabei durch die Auflösung des ADCs, sprich die Anzahl der ihm zur Verfügung stehenden Bits, bestimmt. Unterteilt man nun – wie in *Abb. 3-11* zu sehen – den Bereich zwischen minimalem und maximalem analogen Messwert in regelmäßige Intervalle, führt das zum Erhalt einer Treppenkurve, wobei jede Stufe durch einen Binärwert repräsentiert wird. Dieser Vorgang wird auch als Quantisierung bezeichnet. Die Höhe einer Stufe ergibt sich aus dem kleinstmöglichen Intervall, das mit der gegebenen Bit-Anzahl dargestellt werden kann, also aus dem least-significant-bit (LSB), dem Bit, das bei Änderung die kleinstmögliche Änderung der Binärzahl verursacht. Ein analoges Signal wird bei Überführung in einen digitalen Wert als der ihm am nächsten liegende Binärwert dargestellt. Das bedeutet, dass das digitale Signal durch die Quantisierung eine relative und sich daraus ergebende absolute Abweichung zu dem ursprünglichen analogen Signal aufweisen kann. Diese Abweichung ist umso größer, je kleiner die Auflösung des ADCs ist. Für einen 3-bit-Wandler beträgt die maximale relative Abweichung beispielsweise $\frac{1}{7} = 0.143$, da der Messbereich in sieben Intervalle, entsprechend den sieben darstellbaren Werten, unterteilt wird. Dies entspricht einem Bereich von ± 0.5 LSB [21,22].

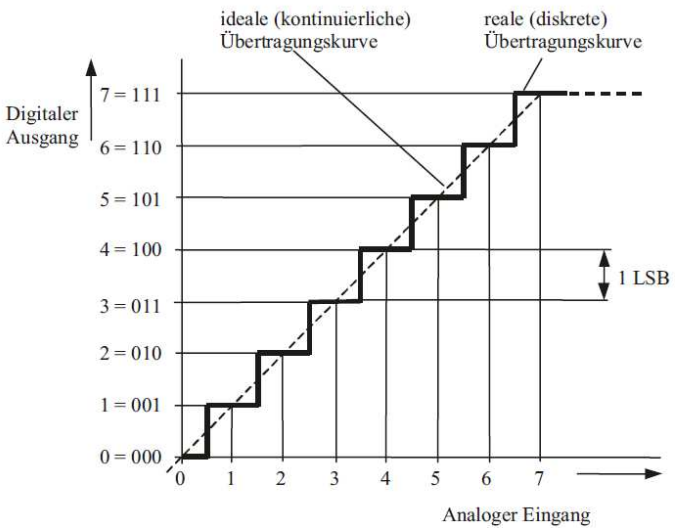


Abb. 3-11: Übertragungskennlinie eines 3-bit-A/D-Wandlers [22]

Ein ADC kann durch seinen inneren Aufbau nicht kontinuierlich arbeiten. Das analoge Signal wird stattdessen in regelmäßigen Zeitabständen abgetastet und in einen digitalen Wert überführt. Der Kehrwert dieses Zeitintervalls wird als Abtastfrequenz des ADCs bezeichnet. Diese Abtastfrequenz muss nur so hoch gewählt werden, dass sich das ursprüngliche Signal aus dem digitalen rekonstruieren lässt. In *Abb. 3-12* ist der Ablauf der zeit- und wertediskreten Digitalisierung eines Messsignals dargestellt [23–25]:

- (1) – ursprüngliches analoges Signal
- (2) – zeitdiskrete Abtastung des Signals
- (3) – Quantisierung des Wertebereichs und Zuordnung der analogen Messwerte zu den diskreten Binärwerten
- (4) – Umrechnung der erhaltenen Binärwerte in die zugeordneten digitalen Messwerte anhand der Kennlinie des ADCs

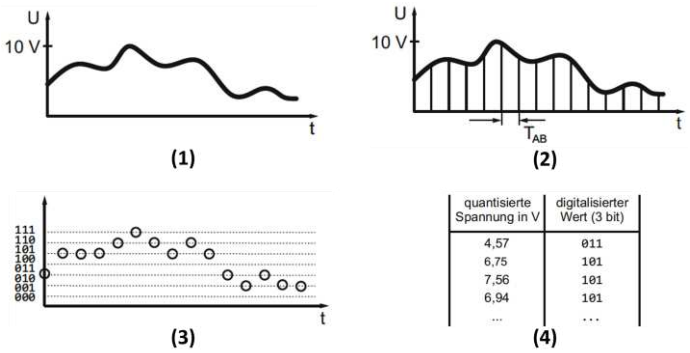


Abb. 3-12: Umwandlung eines analogen in einen digitalen Messwert [25]

Es existieren verschiedene Funktionsprinzipien für ADCs, auf die hier allerdings nicht näher eingegangen wird [21–25].

3.3 Speicherprogrammierbare Steuerung

Auf der zweituntersten Ebene der Automatisierungspyramide (siehe *Abb. 3-13*) sitzt zumeist eine speicherprogrammierbare Steuerung (SPS), deren Aufgabe es ist, durch das Verarbeiten und Auswerten von eingehenden Sensorsignalen die auf der Feldebene stattfindenden Prozesse zu regeln und zu steuern [15].

Abb. 3-13: Vereinfachte Automatisierungspyramide (in Anlehnung an [15])

Unter einer SPS versteht man laut DIN IEC 60050–341 eine „*rechnergestützte programmierte Steuerung, deren logischer Ablauf über eine Programmiereinrichtung, zum Beispiel ein Bedienfeld, einen Hilfsrechner oder ein tragbares Terminal, veränderbar ist*“ [25]. In anderen Worten handelt es sich bei SPSen also um frei programmierbare Steuergeräte, die über extern angeschlossene Geräte programmiert werden können. Diese sind für gewöhnlich modular aufgebaut und lassen sich somit gezielt an die gewünschte Anwendung anpassen.

Die Grundausstattung einer SPS setzt sich aus den folgenden wesentlichen Funktionsbaugruppen zusammen (siehe *Abb. 3-14*) [30]:

- Programmspeicher
- Ein-/Ausgabe-Baugruppen
- Datenspeicher
- Zentraleinheit/CPU

Im Programmspeicher erfolgt die Speicherung des Anwenderprogramms. Darunter versteht man die vom Benutzer, beispielsweise über einen Computer, definierten Anweisungen, die die SPS ausführen soll. Diese Anweisungen werden permanent–zyklisch durchlaufen, wobei sich ein Zyklus aus mehreren Anweisungen zusammensetzt. Am Zyklusbeginn werden die an den Eingabe-Baugruppen anliegenden Signale, sprich Sensorsignale, abgefragt. Diese werden dann in den Datenspeicher überschrieben und dort für die Dauer der Bearbeitung eines Zyklus zwischengespeichert. Dieser Vorgang wird auch als das Erstellen eines Prozessabbilds der Eingabeebene bezeichnet. Das Prozessabbild stellt eine Momentaufnahme der Eingänge beim Einlesevorgang dar. Etwaige Änderungen der Eingangswerte während

des Durchlaufens eines Zyklus können somit erst zu Beginn des nächsten Zyklus erkannt und berücksichtigt werden. Gemäß den im Programmspeicher hinterlegten Anweisungen werden aufgrund der erhaltenen Eingangssignale durch die Zentraleinheit die Zustandsinformationen für die Ausgabeebene berechnet. Die erhaltenen Ergebnisse, auch als Prozessabbild der Ausgabeebene bezeichnet, werden im letzten Schritt zu den Ausgabe-Baugruppen übertragen, wo eine Umsetzung der digitalen Information in die entsprechenden Pegel zur Ansteuerung der Aktorik erfolgt. Nach der Beendigung eines Zyklus beginnt dieser Vorgang erneut. Die Zeit zwischen den Schreibvorgängen auf die physikalischen Ausgänge wird als Zykluszeit der SPS bezeichnet. Diese ist abhängig von der Anzahl der Anweisungen pro Zyklus und befindet sich für gewöhnlich im Bereich zwischen 20 und 50 ms. Durch die geringen Zykluszeiten und die damit einhergehende kurze Dauer der Verarbeitung einzelner Anweisungen kann man, trotz eigentlicher serieller Abarbeitung, von einer „Quasi-Parallelverarbeitung“ der Anweisungen sprechen [23,30].

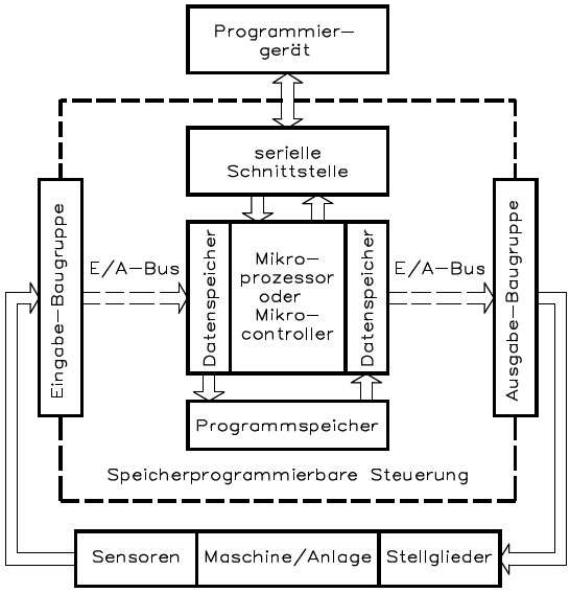


Abb. 3-14: Aufbau einer SPS [30]

Neben den Ein- und Ausgabeeinheiten für Sensorik und Aktorik, die sowohl für digitale als auch analoge Signalverarbeitung verfügbar sind, und etwaigen bereits vorhandenen Standardschnittstellen der SPS gibt es weiters die Möglichkeit die Hardware mit verschiedenen Funktionsbaugruppen – beispielsweise Kommunikationsbaugruppen (Bussysteme, WLAN, …) – zu erweitern, um so das System an den gegebenen Anwendungsfall anzupassen [30].

Die Programmierung einer SPS erfolgt über ein externes Gerät, beispielsweise einen PC. Dieser wird über eine geeignete Schnittstelle, für gewöhnlich Ethernet, mit der Zentraleinheit der SPS verbunden. Die verschiedenen zur Programmierung genutzten Programmiersprachen sind in der Norm IEC 61131 geregelt und lassen sich in grafische und textbasierte Sprachen unterteilen. Zu den grafischen zählen

der Kontaktplan (KOP), die Funktionsbausteinsprache (FBS) und die Ablaufsprache (AS). Die textbasierten Sprachen umfassen die Anweisungsliste (AWL) und den Strukturierten Text (ST) [23].
Der Aufbau eines SPS-Programms ist ebenfalls in der Norm IEC 61131 festgelegt und umfasst unter dem Überbegriff Programm-Organisations-Einheiten (POE) die Bausteine: Programm (PROGRAM), Funktionsbausteine (FUNCTION BLOCK – FB) und Funktion (FUNCTION – FC). Funktionen haben, im Gegensatz zu Funktionsbausteinen, kein Speicherverhalten. Sie arbeiten also nur mit Ein- und Ausgangsvariablen und liefern bei gleichem Inputwert immer denselben Output zurück. Bei Funktionsbausteinen dagegen ist der Outputwert sowohl vom Input als auch vom inneren Zustand des Bausteins abhängig. Weiters steht der innere Zustand eines FB nach Durchlaufen weiterhin zur Verfügung. Ein Programm kann ein oder mehrere FCs und FBs und weitere Operationen beinhalten und erfüllt Aufgaben ohne die Rückgabe von Parametern. Programme wiederum sind einem TASK zugeordnet, durch den das Laufzeitverhalten festgelegt wird. Ein TASK kann aus mehreren Programmen bestehen und wird pro SPS-Zyklus einmal durchlaufen. Sind mehrere unterschiedliche TASKs vorhanden werden diesen Prioritäten zugewiesen, um die Reihenfolge der Abarbeitung und die Wichtigkeit einzelner Prozesse festzulegen. Um mehrere TASKs definieren und abarbeiten zu können ist eine multitaskfähige Steuerung vonnöten [21,23].

3.4 Human–Machine Interaction

Mit zunehmendem technologischen Fortschritt und immer weiter steigendem Automatisierungsgrad wird auch die Kommunikation zwischen Mensch und Maschine immer wichtiger. Stichwort hierbei ist die „Human-Machine Interaction“ (HMI), sprich die Möglichkeit der Interaktion zwischen Mensch und Maschine. Diese wird bezüglich der Automatisierungspyramide der Prozessleitebene zugeordnet und sitzt damit eine Ebene über der Steuerungsebene (*Abb. 3-15*). Die Schnittstelle zur Kommunikation bildet dabei ein sogenanntes „User-Interface“ (UI). Unabhängig davon, welche Art von UI zum Einsatz kommt, sollte auf alle Fälle darauf geachtet werden, dass es möglichst benutzerfreundlich, kosteneffizient und flexibel gestaltet wird [31,32].
Ein UI setzt sich zum einen aus einer Software und zum anderen aus der dazugehörigen Hardware zusammen. Die zur Kommunikation benötigte Hardware schließt dabei unter anderem Bildschirme, Mäuse, Tastaturen und Touchscreens mit ein, die einerseits der Eingabe von Befehlen dienen und es andererseits dem Benutzer ermöglichen Prozessdaten zu überwachen oder eine Rückmeldung auf getätigte Aktionen und etwaige Fehlerbenachrichtigungen zu erhalten [32].

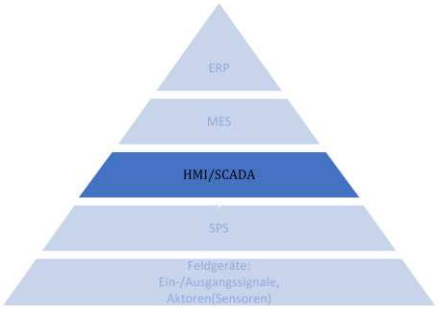


Abb. 3-15: Vereinfachte Automatisierungspyramide (in Anlehnung an [15])

Im Verlauf der letzten Jahre hat sich eine Vielzahl verschiedener Möglichkeiten zur Interaktion des Menschen mit Maschinen etabliert. Eine Möglichkeit der Einordnung ist beispielsweise die Aufteilung in textbasierte, grafische und neu aufkommende Interfaces. Zu den etablierten textbasierten Interfaces zählt unter anderem das „Command Line Interface“, bei dem die Eingabe von Befehlen über eine Kommandozeile mittels Tastatur erfolgt. Dieses wird beispielsweise bei der Distribution Linux primär zur Steuerung verwendet. Die am weitesten verbreitete Mensch-Maschine Schnittstelle ist das, zu den grafischen Interfaces zählende, Graphical User Interface (GUI). Hierbei steht dem Nutzer ein Bildschirm mit grafischer Benutzeroberfläche zur Verfügung, bei der durch Tastatur, Maus oder auch Touchpad die Eingabe von Befehlen vorgenommen werden kann. Weiters dient die Bildschirmanzeige der Ausgabe von Informationen und Meldungen. Neben diesen bereits seit längerem etablierten Methoden haben sich in den vergangenen Jahren weitere Möglichkeiten zur Interaktion mit Maschinen entwickelt. So ist die Kommunikation mittels Spracherkennung, durch Smartphones und weitere „smarte“ Geräte, bereits im Alltag integriert. Durch Fortschritte bei der Bilderkennung sind auch gesten- und augenbewegungsgesteuerte Interaktionen möglich. Ein weiteres Beispiel für eine neuartige Schnittstelle ist das Brain Signal Interface (BCI). Dabei wird die Gehirnaktivität des Benutzers in Echtzeit gemessen und aufgrund von Vergleichsdaten Befehle ausgeführt. Diese Art von Interface bietet vor allem Menschen mit eingeschränkter Mobilität die Chance gewisse Dinge ohne den Einsatz von Muskelkraft bewerkstelligen zu können [31,32].

3.4.1 HMI im Kontext der SPS

Bezogen auf die im *Kapitel 3.3* beschriebene speicherprogrammierbare Steuerung spielt, was Human-Machine Interaction anbelangt, vor allem die Visualisierung eine große Rolle. Diese stellt ein GUI dar und dient einerseits dem Beobachten und andererseits der Steuerung von Prozessen. Moderne SPSen bieten verschiedene Möglichkeiten zur Visualisierung [23]:

- Visualisierung mittels Panel:

Dabei kommt ein Bildschirm zum Einsatz, der gegebenenfalls weitere Bedienelemente aufweist oder via Touchpad bedient werden kann. Dieser wird über eine geeignete Schnittstelle mit der SPS verbunden. Die zur Visualisierung benötige Software wird dabei direkt auf das Panel geladen.

- Visualisierung mittels PC:
  Auf dem zur Visualisierung verwendeten PC muss ein spezielles Programm implementiert werden, das es dem Benutzer erlaubt, in einem Entwicklungsmodus ein an die Bedürfnisse angepasstes GUI zu erstellen und in einem Runtimemodus die gewünschten Prozessdaten darzustellen und Eingaben vorzunehmen. Die Anbindung an die SPS kann beispielsweise über Ethernet oder einen Feldbus erfolgen.
- Target-Visualisierung:
  Hierbei läuft die Visualisierungssoftware direkt auf der SPS. Es werden lediglich ein Bildschirm und etwaige Steuerelemente oder ein Touchpad zur Anzeige und Steuerung benötigt.
- Web-Visualisierung:
  Darunter versteht man die grafische Darstellung der SPS-Prozessdaten mittels einem Standard-Webbrowser (Internet-Explorer, Chrome, Firefox, …). Durch die Verbreitung des TCP/IP-Protokolls in der Automatisierungstechnik gewinnt diese Art der Visualisierung immer mehr an Bedeutung. Vorteil hierbei ist die flexible örtliche Gestaltung von Bedien- und Beobachtungsterminals, da prinzipiell jedes mit dem lokalen Netzwerk verbundene Endgerät als solches dienen kann.

3.5 Retrofitting

Wie schon in *Kapitel 2* beschrieben sieht sich die moderne Industrie, um konkurrenzfähig zu bleiben, immer mehr mit der Notwendigkeit der Digitalisierung ihrer Prozessabläufe über die gesamte Wertschöpfungskette hinweg konfrontiert. Diese, unter dem Begriff Industrie 4.0 zusammengefasste, Bestrebung baut auf folgende Trends auf [33]:

- Neue Produktionsmodelle und groß angelegte Individualisierung:
  Anstelle einer Massenproduktion auf Vorrat („Made-to-Stock“) tritt immer mehr eine an die Kundenwünsche angepasste individuelle Produktion im großen Maßstab („Made-to-Order“, „Configure-to-Order“ oder „Engineering-to-Order“).
- „Reshoring“ der Produktion:
  Im Zuge der dritten industriellen Revolution haben viele westliche Länder ihre Produktionsstandorte in Drittstaaten verlegt, um kostengünstiger produzieren zu können („off-shoring“).

Die Entwicklungs- und Designstandorte hingegen wurden im Ursprungsland belassen. Dieser Prozess wird nun im Zuge der vierten industriellen Revolution zunehmend umgekehrt („re-shoring“), da sich durch den fortschreitenden Automatisierungsgrad die Personalkosten sukzessive verringern lassen.

- „Proximity Sourcing“:
  Es wird darauf abgezielt die Transportwege für Rohstoffe und Halbzeuge möglichst kurz zu halten, um Kosten und Zeit zu sparen. Die damit einhergehende notwendige Verarbeitung von großen Mengen an Daten bezogen auf die Wertschöpfungskette und die dahinterstehende Logistik ist ein Hauptantrieb für die Industrie 4.0.
- Menschenzentrierter Produktionsprozess:
  Der Mensch bleibt als Arbeitskraft wesentlicher Bestandteil des Produktionsprozesses. Allerdings geht der Trend weg von körperlich intensiven Arbeiten, die immer mehr von Robotern und Maschinen übernommen werden, hin zu wissensbasierten Aufgaben, die Kenntnisse über den gesamten Produktionsprozess verlangen. Weiters steht die Kooperation zwischen Maschine und Mensch stark im Fokus der Industrie 4.0.

Diese Ziele werden durch Zuhilfenahme neuer und sich permanent weiter entwickelnder Technologien erreicht. Dazu zählen beispielsweise die teilweise ebenfalls schon in *Kapitel 2* beschriebene *künstliche Intelligenz*, *Big Data*, das *IIoT* und die *CPPS* [33].

Auch wenn das Konzept Industrie 4.0 einen Prozess darstellt, der nicht von heute auf morgen umgesetzt werden kann, so gibt es doch schon eine Vielzahl konkreter Anwendungsfälle, bei denen Methoden der vierten industriellen Revolution gewinnbringend eingesetzt werden. Dazu gehören [33]:

- Flexibilität von Automatisierungsstrukturen und Konfigurationen:
  Möglichkeit der schnellen und individuellen Anpassung von Produktionslinien an verschiedene und sich schnell ändernde Anforderungen.
- Vorhersagen von notwendigen Wartungsarbeiten:
  Anstelle von vorbeugenden Wartungen, die in fixen Zeitintervallen durchgeführt werden, tritt vermehrt eine, durch Technologien der Industrie 4.0 ermöglichte, Vorhersage von notwendigen Wartungen. Dadurch kann die Lebensdauer von Anlagekomponenten besser ausgenützt, die Wartungszeiten optimal gestaltet und Wartungsaufwand und -kosten verringert werden.
- „Zero Defect Manufacturing“:
  Mit zunehmender Menge an zur Verfügung stehenden Daten des physikalischen Produktionsprozesses und der daraus resultierenden Produkte bieten sich immer bessere Möglichkeiten

für die Qualitätskontrolle. Dazu gehört beispielsweise das Identifizieren von fehlerhaften Produkten und Fehlerquellen. Es werden selbstlernende Systeme implementiert, die aufgrund vorangegangener Probleme den Produktionsprozess selbsttätig anpassen und optimieren.

- Digitale Simulation und Digitaler Zwilling („Digital Twin“):
  Um beispielsweise alternative Prozesskonfigurationen zu testen, ohne dabei den Betrieb einer Anlage unterbrechen zu müssen, wird vermehrt auf digitale Modelle zurückgegriffen. Der Digitale Zwilling stellt dabei eine möglichst exakte digitale Repräsentation einer physikalischen Einheit – eines Produkts, eines Prozesses, etc. – dar.
- Informationsfluss über die gesamte Wertschöpfungskette:
  Der Austausch von Daten ist, durch eine ganzheitliche Verknüpfung aller in den Wertschöpfungsprozess eingebundenen Instanzen, nicht mehr nur auf einzelne Schnittstellen dieser Instanzen begrenzt. Durch die permanent zur Verfügung stehenden und abrufbaren Informationen lässt sich ein effizienter und kostensparender Produktionsprozess verwirklichen.
- Arbeitssicherheit:
  Der Einsatz von „Virtual Reality“ und „Augmented Reality“ kann beispielsweise dazu genutzt werden, Arbeitskräfte durch den Einsatz von digitalen Modellen gefahrlos einzuschulen.

Grundvoraussetzung für die Implementierung dieser und weiterer Anwendungen ist eine geeignete Infrastruktur. Dabei kommt das Problem auf, dass viele Produktionsanlagen nicht für die modernen Ansprüche, die die Industrie 4.0 mit sich bringt, ausgelegt sind. Beispielsweise betrug im Jahr 2018 das durchschnittliche Maschinenalter in Frankreich, einem der wichtigsten Industriestandorte Europas, 19 Jahre und in den USA 10 Jahre [34]. Die Neuanschaffung von Maschinen ist allerdings mit hohen Kosten verbunden, weswegen Betriebe vermehrt darauf setzen, veraltete Anlagen aufzurüsten, um sie so an die modernen Ansprüche anzupassen. Dieser Vorgang wird als **Retrofitting**, beziehungsweise im Kontext der Industrie 4.0 als **„Smart Retrofitting“**, bezeichnet [13,34].

Das Bestreben des Smart Retrofittings ist es mit minimalem finanziellen und zeitlichen Aufwand bestehende Anlagen an die Anforderungen der Industrie 4.0 anzupassen. Ein Hauptaugenmerk liegt dabei auf dem Sammeln von Daten, wobei sowohl bereits vorhandene als auch neue (smarte) Sensoren zum Einsatz kommen können. Ein weiterer wichtiger Punkt ist die Möglichkeit der Kommunikation und des Datenaustausches der einzelnen Geräte untereinander oder auch mit dem Menschen über externe Geräte wie beispielsweise Tablets oder Computer. Auch das Auswerten von Daten, um daraus nutzbare Informationen zu gewinnen, steht im Fokus des Smart Retrofittings [35].

Bei der Umsetzung dieser Punkte sehen sich Unternehmen häufig mit zwei Hauptherausforderungen konfrontiert [34]:

1. Aktuelle Industrieanlagen basieren zumeist auf, über Jahre hinweg gewachsenen, heterogenen Strukturen und Architekturen hinsichtlich der verwendeten Technologien wie Kommunikationsprotokolle, Steuerungssysteme und elektrischen bzw. mechanischen Komponenten. Das erschwert das Einbinden der einzelnen Teilsysteme in ein einheitliches Gesamtsystem, oder macht es sogar unmöglich.
2. Es ist gut ausgebildetes Fachpersonal vonnöten, um die verschiedenen Aspekte der Industrie 4.0 hinreichend abzudecken und ein fehlerfreies Ineinandergreifen der sensiblen Einzelsysteme zu ermöglichen.

Dem CPS kommt in der Industrie 4.0 und somit auch beim Smart Retrofitting eine tragende Rolle zu. So dienen interne und externe eingebettete Systeme nicht nur dem Sammeln von Daten, sondern bieten auch die Möglichkeit verwendete alte und inhomogene Kommunikationsprotokolle einzelner Anlagen in ein einheitliches, zeitgemäßes Protokoll – wie beispielsweise OPC–UA oder MQTT – zu konvertieren, um eine ganzheitliche Vernetzung durch ein IIoT zu ermöglichen [13,34].

Häufig ist das endgültige Ziel der Aufbau einer „Smart Factory“, also einer Produktionsanlage, deren Maschine, Produkte und Prozesse zu großen Teilen selbsttätig miteinander kommunizieren und sich aufeinander abstimmen können. Um das zu erreichen ist es sinnvoll die nötigen Schritte in kurz-, mittel- und langfristige Projekte einzuteilen und somit die Produktionsumgebung kontinuierlich an das gewünschte Resultat heranzuführen [34].

3.6 Induktionserwärmung

Im Folgenden werden zusammenfassend die physikalischen und mathematischen Grundlagen der Induktionserwärmung beschrieben. Weiters werden die für diese Arbeit relevanten benötigten Gerätschaften sowie typische technische Anwendungsfälle der Induktionserwärmung dargelegt.

3.6.1 Elektromagnetische Induktion

Um die Induktionserwärmung zu verstehen, muss zuerst der Funktionsmechanismus der elektromagnetischen Induktion allgemein betrachtet werden.

Wird eine elektrische Ladung Q in einem Magnetfeld $\vec{B}$ mit der Geschwindigkeit $\vec{v}$ bewegt, wirkt eine Kraft $\vec{F}_m$ auf diese Ladung. Diese Kraft wird als Lorentzkraft bezeichnet und lässt sich mathematisch wie folgt beschreiben [36]:

$$\vec{F}_m = Q \cdot (\vec{v} \times \vec{B}) \tag{3-5}$$

Bewegt man nun nicht nur eine Ladung, sondern einen ganzen Leiter in einem Magnetfeld, werden die freien Elektronen in diesem Leiter gemäß der Lorentzkraft abgelenkt. Das führt dazu, dass sich ein Ladungsgradient im Leiter einstellt, woraus ein elektrisches Feld $\vec{E}$ und somit eine induzierte Spannung u_i resultiert (siehe *Abb. 3-16*) [36].

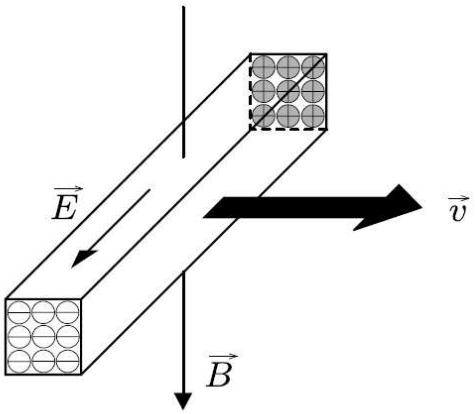


Abb. 3-16: Entstehung des elektrischen Feldes durch Induktion [36]

Betrachtet man nun denselben Leiter als Teil eines geschlossenen Kreises, in dem ein Strom fließen kann, so lässt sich diese Spannung in der einfachsten Form des Induktionsgesetzes beschreiben als [36]:

$$u_i = -N \frac{d\Phi}{dt} \tag{3-6}$$

Wobei Φ den magnetischen Fluss bezeichnet und die induzierte Spannung somit der negativen Änderung des magnetischen Flusses entspricht. Diese ändert sich proportional mit der Anzahl N der Windungen der Leiterspule. Eine Spannung wird also einerseits induziert, wenn sich ein Leiter in einem Magnetfeld bewegt und andererseits ebenfalls, wenn der Leiter ruht und das Magnetfeld sich ändert, da sich in beiden Fällen der magnetische Fluss durch den Leiter zeitlich ändert [36].
Hier sei noch die Integrale Form des Induktionsgesetztes – auch als 2. Maxwellsche Gleichung bezeichnet – erwähnt, deren exakte Herleitung in der Fachliteratur ausreichend behandelt wird und hier daher darauf verzichtet wird [36]:

$$\oint \vec{E} \cdot d\vec{s} = \oint (\vec{v} \times \vec{B}) \cdot d\vec{s} - \iint \frac{\partial \vec{B}}{\partial t} \cdot d\vec{A} \tag{3-7}$$

Die linke Seite beschreibt die induzierte Spannung, das Wegintegral der rechten Seite die durch Bewegung eines Leiters verursachte Induktion und das Flächenintegral die Induktion in Ruhe, die durch eine Änderung des Magnetfeldes verursacht wird [36].

3.6.2 Grundlagen der Induktionserwärmung

Die Induktionserwärmung macht sich das Prinzip der zuvor beschriebenen elektromagnetischen Induktion zunutze. Dabei wird ein Leiter – als Induktor bezeichnet – mit Wechselstrom durchflossen. Dadurch baut sich um diesen Leiter ein elektromagnetisches Feld auf, das in der Frequenz des Wechselstroms um einen Nullpunkt schwankt. In dieses Feld wird ein elektrisch leitendes Werkstück eingebracht. Die in das Werkstück induzierte Spannung hat einen Wechselstrom zur Folge, der in entgegengesetzter Richtung zum Strom im Induktor fließt. Durch den erzeugten Stromfluss entsteht eine Wärme Q_i im Werkstück. Diese wird als Wirbelstrom– oder Widerstandswärme bezeichnet. Bei nicht ferromagnetischen Werkstoffen ist diese Wärme vom Widerstand R_Q, dem Strom I_i und der Zeit t abhängig und folgt dem Joulschen Gesetz [37]:

$$Q_i = {I_i}^2 R_Q t \tag{3-8}$$

Die Wärme bei der Induktionserwärmung wird also unmittelbar im Werkstück erzeugt und muss nicht erst über Konvektion, Strahlung oder Wärmeleitung übertragen werden. Bei ferromagnetischen Werkstoffen kommt noch die Wärme hinzu, die sich aus der stetigen Ummagnetisierung des Werkstoffes ergibt. Diese wird als Ummagnetisierungs– oder Hysteresewärme bezeichnet. Da allerdings die Wirbelstromwärme mit höherer Stromfrequenz um ein Vielfaches stärker steigt als die Hysteresewärme, ist die Hysteresewärme für die praktische Anwendung zumeist nicht von Bedeutung. Weiters verlieren ferromagnetische Werkstoffe oberhalb der Curie–Temperatur ihre ferromagnetischen Eigenschaften [37].

Die Form des Induktors wird an die des Werkstücks angepasst, um eine möglichst gleichmäßige und effiziente Erwärmung zu erhalten und besteht, aufgrund der hohen elektrischen Leitfähigkeit, für gewöhnlich aus Kupfer. Bei runden Werkstücken wird der Induktor beispielsweise als Leiterschleife oder

mehrwindige Spule ausgeführt, die das Werkstück umschließt [37]. In *Abb. 3-17* ist der Wirkmechanismus der Induktionserwärmung dargestellt. Die sich ergebenden, für die Erwärmung verantwortlichen Wirbelströme I_{WS} sind in Rot eingezeichnet.

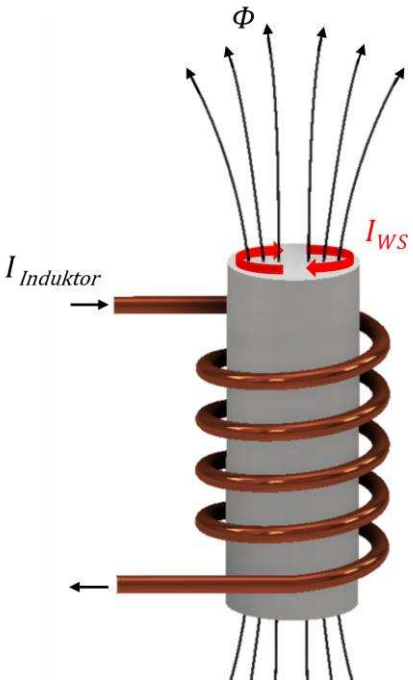


Abb. 3-17: Induktive Erwärmung eines Metallzylinders (in Anlehnung an [37])

Die durch höherfrequente Wechselspannungen induzierten Ströme fließen nicht gleichmäßig im Werkstück verteilt, sondern zum größten Teil entlang seiner Oberfläche. Das rührt daher, dass der im Bauteil fließende Wechselstrom durch Selbstinduktion wiederum Wirbelströme erzeugt, die sich mit dem primären Strom überlagern und somit den Widerstand im Inneren erhöhen. Dieses Phänomen wird Skineffekt genannt. Die Verteilung des Stromflusses hängt dabei von der Erregerfrequenz sowie den elektrischen und magnetischen Eigenschaften des Werkstoffes ab. Allgemein nimmt die Stromdichte J zum Werkstückkern hin exponentiell ab – umso schneller, je höher die Frequenz ist. Der Abstand vom Rand, an dem die Stromdichte auf 37% ihres Maximalwertes J_0 abgesunken ist, wird als Eindringtiefe δ bezeichnet (siehe *Abb. 3-18*). Die Leistungsdichte wiederum beträgt hier nur mehr ca. 14% des Oberflächenwertes. Daraus folgt, dass 86% der Leistung für die Erwärmung des Oberflächenbereichs bis zur Eindringtiefe aufgewendet werden. Höhere Frequenzen eignen sich somit beispielsweise für das Oberflächenhärten, wogegen niedrigere Frequenzen für das Durchwärmen eines Bauteils genutzt werden können. Folgende Gleichung gibt den Zusammenhang der Eindringtiefe mit dem spezifischen elektrischen Widerstand ρ, der relativen magnetischen Permeabilität μ_r und der Frequenz f an [38]:

$$\delta = \frac{1}{2\pi}\sqrt{\frac{\rho * 10^7}{\mu_r f}} \approx 503\sqrt{\frac{\rho}{\mu_r f}} \tag{3-9}$$

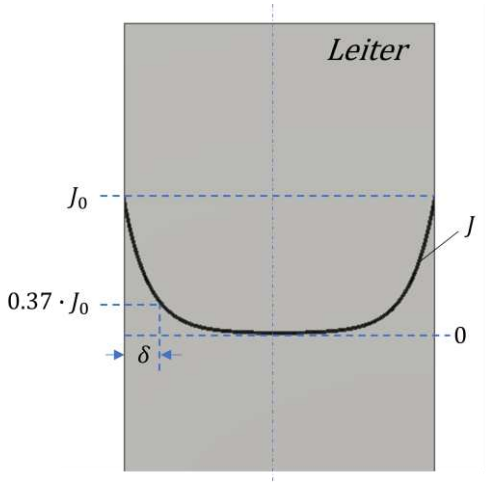


Abb. 3-18: Stromdichteverteilung und Eindringtiefe aufgrund des Skineffekts (in Anlehnung an [37])

Der Aufbau einer Induktionsanlage setzt sich aus einem Leistungs– und einem Arbeitskreis zusammen. Der Leistungskreis besteht dabei aus dem Netztransformator, dem Frequenzumrichter und der Kondensatorbatterie. Der Induktor und das Werkstück bilden den Arbeitskreis. Ein Hochfrequenz-Transformator trennt Arbeits- und Leistungskreis galvanisch voneinander [37] (siehe *Abb. 3-19*).

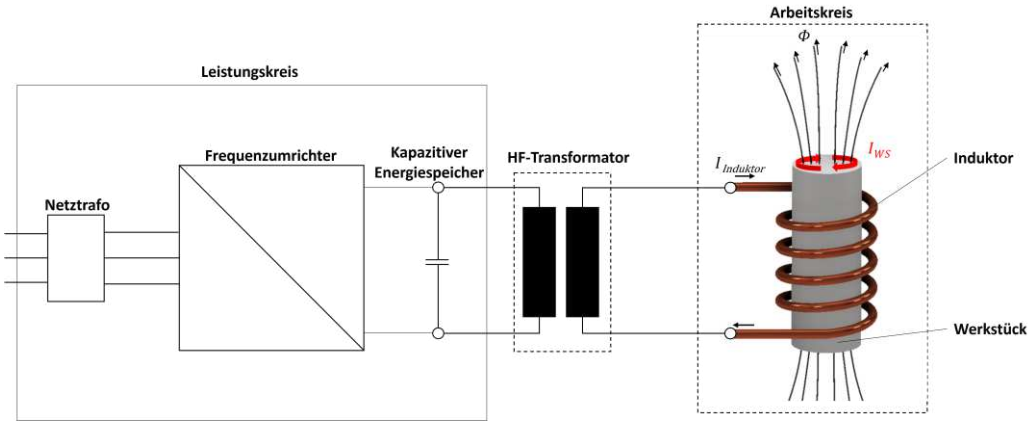


Abb. 3-19: Aufbau einer Induktionsanlage (in Anlehnung an [39])

Der Netztransformator wandelt die Netzfrequenz in die Betriebsspannung des Frequenz-Umrichters um. Dieser wiederum wandelt die vom Transformator erhaltene Dreiphasen-Wechselspannung zunächst in einen Gleichstrom und in weiterer Folge in einen einphasigen Wechselstrom mit der benötigten Betriebsfrequenz um. Die Kondensatorbatterie, als kapazitiver Energiespeicher, bildet zusammen mit dem Induktor den Schwingkreis. Der Induktor wiederum überträgt die zur Verfügung gestellte Leistung auf das Werkstück. Der Induktor selbst ist neben der Eigenerwärmung durch den Stromfluss

auch einer Strahlungserwärmung durch das Werkstück ausgesetzt und wird daher, wie auch die anderen Komponenten der Anlage, mittels Wasserkühlung gekühlt [37].
Die Einsatzbereiche der Induktionserwärmung sind vor allem die Wärmebehandlung und das Induktionsschmelzen, aber auch die Erwärmung von Werkstoffen zur anderweitigen Weiterverarbeitung [37,39].

3.6.3 Die BH-Kurve

Zur Charakterisierung ferromagnetischer Materialien wird in vielen Fällen deren Hysteresekurve herangezogen. Diese wird beispielsweise am MCL im Zuge von Versuchsdurchführungen am Induktionsprüfstand häufig ermittelt. Sie stellt den Zusammenhang zwischen der magnetischen Flussdichte B und der magnetischen Feldstärke H, auch magnetische Erregung genannt, dar [28]. Ein typischer, schematischer Verlauf einer solchen Kurve ist in *Abb. 3-20* dargestellt. Betrachtet man ein unmagnetisiertes, ferromagnetisches Material, das keiner magnetischen Erregung ausgesetzt ist, so wird dessen Zustand durch Punkt 0 der Abbildung beschrieben. Wird dieses Material nun einer magnetischen Feldstärke ausgesetzt und diese immer weiter erhöht, so ändert sich dadurch die magnetische Flussdichte im Material. Diese Änderung folgt dabei der sogenannten Neukurve, welche durch die strichlierte Linie im Diagramm beschrieben wird. Die Magnetisierung kann dabei bis zum Erreichen einer Sättigungsmagnetisierung B_s zunehmen. Wird die magnetische Feldstärke nachfolgend wieder auf null reduziert (Punkt 2), bleibt eine Restmagnetisierung B_r (Remanenz) im Material zurück. Um dieser Remanenz entgegenzuwirken und die magnetische Flussdichte wiederum auf null zu bringen, muss eine magnetische Feldstärke H_c(Koerzitivfeld) in die entgegengesetzte Richtung als zuvor aufgebracht werden (Punkt 3). Die Höhe des Koerzitivfelds ist dabei ein wichtiger Kennwert vor allem weicher ferromagnetischer Materialien, da die Energieverluste bei der Magnetisierung mit der Fläche, die die Hysteresekurve überstreicht, zusammenhängen [28,40].

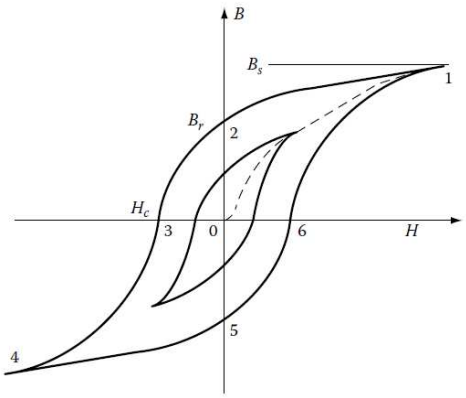


Abb. 3-20: Schematische Darstellung einer Hysteresekurve eines ferromagnetischen Materials [28]

4 Der Induktionsprüfstand

Der Induktionsprüfstand am Material Center Leoben (MCL) setzt sich aktuell aus mehreren Einzelkomponenten zusammen, die unabhängig voneinander arbeiten und bedient werden müssen. Ziel ist es diesen Prüfstand einem Retrofitting zu unterziehen und im Zuge dessen eine zentrale Steuereinheit zu implementieren, die die vorhandenen proprietären Komponenten miteinander verknüpft und eine automatisierte Versuchsdurchführung bzw. Wärmebehandlung ermöglicht. Weiters soll die Datenerfassung und –verarbeitung zentralisiert und vereinheitlicht werden.

4.1 Status Quo

Die Einzelkomponenten des Prüfstandes sind in *Abb. 4-1* ersichtlich und im Folgenden kurz beschrieben:

- Induktionsanlage:
  Dabei handelt es sich um eine induktive Härteanlage der Firma „Ideal Thermal Processes GmbH“ mit Baujahr 2016. Als SPS ist eine Simatic S7–1500 verbaut. Zur Bedienung und zur Prozessüberwachung steht ein 12” Touchdisplay zur Verfügung. Der benötige Wechselstrom für den Induktor wird von einem Mittelfrequenzgenerator „MS15“ der Firma „ThermProTEC“ bereitgestellt.
- Magnetjoch mit Linearantrieb (Linearantrieb nicht in der Abbildung ersichtlich):
  Das Magnetjoch dient der Messung magnetischer Eigenschaften und kann mittels Linearantrieb, der manuell bedient wird, an die Probe herangefahren werden. Das Joch besteht aus einer Primär- und zwei Sekundärspulen. Wird die Primärspule bei der Messdurchführung mit einer Wechselspannung beaufschlagt, um die Probe anzuregen, spricht man von einer „aktiven Messung“. Wird hingegen nur die Systemantwort der Probe aufgrund der Anregung des Induktors gemessen, ohne dass die Primärspule aktiv ist, spricht man von einer „passiven Messung“. Die Sekundärspulen dienen zur Aufnahme der Messsignale.
- Signalgenerator mit Verstärker:
  Der Signalgenerator der Firma „GWInstek“ mit der Typenbezeichnung „AFG–3021“ stellt die benötigte Messfrequenz und Wellenform für das Magnetjoch bei aktiver Messung bereit. Der nachgeschaltete Verstärker wird verwendet, um die Ausgangsspannung des Signalgenerators auf den benötigten Wert einzustellen. Sowohl der Signalgenerator als auch der Verstärker werden manuell bedient.

- Oszilloskop:

  Das digitale Speicheroszilloskop „Pico 5442D MSO“ der Firma „Pico Technology“ dient zur Messung und Aufzeichnung der teils hochfrequenten Spannungen und Ströme der Sekundärspulen des Magnetjochs. Dabei sind maximal vier Messkanäle zum Anschluss der Tastköpfe verfügbar. Im weiteren Verlauf wird der Produktname „Picoscope“ zur Bezeichnung dieses Messgerätes verwendet.

- Chessel mit externen Thermoelementen (nicht in der Abbildung ersichtlich):

  Externe Thermoelemente des Typs K kommen zum Einsatz, um Temperaturen zu messen, die nicht von der Induktionsanlage erfasst werden. Als Messgerät wird ein Eurotherm 6100A Grafikschreiber verwendet, der mit dem Laptop verbunden werden kann.

- Laptop:

  Dieser dient als zentrales Element, auf den, nach der Beendigung einer Messung, alle erfassten Daten übertragen werden. Weiters erfolgt hier die manuelle Weiterverarbeitung der erhaltenen Messdaten.

Abb. 4-1: Induktionsprüfstand

Bei der Durchführung eines Versuchs wird zuerst eine metallische Rundstabprobe in die Induktionsanlage eingebracht und das gewünschte Wärmebehandlungsrezept via USB–Stick an die Anlage übergeben. Die Tastköpfe des Picoscope werden mit den Spulen des Magnetjochs verbunden. Die externen Thermoelemente werden platziert und mit dem Grafikschreiber verbunden. Nach dem Start des Versuchs beginnt die Anlage mit der Aufzeichnung der Anlagenparameter. Zur Durchführung einer Messung mit dem Joch wird dieses durch händische Bedienung des Linearantriebs an die Probe herange-

fahren. Durch manuelles Einschalten des Verstärkers beginnt die Primärspule des Jochs die Probe anzuregen. Die erhaltenen Signale der Sekundärspulen können auf dem Laptop, der mit dem Picoscope verbunden ist, über die Software „PicoLog6“ betrachtet werden. Zum gewünschten Zeitpunkt kann manuell eine Aufzeichnung mit nachfolgender Speicherung der Messwerte gestartet werden. Nach Messungsende erhält man so drei voneinander unabhängige Datensätze von der Anlage, vom Picoscope und vom Grafikschreiber. Diese Datensätze liegen in unterschiedlichen Formaten vor und sind zeitlich nicht miteinander synchronisiert.

In *Abb. 4-2* ist der Ablauf der aktuellen Messdatenerfassung schematisch dargestellt und im Folgenden noch einmal kurz zusammengefasst:

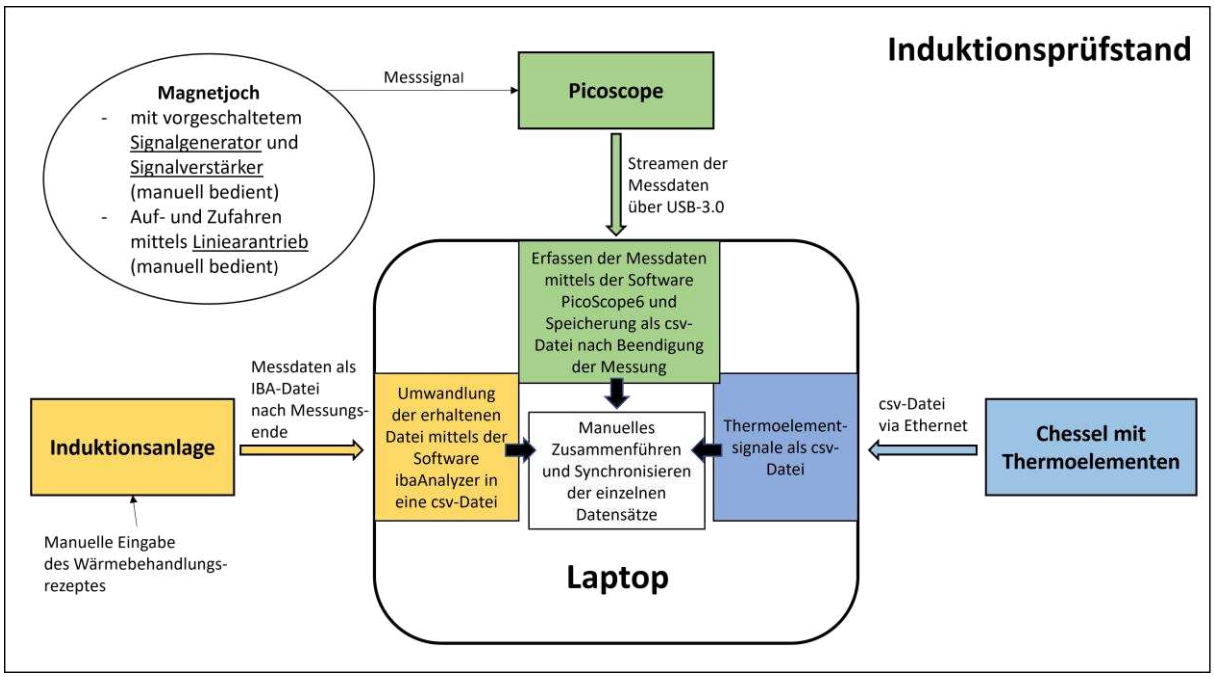


Abb. 4-2: Ablauf der aktuellen Messdatenerfassung

- Die Messgrößen der Induktionsanlage werden durch die Anlagen-SPS erfasst und stehen nach Beendigung einer Messung als IBA-Datei zur Verfügung, die auf den Laptop übertragen werden kann. Die von der Anlage aufgezeichneten Sensorsignale sind in *Tabelle 2* aufgelistet. Zur Weiterverarbeitung muss die erhaltene Datei zuerst mittels einer geeigneten Software in eine csv-Datei überführt werden. Im konkreten Anwendungsfall kommt die Software „ibaAnalyzer“ zum Einsatz.
- Bei der Messung mittels Magnetjoch wird die Bedienung des Signalgenerators und –verstärkers manuell durchgeführt. Ebenso wird der Linearantrieb für das Auf– und Zufahren des Jochs per Hand gesteuert. Nach dem Einschalten des Verstärkers werden die Messsignale des Magnetjochs durch das Picoscope erfasst. Hierbei stehen maximal 4 Eingangskanäle für die Tastköpfe zur Verfügung. Um diese am Laptop anzeigen und speichern zu können ist die Software

„PicoLog“ vonnöten. Auch hier müssen die Daten, nach Beendigung der Messung, wiederum manuell in ein csv-Format konvertiert werden, um diese weiterverarbeiten zu können.

- Etwaig benötigte externe Thermoelemente werden ebenfalls separat über ein Chessel erfasst und als csv-Datei auf den Laptop übertragen.

Zur Auswertung einer Messung müssen die erhaltenen einzelnen Datensätze im csv-Format manuell zeitlich synchronisiert werden.

Tabelle 2: Messgrößen der Induktionsanlage

Messgröße	Regelgröße		Output	
	Wert	Einheit	Wert	Einheit
Induktor Kühlung Rücklauf (PT100)	–	–	xxxx	K
Induktor Kühlung Zulauf (PT100)	–	–	xxxx	K
Temperatur Abschreckmitteltank (PT100)	–	–	xxxx	K
Temperatur Brause (PT100)	–	–	xxxx	K
Temperatur Gas (PT100)	–	–	xxxx	K
Poti Leistung Umrichter	–	–	xxxx	Watt
Istwert Drehantrieb Wärmeregelung	0...100	%	2…10	V
Durchfluss Brause 1	1...100	L/min	4…20	mA
Druck Brause	0...10	bar	4…20	mA
Pyrometer Temperatur	Xxxx	C	4…20	mA
Druck Gas	0…10	bar	4…20	mA
Durchfluss Gasbrause 1	6…600	L/min	4…20	mA
Druck Gasbrause 1	0…10	bar	4…20	mA
Durchfluss Gasbrause 2	6…600	L/min	4…20	mA
Druck Gasbrause 2	0…10	bar	4…20	mA
Sollwert Drehantrieb Wärmeregelung	0…100	%	2…10	V
Sollwert Motor Rotation	20...400	U/min	0…10	V

4.2 Allgemeine Zielsetzung

Im Zuge des Projekts sollen die in *Kapitel 4.1* beschriebenen Teilkomponenten des Induktionsprüfstandes in einer zentralen Steuerungseinheit zusammengefasst werden, um den Messablauf und die dazugehörige Datenerfassung weitgehend zu automatisieren. Besonderes Augenmerk wird dabei auf den Erhalt zeitsynchroner Daten gelegt. Sprich die Daten der einzelnen Teilsysteme sollen in Echtzeit abgegriffen und zeitlich definiert abgelegt werden, um eine korrekte relative Zuordnung zueinander zu ermöglichen. In *Abb. 4-3* ist der angedachte Aufbau des Induktionsprüfstandes nach Abschluss der Erweiterung dargestellt.

Ziel ist ein System zu erhalten, dass wie folgt aufgebaut ist:

Als zentrale Steuereinheit soll eine SPS implementiert werden. Diese übernimmt die Steuerung und Koordination der einzelnen Komponenten. Die Kommunikation mit dem Benutzer findet über den Anlagenlaptop mittels einem GUI statt. Dieses ermöglicht die Eingabe von Versuchsparametern. Diese Parameter setzen sich aus dem Wärmebehandlungsrezept für die Induktionsanlage, der Vorgabe für Signalform und –amplitude des durch den Signalgenerator erzeugten Signals und Inputwerten zur Spezifizierung der Picoscope–Datenerfassung zusammen. Nach Eingabe der Parameter kann die Messung über das GUI gestartet werden. Das Starten der Anlage, das Auf- und Zufahren des Magnetjochs und die Bedienung des Signalgenerators und –verstärkers werden, so wie das Aufzeichnen von Messdaten durch das Picoscope und die Thermoelemente, automatisiert von der SPS geregelt. Weiters werden die Anlagedaten und die Messwerte der externen Thermoelemente während einer Messung in Echtzeit auf dem GUI angezeigt. Nach Abschluss einer Messung überträgt die SPS die gesammelten und zeitsynchronen Daten automatisch auf den Laptop bzw. in das lokale Netzwerk.

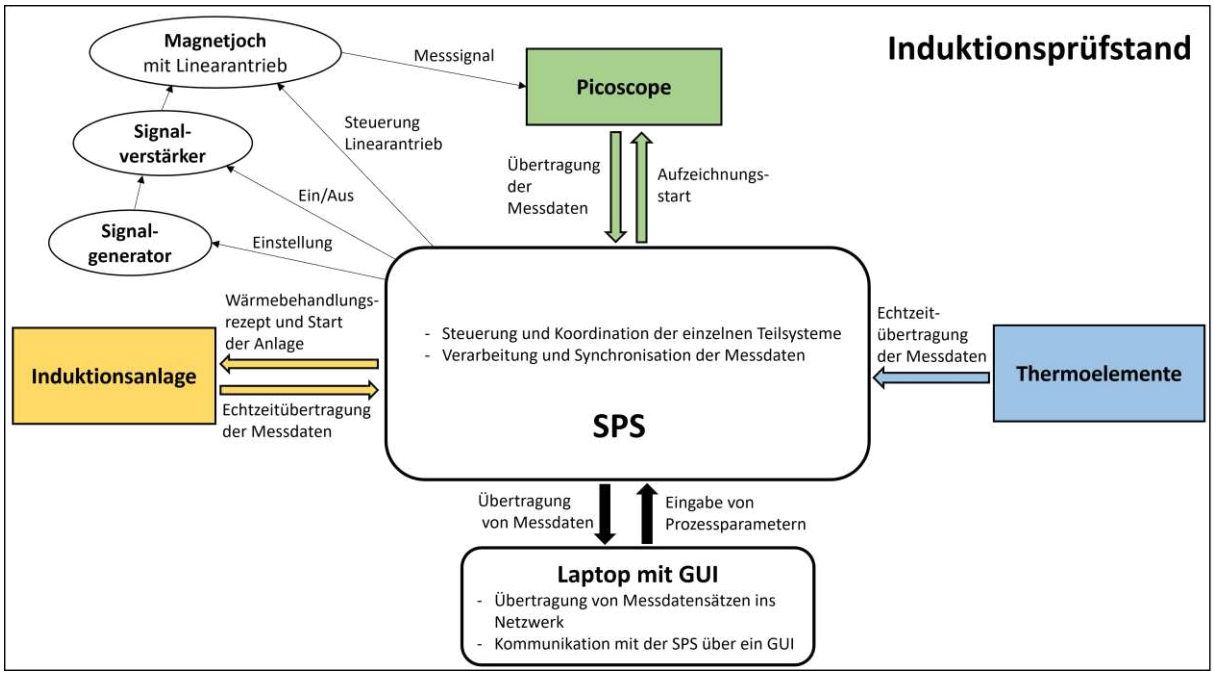


Abb. 4-3: Gewünschter Aufbau des Induktionsprüfstandes

4.3 Zielsetzung für diese Arbeit

Da die Umsetzung des vollständigen Projekts den Rahmen einer Masterarbeit sprengen würde, wurde es von vornherein in zumindest drei Masterarbeiten aufgeteilt. Der hier vorgestellte Teil markiert dabei den Beginn des Projekts und soll als Ausgangspunkt für die weiterführenden Arbeiten dienen.

Primäres Ziel dieser Arbeit ist der Aufbau der benötigten Hardware und die Umsetzung der automatisierten Datenerfassung. Dazu zählen einerseits die Aufzeichnung der Anlage– und Thermoelementdaten und andererseits das Triggern und Aufzeichnen der Picoscopemessung. Diese Daten sollen von der SPS zeitlich synchronisiert erfasst werden und nach Beendigung einer Messung auf dem RevPi abgelegt werden. Weiters wird ein vorläufiges GUI erstellt, das als Basis für die Einbindung weiterer Funktionalitäten dienen soll. Das Einbinden der Steuerung der Anlage, sowie die Ansteuerung der Signalkette und des Linearantriebs für das Joch werden in dieser Arbeit nicht umgesetzt.

5 Implementierung

Im nachfolgenden Kapitel wird die Umsetzung der in *Kapitel 4.3* behandelten Zielsetzungen beschrieben. Zu Beginn wird ein Gesamtüberblick über den hardwaretechnischen Aufbau des Systems gegeben, bevor dann in weiterer Folge die Signalwege und -verarbeitungen der einzelnen Teilsysteme im Detail beschrieben werden. Zum Schluss wird noch einmal der detaillierte Aufbau der Hard- und Software als Gesamtsystem dargelegt.

5.1 Gesamtübersicht

Zur Zusammenführung der Teilsysteme wird ein Schaltschrank errichtet, der, neben den benötigten elektrischen Komponenten, eine SPS als zentrales Element beherbergt. Der Schaltschrank bietet über Schnittstellen die Möglichkeit zur Anbindung aller Teilsysteme – direkt oder indirekt – an die SPS (siehe *Abb. 5-1*). Die Verbindung zwischen SPS und Picoscope wird mittels USB A bewerkstelligt. Die Messsignale der Anlage werden über Signalleitungen abgegriffen, die mit den Input Kanälen der SPS verbunden werden. Diese sind durch einen Stecker lösbar mit dem Schaltschrank verbunden. Der Anschluss der Thermoelemente erfolgt über eine Konsole mit Thermoelement-Steckplätzen und die Kommunikation der SPS mit dem Laptop findet über Ethernet statt. Näheres dazu wird in den nachfolgenden Kapiteln beschrieben.

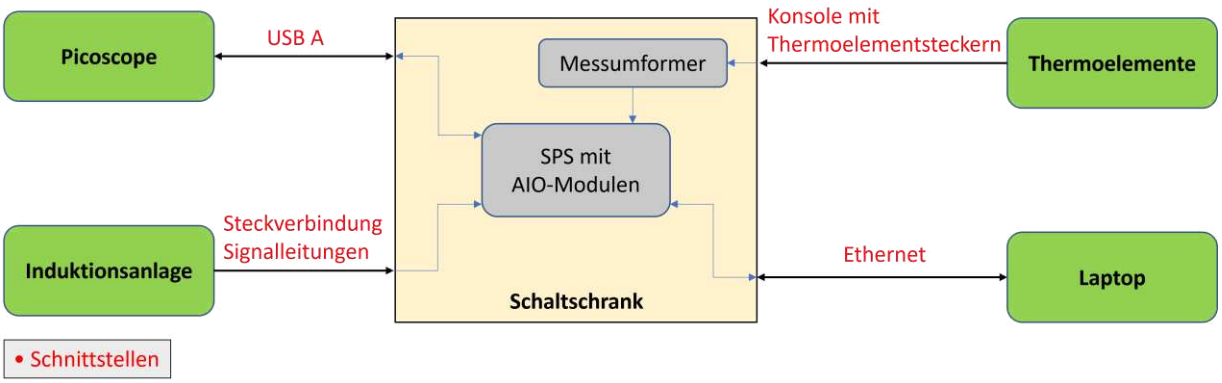


Abb. 5-1: Teilsysteme mit Schnittstellen

Der innere Aufbau des Schaltschrankes ist in *Abb. 5-2* dargestellt. Als SPS wird ein „RevPi Connect+ feat. Codesys“ mit fünf daran angeschlossenen analogen In–/Output–Modulen (AIO–Module) verwendet (siehe *Kapitel 5.1.1*). Die zehn Messumformer dienen der Umwandlung der Thermoelementsignale in normierte Messströme von 4–20 mA, die an die AIO–Module weitergegeben werden (siehe *Kapitel*

5.4.1). Der Schaltschrank wird über eine, durch einen Schalter trennbare, 230 V AC-Spannungsversorgung mit Energie versorgt. Da die benötige Versorgungsspannung sowohl für den RevPi und die AI/O-Module, als auch für die Messumformer 24V DC beträgt, wurde ein Netzteil eingebaut, das die Spannung wie benötigt transformiert (siehe *Datenblatt Anhang D3*). Dieses Netzteil wurde mit einem maximal zulässigen Strom von 10 A gewählt. In

Tabelle 3 ist der maximale Stromverbrauch der Schaltschrankkomponenten aufgelistet.

Tabelle 3: Stromverbrauch der Schaltschrankkomponenten

Komponente	Maximaler Stromverbrauch pro Komponente [mA]	Anzahl	Maximaler gesamter Stromverbrauch [mA]
RevPi	833	1	833
AI/O-Modul	500	5	2500
Messumformer [41]	<23	10	<230
		Summe	<3563

Durch den sich ergebenden zu erwartenden maximalen Stromverbrauch von weniger als 3563 mA lässt sich die Auswahl dieses Netzteiles rechtfertigen.

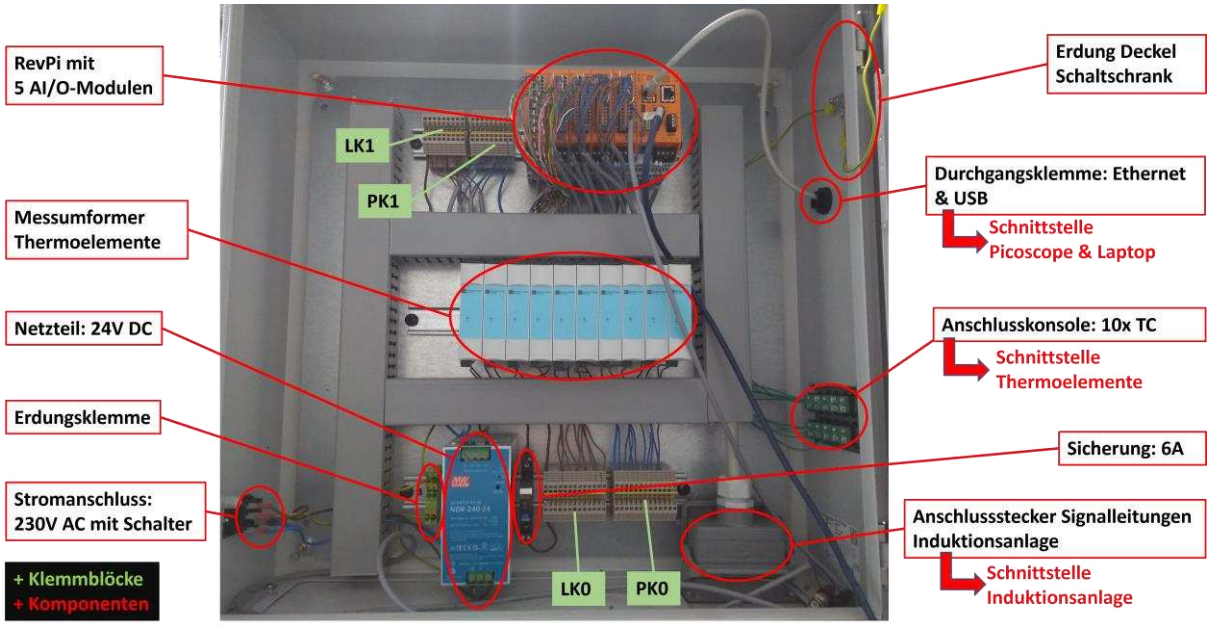


Abb. 5-2: Schaltschrank mit Komponenten

In *Abb. 5-2* ebenfalls ersichtlich sind die Anschlusskonsole zum Anschließen von bis zu 10 Thermoelementen und der Stecker mit Signalleitungen zum Abgreifen der Induktionsanlagensignale. Weiters wurde eine 6 A Sicherung in den Schaltschrank eingebaut. Sowohl der Schaltschrankdeckel als auch der -korpus werden über geeignete Klemmen elektrisch geerdet. Die Klemmblöcke PK0 und PK1 bezeichnen Steckplätze mit Nullpotential und LK0 bzw. LK1 die spannungsführenden Klemmen. Im weiteren Verlauf der Arbeit werden die Messumformer mit MU abgekürzt und zur Unterscheidung von

rechts nach links nummeriert. Der am weitesten rechts liegende Messumformer wird als MU1 bezeichnet, der links daneben als MU2, usw.

In *Abb. 5-3* ist die Klemmenbelegung für die Spannungsversorgung des RevPis und der AIO–Module dargestellt. Die weiteren Anschlüsse des Schaltschranks werden in den dazugehörigen Kapiteln behandelt.

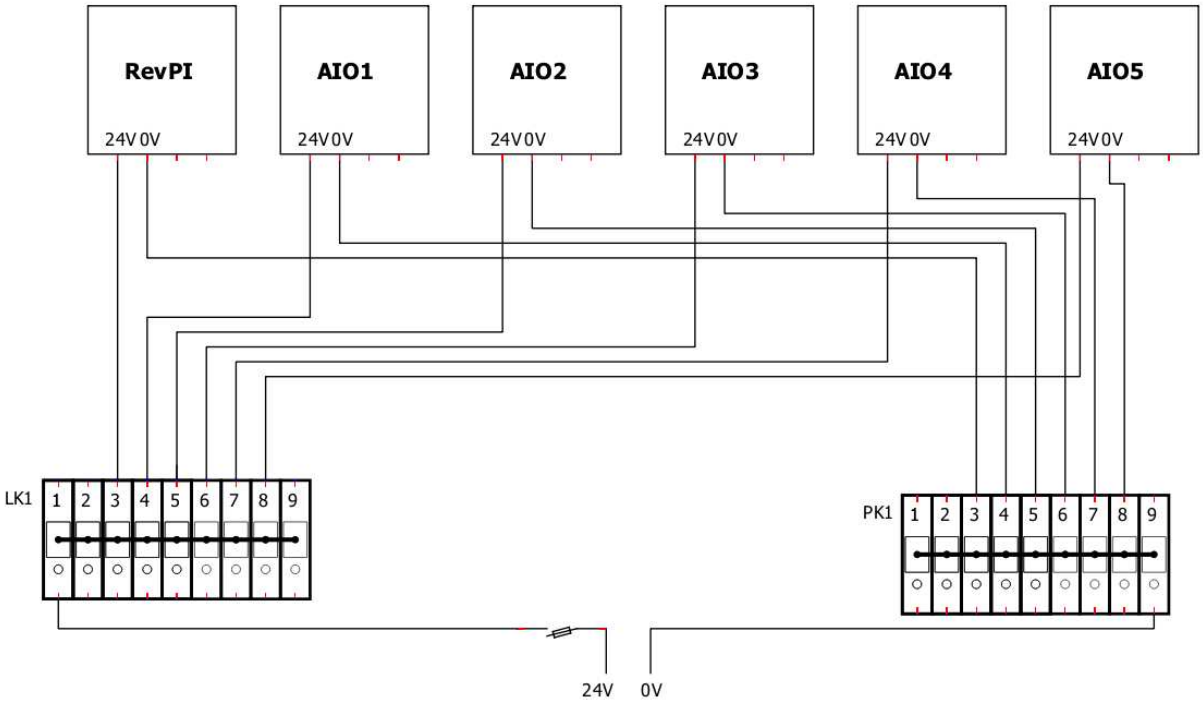


Abb. 5-3: Stromanschlussplan des RevPis und der AIO–Module

5.1.1 RevPi

Als SPS wird ein „RevPi Connect+ feat. Codesys" der Firma KUNBUS verwendet (siehe *Datenblatt Anhang D1).* Dieser basiert auf dem Rasperry Pi, besitzt jedoch im Gegensatz zu diesem gemäß EN 61131-2 bzw. IEC 61131-2 Industrietauglichkeit. Als Betriebssystem kommt das auf dem Linux-Kernel basierende „Buster" zum Einsatz, welches unter anderem mit einem Realtime-Patch ausgestattet ist [42]. Dieses bietet den Vorteil, dass man nicht an eine bestimmte Software oder einen Hersteller gebunden ist und somit in der Lage ist ein nicht proprietäres System aufzubauen.

An den RevPi angeschlossen sind fünf AIO–Module (siehe *Datenblatt Anhang D2*). Diese bieten jeweils vier analoge Input– und zwei analoge Outputkanäle (siehe *Abb. 5-4*). Diese sind konfigurierbar und können sowohl als Spannungs– als auch als Stromeingang bzw. –ausgang benutzt werden. Die dabei einstellbaren Wertebereiche sind in *Tabelle 4* aufgelistet.

Tabelle 4: Mögliche Konfiguration der IO's des AIO–Moduls [42]

Einstellbarer Inputbereich	Einstellbarer Outputbereich
-10 – 10 V	Off (Ausgang inaktiv)
0 – 10 V	-10 – 10 V
0 – 5 V	0 – 10 V
-5 – 5V	0 – 5 V
0 – 20 mA	-5.5 – 5.5 V
0 – 24 mA	-11 – 11 V
4 – 20 mA	0 – 20 mA
-25 – 25 mA	0 – 24 mA
	4 – 20 mA

Weiters bieten die AIO–Module noch die Möglichkeit zum Anschluss von bis zu zwei Pt100 bzw. Pt1000–Sensoren.

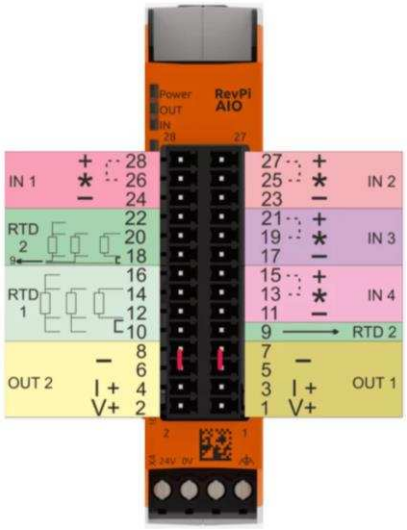


Abb. 5-4: Pinbelegung AIO–Modul [42]

In *Abb. 5-5* ist der RevPi mit den angeschlossenen AIO–Modulen und vorhandenen, relevanten Schnittstellen dargestellt. Hier wird gleich die für den weiteren Verlauf dieser Arbeit verwendete Nomenklatur der AIO–Steckplätze eingeführt. Das dem RevPi am nächsten liegende Modul wird mit AIO1 bezeichnet, die darauffolgenden mit AIO2 bis AIO5. Die Nummerierung der einzelnen Pins erfolgt wie in *Abb. 5-4*. Der mit „+" beschriftete Input1–Pin (IN 1) des AIO1 wird in weiterer Folge beispielsweise als AIO1.28 bezeichnet, der mit „-" beschriftete als AIO1.24 etc. Wird nicht der Pin direkt, sondern der Input allgemein angesprochen, so wird dieser zum Beispiel mit AIO1–IN1, für den Input 1 des AIO1, bezeichnet.

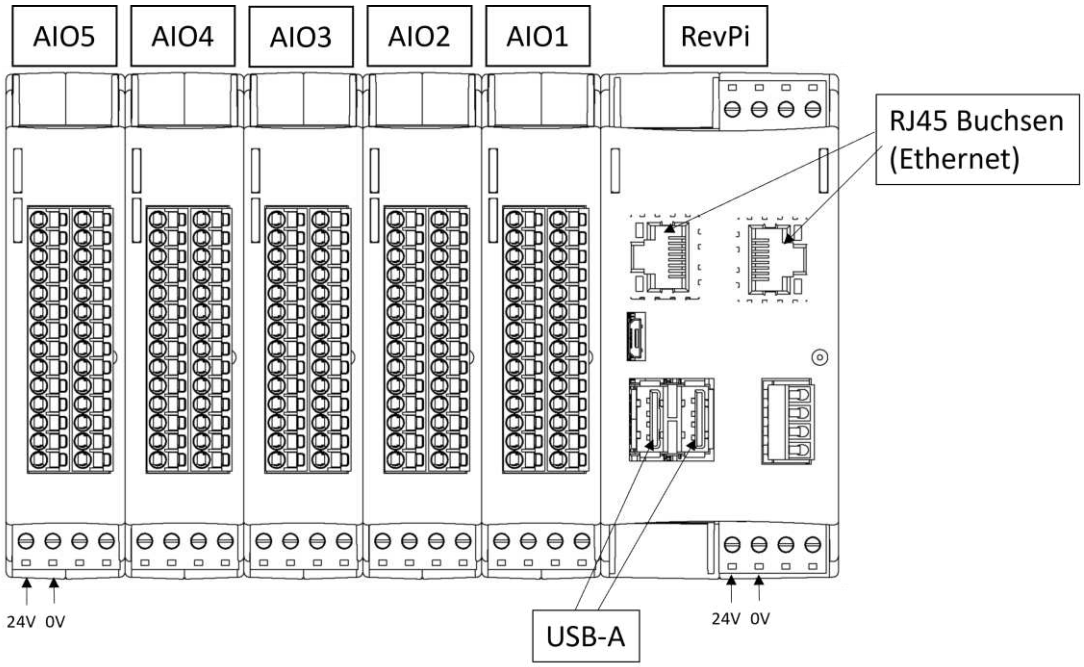


Abb. 5-5: RevPi mit Schnittstellen und Benennung der AIO–Module (in Anlehnung an [42])

Es ist noch zu erwähnen, dass bei elektrischen Strömen als Input–Signal die in *Abb. 5-4* mit „+“ und mit „∗“ beschrifteten Pins eines Inputs gebrückt werden müssen, bei der Messung von Spannungen hingegen nicht.

5.2 Software

Zur Programmierung kommt einerseits die herstellerunabhängige IEC-61131-3-Automatisierungssoftware „CODESYS“ und andererseits die objektorientierte Programmiersprache „Python“ zum Einsatz. Codesys wurde gewählt, da es sich dabei um eine, bis auf einige Spezialmodule, die hier nicht zwingend benötigt werden, frei zugängliche Software handelt. Weiters ist ein Framework für den RevPi bereits in der Software inkludiert. Auch bietet Codesys die Möglichkeit zur Verwendung von ST als Programmiersprache, welche bei der Programmierung von SPSen heutzutage Standard ist (siehe *Kapitel 3.3*). Um einige von Codesys nicht unterstütze Aufgaben, wie das Abspeichern von Daten im HDF-5 Format, zu übernehmen, wird Python als zweite Programmiersprache eingesetzt. Die Wahl von Python kam zustande, da es sich dabei um eine „Open Source“ Programmiersprache handelt, die viele bereits vorgefertigte Module und Funktionen anbietet. Aufgrund der weiten Verbreitung und der damit einhergehenden großen „Fangemeinde“ werden diese Module stetig weiterentwickelt und verbessert. Weiters ist Python – Stand Jänner 2023 – laut TIOBE–Index die meistgenutzte Programmiersprache weltweit.

Jan 2023	Jan 2022	Change	Programming Language	Ratings	Change
1	1		Python	16.36%	+2.78%
2	2		C	16.26%	+3.82%
3	4	▲	C++	12.91%	+4.62%
4	3	▼	Java	12.21%	+1.55%
5	5		C#	5.73%	+0.05%
6	6		Visual Basic	4.64%	-0.10%
7	7		JavaScript	2.87%	+0.78%
8	9	▲	SQL	2.50%	+0.70%
9	8	▼	Assembly language	1.60%	-0.25%
10	11	▲	PHP	1.39%	-0.00%
11	10	▼	Swift	1.20%	-0.21%
12	13	▲	Go	1.14%	+0.10%
13	12	▼	R	1.04%	-0.21%
14	15	▲	Classic Visual Basic	0.98%	+0.01%
15	16	▲	MATLAB	0.91%	-0.05%

Abb. 5-6: TIOBE Index Stand Jänner 2023 [43]

5.2.1 HDF5 Dateiformat

Das Ablegen der erhaltenen Messdaten soll im HDF5–Dateiformat erfolgen. Dabei handelt es sich um ein hierarchisches Datenformat, das es dem Benutzer ermöglicht Messdaten in einer ordnerähnlichen Struktur, hier als „Gruppen“ bezeichnet, abzulegen (siehe *Abb. 5-7*). Weiters lassen sich zugeordnet zu jeder dieser Gruppen, sowie auch zu den Datensätzen selbst, Metadaten hinterlegen, die es erlauben die gespeicherten Daten mit selbst gewählten beschreibenden Parametern – wie Erstelldatum, Messzeitpunkt, Einheiten, etc. – zu versehen [44].

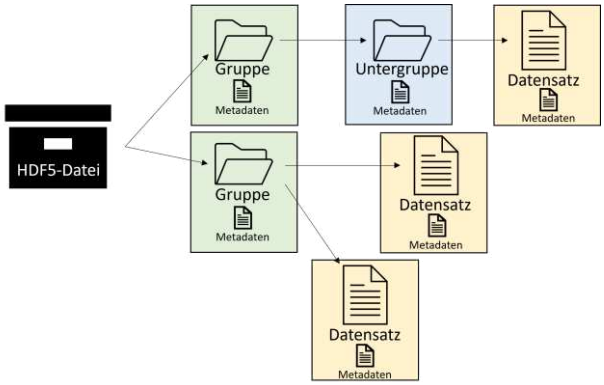


Abb. 5-7: Schematischer Aufbau eines HDF–Files (in Anlehnung an [44])

Auch handelt es sich bei dem HDF5–Format um ein binäres Dateiformat. Dadurch können große Mengen an Daten schnell und effizient geschrieben und gelesen werden [45].

5.3 Messkette Induktionsanlage

Das erste Teilsystem, das genauer betrachtet wird, ist die Induktionsanlage. Die ursprüngliche Idee war es, die Messdaten der Anlage über die Anlagen–SPS via Ethernet auszulesen und auf dem RevPi weiterzuverarbeiten. Allerdings war es aufgrund der proprietären Softwarelösung der Induktionsanlage nicht möglich in einem ausreichenden Maß in das System einzugreifen, um dies zu bewerkstelligen. Auch konnte der Hersteller der Anlage nicht erreicht werden, um etwaige Hilfestellungen zu erhalten. Deshalb wurde beschlossen die gewünschten Messsignale, als Strom– oder Spannungssignal, direkt an den zugehörigen Anschlussklemmen im Anlagen–Schaltschrank abzugreifen, diese über die AIO–Module zu digitalisieren und mittels dem RevPi in einen dazu proportionalen Messwert umzurechnen.

5.3.1 Hardware

In erster Instanz beschränkten sich die Bemühungen darauf diejenigen Signale der Anlage abzugreifen, deren Messung keine Abänderung der Funktionalität des bestehenden Systems nach sich zieht. Das sind die Signale all jener Sensoren, die einen standardisierten Spannungs– oder Stromoutput liefern, der mit einer geeigneten Schaltung über die AIO–Module umgeleitet und erfasst werden kann. In **Tabelle 5** sind diese Signale aufgelistet. Gemäß der in dieser Tabelle angegebenen Werte und der in *Anhang G* hergeleiteten Formel kann die Umrechnung der Sensorsignale in einen Messwert erfolgen.

Tabelle 5: Abgegriffene Signale der Anlage

Messgröße	Wertebereich		Sensoroutput		Bezeichnung
	Wert	Einheit	Wert	Einheit	
Istwert Drehantrieb Wärmeregelung	0...100	%	2…10	V	n_Waermeregelung
Durchfluss Brause 1	1...100	L/min	4…20	mA	Q_Brause
Druck Brause	0...10	bar	4…20	mA	p_Brause
Druck Gas	0…10	bar	4…20	mA	p_Gas
Durchfluss Gasbrause 1	6…600	L/min	4…20	mA	Q_Gasbrause1
Druck Gasbrause 1	0…10	bar	4…20	mA	p_Gasbrause1
Durchfluss Gasbrause 2	6…600	L/min	4…20	mA	Q_Gasbrause2
Druck Gasbrause 2	0…10	bar	4…20	mA	p_Gasbrause2

In *Abb. 5-8 (a)* ist der schematische Signalweg der verschiedenen Messgrößen zu sehen. Die Schnittstelle zur Verbindung der Signalleitungen des Anlagenschaltschranks und des Schaltschranks des RevPi‘s stellt ein 40-poliger Stecker „HAN D 40“ des Herstellers „Harting“ dar (siehe *Abb. 5-2*). Die grundlegende Schaltung zum Abgreifen eines Spannungssignals ist in *Abb. 5-8 (c)* abgebildet. Hierbei wird

der Input eines AIO–Moduls parallel zu der anlageinternen Messeinrichtung geschalten, um die Messspannung U_{Sensor} zu erfassen. Dabei ist keine weitere Adaptierung der vorliegenden Verkabelung, über das Anschließen der notwendigen Signalleitungen hinaus, vonnöten. Das Abgreifen der Stromsignale dagegen muss in Serie erfolgen. Dabei wird das Signal I_{Sensor} der Anlage über das AIO–Modul umgeleitet und wieder zurückgeführt (siehe *Abb. 5-8 (b)*). Das macht es notwendig, dass eine im Anlagen–Schaltschrank ursprünglich vorhandene Verbindung – in der Abbildung rot dargestellt – zwischen der negativen Klemme des Signals und Masse gelöst werden muss.

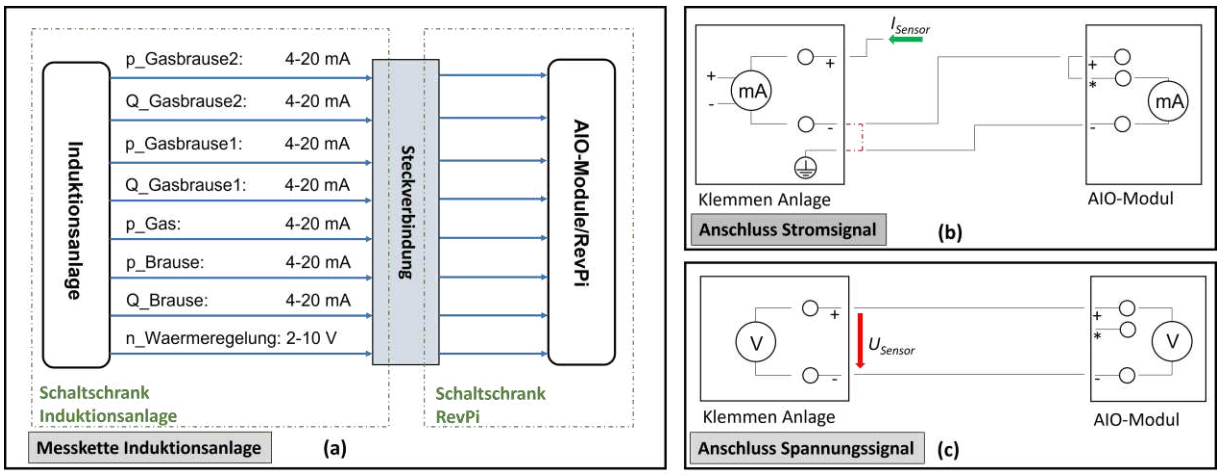


Abb. 5-8: Abgreifen der Signale der Induktionsanlage

In *Tabelle 6* sind die verwendeten Anschlusspins, sowohl der Anlage als auch des RevPis, für die Signalleitungen aufgelistet. Die „Hinleitung“ bezeichnet dabei die Signalleitung, die zwischen der Induktionsanlage und dem positiven Input Pin des AIO–Moduls verläuft und die „Rückleitung“ die, die mit dem negativen Input Pin des AIO–Moduls verbunden ist. Die Benennung der Anlagepins erfolgt wie in den originalen Hardwareplänen der Induktionsanlage (siehe *Anhang E*). Die Spalte „gelöste Verbindung“ gibt an, welche ursprünglich im Anlagenschaltschrank vorhandene Verbindung entfernt werden musste, um ein Abgreifen der Stromsignale in Serie möglich zu machen.

Tabelle 6: Anschlüsse Signalleitungen zwischen Induktionsanlage und AIO–Modulen

Variable	Output		Hinleitung / Rückleitung		Gelöste Verbindung	
	Wert	Einheit	von	zu	Von	zu
n_Waermeregelung	2…10	V	-1X4:4 -1A9:1C	AIO5.15 AIO5.11	-	-
Q_Brause	4…20	mA	-1A10:5 Klemme*	AIO4.28 AIO4.24	-1A10:5	Klemme*
p_Brause	4…20	mA	-1A10:6 -1A10:3C	AIO4.27 AIO4.23	-1A10:6	-1A10:3C
p_Gas	4…20	mA	-1A10:8 -1A10:3C	AIO4.21 AIO4.17	-1A10:8	-1A10:3C
Q_Gasbrause1	4…20	mA	-1A11:5 -1A11:4C	AIO4.15 AIO4.11	-1A11:5	-1A11:4C
p_Gasbrause1	4…20	mA	-1A11:6 -1A11:2C	AIO5.28 AIO5.24	-1A11:6	-1A11:2C
Q_Gasbrause2	4…20	mA	-1A11:7 -1A11:3C	AIO5.27 AIO5.23	-1A11:7	-1A11:3C
p_Gasbrause2	4…20	mA	-1A11:8 -1A11:4C	AIO5.21 AIO5.17	-1A11:8	-1A11:4C

*Wagoklemme: Verbindung von Signalleitung von AIO4.24 kommend zu Kabel das an Pin -1A10.5 angeschlossen war

Abb. 5-9 stellt die erhaltene Pin-Belegung der verwendeten AIO–Module grafisch dar. Hierbei werden weiters die verwendeten Leiterfarben und auch die verwendeten Steckplätze des Zwischensteckers (S:XX) angegeben.

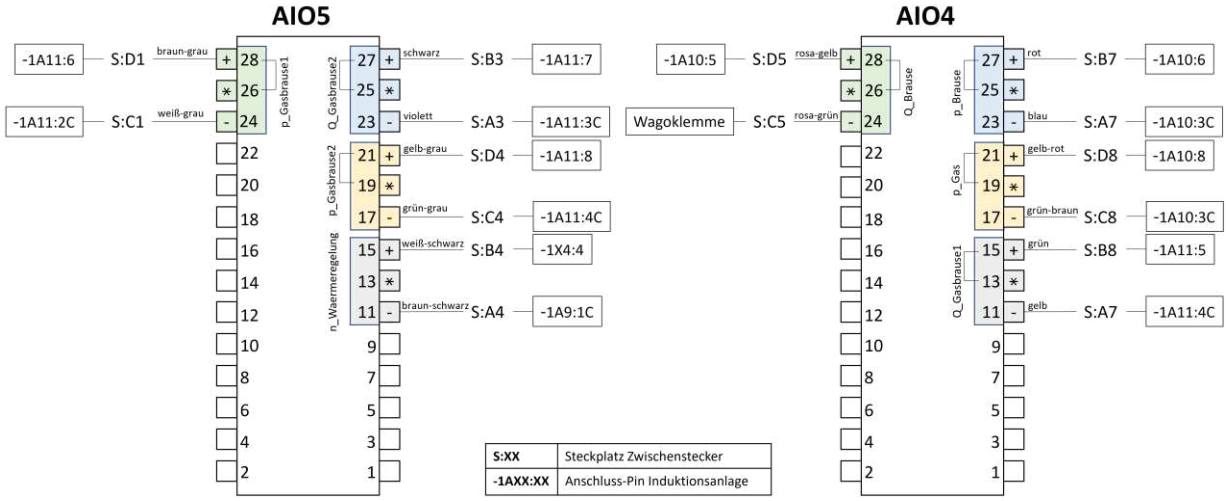
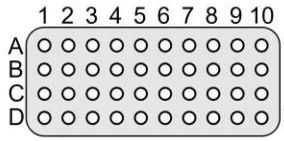


Abb. 5-9: Pin-Belegung der AIO–Module „AIO4“ und „AIO5“

In *Abb. 5-10* ist die gesamte Belegung des Zwischensteckers dargestellt. Die bereits angeschlossenen, aber noch nicht mit der Anlage bzw. dem RevPi verknüpften Signalleitungen sind für das Abgreifen etwaiger weiterer Messsignale in Zukunft angedacht.



„-“	Zu negativem Pin von AIO-Modul
„+“	Zu positivem Pin von AIO-Modul

	1	2	3	4	5	6	7	8	9	10
A	Weiß/Gelb	Braun/Grün	Q_Gasbrause2 - Violett	n_Waermeregelung - Braun/Schwarz	Gelb/Schwarz	Rosa	p_Brause - Blau	Q_Gasbrause1 - Gelb	Weiß	FREI
B	Gelb/Braun	Grün/Weiß	Q_Gasbrause2 + Schwarz	n_Waermeregelung + Weiß/Schwarz	Grün/Schwarz	Grau	p_Brause + Rot	Q_Gasbrause1 + Grün	Braun	FREI
C	p_Gasbrause1 - Weiß/Grau	Weiß/Pink	Rot/Blau	p_Gasbrause2 - Grün/Grau	Q_Brause - Rosa/Grün	Blau/Braun	Rot/Braun	p_Gas - Grün/Rotbraun	Blau/Gelb	FREI
D	p_Gasbrause1 + Braun/Grau	Braun/Pink	Grau/Rosa	p_Gasbrause2 + Gelb/Grau	Q_Brause + Rosa/Gelb	Blau/Weiß	Rot/Weiß	p_Gas + Gelb/Rot	Blau/Grün	FREI

Abb. 5-10: Steckerbelegung mit Kabelfarbe und jeweiligen Messsignalen

5.3.2 Programmierung

Dem Programm kommt einerseits die Aufgabe des Einlesens und Umrechnens der Messdaten und andererseits die Speicherung ebendieser zu. Der Programmteil zum Einlesen der Anlagemessdaten über die AIO–Module wird über Codesys mit ST verfasst. Das Verarbeiten und Ablegen der erhaltenen Daten erfolgt via Pythonskript, da das Arbeiten mit dem HDF5–Dateiformat von den frei zugänglichen Codesys–Modulen nicht unterstützt wird. Ein Problem, das sich hierbei ergibt, ist, dass von Codesys aus zwar prinzipiell Pythonskripte gestartet werden können, der zyklische Ablauf des SPS–Programms allerdings so lange zum Stillstand kommt, bis das gestartete Pythonskript fertig abgearbeitet wurde. Das lässt sich nicht mit der gewünschten Echtzeitfähigkeit und dem notwendigen (quasi)parallelen Abarbeiten von mehreren Softwareaufgaben, zur Erfassung der einzelnen Teilsysteme, vereinbaren. Um dieses Problem zu umgehen, wurde folgendes Prinzip verfolgt (siehe *Abb. 5-11):*

- Der Programmcode der SPS – in ST geschrieben – läuft zyklisch ab. Dieser ist für die Datengenerierung verantwortlich.
- Ein Pythonskript, als „finite state machine“(FSM) aufgebaut, läuft unabhängig vom ST–Programm parallel dazu ebenfalls in Dauerschleife ab. Dieses Skript dient der Weiterverarbeitung der Messdaten.
- Es wird ein Speicherplatz im Arbeitsspeicher eingerichtet, auf den beide Programme Zugriff haben. So ein Speicherplatz wird als „shared memory“(SM) bezeichnet.
- Solange keine Datenaufzeichnung erfolgt, befindet sich das Pythonskript in einem Wartezustand.

- Wird eine Datenaufzeichnung gestartet, wird durch das ST–Programm der „shared memory“ mit den Mess- und zugeordneten Metadaten beschrieben und ein Trigger für das Pythonskript gesetzt, um anzuzeigen, dass Daten bereit zum Abrufen sind.
- Wird ein Trigger erkannt, werden in weiterer Folge die Daten von dem Python–Programm aus dem SM ausgelesen.
- Nach Ende der Datenaufzeichnung legt das Python–Programm die gesammelten Messdaten als HDF5–Datei auf dem RevPi ab und kehrt in den Wartezustand zurück.

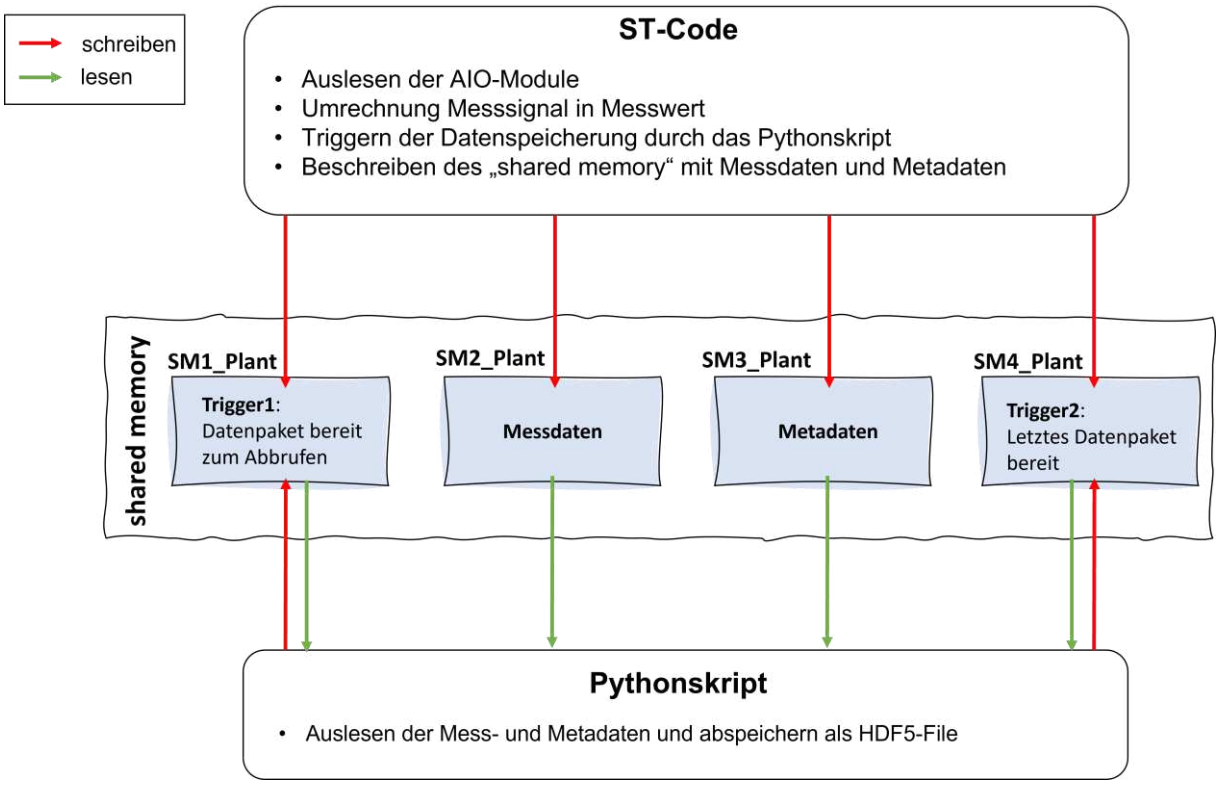


Abb. 5-11: Programmstruktur für die Datenerfassung der Induktionsanlage

Bei der Umsetzung dieses Aufbaus sind einige Dinge zu beachten, weswegen im Nachfolgenden noch einmal auf den exakten, detaillierten Ablauf der Datenaufzeichnung eingegangen wird. Hierbei wird sich wiederum auf *Abb. 5-11* bezogen:

Es werden in Summe vier shared memories eingerichtet:

- **SM1_Plant**: Dieser Speicherplatz beinhaltet einen Integer, der die Werte 0 oder 1 annehmen kann. „0“ bedeutet dabei „Trigger inaktiv“ und „1“ bedeutet, es sind Daten zum Abrufen vorhanden.
- **SM2_Plant**: Diese Speicheradresse bietet Platz für ein STRUCT, welches die Messwerte der aktuellen Messung beinhaltet. Der Aufbau dieses STRUCTs ist im *Anhang C15* einzusehen.

- **SM3_Plant**: Diese Speicheradresse bietet Platz für ein STRUCT, welches Metadaten der durchgeführten Messung beinhaltet. Diese werden in weiterer Folge zur Verarbeitung der Messdaten benötigt. Der Aufbau dieses STRUCTs kann *Anhang C15* entnommen werden.
- **SM4_Plant**: Wie SM1_Plant beinhaltet dieser Speicherplatz einen Integer mit den möglichen Werten 0 oder 1. Hierbei bedeutet „1“, dass die aktuelle Datenaufzeichnung beendet wurde.

Wird eine Datenaufzeichnung gestartet, werden die erhaltenen Messwerte in Codesys in einem STRUCT zwischengespeichert. Dieses STRUCT bietet Platz für die Messdaten von 1000 Zyklen, wobei in jedem Zyklus alle zuvor besprochenen Messgrößen einmal ausgelesen werden. Sind die 1000 Zyklen erreicht, wird das vollständig beschriebene STRUCT in den SM2_Plant übertragen. Weiters wird der Integer im SM1_Plant (Trigger1) auf „1“ gesetzt. Das Pythonskript registriert den Trigger, liest daraufhin den SM2_Plant aus und speichert das erhaltene STRUCT zwischen. Auch wird der Trigger1 durch das Pyhtonskript wieder auf „0“ gesetzt. Dieser Vorgang wird so lange wiederholt, bis die Datenaufzeichnung beendet wird. Geschieht dies, wird der SM2_Plant mit dem letzten Datenblock und der SM3_Plant mit den Metadaten beschrieben. Auch wird der SM4_Plant (Trigger2) auf „1“ gesetzt. Dieser Trigger, der den Abschluss der Messung signalisiert, wird vom Pythonskript registriert. Daraufhin werden die Metadaten und die letzten Messdaten ausgelesen, der Trigger2 auf „0“ zurückgesetzt und alle gesammelten Daten weiterverarbeitet.
Der Grund, warum hier zwei verschiedene Speicherplätze (SM1_Plant und SM4_Plant) zum Triggern zum Einsatz kommen, – und nicht etwa ein SM mit zwei verschiedenen Variablenwerten – ist der, dass es bei ungünstigem Timing des Messendes ansonsten sein könnte, dass sowohl Codesys als auch Python zugleich in denselben SM schreiben wollen, was einen Absturz beider Programme zur Folge hätte.
Weiters unterscheidet sich der letzte Datenblock von den vorhergehenden insofern, dass im Allgemeinen ein nicht vollständig mit sinnvollen Daten beschriebenes Messdaten–STRUCT an den SM2_Plant übergeben wird, da die Messung innerhalb der 1000 Messzyklen unterbrochen wurde. Daher muss durch Zurückgreifen auf die Metadaten eruiert werden, bis zu welchem Eintrag das erhaltene STRUCT berücksichtigt werden soll.
In *Abb. 5-12* ist der beschriebene Ablauf der Messdatenerfassung der Induktionsanlage noch einmal grafisch dargestellt.

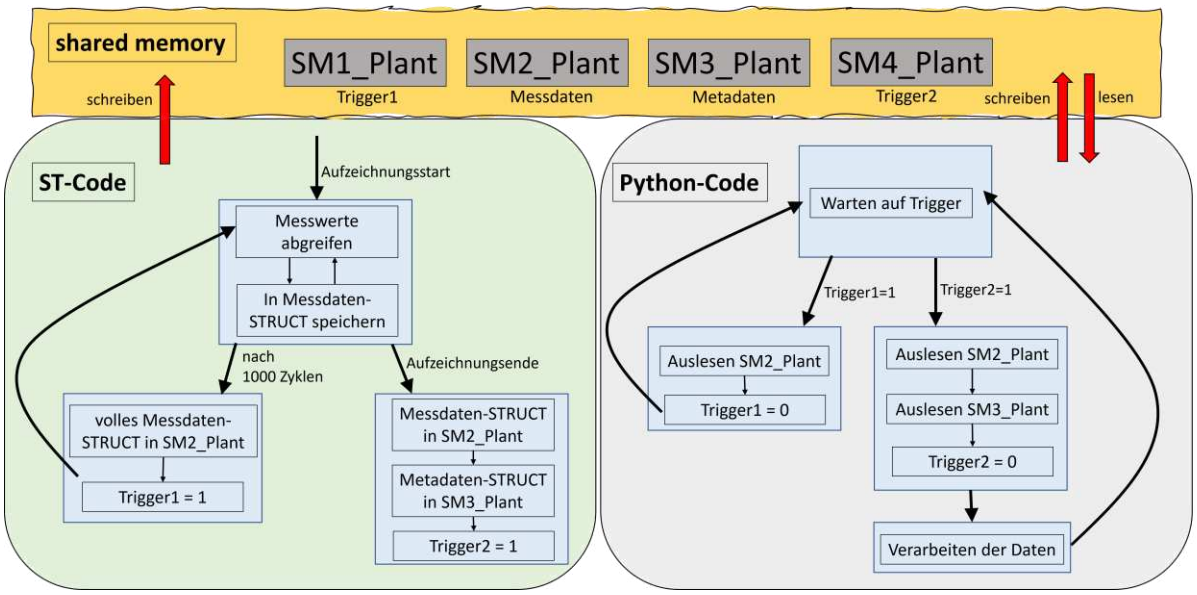


Abb. 5-12: Ablauf der Datenerfassung der Induktionsanlage

ST–Code (siehe *Anhang C9 und C11*):

Der TASK „PlantLoggerTask" beinhaltet die relevanten Programme zur Datenerfassung der Induktionsanlage (siehe *Abb. 5-13*).

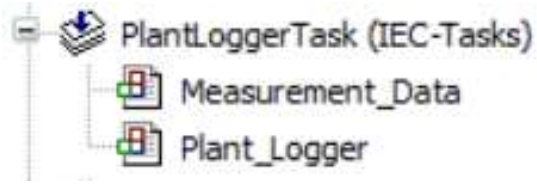


Abb. 5-13: Codesys Programmaufbau (Anlagedaten)

Das PROGRAM „Measurement_Data" dient der Umrechnung der durch die AIO–Module erhaltenen, digitalisierten Strom– und Spannungsmesssignale in die proportionalen Messwerte. Das PROGRAM „Plant_Logger" dient einerseits zur Einrichtung der SMs und andererseits zum Beschreiben des Speichers mit den Datenblöcken und zum Setzen der Trigger für das Pythonskript.

Python–Code (siehe *Anhang B1*):

Das Pythonskript „memMap_Data.py" dient der Verarbeitung der Messdaten der Induktionsanlage. Dieses ist, wie schon erwähnt, als FSM aufgebaut, die in *Abb. 5-14* dargestellt ist. Die vom Code einnehmbaren Zustände sind dabei folgende:

- **State = 0**: Beim Starten des Skripts wird die Verbindung zu den SMs, die durch Codesys initialisiert wurden, aufgebaut.
- **State = 1**: Dies ist der „Standby"–Modus. Hier werden in Dauerschleife Trigger1 und Trigger2 abgefragt.

- **State = 2**: Hier erfolgt das Auslesen der Messdaten aus dem SM. Die der Reihe nach erhaltenen STRUCTs werden in einer Liste zwischengespeichert. Trigger1 wird zurückgesetzt.
- **State = 3**: Wurde das Ende einer Datenaufzeichnung erkannt, erfolgt das letzte Auslesen eines Datenblocks und das Auslesen der Metadaten aus den SMs. Trigger2 wird zurückgesetzt.
- **State = 4**: Um das Programm sauber zu beenden, werden alle belegten Speicherplätze freigegeben und alle Variablen gelöscht.
- **State = 10**: Hier erfolgt die Weiterverarbeitung der Messdaten. Dabei wird die Liste der erhaltenen Messdaten–STRUCTs in ein numpy.ndarray umgewandelt. Dafür werden die zuvor ausgelesenen Metadaten benötigt.

Was bisher außer Acht gelassen wurde, aber in *Abb. 5-14* zu erkennen ist, ist die Rolle des Pythonskripts „startProcess.py“. Dieses dient der Zusammenfassung der Messungen der einzelnen Teilsysteme und wird in *Kapitel 5.6.2* im Detail behandelt. Hier sei nur gesagt, dass „startProcess.py“ für das Ausführen von „memMap_Data.py“ verantwortlich ist und am Ende einer Messung die erhaltenen Messdaten in Form eines numpy.ndarray von „memMap_Data.py“ erhält. Die Speicherung der Daten als HDF5–File erfolgt also eigentlich nicht im Skript „memMap_Data.py“ selbst, sondern erst in „startProcess.py“.

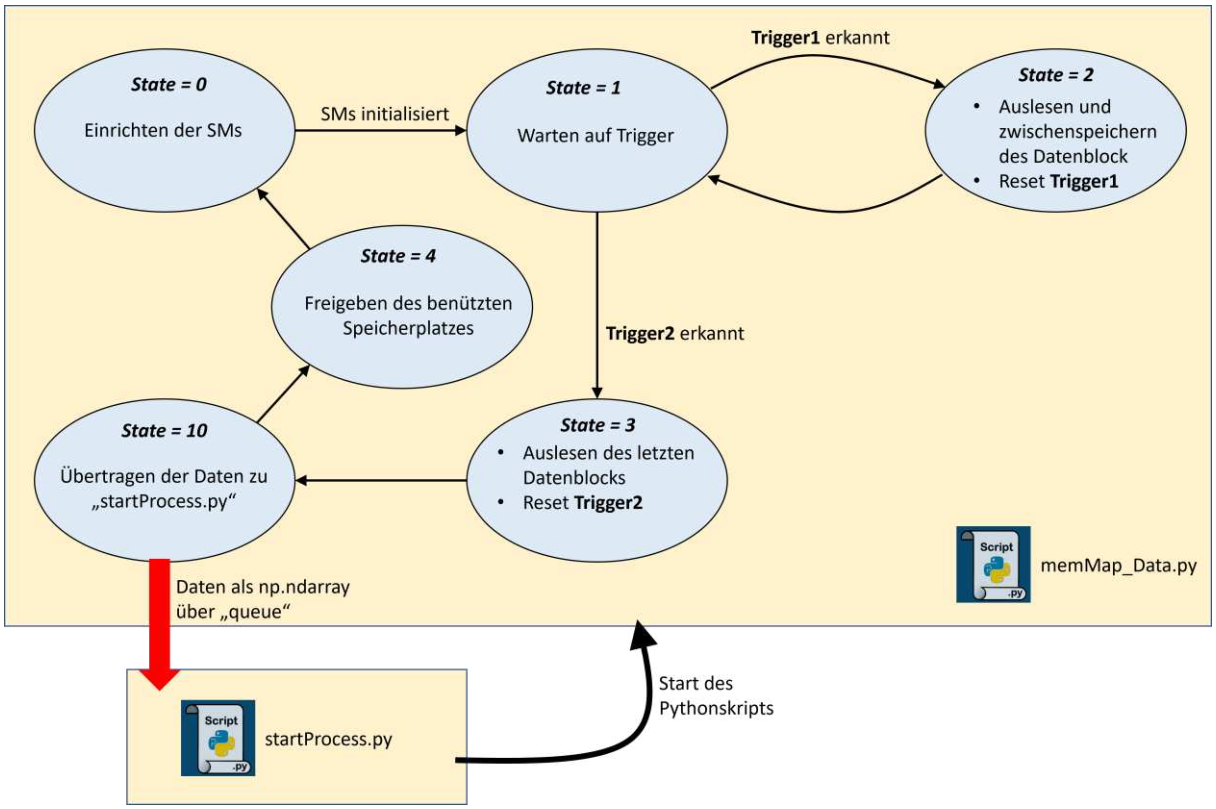


Abb. 5-14: Programmaufbau memMap_Data.py

5.4 Messkette Thermoelemente

Das nächste zu erläuternde Teilsystem stellt die Temperaturmessung mittels der Thermoelemente dar. Es soll die Möglichkeit geschaffen werden, im Rahmen einer Messung bis zu 10 zusätzliche externe Thermoelemente vom Typ K anschließen zu können. Im Nachfolgenden wird sowohl der dafür verwendete hardware– als auch der softwaretechnische Aufbau beschrieben.

5.4.1 Hardware

Der Aufbau der Messkette zur Erfassung der Temperatur durch die Thermoelemente ist in *Abb. 5-15* dargestellt. Die Thermoelemente werden über eine Konsole, die sich seitlich am Schaltschrank des RevPis befindet, angesteckt. Diese bietet Platz für den Anschluss von bis zu 10 Thermoelementen. Das erhaltene Messsignal, in Form der Thermospannung, wird innerhalb des Schaltschranks über geeignete Messumformer in ein proportionales Ausgangsstromsignal zwischen 4 und 20 mA umgewandelt. Dieses wiederum wird dann durch die AIO–Module des RevPis digitalisiert und in weiterer Folge über die Kennlinie des Messumformers in einen Temperaturmesswert umgerechnet.

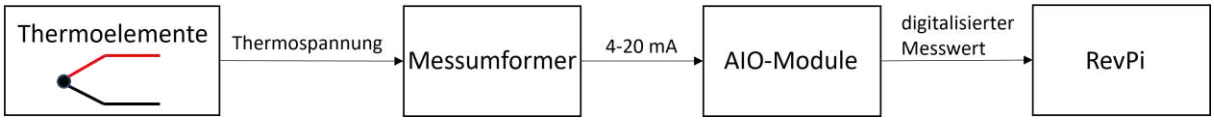


Abb. 5-15: Messkette Thermoelemente

Die Nummerierung der Konsolensteckplätze des Schaltschranks erfolgt wie in *Abb. 5-16* ersichtlich.

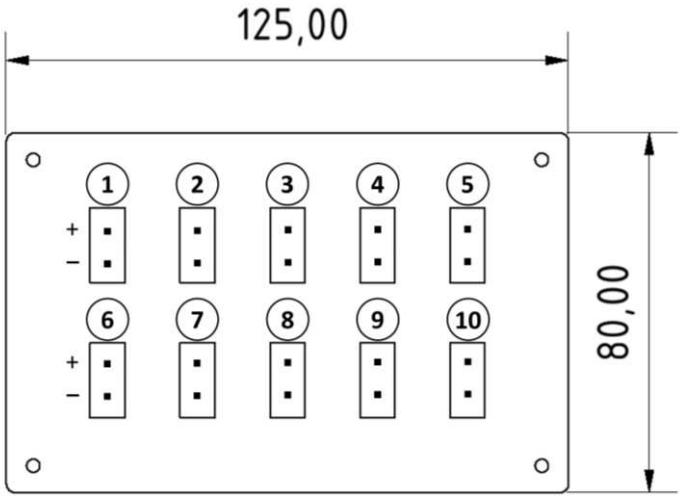


Abb. 5-16: Konsole Thermoelementstecker

Zugeordnet zu den Steckplätzen und den Thermoelementen erfolgt auch die Bezeichnung der Variablen der erfassten Temperatur–Messgrößen. Diese sind in *Tabelle 7* einsehbar.

Tabelle 7: Erfasste Thermoelement–Messwerte

Messgröße	Sensor	zulässiger Mess-bereich	Variablenbe-zeichnung	Steckplatz Konsole	Signaloutput Mess-umformer
Temperatur	TC–TypK	-180…1350°C	TC1	1	4…20 mA
Temperatur	TC–TypK	-180…1350°C	TC2	2	4…20 mA
Temperatur	TC–TypK	-180…1350°C	TC3	3	4…20 mA
Temperatur	TC–TypK	-180…1350°C	TC4	4	4…20 mA
Temperatur	TC–TypK	-180…1350°C	TC5	5	4…20 mA
Temperatur	TC–TypK	-180…1350°C	TC6	6	4…20 mA
Temperatur	TC–TypK	-180…1350°C	TC7	7	4…20 mA
Temperatur	TC–TypK	-180…1350°C	TC8	8	4…20 mA
Temperatur	TC–TypK	-180…1350°C	TC9	9	4…20 mA
Temperatur	TC–TypK	-180…1350°C	TC10	10	4…20 mA

Als Messumformer kommt ein „iTemp TMT128– AKAIA“ von dem Hersteller „Endress+Hauser“ zum Einsatz (siehe *Datenblatt Anhang D4*). Dieser benötigt eine Versorgungsspannung von 24 V und bietet ein lineares Messverhalten im Temperaturbereich von 0°C bis 1400°C (siehe *Tabelle 8*). Da der zulässige Messbereich der Thermoelemente selbst nur bis etwa 1350°C reicht und Messungen bei Temperaturen unter 0°C nicht durchgeführt werden, ist der Messbereich des Messumformers für diese Anwendung ausreichend. Aufgrund der in der Tabelle angegebenen Werte kann der erhaltene Stromoutput einfach in die dazu proportionale Temperatur umgerechnet werden.

Tabelle 8: Linearer Messbereich des Messumformers

Stromoutput	proportionale Temperatur
4 mA	0°C
20 mA	1400°C

Zum Abgreifen des sich ergebenden Stromoutputs des Messumformers wird der Signalinput des AIO–Moduls in Serie in die Versorgungsleitung des Messumformers eingebunden (siehe *Abb. 5-17*). Weiters ist zu beachten, dass die Konsolenstecker mit den Messumformern innerhalb des Schaltschrankes mit einem geeigneten Thermoelementkabel des Typs K verbunden werden müssen, da ansonsten – beispielsweise bei der Verwendung standardmäßiger Kupferleitungen – die sich ergebende Thermospannung verfälscht würde.

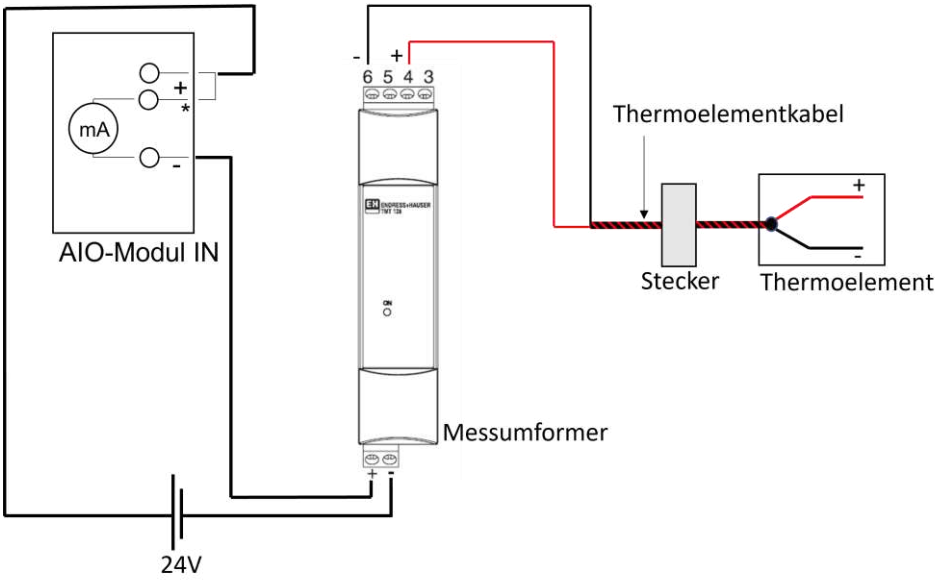


Abb. 5-17: Abgreifen des Stromoutputs der Messumformer

Der Anschlussplan der Messumformer im Schaltschrank ist in *Abb. 5-18* einzusehen. Die Bezeichnung der Klemmblöcke PK0 und LK0 sowie der Messumformer MU1 bis MU10 und der AIO–Module AIO1 bis AIO3 erfolgt entsprechend *Kapitel 5.1*.

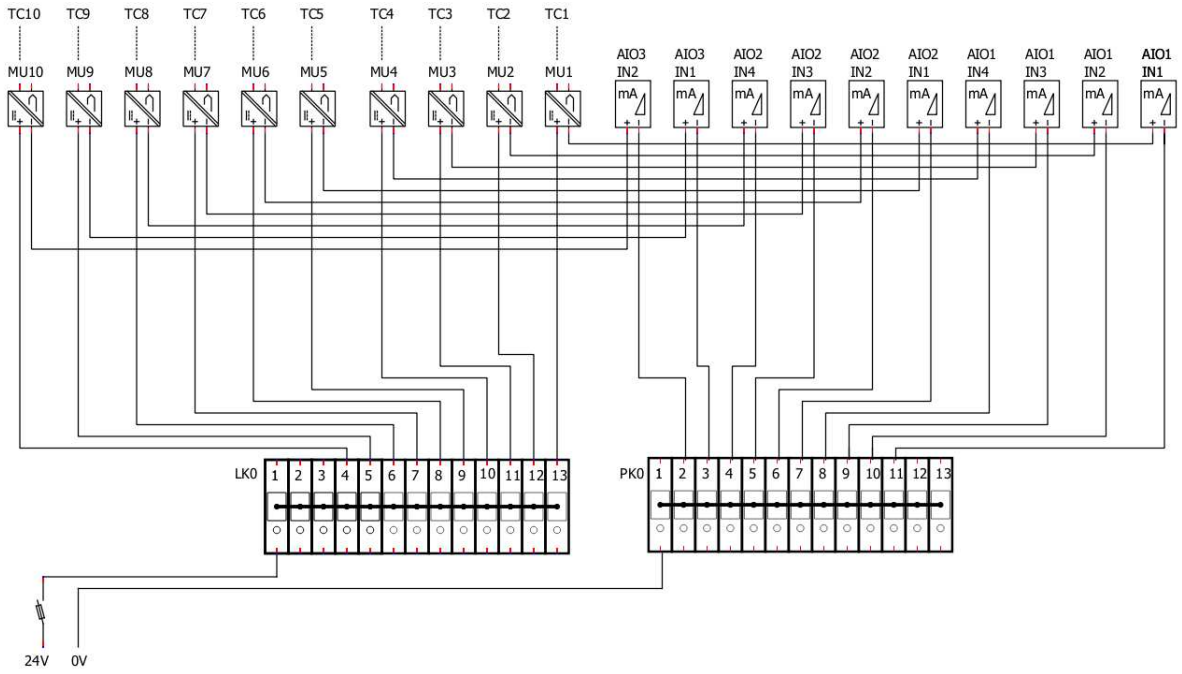


Abb. 5-18: Anschlussplan der Messumformer

5.4.2 Programmierung

Die Verarbeitung der Messwerte der Thermoelemente erfolgt Hand in Hand mit denen der Induktionsanlage. So übernehmen die in *Kapitel 5.3.2* beschriebenen Codesys–Programme „Measurement_Data“

und „Plant_Logger“ auch hier wieder das Umrechnen der Stromsignale in die Messwerte sowie die Weiterverarbeitung der Daten. Die Temperaturmesswerte werden dabei in das gleiche Messdaten–STRUCT wie die Anlagemessdaten geschrieben. Das Messdaten–STRUCT beinhaltet also nicht nur die Daten der Induktionsanlage, sondern auch die Temperaturmesswerte der 10 Thermoelemente. Programmtechnisch besteht also keine Trennung zwischen der Messkette der Thermoelemente und der der Induktionsanlage. Der weitere Programmablauf wurde im besagten Kapitel bereits ausreichend behandelt.

5.5 Messkette Picoscope

Das letzte Teilsystem, das es zu betrachten gilt, ist die Erfassung der Magnetjochsignale mittels des Picoscopes. Bei einer Magnetjochmessung wird die Systemantwort einer ferromagnetischen Rundstabprobe auf eine magnetische Erregung untersucht. Das Magnetjoch besteht dabei aus einer Primär- und zwei Sekundärspulen, wobei die Detektion der Systemantwort über die Sekundärspulen erfolgt. Die zu messenden Signale der Sekundärspulen werden mittels der Tastköpfe des Picoscopes erfasst. Um diese Messdaten aufzuzeichnen, musste bis dato das Starten der Aufzeichnung durch das Picoscope über die Software „PicoScope“ am Laptop durchgeführt werden. Nun soll diese Datenerfassung dahingehend automatisiert werden, dass das Starten einer Picoscopemessung über das GUI der SPS erfolgen kann und der weitere Verlauf der Messung, sowie auch die nachfolgende Datenverarbeitung, durch die SPS gesteuert wird. Dabei gibt es zwei verschiedene Messszenarien zu betrachten.

<u>Aktiver Modus:</u>

Im aktiven Modus erfolgt die Anregung der Probe durch die Primärspule des Magnetjochs. Der Ablauf einer Messung soll dabei wie folgt aussehen:

- Die Induktionsanlage führt entweder ein Wärmebehandlungsrezept aus, oder steht still, wenn eine Messung bei Raumtemperatur durchgeführt wird.
- Das Magnetjoch wird zur Probe gefahren.
- Der Signalgenerator und der Verstärker werden aktiviert.
- Die Picoscopemessung wird gestartet. Die Messdaten werden im Bufferspeicher des Picoscopes zwischengespeichert.
- Nach Abschluss der Messung werden die Messdaten via USB–A zur SPS übertragen.
- Je nach Einstellung erfolgen daraufhin weitere Messungen, oder die Messung ist beendet.

- Der Signalgenerator und der Verstärker werden deaktiviert und das Joch von der Probe weggefahren.

Die für die Picoscopemessung nötigen einstellbaren bzw. zu errechnenden Parameter sind in *Tabelle 9* angegeben. Die in der Tabelle orange hinterlegten Parameter SF_{SG_a}, f_{SG_a} und A_{SG_a} dienen der Einstellung des Signalgenerators. Dieser wird, wie eingangs bereits erwähnt, in dieser Phase des Projektes noch nicht in den Automatisierungsprozess miteinbezogen. Die Eingabe dieser Parameter ist, obwohl die Einstellung noch per Hand erfolgt, dennoch notwendig, da die Parameter f_{pico} und t_{pico}, die für die Picoscopemessung benötigt werden, von diesen abhängig sind. Alle Parameter, die nicht aus anderen Parametern errechnet werden, sollen über das UI als User–Input zur Verfügung stehen. Zu achten ist darauf, dass der Bufferspeicher des Picoscopes, abhängig von der eingestellten vertikalen Auflösung, begrenzt ist. Das Fassungsvermögen des Bufferspeichers, in Megasamples (MS) angegeben, kann dem Datenblatt des Picoscopes entnommen werden und darf bei einer Messung nicht überschritten werden, da dieser Speicher erst nach Messende ausgelesen wird.

Die Ansteuerung des Verstärkers und des Linearantriebs des Jochs, sowie die Eingabe des Wärmebehandlungsrezeptes werden aktuell noch nicht in den Automatisierungsprozess eingebunden.

Tabelle 9: Parameter der aktiven Picoscope–Messung

Bezeichnung	Beschreibung	Wertebereich|Default	Einheit
f_{SPS_a}	Sampling Rate des RevPis	[0.01 – 100] | 20	Hz
SF_{SG_a}	Signalform des Signalgenerators	[Dreieck, Sinus] | Dreieck	-
f_{SG_a}	Anregungsfrequenz des Signalgenerators	[0.01 – 10] | 0.05	Hz
A_{SG_a}	Anregungsamplitude des Signalgenerators	[1 – 10] | 6	V_{pp}
f_{pico_a}	Sampling Rate des Picoscopes	$f_{pico\_a} = N_{sampes,P\_a} * f_{SG\_a}$	Hz
t_{pico_a}	Dauer einer Picoscope-Messung	$t_{pico\_a} = N_{p\_a}/f_{SG\_a}$	s
N_{sampes,P_a}	Anzahl der zu messenden Datenpunkte pro Periode des Erregersignals	[1E3 – 1E6] | 1E6	-
N_{p_a}	Anzahl der zur Messung herangezogenen Perioden des Erregersignals	[1 – 10] | 2	-
N_{EM_a}	Anzahl der Einzelmessungen die mit dem Picoscope durchgeführt werden	[1 – 100] | 1	-
Δt_{EM_a}	Zeitabstand zwischen den Einzelmessungen des Picoscopes	[0.05 – 1000] | 1	s

Passiver Modus:

Im passiven Modus erfolgt die elektromagnetische Anregung der Probe über den Induktor der Induktionsanlage. Die Primärspule des Magnetjochs ist dabei nicht aktiv. Die Aufnahme der Systemantwort erfolgt wiederum über die Sekundärspulen. Der Ablauf einer Messung soll dabei wie folgt aussehen:

- Das Magnetjoch wird zur Probe gefahren.
- Die Induktionsanlage führt ein Wärmebehandlungsrezept aus.
- Mit dem Start der Wärmebehandlung startet auch die erste Picoscopemessung. Die Zwischenspeicherung der Messdaten erfolgt im Bufferspeicher des Picoscopes.
- Am Ende der Messung wird der gespeicherte Datenblock vom Bufferspeicher des Picoscopes zum RevPi übertragen.
- Je nach Einstellungen erfolgen noch weitere Messungen. Die Anzahl der insgesamt durchgeführten Messungen N_{EM_p} und die Zeitspanne t_{pico_p} zwischen dem Starten der Einzelmessungen werden dabei vom Benutzer festgelegt.
- Am Ende der Wärmebehandlung fährt das Joch zur Seite. Im Fall, dass direkt nach der passiven Messung noch eine aktive erfolgen soll, soll das Joch nicht weggefahren werden.

Auch hier wird wiederum die Ansteuerung des Linearantriebs des Jochs noch nicht im Automatisierungsprozess implementiert, sondern der Fokus auf den prinzipiellen Mechanismus zur Erfassung der Messdaten gelegt. In *Tabelle 10* sind die relevanten Parameter der passiven Picoscope–Messung aufgelistet. Hier ist zu sagen, dass die Messgrößen des Signalgenerators der Induktionsanlage – in der Tabelle orange hinterlegt – bis dato noch nicht abgegriffen werden. Diese sind zwar in weiterer Folge zur Durchführung aussagekräftiger Messungen notwendig, ihr aktuelles Fehlen hat aber keinen Einfluss auf die prinzipielle Programmstruktur zur automatisierten Durchführung der Magnetjochmessung.

Tabelle 10: Parameter der passiven Picoscope–Messung

Bezeichnung	Beschreibung	Wertebereich|Default	Einheit
f_{SPS_p}	Sampling Rate des RevPis	[0.01 – 100] | 20	Hz
SF_{SG_p}	Signalform des Signalgenerators	Von Induktionsanlage gegeben	-
f_{SG_p}	Anregungsfrequenz des Signalgenerators	[10 – 30] Gegeben und variabel |30	kHz
A_{SG_p}	Anregungsamplitude (Signalgeneratorspannung)	Gegeben und variabel	V_{pp}
f_{pico_p}	Sampling Rate des Picoscopes	$f_{pico\_p} = N_{sampes,P\_p} * f_{SG\_p}$	Hz
t_{pico_p}	Dauer einer Picoscope-Messung	$t_{pico\_p} = N_{p\_p}/f_{SG\_p}$	s
N_{sampes,P_p}	Anzahl der zu messenden Datenpunkte pro Periode des Erregersignals	[1E2 – 1E4] | 1E3	-
N_{p_p}	Anzahl der zur Messung herangezogenen Perioden des Erregersignals	[1 – 10] | 2	-
N_{EM_p}	Anzahl der Einzelmessungen die mit dem Picoscope durchgeführt werden	[10 – 10000] | 1000	-
Δt_{EM_p}	Zeitabstand zwischen den Einzelmessungen des Picoscopes	[0.05 – 10] | 0.05	s

Die in *Tabelle 9* und *Tabelle 10* aufgelisteten Parameter sind, wie bereits erwähnt, jene Parameter, die vom Benutzer über das GUI eingestellt werden können bzw. sich aus den eingegebenen Werten errechnen lassen. Bei der Konfiguration des Picoscopes für eine Einzelmessung sind nur die Frequenz der Picoscopemessung (f_{pico_a} bzw. f_{pico_p}) bzw. deren Kehrwert – als der zeitliche Abstand der einzelnen Messpunkte – und die Gesamtanzahl der zu ermittelnden Messpunkte relevant. Die Frequenz wird dabei in eine „timebase“ umgerechnet, deren Berechnungsmethode je nach gewählter Picoscope–Auflösung und Anzahl der aktiven Kanäle variiert (siehe *Anhang F*).

5.5.1 Hardware

Die relevanten Teilsysteme zur Erfassung der Magnetjochsignale sind in *Abb. 5-19* dargestellt. Diese sind einerseits das Picoscope „5442D MSO“ mit den bis zu vier Tastköpfen zur Erfassung der Messsignale und andererseits der RevPi, dem die Aufgabe der Konfiguration des Picoscopes, der Triggerung der Messung und der Weiterverarbeitung der Daten zukommt. Die Kommunikation zwischen diesen findet über USB–A statt.

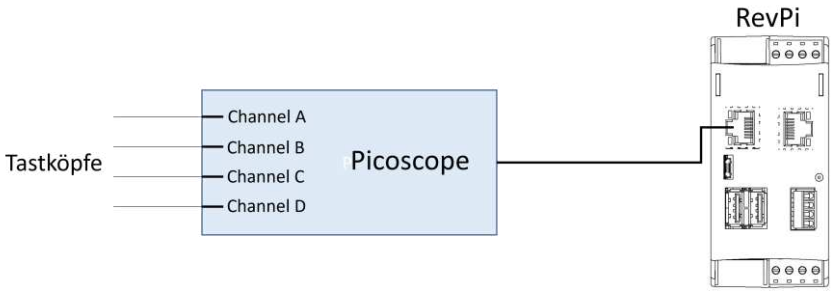


Abb. 5-19: Messaufbau Picoscope

In *Abb. 5-20* ist noch das Magnetjoch, welches der Messdurchführung dient, dargestellt. Dieses wird hier jedoch nicht näher beschrieben (siehe [46,47]).

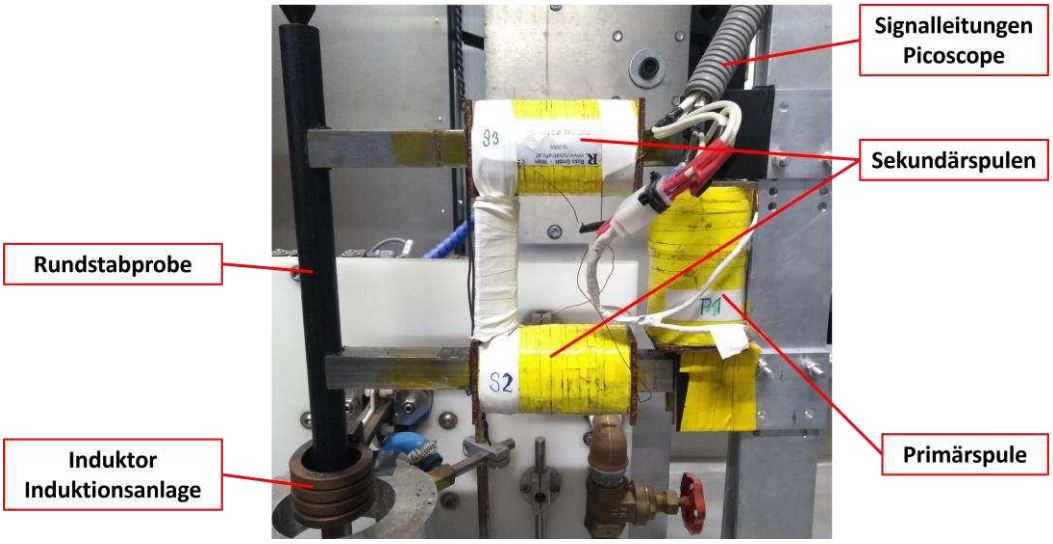


Abb. 5-20: Magnetjoch

5.5.2 Programmierung

Dem Programm zur Erfassung der Messdaten des Picoscopes kommen folgende Aufgaben zu:

- Konfigurierung des Picoscopes vor der Messung
- Starten der Picoscope-Einzelmessungen, auslesen des Bufferspeichers und zwischenspeichern der Daten
- Verarbeitung der erhaltenen Daten nach Messungsende

Zur Umsetzung dieser Aufgaben kommt sowohl Codesys als auch Python zum Einsatz. Das zyklische SPS–Programm, in ST geschrieben, dient dabei zum zeitlich definierten Starten der Picoscope Einzelmessungen. Die Kommunikation der SPS mit dem Picoscope findet ausschließlich mittels Python statt, da dieses vom „application programming interface“ (API) des Picoscopes unterstützt wird. Die Weiterverarbeitung der erhaltenen Daten erfolgt ebenfalls mittels Python. Es kommen wiederum shared me-

mories zum Einsatz, die eine Kommunikation zwischen den Programmteilen in ST und Python ermöglichen. In *Abb. 5-21* ist der grundlegende Programmaufbau bei der Erfassung der Magnetjochsignale durch das Picoscope grafisch dargestellt. Dieser sieht wie folgt aus:

- Ein Pythonskript, als FSM aufgebaut, befindet sich im Wartezustand. Dieses kann über SMs mit dem ST–Code kommunizieren.
- Vor dem Start einer Messung werden über das GUI die gewünschten Messeinstellungen getroffen.
- Diese Einstellungen werden von Codesys über einen SM (SM2_Pico) an das Pythonskript übertragen, welches die Konfiguration an das Picoscope weiterleitet. Die übertragenen Konfigurationsdaten enthalten unter anderem eine Triggervariable, die eine vorgenommene Änderung der Einstellungen anzeigt. Diese Variable wird, nach Übernahme der Daten, vom Pythonskript zurückgesetzt.
- Wird eine Messung gestartet, wird mittels ST-Code über einen SM (SM1_Pico) ein Trigger gesetzt. Das Pythonskript erkennt den Trigger und startet daraufhin die Aufzeichnung von Messdaten durch das Picoscope. Die Messdaten werden während einer Einzelmessung in dem Bufferspeicher des Picoscopes abgelegt.
- Wurde die gewünschte Anzahl an Datenpunkten gemessen, wird der Bufferspeicher ausgelesen und der Trigger zurückgesetzt. Die erhaltenen Daten werden im Arbeitsspeicher des RevPis zwischengespeichert.
- Nun erfolgt, je nach zuvor getroffenen Einstellungen, ein weiteres Triggern einer Messung nach einem definierten Zeitintervall, oder das Messungsende.
- Nach Abschluss der Messung werden die Messdaten als HDF5-File abgespeichert.

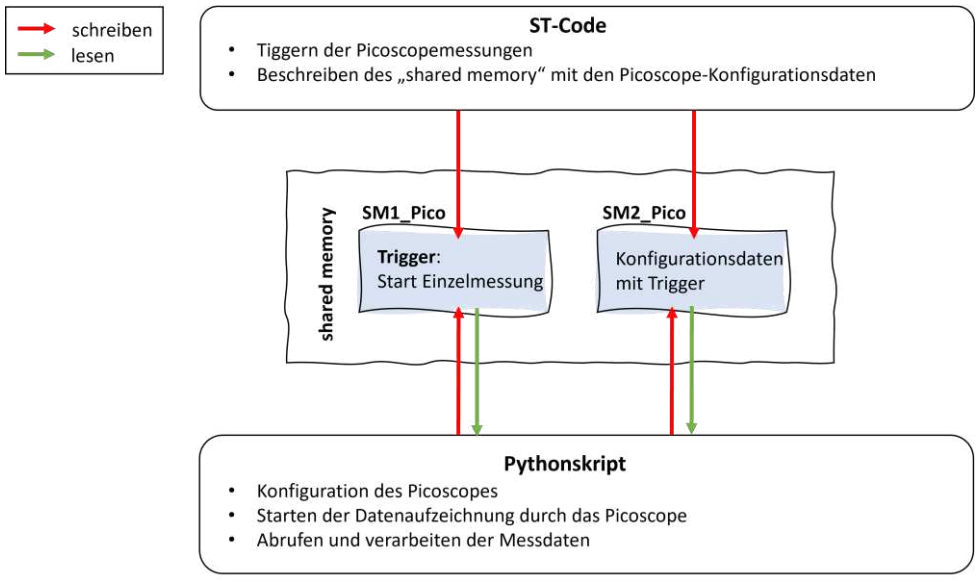


Abb. 5-21: Programmstruktur zur Datenerfassung mittels Picoscope

ST–Code (siehe *Anhang C10 und C12*):

Der mit Codesys verfasste Code besteht aus zwei TASKs (siehe *Abb. 5-22*). Der TASK „PicoLoggerTask“ beinhaltet dabei das Programm „Pico_Logger“, das zum Setzen des Triggers, der den Start einer Picoscopemessung initiiert, dient. Dieser TASK besitzt eine konstante Zykluszeit von 100 µs, um ein möglichst präzises Starten einer Messung zu gewährleisten.

Abb. 5-22: Codesys Programmaufbau (Picoscopemessung)

Das im TASK „SetupTask“ befindliche Programm „Setup_Measurement“ dient der Übertragung der Konfigurationsdaten für das Picoscope mittels des SM2_Pico an das zugehörige Pythonskript. Die Konfigurationsdaten bestehen dabei aus den in *Tabelle 9* bzw. Tabelle 10 aufgelisteten Parametern der Picoscopemessung.

Python–Code (siehe *Anhang B2*):

Das Pythonskript „getData_Pico.py“ dient dem Starten einer der Picoscopemessungen und dem Weiterverarbeiten der Daten. Dieses ist als FSM aufgebaut, die in *Abb. 5-23* dargestellt ist. Die möglichen Zustände sind dabei folgende:

- **State = 100**: Dieser State wird beim Programmstart aufgerufen und stellt eine Verbindung zu dem angeschlossenen Picoscope her. Dabei werden einige Parameter – wie die gewünschte Auflösung und die verwendeten Kanäle – initialisiert. Diese sind zurzeit noch „hard coded“, sprich nicht einstellbar, da sich dieses Skript zum Zeitpunkt der Verfassung der Arbeit noch in der Testphase befindet. Es soll hier allerdings gesagt sein, dass es durchaus Sinn machen würde, diese Parameter in Zukunft über das UI einstellbar zu gestalten, da davon beispielsweise die verfügbare Größe des Bufferspeichers abhängt. Weiters werden hier die zuvor übertragenen Konfigurationsdaten genutzt, um die Messfrequenz, die Anzahl der zu erfassenden Datenpunkte pro Messung und die Anzahl der Einzelmessungen festzulegen.
- **State = 0**: Hier erfolgt der Verbindungsaufbau zum SM1_Pico. Dieser beinhaltet die Triggervariable zum Starten einer Messung. Die Triggervariable ist, wie schon zuvor, ein Integer, der entweder den Wert „0“ (nichts tun) oder „1“ (Messung starten) annehmen kann.
- **State = 1**: Dies ist der „Standby“–Modus. Hier wird in Dauerschleife der Trigger ausgelesen.
- **State = 2**: Wurde ein Trigger = 1 erkannt, dann wird hier die Messdatenaufzeichnung durch das Picoscope aktiviert. Dies geschieht im „Block Mode“, sprich es wird ein Datenblock der gewünschten Größe erfasst und im Bufferspeicher des Picoscopes abgespeichert. Ein Datenblock beinhaltet die Daten einer Einzelmessung. Ist der Datenblock bereit, wird dieser aus dem

Bufferspeicher auf die SPS übertragen. Der Datenblock wird in weiterer Folge als „numpy.ndarray“ über eine „queue“ an das übergeordnete Pythonskript „startProcess.py“ weitergegeben. Die genaue Funktion dieses Skripts wird in *Kapitel 5.6.2* behandelt.

- **State = 3**: Hier erfolgt das Zurücksetzen der Triggervariable auf den Wert „0“.
- **State = 4**: Wurden alle Einzelmessungen durchgeführt, wird die Verbindung zum Picoscope geschlossen, der Trigger ein letztes Mal zurückgesetzt und der benutzte Speicherplatz freigegeben.

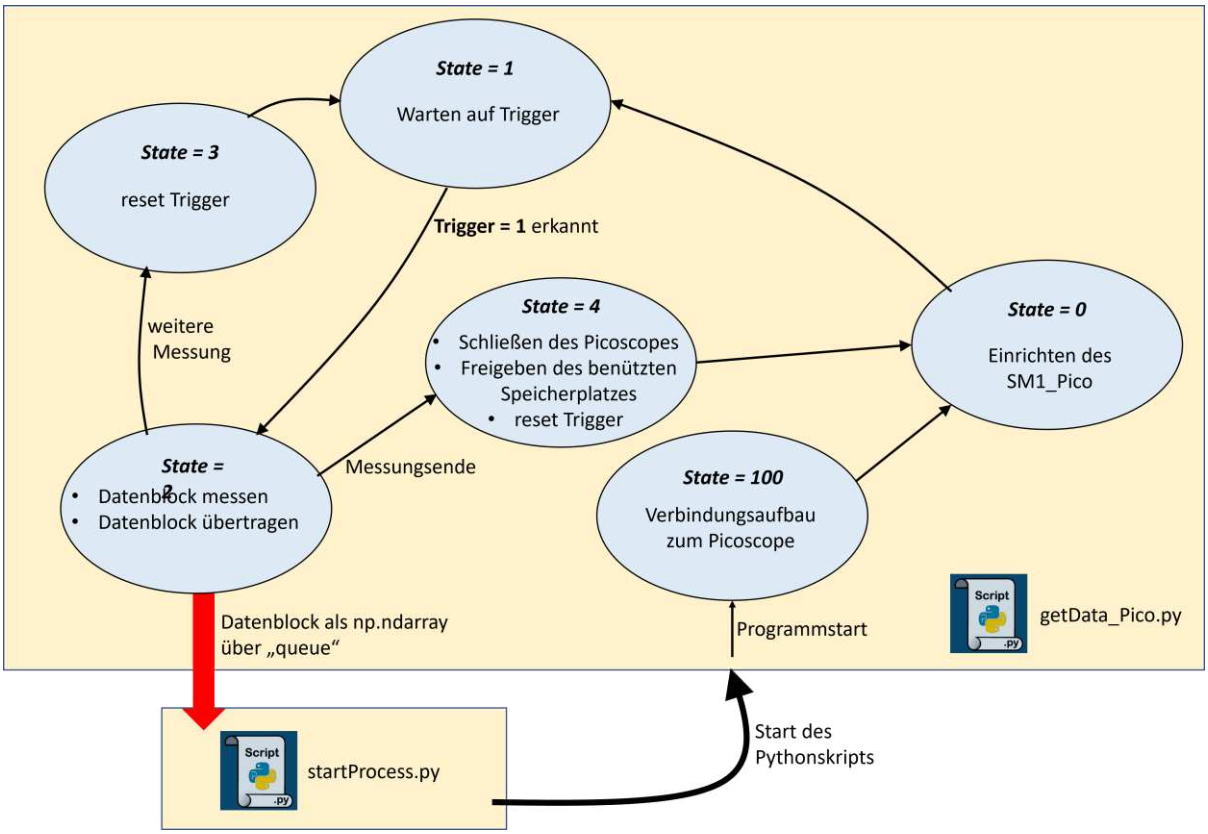


Abb. 5-23: Programmaufbau „getData_Pico.py“

Bezogen auf das Skript „startProcess.py“ ist noch zu sagen, dass dieses, ähnlich wie bei der Verarbeitung der Messwerte der Induktionsanlage, einerseits zur Ausführung des Skripts „getData_Pico.py“ und andererseits zur Zusammenführung und Koordination der einzelnen Teilsysteme dient. Darüber hinaus werden durch „startProcess.py“ die Konfigurationsparameter zum Vornehmen der Messeinstellung an „getData_Pico.py“ übergeben. Das Auslesen des SM2_Pico, der die Konfigurationsparameter beinhaltet, geschieht also nicht direkt in „getData_Pico.py“, sondern bereits übergeordnet in „startProcess.py“. Dieses gibt dann die gewünschten Einstellungen an „getData_Pico.py“ weiter (siehe *Kapitel 5.6.2*).

Um das Zusammenspiel der einzelnen Programmteile besser zu verstehen ist dieses in *Abb. 5-24* noch einmal grafisch dargestellt.

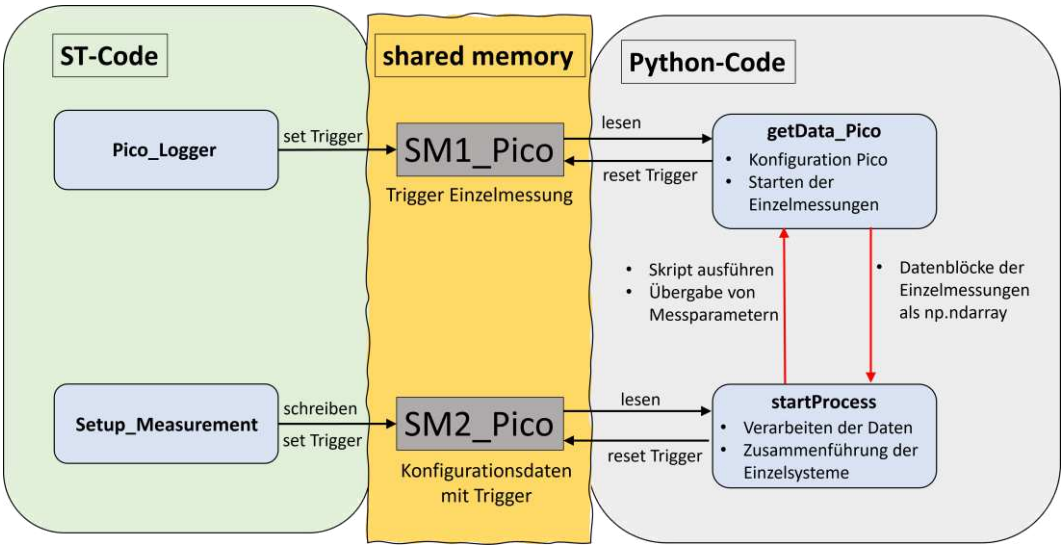


Abb. 5-24: Zusammenspiel der einzelnen Software–Module bei der Durchführung einer Picoscopemessung

5.6 Zusammenführung der Teilsysteme

Im Nachfolgenden wird das sich ergebende Gesamtsystem, bestehend aus den drei zuvor beschriebenen Einzelsystemen, betrachtet. Dabei wird zuerst noch einmal ein Überblick über die Belegung der AIO–Module gegeben. Nachfolgend wird auf die Zusammenführung der vorgestellten einzelnen Programmbausteine eingegangen.

5.6.1 Übersicht AIO–Module

Der hardwaretechnische Aufbau des Schaltschrankes des RevPis und die Anbindung an die einzelnen Messsysteme wurde in den jeweiligen Kapiteln bereits ausreichend behandelt. Hier wird – in *Abb. 5-25* – noch einmal ein Gesamtüberblick über die Belegung der Inputs der AIO–Module mit Angabe der erfassten Messgrößen gegeben.

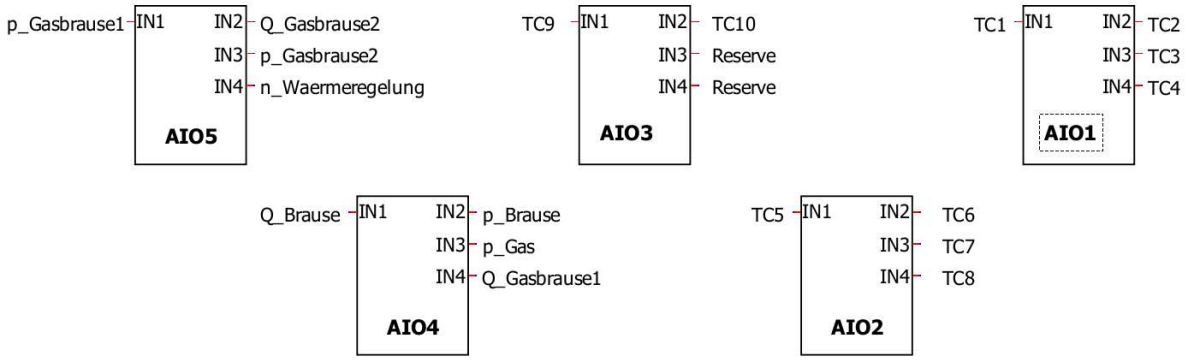


Abb. 5-25: Input–Belegung der AIO–Module

5.6.2 Gesamtübersicht der Programmierung

Bevor das softwaretechnische Gesamtsystem betrachtet werden kann, muss das zuvor schon öfter erwähnte Pythonskript „startProcess.py“ im Detail besprochen werden (siehe *Anhang B3*). Dieses Skript wird mit dem Hochfahren des RevPis gestartet und läuft dauerhaft im Hintergrund ab. Dies ist das einzige Pythonskript, das durchgehend läuft, die anderen bisher besprochenen werden je nach Bedarf gestartet und wieder beendet. *In Abb. 5-26* ist der Aufbau von „startProcess.py“ dargestellt. Dieses ist wiederum als FSM aufgebaut, wobei die States 10, 20 und 30 selbst untergeordnete FSMs darstellen. Bei der Durchführung einer Messung passiert nun folgendes:

- Das Pythonskript befindet sich in State = 1 und wartet auf einen Input.
- Werden nun beispielweise die Messeinstellungen für das Picoscope verändert, wird dies erkannt und die neuen Konfigurationsdaten aus dem schon bekannten SM2_Pico übernommen. Hierbei wird noch keine Verbindung zum Picoscope selbst aufgebaut, die Einstellungen werden lediglich zwischengespeichert und zur Übergabe vorbereitet.
- Weiters wird in Dauerschleife der Inhalt des SM „SM_Setup“ abgefragt. Dies ist ein SM, der eine Triggervariable in Form eines Integers beinhaltet. Dieser Integer gibt an, welche Art von Messung durchgeführt werden soll und kann dabei die Werte Trigger=0 (nichts tun) Trigger=1 (sowohl Anlagen- als auch Picoscopemessung), Trigger=2 (nur Anlagenmessung) und Trigger=3 (nur Picoscopemessung) annehmen. Der Wert Trigger=4 ist ebenfalls möglich. Dieser wird am Ende einer Messung vom Pythonskript in den SM geschrieben, um dem Codesys-Programm den Abschluss eines Messzyklus zu signalisieren.
- Wird eine Messung gestartet, wechselt das Pythonskript je nach Zustand von SM_Setup in den State 10, 20 oder 30. Dort erfolgt dann via dem Python–Modul „multiprocessing“ die Ausführung von entweder „memMap_Data.py“, „getData_Pico.py“ oder beider Skripten. Die so gestarteten Skripts werden als „child-processes“ bezeichnet, wogegen „startProcess.py“ den „parent-process“ darstellt. Die Messspezifikationen für das Picoscope werden hier, falls notwendig, als Funktionsparameter an „getData_Pico.py“ übergeben.
- Nun führen die untergeordneten Skripten die in den jeweiligen Kapiteln bereits besprochenen Funktionen aus. Zwischen den child-processes und dem parent-process wird eine „Queue“ als Schnittstelle genutzt. Diese Queue wird mit den erhaltenen Messdaten befüllt. Von „memMap_Data.py“ kommt es dabei zu einer einmaligen Übertragung der gesamten Messdaten, bei „getData_Pico.py“ wird nach jeder Einzelmessung ein Datenblock übertragen.
- Das Ende einer Messung wird signalisiert, indem von den child-processes der Eintrag ‚End‘ in die Queue geschrieben wird. Wurde von „startProcess.py“ der Abschluss der Datenerfassung aller beteiligten Systeme erkannt, erfolgt die Weiterverarbeitung der erhaltenen Messdaten.

- Nach Abspeicherung der Daten im HDF5-Format werden die child-processes beendet. Weiters wird „SM_Setup“ mit dem Wert Trigger=4 beschrieben, um Codesys den Abschluss einer Messung zu signalisieren. Dieser Trigger wird von Codesys nachfolgend wieder auf 0 zurückgesetzt.

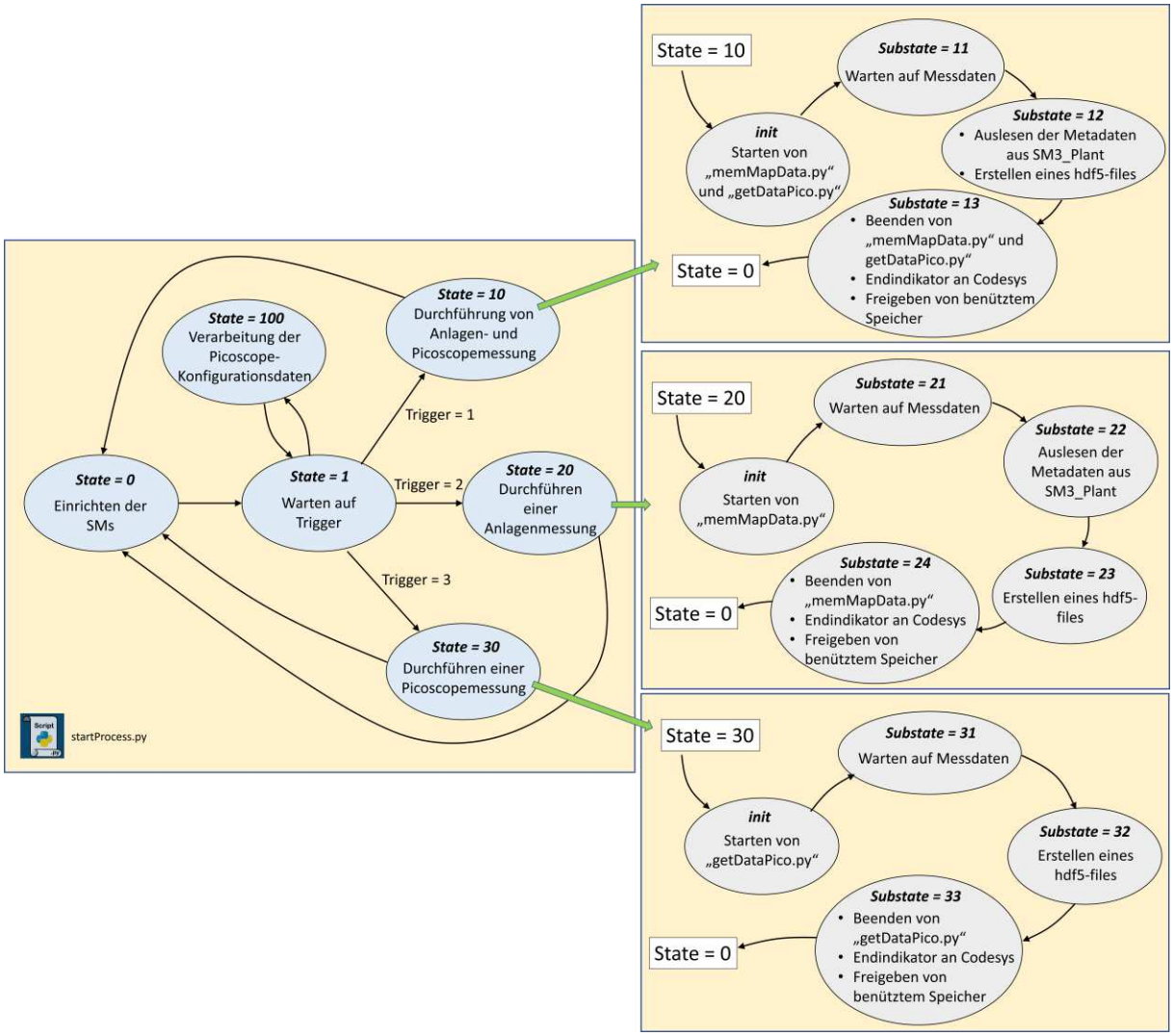


Abb. 5-26: Programmaufbau „startProcess.py“

Das Pythonskript „startProcess.py“ kann folgende Zustände einnehmen:

- **State = 0:** Hier erfolgt die Initialisierung der benötigten SMs.
- **State = 1**: Hier werden die Triggervariablen des SM_Setup und des SM2_Pico in Dauerschleife ausgelesen.
- **State = 100**: Bei Änderung der Einstellungen zur Picoscopemessung werden diese hier zwischengespeichert und die Triggervariable im SM2_Pico zurückgesetzt.
- **State = 10**: Beide Messungen werden ausgeführt, da der Inhalt des SM_Setup von Codesys auf „1“ gesetzt wurde.

- **Init:** Die benötigten child-processes „memMap_Data.py“ und „getDataPico.py“ werden ausgeführt und jeweils eine Queue zur Kommunikation eingerichtet.
- **Substate = 11:** Die Queues werden in Dauerschleife ausgelesen. Bei dem Erhalt von Daten von „getData_Pico.py“ werden diese in einer Liste zwischengespeichert. Die Daten von „memMap_Data.py“ werden im gesamten als „numpy.ndarray“ übernommen.
- **Substate = 12:** Wurde das Ende der Messungen von beiden child-processes signalisiert, werden die erhaltenen Daten weiterverarbeitet. Dazu werden unter anderem die Metadaten der Anlage aus dem SM3_Plant ausgelesen. Dann erfolgt die Ablage der Daten als HDF5–Datei.
- **Substate = 13**: Die laufenden child-processes werden beendet und der benützte Speicher freigegeben. Weiters wird durch das Setzen von Trigger=4 im SM_Setup der Abschluss der Messung an Codesys signalisiert.

- **State = 20:** Es werden nur die Messwerte der Induktionsanlage gespeichert, da der Inhalt des SM_Setup von Codesys auf „2“ gesetzt wurde.
  - **Init:** Ausführen von „memMap_Data.py“ und Einrichten der Queue.
  - **Substate = 21**: Die Queue wird bis zum Erhalt von Daten in Dauerschleife ausgelesen.
  - **Substate = 22**: Hier werden die Metadaten der Anlage aus dem SM3_Plant ausgelesen.
  - **Substate = 23**: Die Messdaten werden als HDF5-Datei abgespeichert.
  - **Substate = 24**: Das Skript „memMap_Data.py” wird beendet und der benützte Speicher freigegeben. Weiters wird durch das Setzen von Trigger=4 im SM_Setup der Abschluss der Messung an Codesys signalisiert.
- **State = 30:** Es wird nur eine Messung mit dem Picoscope durchgeführt.
  - **Init:** Ausführen von „getData_Pico.py“ und Einrichten der Queue.
  - **Substate = 31**: Die Queue wird in Dauerschleife ausgelesen. Die erhaltenen Datenblöcke der Einzelmessungen werden in einer Liste zwischengespeichert.
  - **Substate = 32**: Wurde das Ende der Messung signalisiert, erfolgt die Abspeicherung der Messdaten als HDF5-Datei.
  - **Substate = 24**: Das Skript „getData_Pico.py” wird beendet und der benützte Speicher freigegeben. Weiters wird durch das Setzen von Trigger=4 im SM_Setup der Abschluss der Messung an Codesys signalisiert.

In *Abb. 5-27* ist das Zusammenspiel der einzelnen ST– und Python-Programmbausteine grafisch dargestellt.

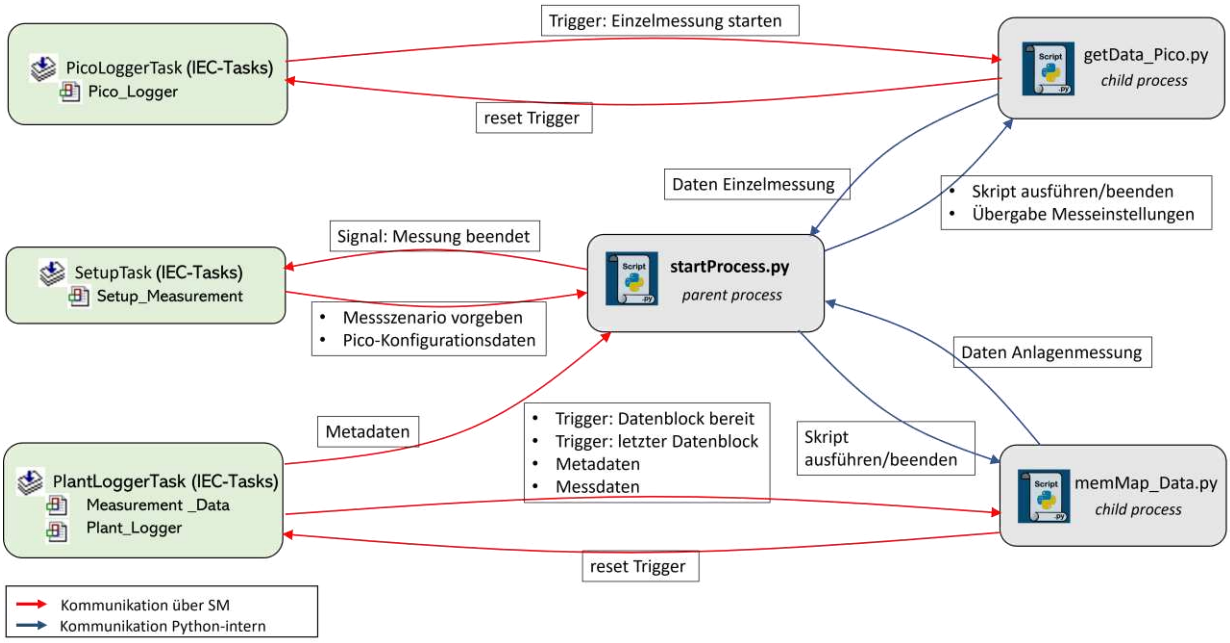


Abb. 5-27: Gesamtübersicht Programmierung

In *Tabelle 11* sind alle Codesys Tasks und die zugeordneten Programme noch einmal aufgelistet. Die Priorität gibt dabei an, welcher Task von Codesys zuerst abgehandelt wird. Da die Triggerung der Picoscope–Einzelmessungen den zeitkritischsten Prozessschritt darstellt, wird diesem die Priorität 1 zugewiesen. Die Zykluszeit der Datenerfassung der Induktionsanlage ist dagegen für gewöhnlich um ein Vielfaches höher, daher genügt es, dem „PlantLoggerTask“ die Priorität 3 zuzuweisen. Der VISU_TASK ist für das GUI zuständig, welches in *Kapitel 5.8* kurz beschrieben wird. Der ShutdownTask dient dem Herunterfahren des RevPis über das GUI.

Tabelle 11: Übersicht der Codesys Tasks

Task	Programm	Aufgabe	Zyklus-zeit	Prio-rität
PicoLoggerTask	Pico_Logger	Triggerung der Picoscope-Einzelmessungen	100 µs	1
PlantLoggerTask	Measurement_Data	Umrechnung der Messsignale der AIO-Module in Messwerte	10 ms/ variabel	3
	Plant_Logger	- Beschreiben des SM mit den Messdaten der Anlage. - Triggerung des Abrufens der Messdaten.		
SetupTask	Setup_Measurement	- Beschreiben des SM mit den Messparametern der Picoscopemessung - Triggern des Messszenarios	1 ms	2
VISU_TASK	VisuElems.Visu_Prg	Visualisierung	100 ms	31
ShutdownTask	Shutdown	Herunterfahren des RevPis über die Visualisierung	Status	20

In *Tabelle 12* sind alle verwendeten Pythonskripts und deren Aufgaben kurz zusammengefasst.

Tabelle 12: Übersicht Pythonskripts

Skript	Aufgabe
startProcess.py	- Koordination der Messungen - Abspeichern der Daten
memMap_Data.py	Erfassen der Anlagemessdaten und Übergabe an „startProcess.py“
getData_Pico.py	Durchführung der Picoscopemessungen und Übergabe der Daten an „startProcess.py“

Tabelle 13 gibt eine Übersicht über alle SMs, die zur Kommunikation zwischen Codesys und Python zum Einsatz kommen.

Tabelle 13: Übersicht der benutzten SMs (1)

Bezeichnung	Name	Inhalt	Aufgabe
SM1_Plant	"_CODESYS_TRIGGER_Plant”	Integer: 0/1	Trigger: Datenblock bereit
SM2_Plant	"_CODESYS_MEMORY_Data"	Messdaten-STRUCT	Beinhaltet die Messdaten der Anlagenmessung
SM3_Plant	"_CODESYS_MEMORY_Meta"	Metadaten-STRUCT	Beinhaltet die Metadaten der Anlagenmessung
SM4_Plant	"_CODESYS_TRIGGER_End”	Integer: 0/1	Trigger: letzter Datenblock bereit
SM1_Pico	"_CODESYS_TRIGGER_Pico"	Integer: 0/1	Trigger: Einzelmessung starten
SM2_Pico	"_CODESYS_MEMORY_PicoSetup"	Konfigurations-daten-STRUCT	Beinhaltet die Konfigurationsdaten zur Durchführung der gewünschten Picoscopemessung
SM_Setup	"_CODESYS_MEMORY_Parent"	Integer: 0/1/2/3/4	Indikator welche Messung gestartet werden soll/Endindikator für Codesys

In *Tabelle 14* sind die Kommunikationsdetails der SMs angegeben.

Tabelle 14: Übersicht der benutzten SMs (2)

Bezeichnung	Kommunikation zwischen	
	Codesys PROGRAM	Pythonskript
SM1_Plant	Plant_Logger	memMap_Data.py
SM2_Plant	Plant_Logger	memMap_Data.py
SM3_Plant	Plant_Logger	memMap_Data.py; startProcess.py
SM4_Plant	Plant_Logger	memMap_Data.py
SM1_Pico	Pico_Logger	getData_Pico.py
SM2_Pico	Setup_Measurement	startProcess.py
SM_Setup	Setup_Measurement	startProcess.py

Abschließend ist zur Programmierung noch zu sagen, dass es bis dato Probleme bei der Kommunikation von Python mit dem Picoscope gibt, die noch nicht restlos geklärt werden konnten. So erfolgt zwar die Triggerung und Durchführung der Messung wie gewünscht, jedoch erfolgt die Rückmeldung, dass ein Datenblock bereit zum Abrufen ist, sehr spät, oft erst nach einigen Sekunden. Dieses Problem tritt selbst dann auf, wenn der Datenblock nur aus wenigen einzelnen Messpunkten – beispielsweise 5 Messpunkten – besteht. Laut Datenblatt des Picoscopes (*Anhang D5*) sollte im „worst case“ zumindest eine Übertragungsrate von 15 Megasamples pro Sekunde (MS/s) möglich sein. Diesbezügliche Nachfragen bei „Pico Technology“ blieben bisher unbeantwortet. Aus diesem Grund konnte der Softwareteil zur Durchführung der Picoscopemessung nicht ausreichend getestet werden. Deshalb wurde auch die Weiterverarbeitung der Daten der Picoscopemessung noch nicht implementiert, da dies zu diesem Zeitpunkt wenig Sinn machen würde. Der Code wurde allerdings dahingehend vorbereitet, dass eine Implementierung dieser Funktionen keinen großen Aufwand nach sich ziehen sollte.

5.6.2.1 Synchronisation der Datensätze

Bisher außer Acht gelassen wurde die, eingangs erwähnte, zeitliche Synchronisation der Messdaten. Eine Synchronisation ist dann erforderlich, wenn sowohl die Anlagedaten aufgezeichnet als auch Picoscopemessungen durchgeführt werden. Der Startzeitpunkt der Anlagemessung ist in den Metadaten, die von Python abgefragt werden, hinterlegt. Der Zeitstempel des Startens der Picoscope–Einzelmessungen wird beim Triggern einer Messung gespeichert und mit den jeweiligen Messdatenblöcken an „startProcess.py“ übergeben. Sind diese zwei Zeitinformationen bekannt, kann beim Abspeichern der Messung als HDF5–File der Zeitstempel des Picoscopes in einen Zeitpunkt relativ zum Start der Datenerfassung der Induktionsanlage umgerechnet werden.

Das Abspeichern der Zeitstempel ist zwar bereits im Code implementiert, die Umrechnung dieser in eine relative Zeit jedoch noch nicht, da es, wie gesagt, noch Probleme bei der Datenerfassung mit dem Picoscope gibt. Solange noch nicht restlos geklärt ist, welche genaue Datenstruktur nach Durchführung einer Picoscopemessung vorliegt, ist es auch nicht sinnvoll einen Code zu deren Weiterverarbeitung zu implementieren.
Eine weitere Möglichkeit zur Synchronisation der Daten wäre, die Nummer des Datenpunktes der Anlagemessung, bei dem die Picoscopemessung gestartet wird, zu speichern und diese Nummer dann beispielsweise mittels Codesys mitsamt dem Trigger für das Picoscope in den SM zu schreiben. In weiterer Folge könnte dieser Wert dann von „getData_pico.py“ an „startProcess.py“ übergeben werden. Dann wäre bekannt, bei dem wievielten Messwert der Anlage die Picoscopemessung gestartet wurde. Dies könnte in den Metadaten des Picoscope–Datensatzes im HDF5–File hinterlegt werden, um somit eine Zuordnung der Messdatensätze zueinander zu ermöglichen.
Es wurde allerdings bereits eine Möglichkeit geschaffen, die bei einer herkömmlichen, manuell durchgeführten Messung erhaltenen Datensätze zeitlich zu synchronisieren. Dazu müssen diese Datensätze als csv–Datei vorliegen und der Startzeitpunkt der Picoscopemessung relativ zum Startzeitpunkt der Datenaufzeichnung der Induktionsanlage bekannt sein. Dann kann via dem Pythonskript in Anhang A eine automatisierte Synchronisation durchgeführt werden. Dabei erhält man drei csv–Dateien als Resultat. Eine beinhaltet alle ursprünglichen Daten der beiden Messungen mit synchronisierten Zeitstempeln. Eine weitere enthält die Anlage– und Picoscopemessdatenpaarungen deren Zeitstempel einander am nächsten liegen und die dritte Datei beinhaltet alle Messdaten der ursprünglichen csv-Dateien in synchronisierter Form, wobei hier im Zeitbereich einer Picoscopemessung die aufgrund der größeren Zeitintervalle der Datenerfassung fehlenden Anlagemessdaten mittels linearer Interpolation ergänzt werden.

5.7 Datenstruktur

Die Messdaten werden im HDF5 Format abgespeichert. In *Abb. 5-28* ist ein Beispiel angegeben, wie so eine Datei aussehen könnte. Der Dateiname setzt sich aus dem Datum und der Uhrzeit der Messdurchführung zusammen. Die Gruppe „Induction Plant Data“ ist in die Untergruppen „Druck“, „Durchfluss“, „Temperatur“, „Zeit“ und „Andere“ unterteilt, die wiederum die jeweiligen Datensätze, die von der Anlage erfasst werden, beinhalten. Weiters sind in den Metadaten das Erstelldatum, der Start– und Endzeitpunkt der Messung und die Frequenz der Datenerfassung der Anlagedaten hinterlegt. Die Daten der Picoscopemessung sind – aus zuvor behandelten Gründen – hier noch nicht eingebunden. Der Code zur Abspeicherung der Anlagedaten kann aber als Vorlage dienen, um in weiterer Folge auch die

Picoscopemessdaten einzugliedern. Die Datenstruktur wird bei der Speicherung der Daten in „start-Process.py“ festgelegt und kann nach Bedarf angepasst werden.

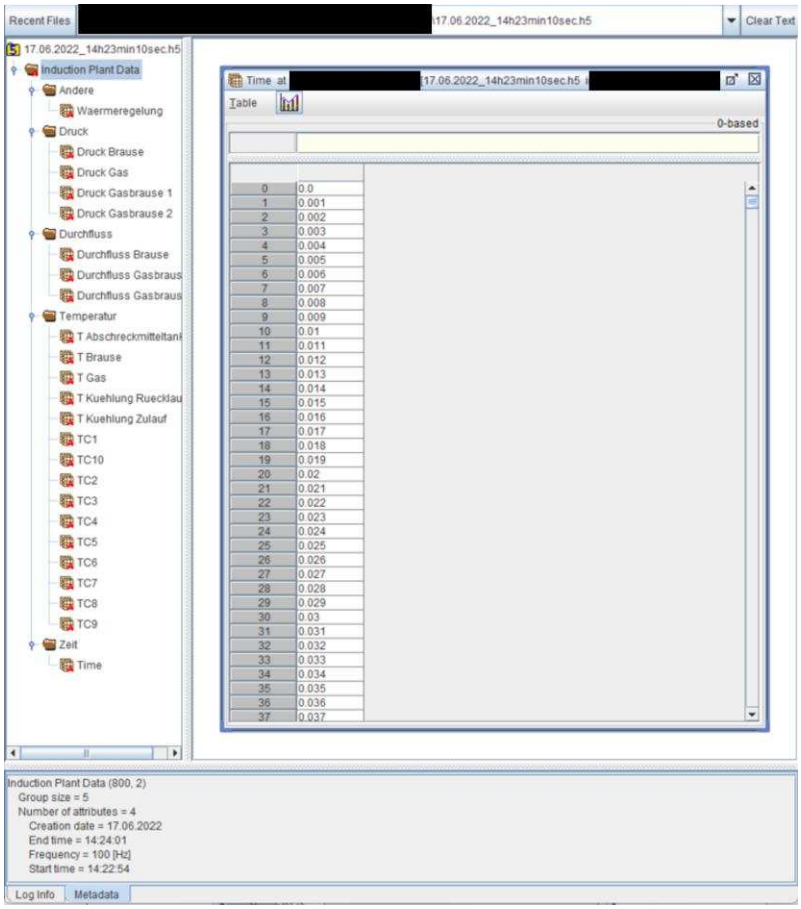


Abb. 5-28: Aufbau der HDF5-Datei

5.8 Graphical User Interface

Das erstellte GUI ist in *Abb. 5-29* dargestellt. Dieses ist als Webvisualisierung ausgeführt, sprich es kann über einen Standard–Internetbrowser aufgerufen werden, sofern sich der RevPi und der Laptop des Benutzers im selben Netzwerk befinden. Die einzugebende Adresse setzt sich dabei wie folgt zusammen: „http://<IP address of webserver>:<port of webserver>/<name of HTM-file>“. Im konkreten Fall ist dies die Adresse „http://169.254.107.36:8080/webvisu.htm“. Diese kann aber unter Umständen variieren.

Das GUI dient als vorläufige Benutzeroberfläche – vorrangig zu Testzwecken – und muss in weiterer Folge noch an die exakten Bedürfnisse der Messdurchführung angepasst werden. Die erfassten Messdaten der Thermoelemente und der Anlage werden in Echtzeit angezeigt. Dabei ist zu sagen, dass die Temperaturen, die von der Anlage mittels PT100 gemessen werden, zwar am Interface aufscheinen,

aber bis dato noch nicht abgegriffen werden. Das Starten einer Datenaufzeichnung gestaltet sich bisweilen noch etwas kompliziert und sollte, sobald die Testphase der Software abgeschlossen ist, abgeändert werden.

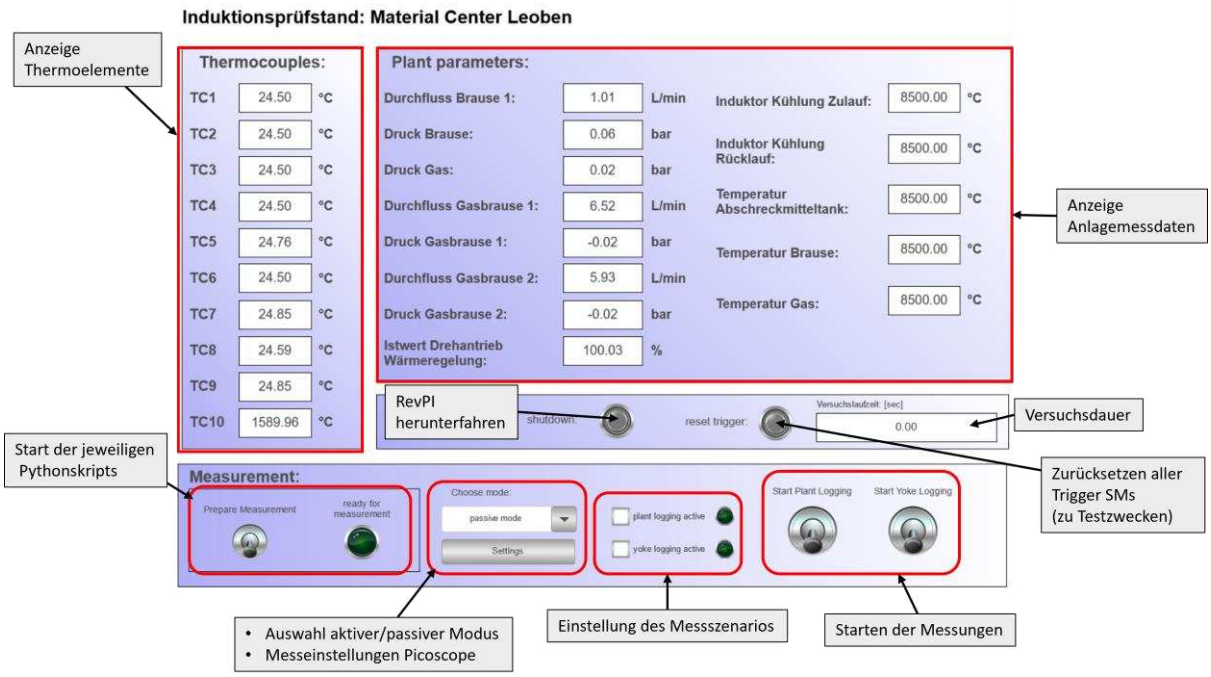


Abb. 5-29: Vorläufiges User Interface

Soll eine Messung durchgeführt werden, sind folgende Schritte durchzuführen (*Abb. 5-30*):

- Zuerst werden die gewünschten Messeinstellungen vorgenommen (*Abb. 5-30–(1)*). Dazu zählen die Auswahl der zu erfassenden Systeme („plant logging active“, „yoke logging acitve“) und gegebenenfalls die Auswahl des aktiven oder passiven Modus (Dropdown–Menü) für die Durchführung einer Magnetjochmessung. Über „Settings“ können die Parameter für die Picoscopemessung eingestellt werden. Das sich dabei öffnende Menü ist in *Abb. 5-31* dargestellt. Dieses sieht für die aktive und passive Messung ident aus.
- Wurden alle gewünschten Einstellungen getroffen, wird der Schalter „Prepare Measurement“ betätigt (*Abb. 5-30–(2)*). Dieser sorgt dafür, dass über das Skript „startProcess.py“ die für das jeweilige Messszenario notwendigen weiteren Skripten ausgeführt werden. Nun laufen diese Skripts im Hintergrund ab und warten auf die Triggerung einer Messung.
- Zu guter Letzt können nun die Messungen der Anlage und/oder des Magnetjochs mittels der jeweiligen Schalter gestartet werden (*Abb. 5-30–(3)*). Nach Durchführung der eingestellten Einzelmessungen wird der Schalter „Start Yoke Logging“ automatisch deaktiviert. Die Aufzeichnung der Anlagedaten muss manuell über ein erneutes Betätigen des Schalters „Start Plant Logging“ beendet werden.

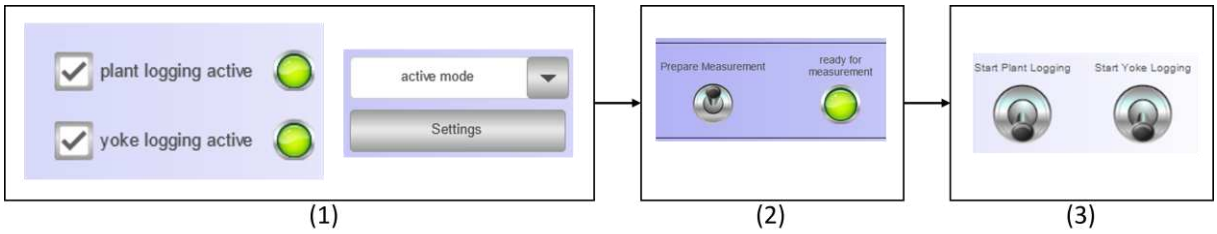


Abb. 5-30: Starten einer Datenaufzeichnung

Diese Vorgehensweise beim Start einer Messung ist für den praktischen Gebrauch nicht ideal und anfällig für benutzerseitige Fehler, sie eignet sich jedoch gut um in der Entwicklungsphase bestimmte Aspekte der Software gezielt testen zu können.

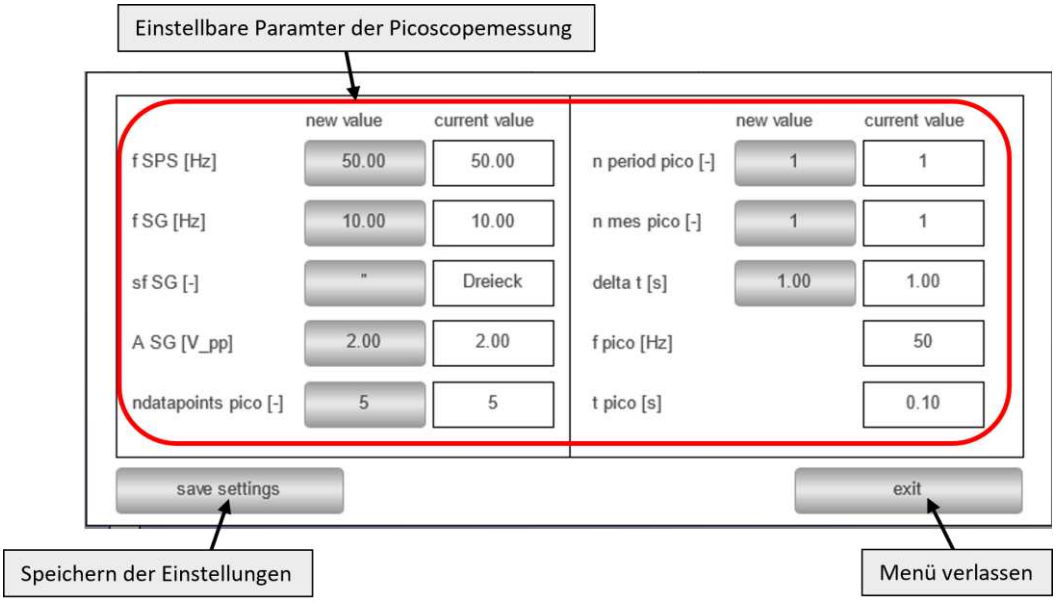


Abb. 5-31: Menü zur Einstellung der Picoscope-Parameter des aktiven/passiven Modus

6 Zusammenfassung und Ausblick

Da für dieses Projekt von vornherein zumindest drei Masterarbeiten angedacht waren, war nicht zu erwarten, dass am Ende dieser Arbeit ein komplettes, funktionierendes Gesamtsystem stehen wird. Es sind im Zuge dieser Arbeit allerdings ein paar Probleme aufgetreten, die im nachfolgenden kurz behandelt werden, um zukünftige Arbeiten an dem Projekt zu erleichtern.

So war es beispielsweise nicht möglich, in die Simatic S7–1500, die SPS der Induktionsanlage, einzugreifen und somit das Erfassen der Messwerte zu vereinfachen. Hier musste der Umweg über das Abgreifen der physikalischen Messsignale gemacht werden.

Das führt direkt zum nächsten Problem. Nicht alle Signale sind so leicht zugänglich wie die, die in dieser Arbeit bereits implementiert wurden. So gestaltet sich das Abgreifen der PT100 Messwerte der Anlage insofern komplex, da bei einem Pt100 ja ein bekannter Messstrom zur Berechnung des Widerstandes herangezogen wird. Dieser Messstrom wird von der Anlagen–SPS bereitgestellt. Das führt dazu, dass die Pt100/Pt1000 Eingänge der AIO–Module des RevPis nicht verwendet werden können, ohne die Signalleitungen der Anlage zu unterbrechen, was allerdings einen zu großen Eingriff in das bestehende System darstellen würde. Eine Möglichkeit dieses Problem zu umgehen wäre, sowohl die Messströme, die von der Anlagen–SPS kommen, als auch die Spannungsabfälle an den Pt100 mittels der Inputkanäle des RevPis zu messen und sich daraus die ergebenden Temperaturen zu berechnen. Dafür wären allerdings pro Pt100 zwei Inputkanäle notwendig, die beim derzeitigen Aufbau nicht zur Verfügung stehen. Auch ist die Präzision des Messergebnisses bei einem solchen Aufbau zu hinterfragen. Es ist weiters noch nicht restlos geklärt, wie und wo die Kennwerte des Signalgenerators der Anlage abgegriffen werden können.

Das Problem der zu geringen Anzahl an zur Verfügung stehenden Inputs wurde bereits kurz angesprochen. Dieses rührt daher, dass der gekaufte „RevPi Connect+ feat. Codesys“ nur die Möglichkeit zur Anbindung von fünf AIO–Modulen bietet, was in Summe 20 Inputkanälen entspricht. Das limitiert den Spielraum zur Erfassung zusätzlicher Signale erheblich. Ein „RevPi Core“ würde dagegen die Möglichkeit der Anbindung von bis zu 10 AIO–Modulen, sprich 40 Inputkanälen, bieten. Bevor allerdings der RevPi voreilig ausgetauscht wird, sollte überlegt werden, ob die Durchführung aller gewünschten Aufgaben in weiterer Folge mit dem neuen System möglich wäre.

Ein weiteres Problem mit dem gekauften RevPi ist, dass dieser mit einem 32-bit und nicht mit einem 64-bit Prozessor ausgestattet ist. Das hat unter anderem zu Problemen bei der Installation der benötigten Picoscopetreiber geführt, da herstellerseitig, bezogen auf Linux, nur noch 64-bit Systeme unterstützt werden. Es sind jedoch noch ältere Treiber für 32-bit Systeme in Archiven von Pico Technology auffindbar. Es ist durchaus möglich, dass das Problem des langsamen Datentransfers zwischen RevPi

und Picoscope auf diesen Umstand zurückzuführen ist. Mittlerweile ist allerdings ebenfalls eine 64-bit Version des gekauften RevPis im Handel verfügbar.

Bezogen auf die Programmierung wurde versucht die für die SPS–Programmierung standardmäßig eingesetzte Programmiersprache ST mit der „Open Source"–Programmiersprache Python zu verbinden, um so die jeweiligen Vorteile der einzelnen Sprachen miteinander zu verknüpfen. Bezogen auf ST ist dies vor allem die Nutzung der Entwicklungsumgebung Codesys, die das Einlesen der Inputkanäle und den Aufbau des zyklischen Programmablaufs der SPS stark vereinfacht. Da allerdings durch Codesys das Ausführen von Pythonskripts parallel zum zyklischen ST–Programm nicht unterstützt wird, gestaltet sich die Verbindung dieser beiden Sprachen – wie in der Arbeit besprochen – relativ kompliziert. Hier ist abzuwägen, ob es nicht sinnvoll wäre den Mehraufwand bei der Programmierung in Kauf zu nehmen und das gesamte System als Pythonprogramm aufzubauen, da ja Python–Module zur Verfügung stehen, die gezielt für die zyklische Programmierung des RevPis entwickelt wurden.

Abschließend ist zu sagen, dass die hier vorgestellte Arbeit als Framework für weiterführende Arbeiten dienen soll. Es wurde ein Grundgerüst geschaffen, das prinzipiell funktioniert, aber das noch erweitert bzw. unter Umständen abgeändert werden muss. Auch wurden im Verlauf der Arbeit, wie so oft bei größeren Projekten, Kenntnisse darüber gewonnen, was nicht funktioniert, oder was vielleicht von Anfang an hätte anders gemacht werden können. So bietet der RevPi als SPS zwar durch die offene Linux–basierte Plattform viele Möglichkeiten, seine Einsatzmöglichkeiten sind jedoch durch die Limitierung auf fünf bzw. zehn AIO–Module begrenzt. Will man allerdings ein nicht proprietäres System auf (Teil–)Basis von Python aufbauen und kommt mit den verfügbaren Kanälen aus, ist der RevPi sicher keine schlechte Wahl.

Als weiterführende Schritte dieses Projekts würde ich folgendes empfehlen:

- Abgreifen der weiteren notwendigen Anlagensignale. Das sind konkret die Kennwerte des Signalgenerators und die Messwerte der Pt100–Widerstände. Dazu wäre allerdings ein Austausch des vorhandenen RevPis notwendig, um mehr AIO–Module anschließen zu können. Zuvor sollte auch überlegt werden, ob und wie das Abgreifen dieser Messwerte auf sinnvolle Art und Weise machbar ist. Unter Umständen muss noch einmal versucht werden in die Anlagen–SPS einzugreifen, falls der Zugang zu manchen Messwerten sonst nicht möglich ist.
- Das Picoscope als Einzelsystem sollte noch einmal genauer behandelt werden, um die Ursache für die langsame Datenübertragung herauszufinden. Dabei ist es wie gesagt möglich, dass sich die Probleme aufgrund des 32-bit Systems des RevPis ergeben.
- Sobald die Anlagedaten vollständig sind und die Picoscopemessung wie gewünscht funktioniert, kann der bestehende Code adaptiert werden, um die neuen Parameter bzw. die Messergebnisse einzubinden.

- In weiterer Folge kann dann das GUI auf die konkreten Bedürfnisse angepasst werden.
- Weiters wäre es eine Überlegung wert, die gesamte Software als Pythonprogramm aufzubauen, um so die Kommunikation der einzelnen Teilsysteme untereinander zu vereinfachen und das Fehlerpotential zu minimieren.
- Zu guter Letzt kann dann das Einbinden der Steuerung des Signalgenerators und Verstärkers bzw. des Linearantriebs erfolgen. Dies kann softwaretechnisch durch das Hinzufügen weiterer „States“ in den betreffenden Skripten umgesetzt werden.

Abbildungsverzeichnis

Tabellenverzeichnis

Literaturverzeichnis

1. Culot, G.; Nassimbeni, G.; Orzes, G.; Sartor, M. Behind the definition of Industry 4.0: Analysis and open questions. *International Journal of Production Economics* **2020**, doi:10.1016/j.ijpe.2020.107617.
2. Pistorius, J. *Industrie 4.0 – Schlüsseltechnologien für die Produktion;* Springer Berlin Heidelberg: Berlin, Heidelberg, 2020, ISBN 978-3-662-61579-9.
3. Patnaik, S. *New Paradigm of Industry 4.0*: *Internet of Things, Big Data & Cyber Physical Systems;* Springer Nature Switzerland AG: 6330 Cham, 2020.
4. Emiliano Sisinni; Abusayeed Saifullah; Song Han; Ulf Jennehag; Mikael Gidlund. Industrial Internet of Things: Challenges, Opportunities, and Directions.
5. Da Silva, G.C.; Kaminski, P.C. From Embedded Systems (ES) to Cyber-Physical Systems (CPS): An Analysis of Transitory Stage of Automotive Manufacturing in the Industry 4.0 Scenario. In *SAE Technical Paper Series.* 25th SAE BRASIL International Congress and Display, OCT. 25, 2016; SAE International400 Commonwealth Drive, Warrendale, PA, United States, 2016.
6. Sorger, M.; Ralph, B.J.; Hartl, K.; Woschank, M.; Stockinger, M. Big Data in the Metal Processing Value Chain: A Systematic Digitalization Approach under Special Consideration of Standardization and SMEs. *Applied Sciences* **2021**, *11*, 9021, doi:10.3390/app11199021.
7. Ustundag, A.; Cevikcan, E. *Industry 4.0: Managing The Digital Transformation;* Springer International Publishing: Cham, 2018, ISBN 978-3-319-57869-9.
8. Wu, X.; Goepp, V.; Siadat, A. Concept and engineering development of cyber physical production systems: a systematic literature review. *Int J Adv Manuf Technol* **2020**, *111*, 243–261, doi:10.1007/s00170-020-06110-2.
9. Monostori, L. Cyber-physical Production Systems: Roots, Expectations and R&D Challenges. *Procedia CIRP* **2014**, *17*, 9–13, doi:10.1016/j.procir.2014.03.115.
10. Lee, J.; Noh, S.D.; Kim, H.-J.; Kang, Y.-S. Implementation of Cyber-Physical Production Systems for Quality Prediction and Operation Control in Metal Casting. *Sensors (Basel)* **2018**, *18*, doi:10.3390/s18051428.
11. Kagermann, H.; Wahlster, W. Ten Years of Industrie 4.0. *Sci* **2022**, *4*, 26, doi:10.3390/sci4030026.
12. Liao, Y.; Deschamps, F.; Loures, E.d.F.R.; Ramos, L.F.P. Past, present and future of Industry 4.0 - a systematic literature review and research agenda proposal. *International Journal of Production Research* **2017**, *55*, doi:10.1080/00207543.2017.1308576.
13. Al-Maeeni, S.S.H.; Kuhnhen, C.; Engel, B.; Schiller, M. Smart retrofitting of machine tools in the context of industry 4.0. *Procedia CIRP* **2020**, *88*, 369–374, doi:10.1016/j.procir.2020.05.064.
14. *Industrie 4.0 in Produktion, Automatisierung und Logistik*: *Anwendung, Technologien, Migration;* Bauernhansl, T., Ed.; Springer Vieweg: Wiesbaden, 2014, ISBN 978-3-658-04682-8.
15. Meudt, T.; Pohl, M.; Metternich, J. Die Automatisierungspyramide - Ein Literaturüberblick **2017**.
16. Körner, M.-F.; Bauer, D.; Keller, R.; Rösch, M.; Schlereth, A.; Simon, P.; Bauernhansl, T.; Fridgen, G.; Reinhart, G. Extending the Automation Pyramid for Industrial Demand Response. *Procedia CIRP* **2019**, *81*, 998–1003, doi:10.1016/j.procir.2019.03.241.
17. Theuer, H.K. Beherrschung komplexer Produktionsprozesse durch Autonomie; Universität Potsdam, 2022.
18. Kessler, S.; Lenz, H.; Rytir, C.; Schneiderhan, M. ATP edition: Automatisierungstechnische Praxis **2017**, 20–35.
19. Rocca, R.; Rosa, P.; Sassanelli, C.; Fumagalli, L.; Terzi, S. Integrating Virtual Reality and Digital Twin in Circular Economy Practices: A Laboratory Application Case. *Sustainability* **2020**, *12*, 2286, doi:10.3390/su12062286.
20. *Sensoren in Wissenschaft und Technik*: *Funktionsweise und Einsatzgebiete;* Hering, E.; Schönfelder, G., Eds., 2., überarbeitete und aktualisierte Auflage; Springer Fachmedien Wiesbaden GmbH: Wiesbaden, 2018, ISBN 3658125616.
21. *Dubbel Taschenbuch für den Maschinenbau;* Bender, B.; Göhlich, D., Eds., 26., überarbeitete Auflage; Springer Vieweg: Berlin, Heidelberg, 2020, ISBN 978-3-662-59713-2.
22. Parthier, R. *Messtechnik: SI-Einheitensystem – Messergebnisse bewerten – Elektrische Messtechnik anwenden*, 10th ed.; Springer Fachmedien Wiesbaden GmbH, 2022, ISBN 978-3-658-37970-4.

23. Lerch, R. *Elektrische Messtechnik*, 7., aktualisierte Auflage; Springer Vieweg: Berlin, Heidelberg, 2016, ISBN 978-3-662-46941-5.
24. Bernstein, H. *Messelektronik und Sensoren;* Springer Fachmedien Wiesbaden: Wiesbaden, 2014, ISBN 978-3-658-00548-1.
25. Heinrich, B.; Linke, P.; Glöckler, M. *Grundlagen Automatisierung*: *Erfassen - Steuern - Regeln*, 3., überarbeitete und erweitere Auflage; Springer Vieweg: Wiesbaden, Heidelberg, 2020, ISBN 978-3-658-27322-4.
26. Die piezoresistive Druckmesstechnik | KELLER Druckmesstechnik. Available online: https://keller-druck.com/de/unternehmen/news/die-piezoresistive-druckmesstechnik (accessed on 7 February 2023).
27. Induktive Druckaufnehmer - HAWE Österreich. Available online: https://www.hawe.com/de-at/fluidlex-ikon/induktive-druckaufnehmer/ (accessed on 7 February 2023).
28. Tumański, S. *Handbook of magnetic measurements*, Online-Ausg; Taylor & Fracis: Boca Raton, 2011, ISBN 9781283257442.
29. Industrial Raspberry Pi - Revolution Pi |. RTD-Messung | Industrial Raspberry Pi - Revolution Pi. Available online: https://revolutionpi.de/tutorials/revpi-aio/rtd-messung/ (accessed on 24 October 2022).
30. *Speicherprogrammierbare Steuerung - SPS*: *Praktisches Programmieren mit STEP5 und STEP7 nach IEC 61131;* Bernstein, H., Ed.; De Gruyter Oldenbourg: Berlin, 2018, ISBN 9783110556018.
31. Mahmud, S.; Lin, X.; Kim, J.-H. Interface for Human Machine Interaction for assistant devices: A Review. In *2020 10th Annual Computing and Communication Workshop and Conference (CCWC).* 2020 10th Annual Computing and Communication Workshop and Conference (CCWC), Las Vegas, NV, USA, 06–08 Jan. 2020; IEEE, 2020; pp 768–773, ISBN 978-1-7281-3783-4.
32. Abdur, M.; Ali, M.; Hussain, K.; Ullah, S. A Survey on User Interfaces for Interaction with Human and Machines. *ijacsa* **2017**, *8*, doi:10.14569/IJACSA.2017.080763.
33. Soldatos, J.; Lazaro, O.; Cavadini, F. The Digital Shopfloor: Industrial Automation in the Industry 4.0 Era; pp 1–496.
34. Guerreiro, B.V.; Lins, R.G.; Sun, J.; Schmitt, R. Definition of Smart Retrofitting: First Steps for a Company to Deploy Aspects of Industry 4.0. In *Advances in Manufacturing*; Hamrol, A., Ciszak, O., Legutko, S., Jurczyk, M., Eds.; Springer International Publishing: Cham, 2018; pp 161–170, ISBN 978-3-319-68618-9.
35. Sanchez-Londono, D.; Barbieri, G.; Fumagalli, L. Smart retrofitting in maintenance: a systematic literature review. *J Intell Manuf* **2022**, doi:10.1007/s10845-022-02002-2.
36. Marinescu, M. *Elektrische und magnetische Felder*: *Eine praxisorientierte Einführung ; [Extras im Web*, 3., bearb. Aufl.; Springer: Berlin, Heidelberg, 2012, ISBN 9783642242199.
37. Benkowsky, G. *Induktionserwärmung*: *Härten, Gühen, Schmelzen, Löten, Schweißen*, 4th ed.; VEB Verlag Technik Berlin: Berlin/DDR, 1980.
38. Rudnev, V.; Loveless, D.; Cook, R.L. *Handbook of induction heating*, Second edition; CRC Press Taylor & Francis Group: Boca Raton, London, New York, 2017, ISBN 978-1-1387-4874-3.
39. Induktionsanlage - SMS ELOTHERM. Available online: https://www.sms-elotherm.com/ (accessed on 20 December 2022).
40. Henke, H. *Elektromagnetische Felder*: *Theorie und Anwendung*, 6., erweiterte Auflage; Springer Vieweg: Berlin, Heidelberg, 2020, ISBN 978-3-662-62234-6.
41. Humans.txt. Thermoelement Hutschienen-Temperaturtransmitter Endress+Hauser iTEMP TMT128-AJAHA. Available online: https://www.automation24.at/thermoelement-hutschienen-temperaturtransmitter-endress-hauser-itemp-tmt128-ajaha (accessed on 16 January 2023).
42. Industrial Raspberry Pi - Revolution Pi |. Industrie PC auf Raspberry Pi Basis Überblick | Industrial Raspberry Pi - Revolution Pi. Available online: https://revolutionpi.de/revolution-pi-serie/ (accessed on 17 January 2023).
43. TIOBE. TIOBE Index - TIOBE. Available online: https://www.tiobe.com/tiobe-index/ (accessed on 17 January 2023).
44. Hierarchical Data Formats - What is HDF5? | NSF NEON | Open Data to Understand our Ecosystems. Available online: https://www.neonscience.org/resources/learning-hub/tutorials/about-hdf5 (accessed on 23 January 2023).
45. The HDF5 File Format and MODFLOW | Aquaveo.com. Available online: https://www.aquaveo.com/blog/2020/09/02/hdf5-file-format-and-modflow (accessed on 23 January 2023).

46. Jászfi, V.; Raninger, P.; Riedler, J.M.; Prevedel, P.; Mevec, D.G.; Wilson, J.; Ebner, R. Indirect yoke-based B-H hysteresis measurement method determining the magnetic properties of macroscopic ferromagnetic samples part I: Room temperature. *Journal of Magnetism and Magnetic Materials* **2022**, *560*, 169655, doi:10.1016/j.jmmm.2022.169655.
47. Jászfi, V.; Raninger, P.; Riedler, J.M.; Prevedel, P.; Mevec, D.G.; Godai, Y.; Ebner, R. Introduction of a novel yoke-based electromagnetic measurement method with high temperature application possibilities. *Journal of Magnetism and Magnetic Materials* **2021**, *537*, 168159, doi:10.1016/j.jmmm.2021.168159.
48. Industrial Raspberry Pi - Revolution Pi |. Downloads | Industrial Raspberry Pi - Revolution Pi. Available online: https://revolutionpi.de/tutorials/downloads/ (accessed on 13 February 2023).
49. Mean Well NDR-240-24. Available online: https://www.meanwell-web.com/en-gb/ac-dc-single-output-industrial-din-rail-power-ndr--240--24 (accessed on 13 February 2023).
50. Temperaturtransmitter TMT128 für TC, fest eingestellt | Endress+Hauser. Available online: https://www.de.endress.com/de/messgeraete-fuer-die-prozesstechnik/temperaturmesstechnik-thermometer-transmitter/TC-Hutschienen-Temperatur-Transmitter-TMT128?t.tabId=product-downloads (accessed on 13 February 2023).
51. PicoScope oscilloscope software and PicoLog data logging software. Available online: https://www.picotech.com/downloads (accessed on 13 February 2023).

Anhang

Anhang A: Python-Code Datensynchronisation

Dem folgenden Anhang kann der Quellcode entnommen werden, der zum Synchronisieren der Messdaten der Anlage und des Picoscopes dient, wenn diese jeweils als csv-Datei vorliegen.

Anhang A1: Quellcode „synchronizeData.py

```
import numpy as np
import pandas as pd
import re
import synchronizeDataFunctions as synFunc

#-----------------------------------------------------------------------------------------------
# Author: Dominik Mueller
# Project: MatDatSys - Material Center Leoben
# Last modified: 06.02.2023
# Version: PyCharm Community Edition 2021.3
# Python Version: 3.9

# Description:
# This is the main script for synchronizing plant and yoke measurement data
# Each measurement has to be a .csv-file of a specific format. This format is that that one
# usually automatically gets when saving the measurement data as .csv.
# The start time of the yoke measurement relative to the start time of the plant measurement
# has to be known beforehand
#
# Needed format:
# plant data:
#-----------------------------------------------------------------------------------------------
## first line: Time;[0:12];[0:14];[0:15];[0:16];[0:21];[0:22]
## second line: time;DB100Kommunikation_IBA\Generator Leistung
# Soll;DB100Kommunikation_IBA\Generator Spannung;DB100Kommunikation_IBA\Generator
# Strom;DB100Kommunikation_IBA\Generator Frequenz;DB100Kommunikation_IBA\Generator Induktor
# Vorlauf Temperatur;DB100Kommunikation_IBA\Generator Induktor Rücklauf Temperatur;
## third line: sec;;;;;;;
## data: x0;x1; x2; x3; x4; x5; x6
#-----------------------------------------------------------------------------------------------
# yoke data:
#-----------------------------------------------------------------------------------------------
# Zeit;   Kanal A;  Kanal B;  Kanal C;  Kanal D
# (s);    (V);      (V);      (V);      (V)
#
# x0;     x1;       x2;       x3;       x4
#-----------------------------------------------------------------------------------------------

# the headers for the files to write
headerList = ["Zeit[s]", "Generator Leistung Soll[W]", "Generator Spannung[U]", "Generator
          Strom[A]", "Generator Frequenz[hz]", "Generator Induktor Vorlauf Temperatur[°C]",
          "Generator Induktor Rücklauf Temperatur[°C]", "Pico Zeit[s]", "Pico-Kanal A[V]",
          "Pico-Kanal B[V]", "Pico-Kanal C[V]", "Pico-Kanal D[V]"]

# get the filenamepart that contains the current date and create filenames
fileName = synFunc.getFileName()
fileNameSynchronized = fileName+"_synchronized.csv"
fileNameOnlySynchronized = fileName+"_onlySynchronized.csv"
fileNameInterpolated = fileName+"_interpolated.csv"

# manipulate the measurement csv-files for further use
dataframes = synFunc.prepareCsvFiles()

# Compute the start time of the yoke measurement in seconds via a User-Input
start_time = input("Enter the start time of the yoke measurement in the format:
                   hh:mm:ss:msmsmsms")
time_list = re.split(':', start_time)
start_time = float(time_list[0]) * 3600 + float(time_list[1]) * 60 +
float(time_list[2])+float(time_list[3])/10000.0

# extract the dataframes
```

```
plant_data = dataframes[0].dropna().to_numpy(dtype='float64')
yoke_data = dataframes[1].dropna().to_numpy(dtype='float64')

# Manipulate the yoke timestamp by adding the start time for the yoke measurement
yoke_data[:, 0] += start_time

# Get the dimensions of the matrices
nr_columns_plant = plant_data.shape[1]
nr rows plant = plant data.shape[0]
nr_columns_yoke = yoke_data.shape[1]
nr_rows_yoke = yoke_data.shape[0]


# Extract the yoke timestamp for further use
yoke time = yoke data[:, 0]

# Compute the start and end timestamp of the plant where the yoke data has to be synchronized
start_index = np.argmin(abs(plant_data[:, 0] - yoke_time[0]))
end_index = np.argmin(abs(plant_data[:, 0] - yoke_time[-1]))

# NaN-matrix to populate with the data
whole_data = np.empty((nr_rows_yoke + nr_rows_plant - end_index + start_index - 1,
          nr_columns_plant + nr_columns_yoke),'float64')
whole_data[:] = np.NaN

# The time before the pico measurement
whole_data[:start_index, :nr_columns_yoke] = plant_data[:start_index, :nr_columns_yoke]

# The time during the pico measurement
plant_time = plant_data[start_index:end_index + 1, 0]
whole_data[start_index:start_index + nr_rows_yoke, nr_columns_plant:] = yoke_data[:]

index_list = []
yoke_index_list = []

for counter, entry in enumerate(plant_data[start_index:end_index + 1, :]):
    r = abs(entry[0] - yoke_time)
    min_index = np.argmin(r)
    whole_data[start_index + min_index, :nr_columns_plant] = plant_data[start_index + counter,
          :]
    index_list.append(start_index + min_index)
    yoke_index_list.append(min_index)

# The time after the pico measurement
whole_data[nr_rows_yoke + start_index:, :nr_columns_plant] = plant_data[end_index + 1:, :]
only_synchronized_data = whole_data[~np.isnan(whole_data).any(axis=1), :]

# write the first two .csv-files
pd.DataFrame(whole_data).to_csv(path_or_buf=fileNameSynchronized, header=headerList,
          index=False, sep=';', encoding='utf-8')
pd.DataFrame(only_synchronized_data).to_csv(path_or_buf=fileNameOnlySynchronized,
          header=headerList, index=False, sep=';', encoding='utf-8')

# Perform the interpolation for the missing values in the whole dataset(whole_data):
for counter, index in enumerate(index_list):
    if counter == 0:
        continue
    else:
        x = [whole_data[index_list[counter - 1], 0], whole_data[index, 0]]
        y = [whole_data[index_list[counter - 1], 1:nr_columns_plant], whole_data[index,
          1:nr_columns_plant]]

        params = synFunc.interpolate_getParameter(x, y)

        coeff matrix = np.ones((index - index list[counter - 1] - 1, 2))
        coeff_matrix[:, 1] = yoke_data[yoke_index_list[counter - 1] + 1:
          yoke_index_list[counter], 0].ravel()

        whole_data[index_list[counter - 1] + 1:index, 1:nr_columns_plant] =
          np.round(np.matmul(coeff_matrix, params), decimals=2)
        whole data[index list[counter - 1] + 1:index, 0] = yoke data[yoke index list[counter -
          1] + 1:yoke_index_list[counter], 0]

# Write the interpolated data to a .csv-file
pd.DataFrame(whole_data).to_csv(path_or_buf=fileNameInterpolated, header=headerList,
          index=False, sep=';', encoding='utf-8')
```

Anhang A2: Quellcode „synchronizeDataFunctions.py

```
import pandas as pd
import tkinter as tk
from tkinter import filedialog
import numpy as np
import datetime

#-----------------------------------------------------------------------------------------------
# Author: Dominik Mueller
# Project: MatDatSys - Material Center Leoben
# Last modified: 06.02.2023
# Version: PyCharm Community Edition 2021.3
# Python Version: 3.9

# Description:
# This script contains the necessary functions to synchronize the plant and picoscope data
# from csv.-files
# created from the original measurement data
#
# getFileName():
    # return: string; a filenamepart for usage in "synchronizeData.py" consisting of a choosen
    # directory and the current date and time
    # format: {directory}/{day}.{month}.{year}_{hour}h{minutes}min{seconds}sec
#
# interpolate_getParameter(x, *y):
    # This is used for computing the parameters b1 and b2 for a polynomial of degree 1 in the
    # form: x*b1+b2=y
    # They are then used to compute the linearly interpolated measurement values of the plant
    # data between x1 and x2
    #
    # input:
    # x: float-array of the form [x1,x2] where x1 is the startpoint for the interpolation and
    # x2 is the endpoint
    # y: float-array fo the form [y11,y12; y21,y22;...] that contains the corresponding y
    # values to the x values.
    # Here you can put several pairs of y-values at once for every measured parameter at the
    # given timestamp
    #
    # return: 2 x [number of y entries] - matrix with every column containing the
    # corresponding b1 and b2 values for computing the interpolated function values

# prepareCsvFile():
    # This function takes the original plant data and yoke data as .csv-files and prepares it
    # for further use by unifying their format
    #
    # return: list [df_plant, df_yoke] with one panda.dataframe(df) per csv. file;
#-----------------------------------------------------------------------------------------------

# gives back a filename consisting of the chosen directory and the current date and time
def getFileName():
    # we don't want to show a window
    root = tk.Tk()
    root.withdraw()

    # get the directory
    directory = filedialog.askdirectory(title="Select a directory to store the synchronized
          files")

    # get the current date and time
    currentTime = datetime.datetime.now()
    year = currentTime.year
    month = currentTime.month
    day = currentTime.day
    hour = currentTime.hour
    minute = currentTime.minute
    second = currentTime.second

    fileName =
          "{directory}\{day}.{month}.{year}_{hour}h{minute}min{second}sec".format(day=day,
                    month=month,year=year, hour=hour, minute=minute,
                    second=second, directory=directory)
    return fileName

# Method to compute multiple parameters *b1 and *b2 for a polynomial of the form: b1*x+b2=y at
# once:
```

```
75  def interpolate_getParameter(x, *y):
76      y = np.asarray(y)
77      x_matrix = np.matrix([[1, x[0]], [1, x[1]]])
78      b = np.matrix.getI((x matrix.T * x matrix)) * x matrix.T * y
79      return b
80
81  # This function takes the plant and yoke data in the original format and manipulates it for
82  # further synchronization
83  # It returns a list containing a dataframe for each the plant and the yoke measurement
84  def prepareCsvFiles():
85      # we don't want to show a window
86      root = tk.Tk()
87      root.withdraw()
88
89      # choose the files
90      path_plant = filedialog.askopenfilename(title="Choose the PLANT DATA csv-file")
91      path_yoke = filedialog.askopenfilename(title="Choose the YOKE DATA csv-file")
92
93      # read them into dataframes
94      df_plant = pd.read_csv(path_plant, skiprows=1, delimiter=';', quotechar='"',
95            encoding='utf-8', on_bad_lines='skip')
96      df_plant.drop(df_plant.filter(regex="Unnamed"), axis=1, inplace=True)
97
98      df_yoke = pd.read_csv(path_yoke, delimiter=';', quotechar='"', encoding='utf-8',
99            on_bad_lines='skip')
100
101     # Give them consistent headers for the time column
102     df_plant.rename(columns={'time': 'Time'}, inplace=True)
103     df_yoke.rename(columns={'Zeit': 'Time'}, inplace=True)
104
105     # format them the same way
106     df_yoke.replace(',', '.', regex=True, inplace=True)
107
108     df_plant.drop(0, inplace=True)
109     df_plant.reset_index(inplace=True, drop=True)
110     df_plant = df_plant.apply(pd.to_numeric, errors='coerce')
111
112     df_yoke.drop(0, inplace=True)
113     df_yoke.reset_index(inplace=True, drop=True)
114     df_yoke = df_yoke.apply(pd.to_numeric, errors='coerce')
115
116     # save the files
117     # df_plant.to_csv('Anlagedaten.csv', sep=';', index=False)
118     # df_yoke.to_csv('Jochdaten.csv', sep=';', index=False)
119
120     return [df_plant, df_yoke
```

Anhang B: Python-Code Induktionsprüfstand

Dem nachfolgenden Anhang kann der Python-Code entnommen werden, der zum Erfassen und Speichern der Messdaten der Induktionsprüfanlage dient.

Anhang B1: Quellcode „memMap_Data,py“

```
# ----------------------------------------------------------------------------------------------------
# Author: Dominik Mueller
# Project: MatDatSys - Material Center Leoben
# Last modified: 06.02.2023
# Version: PyCharm Community Edition 2021.3
# Python Version: 3.9
#
# Description:
# get_data_plant(returnArray):
# This function is started as a child-process using multiprocessing by "startProcess.py"
# A finite state machine is used to read shared memory data from Codesys, which is then
# send to "startProcess.py" as a np.ndarray using a queue
## input:
# returnArray: multiprocessing.Queue()-object; used for returning the np.ndarray
## return:
# np.ndarray
# memory_to_struct(buffer, struct)
# This function copies a memory entry to a struct. The sizes have to match
## input:
# buffer: mmap.mmap
# struct: c.Structure
## return: c.Structure
# ----------------------------------------------------------------------------------------------------

def get_data_plant(returnArray):
    import mmap
    import posix_ipc as pos
    import ctypes as c
    import numpy as np

    # INITIALIZE VARIABLES
    # ------------------------------------------------------------------------------------------------
    initMemFlag = False  # boolean to check if "_CODESYS_TRIGGER_Plant" was successfully
                         # initialized
    initMemData = False  # boolean to check if "_CODESYS_MEMORY_Data" was successfully
                         # initialized
    initMemMeta = False  # boolean to check if "_CODESYS_MEMORY_Meta" was successfully
                         # initialized
    initMemFlagEnd = False  # boolean to check if "_CODESYS_TRIGGER_End" was successfully
                         # initialized
    state = 0  # initialize state
    nrValues = 1000  # number of entries in one struct in "_CODESYS_MEMORY_Data"
    dataStorage = list()  # list to store the PlantData-structs

    # header for the datasets:
    headerList = ['Time', 'Durchfluss Brause', 'Druck Brause', 'Druck Gas',
                  'Durchfluss Gasbrause 1',
                  'Druck Gasbrause 1', 'Durchfluss Gasbrause 2', 'Druck Gasbrause 2',
                  'Waermeregelung', 'T Kuehlung Zulauf', 'T Kuehlung Ruecklauf',
                  'T Abschreckmitteltank', 'T Brause', 'T Gas', 'TC1', 'TC2', 'TC3', 'TC4',
                  'TC5', 'TC6', 'TC7', 'TC8', 'TC9', 'TC10']

    # units for the 'headerList'
    unitList = ['s', 'L/min', 'bar', 'bar', 'L/min', 'bar', 'L/min', 'bar', '%', '°C', '°C',
                '°C', '°C', '°C', '°C', '°C', '°C', '°C', '°C', '°C', '°C', '°C', '°C']

    lenHeader = len(headerList)  # number of entries in the 'headerList'

    # Assign the names to the shared memories
    name_SM2_Plant = "_CODESYS_MEMORY_Data"
    name_SM1_Plant = "_CODESYS_TRIGGER_Plant"
    name_SM3_Plant = "_CODESYS_MEMORY_Meta"
    name_SM4_Plant = "_CODESYS_TRIGGER_End"
```

```
    # variables for printing (testing)
    k = 1
    m = 1

    # ------------------------------------------------------------------------------------------

    # DEFINE THE FUNCTION
    # ------------------------------------------------------------------------------------------
    def memory to struct(buffer, struct):
        newStruct = struct.from_buffer_copy(buffer)
        return newStruct

    # ------------------------------------------------------------------------------------------

    # CREATE THE C-TYPE STRUCTURES
    # ------------------------------------------------------------------------------------------
    # create a class for the meta data, has to be the same structure as the STRUCT in Codesys
    class MetaData(c.Structure):
        _fields_ = [("nrMes", c.c_int16),
                    ("nrSamples", c.c_int16),
                    ("fileName", c.c_char * 80),
                    ("freq", c.c_int16),
                    ("startTimePlant", c.c_uint64)]

    # create a class for the measurement data, has to be the same structure as the STRUCT in
# Codesys
    class PlantData(c.Structure):
        _fields_ = [("time", c.c_float * nrValues), ("qb", c.c_float * nrValues),
                    ('pb', c.c_float * nrValues), ('pg', c.c_float * nrValues),
                    ('qg1', c.c_float * nrValues), ('pg1', c.c_float * nrValues),
                    ('qg2', c.c_float * nrValues), ('pg2', c.c_float * nrValues),
                    ('wr', c.c_float * nrValues), ('tKzu', c.c_float * nrValues),
                    ('tKr', c.c_float * nrValues), ('tab', c.c_float * nrValues),
                    ('tb', c.c_float * nrValues), ('tg', c.c_float * nrValues),
                    ('TC1', c.c_float * nrValues), ('TC2', c.c_float * nrValues),
                    ('TC3', c.c_float * nrValues), ('TC4', c.c_float * nrValues),
                    ('TC5', c.c_float * nrValues), ('TC6', c.c_float * nrValues),
                    ('TC7', c.c_float * nrValues), ('TC8', c.c_float * nrValues),
                    ('TC9', c.c_float * nrValues), ('TC10', c.c_float * nrValues)]

    # ------------------------------------------------------------------------------------------
-

    while True:
# ------------------------------------------------------------------------------------------#
# ----------------------------------------STATE=0-------------------------------------------#
# ------------------------------------------------------------------------------------------#
        # connect to the shared memories
        if state == 0:
            while not initMemData and not initMemFlag and not initMemMeta and not
                    initMemFlagEnd:
                if not initMemData:
                    try:
                        SM2_Plant = pos.SharedMemory(name_SM2_Plant)  # SM for the data
                        initMemData = True
                    except pos.ExistentialError:
                        pass
                if not initMemFlag:
                    try:
                        SM1_Plant = pos.SharedMemory(name_SM1_Plant)  # SM for the trigger
                        initMemFlag = True
                    except pos.ExistentialError:
                        pass
                        pass
                if not initMemMeta:
                    try:
                        SM3_Plant = pos.SharedMemory(name_SM3_Plant)  # SM for the metadata
                        initMemMeta = True
                    except pos.ExistentialError:
                        pass
                if not initMemFlagEnd:
                    try:
                        SM4_Plant = pos.SharedMemory(name_SM4_Plant)  # SM for the end trigger
                        initMemFlagEnd = True
                    except pos.ExistentialError:
                        pass
```

```
            if initMemFlag and initMemData and initMemMeta and initMemFlagEnd:
                # change state
                state = 1
                initMemData = False
                initMemFlag = False
                initMemFlagEnd = False
                initMemMeta = False

                if m == 1:
                    print('PLANT: Successfully initialized')
                    m = 2

# ------------------------------------------------------------------------------------------
# ----------------------------------------STATE=1-------------------------------------------#
# ------------------------------------------------------------------------------------------#
        # wait for a trigger
        elif state == 1:
            # test
            if k == 1:
                print('PLANT: State 1: Waiting for trigger')
                k = 0

            # Trigger that indicates last datablock
            mapFlagEnd = mmap.mmap(SM4_Plant.fd, SM4_Plant.size)
            flagArrayEnd = mapFlagEnd.read()
            flagValueEnd = int.from_bytes(flagArrayEnd, "little")

            # Trigger that indicates that a datablock is ready
            mapFlag = mmap.mmap(SM1_Plant.fd, SM1_Plant.size)  # map the memory onto 'mapFlag'
            flagArray = mapFlag.read()  # read the bytes from the memory
            flagValue = int.from_bytes(flagArray, "little")  # convert the bytes to an integer

            # change state
            if flagValue == 1:
                state = 2
            elif flagValueEnd == 1:
                state = 3

# ------------------------------------------------------------------------------------------#
# ----------------------------------------STATE=2-------------------------------------------#
# ------------------------------------------------------------------------------------------#
        # if a block is ready (flagValue == 1) read the data from the shared memory
        # and set the trigger back to zero
        elif state == 2:
            # test
            print('PLANT: State 2: Get data')

            # map the data memory
            mapFile = mmap.mmap(SM2_Plant.fd, SM2_Plant.size)

            # test
            print('PLANT: State 2: Reset flag')

            # reset flag
            newFlag = 0
            newFlag = newFlag.to_bytes(1, "little")
            mapFlag.seek(0)
            mapFlag.write(newFlag)
            mapFlag.close()

            # data is an object of type "PlantData"
            data = memory_to_struct(buffer=mapFile, struct=PlantData)
            dataStorage.append(data)

            # close the mapped file
            mapFile.close()

            # change state
            state = 1

# ------------------------------------------------------------------------------------------#
# ----------------------------------------STATE=3-------------------------------------------#
# ------------------------------------------------------------------------------------------#
        # last measurement (flagValue == 2) read the last data and set the trigger back to
        # zero
        elif state == 3:
            #test
```

```
            print('PLANT: State 3: get last data')

            # set the flag back to zero
            newFlagEnd = 0
            newFlagEnd = newFlagEnd.to_bytes(1, "little")
            mapFlagEnd.seek(0)
            mapFlagEnd.write(newFlagEnd)
            mapFlagEnd.close()  # close the mapped file

            # read the last data packages from the measurement
            mapFile = mmap.mmap(SM2_Plant.fd, SM2_Plant.size)  # map the data memory
            mapMeta = mmap.mmap(SM3_Plant.fd, SM3_Plant.size)  # map the meta data memory

            # read the data and metadata into a c-type struct
            metaData = memory_to_struct(buffer=mapMeta, struct=MetaData)
            data = memory_to_struct(buffer=mapFile, struct=PlantData)

            dataStorage.append(data)

            # close the mmap-files
            mapFile.close()
            mapMeta.close()

            # change state
            state = 10

# ------------------------------------------------------------------------------------------#
# ----------------------------------------STATE=4-------------------------------------------#
# ------------------------------------------------------------------------------------------#
        # clean everything up
        elif state == 4:
            # test
            print("PLANT: State 5: Clean everything up")

            # close all shared memories
            SM1_Plant.close_fd()
            SM2_Plant.close_fd()
            SM4_Plant.close_fd()
            SM3_Plant.close_fd()

            # delete created objects
            del dataStorage
            del data
            del metaData

            # put an end inidicator to the queue
            returnArray.put('End')

            # change state
            state = 0

# ------------------------------------------------------------------------------------------#
# ----------------------------------------STATE=10------------------------------------------#
# ------------------------------------------------------------------------------------------#
        # extract the data from the dataBlocks and write them to a numpy.ndarray
        elif state == 10:
            # test
            print("PLANT: State 10: putting file to queue")

            nrSamples = metaData.nrSamples  # total number of dataBlocks caught
            nrMes = metaData.nrMes  # numer of values in the current dataBlock

            row = 0  # determines at which datablock in dataStorage we are currently at when
                     # writing it to an ndarray

            # compute the size of the necessary array
            sizeArray = nrSamples * nrValues - nrValues + nrMes  # number of rows of dataArray
            dataArray = np.ndarray(shape=(sizeArray, lenHeader),
                                   dtype=float)  # array to store the whole data

            # loop over all datablocks to fill the dataArray with values
            for dataBlock in dataStorage:

                field_list = [getattr(dataBlock, f[0]) for f in
                              dataBlock._fields_]  # get each field in the current dataBlock
                column = 0  # number of column in the dataArray we are currently at
```

```
    for i in range(len(field_list)):
        if dataBlock == dataStorage[-1]:  # for the last block captured
            lower_bound = row * nrValues  # start position in dataArray
            upper_bound = row * nrValues + nrMes  # end position in dataArray
            dataArray[lower_bound:upper_bound, column] =
                 field_list[column][:nrMes]
            if column == len(field_list):
                # print('total number of meassurement: ', sizeArray)
                break
        else:
            lower_bound = row * nrValues  # start position in dataArray
            upper_bound = (row + 1) * nrValues  # end position in dataArray
            dataArray[lower_bound:upper_bound, column] = field_list[column][:]

        column += 1

    row += 1

# round the float values in the array to four decimals and put it to the queue
dataArray = np.round(dataArray, decimals=4)
returnArray.put(dataArray)

# change state
state = 4
```

Anhang B2: Quellcode „getData_Pico.py“

```
# ------------------------------------------------------------------------------------------
# Author: Dominik Mueller
# Project: MatDatSys - Material Center Leoben
# Last modified: 06.02.2023
# Version: PyCharm Community Edition 2021.3
# Python Version: 3.9
#
# Description:
# get_data_pico(queuePico, numberOfMes, maxSamples, timebase):
    # This function is started as a child-process using multiprocessing by "startProcess.py"
    # A finite state machine is used to start a picoscope measurement, indicated by Codesys.
    # The measurement data from the picoscope is then captured and pushed to "startProcess.py"
    # as a np.ndarray using a queue.
    ## input:
        # queuePico: multiprocessing.Queue(); used for returning the np.ndarray
        # numberOfMes: int; indicates how many single picoscope measurements should be made
        # maxSamples: int; number of datapoints captured during one picoscope measurement
        # timebase: int; the timebase for the picoscope computed in "startProcess.py"
    ## return:
        # list of np.ndarrays
# ------------------------------------------------------------------------------------------

def get_data_pico(queuePico, numberOfMes, maxSamples, timebase):
    import posix_ipc as pos
    import mmap
    import sys
    from ctypes import Structure, c_float, c_char, c_int16, c_int32, byref, POINTER
    from picosdk.ps5000a import ps5000a as ps
    from picosdk.functions import adc2mV, assert_pico_ok, mV2adc
    import numpy as np
    from time import time_ns

    name_SM1_Pico = '_CODESYS_TRIGGER_Pico'  # shared memory for the trigger, must have the
                                             # same name as in codesys
    initFlagMem = False                    # indicator if the SM was initialized successfully

    state = 100     # set the state = 100 to set up the pico in the beginning
    initMes = True  # indicator that a new measurement was started
    counter = 0     # the current number of single picoscope measurements

    bufferAMax = (c_int16 * maxSamples)()  # buffer were the data of one measurment will be
                                           # stored

    # just here to print out stuff at the right time for testing purposes
    k = 1  # not relevant - just for testing
    m = 1  # not relevant - just for testing

    while True:
#-------------------------------------------------------------------------------------------#
#-----------------------------------------STATE=100-----------------------------------------#
#-------------------------------------------------------------------------------------------#
        if state == 100:
            # test
            print('YOKE: State 100: Yoke process started')

            # CONNECT TO THE PICOSCOPE
            # ------------------------------------------------------------------------------
            # create a handle and a status-dict for the picoscope
            chandle = c_int16()  # this is the handel to communicate with the picoscope
            status = {}          # dict to store relevant information

            # set the wanted resolution. The resolution is here set to 12 Bit -- currently
            # hardcoded for testing purposes
            resolution = ps.PS5000A_DEVICE_RESOLUTION[
                "PS5000A_DR_12BIT"]  # the resolution of the picoscope

            # Open the PicoScope
            status["openunit"] = ps.ps5000aOpenUnit(byref(chandle), None, resolution)
            assert_pico_ok(status["openunit"])  # check if connection was successfull
            # ------------------------------------------------------------------------------

            # SET UP THE CHANNELS AND THE PARAMETERS FOR THE MEASUREMENT
            # ------------------------------------------------------------------------------
            # This is hard coded at the moment for testing purposes
```

```
            # Set up channel A
            channel = ps.PS5000A_CHANNEL["PS5000A_CHANNEL_A"]    # select channel A
            coupling_type = ps.PS5000A_COUPLING["PS5000A_DC"]    # set the coupling type to DC
            chARange = ps.PS5000A RANGE["PS5000A 20V"]           # set the range for the input
                                                                    # channel

            status["setChA"] = ps.ps5000aSetChannel(chandle, channel, 1, coupling_type,
                 chARange, 0)      # activate Channel A
            assert pico ok(status["setChA"])   # check if successful

            # get the time interval in ns
            timeIntervalns = c_int32()              # time interval in ns
            returnedMaxSamples = c_int32()          # number of samples available; returned by
                                                        # ps5000aGetTimebase()
            status["getTimebase2"] = ps.ps5000aGetTimebase(chandle, timebase, maxSamples,
byref(timeIntervalns), byref(returnedMaxSamples), 0)
            assert_pico_ok(status["getTimebase2"])      # check if successful

            # Set data buffer location for data collection from channel A
            source = ps.PS5000A_CHANNEL["PS5000A_CHANNEL_A"]   # set the source for the Buffer
            status["setDataBuffersA"] = ps.ps5000aSetDataBuffer(chandle, source,
                   byref(bufferAMax), maxSamples, 0,
                   ps.PS5000A_RATIO_MODE['PS5000A_RATIO_MODE_NONE'])
            assert_pico_ok(status["setDataBuffersA"])          # check if successful

            # switch off all other channels for now -- for testing purposes
            ps.ps5000aSetChannel(chandle, ps.PS5000A_CHANNEL["PS5000A_CHANNEL_B"], 0,
                       coupling_type, chARange, 0)
            ps.ps5000aSetChannel(chandle, ps.PS5000A_CHANNEL["PS5000A_CHANNEL_C"], 0,
                       coupling_type, chARange, 0)
            ps.ps5000aSetChannel(chandle, ps.PS5000A_CHANNEL["PS5000A_CHANNEL_D"], 0,
                       coupling_type, chARange, 0)
            # ----------------------------------------------------------------------------


            # INITIALIZE VARIABLES FOR THE PICOSCOPE MEASUREMENT
            # ----------------------------------------------------------------------------
            maxADC = c_int16()  # maximum ADC value
            ps.ps5000aMaximumValue(chandle, byref(
                maxADC))  # get the maximum ADC value for the chosen resolution and write it
                            # to maxADC

            # create converted type maxSamples for later use
            cmaxSamples = c_int32(maxSamples)    # number of points captured in one measurement
                                                 # as a c_type object
            overflow = c_int16()                 # indicates if over-voltage has occurred

            # create an array to put the measured data to then push it to the parent script
            # channels + 2 columns for: measurement data Channel A; corresponding time; start
            # time of each block
            # picoDataArray[0,0] will be the time since epoch in ns
            nrChannelsActive = 1     # number of active channels, currently hard coded to 1
                                        # since only Channel A is active
            picoDataArray = np.ndarray(shape=(maxSamples, nrChannelsActive + 2), dtype=float)
            # ----------------------------------------------------------------------------

            # change state
            state = 0

#-------------------------------------------------------------------------------------------#
#-----------------------------------------STATE=0-------------------------------------------#
#-------------------------------------------------------------------------------------------#
        # access the shared memory and change state when all are successfully initialized
        elif state == 0:
            # test
            if m == 1:
                print("YOKE: State 0: Memory initialization")
                m = 2

            # initialize SM
            if not initFlagMem:
                try:
                    SM1_Pico = pos.SharedMemory(name_SM1_Pico)
                    initFlagMem = True
                    # change state
                    state = 1
                except pos.ExistentialError:
```

```
                    pass

#-------------------------------------------------------------------------------------------#
#-----------------------------------------STATE=1-------------------------------------------#
#-------------------------------------------------------------------------------------------#
        # Wait for a trigger from codesys
        elif state == 1:
            # test
            if k == 1:
                print("YOKE: State 1: Waiting for trigger")
                k = 2

            # read SM1_Pico
            mapFlag = mmap.mmap(SM1_Pico.fd, SM1_Pico.size)
            flagArray = mapFlag.read()
            flagValue = int.from_bytes(flagArray, "little")

            if flagValue == 1:
                if initMes:
                    counter = 1
                    initMes = False
                # change state
                state = 2

#-------------------------------------------------------------------------------------------#
#-----------------------------------------STATE=2-------------------------------------------#
#-------------------------------------------------------------------------------------------#
        # capture a block and put it to the queue
        elif state == 2:
            # test
            print("YOKE: State 2: Capture a block")

            startTime_ns = float(time_ns())  # save the start time of the picoscope
                  # measurement [ns]

            # starts the capturing of a datablock
            status["runBlock"] = ps.ps5000aRunBlock(chandle, 0, maxSamples, timebase, None, 0,
                                                    None, None)
            assert_pico_ok(status["runBlock"])  # check if successful

            # Check for data collection to finish using ps5000aIsReady
            # if 'ready' != 0 then the data collection has finished
            ready = c_int16(0)  # variable for checking if datablock is ready
            check = c_int16(0)  # variable for checking if datablock is ready

            # wait for a block to be ready
            while ready.value == check.value:
                ps.ps5000aIsReady(chandle, byref(ready))  # ready will be overwritten as soon
                   # as a datablock is ready

            # test
            print('YOKE: Block ready')

            # Get a data block
            ps.ps5000aGetValues(chandle, 0, byref(cmaxSamples), 0, 0, 0,
                                byref(overflow))

            # convert ADC counts data to mV
            adc2mVChAMax = adc2mV(bufferAMax, chARange, maxADC)  # these are the final
                      # measurement values

            # Create time data
            time = np.linspace(0, (cmaxSamples.value - 1) * timeIntervalns.value,
                 cmaxSamples.value)

            # fill the data array - has to be unpacked in 'startProcess.py'
            picoDataArray[0, 0] = startTime_ns                # starting time timestamp in ns
            picoDataArray[:cmaxSamples.value, 1] = time[:]    # timedata for measurement
            picoDataArray[:cmaxSamples.value, 2] = adc2mVChAMax[:cmaxSamples.value] # data
                          # from measurement

            # put the data-array to the queue
            queuePico.put(picoDataArray)

            # if the number of wanted measurement is reached go to state = 4 to finalize the
            # measurement
            if counter == numberOfMes:
```

```
                    # test
                    print('YOKE: number of measurements reached')

                    # change state
                    state = 4
                else:
                    counter += 1
                    # change state
                    state = 3

    #-------------------------------------------------------------------------------------------#
    #----------------------------------------STATE=3--------------------------------------------#
    #-------------------------------------------------------------------------------------------#
            # reset the trigger
            elif state == 3:
                # test
                print('YOKE: State 3: Reset flag')

                newFlag = 0
                newFlag = newFlag.to_bytes(1, "little")
                mapFlag.seek(0)
                mapFlag.write(newFlag)
                mapFlag.close()
                k = 1
                # change state
                state = 1

    #-------------------------------------------------------------------------------------------#
    #----------------------------------------STATE=4--------------------------------------------#
    #-------------------------------------------------------------------------------------------#
            # put an end indicator to the queue
            elif state == 4:
                # clean everything up
                newFlag = 0
                newFlag = newFlag.to_bytes(1, "little")
                mapFlag.seek(0)
                mapFlag.write(newFlag)
                mapFlag.close()
                SM1_Pico.close_fd()

                # Stop the picoscope
                status["stop"] = ps.ps5000aStop(chandle)
                assert_pico_ok(status["stop"])
                status["closed"] = ps.ps5000aCloseUnit(chandle)
                assert_pico_ok(status["closed"])

                # send end indicator to "startProcess.py"
                queuePico.put('End')
                # change state
                state = 0
                # End the process. It doesn't matter if it's done here or in "startProcess.py"
                # If it is ended here we do not have to worry about further triggers coming from
                # Codesys
                sys.exit()
```

Anhang B3: Quellcode „startProcess.py“

```
# ------------------------------------------------------------------------------------------------
# Author: Dominik Mueller
# Project: MatDatSys - Material Center Leoben
# Last modified: 06.02.2023
# Version: PyCharm Community Edition 2021.3
# Python Version: 3.9
#
# Description:
# This program is used to coordinate the start of "getData_Pico.py" and "memMap_Data.py"
# according to the measurement initialized via Codesys. The received measurement data is saved
# in hdf5-format.after the capturing of all wanted data has finished.
#
# memory_to_struct(buffer, struct)
# This function copies a memory entry to a struct. The sizes have to match
## input:
# buffer: mmap.mmap
# struct: c.Structure
## return: c.Structure
# ------------------------------------------------------------------------------------------------

if __name__ == '__main__':

    init = True  # boolean to make sure all the initialisations at the start of the loop are
                 # only done once

    while True:

        # Iinitialize everything in preperatation for a new measurement
        if init:
            from multiprocessing import Process, Queue
            import queue
            from numpy import ndarray
            from memMap_Data2 import get_data_plant
            from getData_Pico2 import get_data_pico
            import posix_ipc as pos
            from mmap import mmap
            from ctypes import Structure, c_float, c_char, c_int16, c_ulong, c_bool, c_uint64
            import h5py as h5
            from datetime import datetime as dt

            # SET THE FUNCTIONS FOR MULTIPROCESSING
            # ------------------------------------------------------------------------------------
            fun_plant = get_data_plant  # function from memMap_Data.py; used for
                                        # multiprocessing
            fun_pico = get_data_pico  # function from getData_Pico.py; used for
                                      # multiprocessing
            nrValues = 1000  # number of values captured in each datablock from the
                             # plant
     # ------------------------------------------------------------------------------------------

            # DEFINE THE FUNCTION
            # ------------------------------------------------------------------------------------
            def memory_to_struct(buffer, struct):
                newStruct = struct.from_buffer_copy(buffer)
                return newStruct
            # ------------------------------------------------------------------------------------

            # CREATE THE C-TYPE STRUCTURES
            # ------------------------------------------------------------------------------------
            # create a class for the meta data, has to be the same structure as the STRUCT in
            # Codesys
            class MetaData(Structure):
                _fields_ = [("nrMes", c_int16),
                            ("nrSamples", c_int16),
                            ("fileName", c_char * 80),
                            ("freq", c_int16),
                            ("startTimePlant", c_uint64)]


            # create a class for the measurement data, has to be the same structure as the
            # STRUCT in Codesys
            class PlantData(Structure):
                _fields_ = [("time", c_float * nrValues), ("qb", c_float * nrValues),
                            ('pb', c_float * nrValues), ('pg', c_float * nrValues),
```

```
                                ('qg1', c_float * nrValues), ('pg1', c_float * nrValues),
                                ('qg2', c_float * nrValues), ('pg2', c_float * nrValues),
                                ('wr', c_float * nrValues), ('tKzu', c_float * nrValues),
                                ('tKr', c_float * nrValues), ('tab', c_float * nrValues),
                                ('tb', c_float * nrValues), ('tg', c_float * nrValues),
                                ('TC1', c_float * nrValues), ('TC2', c_float * nrValues),
                                ('TC3', c_float * nrValues), ('TC4', c_float * nrValues),
                                ('TC5', c_float * nrValues), ('TC6', c_float * nrValues),
                                ('TC7', c_float * nrValues), ('TC8', c_float * nrValues),
                                ('TC9', c_float * nrValues), ('TC10', c_float * nrValues)]


                # create the class for the pico setup data
                class PicoSetupData(Structure):
                     fields  = [("f RevPi", c float),
                                ("f_SG", c_float),
                                ("SF_SG", c_char * 10),
                                ("A_SG", c_float),
                                ("N_samples_pico", c_ulong),
                                ("N_p", c_int16),
                                ("N_EM", c_int16),
                                ("deltaT_EM", c_float),
                                ("f_pico", c_float),
                                ("t_pico", c_float),
                                ("setupTrigger", c_bool)]
                # ------------------------------------------------------------------------------

                # these are the names for the groups in "plant data" in the hdf5-file
                headerList = ['Time', 'Durchfluss Brause', 'Druck Brause', 'Druck Gas',
                              'Durchfluss Gasbrause 1', 'Druck Gasbrause 1',
                              'Durchfluss Gasbrause 2', 'Druck Gasbrause 2',
                             'T Kuehlung Zulauf', 'T Kuehlung Ruecklauf',
                             'T Abschreckmitteltank', 'T Brause', 'T Gas',
                             'TC1', 'TC2', 'TC3', 'TC4', 'TC5', 'TC6',
                             'TC7', 'TC8', 'TC9', 'TC10']

                # units for each entry in the headerList in the according order
                unitList = ['s', 'L/min', 'bar', 'bar', 'L/min', 'bar', 'L/min', 'bar', '%', '°C',
                            '°C', '°C', '°C', '°C', '°C', '°C', '°C', '°C', '°C', '°C', '°C',
                            '°C', '°C', '°C']

                # INITIALIZE VARIABLES
                # ------------------------------------------------------------------------------
                # the names for the shared memories. Must be the same as in codesys
                name_SM_Setup = '_CODESYS_MEMORY_Parent'  # SM-Trigger
                name_SM3_Plant_parent = "_CODESYS_MEMORY_Meta"  # SM-Metadata
                name_SM2_Pico = '_CODESYS_MEMORY_PicoSetup'  # SM-PicoSetup

                # bolleans to check if the SMs are initialized
                initShmTrigger = False  # for SM_Setup
                initShmMeta = False  # for SM3_Plant_parent
                initShmPicoSetup = False  # for SM2_Pico

                startM = True  # if a new measurement is started start the child processes,
                               # then FALSE not to start them again and again every loop
                init = False  # after a measurement everything is reset, init == TRUE at the
                              # end of a measurement to initialize everything again
                setupInit = True  # in the first loop make sure the picoscope is set up with
                                  # default values

                # Variables to print out messages for testing purposes
                mainPrint = 1
                subPrint = 1

                # set the starting state
                state = 0
            # ------------------------------------------------------------------------------

    # ------------------------------------------------------------------------------------------#
    # ----------------------------------------STATE=100-----------------------------------------#
    # ------------------------------------------------------------------------------------------#
            # compute the new measurement setting for the picoscope if a change was detected
            if state == 100:
                # test
                print('MAIN: State 100: --Pico Setup--')

                PicoSetupNew = PicoSetup  # Copy the current PicoSetup-struct
```

```
            PicoSetupNew.setupTrigger = False  # set the trigger in the copied struct = false
            mapPicoSetup.write(PicoSetupNew)  # write the new struct to the SM

            N_p = PicoSetup.N_p  # number of periods measured with pico
            f_SG = PicoSetup.f_SG  # frequency of signal generator
            f_pico = PicoSetup.f_pico  # sampling rate Pico (N_samples_pico*f_SG)
            N_samples_pico = PicoSetup.N_samples_pico  # number of datapoints captured per
                                                       # period of signal generator
            N_EM = PicoSetup.N_EM  # total number of measurements done
                                   # via pico

            totalPointsCaptured = N_samples_pico * N_p  # number of datapoints captured with
                                                # the picoscope during a single measurement

            # since the picoscope doesn't take a measurement frequency as parameter a timebase
            # has to be computed instead which is a number that stands for the time between
            # two captured datapoints.
            # The formula for this can be found in the picoscope programmers guide for the
            # 5000a-series.
            # This is the timebase for the picoscope set to 12-Bit resolution
            # 1/f_pico: time between the capturing of two datapoints
            # maxSamples, N_EM and timebase will be forwarded to getData_pico.py
            timebase = int(((1 / f_pico * 62500000) + 3))

            preTriggerSamples = 0
            postTriggerSamples = totalPointsCaptured
            maxSamples = preTriggerSamples + postTriggerSamples  # this is basically the same
            # as totalPointsCaptured here - just wanted to have the syntax
            # according to the example for picoSDK at github

            # change state
            state = 1

# ----------------------------------------------------------------------------------------#
# ----------------------------------------STATE=0-----------------------------------------#
# ----------------------------------------------------------------------------------------#
        # initialize the shared memories
        elif state == 0:
            # test
            if mainPrint == 1:
                print('MAIN: Initializing')
                mainPrint = 2

            # shared memory to start the wanted measurement
            if not initShmTrigger:
                try:
                    SM_Setup = pos.SharedMemory(name_SM_Setup)
                    initShmTrigger = True
                except pos.ExistentialError:
                    pass
            # shared memory to transfer the meta-data for the plant measurement
            if not initShmMeta:
                try:
                    SM3_Plant_parent = pos.SharedMemory(name_SM3_Plant_parent)
                    initShmMeta = True
                except pos.ExistentialError:
                    pass
            # shared memory to transfer the picoscope settings
            if not initShmPicoSetup:
                try:
                    SM2_Pico = pos.SharedMemory(name_SM2_Pico)
                    initMetaMem = True
                    # change state
                    state = 1
                except pos.ExistentialError:
                    pass
            if initShmMeta and initShmTrigger and initShmPicoSetup:
                # change state
                state = 1

# ----------------------------------------------------------------------------------------#
# ----------------------------------------STATE=1-----------------------------------------#
# ----------------------------------------------------------------------------------------#
        # wait for trigger
        elif state == 1:
            # test
            if mainPrint == 2:
```

```
                print('MAIN: state 1 - waiting for trigger')
                mainPrint = 3

            # read the shared memory for pico setup data
            mapPicoSetup = mmap(SM2_Pico.fd, SM2_Pico.size)
            PicoSetup = memory_to_struct(buffer=mapPicoSetup, struct=PicoSetupData)
            mapPicoSetup.close

            # read the shared memory for starting the subprocesses
            mapFlag = mmap(SM_Setup.fd, SM_Setup.size)
            flagB = mapFlag.read()
            flagValue = int.from_bytes(flagB, "little")

            # if the pico-settings were changed by the user, or it's the first cycle then set
            # up the pico
            if PicoSetup.setupTrigger or setupInit:
                setupInit = False  # set to false after the first cycle

                # change state
                state = 100

            # start yoke and plant measurement scripts
            elif flagValue == 1:
                # test
                print("MAIN: Start both programs")

                counterMax = PicoSetup.N_EM  # number of single measurements done via pico

                # set the flags back to 0
                newFlag = 0
                newFlag = newFlag.to_bytes(1, "little")
                mapFlag.seek(0)
                mapFlag.write(newFlag)
                mapFlag.close()

                startTime = dt.now().strftime("%H{sym}%M{sym}%S").format(sym=':')  # start
                   # time for file name

                # change state
                state = 10

            # start only plant measurement script
            elif flagValue == 2:
                # test
                print("MAIN: Start only plant program")

                # set the flags back to 0
                newFlag = 0
                newFlag = newFlag.to_bytes(1, "little")
                mapFlag.seek(0)
                mapFlag.write(newFlag)
                mapFlag.close()

                startTime = dt.now().strftime("%H{sym}%M{sym}%S").format(sym=':')  # start
                   # time for file name

                # change state
                state = 20

            # start only yoke measurement script:
            elif flagValue == 3:
                # test
                print('MAIN: Only yoke measurement')

                counterMax = PicoSetup.N EM  # number of single measurements done via pico

                # set the flags back to 0
                newFlag = 0
                newFlag = newFlag.to_bytes(1, "little")
                mapFlag.seek(0)
                mapFlag.write(newFlag)
                mapFlag.close()

                # change state
                state = 30

# ------------------------------------------------------------------------------------------#
```

```
# ----------------------------------------STATE=10----------------------------------------#
# ----------------------------------------------------------------------------------------#
        # start both measurements
        # Here - in substate 12 - there is the problem that there is no constant output from
        # the picoscope yet. So the computation of the picoscope data is not yet implemented,
        # but it can be done in the same way es with the plant data
        elif state == 10:

            # set up the queues and processes and start them
            if startM:
                startM = False  # switch to the sub-FSM after the initialization of the
                # measurement setup
                # create a communication queue for both measurement and start the scripts as
                # subprocesses
                q1 = Queue()
                q2 = Queue()

                # create both processes
                p1 = Process(target=fun_plant, args=(q1,))
                p2 = Process(target=fun_pico, args=(q2, N_EM, maxSamples, timebase))

                # start both processes
                p1.start()
                p2.start()

                # create a list to store the yoke measurement-arrays in
                yokeMesList = list()

                # change substate
                subState = 11

                # booleans to check which measurement has finished
                yokeFinished = False  # pico measurement indicator
                plantFinished = False  # plant measurement indicator

        # ----------------------------------------------------------------------------------#
        # ----------------------------------SUBSTATE=11-------------------------------------#
        # ----------------------------------------------------------------------------------#
            # wait for queues to give back data
            if subState == 11:

                # check the queue of memMapData.py
                try:
                    plantOut = q1.get_nowait()  # Queue-output

                    if isinstance(plantOut, ndarray):  # np.ndarray from queue means we got
                                                       # the data
                        plantArray = plantOut
                        # test
                        print('MAIN: Got data from plant')
                    elif plantOut == 'End':  # indicates the end of the measurement
                        plantFinished = True
                        # test
                        print('MAIN: Got end indicator from plant')
                except queue.Empty:
                    pass

                # check the queue of getData_Pico.py
                try:
                    yokeMesData = q2.get_nowait()  # Queue-output

                    if not yokeMesData == 'End':  # as long as we get no end indicator
                                                  # append the data to a list
                        yokeMesList.append(yokeMesData)
                    elif yokeMesData == 'End':  # indicates the end of the measurement
                        yokeFinished = True
                        # test
                        print('MAIN: Got end indicator from pico')
                except queue.Empty:
                    pass

                if yokeFinished and plantFinished:  # 'End' from both measurements
                    # test
                    print('MAIN: Both measurement are finished')
                    # change substate
                    subState = 12
```

```
383             # ------------------------------------------------------------------------------------#
384             # ----------------------------------SUBSTATE=12----------------------------------------#
385             # ------------------------------------------------------------------------------------#
386                 # write the data to a file
387                 elif subState == 12:
388                     # read the metadata into a c-type struct
389                     mapMeta = mmap(SM3_Plant_parent.fd, SM3_Plant_parent.size)  # meta data
390                     metaData = memory_to_struct(buffer=mapMeta, struct=MetaData)
391                     mapMeta.close()
392 
393                     # extract the meta data from the struct
394                     filename = metaData.fileName.decode("utf-8")  # filename
395                     nrSamples = metaData.nrSamples  # total number of dataBlocks caught
396                     nrMes = metaData.nrMes  # numer of values in the current dataBlock
397                     freq = metaData.freq  # frequency of the measurement [Hz]
398                     startTime_ns = metaData.startTimePlant  # start time of data logging [ns]
399 
400                     # create a file and write the plant data to it
401                     file = h5.File(filename, 'w')
402                     nameDataSet = "Induction Plant Data"  # name of the plant data group
403                     nameDataSetPico = "Yoke Data"  # name of the yoke data group
404                     for index, name in enumerate(headerList):
405                         if unitList[index] == 's':
406                             grp = file.create_dataset(nameDataSet + '/' + 'Zeit' + '/' + name,
407                                 data=plantArray[:, index])
408                             grp.attrs['unit'] = unitList[index]
409                         elif unitList[index] == 'L/min':
410                             grp = file.create_dataset(nameDataSet + '/' + 'Durchfluss' + '/' +
411                                 name, data=plantArray[:, index])
412                             grp.attrs['unit'] = unitList[index]
413                         elif unitList[index] == 'bar':
414                             grp = file.create_dataset(nameDataSet + '/' + 'Druck' + '/' + name,
415                                 data=plantArray[:, index])
416                             grp.attrs['unit'] = unitList[index]
417                         elif unitList[index] == '°C':
418                             grp = file.create_dataset(nameDataSet + '/' + 'Temperatur' + '/' +
419                                 name, data=plantArray[:, index])
420                             grp.attrs['unit'] = unitList[index]
421                         elif unitList[index] == '%':
422                             grp = file.create_dataset(nameDataSet + '/' + 'Andere' + '/' + name,
423                                 data=plantArray[:, index])
424                             grp.attrs['unit'] = unitList[index]
425 
426                     file.get(nameDataSet).attrs['Creation date'] = str(
427                         dt.now().strftime("%d{sym}%m{sym}%Y").format(sym='.'))
428                     file.get(nameDataSet).attrs['Start time'] = startTime
429                     file.get(nameDataSet).attrs['End time'] =
430                        str(dt.now().strftime("%H{sym}%M{sym}%S").format(sym=':'))
431                     file.get(nameDataSet).attrs['Frequency'] = "{f} [Hz]".format(f=freq)
432 
433                     # write the yoke data to the file
434                     for index, data in enumerate(yokeMesList):
435                     # data is an array: data[0, 0] = time of the measurement in ns since epoch
436                         startTimeYoke = (data[0, 0] - float(startTime_ns))/1000000000.0 # relative
437                                                                                 # start time [s]
438                         grp = file.create_group(nameDataSetPico + '/' +
439                                 'Messung {i}'.format(i=index+1))
440                         grp.create_dataset('Zeit', data=data[:, 1]/1000000000.0)
441                         grp.create_dataset('Channel A', data=data[:, 2])
442                         grp.attrs['Startzeit relativ zur Anlage [s]'] =
443                                 ("{:10.6f}".format(startTimeYoke))
444                         file.get(nameDataSetPico).attrs['Frequenz Pico [Hz]:'] = f_pico
445                         file.get(nameDataSetPico).attrs['Frequenz SG [Hz]'] = f_SG
446                         file.get(nameDataSetPico).attrs['Anzahl gemessener Perioden (SG)'] = N_p
447                         file.get(nameDataSetPico).attrs['Anzahl Einzelmessungen'] = N_EM
448 
449                     file.close()
450 
451                     #test
452                     print('MAIN: !!Writing successfull!!')
453 
454                     # change substate
455                     subState = 13
456 
457             # ------------------------------------------------------------------------------------#
458             # ----------------------------------SUBSTATE=13----------------------------------------#
459             # ------------------------------------------------------------------------------------#
```

```
460                 # clean everything up
461                 elif subState == 13:
462                     # send codesys the signal that the measurement has finished
463                     mapFlag = mmap(SM_Setup.fd, SM_Setup.size)
464                     newFlag = 4
465                     newFlag = newFlag.to_bytes(1, "little")
466                     mapFlag.seek(0)
467                     mapFlag.write(newFlag)
468                     mapFlag.close()
469 
470                     # close the shared memories
471                     SM_Setup.close_fd()
472                     SM2_Pico.close_fd()
473 
474                     # end the child processes
475                     p1.kill()
476                     p2.kill()
477                     q1.close()
478                     q2.close()
479 
480                     # close the last shared memory
481                     SM3_Plant_parent.close_fd()
482 
483                     # delete all created objects and free the memory
484                     for element in dir():
485                         if element[0:2] != "__":
486                             del globals()[element]
487 
488                     # prepare for a new measurement
489                     init = True
490 
491                     # change state
492                     state = 0
493 
494     # ------------------------------------------------------------------------------------------#
495     # ----------------------------------------STATE=20------------------------------------------#
496     # ------------------------------------------------------------------------------------------#
497             # start only the plant measurement
498             elif state == 20:
499                 if startM:
500                     startM = False  # start the subprocess just once
501                     q1 = Queue()
502                     p1 = Process(target=fun_plant, args=(q1,))
503                     p1.start()
504                     subState = 21
505 
506             # ------------------------------------------------------------------------------------#
507             # ----------------------------------SUBSTATE=21---------------------------------------#
508             # ------------------------------------------------------------------------------------#
509                 # wait for memMap_Data to return data
510                 if subState == 21:
511                     # test
512                     if subPrint == 1:
513                         print('MAIN: waiting for queue')
514                         subPrint = 2
515 
516                     try:
517                         plantOut = q1.get_nowait()
518                         if isinstance(plantOut, ndarray):
519                             # test
520                             print('MAIN: Got data')
521                             # change substate
522                             subState = 22
523                         elif plantOut == 'End':
524                             # test
525                             print('MAIN: Got end inidcator')
526                             # change substate
527                             subState = 24
528                     except queue.Empty:
529                         pass
530 
531             # ------------------------------------------------------------------------------------#
532             # ----------------------------------SUBSTATE=22---------------------------------------#
533             # ------------------------------------------------------------------------------------#
534                 # read the meta data
535                 elif subState == 22:
536                     # test
```

```
                if subPrint == 2:
                    print('MAIN: Reading meta data')
                    subPrint = 3

                # read the metadata into a c-type struct
                mapMeta = mmap(SM3_Plant_parent.fd, SM3_Plant_parent.size)
                metaData = memory_to_struct(buffer=mapMeta, struct=MetaData)
                mapMeta.close()

                # extract the meta data from the struct
                filename = metaData.fileName.decode("utf-8")  # filename
                nrSamples = metaData.nrSamples  # total number of dataBlocks caught
                nrMes = metaData.nrMes  # numer of values in thecurrent dataBlock
                freq = metaData.freq  # frequency of the measurement [Hz]

                # change substate
                subState = 23

            # ----------------------------------------------------------------------------------#
            # ----------------------------------SUBSTATE=23-------------------------------------#
            # ----------------------------------------------------------------------------------#
            # write the data to a hdf5 file
            elif subState == 23:
                # test
                if subPrint == 3:
                    print('MAIN: writing plant data')
                    subPrint = 4

                file = h5.File(filename, 'w')
                nameDataSet = "Induction Plant Data"
                for index, name in enumerate(headerList):
                    if unitList[index] == 's':
                        grp = file.create_dataset(nameDataSet + '/' + 'Zeit' + '/' + name,
                            data=plantOut[:, index])
                        grp.attrs['unit'] = unitList[index]
                    elif unitList[index] == 'L/min':
                        grp = file.create_dataset(nameDataSet + '/' + 'Durchfluss' + '/' +
                            name, data=plantOut[:, index])
                        grp.attrs['unit'] = unitList[index]
                    elif unitList[index] == 'bar':
                        grp = file.create_dataset(nameDataSet + '/' + 'Druck' + '/' + name,
                            data=plantOut[:, index])
                        grp.attrs['unit'] = unitList[index]
                    elif unitList[index] == '°C':
                        grp = file.create_dataset(nameDataSet + '/' + 'Temperatur' + '/' +
                            name, data=plantOut[:, index])
                        grp.attrs['unit'] = unitList[index]
                    elif unitList[index] == '%':
                        grp = file.create_dataset(nameDataSet + '/' + 'Andere' + '/' + name,
                            data=plantOut[:, index])
                        grp.attrs['unit'] = unitList[index]

                # write the attributes
                file.get(nameDataSet).attrs['Creation date'] = str(
                    dt.now().strftime("%d{sym}%m{sym}%Y").format(sym='.'))
                file.get(nameDataSet).attrs['Start time'] = startTime
                file.get(nameDataSet).attrs['End time'] =
                   str(dt.now().strftime("%H{sym}%M{sym}%S").format(sym=':'))
                file.get(nameDataSet).attrs['Frequency'] = "{f} [Hz]".format(f=freq)
                file.close()

                # test
                print('MAIN: !!Writing successfull!!')
                # change substate
                subState = 24

            # ----------------------------------------------------------------------------------#
            # ----------------------------------SUBSTATE=24-------------------------------------#
            # ----------------------------------------------------------------------------------#
            # clean everythin up
            elif subState == 24:
                # test
                if subPrint == 4:
                    print('MAIN: Clean everything up')
                    subPrint = 1

                # stop the child process
```

```
                p1.kill()
                q1.close()

                # send an end indicator to Codesys
                mapFlag = mmap(SM_Setup.fd, SM_Setup.size)
                newFlag = 4
                newFlag = newFlag.to_bytes(1, "little")
                mapFlag.seek(0)
                mapFlag.write(newFlag)
                mapFlag.close()

                # close the shared memories
                SM_Setup.close_fd()
                SM3_Plant_parent.close_fd()
                SM2 Pico.close fd()

                # delete all elements that would occupy memory space otherwise
                for element in dir():
                    if element[0:2] != "__":
                        del globals()[element]

                # prepare for a new measurement
                init = True
                startM = True
                initShmTrig = False
                initShmMeta = False

                # change state
                state = 0

# ------------------------------------------------------------------------------------------#
# ----------------------------------------STATE=30------------------------------------------#
# ------------------------------------------------------------------------------------------#
        # start only the yoke measurement
        # The writing to a hdf5-file is not yet implemented, since the data received from the
        # yoke is not yet consistent.

        # start 'getData_Pico' as a child process
        elif state == 30:
            # test
            print('MAIN: waiting for yoke data')

            # start the subprocess
            if startM:
                q2 = Queue()  # set up a queue for the data
                p2 = Process(target=fun_pico, args=(q2, N_EM, maxSamples, timebase))
                p2.start()  # start getData_Pico.py
                subState = 31  # set substate to wait for data to arrive
                startM = False  # start childprocess just once
                yokeMesList = list()  # list to store all data-arrays from
                                      # getData_Pico.py

        # ------------------------------------------------------------------------------------#
        # ---------------------------------SUBSTATE=31----------------------------------------#
        # ------------------------------------------------------------------------------------#
            # wait for data to arrive
            if subState == 31:
                try:
                    yokeMesData = q2.get_nowait()
                    if yokeMesData == 'End':  # 'End' is put to the queue from getData_Pico.py
                                              # when the last pico measurement was done
                        # change substate
                        subState = 32
                    elif not yokeMesData == 'End':  # store all received arrays in a list
                        yokeMesList.append(yokeMesData)
                        # test
                        print('MAIN: Got a block')
                except queue.Empty:
                    pass
        # ------------------------------------------------------------------------------------#
        # ---------------------------------SUBSTATE=32----------------------------------------#
        # ------------------------------------------------------------------------------------#
            # Write the data to an hdf5-file
            elif subState == 32:
                # ------------------------------------------------------------------------#
                # Here should come code to process the data in 'yokeMesList' but since the
                # data from pico is not received in the right way yet it doesn't make
```

```
691                 # sense to implement it right now, before we know which exact format we have.
692                 # For later implementation we basically only need to ajust the code for the
693                 # hdf5-file from the plant measurement at state = 10
694                 # ------------------------------------------------------------------------------#
695                 for data in yokeMesList:
696                 # Do something with the data
697                 # data is an array: data[0, 0] = time of the measurement in ns since epoch
698                     pass
699 
700                 # change substate
701                 subState = 33  # go to state = 33 to clean everything up
702 
703             # ------------------------------------------------------------------------------------#
704             # ----------------------------------SUBSTATE=33---------------------------------------#
705             # ------------------------------------------------------------------------------------#
706                 # clean everything up
707                 elif subState == 33:
708                     # stop the cild process
709                     p2.kill()
710                     q2.close()
711 
712                     # signalize codesys that the measurement has finished
713                     mapFlag = mmap(SM_Setup.fd, SM_Setup.size)
714                     newFlag = 4
715                     newFlag = newFlag.to_bytes(1, "little")
716                     mapFlag.seek(0)
717                     mapFlag.write(newFlag)
718                     mapFlag.close()
719 
720                     # close the shared memories
721                     SM_Setup.close_fd()
722                     SM3_Plant_parent.close_fd()
723                     SM2_Pico.close_fd()
724 
725                     # clean everything up
726                     for element in dir():
727                         if element[0:2] != "__":
728                             del globals()[element]
729 
730                     # prepare for a new measurement
731                     startM = True
732                     initShmTrig = False
733                     initShmMeta = False
734                     init = True
735 
736                     # test
737                     print('MAIN: Yoke measurement finished')
738 
739                     # change state
740                     state = 0
```

Anhang C: ST-Code Induktionsprüfstand

Dem nachfolgenden Anhang kann der in Codesys verfasste ST-Code entnommen werden, der der Messdatenerfassung des Induktionsprüfstandes dient.

Anhang C1: Quellcode: FUNCTION „AQUIRE_DATA“

```
// Author: Dominik Müller
// Version: CODESYS V3.5 SP17 Patch 3
// Project: MatDatSys - Material Center Leoben
// Last modified: 06.02.2023

(*This FUNCTION gives back an Array with all the input measurement for the current cycle
If it's the first cycle in the measurement the time is saved as starttime and is used further
on to compute the relative time of the measurement instances
INPUT: -
OUTPUT: AQUIRE_DATA: array with dimensions 24x1 with the measurement data
*)

(*----------------------------------VARIABLE DECLERATION--------------------------------*)
FUNCTION AQUIRE_DATA: ARRAY[1..24] OF REAL
VAR_INPUT
END_VAR
VAR
   n_Waermeregelung: STRING; // measurement value "Istwert Drehantrieb Wärmeregelung" [%]
   p_Brause: STRING;         // measurement value "Druck Brause" [bar]
   p_Gas: STRING;            // measurement value "Druck Gas" [bar]
   p_Gasbrause1: STRING;     // measurement value "Druck Gasbrause 1" [bar]
   p_Gasbrause2: STRING;     // measurement value "Druck Gasbrause 2" [bar]
   Q_Brause: STRING;         // measurement value "Durchfluss Brause 1" [L/min]
   Q_Gasbrause1: STRING;     // measurement value "Durchfluss Gasbrause 1" [L/min]
   Q_Gasbrause2: STRING;     // measurement value "Durchfluss Gasbrause 2" [L/min]
   T_abschreck: STRING;      // measurement value "Temperatur Abschreckmitteltank" [°C]
   T_Brause: STRING;         // measurement value "Temperatur Brause" [°C]
   T_Gas: STRING;            // measurement value "Temperatur Gas" [°C]
   T_ind_rueck: STRING;      // measurement value "Temperatur Induktor Kühlung Rücklauf" [°C]
   T_ind_zulauf: STRING;     // measurement value "Temperatur Inuktor Kühlung Zulauf" [°C]
   temp_TC1: STRING;         // measurement value "Thermoelement 1" [°C]
   temp_TC2: STRING;         // measurement value "Thermoelement 2" [°C]
   temp_TC3: STRING;         // measurement value "Thermoelement 3" [°C]
   temp_TC4: STRING;         // measurement value "Thermoelement 4" [°C]
   temp_TC5: STRING;         // measurement value "Thermoelement 5" [°C]
   temp_TC6: STRING;         // measurement value "Thermoelement 6" [°C]
   temp_TC7: STRING;         // measurement value "Thermoelement 7" [°C]
   temp_TC8: STRING;         // measurement value "Thermoelement 8" [°C]
   temp_TC9: STRING;         // measurement value "Thermoelement 9" [°C]
   temp_TC10: STRING;        // measurement value "Thermoelement 10" [°C]

   // Variables for computing the current time
   sysTimestamp_UTC: Util.SysTimeRtc.SYSTIME;
   sysTime_UTC: ULINT;
   deltaTime: ULINT;
   deltaSec: ULINT;
   deltaMS: ULINT;
   deltaMin: ULINT;
   deltaH: ULINT;
   deltaTimeDate: Util.SYSTIMEDATE;
   deltaTimeFin: UDINT;

```

```
   currentRelTimeSt: STRING;
   sec: STRING;
   ms: STRING;
END_VAR
```

```
(*-----------------------------------EXECUTED CODE-----------------------------------------*)
// for a new measurement, get the start time
IF GVL.newFile THEN
          sysTimestamp_UTC := Util.SysTimeRtc.SysTimeRtcHighResGet(sysTime_UTC);
          GVL.startTime := sysTime_UTC;         // start time plantdata capturing [ms]
          GVL.newFile := FALSE;
END_IF

// compute the relative time for the current cycle
sysTimestamp_UTC := Util.SysTimeRtc.SysTimeRtcHighResGet(sysTime_UTC);
deltaTime := sysTime_UTC - GVL.startTime;

deltaTimeFin := Util.SysTimeRtcConvertHighResToDate(deltaTime, deltaTimeDate);
deltaMS := deltaTimeDate.wMilliseconds;
deltaSec := deltaTimeDate.wSecond;
deltaMin := deltaTimeDate.wMinute;
deltaH := deltaTimeDate.wHour;

// get the time in seconds
deltaSec := deltaSec + deltaMin*60 + deltaH*3600;

// write the time as a string in a defined format --> for python usage
IF deltaMS < 9.5 THEN
          ms := CONCAT('00', ULINT_TO_STRING(deltaMS));
          sec := ULINT_TO_STRING(deltaSec);
ELSIF deltaMS < 99.5 THEN
          ms := CONCAT('0', ULINT_TO_STRING(deltaMS));
          sec := ULINT_TO_STRING(deltaSec);
ELSE
          ms := ULINT_TO_STRING(deltaMs);
          sec := ULINT_TO_STRING(deltaSec);
END_IF

currentRelTimeSt := CONCAT(sec,CONCAT('.',ms));
GVL.currentRelTime := STRING_TO_REAL(currentRelTimeSt);

// fill the array with values
AQUIRE_DATA[1] := STRING_TO_REAL(currentRelTimeSt);
AQUIRE_DATA[2] := GVL.Q_Brause;
AQUIRE_DATA[3] := GVL.p_Brause;
AQUIRE_DATA[4] := GVL.p_Gas;
AQUIRE_DATA[5] := GVL.Q_Gasbrause1;
AQUIRE_DATA[6] := GVL.p_Gasbrause1;
AQUIRE_DATA[7] := GVL.Q_Gasbrause2;
AQUIRE_DATA[8] := GVL.p_Gasbrause2;
AQUIRE_DATA[9] := GVL.n_Waermeregelung;
AQUIRE_DATA[10] := GVL.T_ind_zulauf;
AQUIRE_DATA[11] := GVL.T_ind_rueck;
AQUIRE_DATA[12] := GVL.T_abschreck;
AQUIRE_DATA[13] := GVL.T_Brause;
AQUIRE_DATA[14] := GVL.T_Gas;
AQUIRE_DATA[15] := GVL.temp_TC1;
AQUIRE_DATA[16] := GVL.temp_TC2;
AQUIRE_DATA[17] := GVL.temp_TC3;
AQUIRE_DATA[18] := GVL.temp_TC4;
AQUIRE_DATA[19] := GVL.temp_TC5;
AQUIRE_DATA[20] := GVL.temp_TC6;
```

```
58    AQUIRE_DATA[21] := GVL.temp_TC7;
59    AQUIRE_DATA[22] := GVL.temp_TC8;
60    AQUIRE_DATA[23] := GVL.temp_TC9;
61    AQUIRE_DATA[24] := GVL.temp_TC10;
```

Anhang C2: Quellcode: FUNCTION „COMPUTE_VALUE“

```
// Author: Dominik Müller
// Version: CODESYS V3.5 SP17 Patch 3
// Project: MatDatSys - Material Center Leoben
// Last modified: 06.02.2023

(*This FUNCTION computes the measurement value from an AIO input siganl
INPUT:    valueMin: REAL; measurement value at minimum input signal
          valueMax: REAL; measurement value at maximum input signal
          signalMin: REAL; minimum amplitude of input signal
          signalMax: REAL; maximum amplitude of input signal
          Current_or_voltage: REAL; measured input value from AIO-module
OUTPUT: COMPUTE_VALUE: REAL; the measurement value computed from the AIO-modul input signal*)

(*-----------------------------------VARIABLE DECLERATION---------------------------------*)
FUNCTION COMPUTE_VALUE : REAL
VAR_INPUT
          valueMin: REAL;              // min measurement value [unit of the measured entity]
          valueMax: REAL;              // max measurement value [unit of the measured entity]
          signalMin: REAL;             // min value sensor signal [myA] or [mV]
          signalMax: REAL;             // max value sensor signal [myA] or [mV]
          Current_or_voltage: REAL;    // input signal from AIO-module [myA] or [mV]
END_VAR
VAR
END_VAR
```

```
(*-----------------------------------EXECUTED CODE-----------------------------------------*)

COMPUTE_VALUE := (valueMax-valueMin)/(signalMax-signalMin) * (Current_or_voltage-signalMin) +
                    valueMin;
```

Anhang C3: Quellcode: FUNCTION „CREATE_FILENAME“

```
// Author: Dominik Müller
// Version: CODESYS V3.5 SP17 Patch 3
// Project: MatDatSys - Material Center Leoben
// Last modified: 06.02.2023

(*This FUNCTION is used to create a filename for a new file constisting of the date and time
when the measurement was started which data is contained in this file.
INPUT: filePath: STRING that represents a filepath
OUTPUT: CREATA_FILENAME: STRING in the format: dd.mm.yyyy_xxhxxminxxsec'.h5*)

(*----------------------------------VARIABLE DECLERATION---------------------------------*)
FUNCTION CREATE_FILENAME : CAA.FILENAME
VAR_INPUT
          filePath: STRING;  // the path on the RevPi where the Log-file is stored
END_VAR
VAR
   // necessary variables to compute the system time in the wanted format
   sysTimestamp_UTC: DWORD;
   sysTime_UTC: Util.SysTimeRtc.RTS_IEC_RESULT;
   SysTimeRtcConvertUtcToDate: Util.SysTimeRtc.RTS_IEC_RESULT;
   sysTime_DateTime: Util.SysTimeRtc.SYSTIMEDATE;
   year: STRING;
   month: STRING;
   day: STRING;
   hour: STRING;
   minute: STRING;
   second: STRING;
   fileName: STRING;
END_VAR
```

```
(*----------------------------------EXECUTED CODE----------------------------------------*)
// Get UTC time in milliseconds since Thursday, 1.1.1970 00:00:00
sysTimestamp_UTC := Util.SysTimeRtc.SysTimeRtcGet(sysTime_UTC);
SysTimeRtcConvertUtcToDate := Util.SysTimeRtc.SysTimeRtcConvertUtcToDate(sysTimestamp_UTC,
sysTime_DateTime);

// extract all the necessary information and convert it to strings
year := UINT_TO_STRING(sysTime_DateTime.wYear);
month := UINT_TO_STRING(sysTime_DateTime.wMonth);
day := UINT_TO_STRING(sysTime_DateTime.wDay);
hour := UINT_TO_STRING(sysTime_DateTime.wHour + 2);      // just add 2 to convert to Central
European Standard Time per pedes
minute := UINT_TO_STRING(sysTime_DateTime.wMinute);
second := UINT_TO_STRING(sysTime_DateTime.wSecond);

// add leading 0's where it is necessary
IF sysTime_DateTime.wMonth < 10 THEN
          month := CONCAT('0',month);
END_IF
IF sysTime_DateTime.wHour < 10 THEN
          hour := CONCAT('0', hour);
END_IF
IF sysTime_DateTime.wMinute < 10 THEN
          minute := CONCAT('0',minute);
END_IF
IF sysTime_DateTime.wSecond < 10 THEN
          second := CONCAT('0',second);
END_IF

```

```
30    // create the filname by joining the strings
31    fileName := CONCAT(day,'.');
32    fileName := CONCAT(fileName,month);
33    fileName := CONCAT(fileName,'.');
34    fileName := CONCAT(fileName,year);
35    fileName := CONCAT(fileName,'_');
36    fileName := CONCAT(fileName,hour);
37    fileName := CONCAT(fileName,'h');
38    fileName := CONCAT(fileName,minute);
39    fileName := CONCAT(filename, 'min');
40    fileName := CONCAT(filename, second);
41    fileName := CONCAT(filename, 'sec');
42    fileName := CONCAT(fileName, '.h5');
43
44    // return the filename
45    CREATE_FILENAME := CONCAT(filePath,fileName);
```

Anhang C4: Quellcode: FUNCTION „CREATE_METADATA“

```
// Author: Dominik Müller
// Version: CODESYS V3.5 SP17 Patch 3
// Project: MatDatSys - Material Center Leoben
// Last modified: 06.02.2023

(*This FUNCTION is used to write the metadata into a struct that can be read into a ctype
  struct in python
INPUT:    fileName: STRING containing the whole filename
          sampleNumber: INT which represents number of datablocks measured during a
                    measurement
          nrMeasurementsInArray: INT which represents the written measurementvalues in the
                    current datablock
          frequency: INT which represents the cycle frequency of the PlantLoggerTask [Hz]
OUTPUT:   CREATE_METADATA: Meta_Data_Struct*)

(*----------------------------------VARIABLE DECLERATION---------------------------------*)
FUNCTION CREATE_METADATA : Meta_Data_STRUCT
VAR_INPUT
          fileName: STRING(80);                 // the filename of the current measurment
          sampleNumber: INT;                    // size: 4 bytes; total number of datablocks
measured for the plant measurement
          nrMeasurementsInArray: INT;  // number of measurements made in the current loop
(important for last loop, otherwise this value will be 1000)
          frequency: INT;                                 // frequency of the TASK
"PlantLoggerTask" during the current measurment
END_VAR
VAR
          fileNameByte: ARRAY[1..80] OF BYTE;   // filename of measurement bytearray
          CharP: POINTER TO BYTE;
          Index: INT;
END_VAR
```

```
(*-----------------------------------EXECUTED CODE-----------------------------------------*)
CharP := ADR(fileName);

FOR Index:=1 TO LEN(fileName) DO
          fileNameByte[Index] := CharP^;
          CharP := CharP+1;
END_FOR

CREATE_METADATA.fileName := fileNameByte;
CREATE_METADATA.sampleNumber := sampleNumber;
CREATE_METADATA.nrMeasurementsInArray := nrMeasurementsInArray;
CREATE_METADATA.frequency := frequency;
CREATE_METADATA.startTime := GVL.startTime*1000000;// start time of the plant measurement [ns]
```

Anhang C5: Quellcode: FUNCTION „FILE_PERMISSION”

```
FUNCTION FILE_PERMISSION : BOOL
// Author: Dominik Müller
// Version: CODESYS V3.5 SP17 Patch 3
// Project: MatDatSys - Material Center Leoben
// Last modified: 06.02.2023

//currently not in use

(* This FUNCTION gives permission to the RevPi to manipulate files in the folder
'/home/pi/DataLogger/PlantData' for pushing them into the network *)

(*-----------------------------------VARIABLE DECLERATION---------------------------------*)
VAR_INPUT
END_VAR
VAR
          // necessary variables two execute a consol command
          dwCopySize: DWORD;
          dutResult : RTS_IEC_RESULT;
          szCommand : STRING;
          fbTimer : TON;
          szStdOout : STRING(1000);
          fbTimer1 : TON;
END_VAR
```

```
(*-----------------------------------EXECUTED CODE-----------------------------------------*)
SysProcess_Implementation.SysProcessExecuteCommand2(pszCommand:='sudo chmod -R a+rwx
                                                                 /home/pi/DataLogger/PlantData',
                                                                 pszStdOut:=szStdOout,
                                                                 udiStdOutLen:= SIZEOF(szStdOout),
                                                                 pResult := ADR(dutResult));
FILE_PERMISSION := TRUE;
```

Anhang C6: Quellcode: FUNCTION „STRING_TO_ARRAYBYTE“

```
// Author: Dominik Müller
// Version: CODESYS V3.5 SP17 Patch 3
// Project: MatDatSys - Material Center Leoben
// Last modified: 06.02.2023

(* This FUNCTION does the following:
In order to transfer a string via the shared memory to a python script the string
has to be converted to an array of bytes, which is then converted back to a string.
This function converts a given string to an array of byte
INPUT: stringIn: STRING
OUTPUT: ARRAY that contains the string in byte-format*)

(*----------------------------------VARIABLE DECLERATION--------------------------------*)
FUNCTION STRING_TO_ARRAYBYTE : ARRAY[1..10] OF BYTE
VAR_INPUT
          stringIn: STRING;
END_VAR
VAR
          stringByte: ARRAY[1..10] OF BYTE;     // filename of the current measurement
          CharP: POINTER TO BYTE;
          Index: INT;
END_VAR
```

```
(*-----------------------------------EXECUTED CODE----------------------------------------*)
CharP := ADR(stringIn);

FOR Index:=1 TO LEN(stringIn) DO
          stringByte[Index] := CharP^;
          CharP := CharP+1;
END_FOR

STRING_TO_ARRAYBYTE := stringByte;
```

Anhang C7: Quellcode: FUNCTION „OB001”

```
// Author: Dominik Müller
// Version: CODESYS V3.5 SP17 Patch 3
// Project: MatDatSys - Material Center Leoben
// Last modified: 06.02.2023

(* This FUNCTION does the following: On startup the python scripts are copied to the intended
location
This is currently not important, but I wanted to have the scripts stored at the
PLC-programs location *)
(*----------------------------------VARIABLE DECLERATION---------------------------------*)
FUNCTION OB001 : DWORD
VAR_IN_OUT
          EventPrm: CmpApp.EVTPARAM_CmpApp;
END_VAR
VAR
          dwCopySize: DWORD;
          dutResult : RTS_IEC_RESULT;
          szCommand : STRING(200);
          fbTimer : TON;
          szStdOout : STRING(1000);
          fbTimer1 : TON;
          Command : STRING(200);

          ifSetup: BOOL := TRUE;
END_VAR
```

```
(*-----------------------------------EXECUTED CODE-----------------------------------------*)
//Copy the scripts to local
IF ifSetup THEN
          extTriggerPico := 0;

          SysFile.SysFileCopy('/home/pi/Desktop/memMap_Data.py'
                                    ,'/var/opt/codesys/PlcLogic/Application/memMap_Data.py'
                                    ,ADR(dwCopySize));

          SysFile.SysFileCopy('/home/pi/Desktop/getData_Pico.py'
                                    ,'/var/opt/codesys/PlcLogic/Application/getData_Pico.py'
                                    ,ADR(dwCopySize));

          SysFile.SysFileCopy('/home/pi/Desktop/startProcess.py'
                                    ,'/var/opt/codesys/PlcLogic/Application/startProcess.py'
                                    ,ADR(dwCopySize));

          ifSetup := FALSE;
END_IF
```

Anhang C8: Quellcode: FUNCTION „OS001“

```
// Author: Dominik Müller
// Version: CODESYS V3.5 SP17 Patch 3
// Project: MatDatSys - Material Center Leoben
// Last modified: 06.02.2023

(*This FUNCTION does the following: On shutdown close all the shared memories for proper
cleanup*)

(*-----------------------------------VARIABLE DECLERATION---------------------------------*)
FUNCTION OS001 : DWORD
VAR_IN_OUT
          EventPrm: Component_Manager.EVTPARAM_CmpMgr_Shutdown;
END_VAR
VAR
          iCloseMem: __UXINT;
END_VAR
```

```
(*-----------------------------------EXECUTED CODE----------------------------------------*)
// shared memories for plant logging
iCloseMem := SysSharedMemoryDelete(hShm := Plant_Logger.hShMemEnd);
iCloseMem := SysSharedMemoryDelete(hShm := Plant_Logger.hShMemFlags);
iCloseMem := SysSharedMemoryDelete(hShm := Plant_Logger.hShMemWriteData);
iCloseMem := SysSharedMemoryDelete(hShm := Plant_Logger.hShMemWriteMeta);

// shared memory for pico triggering
iCloseMem := SysSharedMemoryDelete(hShm := Pico_Logger.hShMemPicoTrigger);

// shared memories for setting up the measurement
iCloseMem := SysSharedMemoryDelete(hShm := Setup_Measurement.hShmParent);
iCloseMem := SysSharedMemoryDelete(hShm := Setup_Measurement.hShmPicoSETUP);
```

Anhang C9: Quellcode: PROGRAM „Measurement_Data“

```
// Author: Dominik Müller
// Version: CODESYS V3.5 SP17 Patch 3
// Project: MatDatSys - Material Center Leoben
// Last modified: 06.02.2023

(* This PROGRAM is responsible for computing the measurement values each cycle.*)

(*-----------------------------------VARIABLE DECLERATION---------------------------------*)
PROGRAM Measurement_Data
VAR
   (* VARIABLES FOR DATA MEASUREMENT *)

   Slope_TC: REAL := 1400.0/16000.0; // [C°/myA]

   maxValue_current: REAL := 20000;   // maximum value for current inputsignals [myA]
   minValue_current: REAL := 4000;    // minimum value for current Inputsignals [myA]
   maxValue_voltage: REAL := 10000;   // maximum value for voltage inputsignals [mV]
   minValue_voltage: REAL := 2000;    // minimum value for voltage inputsignals [mV]

   minTC: REAL := 0;         // minimum temperature value at an input of 4mA for TC [°C]
   maxTC: REAL := 1400;      // maximum temperature value at an input of 20mA for RC [°C]

   minPressure: REAL := 0;   // pressure for 4mA input [bar]
   maxPressure: REAL := 10;  // pressure for 20mA input [bar]

   minPercentWaerme: REAL := 0;       // minium value for n_Waermeregelung at 2V [%]
   maxPercentWaerme: REAL := 100;     // maximum value for n_Waermeregelung at 10V [%]

   minQBrause: REAL := 1;             // minimum flow value for 4mA input [L/min]
   maxQBrause: REAL := 100;           // maximum flow value for 20mA imput [L/min]

   minQGasbrause: REAL := 6;          // minimum flow value for 4mA input [L/min]
   maxQGasbrause: REAL := 600;        // maximum flow value for 20mA input [L/min]


END_VAR
```

```
(*-----------------------------------EXECUTED CODE-----------------------------------------*)
(*-----------------------------------------------------------------------------------------*)
(* READING THE INPUT SIGNALS FROM THE THERMOCOUPLES AND COMPUTING THEM TO MEASUREMENTVALUES*)

//Thermocouple 1(TC1), Analog input 1 (AI1) on RevPi_AIO1
GVL.temp_TC1 := COMPUTE_VALUE(minTC, maxTC, minValue_current, maxValue_current, current_TC1);

//Thermocouple 2(TC2), Analog input 2 (AI2) on RevPi_AIO1
GVL.temp_TC2 := COMPUTE_VALUE(minTC, maxTC, minValue_current, maxValue_current, current_TC2);

//Thermocouple 3(TC3), Analog input 3 (AI3) on RevPi_AIO1
GVL.temp_TC3 := COMPUTE_VALUE(minTC, maxTC, minValue_current, maxValue_current, current_TC3);

//Thermocouple 4(TC4), Analog input 4 (AI4) on RevPi_AIO1
GVL.temp_TC4 := COMPUTE_VALUE(minTC, maxTC, minValue_current, maxValue_current, current_TC4);

//Thermocouple 5(TC5), Analog input 1 (AI1) on RevPi_AIO_2
GVL.temp_TC5 := COMPUTE_VALUE(minTC, maxTC, minValue_current, maxValue_current, current_TC5);

//Thermocouple 6(TC6), Analog input 2 (AI2) on RevPi_AIO_2
GVL.temp_TC6 := COMPUTE_VALUE(minTC, maxTC, minValue_current, maxValue_current, current_TC6);
```

```
23  //Thermocouple 7(TC7), Analog input 3 (AI3) on RevPi_AIO_2
24  GVL.temp_TC7 := COMPUTE_VALUE(minTC, maxTC, minValue_current, maxValue_current, current_TC7);
25
26  //Thermocouple 8(TC8), Analog input 4 (AI4) on RevPi_AIO_2
27  GVL.temp_TC8 := COMPUTE_VALUE(minTC, maxTC, minValue_current, maxValue_current, current_TC8);
28
29  //Thermocouple 9(TC9), Analog input 1 (AI1) on RevPi_AIO_3
30  GVL.temp_TC9 := COMPUTE_VALUE(minTC, maxTC, minValue_current, maxValue_current, current_TC9);
31
32  //Thermocouple 10(TC10), Analog input 2 (AI2) on RevPi_AIO_3
33  GVL.temp_TC10 := COMPUTE_VALUE(minTC, maxTC, minValue_current, maxValue_current,
34  current_TC10);
35  (*-------------------------------------------------------------------------------------------*)
36
37  (*-------------------------------------------------------------------------------------------*)
38  (* READING THE SENSOR VALUES FROM THE PLANT AND COMPUTING THEM TO MEASUREMENTVALUEs*)
39
40  // "Durchfluss Brause 1" [L/min]; AI1 on RevPi_AIO_4
41  GVL.Q_Brause :=  COMPUTE_VALUE(minQBrause, maxQBrause, minValue_current, maxValue_current,
42            current_QB1);
43
44  // "Druck Brause" [bar]; AI2 on RevPi_AIO_4
45  GVL.p_Brause := COMPUTE_VALUE(minPressure, maxPressure, minValue_current, maxValue_current,
46            current_pB);
47
48  // "Druck Gas" [bar]; AI3 on RevPi_AIO_4
49  GVL.p_Gas := COMPUTE_VALUE(minPressure, maxPressure, minValue_current, maxValue_current,
50            current_pG);
51
52  //"Durchfluss Gasbrause 1" AI4 on RevPi_AIO_4
53  GVL.Q_Gasbrause1 := COMPUTE_VALUE(minQGasbrause, maxQGasbrause, minValue_current,
54            maxValue_current, current_QGb1);
55
56  // "Druck Gasbrause 1" AI1 on RevPi_AIO_5
57  GVL.p_Gasbrause1 := COMPUTE_VALUE(minPressure, maxPressure, minValue_current,
58            maxValue_current, current_pGb1);
59
60  //"Durchfluss Gasbrause 2" AI2 on RevPi_AIO_5
61  GVL.Q_Gasbrause2 := COMPUTE_VALUE(minQGasbrause, maxQGasbrause, minValue_current,
62            maxValue_current, current_QGb2);
63
64  // "Druck Gasbrause 2" AI3 on RevPi_AIO_5
65  GVL.p_Gasbrause2 := COMPUTE_VALUE(minPressure, maxPressure, minValue_current,
66            maxValue_current, current_pGb2);
67
68  // "Istwert Drehantrieb Wärmeregelung" AI4 on RevPi_AIO_5
69  GVL.n_Waermeregelung := COMPUTE_VALUE(minPercentWaerme, maxPercentWaerme, minValue_voltage,
70            maxValue_voltage, voltage_n);
71  (*-------------------------------------------------------------------------------------------*)
72
73  (*-------------------------------------------------------------------------------------------*)
74                      (* READING THE PT100 VALUES FROM THE PLANT *)
75
76  (* !!!! NOTE: THIS PART HAS TO BE CHECKED IF "HINLAUF" AND RÜCKLAUF" ARE THE CORRECT CHANNELS
77  SINCE IN THE DATASHEET THEY ARE BOTH NAMED "RÜCKLAUF" !!!!*)
78  (* !!! These measurement values are not yet physically connected to the induction plant !!!*)
79  // "Induktor Kühlung Zulauf" PT100-Channel1 on RevPi_AIO4
80  GVL.T_ind_zulauf := T_IK_zu;
81
82  // "Induktor Kühlung Rücklauf" PT100-Channel2 on RevPi_AIO_5
83  GVL.T_ind_rueck := T_IK_rueck;
84
```

```
85  // "Temperatur Abschreckmitteltank" PT100_Channel1 on RevPi_AIO_4
86  GVL.T_abschreck := T_Ab;
87
88  // "Temperatur Brause" PT100_Channel2 on RevPi_AIO_4
89  GVL.T_Brause := T_Br;
90
91  // "Temperatur Gas" PT100_Channel1 on RevPi_AIO_3
92  GVL.T_Gas := T_G;
93  (*-----------------------------------------------------------------------------------------*)
94
```

Anhang C10: Quellcode: PROGRAM „Pico_Logger“

```
// Author: Dominik Müller
// Version: CODESYS V3.5 SP17 Patch 3
// Project: MatDatSys - Material Center Leoben
// Last modified: 06.02.2023

(*This PROGRAM is used to trigger the start of the picoscope-measurements*)

(*----------------------------------VARIABLE DECLERATION--------------------------------*)
PROGRAM Pico_Logger
VAR
   // Toggle variables for Pico measurement
   picoTrigger: BOOL;                // Triggers the start of the timer
   picoActivate: BOOL;               // writes the trigger variable to the shared memory
   firstMeasurement: BOOL := TRUE;   // when the pico measurement is started the timer starts
                                     // counting

   // sample number of the plant measurement (when needed)
   nrValue: UDINT;

   // struct to Trigger the python pico script
   flag_data: Trigger_STRUCT;        // Object that is written the the shared memory
   sizeFlag:UDINT := SIZEOF(Trigger_STRUCT);   // size of the shared memory for pico
                                               // triggering

   // shared memory for yoke trigger
   PICO_FLAG_NAME: STRING := '_CODESYS_TRIGGER_Pico';             // yoke measurement trigger
   hShMemPicoTrigger: RTS_IEC_HANDLE := RTS_INVALID_HANDLE; // handle of the shared memory
   resultOpenPicoTrigger: RTS_IEC_RESULT;               // result of the shm opening
   resultWritePicoTrigger: RTS_IEC_RESULT;              // result of shm writing
   iWritePicoTrigger: __UXINT;


   // Timer for Pico Trigger
   Timer: TON;              // timer
   startTimer: BOOL;        // start the timer when the measurement starts
   r_EdgeTimer : R_TRIG;    // rising edge dedection for the trigger
   timerReset: BOOL;        // reset the timer after triggering

   // variables for reading the system time
   sysTimestamp_UTC: DWORD;
   sysTime_UTC: Util.SysTimeRtc.RTS_IEC_RESULT;
   SysTimeRtcConvertUtcToDate: Util.SysTimeRtc.RTS_IEC_RESULT;
   sysTime_DateTime: Util.SysTimeRtc.SYSTIMEDATE;
   year: STRING;
   month: STRING;
   day: STRING;
   hour: STRING;
   minute: STRING;
   second: STRING;
END_VAR

(*----------------------------------EXECUTED CODE----------------------------------------*)
(*---------------------------------------------------------------------------------------*)
                            (*Set up the shared memory*)

// shared memory for triggering the pico measurement
IF        hShMemPicoTrigger = RTS_INVALID_HANDLE THEN
          flag_data.flagData := 0;
```

```
8           hShMemPicoTrigger := SysSharedMemoryCreate(pszName := PICO_FLAG_NAME,
9                   ulPhysicalAddress := 0, pulSize := ADR(sizeFlag), pResult :=
10                  ADR(resultOpenPicoTrigger));
11          iWritePicoTrigger := SysSharedMemoryOpen2(pszName := PICO_FLAG_NAME,
12                  ulPhysicalAddress := 0, pulSize := ADR(sizeFlag), pResult :=
13                  ADR(resultOpenPicoTrigger));
14          iWritePicoTrigger := SysSharedMemoryWrite(hShMemPicoTrigger, 0, ADR(flag_data),
15                  sizeFlag, ADR(resultWritePicoTrigger));
16          iWritePicoTrigger := SysSharedMemoryClose(hShm := hShMemPicoTrigger);
17
18  END_IF
19  (*-------------------------------------------------------------------------------------------*)
20
21  (*-------------------------------------------------------------------------------------------*)
22                              (*manage the triggering of the picoscope*)
23  // while no process
24  IF Setup_Measurement.startProcess = FALSE OR Setup_Measurement.getTypeMeasurement = 0 OR
25  Setup_Measurement.getTypeLogging = 0 THEN
26          picoTrigger := FALSE;
27  END_IF
28
29  // if the measurement is cancelled
30  IF Setup_Measurement.fStartProcess THEN
31          picoTrigger := FALSE;
32          picoActivate := FALSE;
33          timerReset := TRUE;
34          startTimer := FALSE;
35          firstMeasurement := TRUE;
36
37  // Timer for Pico triggering
38  ELSE
39          Timer(IN := startTimer, PT := REAL_TO_TIME(Setup_Measurement.deltaT_EM_a*1000));
40          picoActivate := Timer.Q;
41
42          r_EdgeTimer(CLK := Timer.Q);
43          timerReset := r_EdgeTimer.Q;
44
45          IF picoTrigger THEN
46                  startTimer := TRUE;
47                  IF firstMeasurement THEN
48                          picoActivate := TRUE;
49                          firstMeasurement := FALSE;
50                  END_IF
51          END_IF
52          IF timerReset THEN
53          startTimer := FALSE;
54          END_IF
55  END_IF
56
57  // In case the shared memory triggers are reset to zero
58  IF GVL.resetShms THEN
59          picoTrigger := FALSE;
60          picoActivate := FALSE;
61          extTriggerPico := 0;
62  END_IF
63
64  // if the timer triggers and no falling edge for the measurement setup is dedected trigger a
65  pico measurement
66  IF picoActivate AND NOT Setup_Measurement.fStartProcess THEN
67          nrValue := Plant_Logger.nrValues;
68          flag_data.flagData := 1;
69
```

```
70              iWritePicoTrigger := SysSharedMemoryOpen2(pszName := PICO_FLAG_NAME,
71                      ulPhysicalAddress := 0, pulSize := ADR(sizeFlag), pResult :=
72                      ADR(resultopenPicoTrigger));
73              iWritePicoTrigger := SysSharedMemoryWrite(hShMemPicoTrigger, 0, ADR(flag_data),
74                      sizeFlag, ADR(resultWritePicoTrigger));
75              iWritePicoTrigger := SysSharedMemoryClose(hShm := hShMemPicoTrigger);
76              picoActivate := FALSE;
77              startTimer := FALSE;
78      END_IF
79      (*-------------------------------------------------------------------------------------------*)
```

Anhang C11: Quellcode: PROGRAM „Plant_Logger“

```
// Author: Dominik Müller
// Version: CODESYS V3.5 SP17 Patch 3
// Project: MatDatSys - Material Center Leoben
// Last modified: 06.02.2023

(* This PROGRAM is used to fill a Data_Storage_STRUCT with measurement data and send it to the
shared memory. It also sets a trigger for a python script to read the STRUCT from the memory*)

(*----------------------------------VARIABLE DECLERATION---------------------------------*)
PROGRAM Plant_Logger
VAR
(* VARIABLES FOR DATA LOGGING *)
          filePathPlant: CAA.FILENAME := '/home/pi/DataLogger/';   // filepath on the RevPi
          fullFileName: CAA.FILENAME;           // filename consisting of path and filename

          // variables for a new measurement
          newMeasurement: BOOL := TRUE;         // catches a new started measurement
          writeDataTrigger: BOOL := FALSE;      // trigger for a new measurement of the plant
          writeDataTriggerYoke: BOOL := FALSE;
          iLine: INT := 1;             // counter for the lines per Data_Storage_STRUCT
          sampleNbr: INT := 0;         // counter for the total number of Data_Storage_STRUCTs
                                       // written to the shared memory
          nrValues: UDINT := 0;        // total number of values measured

          // measurement variables
          data_storage: Data_Storage_STRUCT;    // Array where the measurement data is stored
          data_storageTemp: ARRAY[1..24] OF REAL;         // array to temporarily store the
                                                          // current measurement data

          meta_data: Meta_Data_STRUCT; // Struct to write the metadata for the current
                                       // measurement block

          flag_data: Trigger_STRUCT;   // Flag to trigger the the memory mapping of the python
                                       // file of the measurement data
          flag_end: Trigger_STRUCT;    // Flag to trigger the the memory mapping of the python
                                       // file of the last measurement data

          // variables for shared memory to communicate with python scripts

          // shared memory names
          MEMORY_NAME:STRING := '_CODESYS_MEMORY_Data';            // datablocks
          FLAG_NAME: STRING := '_CODESYS_TRIGGER_Plant';           // read trigger
          META_NAME: STRING := '_CODESYS_MEMORY_Meta';             // metadata
          ENDMEASUREMENT_NAME: STRING := '_CODESYS_TRIGGER_End';   // end trigger

          // shared memory for plant data
          hShMemWriteData: RTS_IEC_HANDLE := RTS_INVALID_HANDLE;
          resultOpenData: RTS_IEC_RESULT;
          resultWriteData: RTS_IEC_RESULT;
          sizeData: UDINT := SIZEOF(Data_Storage_STRUCT);

          // shared memory for meta data
          hShMemWriteMeta: RTS_IEC_HANDLE := RTS_INVALID_HANDLE;
          resultOpenMeta: RTS_IEC_RESULT;
          resultWriteMeta: RTS_IEC_RESULT;
          sizeMeta: UDINT := SIZEOF(Meta_Data_STRUCT);

          // shared memory for read trigger
          hShMemFlags: RTS_IEC_HANDLE := RTS_INVALID_HANDLE;
```

```
60            resultOpenFlags: RTS_IEC_RESULT;
61            resultWriteFlags: RTS_IEC_RESULT;
62            sizeFlags: UDINT := SIZEOF(Trigger_STRUCT);
63
64            // shared memory for last datapackage flag
65            hShMemEnd: RTS_IEC_HANDLE := RTS_INVALID_HANDLE;
66            resultOpenEnd: RTS_IEC_RESULT;
67            resultWriteEnd: RTS_IEC_RESULT;
68
69            // return values for shared memory access
70            iWriteData: __UXINT;
71            iWriteMeta: __UXINT;
72            iWriteFlags: __UXINT;
73            iWriteEnd: __UXINT;
74
75            // some variables for logic handling
76            reset: BOOL:= FALSE;                  // reset the shared memories
77            endM: BOOL := FALSE;                  // catching the end of the current measurement
78                                                  // --> preparation for a new one
79
80            // variables to read the cycle time from the task info
81            taskInfo: POINTER TO Task_Info2;
82            pResult: DWORD;
83            pResult2: DWORD;
84            cycleTime: DWORD;
85            hTask: RTS_IEC_HANDLE;
86            hPlant_LoggerTask : SysTask.RTS_IEC_HANDLE := SysTask.RTS_INVALID_HANDLE;    // task
87    handle
88
89    END_VAR
90
91    (*-----------------------------------EXECUTED CODE-----------------------------------------*)
92    (* ------------------------------------------------------------------------------------------*)
93                                (* Set up the shared memories *)
94
95    // set up the shared memories for communication with the python scripts
96
97    IF hShMemWriteMeta = RTS_INVALID_HANDLE OR hShMemFlags = RTS_INVALID_HANDLE OR hShMemWriteMeta
98            = RTS_INVALID_HANDLE OR hShMemEnd = RTS_INVALID_HANDLE THEN
99
100           // shared memory for the plant data (plant)
101           IF hShMemWriteMeta = RTS_INVALID_HANDLE THEN
102                   hShMemWriteData := SysSharedMemoryCreate(pszName := MEMORY_NAME,
103                   ulPhysicalAddress := 0, pulSize := ADR(sizeData),
104                   pResult := ADR(resultWriteData));
105           END_IF
106
107           // shared memory for the indicator for python to read plantdata (plant)
108           IF hShMemFlags = RTS_INVALID_HANDLE THEN
109                   flag_data.flagData := 0;
110                   hShMemFlags := SysSharedMemoryCreate(pszName := FLAG_NAME,
111                            ulPhysicalAddress := 0, pulSize := ADR(sizeFlags),
112                            pResult := ADR(resultOpenFlags));
113                   iWriteFlags := SysSharedMemoryOpen2(pszName := FLAG_NAME,
114                            ulPhysicalAddress := 0, pulSize := ADR(sizeFlags),
115                            pResult := ADR(resultOpenFlags));
116                   iWriteFlags := SysSharedMemoryWrite(hShMemFlags, 0, ADR(flag_data),
117                            sizeFlags, ADR(resultWriteFlags));
118                   iWriteFlags := SysSharedMemoryClose(hShm := hShMemFlags);
119           END_IF
120
121           // shared memory for the meta data (plant)
```

```
122             IF hShMemWriteMeta = RTS_INVALID_HANDLE THEN
123                     hShMemWriteMeta := SysSharedMemoryCreate(pszName := META_NAME,
124                     ulPhysicalAddress := 0, pulSize := ADR(sizeMeta),
125                     pResult := ADR(resultWriteMeta));
126             END_IF
127
128             // shared memory for end indicator (plant)
129             IF hShMemEnd = RTS_INVALID_HANDLE THEN
130                     flag_end.flagData := 0;
131                     hShMemEnd := SysSharedMemoryCreate(pszName := ENDMEASUREMENT_NAME,
132                             ulPhysicalAddress := 0, pulSize := ADR(sizeFlags),
133                             pResult := ADR(resultOpenEnd));
134                     iWriteEnd := SysSharedMemoryOpen2(pszName := ENDMEASUREMENT_NAME,
135                             ulPhysicalAddress := 0, pulSize := ADR(sizeFlags),
136                             pResult := ADR(resultOpenEnd));
137                     iWriteEnd := SysSharedMemoryWrite(hShMemEnd, 0, ADR(flag_end), sizeFlags,
138                             ADR(resultWriteEnd));
139                     iWriteEnd := SysSharedMemoryClose(hShm := hShMemEnd);
140             END_IF
141
142             // compute the cycle frequence
143             taskInfo := CmpIecTask.IecTaskGetInfo3(hIecTask:=IecTaskGetCurrent(pResult:=
144                     pResult),pResult:=pResult2);
145             cycleTime := taskInfo^.dwInterval;
146             GVL.frequency := UDINT_TO_INT(1000000/cycleTime);
147     END_IF
148     // In case the shared memory triggers are reset to zero
149     IF GVL.resetShms THEN
150             writeDataTrigger := FALSE;
151     END_IF
152
153     // get the task handle for the Plant_LoggerTask
154     SysTask.SysTaskGetCurrent(ADR(hPlant_LoggerTask));
155     (*-----------------------------------------------------------------------------------------*)
156
157     (*-----------------------------------------------------------------------------------------*)
158             (* Code for writing data and transfering it to the python script: memMap_Data.py *)
159
160     // once per cycle store the measurement values in a struct
161     IF Setup_Measurement.fStartProcess OR NOT Setup_Measurement.startProcess THEN
162             writeDataTrigger := FALSE;
163     END_IF
164
165     IF writeDataTrigger THEN
166             // when a new measurment is started set up all the necessary parameters
167             IF newMeasurement THEN
168                     fullFileName := CREATE_FILENAME(filePath := filePathPlant);
169                     newMeasurement := FALSE;
170                     sampleNbr := 1;
171                     endM := TRUE;
172             END_IF
173
174             // every cycle read all input values and save them to a 'Data_Storage_STRUCT'
175             data_storageTemp := AQUIRE_DATA();
176             data_storage.mTime[iLine] := data_storageTemp[1];
177             data_storage.Q_Brause[iLine] := data_storageTemp[2];
178             data_storage.p_Brause[iLine] := data_storageTemp[3];
179             data_storage.p_Gas[iLine] := data_storageTemp[4];
180             data_storage.Q_Gasbrause1[iLine] := data_storageTemp[5];
181             data_storage.p_Gasbrause1[iLine] := data_storageTemp[6];
182             data_storage.Q_Gasbrause2[iLine] := data_storageTemp[7];
183             data_storage.p_Gasbrause2[iLine] := data_storageTemp[8];
```

```
184             data_storage.n_Waermeregelung[iLine] := data_storageTemp[9];
185             data_storage.T_ind_zulauf[iLine] := data_storageTemp[10];
186             data_storage.T_ind_rueck[iLine] := data_storageTemp[11];
187             data_storage.T_abschreck[iLine] := data_storageTemp[12];
188             data_storage.T_Brause[iLine] := data_storageTemp[13];
189             data_storage.T_Gas[iLine] := data_storageTemp[14];
190             data_storage.temp_TC1[iLine] := data_storageTemp[15];
191             data_storage.temp_TC2[iLine] := data_storageTemp[16];
192             data_storage.temp_TC3[iLine] := data_storageTemp[17];
193             data_storage.temp_TC4[iLine] := data_storageTemp[18];
194             data_storage.temp_TC5[iLine] := data_storageTemp[19];
195             data_storage.temp_TC6[iLine] := data_storageTemp[20];
196             data_storage.temp_TC7[iLine] := data_storageTemp[21];
197             data_storage.temp_TC8[iLine] := data_storageTemp[22];
198             data_storage.temp_TC9[iLine] := data_storageTemp[23];
199             data_storage.temp_TC10[iLine] := data_storageTemp[24];
200
201             // up to 1000 values can be stored in the struct for each input before they are
202             // copied by the python script
203             // count up the number of measurements stored in the current Struct
204             iLine := iLine + 1;
205             // total number of values measured
206             nrValues := nrValues + 1;
207
208             IF NOT writeDataTrigger THEN
209                     endM := TRUE;
210             END_IF
211     END_IF
212
213     // when the data storage is fully filled trigger the python script and reset iLine
214     IF iLine > GVL_Constant.maxNbValues THEN
215
216             //count the current sample number
217             sampleNbr := sampleNbr+1;
218
219             // Open the data memory and write the measurement to it
220             iWriteData := SysSharedMemoryOpen2(pszName := MEMORY_NAME, ulPhysicalAddress := 0,
221                     pulSize := ADR(sizeData), pResult := ADR(resultOpenData));
222             iWriteData := SysSharedMemoryWrite(hShMemWriteData, 0, ADR(data_storage), sizeData,
223                     ADR(resultWriteData));
224             iWriteData := SysSharedMemoryClose(hShm := hShMemWriteData);
225
226             // change the flag value to 1
227             flag_data.flagData := 1;
228
229             // open the flag memory and write 1 to it --> trigger for the python script to get
230             // the measurementdata
231             iWriteFlags := SysSharedMemoryOpen2(pszName := FLAG_NAME, ulPhysicalAddress := 0,
232                     pulSize := ADR(sizeFlags), pResult := ADR(resultOpenFlags));
233             iWriteFlags := SysSharedMemoryWrite(hShMemFlags, 0, ADR(flag_data), sizeFlags,
234                     ADR(resultWriteFlags));
235             iWriteFlags := SysSharedMemoryClose(hShm := hShMemFlags);
236
237             // reset the line counter
238             iLine := 1;
239     END_IF
240
241     // end of the measurement: write the last data block and the meta data to the shared memory
242     IF NOT writeDataTrigger AND endM THEN
243             // create the meta data for the current measurement block
244             meta_data := CREATE_METADATA(sampleNumber := sampleNbr, nrMeasurementsInArray :=
245                     iLine-1, fileName := fullFileName, frequency := GVL.frequency);
```

```
246
247             // Open the metadata memory and write the metadata to it
248             iWriteMeta := SysSharedMemoryOpen2(pszName := META_NAME, ulPhysicalAddress := 0,
249                     pulSize := ADR(sizeMeta), pResult := ADR(resultOpenMeta));
250             iWriteMeta := SysSharedMemoryWrite(hShMemWriteMeta, 0, ADR(meta_data), sizeMeta,
251                     ADR(resultWriteMeta));
252             iWriteMeta := SysSharedMemoryClose(hShm := hShMemWriteMeta);
253
254             // Open the data memory and write the measurement to it
255             iWriteData := SysSharedMemoryOpen2(pszName := MEMORY_NAME, ulPhysicalAddress := 0,
256                     pulSize := ADR(sizeData), pResult := ADR(resultOpenData));
257             iWriteData := SysSharedMemoryWrite(hShMemWriteData, 0, ADR(data_storage), sizeData,
258                     ADR(resultWriteData));
259             iWriteData := SysSharedMemoryClose(hShm := hShMemWriteData);
260
261             // Open the flag memory for the end indicator and write 1 to it --> trigger reading
262             // of the shared memory from pythonscript 'memMap_Data.py'
263             flag_end.flagData := 1;
264             iWriteEnd := SysSharedMemoryOpen2(pszName := ENDMEASUREMENT_NAME,
265                     ulPhysicalAddress := 0, pulSize := ADR(sizeFlags),
266                     pResult := ADR(resultOpenEnd));
267             iWriteEnd := SysSharedMemoryWrite(hShMemEnd, 0, ADR(flag_end), sizeFlags,
268                     ADR(resultWriteEnd));
269             iWriteEnd := SysSharedMemoryClose(hShm := hShMemEnd);
270             flag_end.flagData := 0;
271
272             // reset all necessary booleans and variables
273             endM := FALSE;
274             sampleNbr := 1;
275             GVL.newFile := TRUE;
276             newMeasurement := TRUE;
277             iLine := 1;
278             GVL.currentRelTime := 0;
279             nrValues := 0;
280     END_IF
281     (*-----------------------------------------------------------------------------------------------*)
282
```

Anhang C12: Quellcode: PROGRAM „Setup_Measurement“

```
// Author: Dominik Müller
// Version: CODESYS V3.5 SP17 Patch 3
// Project: MatDatSys - Material Center Leoben
// Last modified: 06.02.2023

(* This PROGRAM sets the trigger for the pythonscript "startProcess.py" to start the wanted
measurement
It also writes the picoscope-setting to the shared memory *)

(*----------------------------------VARIABLE DECLERATION---------------------------------*)
PROGRAM Setup_Measurement
VAR
// variables to reset all trigger shared memories to 0
          iWriteFlags: __UXINT;
          flag_reset: Trigger_STRUCT;
          sizeF: UDINT := SIZEOF(Trigger_STRUCT);
          resultReset: RTS_IEC_RESULT;

          // variables for the visu and the process selection
          fStartProcessCheck: F_TRIG;
          fStartProcess: BOOL;
          startProcess: BOOL := FALSE;
          subProcessesStarted: BOOL := FALSE;
          getTypeMeasurement: INT := 0;         // acitve(=1) or passive(=2) mode
          typeMeasurementArray: ARRAY[0..2] OF STRING := ['not selected', 'active mode',
                    'passive mode'];
          getTypeLogging: INT := 0;    // which data should be captured

          // shared memory for starting the python parent process
          hShmParent: RTS_IEC_HANDLE := RTS_INVALID_HANDLE;
          resultOpenParent: RTS_IEC_RESULT;
          resultWriteParent: RTS_IEC_RESULT;
          flag_parent: Trigger_STRUCT;
          START_PARENT: STRING := '_CODESYS_MEMORY_Parent';        // python parent process
          iWriteParent: __UXINT;

          // shared memory for setting up the pico
          hShmPicoSETUP: RTS_IEC_HANDLE := RTS_INVALID_HANDLE;
          restultOpenPicoSetup: RTS_IEC_RESULT;
          restultWritePicoSetup: RTS_IEC_RESULT;
          pico_setup: Pico_Setup_STRUCT;
          sizePicoSetup: UDINT := SIZEOF(Pico_Setup_STRUCT);
          PICO_SETUP_NAME: STRING := '_CODESYS_MEMORY_PicoSetup';  // set up the pico
                                                                   // measurement
          iWritePicoSetup: __UXINT;
          resultWritePico: RTS_IEC_RESULT;

          // setup variables for the measurement
          // -- set the cycle time and get the new time
          setup: BOOL := FALSE;        // becomes TRUE if picoscope setting were changed via
                                       // the GUI

          f_SPS_PlantLogger: REAL := 100;// sampling frequenc of the SPS in Hz [0.01-100] Hz

          t_cycle: UDINT := REAL_TO_UDINT(1000000/f_SPS_PlantLogger);        // cycle time of
                                                        // the PlantLoggerTask in ns

          // -- variables for active mode
```

```
59          f_SG_a: REAL := 20; // signal generator frequenc (has to be put in manually at the
60                  // signal generator) [0.01-10] Hz
61          SF_SG_a: STRING := 'Dreieck'; // currently just for info in the metadata. Should
62                  // later be used to set up the SG
63          A_SG_a: REAL := 6; // amplitude of the SG [1-6] V_pp
64          N_samples_pico_a: UDINT := 1000000; // number datapoints per period of SG [1E3-1E6]
65          N_p_a: INT := 2;   // number of periods measured with pico [1-10]
66          N_EM_a: INT := 1;  // number of measurements done by pico [1-100]
67          deltaT_EM_a: REAL := 1;      // time between two pico measurements [0.05-1000] s
68          f_pico_a: REAL := UDINT_TO_REAL(N_samples_pico_a)*f_SG_a;// sampling rate picoscope
69                                                                   // [Hz]
70          t_pico_a: REAL := (1/f_SG_a)*INT_TO_REAL(N_p_a); // duration of one pico measurement
71
72          // -- variables for passive mode
73          f_SG_p: REAL := 30000; // generator frequence of the induction plant (has to be put
74                                 // in manually at the moment) 30 kHz
75          SF_SG_p: STRING; // signal form of the generator signal of the induction plant
76          A_SG_p: REAL := 6; // amplitude of the generator signal of the induction plant
77          N_samples_pico_p: UDINT := 1000; // number datapoints per period of SG [1E2-1E4]
78          N_p_p: INT := 2; // number of periods measured with pico [1-10]
79          N_EM_p: INT := 1000; // number of measurements done by pico [10-10000]
80          deltaT_EM_p: REAL := 0.05;   // time between two pico measurements [0.05-10] s
81          f_pico_p: REAL := UDINT_TO_REAL(N_samples_pico_p)*f_SG_p;// sampling rate picoscope
82                                                                   // [Hz]
83          t_pico_p: REAL := (1/f_SG_p)*INT_TO_REAL(N_p_p); // duration of one pico measurement
84
85          PicoSetup: Pico_Setup_STRUCT; // struct to store the picoscope setup data, this is
86                                 // written to the SM
87
88          plantLoggingActive: BOOL := FALSE;    // checks if a Plant measurement is wanted
89          picoLoggingActive: BOOL := FALSE;     // checks if a yoke measurement is wanted
90  END_VAR
```

```
1   (*-----------------------------------EXECUTED CODE-----------------------------------------*)
2   (*-----------------------------------------------------------------------------------------*)
3                               (*reset the trigger shared memories*)
4   IF GVL.resetShms THEN
5           flag_reset.flagData := 0;
6           iWriteFlags := SysSharedMemoryOpen2(pszName := Plant_Logger.FLAG_NAME,
7                   ulPhysicalAddress := 0, pulSize := ADR(sizeF),
8                   pResult := ADR(resultReset));
9           iWriteFlags := SysSharedMemoryWrite(hShm := Plant_Logger.hShMemFlags, ulOffset := 0,
10                  pbyData := ADR(flag_reset), ulSize := sizeF, pResult := ADR(resultReset));
11          iWriteFlags := SysSharedMemoryClose(hShm := Plant_Logger.hShMemFlags);
12
13          iWriteFlags := SysSharedMemoryOpen2(pszName := Plant_Logger.ENDMEASUREMENT_NAME,
14                  ulPhysicalAddress := 0, pulSize := ADR(sizeF),
15                  pResult := ADR(resultReset));
16          iWriteFlags := SysSharedMemoryWrite(Plant_Logger.hShMemEnd, 0, ADR(flag_reset),
17                  sizeF, ADR(resultReset));
18          iWriteFlags := SysSharedMemoryClose(hShm := Plant_Logger.hShMemEnd);
19
20          iWriteFlags := SysSharedMemoryOpen2(pszName := START_PARENT, ulPhysicalAddress := 0,
21                  pulSize := ADR(sizeF), pResult := ADR(resultReset));
22          iWriteFlags := SysSharedMemoryWrite(hShm := hShmParent, 0, ADR(flag_reset), sizeF,
23                  ADR(resultReset));
24          iWriteFlags := SysSharedMemoryClose(hShm := hShmParent);
25
26          iWriteFlags := SysSharedMemoryOpen2(pszName := Pico_Logger.PICO_FLAG_NAME,
27                  ulPhysicalAddress := 0, pulSize := ADR(sizeF),
28                  pResult := ADR(resultReset));
29          iWriteFlags := SysSharedMemoryWrite(Pico_Logger.hShMemPicoTrigger, 0,
```

```
30                  ADR(flag_reset), sizeF, ADR(resultReset));
31          iWriteFlags := SysSharedMemoryClose(hShm := Pico_Logger.hShMemPicoTrigger);
32
33          PicoSetup.setUpTrigger := FALSE;
34          iWritePicoSetup := SysSharedMemoryOpen2(pszName := PICO_SETUP_NAME,
35                  ulPhysicalAddress := 0, pulSize := ADR(sizePicoSetup),
36                  pResult := ADR(resultWritePico));
37          iWritePicoSetup := SysSharedMemoryWrite(hShm := hShmPicoSETUP, ulOffset := 0,
38                  pbyData := ADR(PicoSetup), ulSize := sizePicoSetup,
39                  pResult := resultWritePico);
40          iWritePicoSetup := SysSharedMemoryClose(hShm := hShmPicoSETUP);
41
42          GVL.resetShms := FALSE;
43  END_IF
44  (*-------------------------------------------------------------------------------------------*)
45
46  (*-------------------------------------------------------------------------------------------*)
47  (*create the shared memories for triggering the python parent script and the pico setup*)
48
49  IF hShmParent = RTS_INVALID_HANDLE THEN
50          hShmParent := SysSharedMemoryCreate(pszName := START_PARENT, ulPhysicalAddress := 0,
51                  pulSize := ADR(sizeF), pResult := ADR(iWriteParent));
52  END_IF
53
54  IF hShmPicoSETUP = RTS_INVALID_HANDLE THEN
55          hShmPicoSETUP := SysSharedMemoryCreate(pszName := PICO_SETUP_NAME,
56                  ulPhysicalAddress := 0, pulSize := ADR(sizePicoSetup) ,
57                  pResult := ADR(restultOpenPicoSetup));
58          PicoSetup.A_SG := A_SG_a;
59          PicoSetup.deltaT_EM := deltaT_EM_a;
60          PicoSetup.f_pico := f_pico_a;
61          PicoSetup.f_RevPi := f_SPS_PlantLogger;
62          PicoSetup.f_SG := f_SG_a;
63          PicoSetup.N_EM := N_EM_a;
64          PicoSetup.N_p := N_p_a;
65          PicoSetup.N_samples_pico := N_samples_pico_a;
66          PicoSetup.setUpTrigger := TRUE;
67          PicoSetup.SF_SG := STRING_TO_ARRAYBYTE(SF_SG_a);
68          PicoSetup.t_pico := t_pico_a;
69
70          iWritePicoSetup := SysSharedMemoryOpen2(pszName := PICO_SETUP_NAME,
71                  ulPhysicalAddress := 0, pulSize := ADR(sizePicoSetup),
72                  pResult := ADR(resultWritePico));
73          iWritePicoSetup := SysSharedMemoryWrite(hShm := hShmPicoSETUP, ulOffset := 0,
74                  pbyData := ADR(PicoSetup), ulSize := sizePicoSetup,
75                  pResult := resultWritePico);
76          iWritePicoSetup := SysSharedMemoryClose(hShm := hShmPicoSETUP);
77  END_IF
78  (*-------------------------------------------------------------------------------------------*)
79
80  (*-------------------------------------------------------------------------------------------*)
81  (*write the setup data for the pico to the shared memory and if needed change the cycletime.*)
82
83  IF setup THEN
84          // set the cycle time
85          t_cycle:= REAL_TO_UDINT(1000000/f_SPS_PlantLogger);
86          SysTask.SysTaskSetInterval(Plant_Logger.hPlant_LoggerTask, t_cycle);
87          GVL.frequency := UDINT_TO_INT(1000000/t_cycle);
88
89          // write a setup struct to the shared memory for the pico setup. The variables are
90          // changed according to the input in the GUI
91          // setup for active mode
```

```
92          IF getTypeMeasurement = 1 THEN
93                  PicoSetup.A_SG := A_SG_a;
94                  PicoSetup.deltaT_EM := deltaT_EM_a;
95                  PicoSetup.f_pico := f_pico_a;
96                  PicoSetup.f_RevPi := f_SPS_PlantLogger;
97                  PicoSetup.f_SG := f_SG_a;
98                  PicoSetup.N_EM := N_EM_a;
99                  PicoSetup.N_p := N_p_a;
100                 PicoSetup.N_samples_pico := N_samples_pico_a;
101                 PicoSetup.setUpTrigger := TRUE;
102                 PicoSetup.SF_SG := STRING_TO_ARRAYBYTE(SF_SG_a);
103                 PicoSetup.t_pico := t_pico_a;
104
105                 iWritePicoSetup := SysSharedMemoryOpen2(pszName := PICO_SETUP_NAME,
106                         ulPhysicalAddress := 0, pulSize := ADR(sizePicoSetup),
107                         pResult := ADR(resultWritePico));
108                 iWritePicoSetup := SysSharedMemoryWrite(hShm := hShmPicoSETUP,
109                         ulOffset := 0, pbyData := ADR(PicoSetup),
110                         ulSize := sizePicoSetup, pResult := resultWritePico);
111                 iWritePicoSetup := SysSharedMemoryClose(hShm := hShmPicoSETUP);
112                 setup := FALSE;
113
114         // setup for passive mode
115         ELSIF getTypeMeasurement = 2 THEN
116                 PicoSetup.A_SG := A_SG_p;
117                 PicoSetup.deltaT_EM := deltaT_EM_p;
118                 PicoSetup.f_pico := f_pico_p;
119                 PicoSetup.f_RevPi := f_SPS_PlantLogger;
120                 PicoSetup.f_SG := f_SG_p;
121                 PicoSetup.N_EM := N_EM_p;
122                 PicoSetup.N_p := N_p_p;
123                 PicoSetup.N_samples_pico := N_samples_pico_p;
124                 PicoSetup.setUpTrigger := TRUE;
125                 PicoSetup.SF_SG := STRING_TO_ARRAYBYTE(SF_SG_p);
126                 PicoSetup.t_pico := t_pico_p;
127
128                 iWritePicoSetup := SysSharedMemoryOpen2(pszName := PICO_SETUP_NAME,
129                         ulPhysicalAddress := 0, pulSize := ADR(sizePicoSetup),
130                         pResult := ADR(resultWritePico));
131                 iWritePicoSetup := SysSharedMemoryWrite(hShm := hShmPicoSETUP,
132                         ulOffset := 0, pbyData := ADR(PicoSetup),
133                         ulSize := sizePicoSetup, pResult := resultWritePico);
134                 iWritePicoSetup := SysSharedMemoryClose(hShm := hShmPicoSETUP);
135
136                 setup := FALSE;
137         END_IF
138 END_IF
139 (*-----------------------------------------------------------------------------------------*)
140
141 (*-----------------------------------------------------------------------------------------*)
142 (* START THE WANTED MEASUREMNT SCRIPT: flag_parent.writeFlagData := 1 - both measurement, 2 -
143 only plant or 3 - only yoke. See startProcess.py*)
144
145 IF startProcess  THEN
146         // check if a measurement was selected, otherwise reset startProcess
147         IF getTypeLogging = 0 THEN
148                 startProcess := FALSE;
149                 RETURN;
150         END_IF
151
152         // if a measurement was selected, start the corresponding child-pythonscripts via an
153         // indicator using SM
```

```
154             IF NOT subProcessesStarted THEN
155                     flag_parent.flagData := getTypeLogging; // 1: yoke and plant, 2:
156                                             // only plant, 3: only yoke
157                     iWriteParent := SysSharedMemoryOpen2(pszName := START_PARENT,
158                             ulPhysicalAddress := 0, pulSize := ADR(sizeF),
159                             pResult := ADR(resultWriteParent));
160                     iWriteParent := SysSharedMemoryWrite(hShm := hShmParent, 0,
161                             ADR(flag_parent), sizeF, ADR(resultWriteParent));
162                     iWriteParent := SysSharedMemoryClose(hShm := hShmParent);
163                     subProcessesStarted := TRUE;
164             END_IF
165
166             // check if the measurement is done
167             iWriteParent := SysSharedMemoryOpen2(pszName := START_PARENT,
168                     ulPhysicalAddress := 0, pulSize := ADR(sizeF),
169                     pResult := ADR(resultWriteParent));
170             iWriteParent := SysSharedMemoryRead(hShm := hShmParent, 0, ADR(flag_parent), sizeF,
171                     ADR(resultWriteParent));
172             iWriteParent := SysSharedMemoryClose(hShm := hShmParent);
173             IF flag_parent.flagData = 4 THEN      // trigger that measurement is done
174                     subProcessesStarted := FALSE;
175                     startProcess := FALSE;
176                     flag_parent.flagData := 0;
177                     iWriteParent := SysSharedMemoryOpen2(pszName := START_PARENT,
178                             ulPhysicalAddress := 0, pulSize := ADR(sizeF),
179                             pResult := ADR(resultWriteParent));
180                     iWriteParent := SysSharedMemoryWrite(hShm := hShmParent, 0,
181                             ADR(flag_parent), sizeF, ADR(resultWriteParent));
182                     iWriteParent := SysSharedMemoryClose(hShm := hShmParent);
183             END_IF
184     END_IF
185
186     // in case startProcess is put to false manually then reset everything necessary
187      fStartProcessCheck(CLK := startProcess);
188      fStartProcess := fStartProcessCheck.Q;
189     IF fStartProcess THEN
190             subProcessesStarted := FALSE;
191             GVL.resetShms := TRUE;
192     END_IF
```

Anhang C13: Quellcode: PROGRAM „Shutdown“

```
// Author: Dominik Müller
// Version: CODESYS V3.5 SP17 Patch 3
// Project: MatDatSys - Material Center Leoben
// Last modified: 06.02.2023

(*This PROGRAM is used to shut down the RevPi via a console command activated via the
"Shutdown"-button on the GUI
Not directly neccessary in this form, but convenient for testing purposes*)

(*-----------------------------------VARIABLE DECLERATION---------------------------------*)
PROGRAM Shutdown
VAR
          dwCopySize: DWORD;
          dutResult : RTS_IEC_RESULT;
          szCommand : STRING(200);
          fbTimer : TON;
          szStdOout : STRING(1000);
          fbTimer1 : TON;
          Command : STRING(200);
END_VAR
```

```
(*-------------------------------------EXECUTED CODE-----------------------------------------*)
GVL.shutdown := FALSE;

SysProcess_Implementation.SysProcessExecuteCommand2(pszCommand:='sudo halt'
                                    , pszStdOut:=szStdOout
                                    , udiStdOutLen:= SIZEOF(szStdOout)
                                    , pResult := ADR(dutResult));
```

Anhang C14: Quellcode der globalen Variablen

„GVL“:

```
{attribute 'qualified_only'}
VAR_GLOBAL
// Author: Dominik Müller
// Version: CODESYS V3.5 SP17 Patch 3
// Project: MatDatSys - Material Center Leoben
// Last modified: 06.02.2023

          (* GLOBAL VARIABLES FOR DATA MEASUREMENT *)

          temp_TC1: REAL; // Temperature from TC1
          temp_TC2: REAL;    // Temperature from TC2
          temp_TC3: REAL; // Temperature from TC3
          temp_TC4: REAL; // Temperature from TC4
          temp_TC5: REAL; // Temperature from TC5
          temp_TC6: REAL; // Temperature from TC6
          temp_TC7: REAL; // Temperature from TC7
          temp_TC8: REAL; // Temperature from TC8
          temp_TC9: REAL; // Temperature from TC9
          temp_TC10: REAL; // Temperature from TC10

          Q_Brause: REAL; // Durchfluss Brause [L/min]

          p_Brause: REAL; // Druck Brause [bar]

          p_Gas: REAL;       // Druck Gas [bar]

          Q_Gasbrause1: REAL;          // Durchfluss Gasbrause 1 [L/min]
          p_Gasbrause1: REAL;          // Druck Gasbrause 1 [bar]

          Q_Gasbrause2: REAL;          // Durchfluss Gasbrause 2 [L/min]
          p_Gasbrause2: REAL;          // Druck Gasbrause 2 [bar]]

          n_Waermeregelung: REAL; // Istwert Drehantrieb Wärmeregelung [%]

          T_ind_zulauf: REAL;          // Induktor Kühlung Zulauf PT100 [C°]
          T_ind_rueck: REAL;           // Induktor Kühlung Rücklauf PT100 [C°]
          T_abschreck: REAL;           // Temperatur Abschreckmitteltank PT100 [C°]
          T_Brause: REAL;              // Temperatur Brause PT100 [C°]
          T_Gas: REAL;                 // Temperatur Gas PT100 [C°]

          newFile: BOOL := TRUE;       // check if a new measurement was started

          startTime: ULINT;            // start time of the measurement as UTC-time

          currentRelTime: REAL;        // current system time as UTC-time

          startmemMap_Data: BOOL := FALSE;      // global variables for pico trigger

          resetShms: BOOL := FALSE;    // reset the trigger values in the shared memories to
                                       // standy mode

          frequency: INT;              // current frequency of the PlantLoggerTask

          shutdown: BOOL := FALSE;     // shutdown via interface
END_VAR
```

„GVL_Constant“:

```
{attribute 'qualified_only'}

VAR_GLOBAL CONSTANT
// Author: Dominik Müller
// Version: CODESYS V3.5 SP17 Patch 3
// Project: MatDatSys - Material Center Leoben
// Last modified: 06.02.2023

          (* GLOBAL CONSTANT VARIABLES FOR DATA MEASUREMENT *)

          maxNbValues: INT := 1000; // the number of plant-measurments taken before a STRUCT
is pushed to the shared memory. A constant is used for convenience if the size should be
changed.
END_VAR
```

„GVL_SetupPico“:

```
{attribute 'qualified_only'}
VAR_GLOBAL
// Author: Dominik Müller
// Version: CODESYS V3.5 SP17 Patch 3
// Project: MatDataSys - Material Center Leoben
// Last modified: 15.02.2023

(*variables to safe the picoscope-setting inbetween setup. They are then written to the local
Setup_Measurement variables.*)

          //for active mode
          f_RevPi: REAL := 50; // frequency of the RevPi cycle [0.01 - 100] Hz
          f_SG: REAL := 0.05; // signal generator induction plant (has to be put in manually
                              // at the signal generator) [0.01-10] Hz
          SF_SG: STRING := 'Dreieck'; // signalform signalgenerator
          A_SG: REAL := 6; // amplitude of the SG [1-6] V_pp
          N_samples_pico: UDINT := 1000000; // number datapoints captured by pico per period
                                            // of SG [1E3-1E6]
          N_p: INT := 2; // number of periods measured with pico [1-10]
          N_EM: INT := 1;  // number of measurements done by pico            [1-100]
          deltaT_EM: REAL := 1; // time between two pico measurements [0.05-1000] s
          f_pico: REAL := UDINT_TO_REAL(N_samples_pico)*f_SG; // sampling rate picoscope [Hz]
          t_pico: REAL := (1/f_SG)*INT_TO_REAL(N_p); // duration of one pico measurement [s]

          //for passive mode
          f_RevPi_p: REAL := 50; // frequency of the RevPi cycle [0.01 - 100] Hz
          f_SG_p: REAL;  // signal generator frequenc (has to be put in manually at the signal
                         // generator) [0.01-10] Hz
          SF_SG_p: STRING; // signalform signalgenerator (not used in passive mode
          A_SG_p: REAL; // amplitude of the SG [1-6] V_pp
          N_samples_pico_p: UDINT := 1000; // number datapoints capurted by pico per period of
                                           // SG [1E2-1E4]
          N_p_p: INT := 2;  // number of periods measured with pico [1-10]
          N_EM_p: INT := 1000; // number of measurements done by pico [10-10000]
          deltaT_EM_p: REAL := 0.05; // time between two pico measurements   [0.05-10] s
          f_pico_p: REAL := UDINT_TO_REAL(N_samples_pico_p)*f_SG_p;// sampling rate picoscope
                                                                   // [Hz]
          t_pico_p: REAL := (1/f_SG_p)*INT_TO_REAL(N_p_p); // duration of one pico measurement
END_VAR
```

Anhang C15: Quellcode der STRUCTs

Data_Storage_STRUCT:

```
// Author: Dominik Müller
// Version: CODESYS V3.5 SP17 Patch 3
// Project: MatDatSys - Material Center Leoben
// Last modified: 06.02.2023

(* A STRUCT that contains all the inductionplant and TC measurement values that are written to
a shared memory to be read by a python script to save it in HDF5 format. For every measurement
value there is an array that can store GVL_Constant.maxNbValues. This determines the size of
one datablock to be processed via the shared memory.
If you cange something here you have to change python accordingly*)

TYPE Data_Storage_STRUCT :
STRUCT
          mTime: ARRAY[1..GVL_Constant.maxNbValues] OF REAL;
          Q_Brause: ARRAY[1..GVL_Constant.maxNbValues] OF REAL;
          p_Brause: ARRAY[1..GVL_Constant.maxNbValues] OF REAL;
          p_Gas: ARRAY[1..GVL_Constant.maxNbValues] OF REAL;
          Q_Gasbrause1: ARRAY[1..GVL_Constant.maxNbValues] OF REAL;
          p_Gasbrause1: ARRAY[1..GVL_Constant.maxNbValues] OF REAL;
          Q_Gasbrause2: ARRAY[1..GVL_Constant.maxNbValues] OF REAL;
          p_Gasbrause2: ARRAY[1..GVL_Constant.maxNbValues] OF REAL;
          n_Waermeregelung: ARRAY[1..GVL_Constant.maxNbValues] OF REAL;
          T_ind_zulauf: ARRAY[1..GVL_Constant.maxNbValues] OF REAL;
          T_ind_rueck: ARRAY[1..GVL_Constant.maxNbValues] OF REAL;
          T_abschreck: ARRAY[1..GVL_Constant.maxNbValues] OF REAL;
          T_Brause: ARRAY[1..GVL_Constant.maxNbValues] OF REAL;
          T_Gas: ARRAY[1..GVL_Constant.maxNbValues] OF REAL;
          temp_TC1: ARRAY[1..GVL_Constant.maxNbValues] OF REAL;
          temp_TC2: ARRAY[1..GVL_Constant.maxNbValues] OF REAL;
          temp_TC3: ARRAY[1..GVL_Constant.maxNbValues] OF REAL;
          temp_TC4: ARRAY[1..GVL_Constant.maxNbValues] OF REAL;
          temp_TC5: ARRAY[1..GVL_Constant.maxNbValues] OF REAL;
          temp_TC6: ARRAY[1..GVL_Constant.maxNbValues] OF REAL;
          temp_TC7: ARRAY[1..GVL_Constant.maxNbValues] OF REAL;
          temp_TC8: ARRAY[1..GVL_Constant.maxNbValues] OF REAL;
          temp_TC9: ARRAY[1...GVL_Constant.maxNbValues] OF REAL;
          temp_TC10: ARRAY[1..GVL_Constant.maxNbValues] OF REAL;
END_STRUCT
END_TYPE
```

Meta_Data_STRUCT:

```
// Author: Dominik Müller
// Version: CODESYS V3.5 SP17 Patch 3
// Project: MatDatSys - Material Center Leoben
// Last modified: 06.02.2023

(* A STRUCT that contains the relevant metadata for the plat measurement.
If you cange something here you have to change python accordingly*)

TYPE Meta_Data_STRUCT :
STRUCT
          nrMeasurementsInArray: INT;  // number of measurements in the current
                    // Data_Storage_STRUCT array-->needed for computation in python
          sampleNumber: INT;           // number of Data_Storage_STRUCTs written in total
                                       // -->needed for computation in python
          fileName: ARRAY[1..80] OF BYTE;       // filename as an array of bytes, this is as
                                                // well the starting time of the measurement
```

```
            frequency: INT;    // measurement frequency (frequency of the Task PlantLoggerTask)
            startTime: ULINT;  // start time of the plant measurement
END_STRUCT
END_TYPE
```

Pico_Setup_STRUCT:

```
// Author: Dominik Müller
// Version: CODESYS V3.5 SP17 Patch 3
// Project: MatDatSys - Material Center Leoben
// Last modified: 06.02.2023

(* A STRUCT that contains all neccessary parameters to set up the picoscope for the wanted
measurement.
This is written to a shared memory to be read by a python script that then communicates with
the picoscope. The same struct can be used for active and passive measurements
If you cange something here you have to change python accordingly*)

TYPE Pico_Setup_STRUCT :
STRUCT
          f_RevPi: REAL := 100;        // frequency of the Plantlogging cycle [0.01 - 100] Hz

          f_SG: REAL := 0.05;          // frequency signal generator induction plant (has to be
                              // put in manually at the signal generator) [0.01-10] Hz
          SF_SG: ARRAY[1..10] OF BYTE; // currently just for info in the metadata. Should
                    // later be used to set up the SG. This is an "array of byte" since it's
                    // easier to share it with the python script in that way
          A_SG: REAL := 6; // amplitude of the SG [1-6] V_pp
          N_samples_pico: UDINT := 1000000; // number datapoints per period of SG [1E2-1E4]
          N_p: INT := 2;     // number of periods measured with pico [1-10]
          N_EM: INT := 1;    // number of measurements done by pico [10-10000]
          deltaT_EM: REAL := 1;        // time between two pico measurements [0.05-10] s
          f_pico: REAL := UDINT_TO_REAL(N_samples_pico)*f_SG; // sampling rate picoscope [Hz]
          t_pico: REAL := (1/f_SG)*INT_TO_REAL(N_p); // duration of one pico measurement

          setUpTrigger: BOOL := FALSE; //indicator for the pythonscript if the picoscope
                                       // settings were changed
END_STRUCT
END_TYPE
```

Trigger_STRUCT:

```
// Author: Dominik Müller
// Version: CODESYS V3.5 SP17 Patch 3
// Project: MatDatSys - Material Center Leoben
// Last modified: 06.02.2023

(* INT: unsigned integer, size: 4 bytes
This STRUCT contains an INT that is used as a Triggervariable.
This struct is used for every trigger shared memory created in this project.
If you change something here you have to change python accordingly *)

TYPE Trigger_STRUCT :
STRUCT
          flagData: INT;
END_STRUCT
END_TYPE
```

Anhang C16: Quellcode VISU

Visu_main: yoke logging active (OnValueChanged):

```
// Author: Dominik Müller
// Version: CODESYS V3.5 SP17 Patch 3
// Project: MatDatSys - Material Center Leoben
// Last modified: 14.02.2023

(*Logic for the button "yoke logging active"*)

IF Setup_Measurement.plantLoggingActive AND NOT Setup_Measurement.picoLoggingActive THEN
          Setup_Measurement.getTypeLogging := 2;
ELSIF NOT Setup_Measurement.plantLoggingActive AND Setup_Measurement.picoLoggingActive THEN
          Setup_Measurement.getTypeLogging := 3;
ELSIF Setup_Measurement.plantLoggingActive AND Setup_Measurement.picoLoggingActive THEN
          Setup_Measurement.getTypeLogging := 1;
ELSIF NOT Setup_Measurement.plantLoggingActive AND NOT Setup_Measurement.picoLoggingActive
THEN
          Setup_Measurement.getTypeLogging := 0;
END_IF
```

Visu_main: plant logging active (OnValueChanged):

```
// Author: Dominik Müller
// Version: CODESYS V3.5 SP17 Patch 3
// Project: MatDatSys - Material Center Leoben
// Last modified: 06.02.2023

(*Logic for the button "plant logging active"*)

IF Setup_Measurement.plantLoggingActive AND NOT Setup_Measurement.picoLoggingActive THEN
          Setup_Measurement.getTypeLogging := 2;
ELSIF NOT Setup_Measurement.plantLoggingActive AND Setup_Measurement.picoLoggingActive THEN
          Setup_Measurement.getTypeLogging := 3;
ELSIF Setup_Measurement.plantLoggingActive AND Setup_Measurement.picoLoggingActive THEN
          Setup_Measurement.getTypeLogging := 1;
ELSIF NOT Setup_Measurement.plantLoggingActive AND NOT Setup_Measurement.picoLoggingActive
THEN
          Setup_Measurement.getTypeLogging := 0;
END_IF
```

activeMode: save settings (OnMouseClick):

```
// Author: Dominik Müller
// Version: CODESYS V3.5 SP17 Patch 3
// Project: MatDatSys - Material Center Leoben
// Last modified: 06.02.2023

(*Logic for the button "save setting" in active mode*)

Setup_Measurement.f_SG_a := GVL_SetupPico.f_SG; Setup_Measurement.f_SPS_PlantLogger :=
GVL_SetupPico.f_RevPi;
IF GVL_SetupPico.SF_SG = 'Dreieck' OR GVL_SetupPico.SF_SG = 'Sinus' THEN
          Setup_Measurement.SF_SG_a := GVL_SetupPico.SF_SG;
END_IF
Setup_Measurement.A_SG_a := GVL_SetupPico.A_SG;
Setup_Measurement.N_samples_pico_a := GVL_SetupPico.N_samples_pico;
Setup_Measurement.N_p_a := GVL_SetupPico.N_p;
Setup_Measurement.N_EM_a := GVL_SetupPico.N_EM;
```

```
17  Setup_Measurement.deltaT_EM_a := GVL_SetupPico.deltaT_EM;
18  Setup_Measurement.f_pico_a :=
19           UDINT_TO_REAL(Setup_Measurement.N_samples_pico_a)*Setup_Measurement.f_SG_a;
20  Setup_Measurement.t_pico_a :=
21            (1/Setup_Measurement.f_SG_a)*INT_TO_REAL(Setup_Measurement.N_p_a);
22  Setup_Measurement.setup := TRUE;
```

passiveMode: save settings (OnMouseClick):

```
// Author: Dominik Müller
// Version: CODESYS V3.5 SP17 Patch 3
// Project: MatDatSys - Material Center Leoben
// Last modified: 06.02.2023

(*Logic for the button “save setting” in passive mode*)

Setup_Measurement.f_SG_p := GVL_SetupPico.f_SG_p;
Setup_Measurement.f_SPS_PlantLogger := GVL_SetupPico.f_RevPi;
IF GVL_SetupPico.SF_SG_p = 'Dreieck' OR GVL_SetupPico.SF_SG_p = 'Sinus' THEN
          Setup_Measurement.SF_SG_p := GVL_SetupPico.SF_SG_p;
END_IF
Setup_Measurement.A_SG_p := GVL_SetupPico.A_SG_p;
Setup_Measurement.N_samples_pico_p := GVL_SetupPico.N_samples_pico_p;
Setup_Measurement.N_p_p := GVL_SetupPico.N_p_p;
Setup_Measurement.N_EM_p := GVL_SetupPico.N_EM_p;
Setup_Measurement.deltaT_EM_p := GVL_SetupPico.deltaT_EM_p;
Setup_Measurement.f_pico_p :=
          UDINT_TO_REAL(Setup_Measurement.N_samples_pico_p)*Setup_Measurement.f_SG_p;
Setup_Measurement.t_pico_p :=
          (1/Setup_Measurement.f_SG_p)*INT_TO_REAL(Setup_Measurement.N_p_p);
Setup_Measurement.setup := TRUE;
```

Anhang D: Datenblätter

Dem nachfolgenden Anhang können die Datenblätter der verwendeten Hardwarekomponenten entnommen werden.

Anhang D1: RevPi Connect+ feat. CODESYS

Das Datenblatt des „Revolution Pi Connect+ feat. Codesys“ kann Anhang D1 entnommen werden [48].

REVOLUTION PI

RevPi Connect+
feat. CODESYS
Article No.: 100337

Technical Data

Housing dimensions (H x W x D)	96 x 45 x 110.5 mm
Housing type	DIN rail housing (for DIN rail version EN 50022)
Housing material	Polycarbonate
Weight	approx. 197 g / 224 g (incl. connectors)
IP Code	IP20
Power supply	12-24 V DC -15 % / +20 %, reverse-polarity protected
Max. power consumption	20 Watt (incl. 1 A total USB output current) [1]
Operating temperature	-25 °C....+55 °C
Storage temperature	-40 °C....+85 °C
Humidity (at 40 °C)	93 % (non-condensing)
Interfaces	2 x USB A (Total current draw from both sockets max. 1 A) [2] 2 x RJ45 10/100 Ethernet (using separate MAC addresses) 1 x RS485 screw-type terminal 1 x Micro-USB (solely for image transfer to eMMC) 1 x Micro HDMI 1 x PiBridge system bus 1 x ConBridge system bus
Connectors	1 x 4-pole screw-type terminal for relay contact and signal input 1 x 4-pole screw-type terminal for power supply
Processor	Broadcom BCM2837B0 quad-core ARM Cortex A53
Clock rate	1.2 GHz
Processor cooling	Passive with heat sink
RAM	1 GB
Flash memory	16 GB

[1] The average power consumption without USB load varies greatly and depends on the use of the interfaces, the CPU and the CPU. It is usually well below 4 Watts without HDMI.

[2] 1 A USB output current (total of both USB outputs) is only available for input voltages >11 V. The bridging time of at least 10 ms required by EN 61131-2 is only guaranteed with a 20.4 to 28.8 V power supply. With a 12 V power supply, this time is significantly reduced, especially when power is drawn from the USB ports.

REVOLUTION PI

RevPi Connect+
feat. CODESYS
Article No.: 100337

Technical Data

Number of digital input channels	1
Input type	24 V control voltage (e.g. for power-good signal of a UPS)
Input thresholds	approx. 3.0 V (0 -> 1) / 2.3 V (1 -> 0)
Input protection	against overvoltage, negative voltages
Number of digital output channels	1
Output type	Relay contact, approval up to 30 V switching voltage (e.g. for power supply of a router)
Maximum current load of the contact	2 A @ 30 V DC (resistive load!)
Software integration of input and output	Via GPIOs and process image. Output is optionally switched by hardware watchdog.
Hardware watchdog functionality	Can be disabled by bridging the 4-pole screw-type terminal. Reset by toggling a GPIO or alternatively a bit in the process image.
Hardware watchdog intervall	Trigger after approx. 60 seconds without toggling the reset bit.
Compatible modules for system expansion	All RevPi IO modules and RevPi Gate modules can be connected via the PiBridge system bus. Various transceiver modules can be connected via the ConBridge system bus.
ESD protection	4 kV / 8 kV (according to EN 61131-2 and IEC 61000-6-2)
EMI tests	Passed (according to EN 61131-2 and IEC 61000-6-2)
Surge/Burst tests	Passed (according to EN 61131-2 and IEC 61000-6-2)
Buffer time RTC	min. 24 h
Optical indicator	6 status LEDs (bi-color), two of them freely programmable
CODESYS runtime	Multicore CODESYS Control runtime pre-installed & licensed.
RoHS conformity	Yes
CE conformity	Yes
UL certification	Yes, UL-File-No. E494534 Note: The device may only be supplied from circuits that comply with Class 2 or Safety Extra Low Voltage (SELV) according to Class 9.4 of UL 61010-1.

Anhang D2: RevPi AIO

Der Anhang D2 beinhaltet das Datenblatt des verwendeten AIO-Moduls [48].

REVOLUTION PI

RevPi AIO

Artikelnr.: 100250

Technische Daten

Norm	EN 61131-2
Gehäuseabmessungen (H x B x T)	96 x 22,5 x 110,5 mm
Gehäusevariante	Hutschienengehäuse (Für Hutschienenvariante EN 50022)
Gehäusematerial	Kunststoff
Gewicht	ca. 115 g
Schutzart	IP20
Spannungsversorgung	12 - 24 V (-15 %/+20 %)
Stromaufnahme	max. 200 mA bei 24 V (Volllast) max. 400 mA bei 12 V (Volllast) max. 500 mA im Anlauf
Zulässige Betriebstemperatur	-30...+55 °C
Zulässige Lagertemperatur	-40...+85 °C
Max. relative Luftfeuchtigkeit (bei 40 °C)	93 % (keine Betauung)
Spannungsmessbereiche	±10 V | ±5 V | 0...10 V | 0...5 V
Stromeingangsbereiche	0...20 mA | 0...24 mA | 4...20 mA | ±25 mA
Temperatureingangsbereich	-200...+850 °C
Ausgangsspannungsbereiche	±10 V | ±11 V | ±5 V | ±5,5 V | 0...10 V | 0...11 V | 0...5 V | 0...5,5 V
Ausgangsstrombereiche	0...20 mA | 0...24 mA | 4...20 mA
Anzahl der Eingangskanäle davon für Spannung davon für Strom davon für RTDs (Pt100/Pt1000)	6 max. 4 max. 4 2
Anzahl der Ausgangskanäle davon für Spannung davon für Strom	2 max. 2 max. 2
Galvanische Trennung Eingänge untereinander Eingänge gegen Ausgänge Ausgänge untereinander Systembus gegen Eingänge/Ausgänge	 Nein Ja Nein Ja
Eingangstyp Spannung/Strom Temperatursensor (RTD)	 differential 2-, 3-, 4-Draht
Ausgangstyp	single ended, common ground, kurzschlussfest
ADC Typ	24 Bit ΔΣ
DAC Typ	16 Bit

REVOLUTION PI

RevPi AIO

Artikelnr.: 100250

Technische Daten

Eingangsauflösung Prozessabbild Spannung Strom Temperatur	 1 mV 1 µA 0,1 K
Ausgangsauflösung Prozessabbild Spannung Strom	 1 mV 1 µA
Max. Gesamteingangsfehler (bei 25 °C Umgebungstemperatur) Spannung (alle Eingangsbereiche) Strom (alle Eingangsbereiche) Temperatur (kompletter Bereich)	 ±10 mV (±5 mV @ 0…5 V Bereich) ±20 µA (±24 µA @ 0…24 µA Bereich) ±0,5 K
Max. Gesamteingangsfehler (bei -30…+55 °C Umgebungstemperatur) Spannung (alle Eingangsbereiche) Strom (alle Eingangsbereiche) Temperatur (kompletter Bereich)	 ±10 mV ±72 µA ±1,5 K
Max. Gesamtausgangsfehler (bei 25 °C Umgebungstemperatur) Spannung (alle Eingangsbereiche) Strom (alle Eingangsbereiche)	 ±15 mV ±20 µA
Max. Gesamtausgangsfehler (bei -30…+55 °C Umgebungstemperatur) Spannung (alle Eingangsbereiche) Strom (alle Eingangsbereiche)	 ±15 mV ±72 µA
Wandlungszeit Eingänge (Datenrate im Prozessabbild)	8…1000 ms (einstellbar)
Ausgangsdatenrate	1 PiBridge Zyklus
Flankensteilheit Ausgang Einstellbare digital Flanke	 1 LSB@3,3 kHz bis 128 LSB@258 kHz
Eingangsimpedanz Spannung Strom	 >900 kΩ <250 Ω
Ausgangsimpedanz Spannung maximale kapazitive Last	 <16 Ω 5 nF @ 1 kΩ
Max. Lastwiderstand bei Stromausgang	600 Ω
Min. Lastwiderstand bei Spannungsausgang	1 kΩ
Weitere Eigenschaften	Alle Ein- und Ausgänge können linear skaliert werden Übertemperaturüberwachung Überstromüberwachung Bereichsüberwachung
Optische Anzeige	3 Status LEDs (2-farbig)
Konformität	CE, RoHS
UL-Zertifizierung	Ja, UL-File-Nr. E494534 Hinweis: Das Gerät darf nur von Stromkreisen versorgt werden, die der Klasse II (Class 2) oder Safety Extra Low Voltage (SELV) gemäß Klasse 9.4 von UL 61010-1 entsprechen.

Anhang D3: Netzteil NDR–240–24

Der nachfolgende Anhang D3 enthält das Datenblatt des Netzteils NDR–240–24 [49].

240W Single Output Industrial DIN RAIL **NDR-240 series**

Features

- Universal AC input / Full range
- Built-in active PFC function
- Protections: Short circuit / Overload / Over voltage / Over temperature
- Cooling by free air convection
- Can be installed on DIN rail TS-35/7.5 or 15
- UL 508 (industrial control equipment) approved
- EN61000-6-2(EN50082-2) industrial immunity level
- 100% full load burn-in test
- 3 years warranty

Applications

- Industrial control system
- Semi-conductor fabrication equipment
- Factory automation
- Electro-mechanical

Description

NDR-240 is one economical slim 240W Din rail power supply series, adapt to be installed on TS-35/7.5 or TS-35/15 mounting rails. The body is designed 63mm in width, which allows space saving inside the cabinets. The entire series adopts the full range AC input from 90VAC to 264VAC and conforms to EN61000-3-2, the norm the European Union regulates for harmonic current.

NDR-240 is designed with metal housing that enhances the unit's power dissipation. With working efficiency up to 90%, the entire series can operate at the ambient temperature between -20℃ and 70℃ under air convection. It is equipped with constant current mode for over-load protection, fitting various inductive or capacitive applications. The complete protection functions and relevant certificates for industrial control apparatus (UL508, TUV EN60950-1, and etc.) make NDR-240 a very competitive power supply solution for industrial applications.

Model Encoding

NDR - 240 - 48

- 48 — Output voltage
- 240 — Output wattage
- NDR — Series name

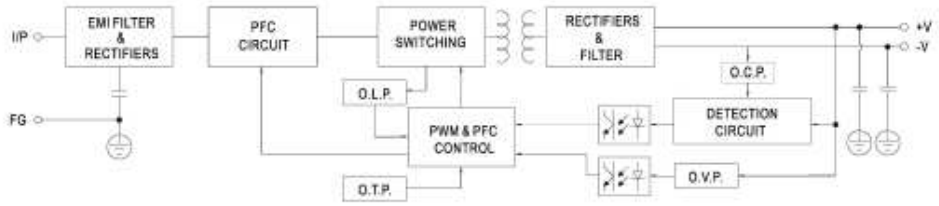
SPECIFICATION

MODEL		NDR-240-24	NDR-240-48
OUTPUT	DC VOLTAGE	24V	48V
	RATED CURRENT	10A	5A
	CURRENT RANGE	0 ~ 10A	0 ~ 5A
	RATED POWER	240W	240W
	RIPPLE & NOISE (max.) Note.2	150mVp-p	150mVp-p
	VOLTAGE ADJ. RANGE	24 ~ 28V	48 ~ 55V
	VOLTAGE TOLERANCE Note.3	±1.0%	±1.0%
	LINE REGULATION	±0.5%	±0.5%
	LOAD REGULATION	±1.0%	±1.0%
	SETUP, RISE TIME	1500ms, 100ms/230VAC 3000ms, 100ms/115VAC at full load	
	HOLD UP TIME (Typ.)	28ms/230VAC 22ms/115VAC at full load	
INPUT	VOLTAGE RANGE Note.4	90 ~ 264VAC 127 ~ 370VDC	
	FREQUENCY RANGE	47 ~ 63Hz	
	POWER FACTOR (Typ.)	PF>0.98/115VAC, PF>0.95/230VAC at full load	
	EFFICIENCY (Typ.)	88.5%	90%
	AC CURRENT (Typ.)	2.5A/115VAC 1.3A/230VAC	
	INRUSH CURRENT (Typ.)	20A/115VAC 35A/230VAC	
	LEAKAGE CURRENT	<1mA / 240VAC	
PROTECTION	OVERLOAD	105 ~ 130% rated output power	
		Protection type : Constant current limiting, recovers automatically after fault condition is removed	
	OVER VOLTAGE	29 ~ 33V	56 ~ 65V
		Protection type : Shut down o/p voltage, re-power on to recover	
	OVER TEMPERATURE	Shut down o/p voltage, recovers automatically after temperature goes down	
ENVIRONMENT	WORKING TEMP.	-20 ~ +70℃ (Refer to "Derating Curve")	
	WORKING HUMIDITY	20 ~ 95% RH non-condensing	
	STORAGE TEMP., HUMIDITY	-40 ~ +85℃, 10 ~ 95% RH	
	TEMP. COEFFICIENT	±0.03%/℃ (0 ~ 50℃)	
	VIBRATION	Component:10 ~ 500Hz, 2G 10min./1cycle, 60min. each along X, Y, Z axes; Mounting: Compliance to IEC60068-2-6	
SAFETY & EMC (Note 4)	SAFETY STANDARDS	UL508, TUV EN60950-1, EAC TP TC 004, BSMI CNS14336-1 approved ;(meet EN60204-1)	
	WITHSTAND VOLTAGE	I/P-O/P:3KVAC I/P-FG:2KVAC O/P-FG:0.5KVAC	
	ISOLATION RESISTANCE	I/P-O/P, I/P-FG, O/P-FG:>100M Ohms / 500VDC / 25℃/ 70% RH	
	EMC EMISSION	Compliance to EN55032 (CISPR32), EN61204-3 Class B, EN61000-3-2,-3, EAC TP TC 020, CNS13438	
	EMC IMMUNITY	Compliance to EN61000-4-2,3,4,5,6,8,11, EN55024, EN61000-6-2 (EN50082-2), EN61204-3, heavy industry level, criteria A, EAC TP TC 020	
OTHERS	MTBF	230.2K hrs min. MIL-HDBK-217F (25℃)	
	DIMENSION	63*125.2*113.5mm (W*H*D)	
	PACKING	1Kg; 12pcs/13Kg/1.1CUFT	
NOTE	1. All parameters NOT specially mentioned are measured at 230VAC input, rated load and 25℃ of ambient temperature. 2. Ripple & noise are measured at 20MHz of bandwidth by using a 12" twisted pair-wire terminated with a 0.1uf & 47uf parallel capacitor. 3. Tolerance : includes set up tolerance, line regulation and load regulation. 4. Derating may be needed under low input voltage. Please check the derating curve for more details. 5. Installation clearances : 40mm on top, 20mm on the bottom, 5mm on the left and right side are recommended when loaded permanently with full power. In case the adjacent device is a heat source, 15mm clearance is recommended. 6. The power supply is considered a component which will be installed into a final equipment. The final equipment must be re-confirmed that it still meets EMC directives. For guidance on how to perform these EMC tests, please refer to "EMI testing of component power supplies." 7. The ambient temperature derating of 3.5℃/1000m with fanless models and of 5℃/1000m with fan models for operating altitude higher than 2000m(6500ft). (as available on http://www.meanwell.com)		

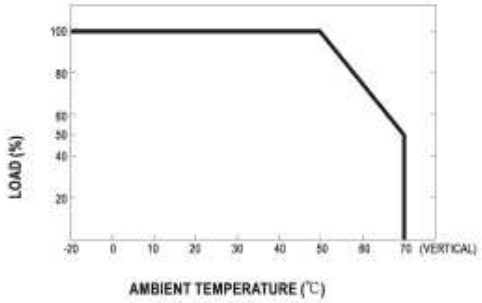
File Name:NDR-240-SPEC 2019-11-05

■ Block Diagram

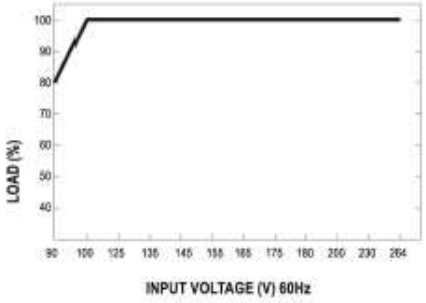
fosc : 70KHz



■ Derating Curve

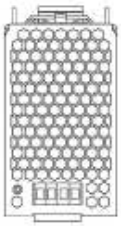


■ Output derating VS input voltage



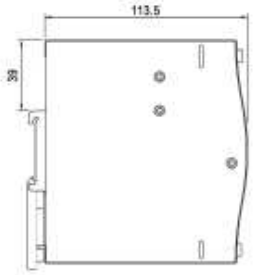
File Name:NDR-240-SPEC 2018-11-05

■ Mechanical Specification



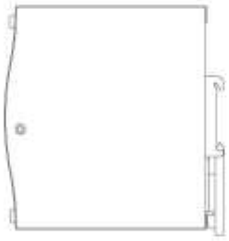
Case No.979C Unit:mm

Top View

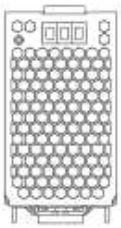


Side View

Front View



Side View



Bottom View

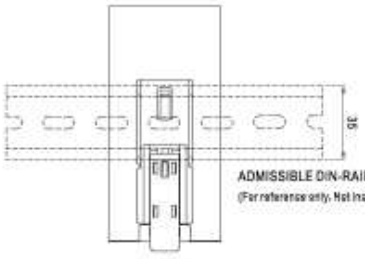
Terminal Pin No. Assignment (TB1)

Pin No.	Assignment
1	FG ⏚
2	AC/N or DC -
3	AC/L or DC +

Terminal Pin No. Assignment (TB2)

Pin No.	Assignment
1,2	DC OUTPUT -V
3,4	DC OUTPUT+V

■ Installation Instruction



ADMISSIBLE DIN-RAIL:TS35/7.5 OR TS35/15
(For reference only, Not included with unit.)

Back View

This series fits DIN rail TS35/7.5 or TS35/15.
For installation details, please refer to the Instruction manual.

■ Installation Manual

Please refer to : http://www.meanwell.com/manual.html

Anhang D4: Messumformer iTEMP TC TMT 128-AKAIA

Diesem Anhang lässt sich das Datenblatt des Thermoelement-Messumformers iTEMP TC TMT 128, mit der Bestellspezifikation AKAIA, entnehmen [50].

Füllstand Druck Durchfluss Temperatur Flüssigkeits-analyse Registrierung Systeme Komponenten Services Solutions

Technische Information

iTEMP® TC TMT128

Temperaturtransmitter für Thermoelemente (TC)
zur Hutschienenmontage

Anwendungsbereiche

- Temperaturtransmitter mit fest eingestellten Messbereichen zur Umwandlung eines TC-Eingangssignals in ein analoges, skalierbares 4 bis 20 mA Ausgangssignal
- Eingang:
  Thermoelemente (TC)

Vorteile auf einem Blick

- Fest eingestellter Messbereich für Thermoelemente
- 2-Drahttechnik, Analogausgang 4 bis 20 mA
- Hohe Genauigkeit im gesamten Umgebungstemperaturbereich
- Ausfallinformation bei Fühlerbruch nach NAMUR NE 43
- EMV nach NAMUR NE 21, CE
- EX-Zulassung
  - ATEX EEx ia, nA
  - CSA IS, NI
  - CSA GP
  - FM IS, NI
- GL Germanische Lloyd Schiffsbauzulassung
- UL Gerätesicherheit nach UL 3111-1
- Galvanische Trennung

TI098E/09/de
No. 51004701

Endress+Hauser
People for Process Automation

Arbeitsweise und Systemaufbau

Messprinzip

Elektronische Erfassung und Umformung von Eingangssignalen in der industriellen Temperaturmessung.

Messeinrichtung

Der DIN rail Temperaturtransmitter iTEMP® TC TMT128 ist ein Zweidrahtmessumformer mit Analogausgang und Messeingang für Thermoelemente.

Eingangskenngrößen

Messgröße

Temperatur

Messbereich

Je nach Applikation sind unterschiedliche Messbereiche bestellbar (siehe 'Produktübersicht').

Eingangstyp

Eingang	Bezeichnung	Messbereichsgrenzen	min. Messspanne
Thermoelemente (TC)	B (PtRh30-PtRh6)	0 bis +1820 °C (32 bis 3308 °F)	500 K
	C (W5Re-W26Re)[1]	0 bis +2320 °C (32 bis 4208 °F)	500 K
	D (W3Re-W25Re)[1]	0 bis +2405 °C (32 bis 4523 °F)	500 K
	E (NiCr-CuNi)	-270 bis +1000 °C (-454 bis 1832 °F)	50 K
	J (Fe-CuNi)	-210 bis +1200 °C (-346 bis 2192 °F)	50 K
	K (NiCr-Ni)	-270 bis +1372 °C (-454 bis 2501 °F)	50 K
	L (Fe-CuNi)[2]	-200 bis +900 °C (-328 bis 1652 °F)	50 K
	N (NiCrSi-NiSi)	-270 bis +1300 °C (-454 bis 2372 °F)	50 K
	R (PtRh13-Pt)	-50 bis +1768 °C (-58 bis 3214 °F)	500 K
	S (PtRh10-Pt)	-50 bis +1768 °C (-58 bis 3214 °F)	500 K
	T (Cu-CuNi)	-270 bis +400 °C (-454 bis 752 °F)	50 K
	U (Cu-CuNi)[2]	-200 bis +600 °C (-328 bis 1112 °F)	50 K
	nach IEC 60584 Teil 1		
	▪ Vergleichsstelle intern (Pt100) ▪ Vergleichsstellengenauigkeit: ±1 K ▪ Sensorstrom: = 350 nA		

1) nach ASTM E088

2) nach DIN 43710

Ausgangskenngrößen

Ausgangssignal

analog 4 bis 20 mA

Ausfallsignal

- Messbereichsunterschreitung: linearer Abfall bis 3,8 mA
- Messbereichsüberschreitung: linearer Anstieg bis 20,5 mA
- Fühlerbruch: ≥ 21,0 mA (>21,5 mA ist garantiert!)

Bürde

max. ($V_{Versorgung}$ - 12 V) / 0,022 A (Stromausgang)

Übertragungsverhalten

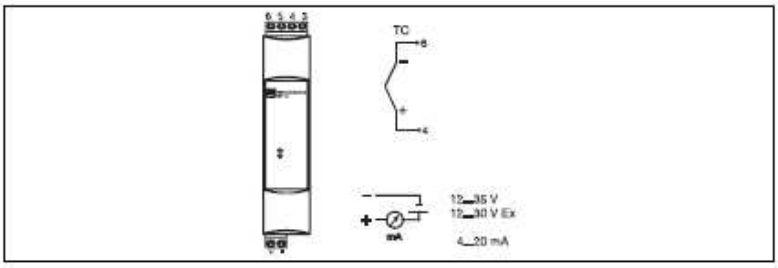
temperaturlinear

Galvanische Trennung

U = 2 kV AC (Eingang/Ausgang)

Hilfsenergie

Elektrischer Anschluss



Klemmenbelegung des Temperaturtransmitters

Versorgungsspannung

U_b= 12 bis 35 V, Verpolungsschutz

Restwelligkeit

Zul. Restwelligkeit $U_{ss} \leq 3$ V bei $U_b \geq 15$ V, $f_{max.}$ = 1 kHz

Messgenauigkeit

Antwortzeit

1 s

Referenzbedingungen

Kalibriertemperatur: +25 °C ± 5 K

Messabweichung

	Bezeichnung	Messgenauigkeit[1]
Thermoelemente (TC)	K, J, T, E, L, U N, C, D S, B, R	typ. 0,5 K oder 0,08% typ. 1,0 K oder 0,08% typ. 2,0 K oder 0,08%

1) % beziehen sich auf die eingestellte Messspanne. Der größere Wert ist gültig.

Einfluss der Versorgungsspannung

- ≤ ±0,01%/V Abweichung von 24 V
  Prozentangaben beziehen sich auf den Messbereichsendwert.

Einfluss der Umgebungstemperatur (Temperaturdrift)

- Thermoelement (TC):
  T_d= ±(50 ppm/K * max. Messbereich + 50 ppm/K * eingestellter Messbereich) * Δ ϑ

  Δ ϑ = Abweichung der Umgebungstemperaur von der Referenzbedingung (25 °C ± 5 K).

Einfluss Bürde

- ± 0,02%/100 Ω
  Angaben beziehen sich auf den Messbereichsendwert.

Langzeitstabilität

- ≤ 0,1 K/Jahr oder ≤ 0,05%/Jahr
  Angaben unter Referenzbedingungen. % beziehen sich auf die eingestellte Messspanne. Der größere Wert ist gültig.

Einbaubedingungen

Einbauhinweise

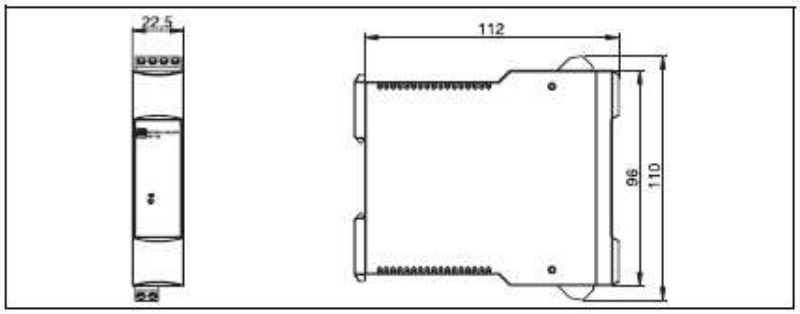
Einbaulage
keine Einschränkungen

Umgebungsbedingungen

Umgebungstemperaturgrenze	-40 bis +85 °C (-40 bis +185 °F), für Ex-Bereich siehe Ex-Zertifikat
Lagerungstemperatur	-40 bis +100 °C (-40 bis 212 °F)
Klimaklasse	nach IEC 60654-1, Klasse C
Schutzart	IP 20
Stoßfestigkeit	4g / 2 bis 150 Hz nach IEC 60068-2-6
Schwingungsfestigkeit	siehe unter 'Stoßfestigkeit'
Elektromagnetische Verträglichkeit (EMV)	Störfestigkeit und Störaussendung nach IEC 61326 und NAMUR NE 21
Betauung	zulässig

Konstruktiver Aufbau

Bauform, Maße



Angaben in mm

Gewicht	ca. 90 g
Werkstoffe	Gehäuse: PC/ABS, UL 94V0
Anschlussklemmen	Steckbare Schraubklemme, max. 2,5 mm² massiv, oder Litze mit Aderendhülse

Anzeige- und Bedienoberfläche

Anzeigeelemente	Leuchtende gelbe LED (2 mm) signalisiert Gerätebetrieb.
Bedienelemente	Am Gerät sind keine Bedienelemente vorhanden.

Zertifikate und Zulassungen

CE-Zeichen

Das Gerät erfüllt die gesetzlichen Anforderungen der EG-Richtlinien. Endress+Hauser bestätigt die erfolgreiche Prüfung des Gerätes mit der Anbringung des CE-Zeichens.

Ex-Zulassung

Über die aktuell lieferbaren Ex-Ausführungen (ATEX, FM, CSA, usw.) erhalten Sie bei Ihrer Endress+Hauser-Vertriebsstelle Auskunft. Alle für den Explosionsschutz relevanten Daten finden Sie in separaten Ex-Dokumentationen, die Sie bei Bedarf anfordern können.

GL Schiffsbauzulassung

GL Germanische Lloyd Schiffsbauzulassung

Externe Normen und Richtlinien

- IEC 60529:
  Schutzarten durch Gehäuse (IP-Code)
- IEC 61010:
  Sicherheitsbestimmungen für elektrische Mess-, Steuer-, Regel- und Laborgeräte
- IEC 61326:
  Elektromagnetische Verträglichkeit (EMV-Anforderungen)
- NAMUR:
  Interessengemeinschaft Automatisierungstechnik der Prozessindustrie (www.namur.de)

Gerätesicherheit UL

Gerätesicherheit nach UL 3111-1

Bestellinformationen

Produktübersicht

TMT128	iTEMP TC DIN rail TMT128			
	zur Temperaturmessung mit Thermoelementen (TC) Analogausgang 4 bis 20 mA; 2-Leiter; Galvanische Trennung; Fehlerverhalten NAMUR NE 43; 22,5 mm breit; für Tragschiene 35 mm nach IEC 60715			
	Zulassung			
	A	Ex-freier Bereich		
	B	ATEX II2(1)G EEx ia IIC T4/T5/T6		
	C	FM IS, NI, Class I, Div. 1+2, Group ABCD		
	D	CSA IS, NI, Class I, Div. 1+2, Group ABCD		
	E	ATEX II3G EEx nA IIC T4/T5/T6		
	I	FM+CSA IS, NI, Class I, Div. 1+2, Group ABCD		
	J	CSA General Purpose		
	K	TIIS Ex ia IIC T5		
	1	NEPSI Ex ia IIC T4-T6		
	2	NEPSI Ex nA II T4-T6		
			Temperatursensor	
		B	Typ B (400 bis 1820 °C, min. Spanne 500 K)	
		C	Typ C (500 bis 2320 °C, min. Spanne 500 K)	
		D	Typ D (500 bis 2495 °C, min. Spanne 500 K)	
		E	Typ E (-200 bis 1000 °C, min. Spanne 50 K)	
		J	Typ J (-200 bis 1200 °C, min. Spanne 50 K)	
		K	Typ K (-200 bis 1372 °C, min. Spanne 50 K)	
		L	Typ L (-200 bis 900 °C, min. Spanne 50 K)	
		N	Typ N (-100 bis 1300 °C, min. Spanne 50 K)	
		R	Typ R (-50 bis 1768 °C, min. Spanne 500 K)	
		S	Typ S (-50 bis 1768 °C, min. Spanne 500 K)	
		T	Typ T (-200 bis 400 °C, min. Spanne 50 K)	
		U	Typ U (-200 bis 600 °C, min. Spanne 50 K)	
			Messbereich	
			AA	Messbereich 0 bis 100 °C
			AB	Messbereich 0 bis 150 °C
			AC	Messbereich 0 bis 250 °C
			AD	Messbereich 0 bis 400 °C
			AE	Messbereich 0 bis 600 °C
			AF	Messbereich 0 bis 900 °C
			AG	Messbereich 0 bis 1000 °C
			AH	Messbereich 0 bis 1200 °C

			Messbereich			
			AI	Messbereich	0 bis 1400 °C
			AJ	Messbereich	0 bis 1600 °C
			AK	Messbereich	0 bis 200 °C
			AL	Messbereich	0 bis 300 °C
			AM	Messbereich	0 bis 500 °C
			DE	Messbereich	-10 bis 200 °C
			JA	Messbereich	-50 bis 200 °C
				Zusatzausstattung	
				A	Grundausführung
				B	Werkskalibrierzertifikat (6 Messpunkte)
TMT128-					⇒ Bestellcode (komplett)

Diese Informationen geben einen Überblick über die verfügbaren Bestellmöglichkeiten. Bestellinformationen und ausführliche Angaben zum Bestellcode erhalten Sie von Ihrer Endress+Hauser Serviceorganisation.

Zubehör

Für dieses Gerät wird kein Zubehör benötigt.

Ergänzende Dokumentation

- ❑ Broschüre "Temperaturmesstechnik" (FA006T09de)
- ❑ Betriebsanleitung "iTEMP® RTD/TC DIN rail TMT127/128" (KA140R09a3)
- ❑ Ex-Zusatzdokumentationen:
  ATEX Sicherheitshinweise II2(1)G (XA013R09a3) und II3G (XA018R09a3)

Deutschland				Österreich	Schweiz
Endress+Hauser Messtechnik GmbH+Co. KG Colmarer Straße 6 79576 Weil am Rhein	Vertrieb ▪ Beratung ▪ Information ▪ Auftrag ▪ Bestellung	Service ▪ Help-Desk ▪ Feldservice ▪ Ersatzteile/Reparatur ▪ Kalibrierung	Technische Büros ▪ Hamburg ▪ Berlin ▪ Hannover ▪ Ratingen ▪ Frankfurt ▪ Stuttgart ▪ München	Endress+Hauser Ges. m.b.H. Lehnergasse 4 1230 Wien	Endress+Hauser Metso AG Kägenstrasse 2 4153 Reinach
Fax 0800 EHFAXEN Fax 0800 343 29 36 www.de.endress.com	Tel. 0800 EHVERTRIEB Tel. 0800 348 37 87 info@de.endress.com	Tel. 0800 EHSERVICE Tel. 0800 347 37 84 service@de.endress.com		Tel. +43 1 880 56 0 Fax +43 1 880 56 335 info@at.endress.com www.at.endress.com	Tel. +41 61 715 75 75 Fax +41 61 715 27 75 info@ch.endress.com www.ch.endress.com

Endress+Hauser
People for Process Automation

TI090R/09/de/10.08
51004791
FM+SGML6.0 / ProMoDo

Anhang D5: Auszug Datenblatt Picoscope

Dem nachfolgenden Anhang kann ein Auszug aus dem Datenblatt des „PicoScope 5442D MSO“ entnommen werden, der alle relevanten technischen Daten enthält [51].

PicoScope 5000D Series Technical Specifications	PicoScope 5242D and 5242D MSO 2-channel, 60 MHz	PicoScope 5442D and 5442D MSO 4-channel, 60 MHz	PicoScope 5243D and 5243D MSO 2-channel, 100 MHz	PicoScope 5443D and 5443D MSO 4-channel, 100 MHz	PicoScope 5244D and 5244D MSO 2-channel, 200 MHz	PicoScope 5444D and 5444D MSO 4-channel, 200 MHz
Vertical (analog channels)						
Analog input channels	2	4	2	4	2	4
Input type	Single-ended, BNC(f) connector					
Bandwidth (−3 dB)	60 MHz		100 MHz[1]		200 MHz[1]	
Rise time (calculated)	5.8 ns		3.5 ns[1]		1.75 ns[1]	
Bandwidth limiter	20 MHz, selectable					
Vertical resolution [2]	8, 12, 14, 15 or 16 bits					
LSB size (quantization step size)[2]	8-bit mode: < 0.6% of input range 12-bit mode: < 0.04% of input range 14-bit mode: < 0.01% of input range 15-bit mode: < 0.005% of input range 16-bit mode: < 0.0025% of input range					
Enhanced vertical resolution	Hardware resolution + 4 bits					
Input ranges	±10 mV to ±20 V full scale, in 11 ranges					
Input sensitivity	2 mV/div to 4 V/div (10 vertical divisions)					
Input coupling	AC / DC					
Input characteristics	1 MΩ ±1% || 14 ±1 pF					
Gain accuracy	12 to 16-bit modes: ±0.5% of signal ±1 LSB[3] 8-bit mode: ±2% of signal ±1 LSB[3]					
Offset accuracy	±500 µV ±1% of full scale[3] Offset accuracy can be improved by using the zero offset function in PicoScope 6.					
Analog offset range (vertical position adjust)	±250 mV (10, 20, 50, 100, 200 mV ranges) ±2.5 V (500 mV, 1 V, 2 V ranges) ±20 V (5, 10, 20 V ranges)					
Analog offset control accuracy	±0.5% of offset setting, additional to basic DC offset accuracy					
Overvoltage protection	±100 V (DC + AC peak)					
[1] In 16-bit mode, bandwidth reduced to 60 MHz and rise time increased to 5.8 ns. [2] On ±20 mV range, in 14 to 16-bit modes, hardware resolution reduced by 1 bit. On ±10 mV range, hardware resolution reduced by 1 bit in 12-bit mode, 2 bits in 14 to 16-bit modes. [3] Between 15 and 30 °C after 1 hour warm-up.						
Vertical (digital channels) – D MSO models only						
Input channels	16 channels (2 ports of 8 channels each)					
Input connector	2.54 mm pitch, 10 x 2 way connector					
Maximum input frequency	100 MHz (200 Mbit/s)					

PicoScope 5000D Series Technical Specifications	PicoScope 5242D and 5242D MSO 2-channel, 60 MHz	PicoScope 5442D and 5442D MSO 4-channel, 60 MHz	PicoScope 5243D and 5243D MSO 2-channel, 100 MHz	PicoScope 5443D and 5443D MSO 4-channel, 100 MHz	PicoScope 5244D and 5244D MSO 2-channel, 200 MHz	PicoScope 5444D and 5444D MSO 4-channel, 200 MHz
Minimum detectable pulse width	5 ns					
Input impedance	200 kΩ ±2% || 8 pF ±2 pF					
Input dynamic range	±20 V					
Threshold range	±5 V					
Threshold grouping	Two independent threshold controls. Port 0: D0 to D7, Port 1: D8 to D15					
Threshold selection	TTL, CMOS, ECL, PECL, user-defined					
Threshold accuracy	< ±350 mV including hysteresis					
Threshold hysteresis	< ±250 mV					
Minimum input voltage swing	500 mV peak to peak					
Channel-to-channel skew	2 ns, typical					
Minimum input slew rate	10 V/µs					
Overvoltage protection	±50 V (DC + AC peak)					
Horizontal						
Maximum sampling rate Any 1 channel Any 2 channels Any 3 or 4 channels More than 4 channels	8-bit mode 1 GS/s 500 MS/s 250 MS/s 125 MS/s	12-bit mode 500 MS/s 250 MS/s 125 MS/s 62.5 MS/s	14-bit mode 125 MS/s 125 MS/s 125 MS/s 62.5 MS/s	15-bit mode[4] 125 MS/s 125 MS/s	16-bit mode[4] 62.5 MS/s	
	"Channel" means any analog channel or 8-bit digital port.					
Maximum equivalent sampling rate (repetitive signals; 8-bit mode only, ETS mode)	2.5 GS/s		5 GS/s		10 GS/s	
Maximum sampling rate (continuous USB streaming into PC memory)[5]	USB 3, using PicoScope 6: 15 to 20 MS/s USB 3, using PicoSDK: 125 MS/s (8-bit mode) or 62.5 MS/s (12 to 16-bit modes) USB 2, using PicoScope 6: 8 to 10 MS/s USB 2, using PicoSDK: ~30 MS/s (8-bit mode) or ~15 MS/s (12 to 16-bit modes)					
Timebase ranges (real-time)	1 ns/div to 5000 s/div in 39 ranges					
Fastest timebase (ETS)	500 ps/div		200 ps/div		100 ps/div	
Buffer memory[6] (8-bit)	128 MS		256 MS		512 MS	
Buffer memory[6] (≥ 12-bit)	64 MS		128 MS		256 MS	
Buffer memory[7] (continuous streaming)	100 MS in PicoScope 6 software					
Waveform buffer (no. of segments)	10 000 in PicoScope 6 software					

PicoScope 5000D Series Technical Specifications	PicoScope 5242D and 5242D MSO 2-channel, 60 MHz	PicoScope 5442D and 5442D MSO 4-channel, 60 MHz	PicoScope 5243D and 5243D MSO 2-channel, 100 MHz	PicoScope 5443D and 5443D MSO 4-channel, 100 MHz	PicoScope 5244D and 5244D MSO 2-channel, 200 MHz	PicoScope 5444D and 5444D MSO 4-channel, 200 MHz
Waveform buffer (no. of segments) when using PicoSDK (8 bits)	250 000		500 000		1 000 000	
Waveform buffer (no. of segments) when using PicoSDK (12 to 16 bits)	125 000		250 000		500 000	
Initial timebase accuracy	±50 ppm (0.005%)		±2 ppm (0.0002%)		±2 ppm (0.0002%)	
Timebase drift	±5 ppm/year		±1 ppm/year		±1 ppm/year	
Sample jitter	3 ps RMS, typical					
ADC sampling	Simultaneous on all enabled channels.					
[4]Any number of 8-bit digital ports can be used in 15-bit and 16-bit modes without affecting the maximum sampling rate. [5]Shared between enabled channels, PC dependent, available sample rates vary by resolution. [6]Shared between enabled channels. [7]Driver buffering up to available PC memory when using PicoSDK. No limit on duration of capture.						
Dynamic performance (typical, analog channels)						
Crosstalk	Better than 400:1 up to full bandwidth (equal voltage ranges).					
Harmonic distortion	8-bit mode: −60 dB at 100 kHz full scale input 12 to 16-bit modes: −70 dB at 100 kHz full scale input					
SFDR	8 to 12-bit modes: 60 dB at 100 kHz full scale input 14 to 16-bit modes: 70 dB at 100 kHz full scale input					
Noise (on ±10 mV range)	8-bit mode: 120 µV RMS 12-bit mode: 110 µV RMS 14-bit mode: 100 µV RMS 15-bit mode: 85 µV RMS 16-bit mode: 70 µV RMS					
Bandwidth flatness	(+0.3 dB, −3 dB) from DC to full bandwidth					
Triggering (main specifications)						
Source	Analog channels, plus: MSO models: Digital D0 to D15; other models: Ext trigger					
Trigger modes	None, auto, repeat, single, rapid (segmented memory)					
Advanced trigger types (analog channels)	Edge, window, pulse width, window pulse width, dropout, window dropout, interval, runt, logic					
Trigger types (analog channels, ETS)	Rising or falling edge ETS trigger available on ChA only, 8-bit mode only					
Trigger sensitivity (analog channels)	Digital triggering provides 1 LSB accuracy up to full bandwidth of scope					

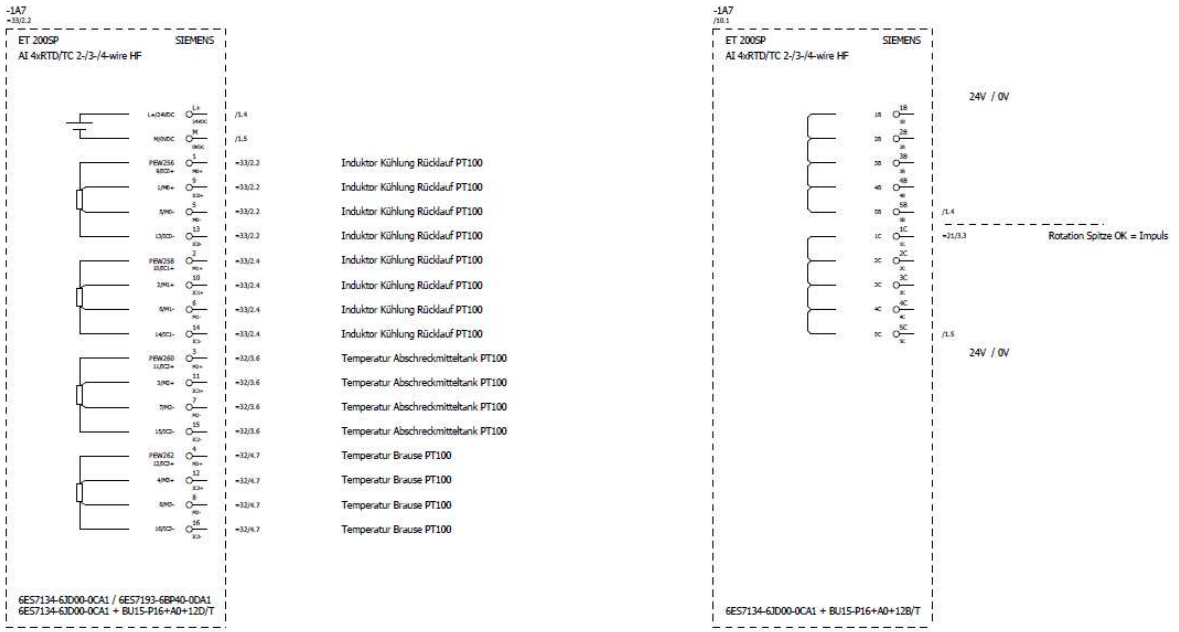
PicoScope 5000D Series Technical Specifications	PicoScope 5242D and 5242D MSO 2-channel, 60 MHz	PicoScope 5442D and 5442D MSO 4-channel, 60 MHz	PicoScope 5243D and 5243D MSO 2-channel, 100 MHz	PicoScope 5443D and 5443D MSO 4-channel, 100 MHz	PicoScope 5244D and 5244D MSO 2-channel, 200 MHz	PicoScope 5444D and 5444D MSO 4-channel, 200 MHz
Trigger sensitivity (analog channels, ETS)	At full bandwidth: typical 10 mV peak to peak					
Trigger types (digital inputs)	MSO models only: Edge, pulse width, dropout, interval, logic, pattern, mixed signal					
Maximum pre-trigger capture	Up to 100% of capture size.					
Maximum post-trigger delay	Zero to 4 billion samples, settable in 1 sample steps (delay range on fastest timebase of 0 to 4 s in 1 ns steps)					
Trigger rearm time	8-bit mode, typical: 1 µs on fastest timebase 8 to 12-bit modes: < 2 µs max on fastest timebase 14 to 16-bit modes: < 3 µs max on fastest timebase					
Maximum trigger rate	10 000 waveforms in a 10 ms burst, 8-bit mode					
External trigger input – not MSO models						
Connector type	Front panel BNC(f)					
Trigger types	Edge, pulse width, dropout, interval, logic					
Input characteristics	1 MΩ ±1% || 14 pF ±1.5 pF					
Bandwidth	60 MHz		100 MHz		200 MHz	
Threshold range	±5 V					
External trigger threshold accuracy	±1% of full scale					
External trigger sensitivity	200 mV peak to peak					
Coupling	DC					
Overvoltage protection	±100 V (DC + AC peak)					
Function generator						
Standard output signals	Sine, square, triangle, DC voltage, ramp up, ramp down, sinc, Gaussian, half-sine					
Pseudorandom output signals	White noise, selectable amplitude and offset within output voltage range. Pseudorandom binary sequence (PRBS), selectable high and low levels within output voltage range, selectable bit rate up to 20 Mb/s					
Standard signal frequency	0.025 Hz to 20 MHz					
Sweep modes	Up, down, dual with selectable start / stop frequencies and increments					
Triggering	Can trigger a counted number of waveform cycles or frequency sweeps (from 1 to 1 billion) from the scope trigger, external trigger or from software. Can also use the external trigger to gate the signal generator output.					
Output frequency accuracy	Oscilloscope timebase accuracy ± output frequency resolution					
Output frequency resolution	< 0.025 Hz					
Output voltage range	±2 V					
Output voltage adjustments	Signal amplitude and offset adjustable in approx 0.25 mV steps within overall ±2 V range					
Amplitude flatness	< 1.5 dB to 20 MHz, typical					
DC accuracy	±1% of full scale					

PicoScope 5000D Series Technical Specifications	PicoScope 5242D and 5242D MSO 2-channel, 60 MHz	PicoScope 5442D and 5442D MSO 4-channel, 60 MHz	PicoScope 5243D and 5243D MSO 2-channel, 100 MHz	PicoScope 5443D and 5443D MSO 4-channel, 100 MHz	PicoScope 5244D and 5244D MSO 2-channel, 200 MHz	PicoScope 5444D and 5444D MSO 4-channel, 200 MHz
SFDR	> 70 dB, 10 kHz full scale sine wave					
Output resistance	50 Ω ±1%					
Connector type	BNC(f)					
Overvoltage protection	±20 V					
Arbitrary waveform generator						
AWG update rate	200 MHz					
AWG buffer size	32 kS					
AWG resolution	14 bits (output step size approx 0.25 mV)					
AWG bandwidth	> 20 MHz					
AWG rise time (10% to 90%)	<10 ns (50 Ω load)					
Other AWG specifications including sweep modes, triggering, frequency accuracy and resolution, voltage range, DC accuracy and output characteristics are as function generator.						
Probe compensation pin						
Output characteristics	600 Ω					
Output frequency	1 kHz					
Output level	3 V peak to peak, typical					
Overvoltage protection	10 V					
Spectrum analyzer						
Frequency range	DC to 60 MHz		DC to 100 MHz		DC to 200 MHz	
Display modes	Magnitude, average, peak hold					
Y axis	Logarithmic (dbV, dBu, dBm, arbitrary dB) or linear (volts)					
X axis	Linear or logarithmic					
Windowing functions	Rectangular, Gaussian, triangular, Blackman, Blackman–Harris, Hamming, Hann, flat-top					
Number of FFT points	Selectable from 128 to 1 million in powers of 2					
Math channels						
Functions	−x, x+y, x−y, x*y, x/y, x^y, sqrt, exp, ln, log, abs, norm, sign, sin, cos, tan, arcsin, arccos, arctan, sinh, cosh, tanh, delay, average, frequency, derivative, integral, min, max, peak, duty, highpass, lowpass, bandpass, bandstop					
Operands	A, B, C, D (input channels), T (time), reference waveforms, pi, D0–D15 (digital channels), constants					
Automatic measurements						
Scope mode	AC RMS, true RMS, frequency, cycle time, duty cycle, DC average, falling rate, rising rate, low pulse width, high pulse width, fall time, rise time, minimum, maximum, peak to peak					
Spectrum mode	Frequency at peak, amplitude at peak, average amplitude at peak, total power, THD %, THD dB, THD+N, SFDR, SINAD, SNR, IMD					
Statistics	Minimum, maximum, average, standard deviation					

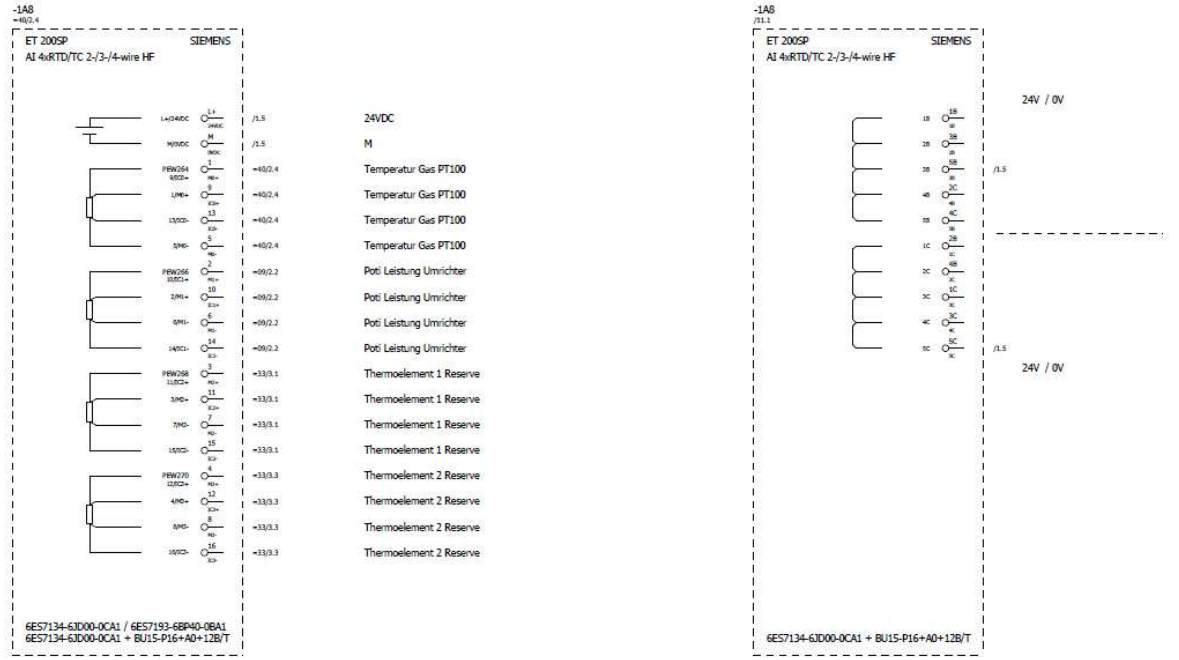
PicoScope 5000D Series Technical Specifications	PicoScope 5242D and 5242D MSO 2-channel, 60 MHz	PicoScope 5442D and 5442D MSO 4-channel, 60 MHz	PicoScope 5243D and 5243D MSO 2-channel, 100 MHz	PicoScope 5443D and 5443D MSO 4-channel, 100 MHz	PicoScope 5244D and 5244D MSO 2-channel, 200 MHz	PicoScope 5444D and 5444D MSO 4-channel, 200 MHz
DeepMeasure™						
Parameters	Cycle number, cycle time, frequency, low pulse width, high pulse width, duty cycle (high), duty cycle (low), rise time, fall time, undershoot, overshoot, max. voltage, min. voltage, voltage peak to peak, start time, end time					
Serial decoding						
Protocols	1-Wire, ARINC 429, CAN & CAN-FD, DCC, DMX512, Ethernet 10Base-T and 100Base-TX, FlexRay, I²C, I²S, LIN, PS/2, MODBUS, SENT, SPI, UART (RS-232 / RS-422 / RS-485), USB 1.1					
Mask limit testing						
Statistics	Pass/fail, failure count, total count					
Mask creation	User-drawn, table entry, auto-generated from waveform or imported from file					
Display						
Interpolation	Linear or sin(x)/x					
Persistence modes	Digital color, analog intensity, custom, fast					
General						
PC connectivity	USB 3.0 SuperSpeed (USB 2.0 compatible)					
Power requirements	2-channel models: powered from single USB 3.0 port 4-channel models: AC adaptor supplied. Can use 2 channels (plus MSO channels if fitted) powered by USB 3.0 or charging port supplying 1.2 A.					
Dimensions	190 x 170 x 40 mm including connectors					
Weight	< 0.5 kg					
Temperature range	Operating: 0 to 40 °C 15 to 30 °C for quoted accuracy after 1 hour warm-up Storage: −20 to +60 °C					
Humidity range	Operating: 5 to 80 %RH non-condensing Storage: 5 to 95 %RH non-condensing					
Environment	Up to 2000 m altitude and EN61010 pollution degree 2					
Safety approvals	Designed to EN 61010-1:2010					
EMC approvals	Tested to EN61326-1:2013 and FCC Part 15 Subpart B					
Environmental approvals	RoHS and WEEE compliant					
Software	PicoScope 6: Windows 7, 8 and 10 (32-bit and 64-bit versions). Beta software also available for 64-bit Linux and macOS. PicoSDK: Windows 7, 8 and 10 (32-bit and 64-bit versions). Drivers also available for 64-bit Linux and macOS. Example programs for supported languages and development environments					
PC requirements	Processor, memory and disk space: as required by the operating system					
	Port(s): USB 3.0 or USB 2.0					
Software languages	Simplified and traditional Chinese, Czech, Danish, Dutch, English, Finnish, French, German, Greek, Hungarian, Italian, Japanese, Korean, Norwegian, Polish, Portuguese, Romanian, Russian, Spanish, Swedish, Turkish					

Anhang E: Auszug aus den Hardwareplänen der Induktionsanlage

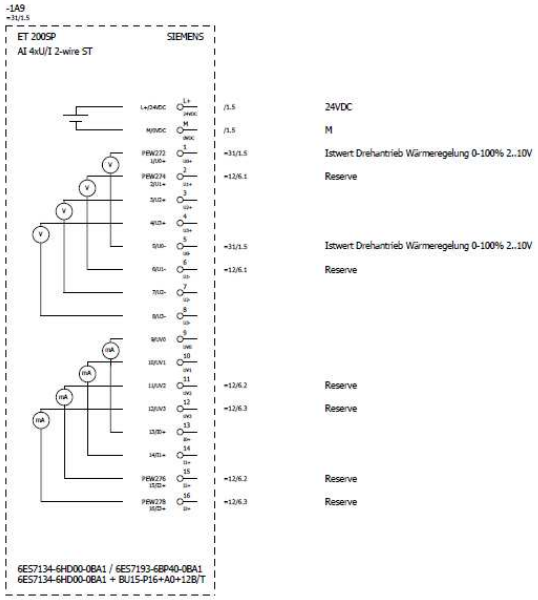
In diesem Anhang sind all jene Seiten der Hardwarepläne der Induktionsanlage aufgelistet, die relevante Anschlusspins zum Abgreifen der verschiedenen Signale beinhalten.

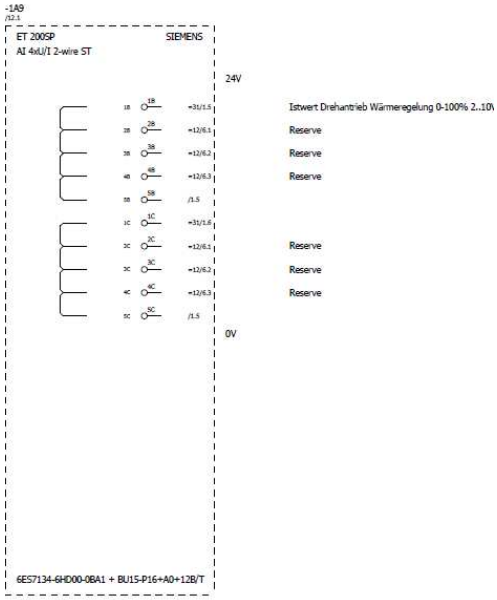


Seite 30

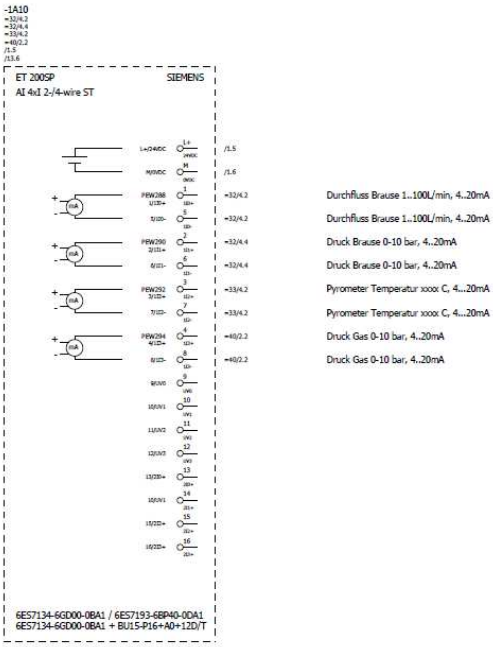


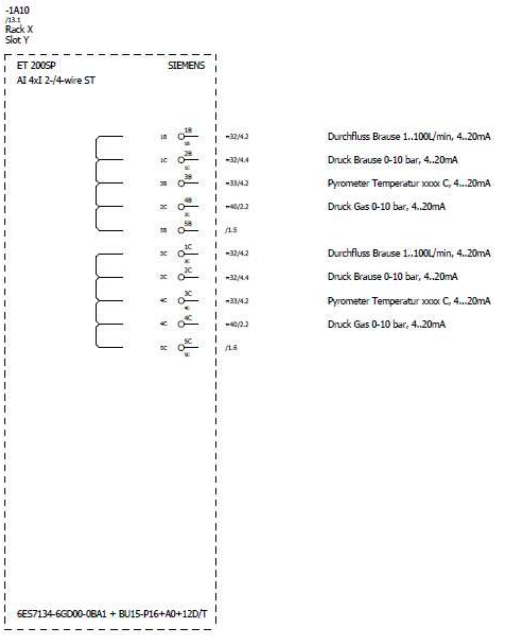
Seite 31



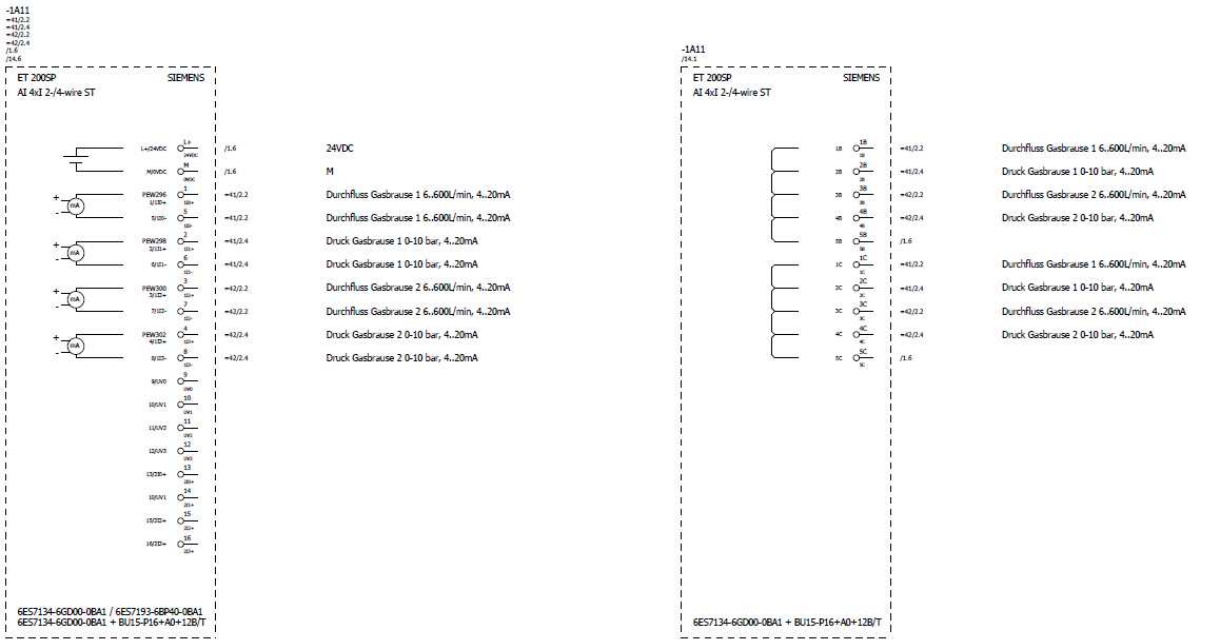


Seite 32

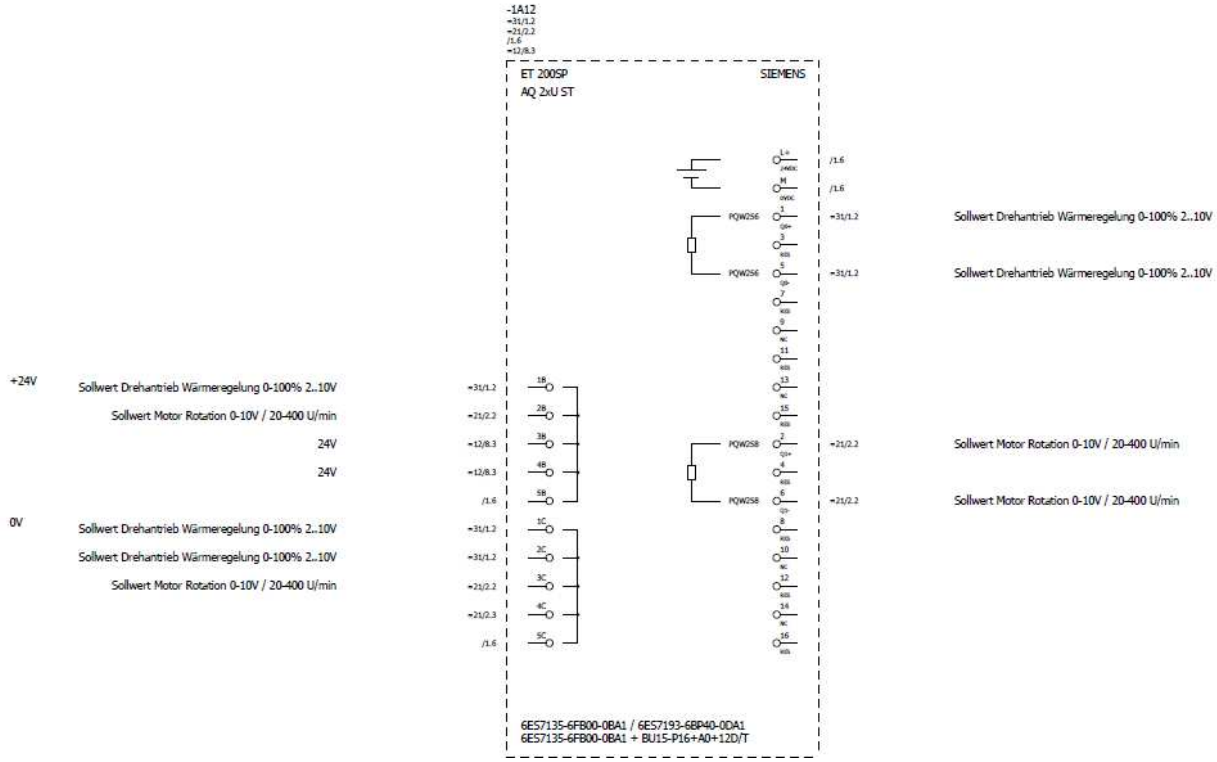




Seite 33



Seite 34



Seite 35

Anhang F: Auszug „Programmer's Guide – PicoScope 5000 Series“

Der nachfolgende Anhang enthält zwei Auszüge aus dem „Programmer's Guide – PicoScope 5000 Series“ [51]. Der erste stellt die relevanten Informationen zu dem bei der Messdatenerfassung verwendeten „Block mode“ zur Verfügung. Der zweite beinhaltet die Formeln zur Berechnung der „timebase“.

Block Mode:

3.5.1 Block mode

In **block mode**, the computer prompts a PicoScope 5000 Series oscilloscope to collect a block of data into its internal memory. When the oscilloscope has collected the whole block, it signals that it is ready and then transfers the whole block to the computer's memory through the USB port.

- **Block size.** The maximum number of values depends upon the size of the oscilloscope's memory. The memory buffer is shared between the enabled channels, so if two channels are enabled, each receives half the memory. These features are handled transparently by the driver. The block size also depends on the number of memory segments in use (see `ps5000aMemorySegments`).
- **Sampling rate.** A PicoScope 5000 Series oscilloscope can sample at a number of different rates according to the selected timebase and resolution. In turn, the available timebases may depend on the combination of channels enabled. See the PicoScope 5000 Series User's Guide for the specifications that apply to your scope model. You can call `ps5000aGetMinimumTimebaseStateless` to find the fastest available timebase.
- **Setup time.** The driver normally performs a number of setup operations, which can take up to 50 milliseconds, before collecting each block of data. If you need to collect data with the minimum time interval between blocks, use rapid block mode and avoid calling setup functions between calls to `ps5000aRunBlock`, `ps5000aStop` and `ps5000aGetValues`.
- **Downsampling.** When the data has been collected, you can set an optional downsampling factor and examine the data. Downsampling is a process that reduces the amount of data by combining adjacent samples. It is useful for zooming in and out of the data without having to repeatedly transfer the entire contents of the scope's buffer to the PC.
- **Segmented memory.** The scope's internal memory can be divided into segments so that you can capture several waveforms in succession. Configure this using `ps5000aMemorySegments`.
- **Data retention.** The data is lost when a new run is started in the same segment, the settings are changed, the resolution is changed, or the scope is powered down or (for flexible power devices) the power source is changed.

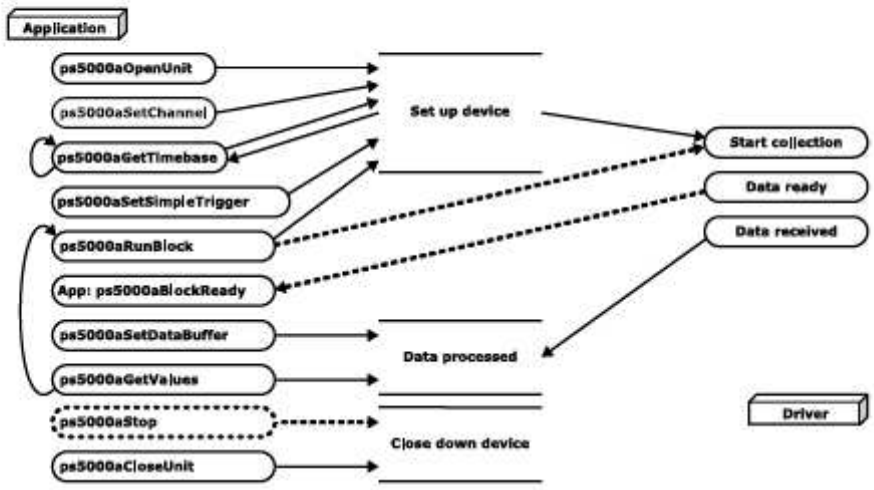
See Using block mode for programming details.

3.5.1.1 Using block mode

You can use block mode with or without aggregation. With aggregation, you need to set up two buffers for each channel to receive the minimum and maximum values: see rapid block mode example 2 for an example of this.

Here is the general procedure for reading and displaying data in block mode using a single memory segment:

1. Open the oscilloscope using `ps5000aOpenUnit`.
2. Select channel ranges and AC/DC coupling using `ps5000aSetChannel`.
2a. Set the digital port using `ps5000aSetDigitalPort` (mixed-signal scopes only).
3. Using `ps5000aGetTimebase`, select timebases until the required nanoseconds per sample is located.
4. Use the trigger setup function `ps5000aSetSimpleTrigger` to set up the trigger if required.
4a. Use the trigger setup functions `ps5000aSetTriggerDigitalPortProperties` and `ps5000aSetTriggerChannelConditionsV2` to set up the digital trigger if required (mixed-signal scopes only).
5. Start the oscilloscope running using `ps5000aRunBlock`.
6. Wait until the oscilloscope is ready using the `ps5000aBlockReady` callback (or poll using `ps5000aIsReady`).
7. Use `ps5000aSetDataBuffer` to tell the driver where your memory buffer is. For greater efficiency when doing multiple captures, you can call this function outside the loop, after step 4.
8. Transfer the block of data from the oscilloscope using `ps5000aGetValues`.
9. Display the data.
10. Repeat steps 5 to 9.
11. Stop the oscilloscope using `ps5000aStop`.
12. Request new views of stored data using different downsampling parameters: see Retrieving stored data.
13. Close the device using `ps5000aCloseUnit`.



Note that if you use `ps5000aGetValues` or `ps5000aStop` before the oscilloscope is ready, no capture will be available. In this case `ps5000aGetValues` would return `PICO_NO_SAMPLES_AVAILABLE`.

3.5.1.2 Asynchronous data retrieval

The ps5000aGetValues function may take a long time to complete if a large amount of data is being collected. For example, it can take 14 seconds (or several minutes on USB 1.1) to retrieve the full 512 megasamples (in 8-bit mode) from the higher-capacity PicoScope 5000 Series models using a USB 2.0 connection. To avoid hanging the calling thread, it is possible to call ps5000aGetValuesAsync instead. This immediately returns control to the calling thread, which then has the option of waiting for the data or calling ps5000aStop to abort the operation.

Timebase:

3.6 Timebases

The timebase is an integer that encodes the sampling interval of the oscilloscope. The API allows you to select any available* timebase down to the minimum sampling interval of your oscilloscope. The available timebases allow slow enough sampling in block mode to overlap the streaming sample intervals, so that you can make a smooth transition between block mode and streaming mode.

Convert a given timebase to a sampling interval using ps5000aGetTimebase. Find the fastest available timebase in a given mode using ps5000aGetMinimumTimebaseStateless.

Accepted timebases for each resolution mode are as follows:

8-bit mode

Timebase (n)	Sampling interval formula	Sampling interval	Notes
0	2^n / 1,000,000,000	1 ns	Only one channel enabled
1		2 ns	
2		4 ns	
3 ... $2^{32}-1$	(n−2) / 125,000,000	8 ns ... ~ 34.36 s	

12-bit mode

Timebase (n)**	Sampling interval formula	Sampling interval	Notes
1	$2^{(n-1)}$ / 500,000,000	2 ns	Only one channel enabled
2		4 ns	
3		8 ns	
4 ... $2^{32}-2$	(n−3) / 62,500,000	16 ns ... ~ 68.72 s	

14-bit mode

Timebase (n)†	Sampling interval formula	Sampling interval	Notes
3	1 / 125,000,000	8 ns	5000A/B Series: only one analog channel enabled. 5000D Series: up to 4 analog channels or digital ports enabled.
4 ... $2^{32}-1$	(n−2) / 125,000,000	16 ns ... ~ 34.36 s	

15-bit mode

PicoScope 5000D MSO Series: any number of digital ports can be enabled without affecting the timebase.

Timebase (n)†	Sampling interval formula	Sampling interval	Notes
3	1 / 125,000,000	8 ns	Up to two analog channels enabled.
4 ... $2^{32}-1$	(n−2) / 125,000,000	16 ns ... ~ 34.36 s	

16-bit mode

PicoScope 5000D MSO Series: any number of digital ports can be enabled without affecting the timebase.

Timebase (n) ‡	Sampling interval formula	Sampling interval	Notes
4	1 / 62,500,000	16 ns	Only one analog channel enabled.
5 ... $2^{32}-2$	(n−3) / 62,500,000	32 ns ... ~ 68.72 s	

* The fastest available sampling rate depends on the combination of channels and ports enabled, the sampling mode, the ETS mode and the power supply mode. Please refer to the oscilloscope data sheet for sampling rate specifications. In streaming mode, the speed of the USB port may affect the rate of data transfer.
** Timebase 0 is not available in 12-bit resolution mode.
† Timebases 0, 1 and 2 are not available in 14 and 15-bit resolution modes.
‡ Timebases 0, 1, 2 and 3 are not available in 16-bit resolution mode.

ETS mode
In ETS mode the sample time is not set according to the above tables, but is instead calculated and returned by `ps5000aSetEts`.

Anhang G: Herleitung: Berechnung Messwert aus Sensorsignal

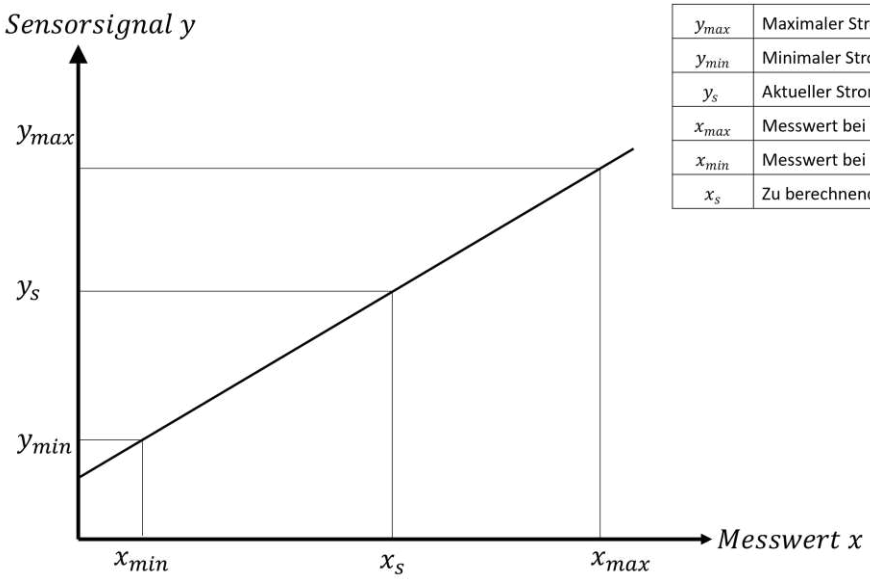
Dem nachfolgenden Anhang kann die Herleitung der Formel zur Berechnung eines Messwertes aus einem normierten Sensorsignal entnommen werden:

Der zu errechnende Messwert x_s ändert sich linear mit dem erhaltenen Sensorsignal y_s. Der Messwert x_{max} bei dem maximal erwarteten Inputwert y_{max} und der Messwert x_{min} bei dem mininmal erwarteten Inputwert y_{min} sind bekannt. Da die Steigung der Geraden in der unten ersichtlichen Abbildung konstant sein muss gilt:

$$\frac{y_{max} - y_{min}}{x_{max} - x_{min}} = \frac{x_s - x_{min}}{y_s - y_{min}}$$

Durch umformen erhält man somit:

$$x_s = \frac{y_{max} - y_{min}}{x_{max} - x_{min}} * (y_s - y_{min}) + x_{min}$$



y_{max}	Maximaler Strom- oder Spannungsinput
y_{min}	Minimaler Strom- oder Spannungsinput
y_s	Aktueller Strom- oder Spannungsinput
x_{max}	Messwert bei maximalem Strom- oder Spannungsinput
x_{min}	Messwert bei minimalen Strom- oder Spannungsinput
x_s	Zu berechnender Messwert