Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

HAW HAMBURG

MASTERTHESIS

Hochschule für Angewandte Wissenschaften Hamburg
Fakultät Life Sciences

Topic:
Initial Design and Implementation of a Control System for a Hydrogen-Based Microgrid

---

To obtain the Degree:
Master of Science (M.Sc.)

On the Major:
Renewable Energy Systems - Environmental and Process Engineering

---

Submitted by:
Jan Moritz Dehler
Student ID:
[redacted]

Date of Submission:
1st of May 2025

1st Supervisor: **Prof. Dr. Carsten Frank (HAW Hamburg)**
2nd Supervisor: **Prof. Dr. Robi Banerjee (Hamburger Sternwarte)**

Abstract

The increasing integration of renewable energy sources into modern power systems necessitates innovative control solutions, particularly in microgrids operating under off-grid conditions. This thesis presents the initial design and implementation of a control and monitoring system for a hydrogen-based microgrid, currently under development on the area of the Hamburg Observatory (*Hamburger Sternwarte*). The system serves as a prototype for a backup power solution at the TIGRE Observatory in La Luz, Mexico, an observatory frequently affected by power outages.

To address the challenges of off-grid reliability, a hybrid energy storage architecture was developed, combining photovoltaic generation, lithium-ion batteries, electrolyzers, and a PEM fuel cell. The planned control infrastructure is centralized around a Raspberry Pi 4, which manages data acquisition via multiple communication protocols. Including CAN bus, Modbus RTU/TCP, and Ethernet, while MQTT serves as a higher-level, publish-subscribe communication layer. Time-critical sensor data and safety-relevant operations are delegated to Arduino-based microcontrollers. A custom, GPIO-based, interrupt-driven interface was implemented between the Raspberry Pi and the Arduino Due to enable a fast and error-minimized data transfer.

The software framework was developed using a modular architecture and implemented in Python and C++, comprising over a dozen dedicated scripts. These include programs for interfacing with the key hardware components (e.g., fuel cell, electrolyzers, multimeters) as well as modules for data logging, plausibility validation, and real-time plotting. Structured classes handle data acquisition, normalization, and error detection. All scripts operate asynchronously via MQTT, enabling scalable and decoupled functionality. Core management routines coordinate parallel script execution, system diagnostics, and watchdog-supervised runtime monitoring. In addition, a modular EMS logic was implemented as an MQTT-subscribing component, forming the foundation for future automation of hydrogen production and consumption control.

A comprehensive requirements analysis was conducted, addressing key aspects such as communication compatibility, fault management, system stability, and resource efficiency. The proposed energy management strategy prioritizes battery use and activates hydrogen-based systems for long-term load balancing. Overall, this work demonstrates the technical feasibility of a flexible, low-cost control system for hybrid microgrids and provides a foundation for future optimization, field deployment, and integration of additional components such as graphical interfaces and further implementation of sensor networks and interfaces.

Acknowledgments

Hiermit möchte ich mich herzlich bei alldenjenigen bedanken, die mir bei der Erstellung meiner wahrscheinlich letzten Studienarbeit geholfen haben, sei es Korrekturhilfen, persönliche oder auch fachliche Unterstützung.

Auch wenn das letzte Jahr aufgrund von diversen Umständen nicht immer einfach war, konnte ich mich auf den Zusammenhalt der Familie und guter Freundschaften stets verlassen. Dies hat mich durch diese Zeit getragen und ich bin dankbar, nun mit Zuversicht in ein neues Kapitel blicken zu können.

Ebenso danke ich den Mitarbeitern der Hamburger Sternwarte, die dafür gesorgt haben, die stets für ein angenehmes Arbeiten gesorgt haben und auf denen ich immer wieder auf spontane, fachlich fundierte Hilfe zurückgreifen konnte.

Einen besonderen Dank gilt an meine Mutter, auf die mir so oft mit gutem Rat und Zuwendung zur Seite steht.

Auch meinem Vater verdanke ich vieles, sei es seine unermüdliche Hilfsbereitschaft und das Gefühl, in ihm einen festen Anker in unserer Familie zu haben.

Ebenso möchte ich mich meiner Freundin bedanken, die mir so viel Kraft und Positivität gibt und die mich somit sehr bei dieser Arbeit unterstützt hat.

I would like to express my sincere thanks to everyone who helped me with what is probably my last student research project, whether through proofreading, personal support, or professional assistance.

Even though the last year has not always been easy due to various circumstances, I have always been able to rely on the support of my family and good friends. This has carried me through this time, and I am grateful to now be able to look forward to a new chapter with confidence.

I would also like to thank the staff at the Hamburg Observatory, who always ensured that I had a pleasant working environment and who were always on hand to provide spontaneous, expert help.

Special thanks go to my mother, who so often supports me with good advice and affection.

I also owe a great deal to my father, both for his tireless helpfulness and for giving me the feeling of having a solid anchor in our family.

I would also like to thank my girlfriend, who has given me so much strength and positivity and has thus supported me greatly in this work.

Table of Contents

List of Figures

List of Tables

List of Abbreviations

Abbreviation	Meaning
AC	Alternating Current
ADC	Analog-to-Digital Converter
AEM	Anion Exchange Membrane
AMQP	Advanced Message Queuing Protocol
BMS	Battery Management System
CAN	Controller Area Network
CoAP	Constrained Application Protocol
CPU	Central Processing Unit
CSV	Comma-Separated Values
DC	Direct Current
DER	Distributed Energy Resources
DES	Distributed Energy Storage
EMS	Energy Management System
GPIO	General Purpose Input/Output
HAT	Hardware Attached on Top (expansion board for Raspberry Pi)
HTTP	Hypertext Transfer Protocol
I^2C	Inter-Integrated Circuit
LED	Light Emitting Diode
MQTT	Message Queuing Telemetry Transport
MPPT	Maximum Power Point Tracking
PEM	Proton Exchange Membrane
PV	Photovoltaic
RAM	Random Access Memory
RTU	Remote Terminal Unit
SOC	State of Charge
SPI	Serial Peripheral Interface
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
UPS	Uninterruptible Power Supply
UART	Universal Asynchronous Receiver Transmitter
VBA	Visual Basic for Applications

1 Introduction

The transition to renewable energy technologies is primarily driven by the urgent need to address climate change and reduce dependency on fossil fuels (IEA , 2021, p. 3). However, renewable energy systems frequently face challenges such as fluctuating energy availability, a discrepancy between power generation and demand, as well as reliability issues, particularly in remote or off-grid areas where access to conventional infrastructure is limited or unstable (Solomon, 2019; Williams, et al., 2015).

These challenges are further amplified by the limitations of the traditional power grid architecture: the centralized structure of today's electrical grids remains largely consistent with the original configurations introduced over a hundred years ago. In principle, electricity is produced at a limited number of large-scale facilities and transmitted over long distances to dispersed consumers. This approach tends to lack flexibility and reliability, as real-time monitoring, regulation, and control rarely extend to lower grid levels. Additionally, integrating non-conventional or renewable energy sources often proves technically demanding. One practical response to these challenges is the incorporation of decentralized small-scale energy sources throughout the grid infrastructure (Kwasinski, Weaver, & Balog, 2016, pp. 3-4). Notably, placing generation facilities in closer proximity to consumption sites can significantly ease power flow across transmission and distribution systems, reducing energy losses and potentially offsetting the need for traditional grid investments. Moreover, local generation can enhance service reliability for users. In critical situations, microgrids can help stabilize the system by reducing network congestion and facilitating fault recovery (Schwaegerl & Tao, 2014, p. 3).

Their potential applications are wide-ranging, from development initiatives that bring power to underserved communities, to remote locations like islands or Antarctic research stations, and even to public or grid-connected systems capable of running independently during blackouts or emergencies. Naturally, microgrids must address the ongoing challenge of matching energy supply with demand. It is rarely possible to generate precisely the amount of energy needed at the exact time it is required, which is why on-site storage is crucial - not only to bridge gaps during outages or voltage drops, but also to manage load variations and respond effectively to emergencies. There are various types of energy storage technologies, each with specific strengths and weaknesses regarding power density, discharge rates, durability, and efficiency. Rechargeable batteries are a common choice due to their simplicity. Typically, battery systems operate with significantly higher nominal efficiency compared to hydrogen-based alternatives (Ferrario, et al., 2020) and are capable of delivering rapid responses to load changes (IRENA, 2019, p. 10).

Yet these systems come with limitations such as self-discharge, finite lifespan, and recycling challenges. One clean energy source that has gained strong momentum is green hydrogen. It offers microgrids a sustainable and emission-free option for long-term energy storage, addressing the issue of seasonal or long-duration storage, something that conventional batteries cannot manage. Its benefits include, among others, a high energy density and a long storage life. Thus, hybrid energy storage systems may provide "the best of both worlds", by combining the strength of battery- and hydrogen storage systems (Enapter S.r.l., 2020, pp. 7, 5, 9 & 6). Figure 1 illustrates an exemplary interaction of energy flows through a microgrid with renewable production and hybrid storage solution.

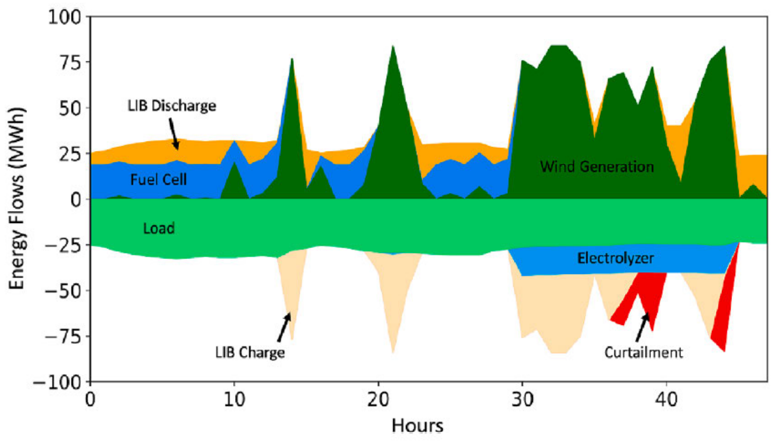


Figure 1: An exemplary illustration of hourly energy flows in a microgrid with a hybrid H_2-lithium-ion battery energy storage system (Giovanniello & Xiao-Yu, 2023).

These interdependencies between generation, storage, and consumption within a microgrid highlight the need for a dedicated control solution. Effective energy management is only possible through a supervisory system capable of coordinating all components cf. (IEEE, 2017, p. 13) and as (Enapter S.r.l., 2020, p. 25) further notices, a functional telemetry is essential for managing microgrid operations and equipment. As systems evolve, having access to historical performance data - covering energy usage, errors, failures, and more - becomes increasingly important. These data must be collected consistently, validated, and securely stored to support ongoing optimization and decision-making.

A real-world example of the above-mentioned challenges is provided with this thesis, focusing on the implementation of a conceptual design of a control and supervision software framework for a hydrogen-integrated microgrid, with special attention to communication and data handling between components.

1.1 Problem Statement and Application Context

This work forms a building block for the development of a microgrid based on a hydrogen-powered energy storage system, which is currently being implemented in the area of the Hamburg Observatory (*Hamburger Sternwarte*). The Hamburg setup is intended to serve as a prototype for a new backup power solution to be deployed at the local power grid of the "TIGRE" Observatory in La Luz (Mexico). This observatory is a collaborative project between the University of Hamburg, the University of Guanajuato, and the University of Liège (Schmitt, et al., 2014). In the past, the site has experienced repeated power outages (González-Pérez, et al., 2022), thus, a reliable power supply is essential for ensuring continuous operation. For further details regarding the hardware setup of the plant, refer to section 4.

1.2 Project Requirements and Contributions to the Project

During the most recent development phase preceding this work, the majority of the hardware setup has already been installed, and most of the utility hardware was available for testing. Most of the main components, responsible for the overall load management can be considered as material for standard industrial application and all controllable units integrate control electronics and provide predefined interfaces. The Microgrid system described here can also be referred to as a hybrid-storage microgrid, see also (Giovanniello & Xiao-Yu, 2023). A comprehensive monitoring and controlling solution for the microgrid as a whole was missing, prior to the thesis. Thus, this thesis focuses on the development and implementation of software structure realized on a Raspberry Pi, responsible for the acquisition of data from all of the system components that provide interfaces, including CANBUS, Modbus RTU/TCP and Ethernet. The software on the Raspberry Pi was written in Python. Furthermore, the MQTT (Message Queuing Telemetry Transport) protocol was chosen to provide for a higher-level communication protocol to abstract and distribute acquired data outgoing from the different interfaces across different software components. Additionally, a concept for an energy management algorithm was developed. Moreover, an extension of software responsible for acquisition of sensor data on an Arduino DUE and serial communication of the Raspberry Pi with the Arduino DUE, via a self-designed interrupt-based data and clock interface using GPIO (General Purpose Input/Output) pins, was realized. The programming language used for the Arduino DUE software was C++. By creating a well-documented and extendable framework, this thesis lays the foundation for future development. It is important to emphasize that this work represents an initial concept rather than a final, optimized solution. The primary objective is to demonstrate the technical viability of the system, providing a basis for further advancements.

1.3 Syllabus

The thesis begins with an extensive literature review, providing a detailed description of all employed technologies, presented systematically to lay the foundation for understanding the grid structure and software concept. This theoretical section will comprehensively cover each technology relevant to the thesis. Subsequently, the setup of the grid at both the La Luz and Hamburg sites is described, focusing on hardware components and their functional relationships. Following this, a requirement analysis is conducted to define the key criteria for the software design, including communication compatibility, modular structure, fault management, and stable and efficient system operation. The subsequent chapters present and explain the developed software concepts and their implementation on both the Arduino DUE and Raspberry Pi platforms. The thesis concludes with a summary and outlook, reflecting on the project's results and outlining directions for future development.

2 Theory

2.1 Microgrids

2.1.1 The Role and Structure of Microgrids

A microgrid can be described as a scalable, local energy infrastructure that integrates electrical with distributed generation technologies. Its structure is composed of decentralized energy resources-including demand response mechanisms, storage technologies, and generation assets-combined with robust communication, network infrastructure, and secured information systems (Federau, 2016, p. 31).

2.1.2 Key Microgrid Features and Operating Modes

Notable characteristics of microgrids include their autonomous capability, meaning they may operate independently from the central grid when needed. Also, they require local voltage and frequency regulation to ensure grid stability through advanced management systems. Moreover, they can be characterized by their ability to integrate into existing network infrastructures (Hesami, et al., 2024). In general, microgrids can function in three primary operating states: 1st: grid-connected mode, where the system remains synchronized with the public network, enabling bidirectional power exchange while the grid dictates voltage and frequency parameters 2nd: island mode, where the microgrid is physically and electrically separated from the upstream grid, often triggered by disturbances and 3rd: transition mode, a dynamic state of synchronized switching between connected and islanded state (Federau, 2016, pp. 35-36).

2.1.3 Classification of System Components

A microgrid consists of several components, which can be categorized and distinguished from one another, as shown in table 1. Particularly noteworthy in this regard is the microgrid control system, which enables the microgrid to operate autonomously or in grid-connected mode, with the capability to manage itself and to connect to or disconnect from the main distribution grid for power exchange and ancillary service provision (IEEE, 2017, p. 13). Integrated within this control infrastructure of the MCS is the Energy Management System (EMS), which supports the use of renewables, reduces peak grid stress, and minimizes emissions from conventional sources. In events such as error conditions or exceeded constraints, the control logic ensures the system returns to a safe and predefined configuration. Such error conditions may include exceeding the limit value of the operating mode, exceeding a limit value of the system fuse, communication failure or Microgrid storage has exceeded its lower limit (Federau, 2016, p. 122).

Table 1: Classification of components commonly found in a microgrid [3, pp. 12 & 27-28].

Hardware responsible to connect the microgrid to the public grid at Point of Interconnection (breakers and disconnects)			
The microgrid control system (MCS) and further control elements			
Local distribution system hardware, e.g. capacitors, switchgear, transformers			
Physical devices	Distributed Energy Resources (DER)	Distributed Generation (DG)	Dispatchable units, that are able to be controlled by the MCS
			Non-dispatchable units, which cannot be controlled by the MCS
		Distributed energy storage (DES)	
	Loads	Critical loads (Loads that have to be served at all of the microgrid's normal operating modes)	
		Priority loads (Loads that can be reduced when required but should ideally remain supplied.)	
		Controllable loads (Loads with adjustable power levels and the ability to be cyclically disconnected if required)	
		Interruptible loads (Loads that are fully interruptible and may be shed at any moment as needed)	
		Diversion & dump loads (Loads that operate when surplus generation is available and curtailing production is either impractical or economically disadvantageous)	

2.1.4 Control Structure and Management Functions

To operate as a cohesive and controllable unit, the microgrid requires a dedicated control system. This system orchestrates energy flow, ensures smooth transitions between operational states, and manages both internal resources and the connection to the main grid. It enables automated dispatch of generation and storage assets, as well as load coordination, depending on reliability metrics or economic signals. The functional logic of a microgrid can be organized into four control layers, or function blocks. These represent different scopes and timeframes of operation, but their practical implementation can vary depending on the specific system design and control architecture (IEEE, 2017, pp. 12, 2, & 37-38), see table 2.

Table 2: Function Assignment of a microgrid in the format of function blocks (IEEE, 2017).

Block No.	**Explanation**
4	Grid-interactive control deals with higher-level objectives such as market participation or grid services and typically operates on longer time scales.
3	Supervisory control services, overseeing the entire microgrid, managing dispatch, scheduling, and mode transitions (grid-connected vs. islanded) operating over minutes to days.
2	Local-area control coordinates devices within a subsystem or zone, such as a building. This level may include load aggregators or localized EMS units and works over seconds to hours.
1	Device-level control handles real-time tasks such as voltage or current regulation and protection. These functions are directly tied to the hardware-e.g., inverters, storage controllers or switches-and operate within milliseconds to seconds.

It is important to note that these functions are conceptual and may be performed by different physical or software components-centralized, decentralized, or hybrid-depending on the microgrid's architecture and operating scenario. The options in regard to the grouping of functions along the control systems can be found in figure 2.

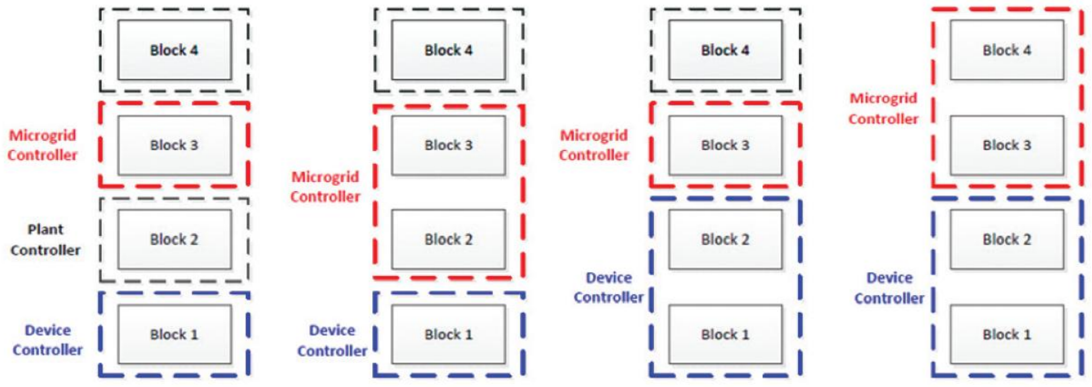


Figure 2: Options regarding the grouping of functions of different control entities (IEEE, 2017, p. 37).

2.1.5 Application examples of Similar Microgrid Projects

(Ziogoua, et al., 2011) presents the design and evaluation of an automation and operation control system for a stand-alone renewable energy-based power system in Neo Olvio, Greece. The system integrates renewable energy generation (photovoltaic with 10 kWp and wind energy 3 kWp) lead-acid batteries (3000 Ah, 48 V nominal DC bus). Battery operation is constrained by a lower voltage limit of 48.2 V, corresponding to a SOC of 75 to 80%, to protect against deep discharge. Excess energy powers a Proton Exchange Membrane (PEM) electrolyzer (1.05-4.2 kW), which produces hydrogen stored in tanks up to 30 bar (~35 kWh equivalent). The hydrogen feeds a 4 kWp PEM fuel cell during energy deficits. A diesel generator serves as backup in emergency conditions. The automation system uses various network protocols (CAN, Ethernet, RS232, Profibus) to enable real-time, remote monitoring and control. A central data acquisition unit evaluates subsystem performance and logs critical variables. The paper highlights the challenge of integrating heterogeneous subsystems, through the implementation of a combination of open communication, modular control logic and centralized remote monitoring implemented since the system is designed to be adaptable to future components, load scenarios or modified operating strategies. The implemented energy management strategy (EMS) relies on the SOC of the accumulator and operates based on a hierarchical control algorithm. The EMS was validated through simulated scenarios under varying load demands (1 kW, 1.5 kW, 2 kW), aiming to: Optimize the contribution of each subsystem to reliable operation, avoid excessive wear (e.g., battery cycling) and preserve the system's eco-friendly characteristics.

In (Agbossou, et al., 2004), the development and experimental validation of an autonomous renewable energy (RE) system created by the Hydrogen Research Institute (HRI) is presented. The energy management is handled by a control system consisting of a master controller and secondary microcontrollers. These regulate the operation of the electrolyzer, and the proton exchange membrane fuel cell (PEMFC) based on the energy level of the battery bank (energy buffer). The system operates on a 48 V DC bus, which may vary up to 56 V due to load fluctuations and component ripple effects. A double hysteresis control strategy is implemented to manage the activation thresholds for the electrolyzer and PEMFC-ensuring stable and efficient switching by using different energy levels for turning devices on and off. Real-time data from the RE system are used to adapt the control algorithm, ensuring system reliability under varying load profiles and environmental conditions.

In (Little, et al., 2007), a stand-alone power system was developed integrating multiple renewable energy sources and uses hydrogen as a long-term energy storage medium. The system combines wind, solar, hydro, and a combined heat and power (CHP) unit with hydrogen storage, fuel cells, and a high-voltage central DC bus to connect all components-including generators, loads, and storage. The system uses Zebra high-temperature batteries, and the hydrogen system includes a high-pressure electrolyzer, compression unit, and storage tanks, as well as two PEM fuel cells-one for combined heat and power, and another for backup power. The electrolyzer must avoid frequent on-off cycling to prevent catalyst degradation. To manage this, an advanced battery buffer was added, and the control strategy focuses on maintaining a high state of charge, while minimizing component cycling-especially for the electrolyzer and fuel cells. System modeling in MATLAB Simulink helped evaluate energy efficiency and component wear over time. It showed that a wide hysteresis band reduces electrolyzer cycling, and that the overall round-trip electrical efficiency of the hydrogen storage is about 25%.

2.2 Photovoltaics

A photovoltaic (PV) system is based on solar cells, typically made of silicon. Their functionality results from the process of doping, which creates a p-n junction and an internal electric field. Sunlight excites electrons, which are guided by this field to generate voltage. The resulting current depends on light intensity and cell size. To increase voltage, cells are connected in series to form modules. The output under standard test conditions (STC) is given in Watt-peak (Wp). Several modules are combined into strings and connected to an inverter (Mertens, 2014, pp. 13–14), see also section 2.3.

2.3 Inverters

Inverters are power electronic circuits whose main task is to convert direct current (DC) into alternating current (AC) (Böcker, 2019, p. 9). In literature, a general distinction is made between line-commutated and self-commutated inverters(cf. Böcker, 2019). However, only self-commutated inverters will be discussed in the following based on the application case in the thesis.

2.3.1 Self-commutated inverters (self-commutating inverters)

Self-commutated inverters differ from line-commutated types by utilizing semiconductor switches like IGBTs or MOSFETs, which can be actively turned on and off via control signals. Furthermore, these inverters show strong resilience to disturbances originating from the utility grid (Ishikawa, 2002, p. 4). In contrast to systems that rely on the grid to function, self-commutated inverters are also capable of operating independently, making them suitable for standalone applications where they supply power directly to local loads without requiring grid support (Eltawil & Zhao, 2010, p. 123).

2.3.2 Hybrid inverter

A hybrid inverter is an inverter type that combines the functions of a PV inverter and a battery inverter in a single device. Technically, it is usually a DC-coupled system: both the PV modules and the battery storage are connected together to a DC intermediate circuit of the hybrid inverter (Swissolar, no date, p. 3).

2.4 UPS Systems

Uninterruptible Power Supply (UPS) systems typically draw their energy from batteries and are generally designed for short-term bridging durations. During this limited time, either the connected systems can be brought into a safe shutdown state, or backup generators can be started to take over the power supply. An AC-based UPS features both an AC/DC and a DC/AC converter, with the battery (DC) located in between. This configuration is commonly referred to as an AC UPS with battery support in the intermediate circuit. It ensures uninterrupted power supply and additionally protects against poor grid quality or voltage fluctuations. UPS systems are primarily dimensioned for short operating times in the range of a few minutes. For smaller loads, durations of up to several hours may be feasible. However, the integrated batteries are subject to continuous aging and gradually lose capacity over time (Paul & Leu, 2017, pp. 158-160).

2.5 Electrolysis

2.5.1 General Information

Electrolysis describes the electrically driven breakdown of a substance through a redox mechanism and acts as the reverse process of what occurs in energy-producing devices like batteries, rechargeable cells, or fuel cells (also refer to sections 2.6 & 2.7). A steady reaction between the electrodes requires a constant supply of direct current. At the cathode (negative terminal), electrons are absorbed during the reduction process (lowering the oxidation number), whereas at the anode (positive terminal), electrons are emitted in an oxidation reaction (increasing the oxidation state). In the process of water electrolysis, H_2O is always decomposed, yielding hydrogen gas at the cathode and oxygen gas at the anode. Adding acids like HCl, bases like KOH, or soluble salts like NaCl enhances the conductivity of the solution (Zapf, 2017, p. 167).

2.5.2 The Anion Exchange Membrane (AEM) Electrolysis

Cell Structure

Anion-exchange membranes typically consist of a hydrocarbon polymer framework with an additional side chain of functional groups that enable anion exchange. Inside the cell, an alkaline medium is established through the membrane interface through positively charged functional groups located either on the polymer framework or on its polymer side chains. In the electrolysis process, hydroxide ions are conducted through this membrane along water molecule chains. This transport mechanism is driven by sequences of hydrogen bond creation and disruption. A crucial advantage of anion-exchange membrane electrolysis is the feasibility of employing catalysts that are free from platinum group metals for hydrogen and oxygen gas formation, e.g. nickel-based composites are commonly utilized for the hydrogen-side reaction (Cavaliere, 2023, pp. 290, 300 & 287; Miller, et al., 2020).

Electrochemical Reactions

Applying a potential across the anion-type membrane triggers electron flow from anode to cathode, coupled with water-splitting reactions occurring at both sides (Cavaliere, p. 291):

$$H_2O \rightarrow H_2 + \frac{1}{2}O_2$$

At the positive electrode (anode), OH⁻ ions from the surrounding medium undergo oxidation, resulting in oxygen gas as part of the oxygen-evolving process (Cavaliere, pp. 291 & 292):

$$2OH^{\ominus} \rightarrow \frac{1}{2}O_2 + H_2O + 2e^{\ominus}$$

On the negative electrode (cathode), hydrogen is generated by reducing water molecules, producing H_2 along with hydroxide ions during the corresponding electrochemical step (Cavaliere, p. 292):

$$2H_2O + 2e^{\ominus} \rightarrow H_2 + 2OH^{\ominus}$$

A general overview of the cell structure can be found in figure 3. Furthermore, the table 3 provides a general overview regarding the materials used for the main cell components and operating parameters.

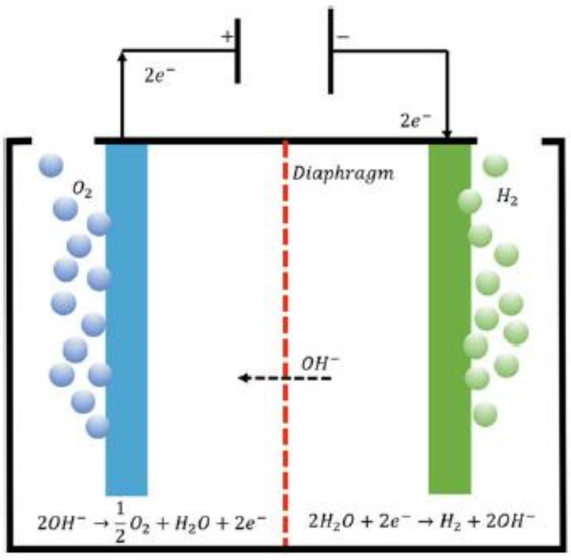

+
−
$2e^-$
$2e^-$
O_2
Diaphragm
H_2
OH^-
$2OH^- \rightarrow \frac{1}{2}O_2 + H_2O + 2e^-$
$2H_2O + 2e^- \rightarrow H_2 + 2OH^-$

Figure 3: Schematic representation of an AEM cell structure including chemical reactions (Cavaliere p. 291).

Table 3: General overview of materials and certain operating parameters for the AEM water electrolysis (Miller, et al., 2020).

Electrolyte	Anion exchange ionomer (e.g. AS-4) + optional dilute caustic solution
Cathode	Ni and Ni alloys
Anode	Ni, Fe, Co oxides
Operating temperature (C)	50–60
Operating pressure (bar)	1–30
Production rate (Nm3 h1)	<1
Gas purity (vol%)	>99.99

2.6 Fuel Cells

2.6.1 General Information

When the electricity supply of electrolysis is interrupted, and hydrogen and oxygen will be provided to continuously flow around the electrodes, the electrolysis will process in reverse, such that the hydrogen-oxygen fuel cell generates a voltage of its own. The redox equations of the electrochemical processes at the electrodes of the hydrogen-oxygen cell are as follows (Kurzweil, 2016, p. 3):

$$\oplus\ Cathode: O_2 + 4\,H^{\oplus} + 4\,e^{\ominus} \rightleftharpoons 2\,H_2O \qquad E^0 = 1.23\,V$$

$$\ominus\ Anode: \qquad 2\,H_2 \rightleftharpoons 4\,H^{\oplus} + 4\,e^{\ominus} \quad E^0 = 0.00\,V$$

Whereas the standard electrode potential (E0) indicates the tendency of a substance to form ions in aqueous solution under standard conditions (25 °C, 1 atm), and the difference between two E_0 values determines the open-circuit voltage of an electrochemical cell (Kurzweil, 2016, p. 4). The hydrogen electrode forms the negative terminal (Anode), while the oxygen electrode (Cathode) forms the positive terminal. In electrochemical cells, reduction (electron uptake) occurs at the cathode, while oxidation (electron release) occurs at the anode. The oxygen molecule O_2 is split; its oxidation state changes from 0 to -2. This means each oxygen atom accepts two electrons, see (Kurzweil, 2016, p. 3):

$$O_2 + 4\,e^{\ominus} \rightarrow 2\,O^{2\ominus} \quad (5)$$

The uptake of electrons and the formation of negatively charged particles (anions) is typical of nonmetals. Metals and hydrogen, on the other hand, tend to form positively charged ions (cations) in chemical reactions. The H_2 molecule dissociates into unstable H atoms, each of which immediately gives up one electron (Kurzweil, 2016, p. 3):

$$H_2 \rightarrow 2\,H^{\oplus} + 2e^{\ominus} \quad (6)$$

The oxidation state of the hydrogen atom changes from 0 to +1 during this oxidation (electron loss). In the overall cell reaction $2H_2 + O_2 \rightarrow 2H_2O$, four electrons are exchanged in the redox equations from two hydrogen molecules, i.e., z = 2. The Gibbs free reaction enthalpy describes the usable energy per mole of fuel gas (Kurzweil, 2016, pp. 3-4):

$$\Delta G^0 = -zF\Delta E^0 = -\frac{4}{2}\cdot 96485\frac{C}{mol}\cdot 1{,}23\,V \approx \frac{-475\,kJ}{2\,mol\,H_2} = -237\,\frac{kJ}{mol} \quad (7)$$

The theoretical capacity - that is, the usable electrical charge of a cell reaction - is zF, where F is Faraday’s constant. A redox reaction involving the transfer of one electron yields 96,485 As/mol = 26.8 Ah/mol. Higher voltages are achieved by connecting multiple fuel cells in series. The resulting operating voltage equals the number of cells multiplied by the single-cell voltage (Kurzweil, 2016, p. 4).

2.6.2 The Polymer Electrolyte Membrane (PEM) Fuel Cell

In PEM fuel cells, the polymer membrane serves as electrolyte, catalyst carrier, and gas separator. These are typically 50-150 µm thick films made from perfluorinated and sulfonated polymers, known as proton exchange membranes (PEMs) (Kurzweil, 2016, p. 79). The basic schematic of the PEM cell’s buildup can be observed in figure 4.

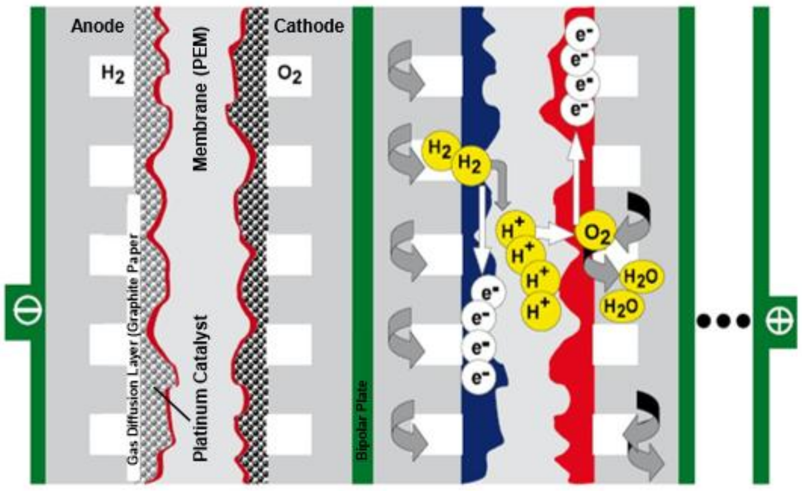


Figure 4: Outline of the basic design of the PEM fuel cell. Modified from (Kurzweil, 2016, p. 77).

The electrolyte is a proton-conducting polymer membrane, with protons (H^+) or hydronium ions (H_3O^+) serving as the charge carriers. Typical operating temperatures are 60-70 °C, with some systems reaching up to 120 °C. The fuel on the anode side is hydrogen or reformate gas, and the oxidizing agent on the cathode side is oxygen or air, which is humidified (Kurzweil, 2016, p. 78).

Operating Behavior

Several factors significantly influence cell voltage, including: the humidity, pressure and excess of air, as well as the operating temperature (Kurzweil, 2016, p. 100). In addition to electrical power, a significant amount of heat is produced during operation, which should be utilized to improve both efficiency and overall energy utilization. Fuel cells are particularly well-suited for steady-load applications (such as continuous battery charging). Frequent load cycling reduces the lifetime of the membrane significantly (Bogensperger, 2022, p. 107).

2.7 Lithium-Ion Batteries

Lithium-ion storage systems typically operate within a charge range of 20% to 80%, as they exhibit reduced capacity losses in a limited SoC range. This makes them particularly suitable for applications where solar power generated during daylight hours should be stored and used at night-indicating regular daily charge-discharge cycles. Compared to lead-acid batteries, they offer a more compact design, deliver higher energy density, and exhibit a more stable terminal voltage relative to their state of charge (SoC) (Bogensperger, 2022, p. 108; Gauthier, et al., 2022).

2.8 Embedded Systems

2.8.1 Technology of Embedded Systems

An embedded system is a specialized device incorporating a processor intended for a particular purpose. Typically, users are neither able nor expected to upgrade the hardware or software or modify its intended operation. The range of tasks that the processor may handle includes data analysis and decision-making, time management in various roles such as measurement or task synchronization, and the execution of real-time interactions involving data processing in areas like sound, image, radar, communication, and networks.

Such systems are characterized by limitations, including compact physical dimensions and their usage in environments where reliability is vital, especially in safety-critical applications that demand real-time performance, whereas the term "real time" refers to scenarios where actions must be completed within a clearly defined short time frame. For embedded systems, this implies responding to urgent situations within a strict deadline. Internal peripherals such as the nested vectored interrupt controller (NVIC) and the processor communicate via the private peripheral bus (PPB) which ensures significantly reduced latencies during execution of interrupt routines. An interrupt, whether initiated by hardware or software, is a software-executed function. One variant occurs due to I/O activity: for example, a hardware signal may indicate new input, which triggers a response where software reads the data and stores it in a shared memory structure. Later, this data can be accessed if available (Valvano, 2017, pp. 20-21, 18, 24-25 & 105). In numerous embedded systems, AVR Microcontrollers are used, as is the case with Arduino Family (Meroth & Sora, 2023, p. 1). In such microcontroller systems, interaction with the outside world relies heavily on input and output management. External modules connect to the controller using groups of lines known as ports, with each individual wire referred to as a pin. Ports combine multiple pins by shared purpose. Among these, the GPIO connections are versatile and can be set for digital input or output, analog input, or specific communication protocols such as the UART (Universal Asynchronous Receiver Transmitter) (Valvano, 2017, p. 35). Furthermore, analog-to-digital converters (ADCs) serve a crucial role for such systems to interface with the analog world by translating continuous sensor signals in the form of analog voltage signals into digital data that the processor can interpret (Bähring, 2010, p. 393). According to (Mehalaine, et al., 2024), watchdogs can be used as a software-based error detection in an embedded system, whereas it is considered as a counter that starts from a set value and reduces at a constant pace. It helps identify when the processor malfunctions due to various causes. The principle is that running processes must regularly refresh the timer by either writing a defined number to a register or invoking a specific routine. In this manner, the timer functions as a monitor to verify the processor's proper behavior.

2.8.2 Arduino

Arduino represents an open-source ecosystem employing boards equipped with programmable microcontrollers. These boards can be easily integrated with computers, networks, and other equipment. Common variants of Arduino boards include models like Uno, Mega, Nano, Leonardo, and Due. Among them, the Arduino Due stands out by incorporating a more powerful 32-bit Atmel SAM3X8E ARM Cortex-M3 processor. Compared to other boards in the series, the Due delivers faster performance and expanded memory capacity. It operates at 84 MHz and includes 512 KB of flash storage, enabling it to handle more advanced applications and larger-scale developments. The board is equipped with 54 digital I/O connections, of which 12 support PWM output. It also includes 12 analog input pins, 2 analog outputs, and multiple serial communication interfaces. In general, Arduino boards benefit from a large base of open resources, a wide selection of accessories and libraries, and an active community (Zhou, 2023, pp. 13-14; A_24).

2.9 The Raspberry Pi

A Raspberry Pi can be described as a fully assembled electronic board. Whereas the Raspberry Pi 4 Model B can be considered a versatile general-purpose platform. It features a quad-core processor running at 1.5 GHz and comes with 1, 2, 4, or 8 GB of memory. Furthermore, it includes four USB connectors (two of them supporting USB 3.0), an Ethernet interface, and dual micro-HDMI outputs for video. Its possible use cases are ranging from acting as a desktop computer, a gaming console, to serving as a controller for both sensors and actuators (Monk, 2023, pp. 13, 13, 20, 173, 503 & 430). The Raspberry Pi runs on a free GNU/Linux open-source operating system. Storage is managed via a Secure Digital (SD) memory card, which also contains the operating system. There is no built-in hard drive (Farrenkopf, 2014, p. 24). Moreover, it provides 28 usable GPIO pins for external interfacing [A_33][1].

2.10 Interfaces

An interface can be understood as the combination of I/O ports, hardware components, external circuitry, and software that together enable a computer system to interact with its environment (Valvano, 2017, p. 35). It functions as the gateway between systems, facilitating data exchange based on a defined set of operational rules. These rules, referred to as protocols, dictate how data is transmitted and interpreted during communication between different information-processing entities (Tröster, 2011, p. 499). Input/output interfaces can be broadly divided into categories based on the type of signal and method of transmission, for example parallel/digital interfaces, which transmit binary values concurrently across multiple lines, serial interfaces, where bits are conveyed sequentially over a single channel or analog interfaces, using electrical properties like voltage or current to represent data. In order to evaluate how well an interface performs, certain metrics such as bandwidth, latency, and priority are used, whereas bandwidth represents the volume of data transferred over a specific time period. Latency, in contrast, refers to the time delay between a request for data and the moment that request is fulfilled (Valvano, 2017, pp. 36 & 363).

2.11 Protocols

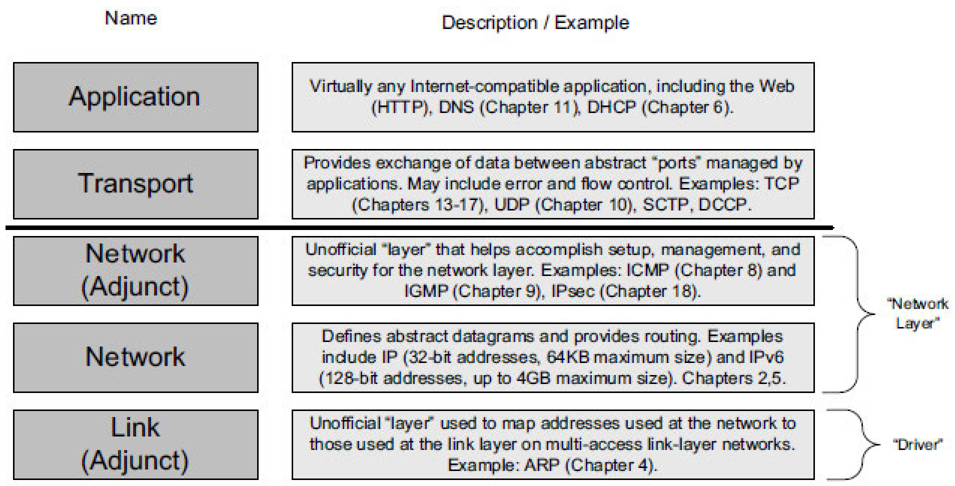
For devices or systems to successfully exchange information, they must be capable of interpreting each other's messages correctly. To achieve this, shared rules are established that govern how communication takes place. These sets of rules are known as protocols. They specify both the structure (syntax) of valid messages and the semantics, which define the vocabulary and the meaning behind each message (Baun, 2012, p. 31). Thus, a software protocol determines the nature of the signals and data units (telegrams) sent across interfaces and outlines how the flow of data is regulated between systems (Tröster, 2011, p. 500).

2.11.1 Protocol stacks: The TCP/IP reference model

Communication systems are typically structured using layered models to manage the complexity and requirements of modern computer networks. One of the most widely recognized models is the TCP/IP reference model, see figure 5.

[1] All references marked with [A_...] refer to documents or files included in the digital appendix

This stack divides communication into five distinct layers: application, transport, network, link, and physical. E.g. on the application layer, data is generated by an application running on the sender's device, and the transport layer provides end-to-end communication between devices. The two main protocols used at this layer are the Transmission Control Protocol (TCP), which ensures reliable, ordered data transmission, and the User Datagram Protocol (UDP), which prioritizes low latency over reliability (Stevens & Fall, 2012, pp. 8, 16 & 15; Baun, 2012, p. 31).



Name	Description / Example	
Application	Virtually any Internet-compatible application, including the Web (HTTP), DNS (Chapter 11), DHCP (Chapter 6).	
Transport	Provides exchange of data between abstract "ports" managed by applications. May include error and flow control. Examples: TCP (Chapters 13-17), UDP (Chapter 10), SCTP, DCCP.	
Network (Adjunct)	Unofficial "layer" that helps accomplish setup, management, and security for the network layer. Examples: ICMP (Chapter 8) and IGMP (Chapter 9), IPsec (Chapter 18).	"Network Layer"
Network	Defines abstract datagrams and provides routing. Examples include IP (32-bit addresses, 64KB maximum size) and IPv6 (128-bit addresses, up to 4GB maximum size). Chapters 2,5.	
Link (Adjunct)	Unofficial "layer" used to map addresses used at the network to those used at the link layer on multi-access link-layer networks. Example: ARP (Chapter 4).	"Driver"

Figure 5: The Protocol layer structure of the TCP/IP protocol stack, including a brief description of the respective layers (Stevens & Fall, 2012, p. 14).

Within such a model, each layer performs specific tasks and interacts with its adjacent layers using well-defined interfaces. A protocol stack can be thought of as a collection of protocols assigned to each layer, with one protocol per layer managing data processing at that level. In this hierarchical framework, when a message is sent from an application on one computer to an application on another, it passes vertically through the protocol layers on the sender's side - from the application down to the physical layer. The message is then transmitted across the medium and processed in reverse order by the receiver, moving upward through its own stack until it reaches the target application (Meinel & Sack, 2012, pp. 32-35). This vertical flow involves a mechanism called encapsulation, where each layer appends its own control data - typically in the form of a header - to the payload received from the layer above. These headers assist in multiplexing during transmission and are later used to demultiplex and interpret the message correctly on the receiving side (Stevens & Fall, 2012, pp. 10-11; Meinel & Sack, 2012, p. 25). A visual representation of the principle of encapsulation can be observed in figure 6.

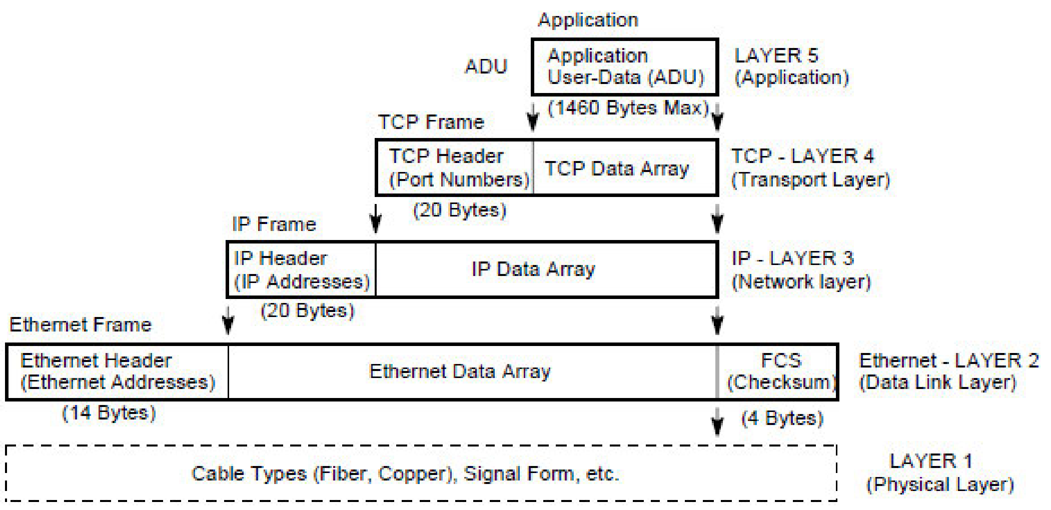

Application
ADU
Application User-Data (ADU)
(1460 Bytes Max)
LAYER 5 (Application)
TCP Frame
TCP Header (Port Numbers)
TCP Data Array
(20 Bytes)
TCP - LAYER 4 (Transport Layer)
IP Frame
IP Header (IP Addresses)
IP Data Array
(20 Bytes)
IP - LAYER 3 (Network layer)
Ethernet Frame
Ethernet Header (Ethernet Addresses)
Ethernet Data Array
FCS (Checksum)
(14 Bytes)
(4 Bytes)
Ethernet - LAYER 2 (Data Link Layer)
Cable Types (Fiber, Copper), Signal Form, etc.
LAYER 1 (Physical Layer)

Figure 6: Illustration of the encapsulation mechanism over the message packets (Acromag Inc., 2005, p. 11).

An example of protocol layering in practice is Modbus (see also section 2.11.5) an application-level protocol that remains independent of the underlying physical medium. It defines the structure and semantics of the information to be exchanged. The same protocol can operate across different physical standards - such as Modbus TCP/IP over Ethernet or Modbus RTU via RS-232 (see also section 2.11.7)- demonstrating the separation of logical communication rules from hardware interfaces (Acromag Inc., 2005, pp. 3-4).

2.11.2 Validation Checks in Protocols

Many protocols use the checksum procedure to validate incoming data packets. A checksum is a value calculated from a sequence of data (e.g. a message or file) and used to detect transmission or storage errors. The sender calculates the checksum from the data to be transmitted and sends it along with the data. The recipient performs the same calculation and compares the result with the received checksum. Simple checksums often consist of the addition of all bytes of a data packet. More complex methods such as the CRC (Cyclic Redundancy Check) interpret the data bits as a mathematical polynomial, perform a polynomial division with a predefined generator polynomial, and use the remainder of this division as a checksum. Checksums are used, among others, in the CANBUS, Modbus and TCP protocol (Meroth & Sora, 2023, pp. 167, 274, 168, 271 & 174).

2.11.3 Ethernet Communication Workflow

Compare to the previous section: once the Ethernet frame is created, it is transmitted over the network using several steps. Standard Ethernet devices contain a Network Interface Card (NIC), which is responsible for sending and receiving Ethernet frames (Stevens & Fall, 2012, p. 951). If two devices are directly connected via Ethernet, the NICs automatically detect the correct transmission mode (sending or receiving?), using a method called Auto-MDIX. In Ethernet networks, where data can be sent and received simultaneously (full-duplex) and devices may operate at different speeds, network switches may need to buffer frames temporarily. This occurs, for example, when multiple devices send data to the same output port. If the combined traffic exceeds the destination's link capacity, switches store the frames by recording MAC addresses in a filtering database table and forward frames coming from multiple sources to their correct ports (Stevens & Fall, 2012, pp. 98-99).

Packets from one element are sent to the appropriate destination, and if a collision occurs, the switch delays one packet to avoid the collision (Valvano, 2017, pp. 449-450).

2.11.4 Technology of MQTT

The Application protocol Message Queuing Telemetry Transport (MQTT) is a protocol built on a client-server architecture and utilizes a publish-subscribe paradigm. It operates over transmission protocols like TCP/IP or comparable alternatives that guarantee reliable, sequenced, and two-way data flow. Designed to be minimalistic, open, and easy to deploy, MQTT is particularly well-suited for resource-limited environments, such as machine-to-machine (M2M) systems, where reduced code size and efficient use of network capacity are critical (Banks & Gupta, 2015, pp. 1-2; Bandyopadhyay & Bhattacharyya, 2013). An MQTT network consists out of MQTT Clients and a Broker (also considered as an MQTT Server), whereas a client refers to any program or device utilizing MQTT (Mishra & A., 2020). In MQTT-based communication, brokers serve as the system's central hub. They manage the entire data exchange process by handling message routing and managing client interactions (Sallat, 2018, p. 52). The client initiates the network connection to the Server. It is able to: publish messages that may interest other clients, subscribe to receive messages on specific topics, unsubscribe from topics it no longer wishes to receive messages from and disconnect from the server. The server on the other hand acts as an intermediary that manages message distribution between clients that publish messages and clients that subscribe to topics. The server accepts connections from clients, processes and forwards messages to matching subscriptions and manages subscription requests. The server ensures that each subscribing client receives a copy of messages relevant to its subscriptions (Banks & Gupta, 2015, pp. 9-10).

Thus, a common MQTT setup includes sensors that periodically publish measurement data (payload) to specific topics. Devices interested in this data subscribe to the respective topics to receive updates whenever new data is published. The key advantage of this publish-subscribe model, as implemented in MQTT, is that data producers and consumers remain decoupled. Sensors are not required to know who receives their data, and consumers do not need to know the origin of the information. This loose coupling significantly enhances scalability. Furthermore, data transmission occurs asynchronously, meaning that senders and receivers do not need to be connected at the same time (Prada, et al., 2016; Ford, et al., 2022; Eugster, et al., 2003).

An illustration highlights the advantage of a potential use of MQTT: If a machine controls its processes based on multiple external data sources, it potentially needs a separate interface for each source. If the same data is also used by other devices, further interfaces and communication connections are theoretically required. If many devices are networked with each other, this may quickly lead to a highly complex system. Instead, a central data source (server) can be used to bundle all the information. All devices (clients) then access this server in a standardized way, which significantly reduces the setup and maintenance effort and minimizes the number of interfaces required - namely one per client (Plenk, 2024, pp. 115-117).

Messages exchanged via the MQTT protocol, or as the official specification of the MQTT protocol calls it control packets, fall into four primary categories: Publish, subscribe, ping and disconnect. Each MQTT Control Packet can be as large as 256 MB and consists of three primary components (Banks & Gupta, 2015, pp. 16-22):

- Fixed Header (included in all MQTT packets)
- Variable Header (present in some MQTT packets)
- Payload (included in some MQTT packets)

The Fixed Header contains (Banks & Gupta, 2015, pp. 16-20):

- The Control Packet Type (Bits 7-4), defining the function (e.g., CONNECT, PUBLISH, SUBSCRIBE)
- Packet-specific Flags (Bits 3-0)
- A 4-bit unsigned value that identifies the type of message (e.g., client request to connect, publish message, acknowledgment, or subscription request). The remaining bits define packet-specific properties (e.g., Quality of Service in PUBLISH messages)

The Variable Header varies by packet type and typically consists of two bytes. For PUBLISH packets, the Variable Header contains (Banks & Gupta, 2015, pp. 20-21):

- Topic Name, which defines where the message is published
- Packet Identifier (for QoS > 0), uniquely identifying the message. This identifier allows the recipient to confirm delivery using a PUBACK message
- The Payload carries the actual message content. The format and data structure depend on the application

Quality of Service (QoS) Levels determine the behavior of message transmission and the reliability with which a message reaches the recipient the categorization goes as follows (Banks & Gupta, 2015, pp. 52-55):

- QoS 0 ("At most once") - Messages are sent once without confirmation, and loss is possible. Suitable for non-critical sensor data
- QoS 1 ("At least once") - Messages are delivered at least once but may be duplicated
- QoS 2 ("Exactly once") - Ensures messages arrive exactly once, making it ideal for use cases like billing systems

At Quality-of-Service level 1, two transmissions are needed: one for publishing and one for acknowledgment (PUBLISH and PUBACK). At level 2, and in the absence of transmission errors, the process involves four exchanges: PUBLISH, PUBREC, PUBREL, and PUBCOMP. As a result, higher QoS levels lead to more traffic, which is generally justified for critical data (Ford, et al., 2022).

In order to maintain active connections, MQTT implements the keep alive mechanism, which helps detect unresponsive clients. When a client connects to a broker, it specifies a keep alive interval (measured in seconds). To signal that it remains active, the client must send a so called “PINGREQ” packet within this interval. The broker then responds with a “PINGRESP” packet, confirming that the connection is still valid (HiveMQ, no date, p. 43).

The MQTT's architecture and its flexibility and simplicity make it effective for integrating embedded devices (Prada, et al., 2016). Eclipse Paho MQTT (see Eclipse Foundation, 2024, provides open-source MQTT client libraries in multiple Programing languages, including Python (pypi.org, 2024) and other programing languages (Mishra & A., 2020).

2.11.5 Modbus

MODBUS is an application-layer protocol used for data exchange between clients and servers over various infrastructures, including Ethernet (TCP/IP), serial lines (e.g., RS-232, RS-422, RS-485), and even radio links (Modbus Organization, Inc., 2012, p. 2). A master-slave architecture is defined, supporting up to 247 slave devices. Only the master is permitted to initiate communication. The system can also operate in a multi-master setup, provided that just one master is active on the line at any given time (Meroth & Sora, 2023, p. 270). Queries include a destination address, function code, relevant parameters, and an integrity check. Replies from slaves confirm actions or return requested data, along with verification fields. During this exchange, the master sends a service request to the slave, which replies accordingly.

Data of the Modbus protocol is structured around four reference types, each identified by a prefix (Acromag Inc., 2005, pp. 3 &13-14):

- 0xxxx: Coils - read/write digital outputs
- 1xxxx: Discrete Inputs - read-only digital inputs
- 3xxxx: Input Registers - read-only analog values
- 4xxxx: Holding Registers - configurable or output values

2.11.6 Modbus RTU

MODBUS communication over serial lines is typically carried out in RTU (Remote Terminal Unit) mode. In this format, each message includes the target slave's address, a function code, a data segment containing 0 to 252 bytes, and a CRC-16 checksum for error detection. Each slave is assigned a unique 1-byte address ranging from 1 to 247, while address 0 is reserved for broadcast messages. The master does not have an address of its own. Function codes, ranging from 1 to 128, specify the operation the slave is expected to perform. To verify message integrity, a 16-bit CRC is added, supplementing any parity checks performed during transmission. In RTU mode, a data word always consists of exactly 11 bits. When a message is sent by the master, the addressed slave responds by echoing its own address in the reply (Meroth & Sora, 2023, pp. 270-271).

2.11.7 The EIA-485 interface (RS-485)

RS-485, also known under its standard designation EIA-485 employs a differential voltage signaling scheme. Two data lines are operated in antiphase (differential mode), and the receiver evaluates the voltage difference between them. As a result, common-mode disturbances (i.e., identical noise signals affecting both wires) are largely canceled out and do not corrupt the transmitted data. The logical states in RS-485 are assigned based on this voltage difference: a differential voltage $A - B < -0.3$ V is interpreted as logical 1, while $A - B > +0.3$ V corresponds to logical 0. According to ISO 8482, RS-485 supports cable lengths of up to 500 meters. However, with the use of modern, symmetric line drivers and proper termination, transmission distances can be extended to 1.2 kilometers. Data rates of up to 1 Mbit/s are achievable under these conditions. An important requirement, especially for long-distance installations, is galvanic isolation between the RS-485 interface and the rest of the circuit. A notable feature of RS-485 is its support for multi-point communication. It allows multiple devices to be connected to the same bus lines, operating in half-duplex mode, where all devices share a common transmission medium but only one may transmit at a time. This bus-like capability is made possible through line drivers, which ensure that only one transmitter is active at a given time (Tröster, 2011, pp. 504-505).

2.11.8 Modbus TCP/IP

Modbus TCP/IP represents an adaptation of the Modbus RTU protocol, utilizing the TCP interface that operating over Ethernet. Essentially, a traditional Modbus data frame is encapsulated inside a TCP packet without alteration. Data integrity is ensured through the inherent error-checking features provided by Ethernet and TCP/IP link layers. This simplifies message construction while maintaining reliability. Once a TCP connection is formed, it typically remains active throughout communication. This persistent channel allows continuous bidirectional transmission of application data between client and server. Furthermore, the client is capable of issuing multiple Modbus requests over the same connection without waiting for previous responses to complete (Acromag Inc., 2005, pp. 4-5 & 27-28).

2.11.9 Control Area Network

The Controller Area Network (CAN) is considered as a “high-integrity” serial data bus designed for real-time applications, capable of operating at data rates of up to 1 Mbit/s. One of its advantages is its strong error detection and confinement capabilities, making it particularly suitable for safety-critical applications. Furthermore, a CAN system can support up to 112 nodes and the CAN bus itself comprises two signal lines (CANH, CANL) that are terminated with 120-Ω resistors at both ends (Valvano, 2017, p. 438), see also figure 7. Furthermore, a CAN-Transceiver manages voltage levels and interfaces the receive (RxD) and transmits (TxD) signals with the CAN bus. It ensures that the digital signals from the CAN controller are correctly transmitted over the bus. (Valvano, 2017, pp. 438-439). CAN provides a Muli-Master principle, which means multiple nodes may attempt to access the bus simultaneously. Furthermore, the CAN employs a mechanism called as “bus arbitration”, which means that in the CAN-network, two signal states can occur: dominant and recessive, whereas all nodes are connected via a “wired-AND” principle. This means that whenever at least one node sends a dominant signal (0), it prevails over all recessive signals (1). During arbitration, each transmitter compares the transmitted bit with the level being present on the bus. If both match, the transmitter may continue. However, if the transmitter detects a dominant bit even though it has sent a recessive one, it loses the arbitration and must stop the transmission immediately without sending any more bits. This ensures that time-critical messages are transmitted first, while lower-priority messages must wait for an available transmission window. In real-time systems, urgent messages take precedence, while low-priority transmissions may experience delays. When transmitting a dominant bit (logic 0), CAN_H reaches 3.5 V and CAN_L drops to 1.5 V, creating a 2 V difference. In the recessive condition (logic 1), both signals level out at 2.5 V, resulting in zero differential voltage (Valvano, 2017, pp. 439 & 441; Robert Bosch GmbH, 1991, pp. 7 & 40). The figure 8 illustrates the differential voltage levels.

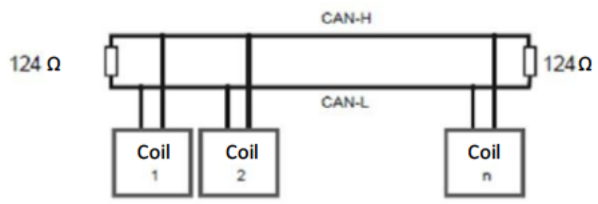


Figure 7: The CAN bus wiring. Modified from (Tröster, 2011, p. 528).

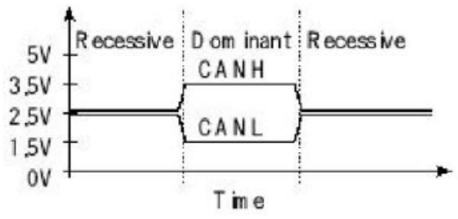


Figure 8: Voltage specifications for the CAN-Interface (Valvano, 2017, p. 439).

The standard format used for data transmission is the data frame, consisting of several sections, including the arbitration field, which determines message priority when multiple nodes contend for bus access. In a CAN system, message- frames are not addressed to specific nodes but instead identified by their content. Each message- frames carries a unique identifier, which defines both its priority and purpose (Valvano, 2017, p. 440; Robert Bosch GmbH, 1991, pp. 11 & 13). A receiver acknowledges a valid message by sending a dominant bit in the acknowledge slot, that confirms successful message reception. In case no dominant bit is detected in the ACK SLOT, an ACKNOWLEDGMENT ERROR is detected, and the message will be retransmitted. The Data Frame ending is marked with the seven recessive-bit section END OF FRAME (EOF). Furthermore, an Interframe Space (IFS) separates one frame from the next. Finally, if no error has occurred before the reception of the END OF FRAME sequence, the bus becomes accessible for other messages again (Valvano, 2017, pp. 440 - 441; Robert Bosch GmbH, 1991, pp. 48, 23, 14, 18 & 54).

2.11.10 Synchronization Methods

Asynchronous interfaces e.g. UART (a common protocol for microcontrollers, see also further information in (Valvano, 2017)) operate using a fixed baud rate for synchronisation to ensure that both the transmitter and receiver function at the same frequency. Transmission must occur at a precisely defined speed-any deviation, for instance due to delays, can lead to errors. In asynchronous communication, the devices rely on independent clock sources, which makes it essential for both ends to use the same protocol and transmission rate. This method is commonly referred to as the start-stop technique: start and stop bits are added to each data unit to enable synchronization on the receiver's side (Tröster, 2011, pp. 20 & 490). Due to slight differences in clock timing, the maximum length of uninterrupted data transmission is limited (Baun, 2012, p. 20). While the bits within a frame (typically one byte) are transmitted in a fixed time pattern-maintaining temporary synchrony between sender and receiver-the individual bytes themselves are not time-aligned, hence the term "asynchronous". This lack of continuous synchronization ultimately limits the performance of such systems (Tröster, 2011, p. 491). When using a program to receive data via an asynchronous interface, the software must process incoming data quickly enough to avoid loss, as UART provides no built-in mechanism to pause or resynchronize transmission. To mitigate this, many microcontrollers and operating systems offer a FIFO (First-In-First-Out) buffer for serial data, allowing temporary storage of incoming bytes until the application retrieves them. However, once the buffer has a limited size (Valvano & Yerraballi, 2022).

In synchronous interfaces however, a shared clock signal determines when data is transmitted and received, thereby synchronizing communication between devices. Both sender and receiver know exactly when to exchange data, as transmission is aligned with this timing signal (Baun, 2012, p. 19). Unlike asynchronous communication, synchronous systems do not rely on a fixed baud rate. Instead, the duration of each clock cycle can vary, allowing for flexible adaptation to system requirements. Although, in theory, the clock signal could remain high or low for extended periods, in practice, an appropriate clock frequency is selected to ensure efficient data transfer-provided both devices can handle the chosen speed, an adequate application example would be the I2C Bus, see also (Bähring, 2010, pp. 287-290). A key advantage of synchronous communication is that it allows for the transmission of much larger amounts of data without the need for repeated resynchronization, as sender and receiver stay aligned over longer periods (Tröster, 2011, p. 491). However, one drawback is that the main program may be affected if it has to wait for data transmission to complete. This limitation can be mitigated by using hardware features such as interrupts or Direct Memory Access (DMA), which help reduce CPU (Central Processing Unit) load and allow non-blocking (cf. Valvano & Yerraballi, 2022).

2.11.11 Technology of I^{2}C

The I^{2}C bus, also referred to as the Inter-IC Bus, was originally created by Philips/Valvo. It operates as a synchronous, clocked communication system using just two lines: one for the serial clock signal (SCL) and one for the data transfer (SDA). Connected nodes communicate via integrated bus controllers equipped with open-drain outputs and pull-up resistors. The bus supports multi-master configurations, meaning any device can take the role of Master or Slave depending on the context. The Master is always responsible for initiating data transmissions and for generating the clock signal on the SCL line, but both Master and Slave nodes are capable of transmitting or receiving data during communication. Each data transaction on the I^{2}C bus begins with a START condition, where the Master signals the beginning of communication and addresses the target Slave device, which is identified via a unique 7-bit address. Communication is terminated with a STOP condition, during which the Master releases the bus. As mentioned in the previous chapter, data rates are not fixed to predefined levels; rather, the I^{2}C standard defines upper speed limits, and actual transmission speeds can vary anywhere between 0 kbit/s and the specified maximum, depending on the devices and system design (Bähring, 2010, pp. 287-289).

3 La Luz: Presentation of the Site

3.1 Overview of the TIGRE Telescope and the La Luz Site

The *Telescopio Internacional de Guanajuato Robótico Espectroscópico* (TIGRE), is a robotic spectroscopy telescope situated at La Luz Observatory, located in central Mexico. This telescope, which has been operational since 2013, is a collaborative project involving the Hamburg Observatory in Germany, as well as the universities of Guanajuato in Mexico and Liège in Belgium. TIGRE is fully automated and performs its observations without the need for human intervention (González-Pérez, et al., 2022, p. 2; Schmitt, et al., 2014).

The observatory itself is positioned approximately 20 kilometers from Guanajuato and about 300 kilometers northwest of Mexico City. It is located on a high plateau at an elevation of 2,435 meters above sea level. The site's geographic position provides excellent conditions for observations during winter. However, summer brings frequent thunderstorms, leading to power failures that can severely impact the system. Electricity supply in this remote area is unreliable, with frequent voltage fluctuations and outages caused by storms and strong winds (Schmitt, et al., 2014; González-Pérez, et al., 2022). The observatory experiences outages several times per month. If an outage lasts longer than five minutes, TIGRE ceases robotic operations and remains idle to prevent damage. In cases where power is lost for several hours, the installed UPS does not restart automatically, requiring human intervention to resume functionality (González-Pérez, et al., 2022). This can be observed in more detail the statistical evaluation regarding system dropouts in (González-Pérez, et al., 2022).

3.2 Current Infrastructure

To address the above-mentioned issues, a combination of solutions has been implemented. The observatory relies on a power backup system that includes a 100-kW diesel generator, which automatically activates within seconds when external power fails. Additionally, a lead-gel battery-based uninterruptible power supply (UPS) ensures voltage stability and provides backup power for at least 30 minutes during normal telescope operations. Furthermore, a three-phase transformer has been installed to regulate voltage levels (Schmitt, et al., 2014). As stated in figure 9, the observatory's daily energy consumption is approximately 20 kWh. Mostly required at night. A UPS (manufacturer Eaton; maximum apparent power: 80 kVA) is connected to both grid electricity and the diesel generator. Positioned between these sources, the UPS continuously monitors voltage and switches power sources as needed. It operates in bypass mode, allowing rapid response to interruptions. Thus, it provides power and acts as a grid-forming unit, switching to island operation when more power is required than can be supplied. However, there are drawbacks: the lead-acid batteries require frequent replacement, their runtime is limited to 10-20 minutes, and the UPS itself has a high-power demand, affecting efficiency.

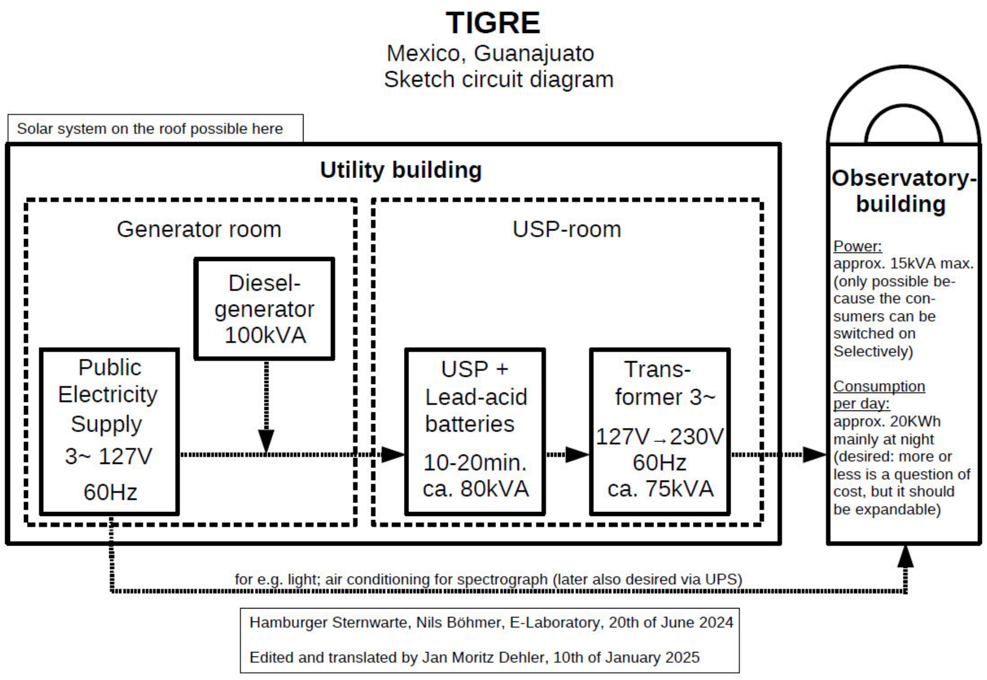

TIGRE
Mexico, Guanajuato
Sketch circuit diagram
Solar system on the roof possible here
Utility building
Generator room
Diesel-generator 100kVA
Public Electricity Supply 3~ 127V 60Hz
USP-room
USP + Lead-acid batteries 10-20min. ca. 80kVA
Trans-former 3~ 127V→230V 60Hz ca. 75kVA
Observatory-building
Power: approx. 15kVA max. (only possible be-cause the con-sumers can be switched on Selectively)
Consumption per day: approx. 20KWh mainly at night (desired: more or less is a question of cost, but it should be expandable)
for e.g. light; air conditioning for spectrograph (later also desired via UPS)
Hamburger Sternwarte, Nils Böhmer, E-Laboratory, 20th of June 2024
Edited and translated by Jan Moritz Dehler, 10th of January 2025

Figure 9: An outline of the circuit diagram of the La Luz site. Modified from (Böhmer, 2024).

3.3 Planned Improvements

Future developments aim to improve sustainability and reliability. Plans include integrating the hydrogen-based energy system into the UPS room, to provide a gradual transition away from the lead-acid batteries. Additionally, an island mode water extraction system is under consideration, which would utilize excess PV energy to collect moisture from the air when battery storage reaches capacity, in case of an islanded operation, to reduce the need of PV-power curtailment.

4 Hardware Architecture of the Microgrid Concerning the Hamburg Setup

4.1 Introduction

This chapter deals with the installed hardware setup at the Hamburg site. The aim for the microgrid's operation is to source as much electricity from locally available solar energy as possible. The lithium-ion battery and hydrogen (H_2) will compensate for any differences between the solar power supply and the power demand of the loads. In the hydrogen subsystem, electrical energy is converted into hydrogen using three AEM electrolyzers. The hydrogen is temporarily stored in a storage system and later converted back into electrical energy via a PEM fuel cell. A simplified overview of the physical setup-including electrical wiring and hydrogen lines is provided in figure 10. Corresponding listings (tables 19–26) of the appendix 13.2 offer a further summary of the system components, regarding the most important technical properties and the current status of the individual hardware components (ready for use, not yet installed, etc.). The following description of the hardware highlights the Hamburg configuration. It should be mentioned here that the planned configuration at the La Luz facility includes a direct connection of the PV modules to an additional DC input on the shared microgrid inverter.

In contrast, this is not feasible in Hamburg, where the local PV system belongs to a nearby Montessori school and feeds into the AC grid via a separate inverter. Other differences between the setup in Mexico and Hamburg will be addressed where relevant. When implemented in the Mexico site, system activation will be planned to be performed on-site, and in the event of a deactivation e.g. due to an error, reactivation is intended to only take place locally. A description of each individual component will follow in this chapter.

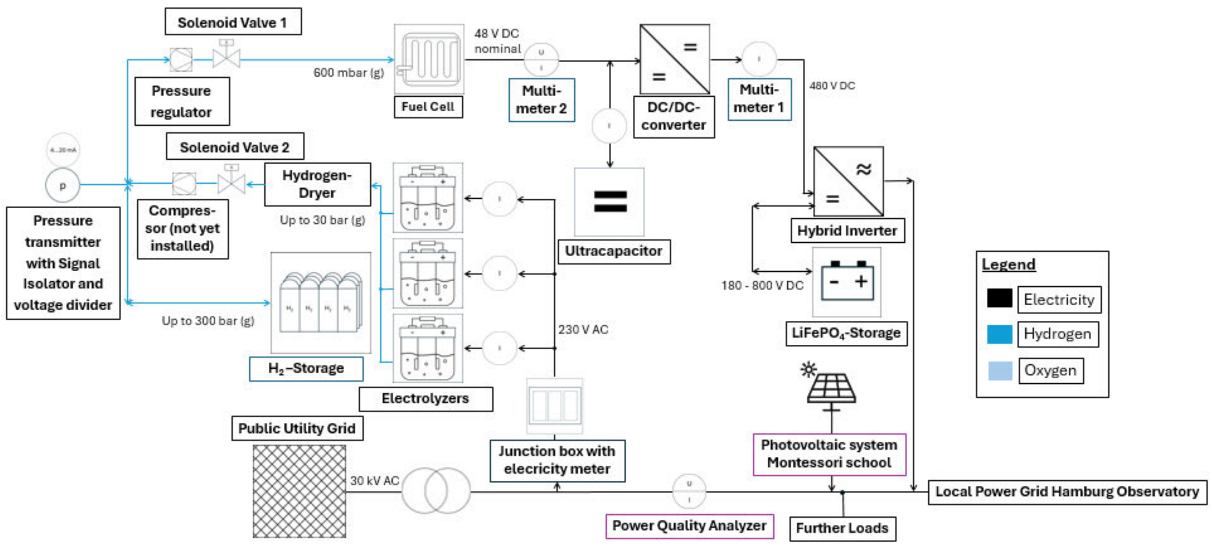


Figure 10: The overview plan of the Microgrid setup in Hamburg. The distances shown in the illustration are not to scale. The communication lines, including placement of the respective controllers, are added in the overview shown in section 6.

4.2 Main components

4.2.1 Fuel Cell

For the selected fuel cell to generate power, both the Enable and Run signals must be active, and fuel pressure must be detected. The Enable signal is set by connecting pins 1 and 2 of the D-sub-DE-9 type connector, while the Run signal is activated by connecting pins 6 and 7, provided that Enable is already present. The FCM-804 is equipped with its own control system, allowing it to function as a controllable load. Furthermore, it communicates via CAN Bus, over the pins 3 and 4 of the connector [A_1]. In the planned setup, the fuel cell will communicate with the Raspberry Pi 1 over Ethernet via a network switch. The Table [A_25] provides an insight on data that can be acquired over the can bus, and its bitwise decryption logic. In [A_26] the decryption regarding the data fields that represent error messages coming from the device are listed.

Behavior during operation

During operation, the fuel cell transitions through different states: it remains off when not powered, switches to inactive when idle, and enters the running state when generating power. If the load drops below 6 Ampere, it automatically switches to standby to optimize fuel consumption. In case of an error, the system enters a fault state, requiring a manual reset before resuming normal operation. The FCM-804 periodically performs a Performance Optimization Cycle (POC), which occurs up to 15 times per hour.

During this process, power output is temporarily reduced or interrupted for a maximum of 12 seconds, requiring an external power source to buffer the load. This buffer is provided by the connected ultracapacitors. This cycle helps maintain optimal performance and prevents system degradation over time. The system also features Maximum Power Point Tracking (MPPT) to optimize power output under varying conditions [A_1]. Further information on the operational characteristics and efficiency calculations based on experiments of a related type of fuel cell (FCM-802) can be found in (Simić, et al., 2021).

4.2.2 Electrolyzers

The device can also be operated and monitored remotely via the Modbus TCP/IP interface. Control signals such as reboot, start, and stop commands, as well as the hydrogen production rate, can be transmitted through the holding registers, also error codes (Enapter AG, 2025a), that can be interpreted according to the logic stated in (Enapter AG, 2025b). Hydrogen obtained from water electrolysis typically contains three main contaminants: nitrogen, oxygen, and water (Ligen, et al., 2020). A requirement of the installed fuel cell regarding the hydrogen purity is that the hydrogen must meet a minimum purity of 99.9% [A_1]. A presence of water in the hydrogen stream can lead to ice formation. Furthermore, water can act as a carrier for water-soluble contaminants such as potassium (K^+) and sodium (Na^+), which, even at trace levels, can degrade the proton conductivity of the fuel cell membrane (Ligen, et al., 2020). Therefore, a dryer is integrated into the system setup in order to remove residual moisture from the produced hydrogen. To take this into account, the electrolyzer system includes a hydrogen dryer (model name: Dryer 2.1) and also a 38-liter water tank (model name: WTM 2.1), that are part of the "Dryer Control Network", a so called "wireless MESH" communication network between the dryer, electrolyzer(s), and water tank that allows monitoring and control via the electrolyzers Modbus TCP/IP interface [A_3, A_4 & A_5]. In the planned setup, the electrolyzers will be connected with the Raspberry Pi 1 over Ethernet via a network switch.

4.2.3 PV-system

Properties and Characteristics

In the Hamburg setup, the PV system feeds into the local AC grid. In contrast, the Mexico setup is planned without a dedicated inverter, connecting instead directly to the DC input of the only hybrid inverter of the microgrid (see section 4.2.4). In Mexico, the model type of the PV modules and the total system capacity will be selected and adjusted according to the specific conditions in La Luz during the planning phase.

4.2.4 Hybrid inverter

Properties and Characteristics

The 15 kW-self-commutated-hybrid inverter is intended to feed into the 400 V/3-phase grid and can be powered by either lithium-ion batteries or the fuel cell. While it primarily accepts 500 V DC from solar input, a DC-DC converter is used to step up the 48 V fuel cell output. The inverter supports multiple modes: Feed-in Priority (uses solar first, charges battery after), Backup Mode (keeps battery charged for outages), Self-Use, Peak Shaving, and Manual Mode (for direct battery control).

Further options include setting SOC limits, enabling grid-charging, and scheduling charge cycles [A_7]. The inverter communicates via CAN bus over a RJ45 connection with the battery management system (BMS) of its connected lithium-ion accumulator [A_8] and is capable of forwarding battery-related information, including inverter-specific data, to the Raspberry Pi 1 using the Modbus RTU over the RS-485 interface.

Current setup vs. original plan

The original hybrid inverter (Solax X3-Hybrid-D) became inoperable due to hardware failure. It was replaced with three 1.6 kW Hoymiles HMS-1600-4T microinverters, which match the fuel cell output, removing the need for DC-DC converters. For consistency, the Solax model remains referenced in this document, although the final inverter type for La Luz is yet to be confirmed.

4.2.5 Lithium-Ion-Accumulator

The battery system incorporates a BMS that provides built-in safety measures, including over-voltage and under-voltage protection, over-current protection, temperature control mechanisms, and short-circuit protection [A_8]. In terms of functionality, the battery serves as an actual storage unit for the local energy grid, working in conjunction with a hybrid and acting as a fast energy buffer.

4.3 Controllers

In general, tasks that require fast data processing are delegated to Arduino boards where applicable, as these devices offer real-time capabilities. A Raspberry Pi however is not a real-time capable platform but has other advantages, since it is considered a computer with an operating system able to easily to be implemented into the system because of its versatile connectivity options. Nevertheless, data acquisition scripts running on the Raspberry Pi are expected to operate with adequate performance for the intended use cases. The Raspberry Pi 1 is planned to act as the grid controller, implementing the EMS algorithm. It refers to the model Raspberry Pi 4. This model was selected due to its relatively modern architecture-being the second newest model in the Raspberry Pi lineup-while also being well-established and widely supported in the community. Its strong market presence ensures plentiful documentation. The Raspberry Pi platform was chosen due to its familiarity among the personnel. In general, tasks that are particularly safety-critical are planned to be primarily handled by the Arduino 1, as Arduinos offer the potential for real-time capabilities. The Raspberry Pi, by contrast, is not a real-time system. However, it is still required that scripts for data acquisition and processing run at an adequate speed (see also section 5). The pin assignments for Arduino 1 and Raspberry Pi 1 are documented in [A_28]. From this point onward, the term "controller" will be used as a simplified reference to both microcontroller boards (such as Arduinos) and single-board computers (such as the Raspberry Pi’s). This simplification is justified by the fact that both types support low-level hardware control and offer the flexibility of user-defined programming.

Table 4: General overview of the grid-controllers to be used in the setup.

Controller Name	Model name	Amount of Components	Technical features and characteristics	Task Component
Raspberry Pi 1	Raspberry Pi 4	1	• Computer with operating system • Offers versatile interfaces • Higher energy consumption than Arduinos	Data acquisition, data management and control of the grid
Raspberry Pi 2	Raspberry Pi 3	1		Data acquisition regarding grid quality
Arduino 1	Arduino DUE	1	• Microcontroller board without operating system • Real-time control capability • Energy efficient	Collecting information content from sensors in the station building
Arduino 2	Arduino MKR (or similar)	1		Collecting sensor data from current sensors located at the power input of the electrolyzers

4.4 Utility Hardware

4.4.1 RS485 CAN HAT

An expansion module for Raspberry Pi, the RS485 CAN HAT (Hardware Attached on Top) is used for the CAN-Communication with the fuel cell and Modbus RTU communication. It provides both RS485 and CAN bus connectivity through the Raspberry's GPIO interface. The RS485 interface, built around the SP3485 transceiver, operates in half-duplex mode, meaning data can be transmitted in both directions but not simultaneously. Communication with the Raspberry Pi 1 is managed via the UART interface, and the RS485 bus typically supports data rates of up to 250 kbps, depending on system requirements. The CAN interface is provided through SPI (Serial Peripheral Interface) communication. The module provides an MCP2515 CAN controller with a SN65HVD230 transceiver and supporting data rates of up to 1 Mbps. Since one singe CAN controller supports multiple nodes on a single bus, the need for multiple SPI connections can be omitted (Waveshare International Limited, no date). For further information regarding the technology of UART and SPI, refer to: (Valvano, 2017; Bähring, 2010).

4.4.2 Analog Sensors Located in the Station Building

Gas Sensors

The gas sensors used are primarily designed for trend monitoring rather than high-precision measurements. The specific sensors operate on a resistance-based principle, where the sensor's resistance decreases as the gas concentration increases and vice versa. The datasheets of the sensors include calibration curves that correlate resistance with gas concentration. However, the sensors are also influenced by temperature and humidity. Meaning no quantitative gas concentration measurement is intended. Further calibration runs are needed to refine the evaluation of invalid values. Each gas sensor features both an analog and a digital output. The analog signal reflects gas concentration trends, while the digital output serves as an alarm function, switching to LOW when a predefined threshold is exceeded. This threshold can be manually adjusted via a small potentiometer on the sensor. Redundancy in this setup applies to the initial installation phase: duplicate sensors are temporarily used to test if they behave similarly [A_9-A_16]. The table 24 in the appendix 13.3 proves an overview and further information regarding the used gas sensors.

Sound Sensors

The Grove - Loudness Sensor is an analog sound sensor designed to measure environmental sound levels. It features a built-in microphone and an LM2904 amplifier, which enhances and filters high-frequency signals, providing an analog output based on sound intensity [A_17]. As it is the case with the gas sensors, it is not regarded as a high precision measurement device and the interpretation of the data coming from it is merely considered as qualitative and a calibration should be provided prior to operation.

Pressure Sensor

According to [A_18] the WIKA IS-3 is stated as "a high-performance pressure transmitter", offering robust measurement capabilities in hazardous environments. Furthermore, it supports gauge pressure ranges from 0.1 bar to 6000 bar and absolute pressure ranges from 0.25 bar to 25 bar, with an accuracy of up to ±0.25% of span. The device operates with a 4-20 mA output signal, a supply voltage of 10-30 V DC, and a response time of ≤2 ms. In order to process the 4.20 mA signal from this sensor, an isolation amplifier is used to convert it to 0-10V before downscaling it to the range of 0-3.3 V via a voltage divider, which provides for an appropriate signal for the Arduino.

For further information regarding the sound- and the pressure sensor, refer to the table 25 in the appendix 13.3.

4.4.3 Digital Sensors Located in the Station Building

Among the digital sensors, the Telaire ChipCap 2 measures temperature and relative humidity and communicates via I^2C with a fixed address of 0x50. Data is retrieved using a standard read command [A_19]. The *ADT7410* also uses I^2C, allows address configuration for multiple sensors on one bus, and offers 13- or 16-bit resolution. Temperature data is stored in two complement format in dedicated registers [A_20]. The *DS18B20* communicates via the One-Wire protocol (for further information in this regard, please refer to the corresponding chapters in (Meroth & Sora, 2023)) and thus supports multiple sensors on a single data line. It allows flexible resolution (9–12 bits) and draws power from the data line if needed. Temperature values are stored as 16-bit two's complement in memory registers [A_21]. It can be used for measuring water temperature due to its design (e.g. In the electrolyzers water tank). More technical details and configuration parameters for all digital sensors can be found in table 26 in the appendix 13.3.

4.4.4 Power Quality Analyzer

To ensure grid quality during operation of the hydrogen plant at the Hamburg site, key electrical parameters such as voltage, current, power, harmonics, frequency, and power factor are continuously measured at the NSHV. This is done using a Hioki Power Analyzer, which records the data every second and stores it on an external hard drive and in a local database via Raspberry Pi 2. The long-term recording is used for comparison with baseline values prior to commissioning of the system in order to analyze their influence. The system has been in operation for over a year without any malfunctions and functions independently of the rest of the control system.

4.4.5 Multimeter 1&2

The HMC8012 (Multimeter 1) is a digital multimeter for measuring DC/AC voltage, current, resistance, capacitance, and frequency, with data accessible via Ethernet [A_23]. The HMC8015 (Multimeter 2) is a power analyzer supporting true RMS values up to 600 V and 20 A, enabling analysis of P, S, Q, power factor (λ), and total harmonic distortion (THD) [A_23]. Both devices are connected via Ethernet to a network switch and interfaced with the Raspberry Pi 1. In the planned setup, current clamps, the HZC50 and HZC51, will be connected to the Multimeters. For the HMC8015, the V input is used for voltage measurement, while the Sensor input is used for current measurement via the current clamp. On the HMC8012, only the Sensor input is used to connect the current clamp for current measurements.

4.4.6 Current Sensor with transformer probe

The SZ 013 sensor enables contactless AC current measurement via electromagnetic induction and is connected to the analog input of Arduino 2. Positioned at the electrolyzers' power input, it serves to verify system activation and fuse integrity by detecting current flow, assuming a corresponding 230 V voltage level. These clamp coils will provide auxiliary data and are not intended for precise analysis. Parameters like power consumption and phase angle will be recorded once during initial startup; detailed monitoring during regular operation will rely on more accurate power analysis tools. The installation is not yet completed (as of 23rd of April 2025).

4.4.7 Hydrogen Storage System

Produced Hydrogen will be stored in a 600-liter storage tank in the form of gas bottles with a filling pressure of 300 bar, at the maximum filling level. A compressor unit will be needed to be installed in order to reach these pressure levels, since the electrolyzes can only provide an output pressure of up to 35 bar. Two magnet valves in the system can open or close the flow of gas-one used for hydrogen inflow, and one used for hydrogen outflow. The valves are designed to remain closed when de-energized. In order to protect the fuel cell from excessive pressure, a Swagelok pressure regulator is installed, ensuring that the pressure never exceeds 600 mBar.

4.4.8 Circuit design: Switching Control via Relay Boards

All switching operations in the microgrid, including control of the fuel cell, power supply to controllers, and warning signals, are handled via relay boards. Relays offer galvanic isolation by design. Since the Raspberry Pi 1 has limited GPIOs, an *MCP23017* I^2C port expander is used to extend I/O capacity. Because GPIOs can't drive relay coils directly, each relay is controlled through a driver circuit using a MOSFET and a 12 V supply, with optocouplers (model name: *H11L1*) ensuring electrical isolation. Two additional safety relays with force-guided contacts manage hydrogen valves and are triggered under fault conditions. All relays operate in active-low mode.

4.5 Concept for the Emergency Stop Chain

An emergency handling concept is in development to ensure system safety during critical events. As illustrated in figure 11, the emergency stop chain is designed to cut power to the hydrogen valves and all 230 V AC-connected devices in the station building, bringing the system into a safe state. Hardware-based protections (e.g., fuses, cut-off switches) take precedence over software. Certain sensors, such as gas detectors or pressure switches, are expected to trigger relays independently of control software. Moreover, a fire alarm system will also be integrated as a safety input. Control units like the Raspberry Pi or Arduino can activate the emergency chain via GPIO. When triggered, all components, including the fuel cell's "run" and "enable" signals, are shut down to prevent off-grid operation. The emergency stop must ensure touch-safe conditions throughout the system, except for the UPS in the La Luz setup, which remains powered to support communication and backup control.

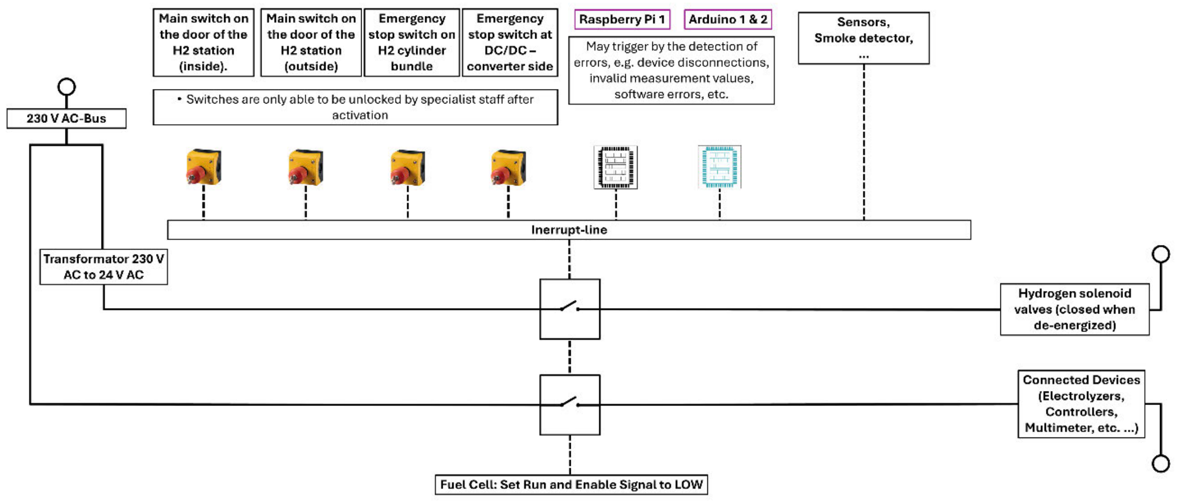

Main switch on the door of the H2 station (inside).
Main switch on the door of the H2 station (outside)
Emergency stop switch on H2 cylinder bundle
Emergency stop switch at DC/DC – converter side
Raspberry Pi 1
Arduino 1 & 2
May trigger by the detection of errors, e.g. device disconnections, invalid measurement values, software errors, etc.
Sensors, Smoke detector, ...
• Switches are only able to be unlocked by specialist staff after activation
230 V AC-Bus
Inerrupt-line
Transformator 230 V AC to 24 V AC
Hydrogen solenoid valves (closed when de-energized)
Connected Devices (Electrolyzers, Controllers, Multimeter, etc. ...)
Fuel Cell: Set Run and Enable Signal to LOW

Figure 11: A functional diagram of the emergency chain.

5 Requirements Analysis for the Developed Control Software

This chapter presents general requirements that were considered before and during the design phase of the developed software structure. It is intended to provide the conceptual framework for the control and monitoring system created within the scope of this thesis. In general, the requirements that are set up in this chapter were considered in the design of the software for the raspberry Pi1 and the Arduino 1. In case, differences have to be made it is being said in the text. The focus lies on key aspects such as communication strategies, system stability, and the flexibility of the software architecture. Furthermore, this section does not claim completeness. At this point, it is worth noting that commercial SCADA and EMS systems could also have been used for this project. Many of such commercial software and hardware-based solutions are currently present in the industry, e.g. (Enapter S.r.l., 2025; OpenEMS Association e.V., 2022; MPI Technologies AG, no date). However, it was deliberately chosen not to rely on commercial solutions for several reasons: Given the diversity of different interfaces within the setup and the general variability of the project itself, providing a system that allows for flexible and seamless integration of all components was the focus.

A self-designed software takes this into account. Moreover, to ensure long-term maintainability and a deep understanding of the system’s behavior, it was strived to retain control over as many aspects of the software and hardware interaction as possible. Furthermore, commercial software solutions are often license-based and associated with significant costs. In contrast, the Raspberry Pi platform supports the Python programming language, which offers a vast ecosystem of open-source libraries. This creates the potential to meet virtually any software requirement on the Raspberry Pi 1 within the project using Python alone cf. (Farrenkopf, 2024, p. 27).

5.1 Communication Compatibility

The system must ensure compatibility with all interfaces present in the setup. A core requirement is that incoming messages must always be processed reliably. No messages should be missed or lost due to software limitations.

5.2 Modular Structure

The software on Raspberry Pi 1 must be modular. This refers to a structure composed of independent programs that can exchange data asynchronously and be modified or updated independently. By developing a structure of individual programs, the system will be extendable, allowing new technologies to be implemented easily.

5.3 Fault Management

This section describes the requirements for ensuring operational fault tolerance across all additional controllers used in the system. It aims to define the expected behavior in case of faults and how the system should react to ensure a safe state.

5.3.1 Definition of Errors

In the context of this system, an error is defined as any condition-detected either by the system itself or reported by a component-that deviates from the expected or intended state of operation and may endanger system functionality, safety, or data integrity. In general, two error categories are recognized in this chapter, see table 5 below.

Table 5: Error categorization and descriptions.

Category	Description
1	Errors outgoing from control units of DER components, send out as a warning or an information message to the requesting client (controller).
2	Errors that have to be detected and marked by the control software itself. Which can be further categorized (see subcategories).
2.1	Errors acquired based on irregularities of acquired data coming from devices without a local control unit, e. g. received measurement values exceeding their allowed range.
2.2	Errors caused by a detected loss of connection during interface usage of any type, indicating either a physical disconnection or a malfunction of the addressed device or program.
2.3	Errors due to high latency during the use of interfaces of any type, indicating a malfunction of the respective addressed device/program or system overloads of the recipient unit (controller).
2.4	Unintentional exceptions due to faulty code in the software of the respective controller (value errors, type errors, syntax errors, …)
2.5	Anomalies of measurement data outgoing from redundant sensory systems in correlation with each other.

5.3.2 Strategies Regarding Error Detection

Errors regarding category 1 should be interpreted following the manufacturer-provided error lists. In general, data acquisition methods should be followed by validation methods as direct as possible. Generally, watchdog timers should be integrated to monitor the arrival of new messages, being configured to "expect" incoming data within a defined time window. In addition, all errors should be printed and logged accordingly, ensuring that printouts are available for historical diagnostics. In any case when errors are detected, it should result in an activation of an error management strategy.

5.3.3 Error Handling

As it is the case with error detection, error handling should be realized as direct as possible. Once an error is identified, response mechanisms should activate immediately, without further propagation of the error. This error management strategy should at least provide the opportunity to activate the emergency chain, via a GPIO Pin. The error management strategy should include the ability to activate an emergency shutdown chain via a GPIO pin. Errors should be categorized by severity. Critical errors (e.g., gas leakage, overheating) must trigger an emergency shutdown. As a baseline, all errors are initially treated as critical, with the possibility to refine strategies as the project develops. For example, in the event of a fuel cell malfunction, the system may attempt a reset before continuing operation. If unsuccessful, the system should run on battery until a critical threshold (e.g., 20 %) is reached, then trigger an emergency shutdown.

Typical Triggers for Emergency Shutdown Should Include:

- Failure of Raspberry Pi 1 (prevents control of fuel cell and electrolyzers)
- Failure of Arduino 1 (sensor data unavailable)
- Gas sensor failure
- Battery failure
- Pressure sensor failure

Examples of Threshold-Based Error Rules:

- If the hydrogen pressure exceeds 30 bar (because at the moment no compressor is installed), the system must shut down immediately. Similarly, if pressure drops below 600 mBar, this may indicate air ingress, posing a risk of hydrogen-oxygen mixing and explosion. In this case, the safety valve must close immediately.
- In case any measurement data exceeds its allowed limits, e. g. hydrogen gas is detected beyond its respective threshold
- Any sensor or device communication goes offline

5.4 Stable System Operation

Applies to the Raspberry Pi 1 software only. In the event of a software error, the system must not crash. Instead, it should restart the affected process and continue operation. Robust exception handling should be implemented throughout.

5.5 Efficient System Operation

Also applies to Raspberry Pi 1. E.g. processes responsible for data acquisition must be designed to be resource-efficient but also fast. The system's resource usage should be monitored and logged, ensuring that sufficient data is available for future scaling or extension of the software.

5.6 Requirements regarding the Design of an Energy Management System

This section outlines requirements to ensure that the EMS operates reliably and efficiently under all expected conditions. The proposed EMS aims to provide a simple yet effective load balancing strategy to optimize energy use throughout the day. In the current microgrid setup, most components do not require constant control. The EMS will primarily manage the timing of electrolyzer and fuel cell activation. Short-term power balancing is handled by the battery inverter. The battery charge level should be maintained between 20% and 80% to preserve lifespan. Hydrogen production is used for long-term load leveling. Electrolyzers act as flexible loads, and the fuel cell serves as a backup during low solar production. Both must not operate simultaneously. The control logic must prevent unnecessary start-stop cycles to reduce wear. Battery usage takes priority, and PV power is either used directly or stored in the battery before activating hydrogen-based systems.

6 Presentation of the Planned Communication Structure

The following descriptions focus exclusively on interfaces that were deliberately selected as part of this project, including the reasoning behind each choice. Interfaces predefined by the connected devices (e.g., Fuel Cell, Electrolyzers) are not discussed in detail here, except to note certain relevant aspects. For instance, Raspberry Pi 1 communicates with the multimeters via Ethernet, as suitable libraries were available exclusively for this interface. CAN bus and Modbus RTU communication were also implemented using Raspberry Pi 1, enabled by the compatible hardware shield (see section 4.4.1) and well-documented Python libraries. The figure 12 shows the overview plan of the Hamburg setup, including the communication lines outgoing from the various interfaces used. Additionally, to the overview of this figure, the so-called master-slave overview tables 14-18 from the appendix 13.2 may also be consulted.

This logic of distributed controllers that can be found in the overview of the setup can be justified by statements from (Valvano, 2017, p. 447), who point out, that such a setup is advantageous, since there might not be sufficient delay margin to permit communication between distant sensors and a CPU. Also, the modularity may enhance easier troubleshooting, additional units may be integrated, and components can be removed accordingly, if requirements are reduced. Lastly, a single device within the network could be dedicated to observing and diagnosing the behavior of the others.

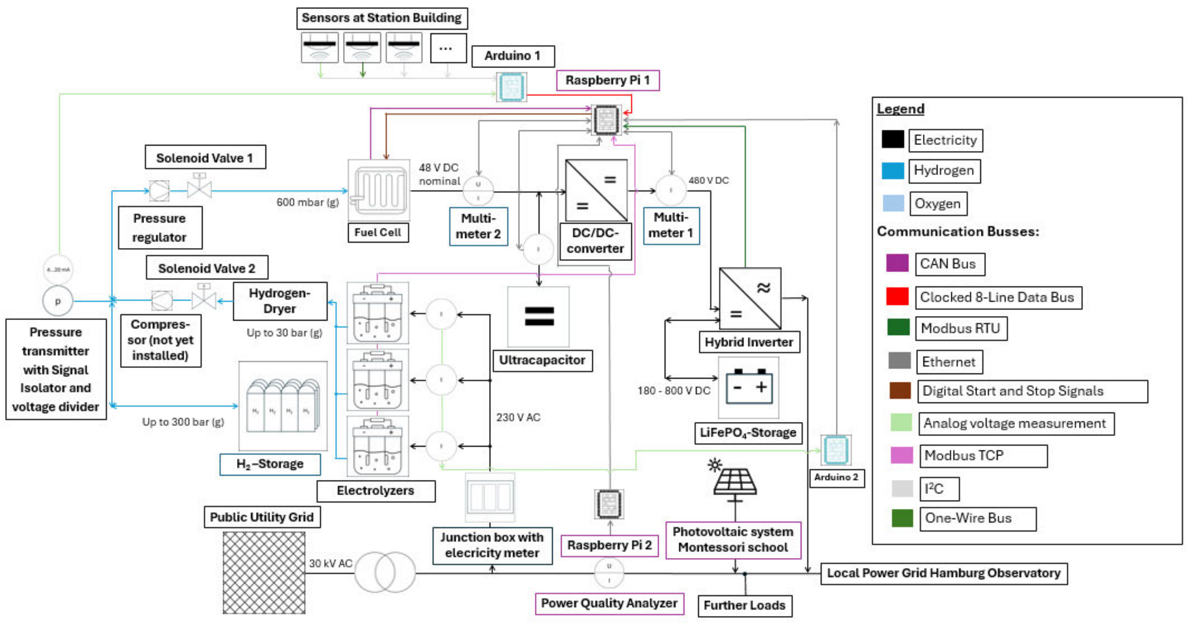


Figure 12: The overview plan of the Microgrid setup in Hamburg (compare to figure 10), including communication infrastructure and implementation of controllers. The distances shown in the illustration are not to scale.

6.1 Interface Arduino 2 - Raspberry Pi 1

An Arduino MKR (or a similar model) will be used to capture the voltage of the coils at the inputs of the electrolyzers and transmit this data to Raspberry Pi 1 via an Ethernet shield, mounted on the Arduino. The timing will not be critical with the Arduino 2, since this information coming from these devices is supposed to act as a validation if the activation signal of the electrolyzes worked. Communication via MQTT would also be possible using the Arduino. Since the coils should have a short connection to the processor, and they are separated from the Arduino 1, this controller is reserved for the safety relevant sensors. The Thesis does not cover the enlightenment of this section, since this connection not yet implemented into the system, nor validated via testing.

6.2 Interface I^{2}C-Sensors - Arduino 1

The digital sensors communicate with the Arduino 1 via I^{2}C, but a custom implementation is used instead of the standard Arduino I^{2}C library (wire.h). This decision was made at the beginning of the project phase, since the current code on the Arduino 1, managing I^{2}C communication, was adopted from previous projects regarding other areas and has proven its reliable functionality over a long period of use. Whereas malfunctions had been observed regarding the use of wire.h.

6.3 Interface Arduino 1 - Raspberry Pi 1 (Clocked-8-Line Data Bus)

The interface between Arduino 1 and Raspberry Pi 1 is implemented as a clocked 8-line data bus, consisting of ten physical parallel connections realized through a flat ribbon cable directly linking the controllers via their respective GPIO pins. From the Raspberry Pi 1, the bus distributes 8 data lines, one clock line, and a separate line named "*Telegram Reset*". The communication through the physical layer is handled purely through the GPIO interfaces on both the Raspberry Pi 1 and Arduino side 1. The design of this custom interface was driven by the need for flexibility in telegram structure and a fast, reliable communication path from Arduino to Raspberry Pi 1.

Moreover, this interface supports synchronous communication, which is in order to avoid e.g. baud rate mismatches and data integrity issues that may occur with asynchronous interfaces when handling long telegrams e.g. with the UART (also refer to section 2.11.10). By having the clock signal generated by the Raspberry Pi, triggering interrupts on the Arduino, a constant handshake mechanism between the controllers is ensured, even under high system loads on the Raspberry, due to the fact the clock signal outgoing from the Raspberry 1 may have HIGH states of flexible time duration. This design choice enables the Raspberry Pi 1 to dictate the pace of the communication regarding this interface, accommodating potential timing fluctuations due to its multitasking software environment. Furthermore, the self-designed protocol design supports a checksum comparison on the receiver side and also supports processing by the Raspberry of the Data send out by the Arduino in under 0.3 seconds per delivered telegram, which encapsulates sensor data- and errors. For more information behind the communication logic, refer to sections 7.2.8 and 13.5.1 of the appendix. The physical cable connection between the two controllers becomes apparent though the pinouts (see [A_28]).

6.4 Application of MQTT as a Superior and System Wide Protocol for Central Data Management and Standardization

According to the stated software requirements concerning the Raspberry Pi 1, it was decided to introduce the MQTT protocol as a superordinate protocol of the microgrid, which should ensure centralized and uniform data availability in the setup and also enabling asynchronous data exchange within the software structure, whereas the Raspberry Pi 1 shall be used as the central broker of the system. According to the chosen communication setup, the overall information of the microgrid outgoing from the various interfaces will ultimately end up being processed by the Raspberry Pi 1. Due to the fact that the developed modular software structure of the Raspberry Pi 1 correlates strongly with the MQTT-communication structure, the explanation and justification of the logical structure regarding this superior communication is outsourced to chapter 8, and especially sections 8.2.2 & 8.2.3 for the sake of simplicity.

7 Presentation of the Developed Software Concept on the Arduino 1

7.1 Introduction

The Arduino program on the Arduino 1 is designed to acquire and process data coming from the multiple connected sensors that are detecting environmental data inside the station-building. The file of the program is containing a Set of subfiles containing the functions running the complete program. In the following, the distribution of tasks among these subfiles are being explained understand the logic of the program. Next to different functions that are responsible for the communication with the sensors, the program contains error detection mechanisms, functions and methods that manage the temporary storage and transmission of the potential errors and the measurement data. The main functionality of the information exchange between the Arduino 1 and the Raspberry Pi 1 is being handled via two Interrupt Service Routines, that are triggered by digitals signals coming from the Arduino 1's Master, the Raspberry Pi 1.

Furthermore, a watchdog mechanism observing the time difference between the incoming of the clock signals coming from the Raspberry Pi 1 is implemented. In order to work with the Arduino 1 code and to follow the explanations in the subsequent sections, a basic understanding of C++ programming as well as familiarity with fundamental concepts such as bitwise operations is assumed. If necessary, further information can be found in appropriate technical literature, such as (Meroth & Sora, 2023). In order to provide a better understanding of this chapter, the reader is encouraged to keep the program code available alongside this document, see [A_31]. An earlier version of the Arduino 1 code had already been developed prior to the start of this thesis. During the course of the work, the communication interface between the Arduino and the Raspberry Pi was redesigned, standardized error-checking mechanisms were implemented, and the sensor communication functions were restructured for consistency.

7.2 General Concept

7.2.1 Initialization of the Program (a_global.ino & b_setup.ino)

The a_global.ino file serves as the global configuration and definition file that includes debug and watchdog settings, as well as configurations for all of the used sensors. Furthermore, it defines pin assignments. For sensor data management, it includes storage buffers and error registers, as well as minimum and maximum thresholds for validity checks. Ultimately, all function prototypes are being initialized in this file. The setup() function initializes the Arduino board and configures various pins and initiates the program code necessary for the operation at the start of each activation of the board.

7.2.2 Main Program Loop (c_loop.ino)

The loop()-function serves as the main execution cycle of the program. It executes the block of code that is being placed inside it periodically. The loop executes the following functions in order, see order for top to bottom in the first column of table 6. The column with the header "Categorization" is introduced to provide an overview of the functionality of the respective function. The more detailed explanation of the functions and the other parts of the program code can be found in the further .ino-file explanation-sections in text form.

Table 6: Overview and categorization of the respective .ino files used for the Arduino 1.

Function Name	Function code located at	Categorization
watchdog()	m_watchdog.ino	Communication with Raspberry Pi 1
gasAlarmStatus()	e_analog_sensors.ino	Sensor data acquisition
gasRead()		
loudnessRead()		
readChipCaps()	g_chipcaps.ino	
Read_ADT7410s()	h_adt_7410.ino	
getDsTemp()	i_one_wire_bus.ino	
copy_Bytearray_for_Pi()	j_copy_Bytearray_for_Pi.ino	Communication with Raspberry Pi 1

The table only shows the functions that are called directly in the loop-function. To enable an overview, the they are being classified as primary functions in this text in order to distinguish them from all the other additional (sub-) functions, or secondary functions that are being called not directly in the main loop. The explanation of the further .ino-files picks up the use of these secondary functions. The explanation of the following parts of the program will initially focus on the primary data acquisition functions. Preceding each section, the secondary functions embedded within the primary data acquisition functions will be introduced first, as they serve function-specific purposes. This approach helps to understand the capabilities of the primary functions. Subsequently, all functions responsible for communication with the master will be covered. As before, the secondary functions will be explained first.

7.2.3 Concept of Data Acquisition Functions

The primary data acquisition functions are responsible for recording measurement data and storing it in a variable. The recorded data is always stored in a 2D array with the following format:

Array_for_measurement_Data[Amount_of_Sensors_of_the_specific_type][3]

This structure of the three time-indices of the second dimension of the array is chosen to support later filtering, as the median filter function requires the current measurement value along with the two subsequent values as input. Each of the primary data acquisition functions increments the time index (e.g., *gasZ*, *ccZ*, *ds18Z* ...) to cycle through multiple stored readings for filtering and error handling. When this index reaches 3, it resets to 0 at the end of the function. The figure 13 shows the functional sequence of any data acquisition functions. The essential function sections are explained in the following. For simplicity reasons, the following description is limited to the general concept of the sensor data acquisition functions. For a more detailed explanation of each data acquisition function, see 13.4 in the appendix.

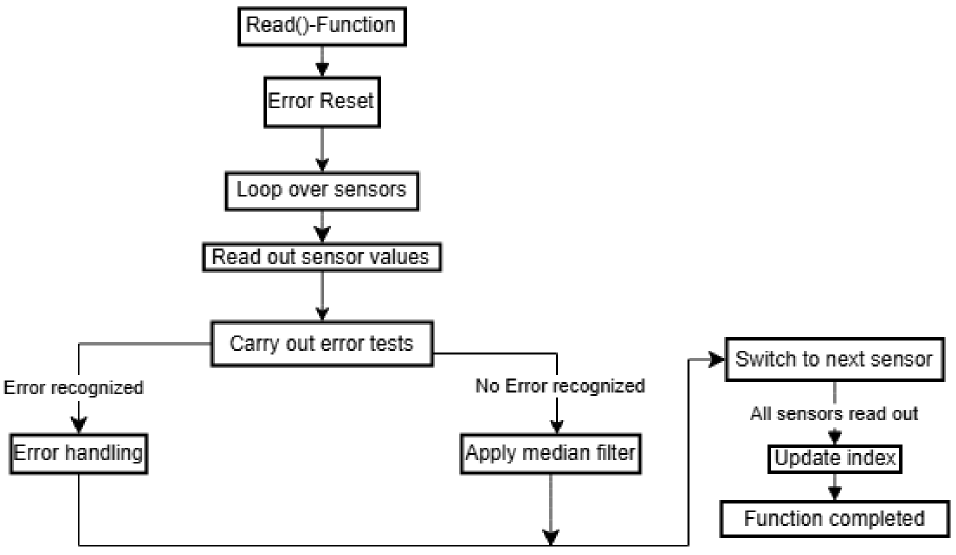


Figure 13: The general procedure of any read()-functions, responsible for data acquisition from sensors.

7.2.4 Error Detection

Error detection occurs immediately after the any measurement-order in the Arduino code. The following errors are detected (exceptions are mentioned), see table 7. In general, the detection of errors result in the activation of the emergency chain (see section 4.5). The resulting error messages are the stored in bytes that are inserted into the telegram structure, which will be sent to the Raspberry Pi upon request (see section 7.8.2).

Table 7: Listing of sensor-related error types, along with further explanation.

Error Type	Explanation
Sensor Unavailability	This applies to cases where a sensor is not connected ("not available") or is connected but does not send a valid signal ("blocked").If a sensor is not connected, the system detects a constant HIGH state due to pull-up resistors (bit-bang logic). If a sensor is blocked, it is connected but fails to transmit valid data.Analog sensor availability cannot be directly determined but is ensured through alarm signaling or redundant analog measurements.
Measurement Value Outside Allowed Limits	Global minimum and maximum values are used to detect whether a measurement is within acceptable limits. Currently, these thresholds are set generously to avoid unnecessary disruptions during test runs. For the gas sensors, certain maximum threshold values have to be set via a screwdriver.
Sudden Spikes in Measurement Values	Currently, this applies only to temperature sensors, not gas sensors, as the gas sensors have not yet been calibrated.

7.2.5 The Error Management Function Prototype

A dedicated Arduino-specific error management function that may fulfill the decision making in case of errors is planned to be implemented for the further use. A function prototype of such kind can be found in the current *at* n_handle_errors.ino in the current software version. This function is called at every point in the code where an error is currently detected. The idea is to introduce error categorization via error codes, allowing predefined actions such as deactivating the emergency shutdown system based on the error type. The error code dictates the decision-making process through a switch case method. It activates the emergency chain in case the error code 0 is transmitted.

7.2.6 Data Filtering

This file includes the two filter functions (*float_filter(float stor[3]*) and *int_filter(uint16_t stor[3])*. They provide the same function-logic but take up different types of input variables (float and integer). The functions identify the median value, or in other words the second biggest value from three stored sensor readings and return their index. This filtering method helps reduce noise by discarding outliers, selecting the most reliable value from the three samples. Occasional fluctuations can cause inaccuracies, especially with analog voltage measurements. By selecting the middle value, the functions enhance the stability and reliability of measurements (cf. Microchip Technology Inc., 2022, pp. 5 -7).

7.2.7 Preparation of Sensor Data into Byte Format

Once a measurement has passed the error detection process, it is stored as a 16-bit integer and then split into a global high byte and low byte using bit manipulation. This method is used, because the respective Raspberry Pi's data acquisition program processes incoming messages byte-wise, making this method compatible with its message interpretation. This approach does not support floating-point values, as transmitting unsigned float values would require 4 bytes, making the communication telegram unnecessarily long.

7.2.8 Concept Behind Primary Functions Responsible for the Communication with the Master

The basic concept behind the Data transmission is being provided by the functionality over the implemented Interrupt service Routines that are initialized to be executed in case a rising signal is detected at the respective Interrupt-Pins (*see setup()*). Whereas a corresponding Data transmission preparation functionality is served by the j_copy_Bytearray_for_Pi.ino file.

Embedding Sensor Data into the Telegram

The *copy_Bytearray_for_Pi()* function formats the acquired sensor data into a structured byte array (*byte_values_to_store[]*) that is being used as a data puffer, containing the acquired measurement data. The *byte_values_to_store* will act as "copy template" for the array *byte_values_to_send[]*, that is used to provide the data the be send to the Raspberry Pi. Since the *copy_Bytearray_for_Pi()* function is placed in the main loop it will thus be periodically executed. The array *byte_values_to_store[]* is a 1D array with a fixed length of 164 elements and represents the structure of the telegram concerning measurement data to be sent to the Raspberry Pi 1. As it is stated in the a_global.ino-file, it is being initialized by starting with a start sign "$" as the element with index 0, the end sign "$" at index 164 and zeroes for all elements in between. Every time, the function *copy_Bytearray_for_Pi()* is being executed, the variables for the *errorCount*s and the *errorRegisters*, the sensor values such as temperature, humidity, and gas concentrations and a checksum are computed and inserted into *byte_values_to_store[]*. The table 8 gives an insight into the structure of the telegram, whereas the total length is always 164 elements, with only the memory locations of the array being continuously transcribed with the current measurement values or current error messages. The number of elements in *byte_values_to_store* that concern error messages (*errorCounts and errorRegisters*) is variable. If each individual sensor of the same sensor type encounters an error, the *errorCount* value for that sensor type would be equal to the number of sensors. Consequently, the number of occupied elements within *byte_values_to_store* reserved for the error registers of that sensor type would also be equal to the number of sensors of that type. The telegram also implements a simple XOR-Checksum, that will be validated on the Raspberry P1 1 side, ensuring secure data transmission.

Table 8: Overview of the telegram structure, including the potential length of bytes of each telegram section.

Variable					
Start marker ('$')	checksum	*errorCountCC*	*errorRegCC[]*	*errorCountADT*	
Max. no. of bytes					
1	1	1	10	1	∑=14
Variable					
errorRegADT[]	*errorCountDS18*	*errorRegDS18[]*	*errorCountLoudness*	*errorRegLoudness[]*	
Max. no. of bytes					
20	1	5	1	2	∑=29
Variable					
errorCountPressure	*errorRegPressure*	*gasAlarms[]*	*ccHygHigh[]*	*ccHygLow[]*	
Max. no. of bytes					
1	1	4	10	10	∑=26
Variable					
ccTempHigh[]	*ccTempLow[]*	*AdtTempHigh[]*	*AdtTempLow[]*	*dsTempHigh[]*	
Max. no. of bytes					
10	10	20	20	5	∑=65
Variable					
dsTempLow[]	*H2ValHigh[]*	*H2ValLow[]*	*petrMethValHigh[]*	*petrMethValLow[]*	
Max. no. of bytes					
5	3	3	2	2	∑=15
Variable					
natValHigh[]	*natValLow[]*	*airQualValHigh[]*	*airQualValLow[]*	*loudnessValHigh[]*	
Max. no. of bytes					
2	2	2	2	2	∑=10
Variable					
loudnessValLow[]	*PressureValLow*	*PressureValHigh*	End marker ('$')		
Max. no. of bytes					
2	1	1	1		∑=5
Total sum of bytes					164

Interrupts Used for Message Delivery

As it is stated in the setup() function, the command:

attachInterrupt(digitalPinToInterrupt(SCL_FOR_PI_PIN, ISR_PI, RISING);

registers an interrupt service routine (ISR_PI) that is automatically executed when a rising edge occurs at the specified pin (*SCL_FOR_PI_PIN*), coming from the Raspberry Ri 1.

This command enhances that the function *ISR_PI(void)* is triggered as an interrupt and calls the function *Serial_Data_to_Pi_write()* to send data in form of the array *byte_values_to_send[]* to the Raspberry Pi 1 (one byte of the telegram per rising signal, see section 13.5.1 of the appendix).

Again, see the following statement in the *setup():*

attachInterrupt(digitalPinToInterrupt(TELEGRAMM_RESET), ISR_PI_II, RISING);

The function void ISR_PI_II(void) manages timing and data transmission. It resets the array *byte_values_to_send[]* by the copying the previously stored byte values from *byte_values_to_store[]* into it. This always ensures that the data being sent is updated before transmission. If the watchdog is enabled (see Boolean variable *enable_watchdog*), it checks for time discrepancies and updates the last signal time (*lastSignalTime_ISR_II*).

The Main Function used for Actual Data Transmission

This code implements serial data transmission from a microcontroller to the Raspberry Pi 1 by sending individual bytes via the GPIO pins according to the information stored in *byte_values_to_send[]*. The function *Serial_Data_to_Pi_bit_write(int sda_case)* sends a single bit (*bitwert*) to one of the eight possible data lines (*SDA_FOR_PI_PIN_X*) based on the specified case number (*sda_case*). *Serial_Data_to_Pi_write()* then sends a byte of data bit by bit, shifting each bit to left after transmission. It then extracts the most significant bit (MSB) from *byte_values_to_send[byte_values_to_send_indx]* and transmits it via *Serial_Data_to_Pi_bit_write(i)*. The byte is then left-shifted (<< 1) to prepare the next bit. After 8 bits are sent, the next byte is selected. If all bytes are sent, the function resets the buffer, reloading data from *byte_values_to_store*.

7.2.9 The Watchdog Function

The *watchdog()* function, located in m_watchdog.ino monitors communication with the Raspberry Pi. If no signal is received in the for a defined period (at the current version it's 10 seconds), a warning message is triggered in debug mode, indicating a potential issue with the communication link.

8 Presentation of the developed software concept on the Raspberry Pi 1

8.1 Introduction

The code was developed using the Geany editor, see (Geany e.V., 2025), and can be executed directly within this environment, other development environments or via the Linux terminal. A solid understanding of Python is assumed for a deeper comprehension of the code and the sections that follow. For background knowledge and further reference, relevant sections in (Monk, 2023) are recommended.

Python scripts related to communication with Arduino 1 were available prior to the start of this thesis, which had to be fundamentally reworked. Although they were based on an earlier version of the Arduino 1 software and required a complete redesign, they served as an initial reference. Additionally, a test script for communication with the hybrid inverter containing random data was already in place and is now being integrated into the main program code.

In order to validate the developed code, several tests were conducted in a laboratory environment. The test setup included the Raspberry Pi 1 connected to the HMC devices via a network switch, the Arduino 1, and an additional Raspberry Pi (model name: Raspberry Pi 5) acting as a simulation node. This "Dummy" Raspberry Pi ran custom scripts designed to emulate other components of the system. It transmitted randomized CAN bus and Modbus data via the RS485 CAN HAT to the Raspberry Pi 1 - simulating the behavior of the fuel cell and the hybrid inverter, see also section 13.8 of the appendix. Additionally, it maintained communication over Modbus TCP/IP (simulating an electrolyzer) and MQTT (simulating Raspberry Pi 2) via Ethernet. In order to provide a better understanding of the following sections of this chapter, the reader is encouraged to keep the program code available alongside this document, see [A_30].

8.2 Description of the software architecture

Each self-designed program within the system is designed to operate independently, meaning it can be executed standalone without dependency-related errors. Due to the Raspberry Pi 4's multi-core processor, multiple programs can run in parallel without interfering with one another. The system follows a modular framework.

8.2.1 Classification of Modules and Program Logic

The following categorization can be made regarding the purpose of the created *_.py-scripts:

- Classes
- "Programs" (indicated by the "*_loop"- sequence at the end of their file name), including:
    - Programs that are not directly included in the MQTT communication network: main.py, blink_loop.py and get_svg_files_loop.py
    - Subscribing programs
    - Publishing programs
- All subscribing and publishing programs can be further classified as:
    - Validated programs that have proven their functionality (under laboratory test runs with all other programs running in parallel) and are expected to be capable in order to fulfill their respective purpose to a satisfactory level also at plant operation.
    - Dummy programs, depicting scripts under construction that are partly validated in their functionality but are restricted for processing only simulated or random data. These programs serve as preparatory work and are implemented for testing purposes.

8.2.2 Internal Communication Structure between the Programs

The system architecture follows a publish-subscribe model, which can be observed in particular through the blue arrows connected to the MQTT broker block, see figure 14. In the current setup, the open-source Mosquitto broker is used, see (Eclipse Foundation AISBL, no date). The software running on the Raspberry Pi 1 thus consists of discrete system components-each implemented as an individual program, whereas that the overall higher-level data management is established through the MQTT-protocol, transacting JSON-formatted ASCII messages over TCP/IP. In the figure, all blue arrows on the picture illustrate the flow of data into and out of the broker and selected subscribing processes are capable of directly controlling physical devices, for example over the EMS.py program, the electrolyzers or the fuel cell can be (de-) activated. In general, publishing programs and selected subscribing services are permitted to trigger the system's emergency shutdown sequence upon detection of critical conditions. For the system to function properly, a broker program must be continuously running in the background.

This loose coupling structure between software modules offers the practical advantage that new devices can be integrated systematically: E.g. such that new programs may first be validated individually regarding their basic functionality and communication interfaces, following a dedicated structure outlined for implementation. Once verified, a new program can seamlessly be integrated into the overall system via MQTT. This modularity also enhances system maintainability, as self-contained programs are easier to debug, update, and manage.

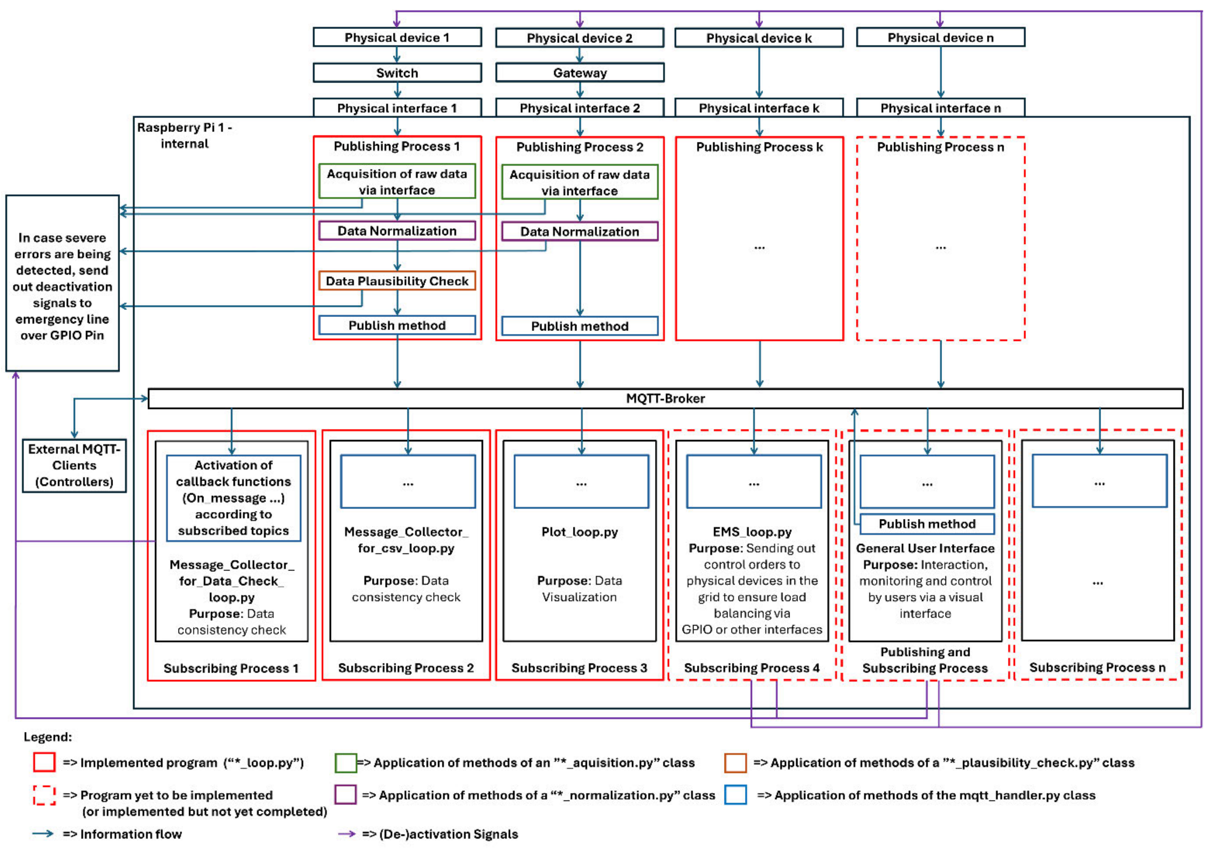

Physical device 1
Physical device 2
Physical device k
Physical device n
Switch
Gateway
Physical interface 1
Physical interface 2
Physical interface k
Physical interface n
Raspberry Pi 1 - internal
Publishing Process 1
Publishing Process 2
Publishing Process k
Publishing Process n
Acquisition of raw data via interface
Data Normalization
Data Plausibility Check
Publish method
...
In case severe errors are being detected, send out deactivation signals to emergency line over GPIO Pin
MQTT-Broker
External MQTT-Clients (Controllers)
Activation of callback functions (On_message ...) according to subscribed topics
Message_Collector_for_Data_Check_loop.py
Purpose: Data consistency check
Message_Collector_for_csv_loop.py
Purpose: Data consistency check
Plot_loop.py
Purpose: Data Visualization
EMS_loop.py
Purpose: Sending out control orders to physical devices in the grid to ensure load balancing via GPIO or other interfaces
General User Interface
Purpose: Interaction, monitoring and control by users via a visual interface
Subscribing Process 1
Subscribing Process 2
Subscribing Process 3
Subscribing Process 4
Publishing and Subscribing Process
Subscribing Process n
Legend:
=> Implemented program ("*_loop.py")
=> Application of methods of an "*_aquisition.py" class
=> Application of methods of a "*_plausibility_check.py" class
=> Program yet to be implemented (or implemented but not yet completed)
=> Application of methods of a "*_normalization.py" class
=> Application of methods of the mqtt_handler.py class
=> Information flow
=> (De-)activation Signals

Figure 14: Overview of the software structure of the Raspberry Pi 1 build around the MQTT broker.

8.2.3 Further Necessary Improvements

The implementation of a graphical user interface (GUI) is not yet supported and only implied in the illustration of figure 14. It may be included in the future in the Raspberry Pi 1's software infrastructure or on a separate computer, since GUI applications may be resource-intensive and could impact the performance of the processes on the Raspberry Pi 1. In general, the outsourcing of MQTT-based programs is possible, via an external TCP/IP connection.

8.2.4 Justification and Discussion for the Use of the MQTT Protocol.

Since the overarching goal was to ensure centralized and standardized data availability across the entire system, several possibilities could have been considered to serve this purpose.

More precisely, alternative overarching "IoT protocols" besides MQTT could have been chosen. Prominent examples next to it include CoAP, AMQP, and HTTP. Each of these protocols brings its own specific advantages and disadvantages. A comprehensive study that explores a comparison of these protocol in greater depth can be found in (Naik, 2017). It comes to the result that compared to CoAP, AMQP, and HTTP, MQTT offers the smallest fixed header size, which makes it ideal for low-bandwidth and also low-power environments. However, since it operates over TCP, it incurs some connection overhead (Naik, 2017). Furthermore, according to (Jaloudi, 2019), HTTP is a synchronous protocol and is thus not compatible with the requirements of chapter 5, which is why it is omitted from here on.

Another Review study about IoT Protocols Smart Grid Communication, see (Tightiz & Yang, 2020) also compares the previously mentioned and addotional IoT protocols reagarding their major (Dis)- Advantages coming and also their complexity. It comes to the result that AMQP, CoAP and MQTT can be condisered as protocols of a low complexity (which is generally viewed as positive with regard to the project), whereas, concerning the disadvantages, AMQP is declared als "not suitable for resource constrained applications" and CoAP's main drawback lies in the limited QoS capability, because as (Thangavel, et al., 2014) explains this is due to the fact that in in contrast to MQTT, CoAP only offers a simplified acknowledgment mechanism that does not subdivide between QoS levels.

Furthermore, a study by Mishra and Kertesz (2020) found that among MQTT, AMQP, and CoAP, MQTT has the highest number of publications, indicating its well-established adoption of a well-researched technology, also considered as a positive trait regarding the project.

While MQTT offers many advantages, certain limitations must be carefully addressed. First, as (Colombo & Ferrari, 2024) states, MQTT does not provide a common data structure such as topic hierarchies and payload formats, which means that interoperability between devices and software components can become challenging. Also, no consistent method for tracking device status is provided by, which is essential for continuous monitoring in such IoT systems.

Also, MQTT lacks security features such as the fact that it does not support mutual authentication between clients and brokers. Also, the protocol also lacks native mechanisms to enforce data encryption and integrity (Şeker, et al., 2023).

Another important consideration is the role of the broker itself. The broker must function reliably, as it represents a critical point of failure. If the broker crashes or becomes unavailable, all message distribution within the system is disrupted. Furthermore, the broker can become a bottleneck when many devices simultaneously publish or subscribe to the same topic (cf. Spohn, 2022).

However, the advantages regarding the use of MQTT outweighed the disadvantages in the decision process and due to the fact that the open-source client library Eclipse Paho, see (pypi.org, 2024), provided for an easy implementation and quick positive results in the testing of the testing phase were responsible for this decision.

8.2.5 Overview of Programs

The following table 9 contains a listing of the programs that the software structure provides. It is intended to support the following explanations.

*Table 9: Complete listing of all *_loop.py programs constituting the Raspberry Pi 1's software structure.*

Name of Program	Categorization	Related Classes Specifically Used for the Respective Program	Purpose
CAN_BUS_01_ loop.py	Validated Publishing Program	Canbus_Data_Acquisition.py	Communication with Fuel Cell
		Canbus_Data_Normalization.py	
HMC8012_loop.py		HMC8012_Data_Acquisition.py	Communication with Multimeter 1
		HMC8012_Data_Normalization.py	
		HMC8012_Plausibility_Check.py	
HMC8015_loop.py		HMC8015_Data_Acquisition.py	Communication with Multimeter 2
		HMC8015_Data_Normalization.py	
		HMC8015_Plausibility_Check.py	
get_resource_usage_loop.py		get_resource_usage_Data_Acquisition.py	Acquisition of System Resource Data
		get_resource_usage_Plausibility_Check.py	
Arduino_ Communication_ loop.py		Arduino_Communication_Data_Acquisition.py	Communication with Arduino 1
		Arduino_Communication_Data_Normalization _and_ Plausibility_Check.py	
		Tools_for_Arduino_Communication_Data_Normalization_ and_Plausibility_Check.py	
		increase_performance.py	
Modbus_RTU_ loop.py	Publishing Program (Dummy)	Modbus_RTU_Data_Acquisition.py	Communication with Hybrid Inverter
		Modbus_RTU_Data_Normalization.py	
modbusTCP_ elektrolyseur_loop.py		modbusTCP_elektrolyseur_Data_ Acquisition.py	Communication with Electrolyzers
SolaxHybrid_loop.py		SolaxHybrid_Data_Acquisition.py	Communication with Hybrid Inverter
Message_ Collector_for_csv_ loop.py	Validated Subscribing Program	Handle_Message_Collector_for_csv.py	Periodic logging of the MQTT server content in a csv file
Message_ Collector_for_ Data_Check_loop.py		Handle_Data_Check.py	Data integrity validation over data coming from MQTT-server
Message_Collector_ for_plot_loop.py		Handle_Plot.py	Visualization of specific data series of MQTT-server in a live-plot
EMS_loop.py	Subscribing Program (Dummy)	EMS_Handler.py	Energy management system
main.py	main.py	screen_handler.py	Initialization of all other programs
		Shutdown_handler.py	
blink_loop.py	blink_ loop.py	-	Visualization of system activity via blinking of the status LED
get_svg_files_loop.py	get_svg_ files_ loop.py	-	Performance profiling through generation of .SVG files

Programs that are not Specifically Mentioned in the Following Parts

From here on, the specific explanation of the programs, is further carried out for the main.py and the subscribing programs, whereas the explanation of the publishing programs will be restricted to the general concept. For further detailed information on the respective publishing programs, refer to sections 13.5.1-13.5.14 of the appendix. Also, the programs blink_loop.py and get_svg_files_loop.py are only worth to be briefly mentioned at this point: blink_loop.py, located in the utils directory, is implemented to merely provide a visual conformation of the functionality of the Raspberry Pi 1 and causes the status LED of the Raspberry Pi to flash periodically. Furthermore, the program get_svg_files_loop.py, located in the utils directory as well, has the purpose is to record .SVG files containing flame graphs that visualize which parts of the code are responsible for performance bottlenecks (cf. Frederickson, 2024). It is regarded merely as a testing tool for performance evaluation, intended for use outside of regular microgrid operations.

8.3 Classes Generally Used as Imports

To provide a better understanding of the further text, generally used classes that were imported by all of the programs are being presented in the following section.

8.3.1 Global Error Management (error_handler.py)

This class error_handler.py acts as the global error handler laying the foundation for a future error management logic. It is intended that this class is included in every python file existing in the total_serial program structure. The future use of this class will be as follows: in case of any recognized error in a program, the *handle_errors()* function of the class will be called whereas the function argument *error_code* provides for the decision-making via a match case method. Due to the fact that at the current time the ongoing operations of program executions were mostly limited for testing purposes, this class only provides an error handling strategy for the *error_code* 0, setting the *EMERGENY_PIN* to LOW, insinuating an activation of the emergency chain. More precisely, a time-based logic is used to detect reoccurrence of the same error, such that only if an error occurs for the second time within a predefined time limit, the respective error management is thought to be executed.

Further Possible Improvements

Depending on the transmitted value of *error_code*, additional error management strategies may be introduced in the future.

8.3.2 Time Conversion for Logging Reasons (convert_to_readable_time.py)

The convert_to_readable_time.py-class, located in the directory utils provides a human-readable timestamp format for logging and debugging. Its primary function, *unix_to_readable_with_ms()*, takes a unix timestamp (generated by Python's time module cf. Python Software Foundation, 2025, used at several points in the software) as a floating-point number, where the integer part represents the number of seconds elapsed since January 1, 1970, and the fractional part contains microseconds. Using Python's datetime module, the method converts the extracted seconds into a datetime object and formats it as a string in the format DD.MM.YYYY HH:MM:SS.NNNNNN, where “NNNNNN” represents the microseconds.

8.3.3 Logging of all Printouts (log_class.py)

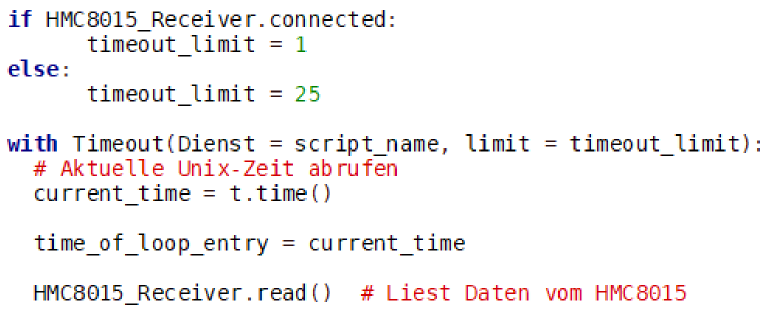
The class, located in utils, provides an automated logging mechanism that redirects all *print()* outputs and error messages (*sys.stderr*) to both a log file and the console. It is designed for process-specific logging by ensuring that each process name gets a dedicated log directory and file. Upon initialization, a base directory for logs is determined, and a subfolder specific to the given process is created. The log files are stored in the log-files directory, located in the parent directory of total_serial. Each process has its own subdirectory named log-files_<process_name>, where the corresponding log files are stored. Log files are restricted to a maximum size of 10 MB while maintaining up to five backup files. If the maximum number of log files is exceeded, the oldest log file is automatically deleted. By overriding *sys.stdout* and *sys.stderr*, all standard print outputs and error messages are automatically logged, eliminating the need for explicit logging statements in the program.

8.3.4 CSV File Management (my_csv.py)

This class provides functionality for storing data in CSV files, implementing file size limits and automatic file creation. The class can be found in the directory utils. Upon initialization, the class defines parameters such as the target file name, column headers, and a predefined storage directory (CSV_for_received_data). This directory is located in the parent directory of total_serial (../CSV_for_received_data). The maximum file size is set to 1 GB, and a flag (*file_full*) tracks whether a new file needs to be created. The *write_csv()* method writes incoming data to a CSV file, storing an array column by column in the current row. If necessary, it automatically creates a new file. It first retrieves the current Unix timestamp and converts it into a readable format using the *time_conversion* class. If no active file exists or the previous file has reached the size limit, a new CSV file is created, with a timestamp and the base file name in the filename ({*file_name*}_{*timestamp*}.csv). The column headers are written in the first row of the new file. If a file reaches the size limit, the *file_full* flag is set, ensuring that subsequent writes go into a new file.

8.3.5 Timeout Rules for Certain Blocks of Code (timeout.py)

The timeout class (timeout.py), also located in utils, is designed to enforce a maximum execution time for certain blocks of code. If an execution of a certain code block exceeds a specified time limit (provided as an argument during initialization), a *TimeoutError* will be raised. The class functions as a context manager, allowing it to be used within a with statement in a program. Upon entering this with-block, the *__enter__()* method (see code in *timeout.py*) sets up an alarm using *signal.alarm(self.limit)*, which triggers a timeout if the specified duration is exceeded. If the timeout occurs, the *timeout_handler()* method raises a *TimeoutError*. The __exit__() method disables the alarm with *signal.alarm(0)* when the block exits normally, ensuring that no unintended timeouts occur afterward. An application example is illustrated in the figure 15.



```
if HMC8015_Receiver.connected:
      timeout_limit = 1
else:
      timeout_limit = 25

with Timeout(Dienst = script_name, limit = timeout_limit):
  # Aktuelle Unix-Zeit abrufen
  current_time = t.time()

  time_of_loop_entry = current_time

  HMC8015_Receiver.read()  # Liest Daten vom HMC8015
```

Figure 15: Example illustration of a code section, showing the application of the Timeout method.

8.3.6 Data Transmission Management via MQTT (mqtt_handler.py)

The *handle_MQTT* class can be thought of as a toolset for enabling communication of any client with the MQTT-broker (mosquitto). In particular, it provides functions for managing MQTT connections, subscribing to topics, publishing data to the broker, monitoring message latencies, and ensuring automatic reconnection if the connection to the broker becomes unavailable. Any program that needs to interact with the MQTT broker must instantiate this class and use its methods. The open-source library paho.mqtt, see (pypi.org, 2024), is used for this class.

Class Initialization and Internal Variables

In order to initialize the class, the argument *client_name*, *client_code* is required (and *debug*, see the table 10 in the subsequent section 8.4.2, which will be ignored for now). It is used to identify which program acts as a client in the process. Moreover, the variable *script_name* will be used for logging and debugging, containing the client's filename (see section 8.4.3). The *client_code* variable determines whether the instance functions as a subscriber or a publisher. In the current version, client codes 0, 1, and 2 represent subscribing programs. More specifically it is intended that the client code 0 should be used for programs subscribing to any topic of the MQTT server, whereas the client codes 1 and 2 are reserved for services that require subscriptions only to specific topics. Any other *client_code* value represents publishing programs, that are thus excluded from the subscriber specific functions of the class. Furthermore, the class defines internal variables to handle message storage and processing. These variables act as intermediate storage variables for different types of data received from various publishers. The MQTT topics are defined in *topics_for_mqtt_names_to_subscribe[]* and each topic is associated with its respective quality of service (QoS) level, specified in *qos_levels_for_subscribing[]*. With the current software version, all QoS values are set to 0, except for the measurement data from Arduino 1, which is assigned a QoS level of 1, since the regulation of the hydrogen consumption/production will rely on the transferred data outgoing from the pressure sensor, connected to the Arduino 1.

Connection Management

Upon instantiation, the class establishes an MQTT connection using the *Connect_to_MQTT()* function. The connection to the broker is established via the command:

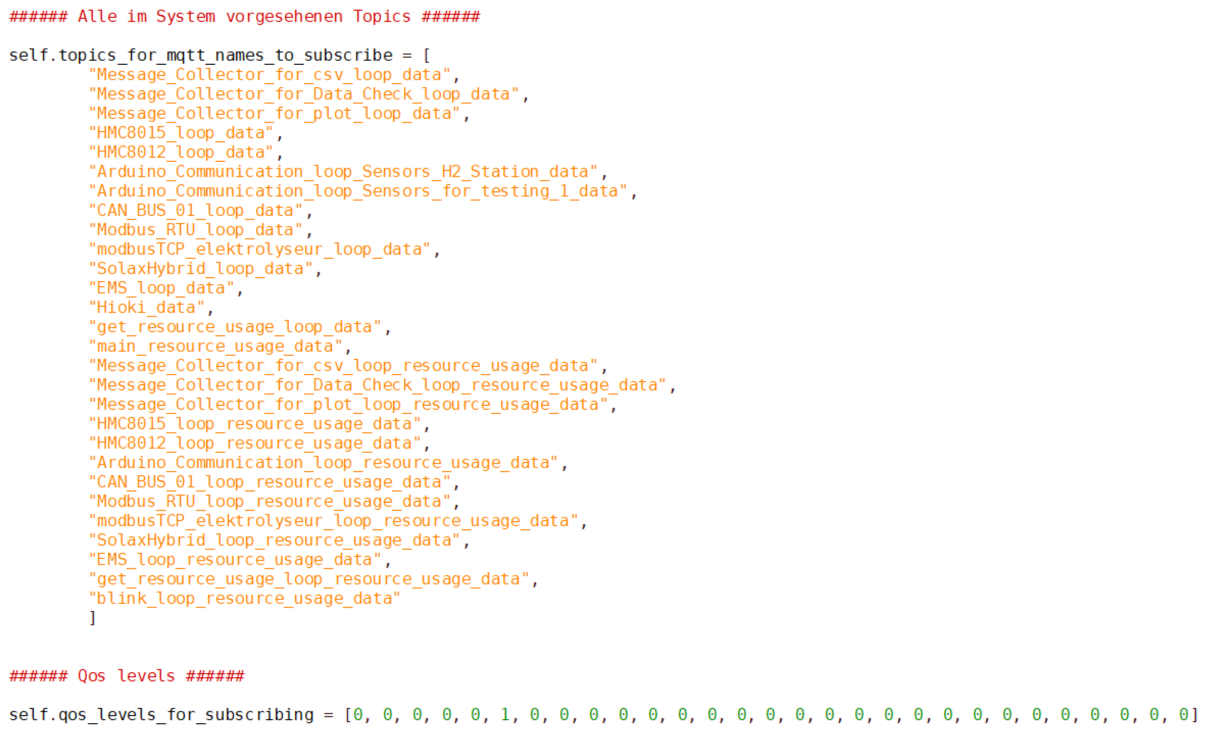
self.client.connect(mqttBroker, 1883, keepalive=1)

where variable *mqttBroker* represents the broker's IP address (currently set to 192.168.0.11), 1883 stands for the standard port for unencrypted MQTT communication, and the *keepalive* parameter ensures that the client sends a message to the broker every second to maintain the connection (see also section 2.11.4). Meaning that for the current setup if the MQTT connection is interrupted, this disconnection will be recognized within the mentioned timeframe and a *on_disconnect* method will be activated, closing the connection and setting the variable *self.connected* to *False*. This flag is constantly checked by any client to detect whether a disconnection has occurred (see section 8.4.3). If, on the other hand, the connection is successfully established upon initialization, the command *self.client.loop_start()* is executed. This starts an internal network thread. Conversely, *self.client.loop_stop()* stops the internal network loop when communication is no longer needed, whereas this command can also be found in the *on_disconnect* method.

Subscription Management

Subscriber clients are the only clients permitted to execute the *manage_subscriptions()* function when a connection is established. This function is executed through an *on_connect()* callback and ensures that the client subscribes to all existing topics while dynamically assigning corresponding callback functions for processing incoming messages. With the current version of the program any subscriber will automatically subscribe to every existing topic. A distinguishing logic could later be implemented to limit the range of subscribed topics for certain subscribing clients, if necessary. That was the main idea behind the implementation of different client codes.

Each subscribed topic corresponds with a dedicated *on_message_** function acting as a callback handler. These functions are automatically triggered on the subscriber side whenever a certain payload is published on the *on_message_** function's corresponding topic to which the client has subscribed to. This topic-specific-*on_message_** function then extracts the payload, decodes the JSON data, and updates the corresponding internal class variables. An optional time-difference calculation mechanism, the *calculate_delta_t()* function, is implemented to measure message latencies and count incoming messages (excluding duplicate messages). This feature is relevant only for subscribing clients, as it is triggered exclusively when an *on_message_* function* is executed. It can be enabled or disabled using the Boolean variable *allow_calculation_of_delta_t*. The mechanism captures timestamps of incoming messages (*software_timestamp*) and calculates time intervals between consecutive messages. Moreover, it maintains records of minimum, maximum, and average message latencies. Periodic resets prevent variable overflow. The figure 16 below shows the code snipped of the *mqtt_handler* that initializes all topics that are implemented including their respective QoS levels. In this regard, it should be noted that all QoS levels are set to 0 except for the topic *Arduino_Communication_loop_Sensors_H2_Station_data*. The QoS level 1 ensures that the sensor data will be Acquired via MQTT at least once in the course of every publishing process because the pressure sensor data is of special relevance for the energy management algorithm, see section 8.5.



```
###### Alle im System vorgesehenen Topics ######

self.topics_for_mqtt_names_to_subscribe = [
        "Message_Collector_for_csv_loop_data",
        "Message_Collector_for_Data_Check_loop_data",
        "Message_Collector_for_plot_loop_data",
        "HMC8015_loop_data",
        "HMC8012_loop_data",
        "Arduino_Communication_loop_Sensors_H2_Station_data",
        "Arduino_Communication_loop_Sensors_for_testing_1_data",
        "CAN_BUS_01_loop_data",
        "Modbus_RTU_loop_data",
        "modbusTCP_elektrolyseur_loop_data",
        "SolaxHybrid_loop_data",
        "EMS_loop_data",
        "Hioki_data",
        "get_resource_usage_loop_data",
        "main_resource_usage_data",
        "Message_Collector_for_csv_loop_resource_usage_data",
        "Message_Collector_for_Data_Check_loop_resource_usage_data",
        "Message_Collector_for_plot_loop_resource_usage_data",
        "HMC8015_loop_resource_usage_data",
        "HMC8012_loop_resource_usage_data",
        "Arduino_Communication_loop_resource_usage_data",
        "CAN_BUS_01_loop_resource_usage_data",
        "Modbus_RTU_loop_resource_usage_data",
        "modbusTCP_elektrolyseur_loop_resource_usage_data",
        "SolaxHybrid_loop_resource_usage_data",
        "EMS_loop_resource_usage_data",
        "get_resource_usage_loop_resource_usage_data",
        "blink_loop_resource_usage_data"
        ]

###### Qos levels ######

self.qos_levels_for_subscribing = [0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

Figure 16: Code section showing the declaration of all current MQTT topics of the setup, including their respective QoS levels, see mqtt_handler.py.

Publishing Messages

Publication of messages is handled by the *publish()*-function of the class. This function exctracts the appropriate QoS level of the message's topic (defined via the array *qos_levels_for_subscribing[]*) and publishes the message with the dedicated paho.mqtt publish command to the broker while also monitoring for potential errors that may occur during publication.

8.4 Programs

8.4.1 General Design Principles Concerning all Programs

These rules and structures account for all programs. Sometimes not for the *main.py* program. If that is the case, this is mentioned in the text.

8.4.2 Common variables

The following table 10 lists and explains common variables that are used consistently across all programs. Their names and functionalities are reused identically.

Table 10: Enumeration of common variable names and functionalities, used throughout the *_loop-programs.

Variable	Type	Purpose
no_errors_in_loop	Boolean	Detects whether the program fell into a secondary *except*-method, and it is set to *False* in such an event. In the program, it is queried to prevent the additional initialization of certain classes at the setup of the program, for example the my_csv-class, will not be initialized more than once in such an event, since this would lead to the creation of a new csv file every 5 seconds in case the python error is not canceled out over time

Variable	Type	Purpose
error_to_print	String	Stores the error message of an exception that will be used later in the code for a printout or a referral via a message that is sent to the MQTT-server
t_delay	Integer	Ensures a minimum execution time of the program
t_delay_for_ publishing	Integer	Sets a time interval for publishing commands to the MQTT-Server
print_interval_ time	Integer	Sets a time interval for periodic printout commands
timeout_limit	Integer	Sets a defined time limit (in seconds) that specifies how long the execution time of a certain block of code is allowed to be
script_name	String	Variable that dynamically stores the local name of any script ensuring clear identification in printouts and logs
create_ internal_csv	Boolean	Acts as a switch to enable or disable the creation of so called "internal" csv-files that log the periodic execution of the respective local file for debugging reasons
time_of_loop_ entry	Float	Stores the current timestamp (*t.time()*) when entering the loop to calculate the duration of the loop. If the processing time is shorter than *t_delay*, the difference is used as the *t.sleep()* duration to ensure a constant loop frequency
software timestamp	String	Logs the current date and time. The class *convert_to_readable_time()* is applied
Error_ observed	Boolean	Detects if any error were observed when running a program. It can be a python exception, an error caused due to a timeout overrun, an error originating from a connected device because of an invalid measurement, etc.
Error	String	Variable responsible to store the observed error
connected	Boolean	Is giving a feedback, whether a connection with a respective device is currently available
debug	Boolean	Acts as a switch, mostly used as an argument at the initialization of classes. Can enable or disable debugging via *print()* statements. Printouts for errors should not be able to be suppressed by this variable
client_code	Integer	Used as an argument for the initialization of the *handle_MQTT*-class. Enables MQTT-methods intended for subscribing programs.

8.4.3 General structure of each developed *_.py program

The general structure of each "_loop"-program consists of a primary *try-except* method that surrounds the complete code sections below it. Its purpose is simply the management of the conduction of a secure shutdown of the respective program in case a termination action (e.g. by pressing ctr + c) is being conducted from a user or an external program (see primary *except* code block).

Below the primary *try-except* method, the primary *while True* loop is stated. Its purpose is to let the underlying code run repeatedly, except the secondary *try-except* method is detecting an unforeseen python-error (e.g. programming errors or runtime errors). In general, two exceptions are defined that deal with the emergence of such errors: An *except* method in case a *Connection Error* is raised in the code and an *except* method for any other error. The reason for the fact that different exceptions are being defined is that, for general python errors the error message is being sent to the MQTT server, the *error_observed* variable will be set to *True* and every other kind of data will be set to *None*. When on the other hand, a connection error is raised, a disconnection from the MQTT server is being detected, because the status of the Boolean variable *mqtt_connect* has changed to *False*. Thus, the error message cannot be sent to the MQTT server, but it can only be printed out (and thus be logged). Because of the disconnection, the program tries to reset the network configuration for the eth0 interface, the IP address will be re-assigned to 192.168.0.11/24 the interface will be activated and the default route will be set via 192.168.0.1 again and the mosqitto broker will be started via a sudo order to attempt to reconnect to the MQTT server. The following finally-method provides the printout (and thus the logging) of the python error. Then a delay of 5 seconds follows before the beginning of the code of the primary while loop is being executed again. The way the current structure of the programs is chosen prevents any occurring python-error from termination of the program. The goal is to prevent an uninterrupted program flow under any circumstances, except the user or any other superior program terminates it. Concerning the regular, error-free execution of the respective program, the setup part of the program is being conducted. It includes the following methods for all programs. The figure 17 illustrates the explained code logic:

```python
if __name__ == "__main__": #Main-Guard method

    try: #primary try method
        #...

        while True: #primary while loop
            #...

            try: #secondary try method
                #...

                while True: #secondary while loop
                    #...

            except ConnectionError as e_x: #secondary except method
                #submit commands to re-establish the network connection


            except Exception as e_y: #secondary except method
                # set connected variable of the device to False
                # send error message to mqtt broker

            finally:
                #print error message with traceback
                #sleep for 5 seconds


    except KeyboardInterrupt: #primary except method
        # Detects manual shutdown (Ctrl+C)
```

*Figure 17: Example illustration of the basic code structure used for any *_loop.py program.*

The following table 11 lists and describes the methods that are foreseen for every primary while loop level in more detail.

*Table 11: Common methods of a primary while True loop, used throughout the different *_loop-programs.*

Method	Purpose
***script_name* extraction**	Extraction of the local script name for debugging and logging purposes
Initialize *global_err_handler*	Initialization of the class responsible for the management of system-wide error handling. Any *global_err_handler.handle_errors(error_code)*-order will result in the execution of the global error management, potentially activating the emergency chain see section 4.5)

Method	Purpose
Initialize *internal_CSV_handler*	If *create_internal_csv* is set to *True*, and the primary while loop is being executed for the first time (*no_errors_in_loop* = *True*), a local CSV file with predefined headers will be created. The class my_csv.py is used
Initialize *print_logger*	Initialization of the logger object to log all printouts, in case of *no_errors_in_loop* = *True*. Usage of the class log_class.py
MQTT client initialization with connection check	Initializes the MQTT client via mqtt_handler.py and checks if a connection exists (by checking the flag *mqtt_client.connected*), if the connection was not successful, a *ConnectionError* is raised
Initializations that are specific to the purpose of the respective publishing/subscribing program	See section 8.4.7 and section 8.4.8
Registering of signal handlers with all variants incl. closing the interface and MQTT connection	Registers signal handlers for *SIGINT, SIGTERM*, etc., to close the MQTT-connection and device connection cleanly, in case any termination signal is being detected
Printout with *script_name* and *readable_time*	Print out the start time of the script and the name of the script in a formatted way

The following table 12 lists and explains the general methods executed within the secondary while loop.

*Table 12: Common methods of a secondary while True loop, used throughout the different *_loop-programs.*

Method	Purpose
MQTT- connection-check	Periodic check, whether the MQTT connection is still active. If not, a *ConnectionError* is raised
Entering *with Timeout* Method	Protects critical code sections with a timeout to prevent hangups. If the timeout exceeds, an exception is raised
Program-specific tasks	See section 8.4.7 and section 8.4.8
Generation of message payload	Creates the JSON format for the MQTT-message
Publication of payload via MQTT	Send messages with measurement data or errors via MQTT
Fill internal CSV with values	If *create_internal_csv* is set to *True*, the recorded data is stored in the internal CSV file (primarily used for debugging and performance evaluation)
Ensuring a Minimum Loop Execution Time	The respective command ensures that each iteration of the loop takes at least *t_delay* seconds by calculating the remaining time and pausing execution (*t.sleep(...)*) if necessary. If the processing time exceeds *t_delay*, no additional delay is applied

Creation of Internal CSV Files

The purpose of the creation of internal CSV files in every program is to enable evaluation of message throughput during post-processing. By storing every *software_timestamp* into an internal CSV file, it becomes possible to analyze communication performance via latencies before MQTT-publication. A subscribing program (*Message_Collector_for_csv_loop.py*) lists the overall data acquired from the MQTT Server in a csv file. Such that Message throughput over MQTT can be analyzed by the comparison of both the internal csv files and the general CSV file from *Message_Collector_for_csv_loop.py*. This feature is only implemented for debugging reasons and can be disabled by setting *create_internal_csv* to *False*.

The Heartbeat-Payload

At this point, it might be counterintuitive that the publication of a payload is considered in the discussion of the general structure of any program, including subscribing programs. The reason is that with each program iteration, also subscribing programs have to publish a heartbeat-payload containing the current *software_timestamp*, the current state of their respective *error_observed* variable (*False* for error-free operation) and a current error message (*None* for error-free operation). This feature is thought to make sure that other subscribing programs can observe their status, checking faultless execution and MQTT-connection and general functionality through the calculated latency through obtaining the *software_timestamp* values (see function *calculate_delta_t()* in mqtt_handler.py). The program Message_Collector_for_Data_Check_loop.py takes this into consideration, see section 8.4.8.

8.4.4 Time Intervals

In the subscribing programs, a print output indicates the program's activity every second. Similarly, the publishing programs generate a print output with e.g. measured values every second. This is controlled by the variable *print_interval_time* (currently set to 1). The same delay logic applies to the execution of commands regarding publishing to the MQTT-server, controlled via *t_delay_for_publishing* (currently set to 1).

Ensuring a Minimum Loop Execution Time

The method “Ensuring a Minimum Loop Execution Time” (see table 12) is applied to all programs, in order to limit execution time of a loop to a fixed time value to save computing capacity. The variable *t_delay* is used for this purpose.

Overview of Timing Parameters

It makes sense to specify how fast certain blocks of code should be executed, mostly for computational capacity reasons. Several if-statements in the program structure that handle the timing of the such cycling times are employed, based on the control variables *t_delay* *t_delay_for_publishing* and *print_interval_time* (also refer to table 10). The configuration of these parameters of the current setup are listed in the tables 13.

Table 13: Overview of the different timing parameters for the respective *_loop programs.

Program Name	*t_delay* in seconds*	*t_delay_for_ publishing* in seconds	*print_interval_ time* in seconds
CAN_BUS_01_loop.py	(-)**	1	1
HMC8012_loop.py	0.1	1	1
HMC8015_loop.py	0.1	1	1
Modbus_01_loop.py	(-)**	1	1
Arduino_Communication_loop.py	0.1	1	1
modbusTCP_elektrolyseur_loop.py	0.1	1	1
get_resource_usage_loop.py	(-)***		

Program Name	*t_delay* in seconds*	*t_delay_for_ publishing* in seconds	*print_interval_ time* in seconds
Message_Collector_for_csv_loop.py	0.1	1	1
Message_Collector_for_Data_Check_loop.py	0.1	1	1
Message_Collector_for_plot_loop.py	1.5****	1	1
EMS_loop.py	0.1	1	1

*This prescribes an error reaction time of ca. *t_delay*.

**No *t_delay* is foreseen here, because these programs are currently set up to be asynchronous. That means it is required that the receiving programs react fast enough to incoming messages. All other programs initiate any data transfer by themselves over inquiries.

***See *get_service_resource_usage()* in get_resource_usage_Data_Acquisition.py: The *interval=0.5* in *cpu_percent = process.cpu_percent(interval=0.5)* provides a CPU utilization measurement of python processes over 0.5 seconds (blocking), which means that the cycle time of the secondary *while True* loop in get_resource_usage_loop.py will be increased for at least 0.5 seconds per detected python process.

**** *t_delay* is limited to 1.5 seconds here due to computational constraints.

8.4.5 Considerations Regarding Error Handling and Debugging

Each type of class, that is periodically used in the secondary while loop of a *_loop-program includes a dedicated method called *handle_errors()*, which is designed to be placed at critical points in the code where errors are expected. This method is responsible for storing the error message in the variable *error*, printing it to the console (thus it will be logged), setting the *error_observed* flag to *True*, and-when applicable-safely closing any active connections with external interfaces. In such cases, the corresponding connected flag is set to *False* to reflect the disconnection status. Furthermore, the *handle_errors()* function of the *global_error_handler* class will be called in order to activate the emergency chain in the case of severe errors (*error_code* 0). The system should always attempt to publish the respective error message to the MQTT broker using the appropriate publish methods. This allows other programs within the system to detect and react to malfunctions. In addition to error messages, informational output can be controlled using the Boolean variable called *debug*. This is intended to allow developers to activate or deactivate print statements as needed for troubleshooting at certain critical parts of the code. To maintain system performance, periodic printouts should be throttled to a maximum frequency of once per second, as higher output rates can lead to performance degradation.

Further Possible Improvements

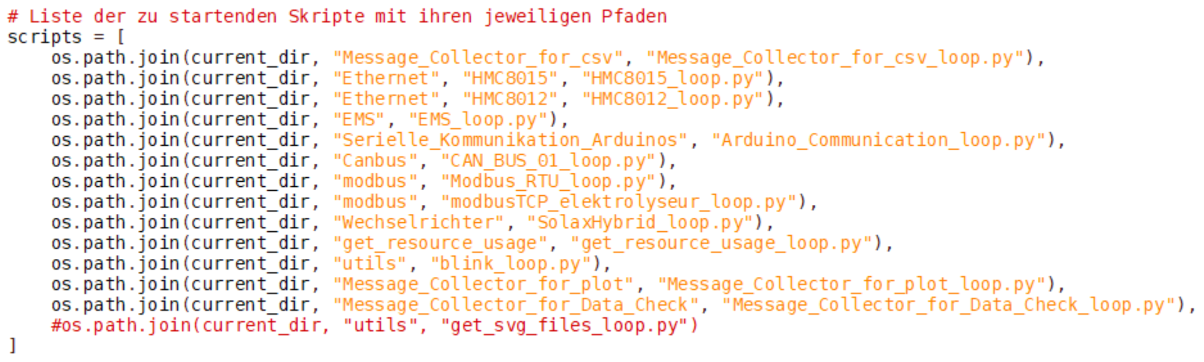
One possible improvement would be the implementation of an exponential or gradual increase in the restart delay to prevent excessive restarting in cases where failures occur frequently. This approach would help mitigate unnecessary system resets and provide more stability.

8.4.6 main.py

The main.py script primarily functions as the initiator of multiple processes required for the system. By executing this script only once, all essential sub-processes are started automatically, and further utilization-functions are enabled. The program utilizes the self-designed classes screen_handler.py and shutdown_handler.py. The objective of the first mentioned class is the execution and organization of multiple scripts in separate terminal windows, such that for each of the programs started, one terminal window is being opened, to provide an enhanced overview. The methods of this class use the number of active programs to calculate the number, size and position of the respective terminal windows. The shutdown_handler.py ensures a controlled shutdown of all currently running python scripts upon receiving a termination signal on main.py, such that if the main.py program is being shut off - all the other processes are shut off simultaneously.

Concept

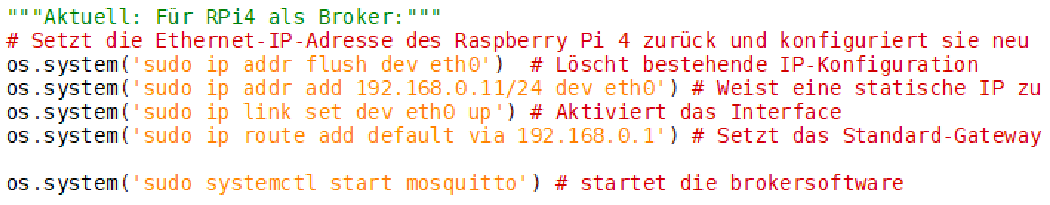
The script maintains a list called "*scripts*", located in the primary while-loop, which lists all Python scripts that need to be executed indirectly. This way scripts can be added or disabled by commenting them out within the array with a "#".. before the respective file path. This is visually implied in figure 18, since the get_svg_files_loop.py program is not foreseen for regular operation. If .SVG files are to be created, the corresponding line must be commented out. In case any new program is planned to be added to the system, it should be added to the list of *scripts*.



```
# Liste der zu startenden Skripte mit ihren jeweiligen Pfaden
scripts = [
    os.path.join(current_dir, "Message_Collector_for_csv", "Message_Collector_for_csv_loop.py"),
    os.path.join(current_dir, "Ethernet", "HMC8015", "HMC8015_loop.py"),
    os.path.join(current_dir, "Ethernet", "HMC8012", "HMC8012_loop.py"),
    os.path.join(current_dir, "EMS", "EMS_loop.py"),
    os.path.join(current_dir, "Serielle_Kommunikation_Arduinos", "Arduino_Communication_loop.py"),
    os.path.join(current_dir, "Canbus", "CAN_BUS_01_loop.py"),
    os.path.join(current_dir, "modbus", "Modbus_RTU_loop.py"),
    os.path.join(current_dir, "modbus", "modbusTCP_elektrolyseur_loop.py"),
    os.path.join(current_dir, "Wechselrichter", "SolaxHybrid_loop.py"),
    os.path.join(current_dir, "get_resource_usage", "get_resource_usage_loop.py"),
    os.path.join(current_dir, "utils", "blink_loop.py"),
    os.path.join(current_dir, "Message_Collector_for_plot", "Message_Collector_for_plot_loop.py"),
    os.path.join(current_dir, "Message_Collector_for_Data_Check", "Message_Collector_for_Data_Check_loop.py"),
    #os.path.join(current_dir, "utils", "get_svg_files_loop.py")
]
```

Figure 18: Image of the scripts-list in main.py, declaration all paths for programs to be started through execution of main.py.

Additionally, the script configures Ethernet settings at runtime to establish MQTT communication. By using OS-level commands, the Raspberry Pi 4's network interface is reset and assigned a static IP address. Moreover, the Mosquitto broker will be activated, see figure 19.



```
"""Aktuell: Für RPi4 als Broker:"""
# Setzt die Ethernet-IP-Adresse des Raspberry Pi 4 zurück und konfiguriert sie neu
os.system('sudo ip addr flush dev eth0')  # Löscht bestehende IP-Konfiguration
os.system('sudo ip addr add 192.168.0.11/24 dev eth0') # Weist eine statische IP zu
os.system('sudo ip link set dev eth0 up') # Aktiviert das Interface
os.system('sudo ip route add default via 192.168.0.1') # Setzt das Standard-Gateway

os.system('sudo systemctl start mosquitto') # startet die brokersoftware
```

Figure 19: Illustration of the routine responsible to resets and reconfigure the Raspberry Pi 1's Ethernet interface with a static IP address, setting the default gateway, and starting the Mosquitto broker service.

With the initialization of the screen handler and the execution of the method *screen_manager.open_scripts_and_seperate_screens()*, the programs, according to *scripts*, are started, upon the opening of their designated terminal. The secondary *while True* loop is restricted to a periodic printout, conducted at each second, concerning the confirmation that main.py is still active.

8.4.7 Publishing Programs

As mentioned before, in this chapter, a detailed explanation of the exact functioning of the individual programs is omitted. By combining the previously provided overview of the general program structure with the following explanation of the main differences between the publishing programs, a sufficient understanding of their functionality can be achieved.

Classes only used by publishing Programs

The publishing programs are responsible for acquiring, processing (or normalizing) and verifying measurement data from connected devices, before transmitting it to the MQTT-broker (publishing). The last-mentioned objective is always being carried out by the publish-function of the *mqtt_handler* (see above mentioned mqtt_handler class). The other three tasks are managed by three publishing-program-specific classes, which are being introduced in the following. Other than the classes that have been introduced before, the different Data_aquisition-, Data_normalization- and Data_check-classes are not to be re-used 1 to 1 by the different respective programs, since 1st, they need different libraries to communicate with their respective devices and by that, 2nd, the format of the acquired raw-data differs from device to device. Also, the check for plausibility of the data by the Data_check class is only implemented, when regarding the communication with devices without a local control unit, that detects device specific errors by itself. That is not the case with the measurement of current and voltage with the current clamps or other devices without any device-specific local control unit.

The *_Data_aquisition class

The *_Data_Acquisition class has the objective of connecting to the measurement device, reading raw data from the measurement query and handling connection issues. It always provides a *connect()* method that establishes communication with the device, detects connection errors, and ensures that the device remains operational. A Boolean variable, *connected*, reflects the connection status, while error and *error_observed* indicates if an issue was detected. The *read()* method retrieves raw measurement data and applies error detection mechanisms while reading measurement data. Any anomalies trigger the *handle_errors()* function, which updates the error- and the connection status and the device-specific error is being stored in the variable "*error*". In case the connection status is being set to *False* from the previous execution, a reconnection with the device with the connect()-function will be attempted at the beginning of every execution of the *read()* method. The figure 20 below illustrates this program flow.

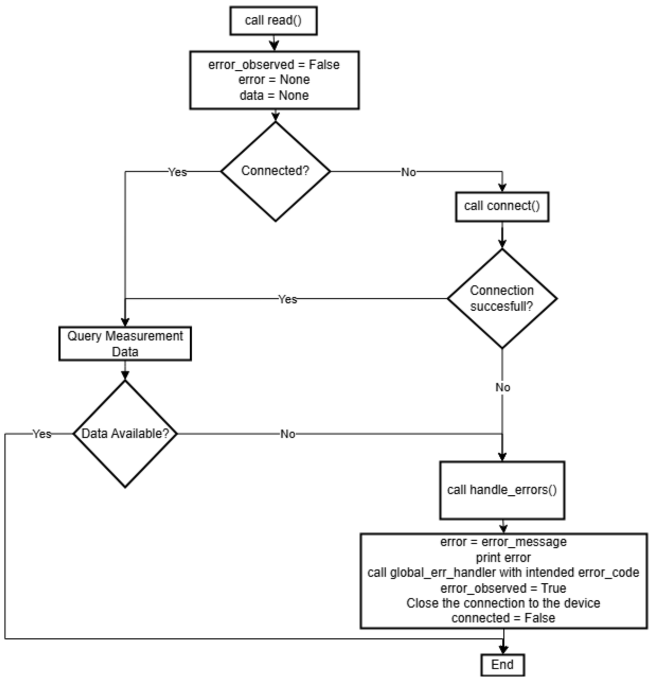


*Figure 20: Flow chart demonstrating the general procedure of any read() function in a *_Data_aquisition class.*

The *_Data_normalization class

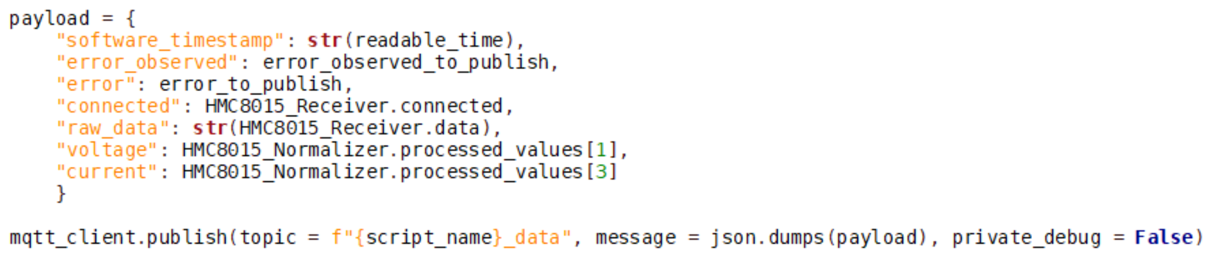
Classes of this type are utilizing the acquired raw data-string coming from a respective Data_aquisition class, filtering it and returning the data in a format that is suitable for further use by the publishing programs. The structure of the normalization class can vary greatly between the individual publishing programs. This is due to the different decoding and reformatting steps required to interpret the raw string data received from various interfaces. For example, in the case of CAN bus communication, the message identifier must first be extracted from the raw string before the data can be interpreted accordingly. More precisely, the message formats used by components such as the fuel cell, electrolyzers, and hybrid inverter may include error codes that must be interpreted based on the respective manufacturer's documentation. This is already demonstrated in Canbus_Data_Normalization.py, where the decoding logic is implemented using an Excel file provided by the manufacturer, which defines how to interpret specific CAN bus IDs, particularly those related to error messages. In contrast, the raw data (or telegrams) received from Arduino 1 follow a different format and must be decrypted using custom logic developed specifically for this interface. This logic is implemented in Arduino_Communication_Data_Normalization_and_Plausibility_Check.py and Tools_for_Arduino_Communication_Data_Normalization_and_Plausibility_Check.py. And regarding the Data_normalization class concerning the Multimeter 1 and 2, it can be merely considered as a set of methods that manipulate a single string (the raw data) to convert it into numerical values.

The *_Data_check class (optional)

This type of class serves in order to flag invalid measurement values, for example values that exceed a maximum value or fall below a minimum value. The *handle_error()*-function of this class is called, in this case. And further plausibility checks can be added here. The *_Data_check class type is marked as optional here because, in the current program version, they are used for programs that interact with components lacking a local controller. Devices equipped with a local controller, such as fuel cells and electrolyzers, will send error messages in case of faulty operation, making the Data_check class redundant for them.

Publishing Data over MQTT Scheme

One of the final steps of each iteration involves publishing the processed data to an MQTT topic. This transmission includes metadata such as timestamps, connectivity status, raw readings, and potential error flags. The publish-order is always stated at the end of each secondary *while True* -loop, that sends the data as a JSON payload via MQTT, see figure 21.



```
payload = {
    "software_timestamp": str(readable_time),
    "error_observed": error_observed_to_publish,
    "error": error_to_publish,
    "connected": HMC8015_Receiver.connected,
    "raw_data": str(HMC8015_Receiver.data),
    "voltage": HMC8015_Normalizer.processed_values[1],
    "current": HMC8015_Normalizer.processed_values[3]
    }

mqtt_client.publish(topic = f"{script_name}_data", message = json.dumps(payload), private_debug = False)
```

Figure 21: An exemplary structure of an MQTT-payload including the respective publish order.

Further Possible Improvements

In general, the expansion of the data validation logic in *_DataCheck to cover additional error-cases would increase reliability, such as the detection of stationary values, unrealistic jumps in the measured values or other anomalies.

Further Necessary Improvements

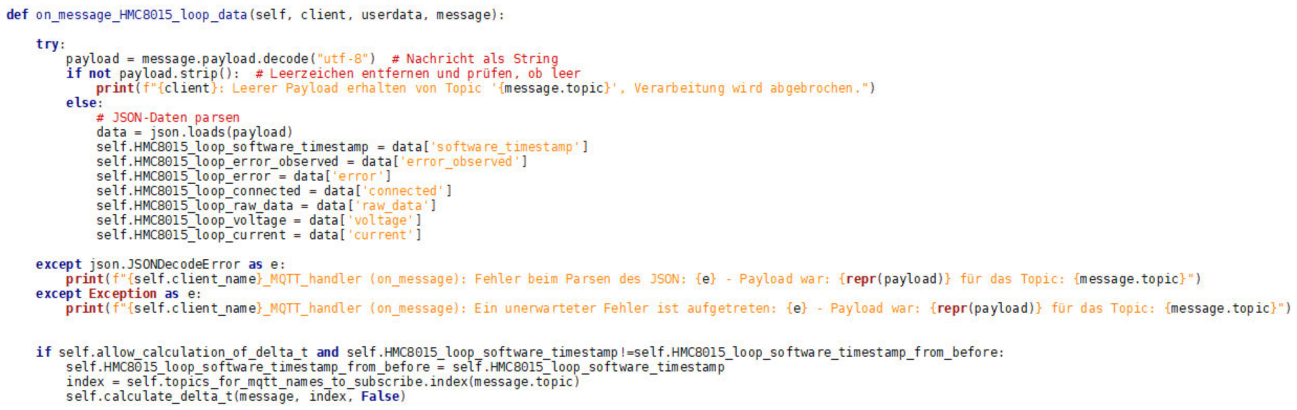
Currently, the *_DataCheck class provides minimum and maximum values that are overly broad and unrealistic. This is due to the fact that the software has not yet been fine-tuned in this regard. At this stage, enforcing strict min/max checks would hinder test applications, as they would constantly trigger errors during test runs.

8.4.8 Subscribing Programs

This section provides a more detailed explanation of the subscribing programs, as their main objectives diverge more strongly from another than it is the case with the publishing programs.

Concept

Subscribing programs retrieve data from the MQTT broker and perform specific tasks based on this data. Currently, a total of four subscribing programs are implemented in the system, see the following sections. As described in section 8.3.6, subscribing programs register callback functions that are automatically triggered whenever a message is published to a topic they are subscribed to. These callbacks are implemented as **_on_message* methods within the *mqtt_handler*-class. Each method is responsible for decoding the incoming message payload, extracting the relevant data fields, and storing them in dedicated class variables. These variables can then be accessed by the subscribing programs for further processing, see figure 22.



```python
def on_message_HMC8015_loop_data(self, client, userdata, message):

    try:
        payload = message.payload.decode("utf-8")  # Nachricht als String
        if not payload.strip():  # Leerzeichen entfernen und prüfen, ob leer
            print(f"{client}: Leerer Payload erhalten von Topic '{message.topic}', Verarbeitung wird abgebrochen.")
        else:
            # JSON-Daten parsen
            data = json.loads(payload)
            self.HMC8015_loop_software_timestamp = data['software_timestamp']
            self.HMC8015_loop_error_observed = data['error_observed']
            self.HMC8015_loop_error = data['error']
            self.HMC8015_loop_connected = data['connected']
            self.HMC8015_loop_raw_data = data['raw_data']
            self.HMC8015_loop_voltage = data['voltage']
            self.HMC8015_loop_current = data['current']

    except json.JSONDecodeError as e:
        print(f"{self.client_name}_MQTT_handler (on_message): Fehler beim Parsen des JSON: {e} - Payload war: {repr(payload)} für das Topic: {message.topic}")
    except Exception as e:
        print(f"{self.client_name}_MQTT_handler (on_message): Ein unerwarteter Fehler ist aufgetreten: {e} - Payload war: {repr(payload)} für das Topic: {message.topic}")


    if self.allow_calculation_of_delta_t and self.HMC8015_loop_software_timestamp!=self.HMC8015_loop_software_timestamp_from_before:
        self.HMC8015_loop_software_timestamp_from_before = self.HMC8015_loop_software_timestamp
        index = self.topics_for_mqtt_names_to_subscribe.index(message.topic)
        self.calculate_delta_t(message, index, False)
```

Figure 22: An exemplary representation of an on_message_*-callback function, see mqtt_hander.py.

Classes only used by subscribing programs

A "*_handler*" class is always implemented within the subscribing programs in order to simplify the combination of MQTT-broker interactions with the execution of tasks that are ought to be performed by the respective subscribing program. Due to this circumstance, the MQTT-handler object will always be instantiated within any of such handler classes.

Message_Collector_for_csv_loop.py

The *Message_Collector_for_csv_handler* class serves as the primary centralized storage program for MQTT data, ensuring the persistence of collected data in a single, comprehensive CSV file. It integrates with the handle_MQTT class via client code 0 to subscribe to all MQTT topics. The task of providing long-term data storage is particularly important, as the MQTT broker data acts only as an intermediate storage, discarding all previous messages under a specific topic upon receiving new ones. The *Message_Collector_for_csv_handler* class first defines a set of attributes from the *handle_MQTT*-class that should be excluded from the CSV output of Message_Collector_for_csv_loop.py, since these specific variables mostly represent fixed control variables. Furthermore, *Message_Collector_for_csv_handler* establishes an MQTT client using the *handle_MQTT* class upon initialization. If the argument *no_errors_in_loop* is set to *True*, the class dynamically generates a *csv_handler* object, ensuring that the CSV file is created with predefined headers. Additionally, the class appends the latency values (*delta_t, delta_t_avg*, etc.) computed by the MQTT handler for each subscribed topic. A crucial mechanism for the maintenance of a consistent storage process is the implementation of the *csv_handler_instance_for_MQTT* variable, that retains the *csv_handler* object of the Message_Collector_for_csv_manager class. This logic ensures that the CSV logging process remains functional without interruptions and creations of multiple files, even if an error occurs. The stored *csv_handler_instance_for_MQTT* object is periodically used in the *write_csv()* call of *Message_Collector_for_csv_manager* in the secondary *while True* loop.

Further Possible Improvements

A mechanism could be implemented that collects the data from the MQTT-server into a dataframe (storage format) of predefined length. Once the dataframe reaches its limit, a buffer CSV file could be created to store its contents whereas a second program would then process this buffer CSV file by loading the data into the database and subsequently deleting this specific buffer file.

This approach would address the current issue of the code that no checks are performed to prevent the Raspberry Pi's storage from running out of space. This logic is derived from the interaction logic implemented on the Raspberry 2 which provides such a logic to store recorded csv files of recorded data into a database (see section 4.4.4).

Message_Collector_for_Data_Check_loop.py

The *data_check_handler* class is designed to monitor and analyze MQTT data integrity by checking connection statuses (via the variable *connected*) and error occurrences (via the variable *error_obsered*) across all MQTT topics. This class plays an important role in measuring connection reliability and tracking error frequencies across all programs simultaneously. It integrates with the handle_MQTT class via using client code 1, allowing it to subscribe to all MQTT topics. This program is thought to act as an additional supervision program, that may activate the error management in case data regarding a respective does not reach this client, indicating a malfunction.

Features

The *data_check()* of the *Handle_Data_Check* class periodically monitors MQTT data. For instance, its main purpose is to observe the values regarding latency (*delta_t_avg*, *delta_t_avg_over_range*, *delta_t_max*, and *delta_t_min*). Currently, in case *delta_t_avg_over_range* reaches a value over 10 seconds, a malfunction in a program must be assumed and the *handle_errors()* method is intended to be called. The following diagrams (see figure 23 and 24) are visualizing the acquired latencies, that are to be queried by Message_Collector_for_csv_loop.py. The diagrams are created in the course of data analysis from recorded csv data produced from Message_Collector_for_csv_loop.py. It can be observed that the latencies spike to values out of the diagrams range. This is due to the fact that unfortunately there were problems the communication between the controllers (checksum mismatch, see .log files stored in the current total_serial folder, see [A_30]), but to the continuous recovery of Arduino_Communication_loop.py, the average latencies remained within acceptable limits. Furthermore, the general behavior of the data coincides with the respective values of *t_delay_for_publishing* (see section 8.4.4).

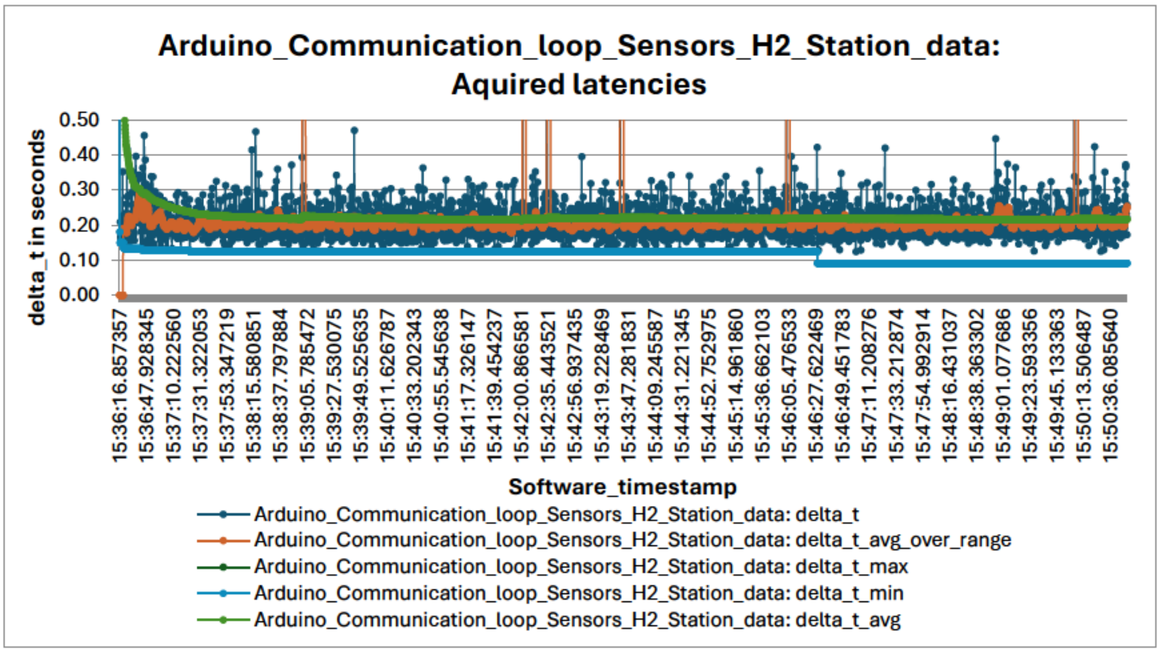


Figure 23: Example of a latency (delta_t) recording of a publishing program.

The figure 24 shows the recorded course of latencies of a subscribing program (Message_Collector_for_csv_loop.py). It can be observed that the latencies are remaining in an acceptable range as well.

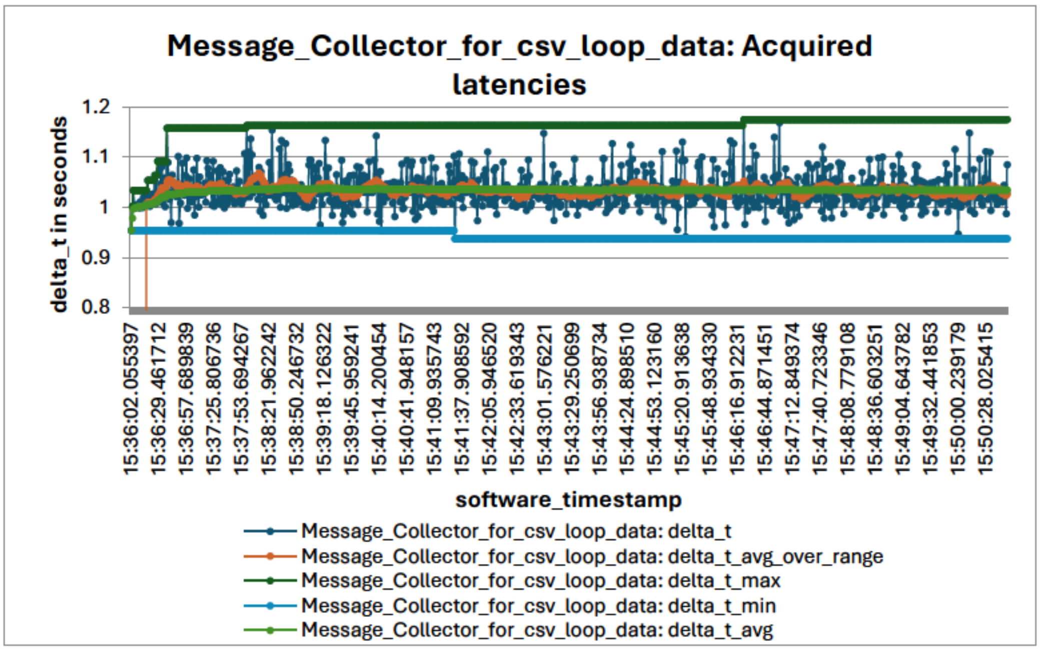


Figure 24: Example of a latency (delta_t) recording of a subscribing program.

The *data_check()* function also has the feature to evaluate connection reliability over a 100-cycle period using the get_connected_percentage() method. It relies on a two-dimensional array *connected_percentage[100][n]*, where *n* represents the number of attributes in the *handle_MQTT* class containing "*_connected*".

A moving average tracks connection states, such that if all devices are consistently connected, the percentage remains at 100%. Similarly, *get_error_percentage()* uses the array *error_percentage[100][n]* to track the frequency of observed errors, based on attributes containing "*_error_observed*". If no errors occur, the percentage stays at zero. Note that *get_connected_percentage()* only evaluates physical device connections, not MQTT client-to-broker connections. Both functions serve as supplementary tools for detecting issues during testing runs.

Further Possible Improvements

If implemented correctly, this could ensure system wide detection of malfunctions for every program. Furthermore, the *data_check()* function could implement a comparison of variables related to different topics that are interconnected in the hardware setup. E. g. a check could be implemented that compares the output current provided by the CANbus of the fuel cell and the measured current of the Multimeter 2. One problem still remains that in case of a program crash or Message_Collector_for_Data_Check_loop.py itself, no malfunction detection can be provided anymore. In order to introduce an increased safety level, an additional program could be designed which iterates through the running python scrips on the system and compares the amount of running programs with the expected number of running programs. This script could maintain a permanent communication with an Arduino. In case the number of running python programs deviates from the expected number or the connected Arduino does not detect any incoming signal for the Raspberry Pi, the emergency chain would be activated.

Message_Collector_for_plot_loop.py

This program introduces a data visualization tool intended to serve as a feedback programs for test runs when interaction with physical devices. The *Handle_Graph* class is responsible for acquiring the current data from the MQTT-server. It establishes an MQTT client using *handle_MQTT* with client code 2. The direct illustration of MQTT data makes it possible to provide a visualization tool without requiring access to a database. Currently, the class defines six data series. The class uses a rolling buffer of 500 data points that is updated dynamically. Next to x- and y-axis settings, configurable grid options, and automatic scaling capabilities, the graphical setup of *Handle_Graph* consists out of user interaction features, allowing users to stop automatic x-axis adjustments when clicking or scrolling on the plot. The update() method processes new data, replacing *None* values with placeholders, ensuring alignment of the plotted data series. The *do_plot()* method updates the rolling data buffer and fetches the latest values from *handle_MQTT*, ensuring that only valid numerical data is displayed. In order to integrate a new data series into the existing code, adjustments that have to be made are listed at the top of the *Handle_Plot* class in the form of a comment block.

Further Possible Improvements

As mentioned before, at the current state, every subscribing program subscribes to all topics for the sake of simplicity, the Message_Collector_for_plot_loop.py program however could be limited to only subscribing to the topics regarding data that is ought to be visualized by the plot.

8.5 Concept for an Energy Management System (EMS_loop.py)

8.5.1 Planned energy management logic

The following figure 25 illustrates the general concept of the proposed energy management logic that is set for the plant in the form of a flow chart. Currently, this logic can be found in the *EMS()* function in the EMS_Handler.py included in EMS_loop.py.

The purpose of the implemented EMS is to provide a functional dispatch logic for both the electrolyzers and the fuel cell, based solely on two control parameters: the SoC of the lithium-ion battery and the fill level of the hydrogen storage unit. These values are intended to be retrieved via MQTT in the final implementation.

At the beginning of each control cycle, the system checks whether both energy sources are critically low-specifically, if the battery's SoC is at or below 20% and the hydrogen storage level is at or below 10%. In such a case, a global error handler is triggered to indicate the critical condition. In order to manage the energy flow, the following rule-based logic has been implemented: If the battery charge drops to 20% or below, the fuel cell is activated to supply electrical energy and dispatching power to the grid, feeding the loads and also charging the battery. Conversely, when the battery reaches 80% or higher, the electrolyzer is activated to convert surplus energy-presumably generated by the PV production-into hydrogen.

The system employs a hysteresis-based control logic to ensure stable operation and avoid frequent switching between the fuel cell and the electrolyzer. Specifically, the fuel cell would be turned off once the battery charge reaches 65%, which allows for an additional 15% buffer to be charged using surplus PV energy (before the electrolyzers begin operating). While also maintaining a 45% margin for battery discharge, ensuring that the fuel cell does not need to be immediately reactivated. The 45 % margin is to reach the minimum SoC level is deliberately chosen as a larger value than the 15 % range before reaching the upper SoC threshold, because it is expected that this specific scenario will mainly be apparent in the nighttime. And thus, the battery consumption will be prioritized before a second fuel cell activation, expecting to enable higher efficiency and reduced switching times.

Similarly, the electrolyzer is deactivated when the battery's SoC falls to 35%, leaving sufficient room for further charging through PV surplus and maintaining a 15% buffer (before the fuel cell must be activated again). Again, the resulting 45 % margin to reach the maximum SoC value is deliberately chosen as a larger value than the 15 % range before reaching the lower SoC threshold, because it is expected that this specific scenario will mainly be apparent in the daytime. And thus, again the battery consumption will be prioritized before a second electrolyzer activation, also expecting to enable higher efficiency and reduced switching times.

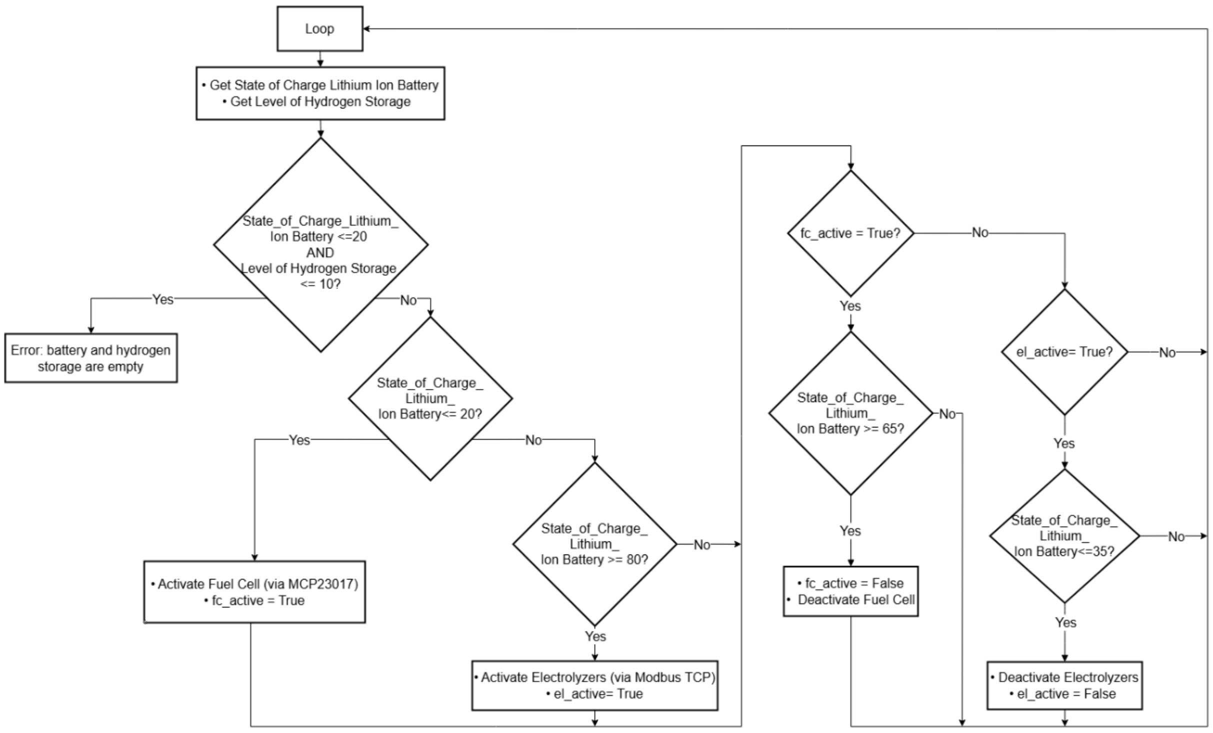

Loop
• Get State of Charge Lithium Ion Battery
• Get Level of Hydrogen Storage
State_of_Charge_Lithium_Ion Battery <=20 AND Level of Hydrogen Storage <= 10?
Yes
No
Error: battery and hydrogen storage are empty
State_of_Charge_Lithium_Ion Battery<= 20?
• Activate Fuel Cell (via MCP23017)
• fc_active = True
State_of_Charge_Lithium_Ion Battery >= 80?
• Activate Electrolyzers (via Modbus TCP)
• el_active= True
fc_active = True?
State_of_Charge_Lithium_Ion Battery >= 65?
• fc_active = False
• Deactivate Fuel Cell
el_active= True?
State_of_Charge_Lithium_Ion Battery<=35?
• Deactivate Electrolyzers
• el_active = False

Figure 25: The EMS logic shown above relies solely on the battery SoC and the hydrogen storage level as control variables, making it applicable for both islanded and grid-connected operation. It incorporates a hysteresis mechanism, with adjustable parameters that can be further fine-tuned during test runs.

8.5.2 Validation Possibilities

At this point it should be noted that the ideas for the implementation of such hysteresis were also presented in (Little, et al., 2007; Ziogoua, et al., 2011). With additional results of simulations. Unfortunately, it cannot be ensured that the system's optimal efficiency of use is demonstrated, as the currently available data is insufficient for such an analysis and no full-scale test runs of the complete system could be conducted yet at the Hamburg site. However, an extensive simulation was carried out in microsoft excel, in spite of some simplification, which implements the EMS logic presented here. For further details, see appendix 13.10 and [A_32].

8.5.3 Further Possible Improvements

Beyond these functional limitations, the operating limits of hysteresis can be changed, according to further insights during the course of the project. Furthermore, additional control strategies could possibly enhance system performance. For example, integrating weather and load forecasting would allow for more accurate planning of PV generation and energy demand. Furthermore, long-term data logging and analysis could be used to continuously improve the control algorithms. However, this would first require the development of a concept for secure and reliable long-term data storage.

8.5.4 Further Necessary Improvements

The current program does not yet support communication with the MCP23017 port expander module and the Modbus TCP/IP-based control has not been validated. Additionally, the fill levels of the storage systems (expressed as percentages) still need to be integrated into the MQTT class. Furthermore, a gradual activation strategy for the three individual electrolyzers should also be implemented in future development steps.

9 Conclusion

Motivated by the challenges such as fluctuating energy availability, limited storage, and unreliable infrastructure in off-grid environments, this thesis contributed to the implementation of a renewable backup power solution for the TIGRE Observatory in La Luz, Mexico, a microgrid, based on a hybrid storage system combining photovoltaic generation, lithium-ion batteries, and hydrogen technologies. More precisely, this thesis has presented the conceptual design and partial implementation of a control and monitoring software for a real-life example of such a hydrogen-based microgrid. A detailed summary of all relevant technical background information could be provided. Also, the description of the software and a comprehensive description La Luz-site, as well as the setup in Hamburg, was delivered. A cross-check with the requirement analysis of section 5 regarding the software design shows that the developed system demonstrates the feasibility of a modular, scalable architecture functionally integrating various interfaces, including CAN bus, Modbus RTU/TCP, Ethernet, and MQTT - across the software on a Raspberry Pi 4 and an Arduino DUE, leaving room for even more interfaces. The structure on both of these platforms enables distributed processing, low-level control, and functional data management. A secure, GPIO-interrupt-based communication between the Raspberry Pi and the Arduino was successfully implemented and showed robust and error-minimized results. It was also achieved that the code structure follows a mostly uniform structure, and operates error-tolerant, without crashing when disconnections occur, and a widespread use of uniform class structures could be included. Error detection and handling are implemented as close as possible to the point of occurrence, ensuring that potential faults are identified and processed immediately. The activation of the emergency chain is considered wherever errors are handled in the code Logging of all printouts on the Raspberry Pi and the intermediate storage of MQTT data concerning through csv files provides a feasible possibility to track down the process, in case malfunctions are to be traced or subsequent data analysis concerning system performance is to be conducted. Resource data of the Raspberry Pi can be acquired and be forwarded via MQTT, to provide and further analyze computational performance of the Raspberry Pi. Latency checking, concerning the MQTT communication between clients ensures a continuous and system-wide investigation regarding functionality of all running programs.

All in all, it can be said that the work carried out in the course of this work has advanced the project and brought it closer to its goal of a regenerative and reliable power supply for an observatory in remote location. Moreover, the team gained valuable insights and experience throughout the process. The efforts undertaken during the course of this thesis lay the foundation not only for the further development of the project but also for other control and monitoring systems that deal with similar systems.

10 Outlook

Several further developments are required to bring the system closer to a production-ready state. Varius further necessary and possible improvements that are listed along the different chapters are to be taken note of. More steps that have to be conducted include: The further transition of dummy programs to validated ones should be followed up on. While the current software setup performs reliably under laboratory conditions, further evaluation of system stability and network behavior is necessary. So far, no long-duration or large-scale stress testing, such as continuous operation of the setup beyond more than one hour and a few minutes, has been carried out in the laboratory. Also, further extended testing with the real devices in the station building are necessary. Another key priority is to provide for automated startup behavior. This may be achieved by implementing a cron job on the Raspberry Pi 1 to automatically execute the main.py script and ensuring that the GPIO pins responsible for the activation of the emergency chain is set to HIGH at boot time. Such a cronjob-file is already implemented on the Raspberry Pi 2 to start the communication with the hioki device and can be used as an exemplary model for this. Further development of the project must also address several outstanding components. These include the full integration of a graphical user interface and a database, ideally through an MQTT-subscribing application running on Raspberry Pi 1 or on a third computer communicating over MQTT via Ethernet. Since the test setup in the laboratory included a dummy Raspberry Pi (model name: Raspberry Pi 5) that was in direct contact with the Raspberry Pi 1 over the CANbus, Modbus RTU, Modbus TCP/IP and also MQTT over Ethernet, it can act as a role model for an external MQTT communication to be implemented into the system. This is also relevant regarding the future realization of the communication interface buildup between the Arduino 2 and the Raspberry Pi 1. Further additional tasks include expanding GPIO capability via the MCP23017 port expander, testing Modbus TCP/IP communication with the electrolyzers, finishing the design and installation of all electrical switchgear and sensor hardware. Also, a communication with the microinverters has to be established within the Hamburg setup. Grid-side monitoring should also be made accessible to Raspberry Pi 1 through MQTT communication with the Raspberry Pi 2. Furthermore, the error handling mechanisms that have been integrated throughout the system currently remain limited to a basic fault detection. The global error handler currently responds primarily to critical conditions via GPIO-triggered emergency shutdown. As operational experience increases, these routines may be needed to be refined to reflect actual failure patterns and enable further context-aware responses based on the severity and source of each issue.

11 Bibliography

Acromag Inc., 2005. *Technical Reference – Modbus TCP/IP - INTRODUCTION TO MODBUS TCP/IP.* [Online]
Available at: https://www.prosoft-technology.com/kb/assets/intro_modbustcp.pdf
[Accessed 18 April 2025].

Agbossou, K. et al., 2004. Electrolytic hydrogen based renewable energy system with oxygen recovery and re-utilization. *Renewable Energy,* 29(8), pp. 1305-1318.

Bähring, H., 2010. *Anwendungsorientierte Mikroprozessoren: Microcontroller und Digitale Signalprozessoren.* 4 ed. s.l.:Springer Heidelberg Dordrecht London New York.

Bandyopadhyay, S. & Bhattacharyya, A., 2013. Lightweight Internet protocols for web enablement of sensors using constrained gateway devices. *International Conference on Computing, Networking and Communications (ICNC),* pp. 334-340.

Banks, A. & Gupta, R., 2015. *MQTT Version 3.1.1 Plus Errata 01.* [Online]
Available at: https://docs.oasis-open.org/mqtt/mqtt/v3.1.1/mqtt-v3.1.1.html
[Accessed 18 April 2025].

Baun, C., 2012. *Computernetze kompakt.* s.l.:Springer-Verlag Berlin Heidelberg.

Böcker, J., 2019. *Leistungselektronik Power Electronics.* [Online]
Available at: https://ei.uni-paderborn.de/fileadmin/elektrotechnik/fg/lea/Lehre/LE/Dokumente/Skript_LE_SS2019_bilingual_2019-07-08.pdf
[Accessed 27 April 2025].

Bogensperger, J., 2022. *Betrieb eines Inselsystems zur Stromversorgung von militärischen Liegenschaften.* 2 ed. Wien: Republik Österreich / Bundesministerium für Landesverteidigung.

Böhmer, N., 2024. *TIGRE Stromlauf,* Hamburg: Böhmer, Nils.

Cavaliere, P., 2023. *Water Electrolysis for Hydrogen Production.* s.l.:Springer Nature Switzerland AG.

Colombo, P. & Ferrari, E., 2024. Access Control Integration in Sparkplug-Based Industrial Internet of. *Proceedings of the 20th International Conference on Web Information Systems and Technologies (WEBIST 2024),* pp. 380-384.

Eclipse Foundation AISBL, n.d. *Eclipse Mosquitto: An open source MQTT broker.* [Online]
Available at: https://mosquitto.org/
[Accessed 19 April 2025].

Eclipse Foundation, 2024. *Eclipse Paho.* [Online]
Available at: https://projects.eclipse.org/projects/iot.paho
[Accessed 18 April 2025].

Eltawil, M. A. & Zhao, Z., 2010. Grid-connected photovoltaic power systems: Technical and potential problems—A review. *Renewable and Sustainable Energy Reviews,* 14(1), pp. 112-129.

Enapter AG, 2025a. *EL 2.1 FW 1.12.1 Modbus TCP Communication Interface.* [Online] Available at: https://handbook.enapter.com/electrolyser/el21_firmware/1.12.1/modbus_tcp_communication_interface.html
[Accessed 19 April 2025].

Enapter AG, 2025b. *Electrolyser 2.1 Rev A. (EL 2.1 Rev A.).* [Online] Available at: https://handbook.enapter.com/electrolyser/el21a/el21a.html
[Accessed 19 April 2025].

Enapter S.r.l., 2020. *White Paper Smart Microgrid.* [Online] Available at: https://www.enapter.com/blog/white-paper-smart-microgrid/
[Accessed 17 April 2025].

Enapter S.r.l., 2025. *The Energy Management System (EMS).* [Online] Available at: https://www.enapter.com/de/energy-management-system-toolkit/
[Accessed 19 April 2025].

Eugster, P. T., Felber, P. A., Guerraoui, R. & Kermarrec, A.-M., 2003. The Many Faces of Publish/Subscribe. *ACM Computing Surveys,* Juni, 35(2), pp. 114-131.

Farrenkopf, S., 2014. *Anbindung von Echtzeitsimulationsmodellen auf embedded Systems an die Leitwarte eines virtuellen Kraftwerks durch ein Kommunikationsgateway von Modbus/TCP zu IEC61850.* [Online]
Available at: https://reposit.haw-hamburg.de/handle/20.500.12738/6914
[Accessed 23 März 2025].

Farrenkopf, S., 2024. *Konzept und Entwicklung einer Leitwarte für die Sektorkopplung mit Wasserstoff am Anwendungsfall Elektrolyse mit biologischer Methanisierung.* [Online] Available at: https://reposit.haw-hamburg.de/handle/20.500.12738/16360
[Accessed 4 April 2025].

Federau, E., 2016. *Ein Beitrag zur Konzeptionierung eines Leitsystems für ein steuerbares Microgrid.* [Online]
Available at: https://opus4.kobv.de/opus4-btu/frontdoor/index/index/year/2016/docId/3768
[Accessed 18 April 2025].

Ferrario, A. M. et al., 2020. Hydrogen vs. Battery in the Long-term Operation. A Comparative Between Energy Management Strategies for Hybrid Renewable Microgrids. *Electronics,* p. 698.

Ford, T. N., Gamess, E. & Ogden, C., 2022. Performance Evaluation of Different Raspberry Pi Models as MQTT Servers and Clients. *International journal of Computer Networks & Communications,* pp. 1-18.

Frederickson, B., 2024. *py-spy 0.4.0.* [Online]
Available at: https://pypi.org/project/py-spy/
[Accessed 27 April 2025].

Gauthier, R. et al., 2022. How do Depth of Discharge, C-rate and Calendar Age Affect Capacity Retention, Impedance Growth, the Electrodes, and the Electrolyte in Li-Ion Cells?. *Journal of The Electrochemical Society,* 4 Februar.169(2).

Geany e.V., 2025. *Geany - The Flyweight IDE*. [Online]
Available at: https://www.geany.org/
[Accessed 27 April 2025].

Giovanniello, M. A. & Xiao-Yu, W., 2023. Hybrid lithium-ion battery and hydrogen energy storage systems for a wind-supplied microgrid. *Applied Energy,* 1 September.Volume 345.

González-Pérez, J. N. et al., 2022. Eight Years of TIGRE Robotic Spectroscopy: Operational Experience and Selected Scientific Results. *Frontiers in Astronomy and Space Sciences,* Volume 9.

Hesami, M., Pourmirasghariyan, M. & Gharehpetian, G. B., 2024. Microgrids: Characteristics and Emerging Challenges. *The 11th Iranian Conference on Renewable Energy & Distributed Generation (ICREDG2024), 6-7 March, 2024, Yazd, Iran.,* pp. 1-8.

HiveMQ, n.d. *MQTT Essentials The Ultimate Guide to the MQTT Protocol for IoT Messaging*. [Online]
Available at: https://www.hivemq.com/resources/quality-of-service-in-mqtt-the-ultimate-guide/
[Accessed 9 Februar 2025].

IEA , 2021. *Renewables 2021*. [Online]
Available at: https://iea.blob.core.windows.net/assets/5ae32253-7409-4f9a-a91d-1493ffb9777a/Renewables2021-Analysisandforecastto2026.pdf
[Accessed 27 April 2025].

IEEE, 2017. *IEEE Standard for the Specification of Microgrid Controllers," in IEEE Std 2030.7-2017*. [Online]
Available at: https://ieeexplore.ieee.org/document/8340204
[Accessed 18 April 2025].

IRENA, 2019. *Innovation landscape brief: Utility-scale batteries*. [Online]
Available at: https://www.irena.org/-/media/Files/IRENA/Agency/Publication/2019/Sep/IRENA_Utility-scale-batteries_2019.pdf#:~:text=plants%20that%20can%20take%20several,to%20such%20requirements%20within%20milliseconds

Ishikawa, T., 2002. *Report IEA-PVPS T5-05: 2002: Grid-connected photovoltaic power systems: Survey of inverter and related protection eqiupments,* Customer Systems Department; 2-11-1, Iwado Kita, Komae-shi, Tokyo 201-8511, Japan: Central Research Institute of Electric Power Industry.

Jaloudi, S., 2019. Communication Protocols of an Industrial Internet of Things Environment: A Comparative Study. *Future Internet,* März.11(3).

Kurzweil, P., 2016. *Brennstoffzellentechnik: Grundlagen, Materialien, Anwendungen, Gaserzeugung*. 3. ed. s.l.:Springer Fachmedien Wiesbaden GmbH.

Kwasinski, A., Weaver, W. & Balog, R. S., 2016. 1 - Introduction. In: *Microgrids and other Local Area Power and Energy Systems.* Austin: Cambridge University Press, pp. 3-22.

Liechti, C., 2020. *pySerial.* [Online]
Available at: https://pyserial.readthedocs.io/en/latest/pyserial.html
[Accessed 20 April 2025].

Ligen, Y., Vrubel, H. & Girault, H., 2020. Energy efficient hydrogen drying and purification for fuel cell vehicles. *International Journal of Hydrogen Energy,* 45(18), pp. 10639-10647.

Little, M., Thomson, M. & Infield, D., 2007. Electrical integration of renewable energy into stand-alone power supplies incorporating hydrogen storage. *International Journal of Hydrogen Energy,* Juli, 32(10), pp. 1582-1588.

Mehalaine, R. et al., 2024. Watchdog Timer for Fault Tolerance in Embedded Systems. *Journal Européen des Systèmes Automatisés,* 57(6), pp. 1713-1720.

Meinel, C. & Sack, H., 2012. *Internetworking - Technische Grundlagen und Anwendungen.* s.l.:Springer Heidelberg Dordrecht London New York.

Meroth, A. & Sora, P., 2023. *Sensor networks in theory and practice: Successfully realize embedded systems projects.* s.l.:Springer Fachmedien Wiesbaden GmbH.

Mertens, K., 2014. *Photovoltaics: Fundamentals, Technology and Practice.* 1 ed. s.l.:John Wiley & Sons, Incorporated.

Microchip Technology Inc., 2022. *Processing Analog Sensor Data with Digital Filtering.* [Online]
Available at:
https://ww1.microchip.com/downloads/en/Appnotes/ProcessAnalogSensorDataDigitalFiltering-DS00004515.pdf#:~:text=and%20more%20advanced%20filters%20is,AN4515

Miller, H. A. et al., 2020. Green hydrogen from anion exchange membrane water electrolysis: a review of recent developments in critical materials and operating conditions. *Sustainable Energy Fuels,* 4(5), pp. 2114-2133.

Mishra, B. & Kertesz, A., 2020. The Use of MQTT in M2M and IoT Systems: A Survey. *IEEE Access,* November, Volume 8, pp. 201071-201086.

Modbus Organization, Inc., 2012. *MODBUS APPLICATION PROTOCOL SPECIFICATION V1.1b3.* [Online]
Available at: https://modbus.org/docs/Modbus_Application_Protocol_V1_1b3.pdf
[Accessed 31 März 2025].

Monk, S., 2023. *Raspberry Pi Cookbook.* 4 ed. 1005 Gravenstein Highway North, Sebastopol, CA 95472.: O'Reilly Media, Inc..

MPI Technologies AG, n.d. *Ignition 8.1.* [Online]
Available at: https://mpi.ch/ignition/
[Accessed 19 April 2025].

Naik, N., 2017. Choice of Effective Messaging Protocols for IoT Systems: MQTT, CoAP, AMQP and HTTP. *International Systems Engineering Symposium (ISSE),* pp. 1-7.

OpenEMS Association e.V., F. G., 2022. *Open Energy Management System: Introduction.* [Online] Available at: https://openems.github.io/openems.io/openems/latest/introduction.html [Accessed 19 April 2025].

Paul, H. & Leu, C., 2017. Ersatzstromversorgung. In: J. Töpler & J. Lehmann, eds. *Wasserstoff und Brennstoffzelle Technologien und Marktperspektiven.* s.l.:Springer-Verlag GmbH, pp. 157-168.

Plenk, V., 2024. *Angewandte Netzwerktechnik kompakt - Dateiformate, Übertragungsprotokolle und ihre Nutzung in Java-Applikationen.* 3 ed. s.l.:Springer Fachmedien.

Prada, M. A. et al., 2016. Communication with resource-constrained devices through MQTT for control education. *IFAC-PapersOnLine,* 49(6), pp. 150-155.

Pymodbus, 2023. *PyModbus - A Python Modbus Stack.* [Online] Available at: https://pymodbus.readthedocs.io/en/latest/ [Accessed 21 April 2025].

pypi.org, 2024. *paho-mqtt 2.1.0.* [Online] Available at: https://pypi.org/project/paho-mqtt/ [Accessed 18 April 2025].

PyPI, 2025. *psutil 7.0.0.* [Online] Available at: https://pypi.org/project/psutil/ [Accessed 20 April 2025].

Python Software Foundation, 2025. *time — Time access and conversions.* [Online] Available at: https://docs.python.org/3/library/time.html [Accessed 27 April 2025].

python-can, n.d. *python-can 4.5.0 documentation.* [Online] Available at: https://python-can.readthedocs.io/en/stable/ [Accessed 20 April 2025].

PyVISA, 2025. *PyVISA: Control your instruments with Python.* [Online] Available at: https://pyvisa.readthedocs.io/en/latest/ [Accessed 20 April 2025].

Robert Bosch GmbH, 1991. *CAN Specification.* [Online] Available at: http://esd.cs.ucr.edu/webres/can20.pdf [Accessed 7 Februar 2025].

Rohde & Schwarz, 2023. *Welcome to the RsInstrument Python Documentation.* [Online] Available at: https://rsinstrument.readthedocs.io/en/latest/ [Accessed 20 April 2025].

Sallat, M., 2018. *Das Anwendungsprotokoll MQTT im Internet of Things.* [Online] Available at: https://opus.hs-offenburg.de/files/2771/THESIS_MARIO_SALLAT.pdf [Accessed 18 April 2025].

Schmitt, J. H. M. M. et al., 2014. TIGRE: A new robotic spectroscopy telescope at Guanajuato, Mexico. *Astron.Nachr.,* 1 Oktober, 335(8), pp. 787-796.

Schwaegerl, C. & Tao, L., 2014. 1 The Microgrids Concept. In: *Microgrids: Architectures and Control.* s.l.:John Wiley & Sons, Ltd., pp. 1-24.

Şeker, Ö., Dalkılıç, G. & Çabuk, U. C., 2023. MARAS: Mutual Authentication and Role-Based Authorization Scheme for Lightweight Internet of Things Applications. *Sensors,* 17 Juni.23(12).

Simić, K. et al., 2021. Experimental evaluation of a commercially available PEM fuel cell for residential buildings application. *Proceedings of the International Conference on Efficiency, Cost, Simulation and Environmental Impact of Energy Systems,* p. 1186–1197.

Solomon, A. A., 2019. Large scale photovoltaics and the future energy system requirement. *AIMS Energy,* 27 September, 7(5), p. 600–618.

Spohn, M. A., 2022. On MQTT Scalability in the Internet of Things: Issues, Solutions, and Future Directions. *Journal of Electronics and Electrical Engineering,* 28 September.1(1).

Stevens, R. & Fall, K. R., 2012. *TCP/IP Illustrated, Volume 1.* 2 ed. s.l.:Pearson Education, Inc..

Swissolar , n.d. *Merkblatt Photovoltaik Nr. 13: Planung und Installation von stationären Batteriespeichern.* [Online]
Available at: https://www.swissolar.ch/de/wissen/solarenergie-kombiniert/batteriespeicher
[Accessed 18 April 2025].

Thangavel, D. et al., 2014. Performance evaluation of MQTT and CoAP via a common middleware. *IEEE Ninth International Conference on Intelligent Sensors, Sensor Networks and Information Processing (ISSNIP),* pp. 1-6.

Tightiz, L. & Yang, H., 2020. A Comprehensive Review on IoT Protocols' Features in Smart Grid Communication. *Energies,* 13(11).

Tröster, F., 2011. *Steuerungs- und Regelungstechnik für Ingenieure.* 3 ed. München: Oldenbourg Wissenschaftsverlag GmbH.

Valvano, J. W., 2017. *Embedded Systems: Real-Time Operating Systems for ARM Cortex-M Microcontrollers.* 4 ed. s.l.:s.n.

Valvano, J. & Yerraballi, R., 2022. *Chapter 9: Serial Communication.* [Online]
Available at:
https://users.ece.utexas.edu/~valvano/Volume1/IntroToEmbSys/Ch9_SerialCommunication.htm#:~:text=into%20the%20receive%20FIFO,tasks%20while%20data%20is%20arriving
[Accessed 28 April 2025].

Waveshare International Limited, n.d. *RS485 CAN HAT.* [Online]
Available at: https://www.waveshare.com/wiki/RS485_CAN_HAT
[Accessed 19 April 2025].

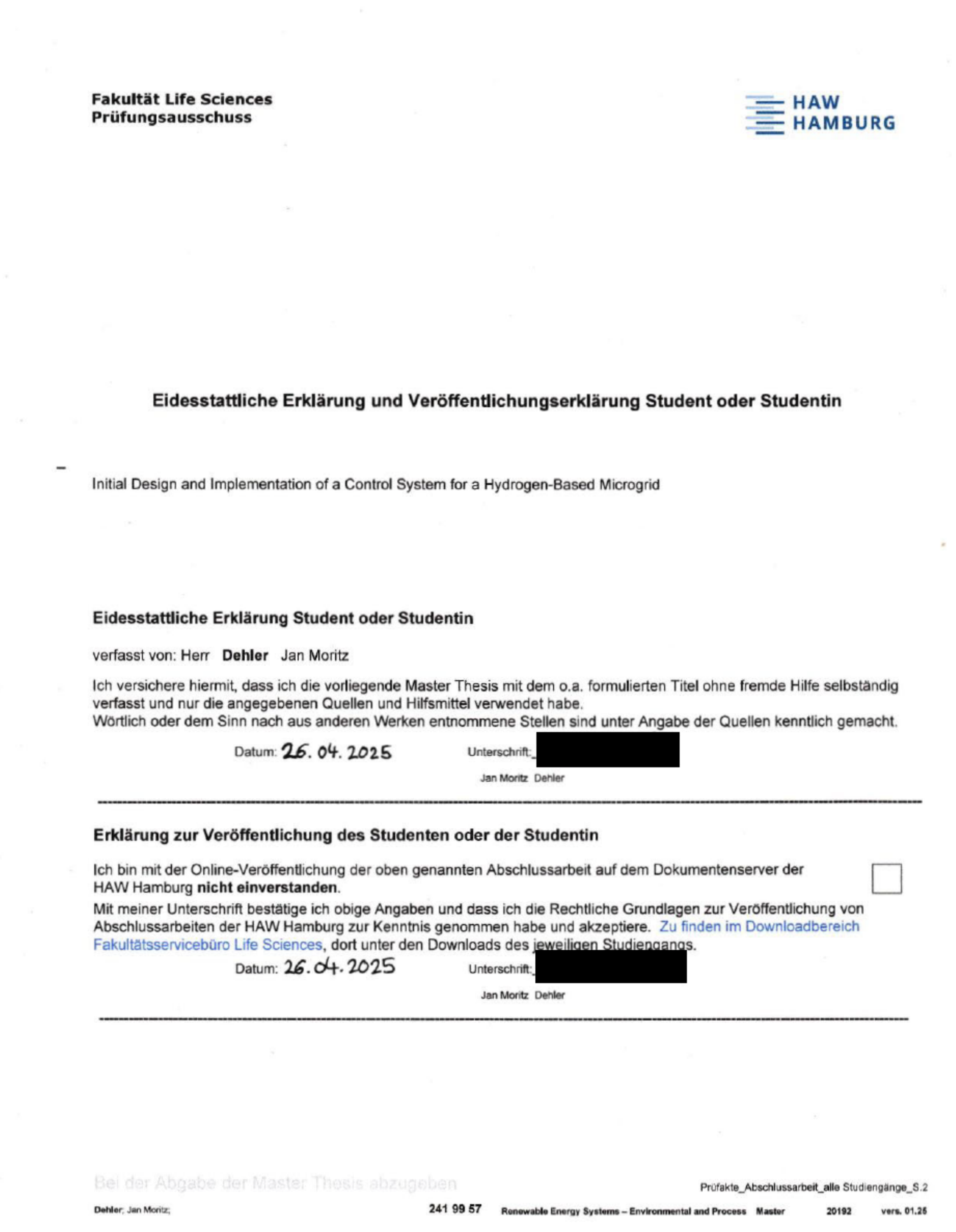
Williams, N. J., Jaramillo, P., Taneja, J. & Ustun, T. S., 2015. Enabling private sector investment in microgrid-based rural electrification in developing countries: A review. *Renewable and Sustainable Energy Reviews,* Volume 52, pp. 1268-1281.

Zapf, M., 2017. *Stromspeicher und Power-to-Gas im deutschen Energiesystem: Rahmenbedingungen, Bedarf und Einsatzmöglichkeiten*. 1 ed. s.l.:Springer Fachmedien Wiesbaden GmbH.

Zhou, Y., 2023. *Programmierung einer Vehicle Control Unit (VCU) für einen elektrifizierten Rasentraktor*. [Online]
Available at: https://monami.hs-mittweida.de/frontdoor/index/index/year/2023/docId/14542
[Accessed 18 April 2025].

Ziogoua, C. et al., 2011. Automation infrastructure and operation control strategy in a stand-alone power system based on renewable energy sources. *Journal of Power Source,* 196(22), pp. 9488-9499.

12 Declaration under Oath



Fakultät Life Sciences
Prüfungsausschuss

HAW HAMBURG

Eidesstattliche Erklärung und Veröffentlichungserklärung Student oder Studentin

Initial Design and Implementation of a Control System for a Hydrogen-Based Microgrid

Eidesstattliche Erklärung Student oder Studentin

verfasst von: Herr **Dehler** Jan Moritz

Ich versichere hiermit, dass ich die vorliegende Master Thesis mit dem o.a. formulierten Titel ohne fremde Hilfe selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel verwendet habe.
Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich gemacht.

Datum: 26.04.2025 Unterschrift: [redacted]
Jan Moritz Dehler

Erklärung zur Veröffentlichung des Studenten oder der Studentin

Ich bin mit der Online-Veröffentlichung der oben genannten Abschlussarbeit auf dem Dokumentenserver der HAW Hamburg **nicht einverstanden**. ☐

Mit meiner Unterschrift bestätige ich obige Angaben und dass ich die Rechtliche Grundlagen zur Veröffentlichung von Abschlussarbeiten der HAW Hamburg zur Kenntnis genommen habe und akzeptiere. Zu finden im Downloadbereich Fakultätsservicebüro Life Sciences, dort unter den Downloads des jeweiligen Studiengangs.

Datum: 26.04.2025 Unterschrift: [redacted]
Jan Moritz Dehler

Bei der Abgabe der Master Thesis abzugeben

Prüfakte_Abschlussarbeit_alle Studiengänge_S.2

Dehler; Jan Moritz; 241 99 57 Renewable Energy Systems – Environmental and Process Master 20192 vers. 01.25

13 Appendix

13.1 Overview Plan of the Planned Setup

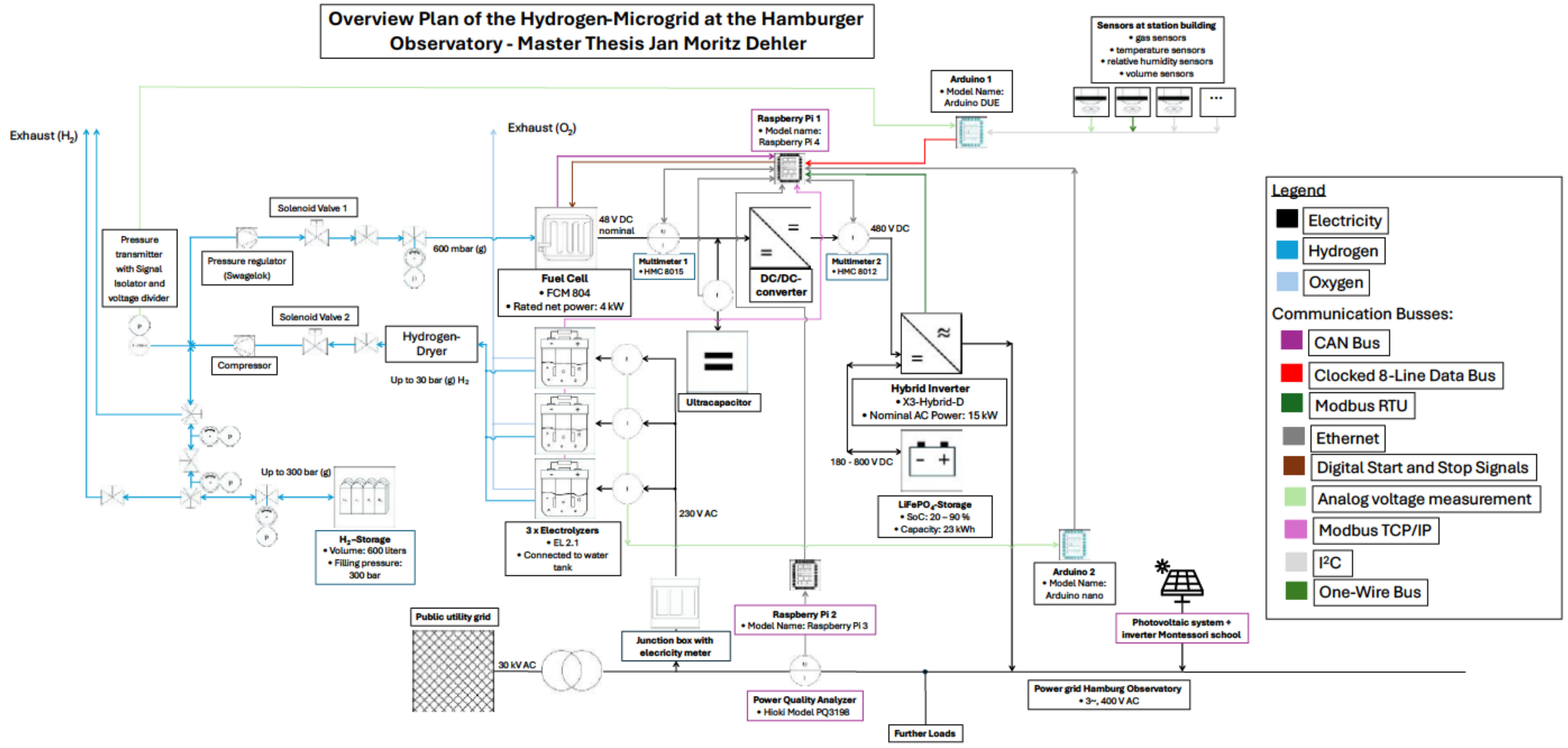

Overview Plan of the Hydrogen-Microgrid at the Hamburger Observatory - Master Thesis Jan Moritz Dehler
Sensors at station building • gas sensors • temperature sensors • relative humidity sensors • volume sensors
Arduino 1 • Model Name: Arduino DUE
Raspberry Pi 1 • Model name: Raspberry Pi 4
Exhaust (H_2)
Exhaust (O_2)
Solenoid Valve 1
Pressure transmitter with Signal Isolator and voltage divider
Pressure regulator (Swagelok)
600 mbar (g)
48 V DC nominal
Multimeter 1 • HMC 8015
Fuel Cell • FCM 804 • Rated net power: 4 kW
DC/DC-converter
480 V DC
Multimeter 2 • HMC 8012
Solenoid Valve 2
Compressor
Hydrogen-Dryer
Up to 30 bar (g) H_2
Ultracapacitor
Hybrid Inverter • X3-Hybrid-D • Nominal AC Power: 15 kW
180 - 800 V DC
Up to 300 bar (g)
H_2-Storage • Volume: 600 liters • Filling pressure: 300 bar
3 x Electrolyzers • EL 2.1 • Connected to water tank
230 V AC
LiFePO$_4$-Storage • SoC: 20 – 90 % • Capacity: 23 kWh
Arduino 2 • Model Name: Arduino nano
Public utility grid
30 kV AC
Junction box with elecricity meter
Raspberry Pi 2 • Model Name: Raspberry Pi 3
Photovoltaic system + inverter Montessori school
Power Quality Analyzer • Hioki Model PQ3198
Further Loads
Power grid Hamburg Observatory • 3~, 400 V AC
Legend
■ Electricity
■ Hydrogen
■ Oxygen
Communication Busses:
■ CAN Bus
■ Clocked 8-Line Data Bus
■ Modbus RTU
■ Ethernet
■ Digital Start and Stop Signals
■ Analog voltage measurement
■ Modbus TCP/IP
■ I²C
■ One-Wire Bus

Figure 26: The overview plan of the Microgrid setup in Hamburg (enlarged and including communication lines). The distances shown in the illustration are not to scale.

13.2 Master Slave Table Hamburg Observatory

13.2.1 Components with Master

Table 14: Master-slave table for the setup of the Hamburg site: Components with Master (1)

Component/Slave	**Fuel Cell**	**Electrolyzers**	**PV-System Montessori School**
Master	Raspberry Pi 1	Raspberry Pi 1	-
Model Name Component/Slave	FCM-804	Enapter EL 2.1	• Model name modules: SF Mono S2 Halfcut • Model name inverter: SolarEdge SE25K-EU-APAC/AUS (v1)
Amount of Components /Slaves	1	3	1
Technical Features and Characteristics Component/Slave	• Membrane: PEM • Output Voltage: 48 V • Electrical Power output up to 4 kW - voltage supply necessary, delivered by the ultracapacitors • Power optimization cycle (POC) reduces output power • Fuel consumption: Less than 70 g per kWh • Start up time: Less than 10 s	• Nominal H2 Production: 0.5 Nm^3/hr resp. 1 kg/24 h • Nominal Power Consumption per Nm^3 of H2 produced: 4.8 kWh/Nm^3 • Nominal power consumption: 2.4 kW (max. 3 kW)	• Generator Power = 27.88 kWp • Performance Ratio: 90,7% • Amount of Modules: 82
Task Component/Slave	Producing electrical power by oxidation-reduction (redox) reaction, through converting hydrogen and oxygen into water	Producing Hydrogen as variable Load through electrolysis	Renewable Power Source
Communication-Bus Slave to Master	CAN-Bus	Modbus TCP/IP via Ethernet	-
Information Content Slave to Master	Information stream concerning measurement parameters, status messages, warnings and error messages	Measurement parameters, status messages, error messages	-
Communication-Bus Master to Slave	Switching signals via MPC23017 port expander	Modbus TCP/IP via Ethernet	Radio that enables communication with a radio module, which has a contactor (3-phase relay) that interrupts the voltage to the inverter. In case the inverter has lost its connection to the outside, the inverter switches off.
Information content Master to Slave	• Sending start/stop signals • (Enable Signal on: Power up (on, but inactive state); Enable Signal off: Power down (off state) • Run Signal on: Running state; Run Signal off: Shutdown)	• Start/stop Signals for Electrolyzer and H2-Dryer • Signals for control of water tank and hydrogen-dryer	Shut down signals

Table 15: Master-slave table for the setup of the Hamburg site: Components with Master (2)

Component/Slave	**Hybrid inverter**	**Power Quality Analyzer**	**Current Sensor with transformer probe**	**Sensors in H2 Station**
Master	Raspberry Pi 1	Raspberry Pi 2	Arduino 2	Arduino 1
Model Name Component/Slave	Solax X3-Hybrid-D	Hioki Model PQ3198	DFRobot SEN0287 or similar	Telaire ChipCap 2; ADT7410; DS18B20; SEN-MQ4; SEN-MQ5; SEN-MQ8; SEN-MQ135; Grove Loudness Sensor; WIKA IS-3
Amount of Components /Slaves	1	1	3	**Sensor Name** / **Amount**: Telaire ChipCap 2: 10; ADT7410: 20; DS18B20: 5; SEN-MQ4: 2; SEN-MQ5: 2; SEN-MQ8: 3; SEN-MQ135: 2; Grove Loudness Sensor: 2; WIKA IS-3: 1
Technical Features and Characteristics Component/Slave	Nominal AC Power: 15 kW	Data storage and export for long-term monitoring	• Contact-less AC-current measurement • Direct communication with Arduino analog input pins possible	See tables 24-26 in section 13.3
Task Component/Slave	• Converting direct current (DC) electricity into alternating current (AC) electricity • Is able to (dis-)charge the battery	Measurement of parameters of AC-Grid	Measurement of AC current at the power input of the electrolyzers	Acquiring Measurement parameters from Sensors inside the station building as well as measurement of parameters from pressure transmitter
Communication-Bus Slave to Master	Modbus RTU	TCP/IP over Ethernet	Analog voltage measurement	Analog voltage measurement and I^2C
Information Content Slave to Master	Measurement parameters, status messages, warnings and error messages	Measurement parameters regarding voltage, current, power and harmonics of the AC-Grid	AC-Current entering power inputs of electrolyzers	Measurement parameters regarding air quality, hydrogen-, natural gas-, and methane concentration, temperature, relative humidity, volume and hydrogen pressure
Communication-Bus Master to Slave	Modbus RTU over EIA-485 interface	TCP/IP over Ethernet	-	I^2C
Information Content Master to Slave	Commands for querying measurement parameters	Commands for querying measurement parameters	-	Querying commands for I^2C-Sensors

Nested table in "Sensors in H2 Station" / "Amount of Components /Slaves":

Sensor Name	**Amount**
Telaire ChipCap 2	10
ADT7410	20
DS18B20	5
SEN-MQ4	2
SEN-MQ5	2
SEN-MQ8	3
SEN-MQ135	2
Grove Loudness Sensor	2
WIKA IS-3	1

Table 16: Master-slave table for the setup of the Hamburg site: Components with Master (3)

Component/Slave	**Multimeter 1**	**Multimeter 2**	**Raspberry Pi 1**	**Raspberry Pi 2**
Master	Raspberry Pi 1	Raspberry Pi 1	-	Raspberry Pi 1
Model Name Component/Slave	R&S HMC 8012	R&S HMC 8015	Raspberry Pi 4	Raspberry Pi 3
Amount of Components/Slaves	1	1	1	1
Technical Features and Characteristics Component/Slave	• Resolution 1 µV , 100 nA , 1 mΩ , 1 pF, 1 Hz, 0.1 °C • Up to 200 Measurements per second are possible	• 16-bit resolution each for current and voltage • Sampling frequency: 500 ksample/s	• Computer with operating system • Offers versatile interfaces • Higher Energy consumption than Arduinos	
Task Component/Slave	Measurement of DC current at DC/DC converter power output	Measurement of DC current and voltage at Fuel Cell power output	Data acquisition, data management and control of the grid	Data acquisition regarding grid quality
Communication-Bus Slave to Master	TCP/IP via Ethernet	TCP/IP via Ethernet	x	MQTT via Ethernet
Information Content Slave to Master	Measurement parameters and error messages		x	Sensor Data and Sensor Errors
Communication-Bus Master to Slave	TCP/IP via Ethernet	TCP/IP via Ethernet	x	x
Information content Master to Slave	Commands for querying measurement parameters		x	x

Table 17: Master-slave table for the setup of the Hamburg site: Components with Master (4)

Component/Slave	**Arduino 1**	**Arduino 2**
Master	Raspberry Pi 1	
Model Name Component/Slave	Arduino DUE	Arduino MKR
Amount of Components /Slaves	1	
Technical Features and Characteristics Component/Slave	• Microcontroller without operating system • Real-time control capability • Energy efficient	
Task Component/Slave	Collecting information content from Sensors in the station building and triggering the emergency stop chain in case critical values are measured	Collecting sensor data from current sensors located at the power input of the electrolyzers
Communication-Bus Slave to Master	Clocked 8-Line Data Bus	MQTT via Ethernet
Information Content Slave to Master	Sensor Data and Sensor Errors	Sensor Data and Sensor Errors
Communication-Bus Master to Slave	Clocked 8-Line Data Bus	x
Information content Master to Slave	Clock Signals	x

13.2.2 Components Without Master

Table 18: Table for the setup of the Hamburg site: Components without Master

Component	DC/DC Converter	Ultracapacitors	Hydrogen Storage
Model name Component	Mean Well 350W Single Output DC-DC Converter (SD-350C)	Maxwell Technologies Ultracapacitors	Hydrogen bottle bundle
Amount of Components	1	1	1
Technical features and characteristics	• DC voltage (output) 48 V • Rated current (output) 7.3 A • 12 pieces in parallel connection	• Nominal voltage: 76 V DC • Capacity: 107 Farads • 2.7V 3000F x 28 pieces in series connection	• Volume: 600 liters • Filling pressure: 300 bar
Task	Conversion of Fuel Cells Output Voltage (24 V DC) to the Inverter's required Input Voltage (480 V DC)	Act as a Battery bank (requirement for Fuel Cell)	Storage of produced hydrogen

13.3 Detailed Overview of Technical Specifications of the Hardware Components

Table 19: An overview of the general properties of the installed fuel cell. This Information is gained from [A_1].

General Properties	
Manufacturer	Intelligent Energy
Model	FCM-804
Type	PEM
Amount of components installed	1
Rated Power Output	48V DC, up to 4 kW
Efficiency (H_2 to Power Conversion)	< 70g H_2 per kWh produced
Start-up Time	< 10 seconds
Cooling & Exhaust	Air-cooled, emission: water vapor
Fuel	Hydrogen (≥99.9% purity, 500-800 mbar)
Current Status	
Installed at the Hamburg site and available for immediate use	

Table 20: A summary of essential specifications of the installed electrolyzer model, see also [A_2].

General Properties	
Manufacturer	H2 Core Systems
Model	EL 2.1
Amount of components installed	3
Type	AEM
Hydrogen Output Pressure	Up to 35 bar
Nominal Hydrogen Production	0.5 Nm^3/hour or 1 kg/24 hours
Nominal Power Consumption per Nm^3 of H_2 produced	4.8 kWh
Nominal Power Consumption	2.4 kW
Current Status	
Installed at the Hamburg site and available for immediate use	

Table 21: Basic technical information about the PV-System installed in Hamburg [A_6].

General Properties	
Model name modules	SF Mono S2 Halfcut 340 W
Model name inverter	SolarEdge SE25K-EU-APAC/AUS (v1)
Total generator capacity	27.88 kWp
PV generator surface	138.4 m^2
Amount of PV modules installed	82
Amount of inverters installed	1
Current Status	
Installed since 2020 at the Hamburg site and continuously feeding into the local grid	

Table 22: General properties of the hybrid inverter model, see also [A_7].

General Properties	
Model name	Solax X3-Hybrid-D
Amount of components installed	1
Nominal AC Power	15 kW
Current Status	
Damaged. Microinverters installed as an alternative solution for the Hamburg site. Not yet tested (as of 9th of April 2025)	

Table 23: Details regarding the installed Lithium-Ion-Battery type [A_8].

General Properties	
Manufacturer	SolaX Power Network Technology
Model	T-BAT H 23.0
Amount of components installed	1
Battery Chemistry	Lithium-ion
Nominal Capacity	50Ah
Operating Voltage Range	100V - 524V (depending on configuration)
Maximum Power Output	3.5 kW
Cycle Life	6000 cycles (at 90% Depth of Discharge)
Current Status	
Installed at the Hamburg site. Electrical connections and communication yet to be established (as of 9th of April 2025)	

Table 24: Overview of the gas sensors connected to the Arduino 1, see also [A_9-A_16].

Sensor Name	SEN-MQ4	SEN-MQ5	SEN-MQ8	SEN-MQ135
Interface	Analog and Digital interface			
Sensor Category	Gas Sensors			
Measured quantity	Concentration of compressed natural gas (CNG), methane (CH4) inside the station building	Concentration of liquefied petroleum gases (LPG), propane, methane, butane, other natural gases, etc. inside the station building	Concentration of hydrogen (H2), many hydrogen-containing gases inside the station building	Concentration of benzene, ammonia, sulfides, smoke, nitrogen oxides, and other air pollutants inside the station building
Type of measurement	Qualitative			
Amount	2	2	3	2

Table 25: Details regarding the Loudness Sensor [A_17] and the Pressure Sensor [A_18]

Sensor Name	Grove Loudness Sensor	WIKA IS-3
Interface	Analog interface	
Sensor Category	Sound Sensors	Pressure Sensor
Measured quantity	Sound pressure inside the station building	Measuring gas pressure at the gas inlet/outlet point
Type of measurement	Qualitative	Quantitative
Amount	2	1

Table 26: Overview of the used digital sensors [A_19, A_20, A_21].

Sensor Name	**Telaire ChipCap 2**	**ADT7410**	**DS18B20**
Interface	I^{2}C		One-Wire
Sensor Category	Chip Caps	ADTs	DS18s
Measured quantity	Measuring Temperature and relative humidity inside the station building	Measuring Temperature inside the station building	Measuring Temperature inside the station building
Measuring range	Relative Humidity: 0 – 100 % Temperature:-40 °C to +125 °C	-55 °C to +150 °C	-55 °C to +125 °C
Precision	Relative Humidity: ±2,0 %RH (at 20-80 %RH, 25 °C, 5 V) Temperature: ±0,3 °C (at 25 °C, 5 V)	±0,5 °C (-40 °C to +105 °C), at 3.0 V Supply voltage	max ±0,5 °C (at -55 °C to +125 °C) max ±2 °C (at -10 °C to +85 °C)
Resolution	Relative Humidity: 14 bit (0,01 %) Temperature: 14 bit (0,01 °C)	13 bit (0,0625 °C) or 16 bit (0,0078 °C)	9 to 12 bit
Supply voltage	2,3 V to 5,5V	2,7 V to 5,5 V	3,0 V to 5,5 V
Amount	10	20	5

13.4 Arduino 1: Detailed explanation of Sensor Data Acquisition Function Files

As noted in the general description of the Arduino Code, see chapter 7, the foundation of the following .ino files was developed by previous colleagues of the team. The author's main contribution focused more on the design and implementation of code segments responsible for the data- and transmission management regarding the Raspberry Pi 1 - Arduino 1 interface (j_copy_Bytearray_for_Pi.ino, k_ISR.ino and l_Serial_Data_to_Pi.ino). Also, the m_watchdog.ino and n_handle_errors.ino files were provided by the author.

e_analog_sensors.ino

The *gasAlarmStatus()*-function and the *gasRead()*-functions manage the gas sensor readings.

gasAlarmStatus() detects gas alarms through the digital inputs for any gas sensor connected to the microcontroller. A global byte-array called *gasAlarms[4]* stores the error-messages, whereas the index of the array stands for the gas-sensor-type. The error detection-logic goes as follows: If an alarm is being detected, for example for the Air Quality gas sensor (=the respective digital pin is on a LOW-level), the byte *gasAlarms[0]* will be updated. Such that if the error is being detected on the first AQ-Sensor, then the first bit of *gasAlarms[0]* will be set to 1. If the second AQ-Sensor is detecting an invalid gas concentration, the second bit will be set to 1. This logic supports up to 8 sensors of each type. This logic will be later revisited by the program responsible for processing the incoming Arduino Data on the Raspberry Pi 1 in order to detect which specific sensor delivered an invalid value. With the current state of the program, unlike it is the case with any other sensor type, the gas sensors will only detect a threshold of maximum values.

The *gasRead()*-function has the task of reading the specific gas concentration levels from the respective gas sensors. Since a sufficient error detection was already established by the *gasAlarmStatus()*-function, no further error detection methods are being considered at this point (which could be implemented in later versions). Furthermore, *gasRead()*, is applying a median filter for noise reduction (*int_filter()*, used within *gasRead()*, see section 7.2.6.

Additionally, e_analog_sensors.ino includes the function *loudnessRead()* and *Read_pressure()*. It processes the sound and pressure sensor data using a similar approach to *gasRead()* (see *loudnessRead()*). Unlike with the gas sensors, plausibility checks are required in this case.

f_I2C.ino

This implementation of I^2C communication manually configures the appropriate data (*SDA*) and clock (*SCL*) lines depending on the selected bus (*I_2_Init(int busNr)*), and handles all key protocol operations: start conditions (*I2C_start()*), stop conditions (*I2C_stop()*), bit-level transmission (*I2C_bit_write(bool bitwert)*), byte-level transmission (*I2C_write(uint8_t bytewert_in, bool ACK)*), bit reception (*byte I2C_bit_read()*), byte reception (*byte I2C_read(bool ACK)*), and timing control via delays (*wait()*). All of these functions were developed in-house and are derived from program code of previous projects. The long-time operation with the same I^2C-sensor types showed high functionality and reliability in these projects. Furthermore, the self-designed I^2C-interface offers the benefit of full transparency and debug possibility, in case any communication issues arise. The standard Arduino I^2C library was excluded from use, as it showed stability issues and system freezes during long-term operation.

g_chipcaps.ino

The function *readChipCaps()* reads temperature and humidity data from the ChipCap sensors over I^{2}C. Furthermore, it initializes the I^{2}C connection for each sensor (*I_2_Init(sensNr + 1)*), starts communication (*I2C_start()*), reads humidity and temperature values (*I2C_read(1), I2C_read(0)*), and stops communication (*I2C_stop()*). The retrieved data is processed according to the sensor's datasheet, see [A_19], including unit conversion and filtering. It also performs error detection by checking for invalid or extreme values, logging errors in its respective error register and applying the median filter (*float_filter(...)*).

h_adt_7410.ino

In this file, the reading of temperature data from the ADT7410 sensors over I^{2}C is accomplished. *ADT_ini()* configures each sensor to 16-bit resolution, establishes I^{2}C communication for each port (*I_2_Init(i)*), and writes the necessary configuration values (*I2C_write()*). The function *Read_ADT7410s()* reads the temperature data by retrieving high and low bytes (*I2C_read()*), converts the raw data into a floating-point temperature value, and applies a median filter to smooth the readings (*float_filter(...)*). Similar to g_chipcaps.ino, the code includes error detection, checking for sensor availability, extreme values, and abrupt temperature changes, logging errors accordingly.

i_one_wire_bus.ino

This code initializes and reads temperature data from multiple DS18B20 sensors using the OneWire protocol, see also (Meroth & Sora, 2023, pp. 278-291). It starts by initializing the OneWire communication and assigning addresses to each sensor (*oneWireIni()*). The resolution for each sensor is currently set to 12-bit precision. The function *getDsTemp()* retrieves temperature readings by sending a request (*sensors.requestTemperatures()*) and then fetching values for each sensor *(sensors.getTemp(ds18_addresses[sensNr])*). The raw data is converted into temperature values, and error detection is performed by checking for invalid readings (all zeros, all ones) or values outside defined thresholds. The function also detects sudden temperature jumps and logs errors accordingly. Afterwards, the median filter is applied to smooth temperature readings (*dsTemp[sensNr] = dsTempStor[sensNr][float_filter(dsTempStor[sensNr])]*). Debugging outputs provide real-time sensor values and error information.

13.5 Detailed explanation of Publishing Programs

13.5.1 Arduino_Communication_loop.py

This program is of special interest, because it enables the communication with Arduino controllers over the GPIO pins including the analysis and normalization of the recorded telegram and publication to the MQTT broker. To ensure a better understanding, it is advisable to review the Arduino_Communication_loop.py script, including its integrated classes next to reading this chapter, due to the considerable complexity of the code, see [A_30].

Classes only used for this program

increase_performance.py

This class defines a static method *set_process_priority(pid, nice_value)*, which adjusts the priority of a running process using the renice command. The method executes a subprocess call to modify the process priority based on the provided pid (process ID) and nice_value (priority adjustment). The function attempts to execute the command via *subprocess.run()* with sudo-privileges, ensuring it runs with sufficient permissions. This class was implemented because communication speeds depend on the pulse time of the clock signal sent by the Raspberry Pi 1. Increased computational power leads to faster program execution, which in turn reduces the pulse time.

Overview

Similar to the general publishing program scheme, this application processes the acquired raw data coming from Arduinos by first normalizing it and then checking its plausibility before finally sending the formatted results to the MQTT broker. The acquisition of the data is achieved through the utilization of the Raspberry Pi's GPIO via the pigpio-library. Previously conducted tests with the standard RPi.GPIO library resulted in faulty measurement data acquisition. The reason why this program is explained in more detail here is that, unlike our other services where the raw data is contained strings that can be simply split into parts via string manipulation, this application processes a telegram that encapsulates a variety of data from different sources that are only being forwarded by the respective interacting Arduino Controllers. Each segment, or "snippet" of an acquired telegram may require a distinct processing approach, for example, interpreting the different error registers and different data format from the different types of sensors. The program also enables a possibility to interface with more than one Arduino. But currently, only the interaction with the Arduino DUE controller is validated.

Main Script Structure and Flow

In the initialization phase of the program, the names of the respective pins for the GPIO-Interface are declared and an index variable called *Arduino_inx* is initialized. It is later used in the secondary *while True* loop in a match-case method to provide for the possibility to switch between different Arduino boards that are to be queried. Furthermore, the list *private_topics* differentiates between the data coming from different Arduinos (see publishing method at the end level of the secondary *while True* loop). High process priority is enabled though increase_performance.py. The reason for this is that only with this method, the loop cycle times are being accomplished for this specific program are in an acceptable range (currently at 0.2 to 0.35 seconds on average) and the majority of the execution time is spent at the execution of the *Arduino_Receiver.read_data()*-function. Subsequently, the pigpio-service is started to enable GPIO operations.

An object (Arduino_*Receiver*) of the class *Arduino_Data_Acquisition* is instantiated to manage the low-level data acquisition from connected Arduino controllers. Simultaneously, instances of *Arduino_X_Normalizer_and_Checker*, are created in order to process the raw data. Currently the code is designed, in such a way that a generic Normalizer_and_Checker cannot be used for all connected Arduinos, because the length and information content of the respective telegrams will deviate. Moreover, due to the design of the transmission protocol, the plausibility check and normalization was designed to go hand in hand, as "normalization" in this context primarily means reconstructing 16-bit integer numbers from the transferred bytes and plausibility check refers through the checksum verification, and extraction the sensor-specific error messages.

Data Acquisition Component

The *Arduino_Communication_Data_Acquisition* class is dedicated to the low-level acquisition of sensor data using the Raspberry Pi's GPIO pins. It sets the appropriate pin modes-INPUT for the SDA lines and OUTPUT for the SCL and interrupt pins-and is responsible for forming data bytes from the incoming bits.

Its key methods include the *start()* function, which resets internal counters and flags, clears previous raw data, and sends a brief signal on the interrupt line via *send_signal_for_telegram_reset()*, which toggles the interrupt pin HIGH and then LOW to signal the start of a new data telegram. This is the method, where the *byte_values_to_send[]* - array on the Arduino 1 (see *ISR_PI_II()* in k_ISR.ino) is prepared with the current measurement data. *read_data()* is the main function of this class. It activates the clock line and sequentially reads bits from four *SDA* pins. In the function, the incoming bits are shifted and accumulated until a full 8-bit byte is constructed (*self.temp_byte*). An if-statement ensures that the first received byte matches the expected start marker-the ASCII code 36 representing the '$' symbol and the process continues until the complete telegram is assembled.

Data Normalization and Plausibility Checking Component

Once the raw data is captured, the respective *Arduino_X_Normalizer_and_Checker* takes over to process and validates the telegram. The processing workflow begins with an initialization phase, where the start() method resets all internal variables, counters, and arrays used for storing processed sensor values and error registers. The *analyze_data()*-function carries out the main process: at the heart of the data processing is a multi-stage *elif*-construct combined with an infinite *while True* loop that sequentially checks individual elements of a Boolean array (*snippet_done*). This structure clearly separates the logical snippets of the telegram. In case an error is detected during the interpretation of a snippet, or once the telegram has been fully processed, *handle_error()* will be called, and the while loop of the function will be terminated via *break*-statements. The program sequence of the *analyze_data()*-function can be observed in the flowchart of figure 27.

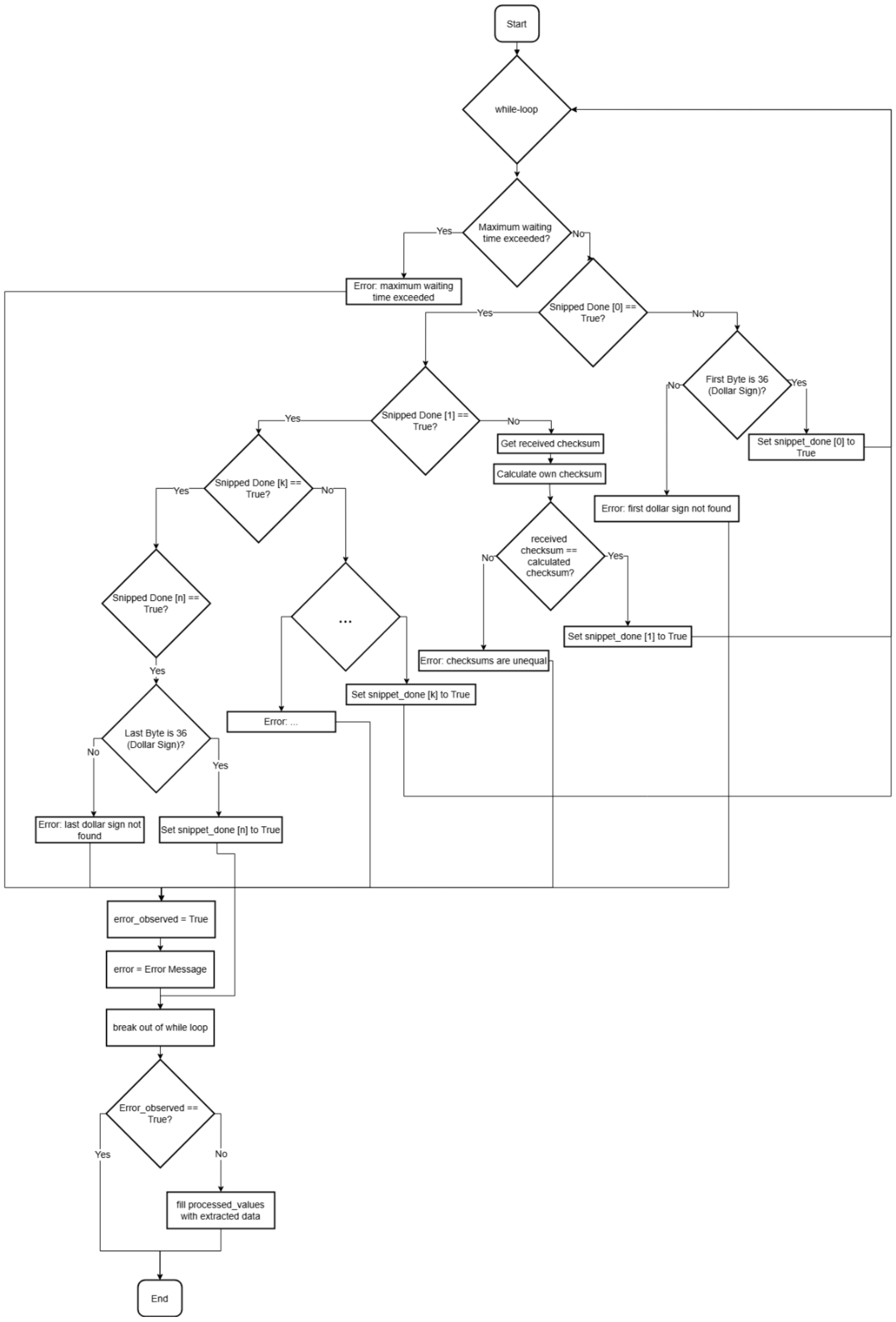

Start
while-loop
Maximum waiting time exceeded?
Yes
No
Error: maximum waiting time exceeded
Snipped Done [0] == True?
First Byte is 36 (Dollar Sign)?
Set snippet_done [0] to True
Error: first dollar sign not found
Snipped Done [1] == True?
Get received checksum
Calculate own checksum
received checksum == calculated checksum?
Set snippet_done [1] to True
Error: checksums are unequal
Snipped Done [k] == True?
...
Set snippet_done [k] to True
Error: ...
Snipped Done [n] == True?
Last Byte is 36 (Dollar Sign)?
Error: last dollar sign not found
Set snippet_done [n] to True
error_observed = True
error = Error Message
break out of while loop
Error_observed == True?
fill processed_values with extracted data
End

Figure 27: Flowchart of the analyze_data()-function, responsible for the encoding of the acquired raw data telegram.

The specific code structure not only defines the logical sections of the telegram but also dynamically adapts to varying snippet lengths, depending on the number of error messages contained. Table 27 provides a structured overview of how the *analyze_data()* function parses the telegram, including the extracted variables and the corresponding index values in *snippet_done[]* that are responsible for each subsection.

Table 27: Illustration of the naming of variables that are to be extracted from snippets of the acquired telegram, including their respective index value of snippet_done[].

Variable	*Start marker ('$')*	*checksum*	*errorCountCC*	*errorRegCC[]*	*errorCountADT*
Index	0	1	2	3	4
Arduino variable	*errorRegADT[]*	*errorCountDS18*	*errorRegDS18[]*	*errorCountLoudness*	*errorRegLoudness[]*
Index	5	6	7	8	9
Arduino variable	*errorCountPressure*	*errorRegPressure*	*gasAlarms[]*		
Index	10	11	12		
Arduino variable	*ccHygHigh[]*	*ccHygLow[]*	*ccTempHigh[]*	*ccTempLow[]*	
Index	13				
Arduino variable	*AdtTempHigh[]*	*AdtTempLow[]*	*dsTempHigh[]*	*dsTempLow[]*	
Index	14		15		
Arduino variable	*H2ValHigh[]*	*H2ValLow[]*	*petrMethValHigh[]*	*petrMethValLow[]*	
Index	16		17		
Arduino variable	*natValHigh[]*	*natValLow[]*	*airQualValHigh[]*	*airQualValLow[]*	
Index	18		19		
Arduino variable	*loudnessValHigh[]*	*loudnessValLow[]*	*PressureValHigh*	*PressureValLow*	*End marker ('$')*
Index	20		21		22

The processing logic checks for possible errors, by applying the *handle_error*()-function. With the current software version, the following errors are being detected:

- Incorrect start marker
- Checksum mismatch
- Python exceptions of the respective snippets
- Incorrect end marker

Whereas the checksum is being validated by a comparison of the received checksum and an own calculated checksum using an XOR operation on the bytes. Errors that are being transmitted through the error register inside the telegram are not listed here, because they are treated as information and not considered as an error (see error handling function in the Arduino code). They are being interpreted with the help of the respective *Error_handling*()-class, see *Tools_for_Arduino_Communication_Data_Normalization_and_Plausibility_Check.py*. In order to convert the pairs of 8-bit numbers of the sensor values back into a single signed 16-bit integer, the helper class *two_byte_value* of this script is used. At the end of the *analyze_data()*-function processed values are accumulated in the array *processed_values*. The table 28 provides a further overview of the most important variables used to orchestrate the program flow of *Arduino_X_Data_Normalization_and_plausibility_check*.

Table 28: Listing and Explanation of certain control variables used for Arduino_X_Data_Normalization_and_plausibility_check.

Variable	Purpose
snippet_done	A list of 20 Boolean values indicating which processing steps have been completed. Each index represents a specific processing step.
indx_for_raw_data	A counter that tracks the current index in the received sensor data during analysis. It is used to iterate through the raw data.
indx_1	A helper index used within certain processing steps as a counter for sensor error messages or value iterations.
indx_2	Another helper index, primarily used for iterating through different sensor values, especially for indexing within sensor groups.
string_to_debug	A string that accumulates detailed debug information during the analysis process. It is used for error tracking and monitoring execution flow.

Other variables store sensor data, error counters, checksums, or flags for data processing. They help organize received values, detect errors, and analyze sensor readings.

13.5.2 Further Possible Improvements

Currently, the data acquisition class waits for the maximum number of elements in the telegram. This method has proven to be functional and reliable, as it ensures proper detection of the start and end of the telegram since both sender and receiver rely on a fixed telegram length. However, this approach may also result in a significant amount of time being spent reading irrelevant data, as the total telegram length is designed to accommodate the worst-case scenario where all sensors send out error messages. In a normal operation mode, when no error messages are going out from the sensors, the relevant data of the telegram does not fill the entire telegram length. A more efficient approach might be to make the total telegram length variable and integrate the checksum verification (which currently takes place in the analyzing function) directly into the data acquisition class. As a test, the checksum could be calculated whenever a second dollar sign is detected in the telegram. If it matches the transmitted checksum, the telegram can be considered fully received and the program may exit the *read()* function. However, a more advanced checksum calculation may be conceivable than the current XOR-checksum verification.

13.5.3 HMC8015_loop.py

HMC8015_Data_Acquisition.py

For the communication with the HMC 015 device, the RsInstrument library is used, see (Rohde & Schwarz, 2023). The class defines the physical quantities to be measured, such as RMS-voltage and RMS-current. Furthermore, the VISA timeout is configured to 2 seconds, meaning that if the instrument does not respond within this time, a timeout error is triggered. Similarly, the OPC timeout is also set to 1 second, defining the maximum time the device waits for a command to complete before considering it unsuccessful. Additionally, the class enables instrument status checking, which ensures that the device is continuously monitored for errors during communication. If the instrument reports an error status, it is immediately detected.

To maintain a reliable operation, the script also calls *self.hmc8015.clear_status()*, which clears all previous error or status messages stored in the device. Furthermore, the script explicitly configures the measurement ranges for voltage (300 V) and current (133 A). By setting these ranges manually, the system avoids the need for an automatic measurement range adjustment, which could introduce measurement delays.

HMC8015_Data_Normalization.py

The class processes the raw data string coming from the *read()*-method of the data acquisition class, removes unwanted characters and separates values into distinct measurement parameters.

HMC8015_Plausibility_Check.py

This class validates the acquired measurement data by applying predefined plausibility checks. It defines threshold values for voltage and current to detect invalid readings. If a measurement exceeds these limits or returns a value of exactly zero, the system flags it as an error. With the current version of the software the maximum and minimum values are chosen to be extremely high/ and extremely low, in order to not throw errors when test runs are being conducted. These values can be specified more in the future. The script also provides a framework for adding further plausibility checks in the future.

13.5.4 HMC8012_loop.py

13.5.5 HMC8012_Data_Acquisition.py

The HMC8012_Data_Acquisition.py script is responsible for communicating with the HMC8012 digital multimeter. It establishes a connection with the instrument using the PyVISA library, see (PyVISA, 2025). It explicitly sets the measurement mode to AC voltage (400V range) and configures the Analog-to-Digital Converter (ADC) to operate at its fastest sampling rate to enhance measurement efficiency.

HMC8012_Data_Normalization.py

This class processes the raw data string obtained from the read() method of the data acquisition class. Since the HMC8012 current clamp connected to the HMC8012 returns values in 10mA increments, the class scales the values accordingly before passing them to the next step.

HMC8012_Plausibility_Check.py

Compare to HMC8015_Plausibility_Check.py.

13.5.6 CAN_BUS_01_loop.py

Since the fuel cell has a local control unit that sends out error messages in case of any malfunction, a Plausibility_Check-class is omitted here. The code structure does not deviate to the other publishing programs. Except for the fact that no minimal loop execution time is provided, because the program has to react fast enough for incoming messages coming from the fuel cell.

Canbus_Data_Acquisition.py

The CAN_Data_Acquisition.py script is responsible for establishing and maintaining communication with the CAN bus. To manage this communication, the python-can library was used (python-can, n.d.). The Raspberry Pi interfaces with the MCP2515 CAN controller, which is mounted on the RS485 CAN HAT, via the SPI protocol. This SPI-based communication enables the MCP2515 to translate messages between the Pi and the CAN transceiver (SN65HVD230), which in turn interfaces with the physical CAN bus. The CAN channel *can0* is configured with a bit rate of 500 kbit/s, following the manufacturer's specification. The system uses the socketcan backend, which integrates CAN interfaces into the Linux network stack, providing native support and low-level efficiency. Error detection, reconnection logic, and debug output are integrated into the script. See previous sections regarding data acquisition. Blocking reads (*recv()*) are used intentionally, due to the overall timeout control delegated to the higher-level script structure. In case of communication problems, the interface is safely shut down and reset (*see handle_errors()*).

CAN_Data_Normalization.py

This class is responsible for processing raw CAN messages by extracting relevant signal values and converting them into a structured format. Each message consists of eight hexadecimal fields, which are parsed and transformed into meaningful numerical values. The format of a raw CAN message typically follows a predefined structure, including a timestamp indicating when the message was received, a unique message ID, an indicator for the type of message (*S Rx*), the data length (always 8 bytes in this case), the hexadecimal values representing the message data, and the CAN bus channel through which the message was received.

The processing flow begins with the *CAN_processing()* function, which first verifies the format of the received data using *check_can_message()*. If the format does not match the expected structure, the function *handle_errors()* is triggered to manage errors accordingly. During validation, the individual components of the message are extracted and stored in class variables. Once the data is properly structured, the message is interpreted based on its CAN ID. The interpretation logic follows specifications provided by the manufacturer, which were documented in an instruction excel table. This follows the logic of hexadecimal values being converted into signed or unsigned decimal numbers as needed.

At plant operation, the Fuel Cell will send out CAN messages one after the other, with different ID's, containing various data points that are extracted and processed accordingly by *CAN_processing()*. For instance, messages with ID 0318 extract the software version from the CAN bus, while ID 0320 determines the total runtime in hours and the total energy consumption. Messages with ID 0328 and 0378 process status flags and retrieve error conditions, whereas ID 0368 extracts system state information, load logic, and output bit status.

Electrical parameters such as output power (W), voltage (V), current (A), and anode pressure (mbar) are read from ID 0338, and temperature values, along with DCDC converter setpoints, are retrieved from ID 0348. The louver position and fan speed duty cycle are monitored through messages with ID 0358. Each message type undergoes a structured process where raw hexadecimal values are processed, necessary conversion formulas are applied, and the results are mapped to predefined output variables.

In cases where the fuel cell outputs an error message (notably from ID 0328 or ID 0378), an excel file containing fault codes is accessed using the openpyxl library. This specific excel file, provided by the manufacturer, is essential as it contains descriptions of the various error messages. The file is being accessed due to the large number of possible error conditions and the character length of their descriptions. It is crucial that this excel table remains unchanged to maintain the functionality of the script and guarantee reliable message interpretation.

13.5.7 get_resource_usage_loop.py

Unlike the previously discussed scripts, this script establishes the connection to the MQTT broker through the *ServiceResourceRessource_Data_Receiver* class. This class is initialized in the main program via the *Ressource_Data_Receiver* object. The script integrates the psutil library, see (PyPI, 2025), to monitor resource usage across all running Python processes. This allows it to track CPU, memory, and other system parameters in real-time, providing general insights into overall system performance.

The reason for initializing the *mqtt_client* object of this program within the *Data_Acquisition* class is that the script publishes not only the payload regarding the program's software timestamp, *error_observed* -parameter and *error* message, but also it publishes acquired system resource data of all detected python programs under the respective program-specific topic. In order to do that, the script initializes the *Ressource_Data_Receiver* object, responsible for collecting and publishing system resource usage metrics and the *get_all_service_resource_usages()* function of the responsible class iterates over all running processes, identifies Python-based services, and invokes the function *get_service_resource_usage()* for each one. This secondary function retrieves critical metrics such as CPU utilization, memory consumption, RAM (Random Access Memory) usage, swap usage, and system load averages. These values are then validated through the *Ressource_Data_Plausibility_Check* class, which ensures they fall within expected operational ranges. If any anomalies are detected, an error message is generated and included in the MQTT payload. The MQTT message's topic will be then adjusted to each of the names of the investigated programs.

Further possible improvements

Currently, the program only generates resource data based on the running Python programs (see function argument *service_name* of *get_all_service_resource_usages()* in *get_resource_usage_Data_Acquisition.py* is set to "Python". This means that, in order to obtain an accurate view of the overall system load, the Raspberry Pi should not run additional programs-such as an internet browser-during test runs in the laboratory or during plant operation.

If non-Python programs (e.g., additional software for data visualization) are introduced to the system in the future, their resource usage should also be considered. This could be achieved by extending the *get_all_service_resource_usages()* function for a second argument.

13.5.8 Modbus_RTU_loop.py (Dummy)

This module was originally intended to establish communication with the inverter. Currently, a test script running on the external Raspberry Pi (model name: Raspberry Pi 5) sends randomly generated data. The setup is functional and can be reused as a code template for future applications, particularly when additional components are to be integrated via Modbus RTU. In terms of code structure, there are no major deviations to the other publishing programs. The test script used for sending data is explained in appendix 13.6.3.

Modbus_RTU_Data_Acquisition.py

This Python class implements Modbus RTU data acquisition over a serial interface. The script uses the pyserial library, see (Liechti, 2020), to establish and manage communication via the /dev/ttyAMA0 UART interface, with a fixed baud rate of 19200 bps as required by the hybrid inverter. The *read_modbus_data()* method is responsible for reading data packets from the serial Modbus interface. Before each read operation, the function checks whether the UART input buffer already contains a significant amount of unread data. If the buffer exceeds 10 bytes, the system raises a warning that the receiving end might be too slow to process incoming messages. This mechanism was implemented due to the testing setup with the sending script, handing over a new message every 50 ms.

If the buffer check is passed, the method reads a fixed block of 8 bytes and evaluates the result. In cases where the received data is empty or invalid, a corresponding error is flagged. When valid data is obtained, a timestamp is recorded, and the data is printed to the console along with the current buffer size.

To maintain stability and transparency, the class also includes a method *is_uart_buffer_full()* for buffer monitoring and a *close_connection()* method to safely terminate the serial connection when needed. The *handle_errors()* function ensures that all errors are consistently processed: it logs the issue, resets the connection status, and clears any existing data to prevent propagation of invalid readings.

13.5.9 modbusTCP_elektrolyseur_loop.py (Dummy)

This script is intended to act as an outline program to enable the communication with the electrolyzers via Modbus TCP/IP. Currently this is only realized by an Ethernet communication over the network switch between the Raspberry Pi1 and the testing-raspberry (model name: Raspberry Pi 5). The description of the sending script can be found in section 13.6.4.

modbusTCP_elektrolyseur_Data_Acquisition.py

For the communication over Modbus TCP/IP, the pymodbus library (see: (Pymodbus, 2023)), is used. The class establishes and manages communication over Ethernet with a Modbus TCP/IP server. Upon initialization, the class sets key communication parameters, including the target device's IP address (*192.168.0.10*), Modbus port (*5020*), and unit ID (1). A dictionary, *REGISTER_DEFINITION*, the register names to their respective Modbus register addresses, lengths, and data types. This register map includes both holding registers and input registers, which corresponds to: (Enapter AG, 2025a). The *connect()* method creates a synchronous TCP connection to the Modbus server using *ModbusTcpClient*. The *read()* method iterates through all entries in the register definition and determines, based on metadata, whether to access the holding register table (function code *3*) or the input register table (function code *4*). Each register is queried individually, and the response is decoded based on the expected data type. For example, 32-bit floats are reconstructed from two 16-bit registers using the struct module. Similar bit-shifting techniques are used for handling 32- and 64-bit integers. The decoded values are stored in the *received_values* list.

Further necessary improvements

Error-handling registers like the dryer or water tank bitmask codes are defined in the Modbus specification (Enapter AG, 2025a) but are not yet respected by the current implementation. An excel file containing the systematics for error interpretation of all error codes lies in the directory and could later be accessed by the script. Such an error-message interpretation strategy via an excel file is already provided in *Canbus_Data_Normalization.py* and can be used as an outline.

13.5.10 SolaxHybrid_loop.py (Dummy)

The SolaxHybrid_loop.py script is based on an original program previously developed by previous colleagues of the team in order to collect data from the previously implemented inverter. As described in section 4.2.4, communication with the inverter and the Raspberry Pi takes place via Modbus RTU. But due to a malfunction of the original inverter model during the course of the project, data is no longer collected through a physical interface. Instead, the current implementation serves as a simulation framework, generating random values to mimic all data that would otherwise be retrieved from the inverter model. These simulated data points are then forwarded to the MQTT-server for further processing. This program is therefore used as a role model for future inverter control programs and also as a dummy-program that periodically causes additional traffic on the MQTT-server for testing reasons.

SolaxHybrid_Data_Acquisition.py

The SolaxHybrid_Data_Acquisition.py script is responsible for acquiring and processing data from the inverter. The class initializes a connection, defines a range of memory addresses to read from, and processes the retrieved values into a structured dictionary. The data consists of parameters such as grid voltage, current, power, PV voltage, PV current, temperature, battery state, and fault messages. Under normal conditions, data would be extracted using the Modbus communication protocol. However, due to the lack of a physical inverter, the script instead generates random values for testing purposes, as mentioned before.

13.5.11 blink.py

This program can be considered as an utility program that makes the Raspberry Pi's internal LED blink, allowing users to visually confirm that the Pi is running during operation. It neither publishes nor subscribes to any topic. It is automatically activated when main.py starts.

13.5.12 get_svg_files_loop.py

This Python script provides the possibility of profiling active Python processes on a system using the py-spy sampling profiler, see (Frederickson, 2024). Its primary purpose is to record performance data from running Python scripts and save the results as .svg flame graphs for further analysis. These visualizations can help identify bottlenecks, excessive CPU usage, or performance anomalies in long-running or resource-intensive programs.

Upon execution, the script first identifies all active Python processes using the pgrep utility. For each process found, it attempts to extract the name of the executing script by querying the command line arguments associated with the process ID. This name is used to generate a timestamped filename for the output file, ensuring that each flamegraph is traceable to its corresponding process and runtime. The script launches a separate py-spy instance for each detected Python process. Each profiler instance runs in the background and writes its output as a flamegraph in SVG format to a designated directory (svg_dateien), which is created automatically if it does not already exist. Note: The .svg files can be opened and viewed directly in any preferred web browser. To activate profiling of all running python scripts, the corresponding script path must be uncommented within the *scripts* lists of main.py. Once this is done, executing main.py will initiate the profiling sequence. Because profiling consumes considerable system resources, it is intended only for testing purposes. Consequently, the timeout settings (*timeout_limit*) in the receiving scripts must be adjusted accordingly to suppress unnecessary error alerts.

The recorded .svg files will be stored in the directory: ...\total_serial_v52\svg_dateien

13.6 Sender Scripts

All of the sender scripts described below are located in the total Serial folder, see [A_30] under the directory ...total_serial_v52\sender. They are also stored on the external Raspberry Pi (model name: Raspberry Pi5 PI-C 8G), which activates the sender scripts. All sender scripts can be started simultaneously by using the send_main.py script or individually, und der the following directories see table 29.

Table 29: Names and directories of the sender scripts, currently stored on the sender device (model name Raspberry Pi 5).

Sending Script Name	Directory
Send_Hioki_Data.py	...\total_serial_v52\sender
modbusTCP_server_elektrolyseur_03.py	
Canbus_send.py	...\total_serial_v52\sender\Modbus_and_Canbus_send\Einzeln
Modbus_send.py	

13.6.1 Send_Hioki_Data.py

This script is designed to read electrical measurement data from a CSV output file containing segments of previously recorded measurement data from the Hioki-PQ3198 Power Quality Analyzer. The script was implemented with the ulterior motive for causing additional network traffic on the MQTT-server to further feasibility checking but also in order to show how an external controller can interact as a client with MQTT over Ethernet.

13.6.2 Canbus_send.py

The script uses the python-can library, see (python-can, no date), to manage CAN communication over the socketcan interface. Similar to the receiving program, the script configures the CAN interface (*can0*) with a bitrate of 500 kbps, activates it, and creates a connection using the *can.interface* at initialization. A predefined list of CAN messages is used, each containing 8 data bytes and a shared arbitration ID. These messages simulate constant system behavior and can easily be adapted or randomized for more dynamic scenarios. The script also handles logging of each sent message into a CSV file for traceability and further analysis. A new log file is created either on first execution or when the previous file exceeds a defined maximum size (~11.9 MB or approx. 50,000 lines). Each CSV entry includes a timestamp, the original CAN message object, the decimal and binary representations of the data bytes, and an ASCII interpretation of the data, cleaned of special characters like semicolons, tabs, or commas to ensure CSV integrity. Message transmission follows a timing pattern, defined by *delta_t* (set to 50 milliseconds). The loop iterates through the message list in order and resets after each complete cycle.

13.6.3 Modbus_send.py

This Python script simulates the Modbus RTU communication, by making use of the serial library for UART communication. The serial interface is initialized on */dev/ttyAMA0* with a baud rate of 115200 bps. Upon execution, the script checks whether a CSV file needs to be created - either because the program has just started or because the previous file exceeded the defined size limit of approximately 11.9 MB (which equates to around 50,000 entries). If so, it generates a new file with a timestamped filename and initializes it with the appropriate header row.

The data transmission loop runs continuously, sending Modbus messages every 50 milliseconds. The data is sanitized by removing any problematic byte values (such as 0x3B, 0x2C, etc.) that could interfere with CSV formatting. Each sent message is logged in the corresponding CSV file with a human-readable timestamp and the decimal representation of the transmitted byte sequence. If the file grows too large.

13.6.4 modbusTCP_server_elektrolyseur_03.py

This Python script implements a functional Modbus TCP/IP server that simulates the behavior of one electrolyzer. Like the receiving script, see section 13.5.9, it is based on the pymodbus library (Pymodbus, 2023) and emulates both holding and input register spaces as specified by the Enapter Modbus communication specification, see (Enapter AG, 2025b).

As part of the startup routine, the script assigns the static IP address 192.168.0.10 to the Raspberry Pi's Ethernet interface and launches the Modbus TCP/IP server on port 5020. Initial values are written to the server's internal datastore to simulate default operating states. Moreover, a background thread continuously updates these register values to mimic realistic behavior. For example, when the "Start/Stop Electrolyser" register is activated, the server transitions to an active state and begins modifying values such as water level and electrolyte temperature accordingly. Input registers are regularly populated with random values that the receiver will receive on request.

13.7 Test and Validation: Practical Parts

13.7.1 Results from Test Run on the 29th of October 2024: Acquiring Data from Fuel Cell

The following results present the analysis and plausibility check of CAN bus output data from a fuel cell operating in standalone mode. The CAN bus output of the fuel cell was connected solely to the Raspberry Pi 1 via an RS485 CAN HAT. The only active script on the Raspberry Pi during testing was *CAN_BUS_01_loop.py*. Furthermore, the fuel cell was connected exclusively to ultracapacitors; no electrical connection to the local power grid was established. On the day of testing, two experiments were carried out: the first without hydrogen pressure at the fuel cell's hydrogen inlet, and the second with hydrogen supplied. Figure 28 and 29 illustrate the progression of the selected process parameters recorded during the test runs (excluding static parameters such as software version of error flags). Moreover, the tables 30 and 31 list the error flags that were observed over the CANbus. The total duration of the test run I was 3 minutes and 23.03 seconds. The test run II was active for a total of 3 minutes and 2.14 seconds.

Process Diagram for Test Run I

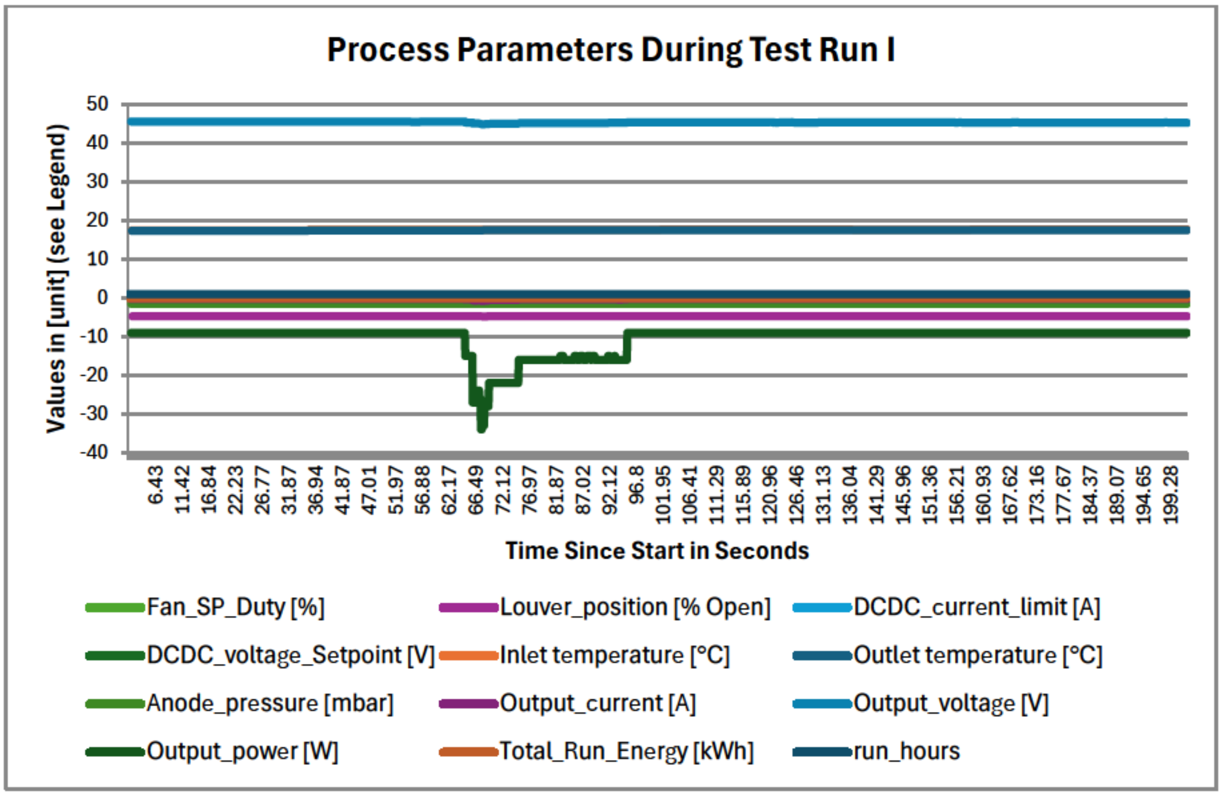


Figure 28: Progress of non-static process parameters during the test run I.

Acquired Error Flags from Test Run I

Table 30: Acquired error messages and their time durations during the test run I.

Flags	Begin of Flag (software_ timestamp & Time Since Start in Seconds)	Ending of Flag (software_ timestamp & Time Since Start in Seconds)	Tag	Description	Flag Level
A	14:41:37.348644 (72.03)	14:43:48.295594 (203.03)	SystemInletUnderPressure	System Inlet fuel pressure is below threshold when the Inlet Valve is open. Most likely causes are external to system but could also be an Inlet Valve issue. Formerly AnodeUnderPressure	4 (Critical)
B	14:41:37.348644 (72.03)	14:43:48.295594 (203.03)	LowLeakTestPressure	Low system inlet fuel pressure during leak test and unable to perform test. This is most likely to be a leak than a supply fault with Software versions higher than v1.158	4 (Critical)

Process Diagram for Test Run II

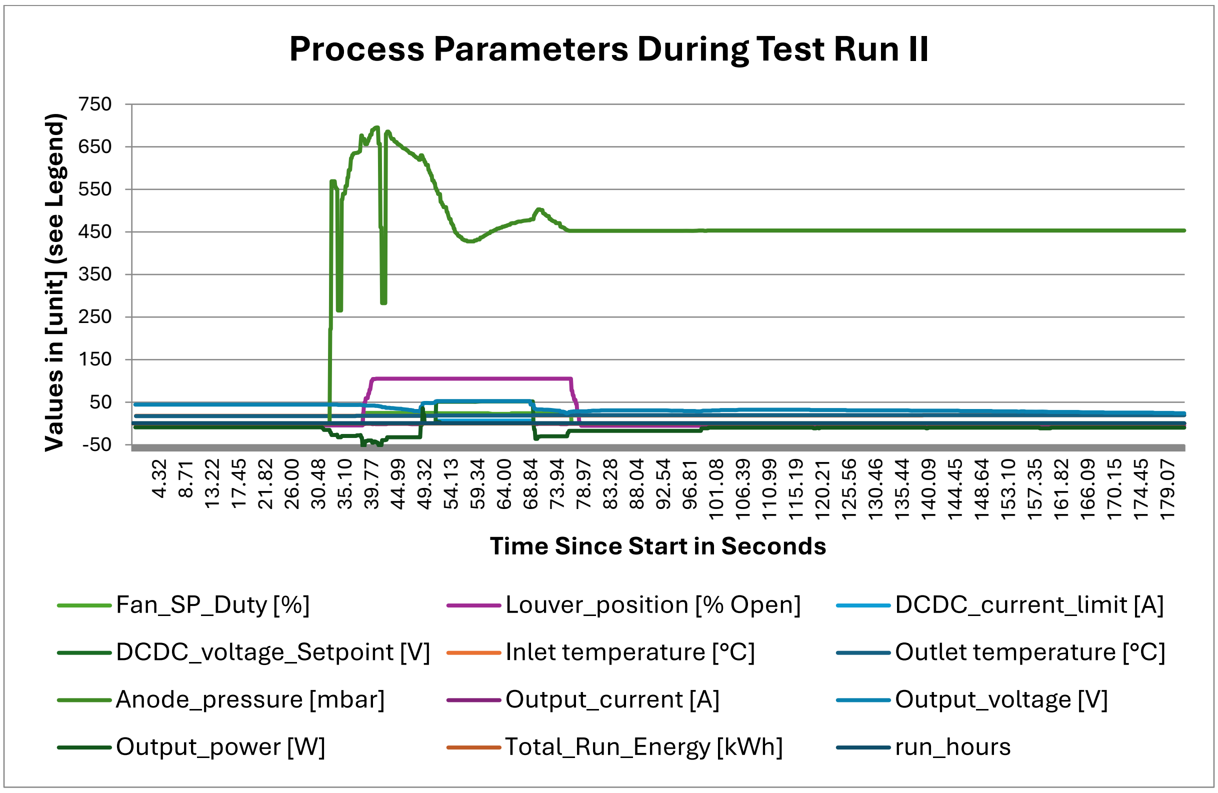


Figure 29: Progress of non-static process parameters during the test run II.

Acquired Error Flags Test Run II:

Table 31: Acquired error messages and their time durations during the test run II.

Flags	Begin of Flag (software_ timestamp & Time Since Start in Seconds)	Ending of Flag (software_ timestamp & Time Since Start in Seconds)	Tag	Description	Flag Level
B	27.09.2024 14:56:27.721645 (0.05)	27.09.2024 14:56:27.721645 (0.05)	DenyStartUV	External voltage below threshold specified in the configuration	2 (Controlled)

13.8 Results from Latency- and General Validation Tests in the Laboratory

13.8.1 General Methodology

Structured validation processes were carried out to ensure the functionality of each component at laboratory test runs. This included feasibility tests of individual processes, data logging, implementation of MQTT communication, data analysis in excel, as well as repeated debugging and standardization. Latency and message throughput served as the key metrics for validating system behavior throughout this process. In general, for the latency measurements it is aimed to achieve stable results with a significant performance buffer, since the system is designed for scalability. Other services may be added on the Raspberry Pi 1 in the future. Ultimately, the goal is to demonstrate how much unused processing capacity remains available.

The continuously conducted performance tests in the laboratory were carried out on the Raspberry Pi 1, using connected hardware: A second Raspberry Pi (model name: Raspberry Pi 5) was used to send dummy datasets via various interfaces, including CANbus, Modbus RTU, Modbus TCP/IP, and TCP/IP (see sending scripts).

Exemplary results of such latency and system performance tests are shown in the section below, hereby considered as “results from Test run III”. Figure 30 provides a visual impression of the laboratory configuration used during testing.

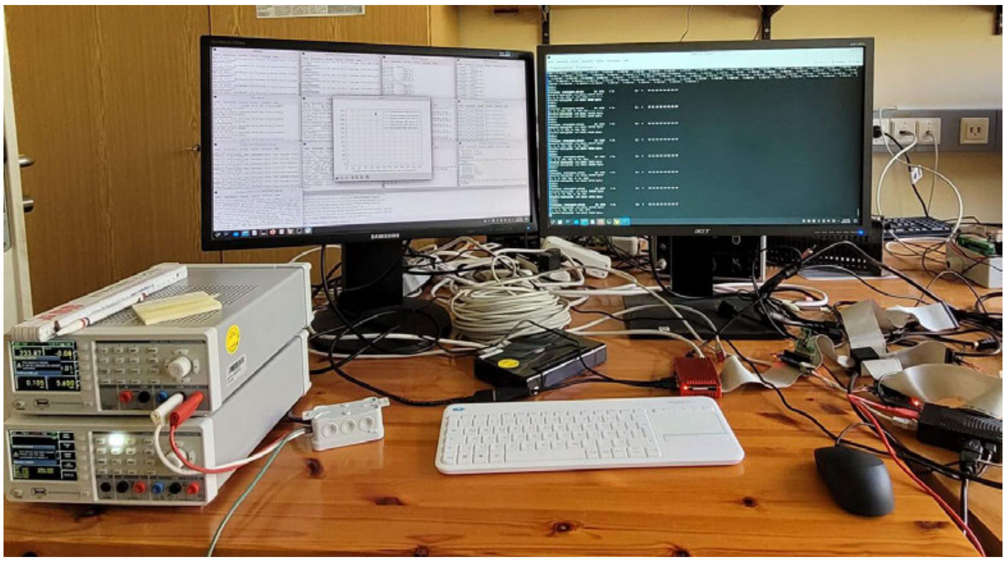


Figure 30: Photo illustrating a lab test. With main.py active on the receiver Raspberry Pi and send_main.py active on the transmitter Raspberry Pi (model name: Raspberry Pi 5).

13.8.2 Results from Test Run III on the 22nd of April 2025

Results of Data Analysis Regarding Interval Time of Logged Events in Internal .CSV files

The table 32 lists the results outgoing from a data evaluation based on the data acquired from all the program specific internal .csv files generated at the test run III. These files are present in the directory ...\total_serial_v52\CSV_for_received_data of [A_30]. The files used for the data analysis are provided in [A_27]. The column “Amount of values” indicates the total number of recorded software_timestamps entries in each respective .CSV file. Based on these timestamp values, the latency between consecutive logging cycles (or the “Publishing Interval Time”) of the respective loop programs was calculated via excel functions and can be compared to the calculated latency values of Message_Collector_for_csv_ loop.py, see table 33. The test run III was conducted from 09:36:17 AM to 10:47:37 AM thus took ca. 1 hour and 11 minutes. The amount of values (see first column) should roughly correspond 1/(*t_delay_for_publishing)* times the total test runtime in seconds. In general, a large deviation between the maximum publishing interval time and the average value indicates an error, because in such a case the respective program fell out of the secondary *while True* loop and had to run through the primary *while True* loop again.

The reason for such an error with the HMC8012 device is that a loss of connection to the MQTT server was observed at 10:17:46.914367 AM. The same problem was observed for SolaxHybrid_loop.py at 10:22:12.158144 AM (see corresponding .log files). For further testing it is recommended to set a higher *keepalive* value to prevent this issue: In the Paho library, the *keepalive* mechanism ensures the client sends a message or a PINGREQ packet to the broker within the specified interval; if no response (PINGRESP) is received in the timeframe specified by the chosen *keepalive* value, the connection is considered lost (also refer to section 2.11.4 and 8.3.6). Currently, the *keepalive* value is set to a relatively low value of 1 second.

In the Arduino_Communication_loop.py, checksum mismatches led to an increase in the maximum error value (see corresponding .log file). Since the errors occur sporadically, a problem with the physical connection is suspected.

Table 32: Data evaluation of all internal .CSV files of test run III.

Program	**Amount of Messages Published**	**Measured Publishing Interval Time**			
		Maximum	**Minimum**	**Average**	**Std Dev.**
CAN_BUS_01_loop.py	4141	1.785913	0.967434	1.029301	0.992208
Modbus_01_loop.py	3610	2.166090	0.002772	1.180070	0.460091
modbusTCP_elektrolyseur_ loop.py	2730	2.705883	0.993710	1.558943	1.086712
HMC8015_loop.py	3933	1.308459	0.050049	1.085832	0.844466
HMC8012_loop.py	4080	8.537046	0.967333	1.047044	1.007946
Arduino_Communication_loop.py	19816	6.631245	0.120317	0.214957	0.433606
get_resource_usage_loop.py	568	8.098721	6.133649	7.480064	2.530338
Message_Collector_for_csv_ loop.py	4140	1.406924	0.958463	1.033704	0.990268
Message_Collector_for_Data_ Check_loop.py	4114	1.221691	0.975999	1.031680	0.995835
Message_Collector_for_plot_ loop.py	2738	3.080886	0.997358	1.547972	1.062094
SolaxHybrid_loop.py	4139	7.284557	0.115262	1.027651	0.802156
EMS_loop.py	4172	1.939289	0.959831	1.023454	0.992812

Results of Message throughput and Latency Measurement through MQTT Data Transfer

The table 33 shows the evaluation of CSV File of the .csv file, outgoing from Message_Collector_for_csv_loop.py, meaning that the acquired time values are originating from the calculations of the *calculate_delta_t* function of the *mqtt_handler*. The standard deviation was calculated in succession via excel formulas.

Table 33: Results from the analysis of the .CSV file generated by Message_Collector_for_csv_loop.py during test run III.

Program	Amount of Messages Received	Maximum Latency	Minimum Latency	Average Latency	Std Dev. of Latencies
CAN_BUS_01_loop.py	4139	1.779705	0.884511	1.029079	0.032861
Modbus_01_loop.py	3605	1.482802	0.025200	1.179317	0.101179
modbusTCP_elektrolyseur_ loop.py	2728	2.716845	0.853303	1.559460	0.394273
HMC8015_loop.py	3933	1.313677	0.938390	1.085941	0.045340
HMC8012_loop.py	4078	9.551708	0.889188	1.047315	0.179745
Arduino_Communication_loop.py	19822	6.854612	0.092667	0.214784	0.243100
get_resource_usage_loop.py	568	8.128402	6.153832	7.480420	0.176392
Message_Collector_for_csv_ loop.py	4140	1.403908	0.915037	1.033501	0.034498
Message_Collector_for_Data_ Check_loop.py	4109	1.223081	0.911042	1.031480	0.033335
Message_Collector_for_plot_ loop.py	2736	3.118965	0.949793	1.547409	0.306803
SolaxHybrid_loop.py	4137	6.129103	0.863508	1.027447	0.085415
EMS_loop.py	4171	1.936249	0.918158	1.023247	0.031516

Deviations of Amount of Messages Published vs. Amount of Messages Received and Publishing Interval Time vs. Latencies through MQTT Data Transfer in Absolute Numbers

See caption above. The absolute deviations are shown in the table 34 below in absolute numbers and in table 35, the relative deviations are listed. In order to reach satisfying results, a low deviation regarding message transmission is expected, see row named "Amount". A negative deviation of amount means that messages were published, but some were no received. All in all, the deviations are low and thus the MQTT transmission lead to satisfactory results in this regard. Interestingly, a surplus of 6 messages were received, outgoing from Arduino_Communication_loop.py, even though the QoS was set to 1 for this category of data (so theoretically the deviation should be 0). But since the deviation is not negative, it can be considered satisfactory.

Table 34: Deviations of amount of messages published vs. amount of messages received and publishing interval time vs. Latencies through MQTT Data Transfer in absolute numbers.

Program		**CAN_BUS_01_ loop.py**	**Modbus_01_ loop.py**	**modbusTCP_ elektrolyseur_ loop.py**	**HMC8015_ loop.py**
Absolute Deviation between Values from Table 32 and 33	**Amount**	-2	-5	-2	0
	Maximum	-0.00621	-0.68329	0.010962	0.005218
	Minimum	-0.08292	0.022428	-0.14041	0.888341
	Average	-0.00022	-0.00075	0.000517	0.00011
	Std Dev.	-0.95935	-0.35891	-0.69244	-0.79913
Program		**HMC8012_loop.py**	**Arduino_ Communication_ loop.py**	**get_resource_ usage_loop.py**	**Message_ Collector_for_csv_ loop.py**
Absolute Deviation between Values from Table 32 and 33	**Amount**	-2	6	0	0
	Maximum	1.014662	0.223367	0.029681	-0.00302
	Minimum	-0.07815	-0.02765	0.020183	-0.04343
	Average	0.000271	-0.00017	0.000356	-0.0002
	Std Dev.	-0.8282	-0.19051	-2.35395	-0.95577
Program		**Message_ Collector_for_ Data_Check_ loop.py**	**Message_ Collector_for_ plot_loop.py**	**SolaxHybrid_ loop.py**	**EMS_loop.py**
Absolute Deviation between Values from Table 32 and 33	**Amount**	-5	-2	-2	-1
	Maximum	0.00139	0.038079	-1.15545	-0.00304
	Minimum	-0.06496	-0.04757	0.748246	-0.04167
	Average	-0.0002	-0.00056	-0.0002	-0.00021
	Std Dev.	-0.9625	-0.75529	-0.71674	-0.9613

Deviations of Amount of Messages Published vs. Amount of Messages Received and Publishing Interval Time vs. Latencies through MQTT Data Transfer in Relative Numbers

Table 35: Deviations of Amount of Messages Published vs. Amount of Messages Received and Publishing Interval Time vs. Latencies through MQTT Data Transfer in Relative Numbers.

Program		CAN_BUS_01_ loop.py	Modbus_01_ loop.py	modbusTCP_ elektrolyseur_ loop.py	HMC8015_ loop.py
Relative Deviation between Values from Table 32 and 33	Amount	-0.05%	-0.14%	-0.07%	0.00%
	Maximum	-0.35%	-31.54%	0.41%	0.40%
	Minimum	-8.57%	809.09%	-14.13%	1774.94%
	Average	-0.02%	-0.06%	0.03%	0.01%
	Std Dev.	-96.69%	-78.01%	-63.72%	-94.63%
Program		**HMC8012_loop.py**	**Arduino_ Communication_ loop.py**	**get_resource_ usage_loop.py**	**Message_ Collector_for_csv_ loop.py**
Relative Deviation between Values from Table 32 and 33	Amount	-0.05%	0.03%	0.00%	0.00%
	Maximum	11.89%	3.37%	0.37%	-0.21%
	Minimum	-8.08%	-22.98%	0.33%	-4.53%
	Average	0.03%	-0.08%	0.00%	-0.02%
	Std Dev.	-82.17%	-43.94%	-93.03%	-96.52%
Program		**Message_ Collector_for_ Data_Check_ loop.py**	**Message_ Collector_for_ plot_loop.py**	**SolaxHybrid_ loop.py**	**EMS_loop.py**
Relative Deviation between Values from Table 32 and 33	Amount	-0.12%	-0.07%	-0.05%	-0.02%
	Maximum	0.11%	1.24%	-15.86%	-0.16%
	Minimum	-6.66%	-4.77%	649.17%	-4.34%
	Average	-0.02%	-0.04%	-0.02%	-0.02%
	Std Dev.	-96.65%	-71.11%	-89.35%	-96.83%

13.9 Resource Data from Test Run III on the 22th of April 2025

This section presents the resource data recorded during test run III, see figures 31-35. The diagrams are based on a data analysis of the overall .CSV file generated by Message_Collector _for_csv_loop.py, see [A_27]. The respective data series were generated by the program get_resource_usage_loop.py and sent via MQTT. This chapter is for documentation purposes only and leaves room for further critical evaluation of the results. It would also be useful to extend get_resource_usage_loop.py to include additional measurement parameters if necessary. See further possible improvements in section 13.5.7.

13.9.1 CPU Usage

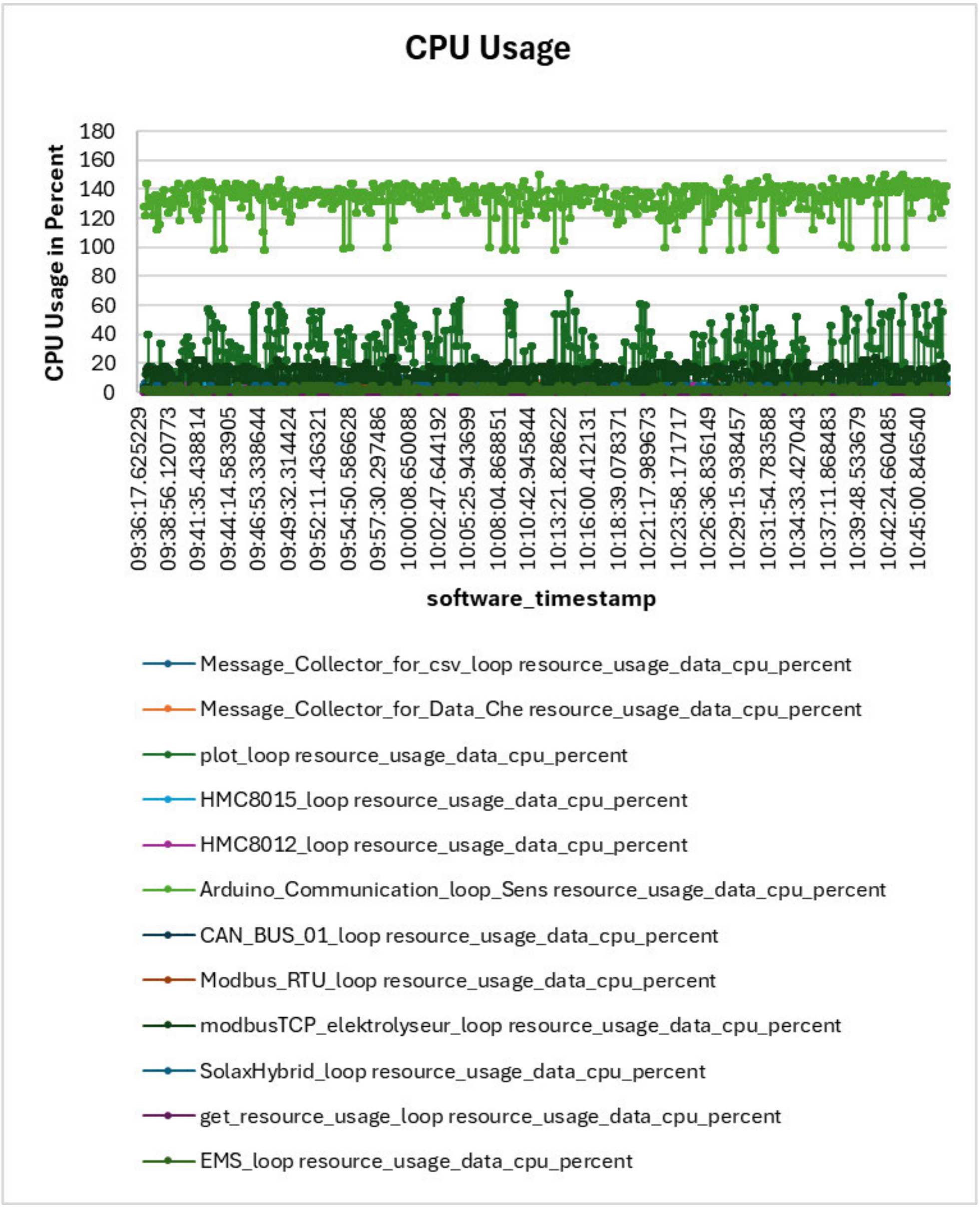


*Figure 31: The acquired values for the CPU usage of the respective *_loop.py programs during test run III.*

13.9.2 Memory usage

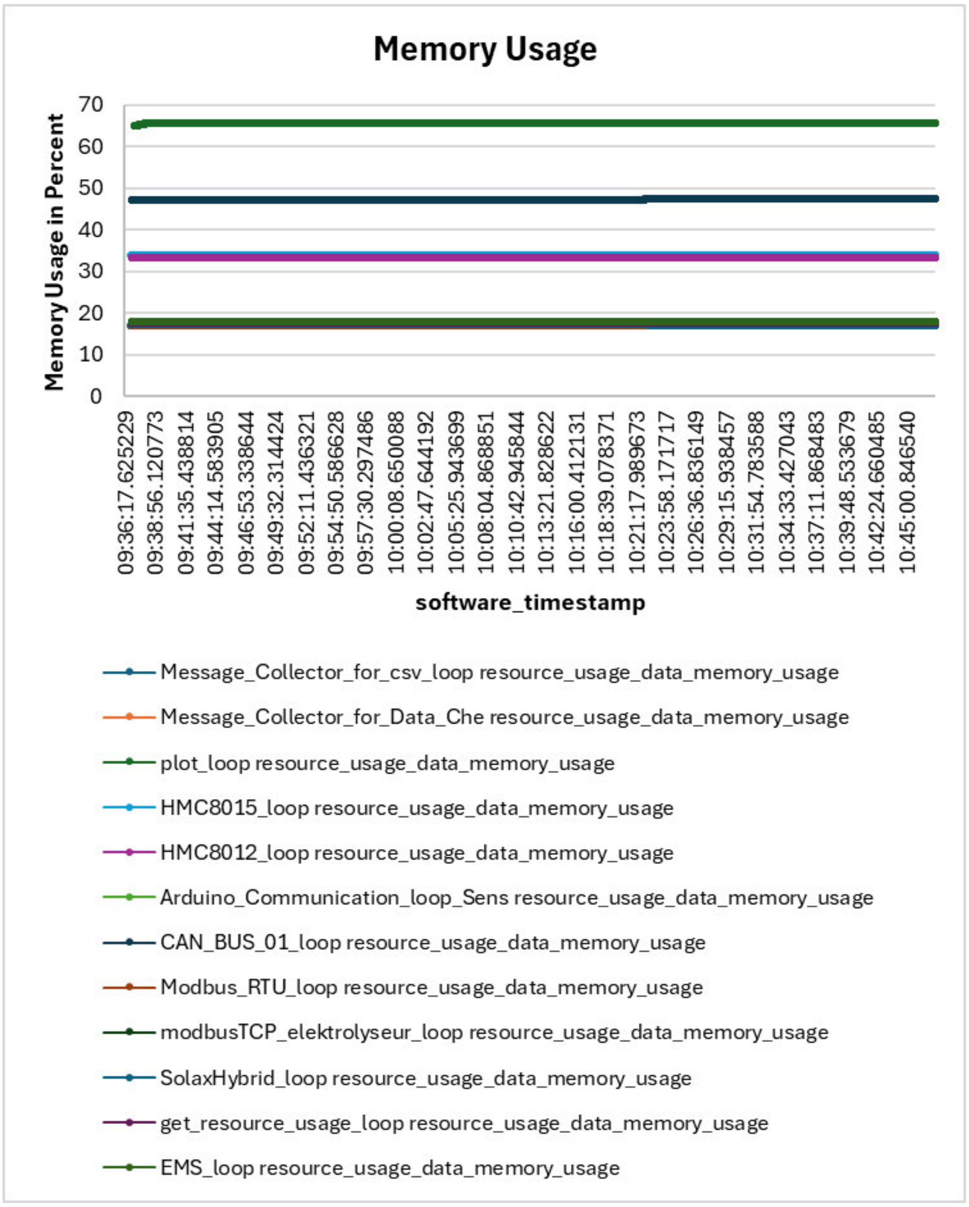


*Figure 32: The acquired values for the memory usage of the respective *_loop.py programs during test run III.*

13.9.3 RAM usage

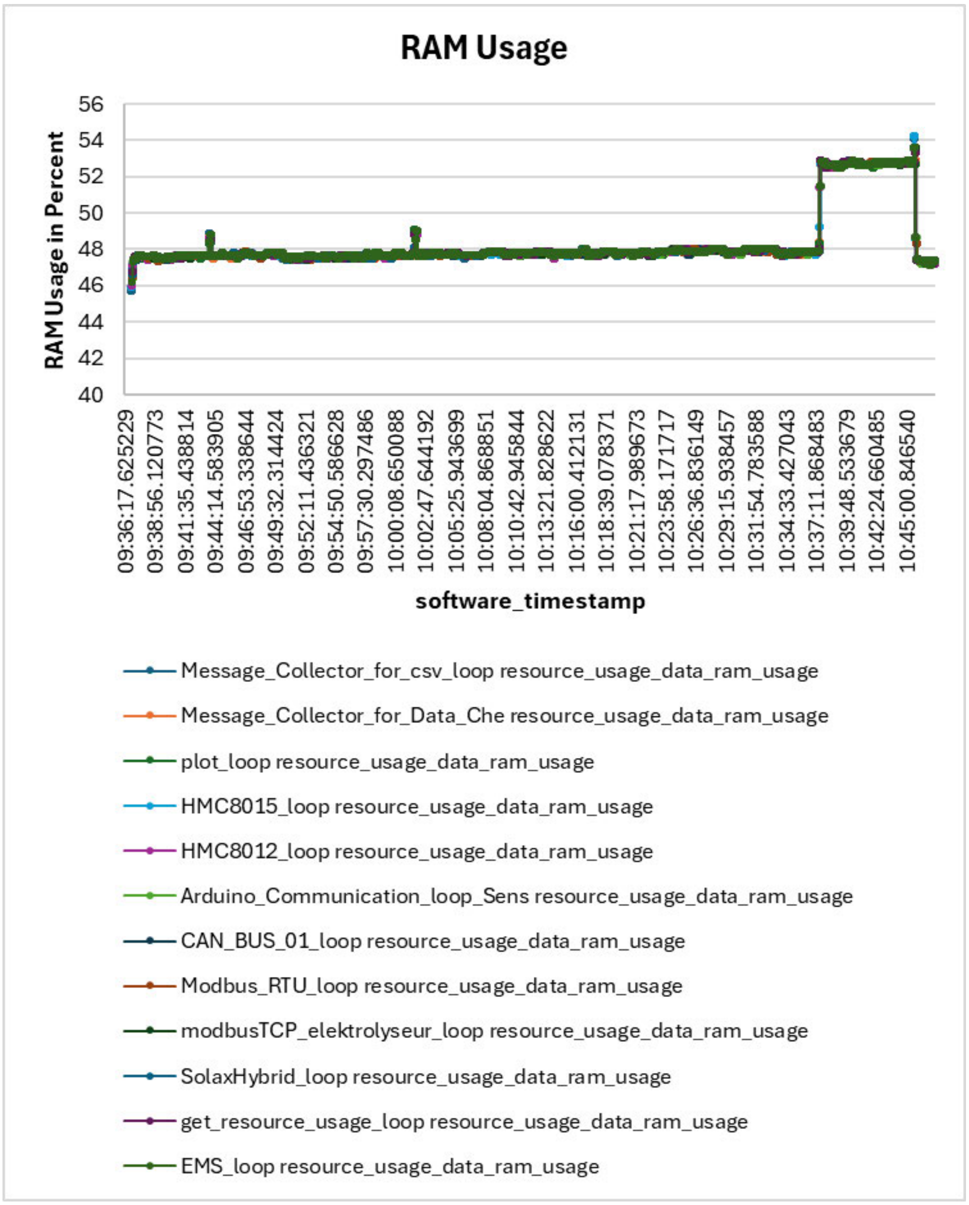


*Figure 33: The acquired values for the RAM usage of the respective *_loop.py programs during test run III.*

13.9.4 Swap usage

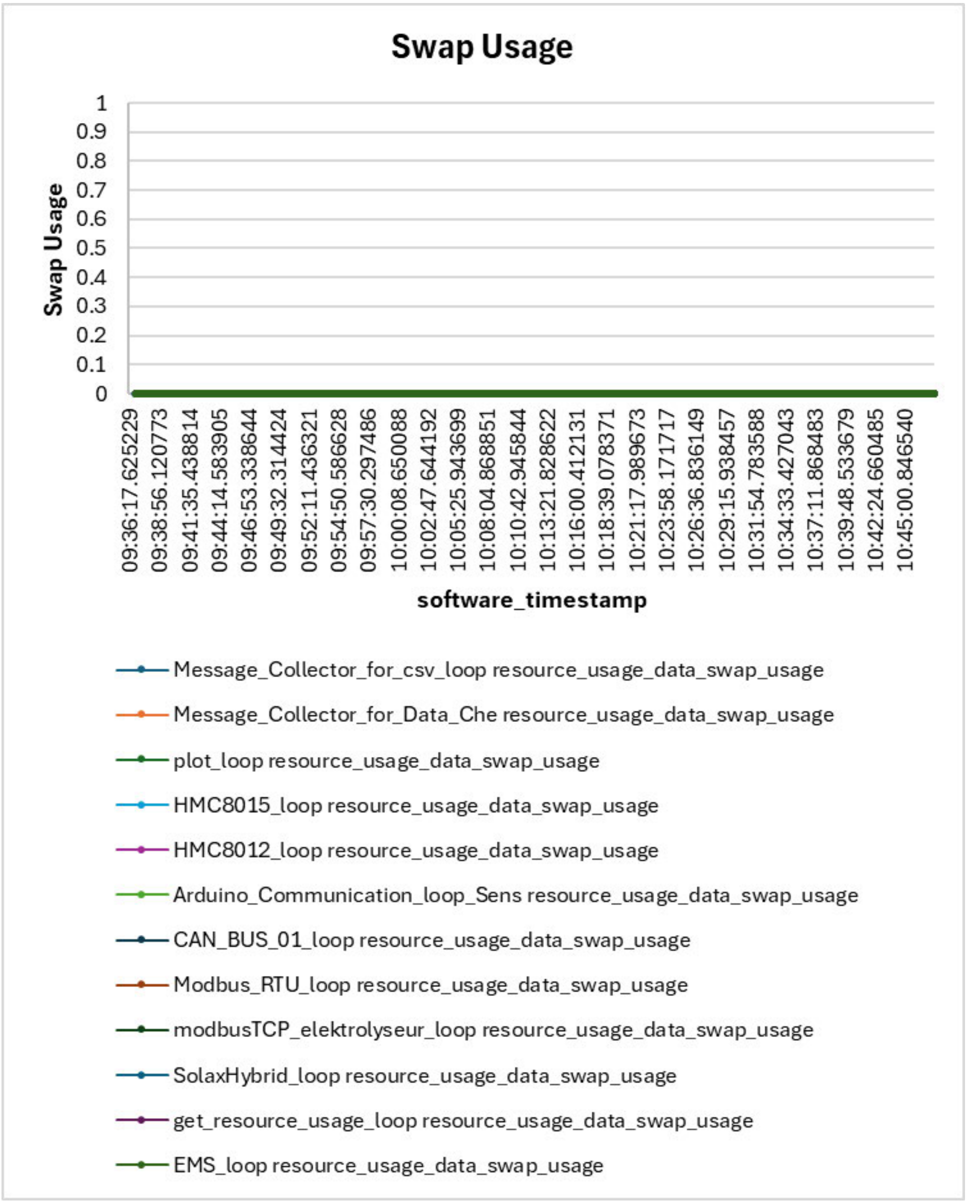


*Figure 34: The acquired values for the swap usage of the respective *_loop.py programs during test run III.*

13.9.5 Load Average

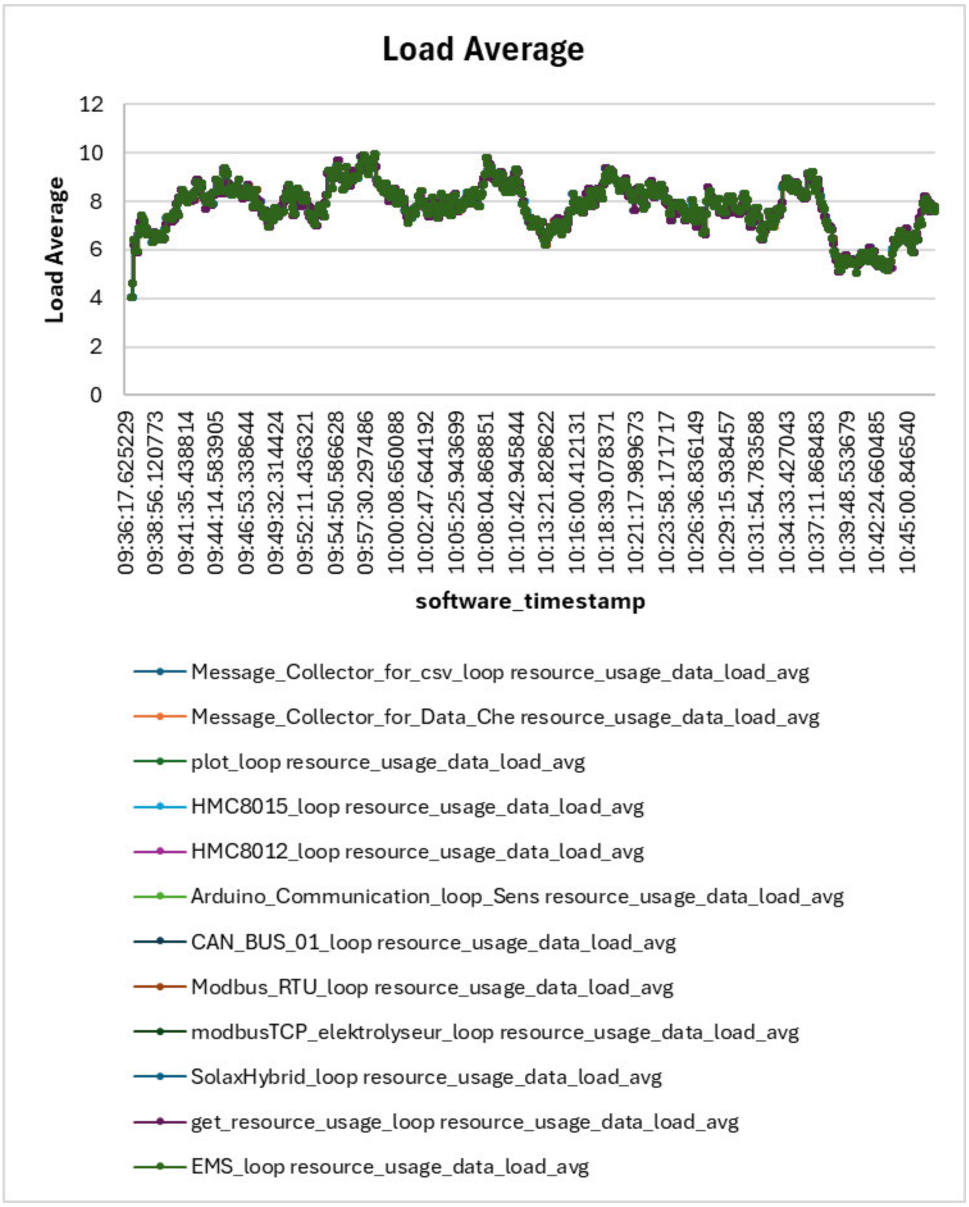


*Figure 35: The acquired values for the load average of the respective *_loop.py programs during test run III.*

13.10 Simulation of the EMS Algorithm via Excel

13.10.1 Overview of the Simulation File

The excel file visualizes the energy flows and the evolution of the states of charge (SOCs) in relation to the proposed EMS algorithm (see section 8.5). It is available in [A_29]. Furthermore, it has to be noticed that the file takes up a longer time to load (30 seconds to one minute). Also, when entering a certain table, the diagrams take up a long time to load (also ca. 30 seconds to one minute).

13.10.2 Simulation Framework and Boundary Conditions

The simulation is based on an islanded grid scenario, meaning that all energy generated by the PV modules or the fuel cell must be consumed directly: either by the load, the battery, the electrolyzers, or a combination of these elements. As no specific PV capacity has been defined at this stage for the La Luz site, the file includes three different simulation scenarios reflecting varying levels of photovoltaic generation: *Max_PV* (first sheet), *Mittel_PV* (second sheet), and *Min_PV* (third sheet). In each of these simulations, the PV output is randomly throttled to emulate fluctuations due to cloud cover. The energy consumption data is based on smoothed, randomized values and totals approximately 23.2 kWh per day, as specified in chapter 3. For the sake of simplicity, the load profile does not reflect realistic day-night patterns such as those observed at the La Luz site, but instead remains relatively evenly distributed. The simulation spans a period of three days, during which both PV generation and consumption patterns repeat cyclically. Based on these parameters, the energy flows, such as fuel cell output, battery charging and discharging, electrolyzer consumption, and the resulting SOCs are calculated dynamically for each time step.

13.10.3 VBA Code Availability

For reference and transparency, the VBA code used for the simulation has been outsourced to a separate text file, also included in directory [A_29].

13.10.4 Visualization of Results

The figures 36 and 37 present the visual representations of the simulation results. The figures shown here illustrate the output from a complete simulation run, based on one of the predefined PV production scenarios from the sheet *Mittel_PV*. As it can be observed, this simulation shows the impact, on the activation of the fuel cell (grey line color), as well as the electrolyzers (pink line color).

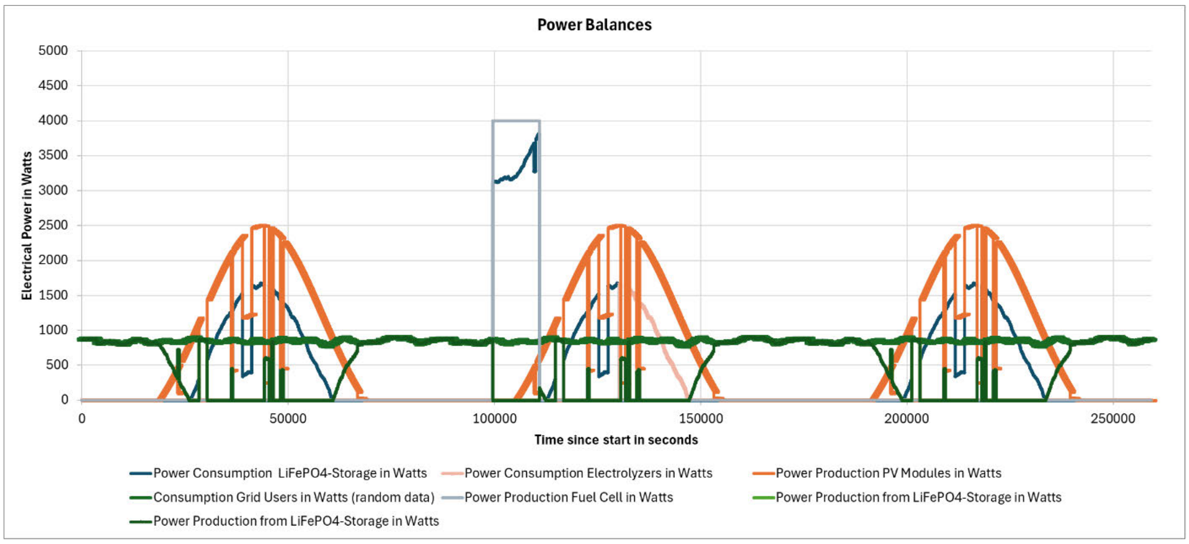


Figure 36: Diagram showing the course of power flows in watts from the different entities after a conducted simulation (see Mittel_PV-sheet).

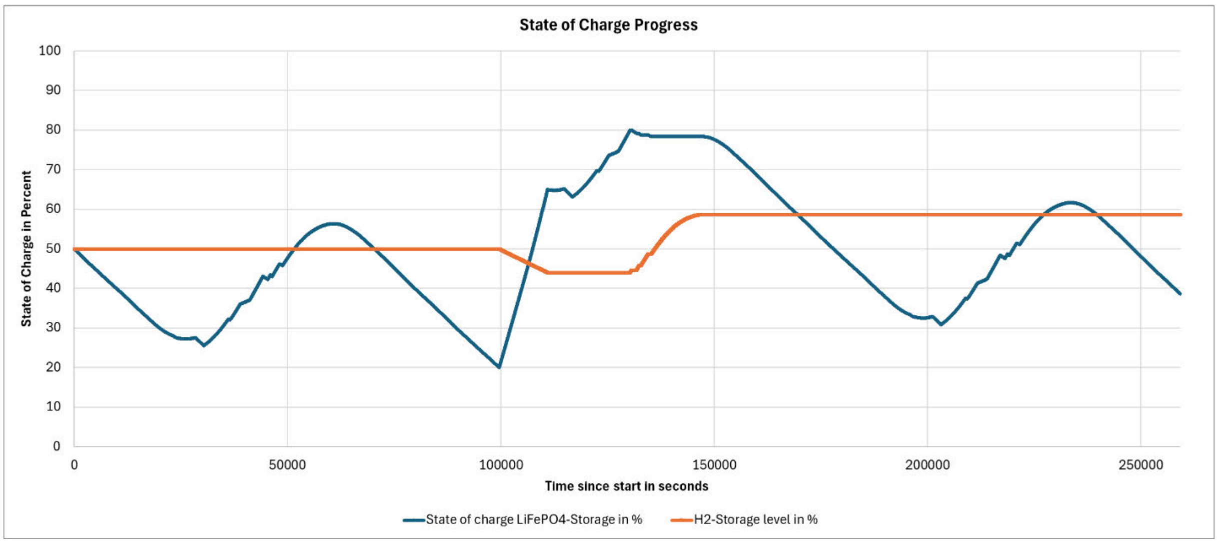


Figure 37: Diagram showing the course of the battery's SOC and the storage level of hydrogen in percent after a conducted simulation (see Mittel_PV-sheet).

13.10.5 Explanation of the VBA Code

The simulation was conducted by a VBA macro, structured as a loop that processes one row of Excel data per iteration, representing a single time step (one second of the day). The macro starts by clearing previous simulation results from specific columns in the worksheet. The simulation proceeds in a loop that reads the prescribed PV production and grid consumption values from the spreadsheet and calculates the resulting power surplus or deficit (*Delta*).

The energy management algorithm of the simulation follows the logic of chapter 8.5: If the system detects a surplus and both the battery and hydrogen storage levels are full, the macro reduces PV input to avoid overproduction. Otherwise, the surplus or deficit is handled based on system conditions. When the battery state-of-charge falls below the 20 percent threshold, the fuel cell is activated to compensate for the energy deficit, consuming hydrogen and increasing power production. Conversely, when the battery SOC is high and excess PV production is available, the electrolyzers are activated to store energy in the form of hydrogen, relative to the current *Delta* value. The hydrogen production coming from the electrolyzers is calculated based on power input and the specific energy requirement per cubic meter of hydrogen, using a linear model derived from datasheet specifications.

Throughout the simulation, energy flows are updated for each component, and new SOC and hydrogen levels are calculated. These values are written to the corresponding cells in the spreadsheet for each time step. The simulation also computes total system production and consumption per iteration and records the net energy balance. To assist in debugging and progress monitoring, the macro prints the current row index to the console and updates the Excel status bar every 1000 iterations and the simulation continues until the last row of input data is reached.

13.10.6 Simplifications

The macro includes assumptions and simplifications, such as:

- The physical constraints of the hydrogen storage system are modeled using the ideal gas law, taking into account parameters such as pressure, volume, temperature, and molar mass of hydrogen
- Instantaneous response from electrolysis and fuel cell systems
- Linear efficiency models, and the exclusion of conversion losses (e.g., from inverters or DC/DC converters)
- Maximum input/output power limits regarding the battery are not explicitly enforced
- POC of the fuel cell is omitted
- The gradual activation of the electrolysis units has not been considered

13.10.7 Application

The simulation macro can be triggered in each of the provided workbooks via the button located in cell AA2. During execution, the macro displays the current row number in the Excel status bar (bottom left) to provide real-time progress feedback. The simulation runs until row 259,201 is reached. But naturally, macros have to be enabled at first.