Technische Hochschule Wildau
Technical University of Applied Sciences
WILDAU

Bachelorarbeit

zur Erlangung des akademischen Grades
Bachelor

Technische Hochschule Wildau

Fachbereich Ingenieur- und Naturwissenschaften

Studiengang Telematik (B. Eng.)

Thema (deutsch):	Entwicklung einer DevBox zur Unterstützung der prototypischen Entwicklung von hardwarebasierten Komponenten für den Tischroboter ROS-E.
Thema (englisch):	Development of a DevBox to support the prototypical development of hardware-based components for the table robot ROS-E.

Autor/in:	Oskar Lorenz
Seminargruppe:	T/19
Betreuer/in:	Prof. Dr. rer. nat. Janett Mohnke
Zweitgutachter/in:	M. Eng. Janine Breßler
Spätestmögliche Abgabe:	17.09.2022
Eingereicht am:	

Verlängerung des Bearbeitungszeitraumes

Auf Grund der Antragstellung des Studierenden wurde der Bearbeitungszeitraum für die Erstellung der Arbeit um 6 Woche(n) verlängert.

Das Abgabedatum nach Antragstellung ist der 17.09.2022.

Technische Hochschule Wildau
Fachbereich Ingenieur- und Naturwissenschaften

Technische Hochschule Wildau Technical University of Applied Sciences
WILDAU

Bachelorarbeit

Antrag vom: 04.04.2022

Name:	Oskar Lorenz	Matrikel-Nr.:	50077964
Studiengang:	Telematik	Seminargruppe:	T/19
		Beginn der Arbeit:	18.04.2022
Betreuende/r Hochschuldozent/in:	Prof. Dr. rer. nat. Janett Mohnke	Abgabetermin:	06.08.2022
Zweitgutachter/in:	M. Eng. Janine Breßler	Fachliche Betreuungsperson des Themenstellers:	Valentin Schröter
Themensteller (z.B. Betrieb, Institution, Behörde):	Technische Hochschule Wildau, RoboticLab Telematik	Straße:	Hochschulring 1
		PLZ, Ort:	15745, Wildau

Kurzthema

Entwicklung einer DevBox zur Unterstützung der prototypischen Entwicklung von hardwarebasierten Komponenten für den Tischroboter ROS-E.

Kurzthema in Englisch

Development of a DevBox to support the prototypical development of hardware-based components for the table robot ROS-E.

Zielstellung

Konzeption und Implementierung eines Prototypingboards für eine einfache, flexible, prototypische Entwicklung von Hardwarekomponenten für den Tischroboter ROS-E des RoboticLab Telematik.

Inhaltliche Anforderungen / Teilaufgaben

- Analyse der Anforderungen an die DevBox
- Entwicklung eines Konzepts für Hardware & Software der DevBox
- Herstellen eines Prototypen der DevBox
- Bau der DevBox
- Erstellen einer Dokumentation für Benutzer

Sprache der Arbeit

Deutsch

Konsultationen erfolgen nach Vereinbarung mit dem betreuenden Hochschullehrer.

Prof. Dr. rer. nat. Janett Mohnke	M. Eng. Janine Breßler	Oskar Lorenz	*genehmigt*
(Hochschuldozent/in)	(Zweitgutachter/in)	(Student/in)	(Prüfungsausschuss)

Eingang der Abschlussarbeit ________________

Sperrvermerk ☐ Ja ☒ Nein

Bibliographische Beschreibung

Lorenz, Oskar

Entwicklung einer DevBox zur Unterstützung der prototypischen Entwicklung von hardwarebasierten Komponenten für den Tischroboter ROS-E.

Bachelorarbeit, Technische Hochschule Wildau 2022, 82 Seiten, 34 Abbildungen, 12 Tabellen; enthält 3 Anhänge und eine beigelegte SD-Karte.

Ziel

Das Ziel dieser Bachelorarbeit ist ein Prototyping-Board, welches als Erweiterung des Tischroboters „ROS-E" die flexible Anbindung verschiedener Hardwarekomponenten wie Aktoren und Sensoren sowie von *Smart Home* Geräten ermöglichen soll. Hierfür soll eine Kombination aus Hardware und Software entwickelt werden, die die Weiterentwicklung von ROS-E vereinfacht und beschleunigt.

Inhalt

Die Bachelorarbeit beschreibt die Konzeption und prototypische Umsetzung eines Entwicklerboards („DevBox") für den Roboter ROS-E, um die Weiterentwicklung am Roboter zu vereinfachen und zu beschleunigen.

Dafür werden, nach einer Einleitung zum Thema, der Tischroboter „ROS-E" und mit diesem verbundene Entwicklungsprozesse näher erläutert. Anschließend werden Anforderungen an die DevBox nach Design, Hardware und Software analysiert. Mit diesen Anforderungen werden eine Systemarchitektur sowie Konzepte für die Hardware und Software der DevBox entwickelt. Als nächstes wird die Implementierung des Konzepts anhand eines Hardwareprototypen erläutert. Zum Schluss werden durchgeführte Tests erläutert und dessen Ergebnisse besprochen.

Dieser Bachelorarbeit ist eine SD-Karte beigelegt, die eine digitale Version der Arbeit enthält. Im Anhang dieser Arbeit befinden sich Verweise auf den entwickelten Quellcode sowie Modelle und Leiterplattendesigns für die DevBox.

Abstract

In dieser Bachelorarbeit wurde ein Entwicklerboard – die „DevBox“ – als Erweiterung des Tischroboters „ROS-E“ konzipiert, um die Erprobung und Entwicklung neuer Hardware des Roboters zu vereinfachen und zu beschleunigen. Zudem soll die DevBox den Zugang zu ROS-E für Neueinsteiger und für die Lehre erleichtern.

Es wurden diverse Anforderungen an die DevBox bezüglich der Hardware und Software auf Grundlage von Anwendungsfällen der Entwickler von ROS-E aufgestellt. Das resultierende Konzept umfasst ein Modell für den Bau einer DevBox, welches auch eigens entwickelte Leiterplatten für eine bessere Ergonomie bei der Entwicklung enthält. Die konzipierte Software der DevBox beschreibt Lösungen für die Anbindung diverser Hardwareschnittstellen wie GPIO, I²C und SPI und von Smart Home Geräten via EnOcean, ZigBee und Z-Wave über die DevBox und wie diese mit dem Roboter ROS-E verwendet werden können.

Anhand eines ersten Hardwareprototypen der DevBox konnten grundlegende Funktionen der Software implementiert und erfolgreich getestet werden. In weiteren Schritten soll die DevBox gebaut und für die Weiterentwicklung von ROS-E eingesetzt und finalisiert werden.

This thesis presents a development board – the “DevBox” – as an extension to the table robot “ROS-E” to support the evaluation and development of new hardware for ROS-E. It intends to both simplify and accelerate further development of ROS-E as well as to facilitate access to ROS-E for beginners and education.

The concept for the DevBox, based on requirements established by analyzing use cases of ROS-E developers, encompasses a hardware model with custom circuit board designs for improved ergonomics during development. The accompanying software provides solutions for integrating various hardware interfaces like GPIO, I2C and SPI as well as smart home devices via EnOcean, ZigBee or Z-Wave into the DevBox and using them with the robot ROS-E.

By using a hardware prototype of the DevBox, fundamental software features were implemented and successfully tested. Further steps include finalizing and building a working DevBox and deploying it in the ROS-E development process.

Inhaltsverzeichnis

Hinweise zum Lesen der Arbeit IV

1 Einleitung 1

1.1 Motivation 1

1.2 Ziele der Arbeit 2

1.3 Zielgruppe und Abgrenzung 3

1.4 Aufbau der Arbeit 3

2 Roboter „ROS-E“ 5

2.1 Aufbau & Hardware 5

2.2 Software 6

2.3 Entwicklungsprozesse 6

2.4 Bestehende Technologien 7

3 Verwendete Technologien 8

3.1 Robot Operating System 8

3.2 Message Queuing Telemetry Transport 9

3.3 Hardwareschnittstellen 10

3.3.1 General Purpose Input/Output 10

3.3.2 Inter-Integrated Circuit 10

3.3.3 Serial Peripheral Interface 11

3.4 Smart Home Protokolle 12

3.4.1 EnOcean 12

3.4.2 ZigBee 13

3.4.3 Z-Wave 13

4 Analyse der Anforderungen an die DevBox 14

4.1 Anwendungsfälle 14

4.2 Definition der Anforderungen 15

4.2.1 Anforderungen an das Design 15

4.2.2 Anforderungen an die Hardware 15

4.2.3 Anforderungen an die Software ... 16
4.3 Tabellarische Übersicht der Anforderungen ... 17

5 Architektur- & Konzeptentwicklung ... 22

5.1 Systemarchitektur ... 22
5.1.1 Verbindung zu ROS-E ... 23
5.1.2 Verbindung zu Smart Home Geräten ... 24
5.1.3 Verbindung zu Aktoren & Sensoren ... 25
5.1.4 Verbindung zu Microcontrollern ... 26
5.1.5 Verbindung zu anderen Geräten ... 27
5.2 Hardware der DevBox ... 27
5.2.1 Auswahl & Zusammenhang der Hardwarekomponenten ... 28
5.2.2 Aufbau & Design der DevBox ... 29
5.2.3 Zusammenfassung der Materialien & Kosten ... 32
5.3 Software für die DevBox ... 33
5.3.1 Kommunikation mit Hardwarekomponenten ... 33
5.3.2 Kommunikation mit Smart Home Geräten ... 37
5.3.3 Übergang von der DevBox zu ROS-E ... 39
5.3.4 Adressierung und Erkundung von DevBoxen ... 40
5.3.5 Programmierschnittstelle für die DevBox ... 44
5.3.6 Betriebssystem & weitere Software ... 44

6 Prototypische Umsetzung des Konzepts ... 45

6.1 Hardwareprototyp ... 45
6.2 Software auf der DevBox ... 46
6.2.1 General Purpose Input/Output ... 46
6.2.2 Inter-Integrated Circuit ... 47
6.2.3 Serial Peripheral Interface ... 48
6.2.4 Message Queuing Telemetry Transport ... 48
6.2.5 EnOcean, ZigBee & Z-Wave ... 48
6.2.6 Adressierung & Erkundung ... 49
6.3 Software für Verwender ... 51
6.3.1 Basisklassen ... 52

6.3.2 Asynchrone Ausführung 53
6.3.3 Ergänzungen zur MqttClient-Klasse 53
6.3.4 „Hello World“-Programme & Dokumentation 54

7 Tests & Ergebnisse 55

7.1 Aufbau & Testfälle 55
7.1.1 Verbindung zwischen der DevBox und ROS-E 55
7.1.2 Adressierung & Erkundung der DevBox 55
7.1.3 Anbindung von Hardwarekomponenten 55
7.1.4 Anbindung von Smart Home Geräten 56

7.2 Auswertung der Ergebnisse 56

8 Fazit 57

8.1 Zusammenfassung 57

8.2 Ausblick 57

9 Anhang 59

9.1 Materialliste für den Bau einer DevBox 59

9.2 Digitale Anhänge 60
9.2.1 Software & Quellcode 60
9.2.2 Modelle & Leiterplattendesigns 60

Abkürzungen V

Glossar VI

Abbildungen VIII

Tabellen X

Quellen XI

Selbstständigkeitserklärung XVII

Hinweise zum Lesen der Arbeit

Die folgenden Hinweise sollen beim Lesen der Arbeit unterstützen. Eine Übersicht der verwendeten Abkürzungen und das Glossar finden sich am Ende dieser Arbeit.

Abkürzungen und im Glossar erläuterte Begriffe werden bei jeder Erscheinung im Text kursiv geschrieben. Verweise auf externe Ressourcen und ergänzende Worte folgen als numerische Fußnoten am jeweiligen Ende einer Seite.

Auf verwendete Literaturquellen wird mit einer numerischen Bezeichnung nach dem *IEEE*-Zitierstil verwiesen. Bei einem direkten Zitat wird unmittelbar nach dem zitierten Text auf die Quelle verwiesen. Bei einem indirekten oder sinngemäßen Zitat wird am Ende des entsprechenden Absatzes auf die Quelle verwiesen. Auf verwendete Quellen für Abbildungen und Tabellen wird in eckigen Klammern der zugehörigen Beschriftung verwiesen.

Auszüge aus Quellcode oder Code-Beispiele werden in Festbreitenschrift dargestellt. Ein ganzer Codeblock (nicht in einer Textzeile stehend) wird leicht grau hinterlegt.

1 Einleitung

Die große Erfindung beginnt mit dem kleinen Versuch.

Dr. phil. Manfred Hinrich

Mit dem Tischroboter „ROS-E" – ein eigenentwickelter, humanoider Roboter – wurde eine „[…] Plattform für zukünftige Entwicklungs- und Forschungsprojekte im Bereich Ambient [Assisted] Living" [1] geschaffen. ROS-E (ausgesprochen „Rosie") wurde im Rahmen der Bachelorarbeit von Valentin Schröter im Jahr 2019 entwickelt. Inzwischen besteht das Team rund um den Roboter ROS-E aus vier Kernentwicklern und wird als Projekt des RobotikLab Telematik an der Technischen Hochschule Wildau von Prof. Dr. Janett Mohnke geleitet. [2]

Neben der Arbeit des Kernteams von ROS-E fließen auch Ergebnisse aus Praktika von Studierenden des Telematik Studiengangs oder aus Firmen und Einrichtungen, die mit dem RobotikLab zusammenarbeiten, ein. Daraus entstehen viele Ideen und Konzepte, die die ständige Weiterentwicklung von ROS-E prägen. Diese Ideen wollen natürlich ausprobiert, erforscht und getestet werden, um einerseits festzustellen, ob eine Integration in das Robotersystem sinnvoll wäre, und um andererseits neue Erkenntnisse zu gewinnen.

1.1 Motivation

Damit diese vielen Ideen schnell und einfach getestet sowie verschiedene Konzepte und Überlegungen ausprobiert werden können, fehlt es zurzeit an einer einfachen Möglichkeit, diverse Hardwarekomponenten mit dem System von ROS-E einfach und schnell zu verbinden und zu nutzen. Doch bei einem Roboter wie ROS-E, bei dem viele Funktionen durch eine Vielzahl von Sensoren und Aktoren ermöglicht werden, ist die Möglichkeit der schnellen und ständigen Entwicklung solcher Komponenten essenziell.

Zurzeit bedeutet das Evaluieren von zusätzlicher Hardware für ROS-E oft, dass diese Hardware entweder separat und unabhängig von ROS-E getestet werden muss oder eine neue Hardwareversion von ROS-E entwickelt werden muss, die die neuen Hardwarekomponenten aufgreift. Beide Vorgehensweisen stellen jedoch Hürden in der Entwicklung von Hardware dar, die Zeit oder im Fall einer neuen Hardwareversion auch Geld kosten und den Entwicklungsprozess ausbremsen. Dabei kann gerade das viele

Ausprobieren von verschiedenen Komponenten Freude beim Entwickeln bereiten und zum Ausprobieren neuer Ideen motivieren.

Eine große Rolle im Zusammenhang mit *Ambient Assisted Living* (*AAL*) spielt das Thema *Smart Home* und die Gebäudeautomation. ROS-E soll als ein Roboter in diesem Bereich Menschen bei Ihrem tagtäglichen Leben unterstützen. Dazu gehört auch, das Licht, die Türen oder andere Geräte in einem Haus zu steuern oder auf Betätigung von Schaltern, Rauchmeldern oder andere Sensoren des Hauses zu reagieren. Damit ROS-E als eine natürliche Schnittstelle zwischen Menschen und ihrem Zuhause agieren kann, darf die Anbindung an *Smart Home* Geräte nicht fehlen. Zu diesem Thema wurden zwar Vorbetrachtungen zu verschiedenen Funkstandards angestellt, jedoch steht die Integration in die Hardware von ROS-E noch aus. [1] [3]

Ein weiterer Leitgedanke von ROS-E ist die Möglichkeit zur Verwendung in Forschungsprojekten, insbesondere bei Praktika und Abschlussarbeiten, als Ablösung des Roboter NAO im Studiengang Telematik an der *TH* Wildau. Diverse Praktika mit ROS-E haben sich vor allem mit Software oder externer Hardware beschäftigt. Die interne Hardware von ROS-E wurde hingegen kaum verändert. Eine Möglichkeit neue Hardware im Zusammenhang mit ROS-E einfach zu entwickeln ist somit durchaus erwünscht.

Nicht nur die Entwickler von ROS-E oder das RobotikLab können sich für ROS-E begeistern. Dies wird besonders durch den Erhalt des Publikumspreises und der Belegung des ersten Platzes beim Innofab_ Ideenwettbewerb[1] mit dem Projekt ROS-E im Juni 2022 deutlich. Damit zeigt sich, dass die Weiterentwicklung von ROS-E nicht das alleinige Interesse des RobotikLab ist und in Zukunft weiterhin vorangetrieben wird. Die flexible und einfache Entwicklung an und für ROS-E ist demnach sehr wichtig. [4]

1.2 Ziele der Arbeit

Aus den bisherigen Beschreibungen geht hervor, dass ein ROS-E Roboter allein nicht ideal für die Entwicklung von zusätzlichen Aktoren oder Sensoren ist. Das Ziel dieser Arbeit ist daher die Konzipierung und Entwicklung eines Entwicklerboards, an das flexibel diverse Hardwarekomponenten angeschlossen und mit dem System von ROS-E verwendet werden können. Zusätzlich zu kabelgebunden und direkt in ROS-E integrierbaren Hardwarekomponenten soll das Entwicklerboard auch die Möglichkeit der Anbindung von *Smart Home* Geräten über geeignete Funkstandards ermöglichen.

Dieses Entwicklerboard – im Folgenden als „DevBox“ bezeichnet – soll die Entwicklung von neuer Hardware, die Weiterentwicklung von bestehenden Komponenten und die Anbindung neuer Geräte beschleunigen und vereinfachen. Daher muss ein Ziel für die

[1] https://innohub13.de/innofab

DevBox auch die einfache Bedienung und Verwendung mit ROS-E sein. In Vorüberlegungen zu dieser Arbeit wurde bereits die grobe Architektur der DevBox in Verbindung mit einer ROS-E angedacht. Die folgende Abbildung 1 visualisiert diese Vorüberlegungen.

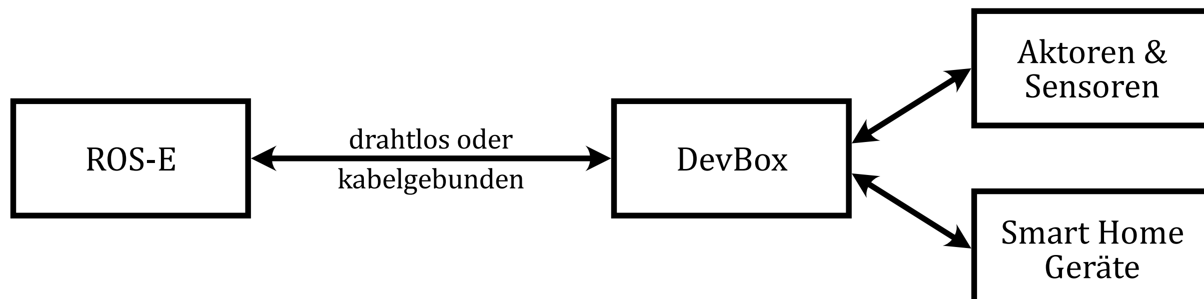

Abbildung 1: Skizze zur Architektur der DevBox in Verwendung mit einer ROS-E

Im Wesentlichen stellt sich heraus, dass die DevBox zu einer kabelgebundenen oder drahtlosen Verbindung mit einer ROS-E fähig sein und mit diversen Aktoren, Sensoren und *Smart Home* Geräten interagieren soll.

1.3 Zielgruppe und Abgrenzung

Die DevBox richtet sich ganz klar an die Entwickler von ROS-E und soll auch primär für diese Zielgruppe konzipiert werden. Darüber hinaus wäre der Einsatz in der Lehre und für Praktika von Studierenden denkbar, sodass diese Zielgruppe sekundär mitbetrachtet wird. Die DevBox muss mit dem System von ROS-E kompatibel sein, sodass die Verwendung mit anderen Systemen zwar nicht ausgeschlossen, aber auch nicht Ziel dieser Arbeit ist. Zudem soll die DevBox explizit für die Entwicklung und Erprobung von Hardware und der dazugehörigen Software verwendet werden, aber nicht für eine dauerhafte Anbindung oder Integration über die DevBox.

1.4 Aufbau der Arbeit

In den weiteren Kapiteln dieser Arbeit wird zunächst in Kapitel 2 ein genauerer Blick auf den Roboter ROS-E und den Ist-Zustand geworfen. In Kapitel 3 folgt eine Erläuterung wichtiger Technologien, die diese Arbeit verwendet. Anschließend werden die Anforderungen an die DevBox basierend auf Anwendungsfällen der Zielgruppen in Kapitel 4 aufgestellt. Mit diesen Anforderungen wird im folgenden Kapitel 5 die Systemarchitektur der DevBox in Verbindung mit ROS-E und das Konzept für die Hardware und Software der DevBox entwickelt. In Kapitel 6 wird daraufhin eine prototypische Umsetzung des Konzepts und in Kapitel 6.2.6 Tests und deren Ergebnisse dieser Umsetzung erläutert. Zum Schluss folgt ein Fazit mit einem Ausblick und einer Zusammenfassung in Kapitel 8 sowie der Anhang in Kapitel 9.

Vor dem Lesen dieser Arbeit wird dem Leser empfohlen, die Hinweise zum Lesen der Arbeit zu beachten. Ein Glossar und verwendete Abkürzungen sowie Verzeichnisse für Abbildungen, Tabellen und referenzierte Quellen befinden sich am Ende dieser Arbeit.

2 Roboter „ROS-E“

In der Einleitung wurde ROS-E bereits als eine Entwicklung des RobotikLab Telematik der *TH* Wildau vorgestellt. In diesem Kapitel wird die Hardware und die darauf ausgeführte Software von ROS-E genauer erläutert. Zudem werden die aktuellen und ROS-E betreffenden Entwicklungsprozesse sowie wichtige vorhandene Technologien betrachtet.

2.1 Aufbau & Hardware

Die Hardware von ROS-E besteht im Kern aus einem Raspberry Pi 4B und einer Hauptplatine, die unter anderem verschiedene Komponenten miteinander verbindet und diese mit Spannung versorgt. Daran angeschlossen sind verschiedene Aktoren und Sensoren. In einigen Fällen wird zusätzlich ein Mikrocontroller für die Anbindung verwendet, die über das *I²C*-Bussystem mit dem Raspberry Pi kommunizieren. Die folgende Abbildung zeigt einen groben Überblick aller Komponenten von ROS-E. [1] [5]

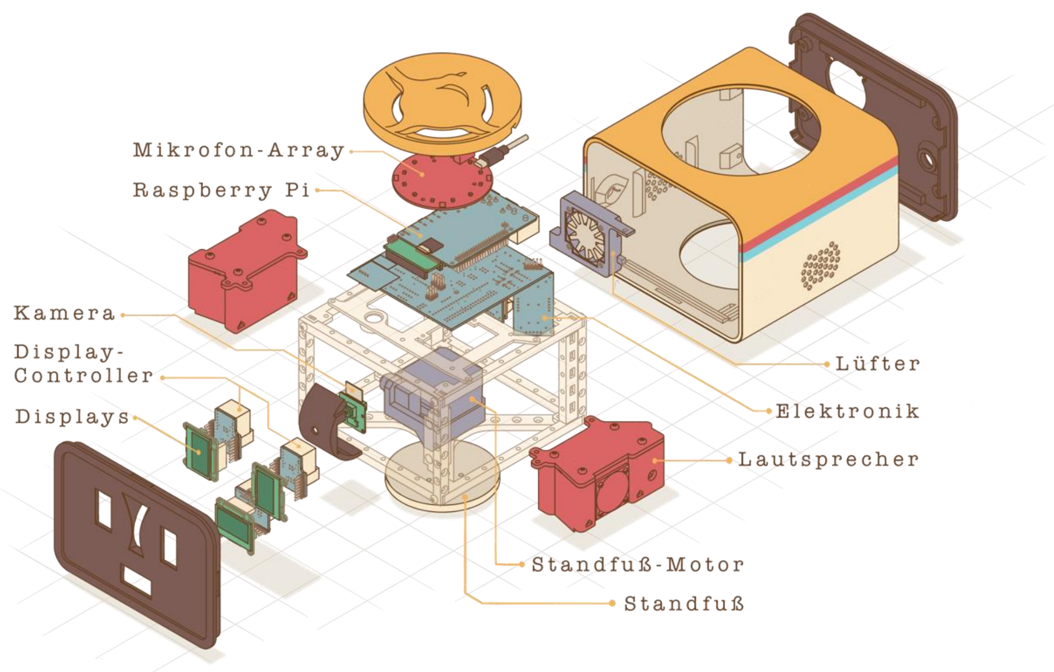

Abbildung 2: Hardwarekomponenten von ROS-E [RobotikLab, TH Wildau]

Das Gesicht von ROS-E setzt sich aus drei Displays für einen Mund und zwei Augen zusammen. Zwischen den beiden Augen sitzt eine vertikal schwenkbare Kamera. Auf der Oberseite befindet sich ein ringförmiges Mikrofonarray. Auf der linken und rechten Seite von ROS-E befindet sich am unteren Ende jeweils ein Lautsprecher. Zwischen den Lautsprechern befindet sich ein Motor, der den Körper von ROS-E horizontal schwenken kann.

Neben den dargestellten Komponenten beinhaltet der verbaute Raspberry Pi die nötige Hardware für eine drahtlose Kommunikation via *Wi-Fi* und *Bluetooth*. Zudem verfügt der Raspberry Pi über zwei *USB* 2.0, zwei *USB* 3.0 und einem *Ethernet*-Anschluss. Wie die nachfolgende Abbildung 4 zeigt, sind diese Anschlüsse gut zugänglich. [1] [5]

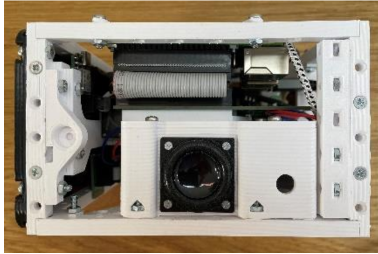

Abbildung 3: Blick in das Innere von ROS-E (rechts)

Abbildung 4: Blick in das Innere von ROS-E (Rückseite)

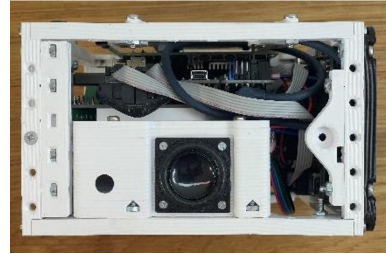

Abbildung 5: Blick in das Innere von ROS-E (links)

Es ist auch geplant, die beiden *Smart Home* Funkstandards *ZigBee* und *Z-Wave* in ROS-E zu integrieren. Jedoch steht jene Integration noch aus, da die entsprechenden *USB*-Funkmodule teilweise nicht in das Gehäuse von ROS-E passen. Die Funkmodule sollen bei einer nächsten Hardwareversion mitbedacht werden. [3] [5]

Die obigen Abbildung 3 und Abbildung 5 zeigen auch, dass andere Anschlüsse des Raspberry Pi schwerer zu erreichen oder bereits mit anderen Komponenten verbunden sind. Dabei sind besonders gängige Hardwareschnittstellen wie *GPIO*, *I^2C* und *SPI* schwer zugänglich.

2.2 Software

Auf dem Raspberry Pi 4 wird Ubuntu 20.04 LTS als Betriebssystem und zusätzlich das *Robot Operating System* (*ROS*) 2 verwendet. Zu der Basissoftware von ROS-E gehören verschiedene Entwicklungs- und Laufzeitumgebungen für C/C++, JavaScript (via Node.js) und Python. Die Software von ROS-E wird größtenteils in Python 3 mit *ROS* entwickelt. Dies gilt vor allem für Kernkomponenten und Software für den Zugriff auf Hardwareschnittstellen. [1] [5] [6]

2.3 Entwicklungsprozesse

Im Kern zielt die Arbeit auf die Frage ab, wie die Weiterentwicklung der Hardware von ROS-E vereinfacht werden kann und neue Ideen schneller ausprobiert werden können. Dazu wird folgend ein Blick auf aktuelle Prozesse und Eigenschaften der Entwicklung von ROS-E geworfen.

Durch die ständige Weiterentwicklung von ROS-E durch studentische Hilfskräfte, in Praktika oder Abschlussarbeiten entstehen neue Anwendungen oder Erweiterungen des Basissystems von ROS-E. Bei vielen dieser Entwicklungen werden neue Technologien in

ROS-E integriert, wovon einige auch neue Hardware von ROS-E benötigen. Der Aufbau von ROS-E integriert verschiedene Hardwarekomponenten fest im Gehäuse, sodass der Zugang meist erschwert ist. Somit bleibt oft nur die Möglichkeit neue Hardwarekomponenten losgelöst von ROS-E zu testen und zu entwickeln, um diese später in das Gesamtsystem zu integrieren, wenn genug Neuerungen in einer neuen Hardwareversion von ROS-E zusammengefasst werden können. Der Bau einer neuen ROS-E für jede zu testende Hardwarekomponente wäre zu aufwändig, langsam und teuer. Bei einer externen Entwicklung ist jedoch die Verwendung der neuen Hardware mit den anderen Systemen von ROS-E aktuell impraktikabel.

Mit der aktuellen Hardwareversion von ROS-E können bereits viele Anwendungen entwickelt werden, womit eine gute Grundlage für die Softwareentwicklung geboten ist. Nichtsdestotrotz gibt es viele Ideen für die Erweiterungen der Hardware, die zurzeit nicht angemessen getestet werden können.

2.4 Bestehende Technologien

Für ROS-E wurde bereits Software entwickelt, die Zugriffe auf Hardwareschnittstellen des Raspberry Pi abstrahiert. Diese Software ist ausschließlich für die Ansteuerung interner Komponenten von ROS-E und nicht für die Verwendung externer Geräte konzipiert. [7]

3 Verwendete Technologien

Im Zusammenhang mit ROS-E wurden bereits diverse Technologien angesprochen. In diesem Kapitel werden diese und andere wichtige Technologien erläutert, um das Verständnis der folgenden Kapitel zu vereinfachen. Weitere Technologien, die einer weniger ausführlichen Erklärung bedürfen, werden kurz im Glossar am Ende dieser Arbeit erklärt.

3.1 Robot Operating System

Das *Robot Operating System* (*ROS*) ist eine Open-Source-Plattform für die Entwicklung von Software für Roboter. Es ist weniger ein vollständiges Betriebssystem, wie es der Name suggeriert, sondern besteht aus Softwarebibliotheken und Programmen, die allgemein bei der Entwicklung von Robotern unterstützen. [8] [9]

Es gibt zwei Hauptversionen von *ROS*. In dieser Arbeit wird, sofern nicht explizit erwähnt, immer von *ROS* 2 ausgegangen, welches auch bei *ROS*-E zum Einsatz kommt.

Ein Programm mit *ROS* besteht aus Knoten (engl. Nodes), die idealerweise jeweils eine Gruppe an Funktionen implementieren. Es kann beispielsweise ein Knoten für die Steuerung einer Kamera und ein weiterer Knoten für die Bilderkennung basierend auf den Kameradaten implementiert werden. *ROS*-Knoten werden als Bestandteil von Paketen entwickelt, die mehrere Knoten und andere Dateien enthalten können. [10]

Zwischen *ROS*-Knoten können Daten mittels Topics, Services und Actions ausgetauscht werden. Unter einem Topic können Nachrichten von Knoten veröffentlicht und abonniert werden. Bei einer Veröffentlichung erhält jeder Knoten, der das entsprechende Topic abonniert hat, diese Nachricht. Die Anzahl der Knoten, die ein Topic abonnieren oder unter diesem veröffentlichen können, ist unbeschränkt. [10] [11]

Im Gegensatz zu Topics, die nach dem Publish-Subscribe-Prinzip arbeiten, können Daten mit Services nach dem Request-Response-Prinzip ausgetauscht werden. Ein Service antwortet nur dann einem Knoten mit Daten, wenn zuvor eine Anfrage von diesem Knoten an den Service gestellt wurde. Es kann nur einen Knoten geben, der die Anfragen für einen Service beantwortet, aber beliebig viele Knoten, die Anfragen stellen. [12]

Eine Action ist eine Kombination aus Topic und Service und ist für eine langlaufende Aufgabe gedacht. Indem ein Ziel (engl. Goal) übermittelt wird, beginnt eine Action und sendet ständig Feedback, während sie ausgeführt wird. Die Action endet mit dem Übermitteln eines Ergebnisses oder einem Abbruch des Ziels. Ähnlich einem Service, kann eine Action nur von einem Knoten angeboten werden. [13]

Ein Topic, Service oder eine Action wird durch einen Namen eindeutig angesprochen. Der Name wird durch eine alphanumerische Zeichenkette angegeben, die ähnlich wie ein Dateipfad aufgebaut ist. Ein möglicher Name für ein Topic, auf dem das Bildmaterial einer Kamera veröffentlicht wird, könnte beispielsweise `/camera/video` lauten. [14]

Zusätzlich kann jeder *ROS*-Knoten Parameter definieren. Somit lassen sich Knoten beim Starten oder dynamisch zur Laufzeit konfigurieren. Ein Parameter hat einen eindeutigen Namen und hält einen skalaren Wert; also eine Zahl, eine Zeichenkette, ein Wahrheitswert oder eine Liste von Werten. [15]

3.2 Message Queuing Telemetry Transport

Das *Message Queuing Telemetry Transport* (*MQTT*) Protokoll ist ein *OASIS* und *ISO*-Standard für die Kommunikation zwischen Maschinen und im *Internet der Dinge* (*IoT*). Das Ziel von *MQTT* ist ein einfaches Format zur effizienten Übertragung von Daten, welches auch von Geräten mit eingeschränkten Ressourcen einfach implementiert werden kann. [16] [17]

Mit *MQTT* können Nachrichten nach dem Publish-Subscribe-Prinzip ausgetauscht werden. Geräte können Daten für andere Geräte veröffentlichen (engl. publish) und die Veröffentlichung von Daten anderer Geräte abonnieren (engl. subscribe). Die Kommunikation erfolgt dabei nicht direkt zwischen den beteiligten Geräten, sondern über einen sogenannten *MQTT*-Broker. Dadurch werden die Geräte sowie die Datenerzeugung und -nutzung voneinander entkoppelt. Die folgende Abbildung 6 zeigt ein Beispiel für diese Architektur. [18]

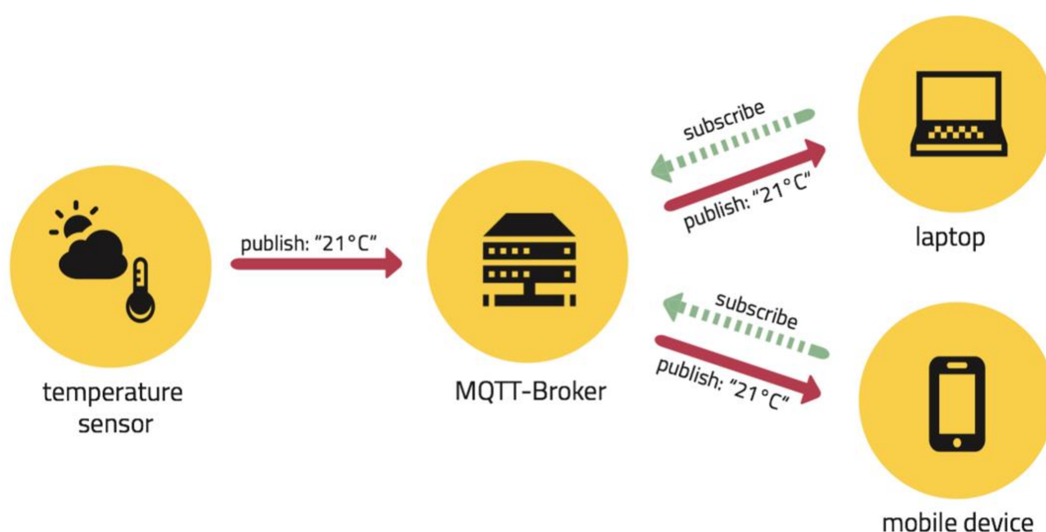

Abbildung 6: Beispiel zur Publish-Subscribe-Architektur von MQTT [heise.de]

Alle Nachrichten werden bei *MQTT* unter einem entsprechenden Topic veröffentlicht. Ein Topic ist eine Zeichenkette, die vom Aufbau an einen Dateipfad erinnert. Das Topic, unter dem der in Abbildung 6 gezeigte Temperatursensor veröffentlicht, könnte beispielsweise `home/garden/temperature` lauten. Wie die Topics organisiert sind und welchen Datentyp die darunter veröffentlichten Nachrichten haben, ist dabei vollständig der Anwendung überlassen. [18] [17]

3.3 Hardwareschnittstellen

Ein Ziel der DevBox ist die Ansteuerung von Aktoren und Sensoren. In den folgenden Unterkapiteln werden wichtige Hardwareschnittstellen erläutert.

3.3.1 General Purpose Input/Output

Ein *General Purpose Input/Output* (*GPIO*, dt. Allzweckeingabe/-ausgabe) ist eine digitale Schnittstelle eines integrierten Schaltkreises, welche mittels Software gesteuert werden kann. Dabei sind *GPIO*s nicht an einen bestimmten Zweck gebunden und können beliebig eingesetzt werden. [19]

Jeder *GPIO* kann in der Regel als Eingabe oder als Ausgabe konfiguriert werden. Als Eingabe kann das digitale Signal, welches an dem *GPIO* anliegt, ausgelesen werden. In der Funktion der Ausgabe kann das Signal auf ein beliebigen Logikpegel eingestellt werden. Somit lässt sich je *GPIO* ein Bit mit den Werten 0 (LOW) oder 1 (HIGH) steuern. Meistens kann auch eine Flankenerkennung für ein *GPIO* eingerichtet werden, sodass ein Interrupt oder ein Ereignis ausgelöst wird, wenn das Signal von LOW zu HIGH (RISING) oder von HIGH zu LOW (FALLING) wechselt. [19]

3.3.2 Inter-Integrated Circuit

Der *Inter-Integrated Circuit* (*I²C*, gesprochen I-Quadrat-C) Bus wurde von Philips Semiconductors (inzwischen NXP Semiconductors) für eine serielle, bidirektionale Kommunikation zwischen integrierten Schaltkreisen (engl. Integrated Circuit) entwickelt. [20]

An einem *I²C*-Bus können Controller und Targets teilnehmen, wobei beide Gerätearten senden und empfangen können, jedoch nur ein Controller den Kommunikationsablauf des Busses steuern kann. Jedes angeschlossene Gerät hat eine eigene Adresse, mit der es Zwecks einer Datenübertragung angesprochen werden kann. Die Geräte eines *I²C*-Busses sind über zwei Leitungen miteinander verbunden, wie in der folgenden Abbildung 7 beispielhaft dargestellt. [20]

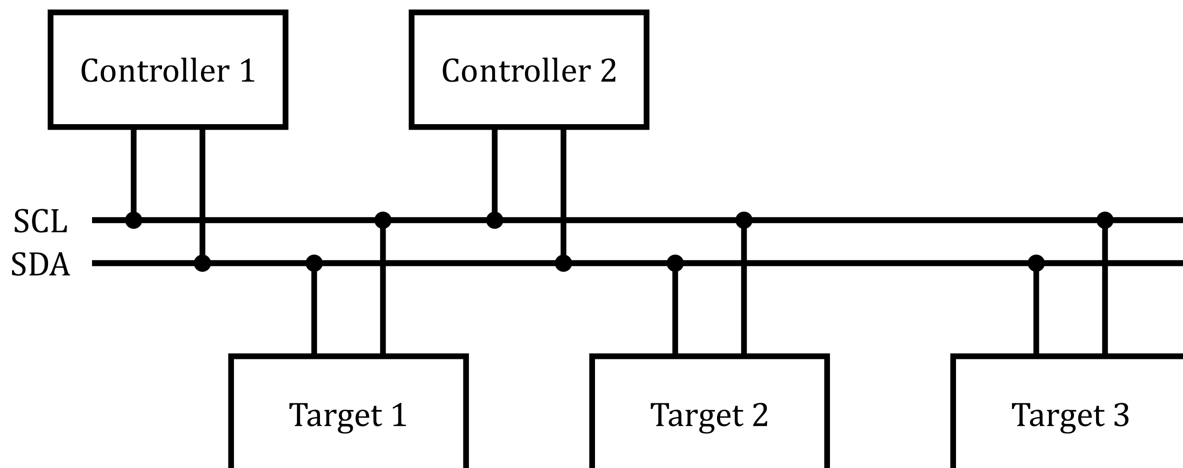

Abbildung 7: I²C-Bus mit zwei I²C-Controllern und drei I²C-Targets

Über die SCL-Leitung (Serial Clock) wird ein Takt zur Synchronisation während der Übertragung von Daten von dem Controller generiert, der die Übertragung initiiert hat. Die Daten werden über die SDA-Leitung (Serial Data) ausgetauscht. Die Richtung der Übertragung (lesen oder schreiben) und die Adresse des *I²C*-Targets wird durch den Controller vor Beginn jeder Übertragung auf der SDA-Leitung ausgegeben. [20]

3.3.3 Serial Peripheral Interface

Das *Serial Peripheral Interface* (*SPI*) erlaubt eine synchrone Duplexkommunikation zwischen einem Controller und peripheren Geräten über einen seriellen Bus. Der Standard wurde ursprünglich unter Motorola im Jahr 1987 entwickelt. In älteren Dokumenten werden Begriffe wie „Master“ und „Slave“ verwendet. Diese Arbeit sieht von einer Verwendung solcher Begriffe ab. Stattdessen werden die offiziellen, neuen Begriffe wie „Controller“, „Peripheral“ und „Chip“ verwendet. [21] [22] [23] [24]

Bei *SPI* gibt der Controller den Ablauf der Kommunikation vor. Nur ein Controller kann eine Übertragung beginnen. An dem Controller per *SPI* angeschlossene *Peripheriegeräte* (engl. Peripherals), folgen dem vorgegeben Ablauf des Controllers. Der Controller und ein *Peripheriegerät* sind über vier Leitungen miteinander verbunden, wie in der folgenden Abbildung 8 dargestellt. [21]

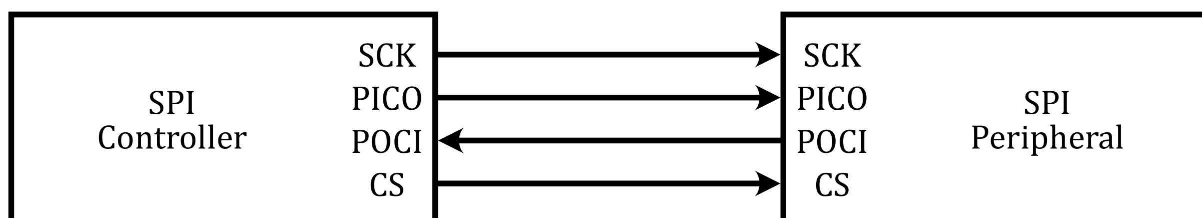

Abbildung 8: SPI-Bus mit einem SPI-Controller und SPI-Peripheral

Der Controller gibt bei *SPI* einen gemeinsamen Takt über die SCK-Leitung (Serial Clock) zur Synchronisation der Übertragung an. Die Daten werden über die Leitungen PICO (Peripheral Input, Controller Output) und POCI (Peripheral Output, Controller Input) zwischen den Geräten übertragen. Über die CS-Leitung (Chip Select) signalisiert der Controller dem *SPI*-Peripheral, ob die Übertragung aktiv ist. In diesem Fall senden und empfangen beide Geräte gleichzeitig Daten über die PICO- und POCI-Leitung. [21]

3.4 Smart Home Protokolle

Ein weiteres Ziel für die DevBox ist die Möglichkeit zur Anbindung von *Smart Home* Geräten. In einem Praktikum zu diesem Thema wurden bereits verschiedene Funkstandards für die Kommunikation mit *Smart Home* Geräten für ROS-E evaluiert und ausgewählt. In den folgenden Unterkapiteln werden diese Standards kurz erläutert. [3]

3.4.1 EnOcean

EnOcean ist ein Funkstandard, der für besonders niedrigen Energieverbrauch optimiert ist. Der Standard ist international unter verschiedenen Frequenzbändern als *ISO*/IEC 14543-3-10 genormt. [25]

Die Pakete des Funkprotokolls (genannt „Telegramme“) sind sehr kurzgehalten, um Kollisionen untereinander zu vermeiden. Zudem nutzt *EnOcean* in Europa ausschließlich das 868 MHz Frequenzband, sodass keine Störungen durch *Wi-Fi*- oder *Bluetooth*-Signale entstehen. Die Kommunikation von *EnOcean* ist mittels *AES* (128 Bit) gegen das Abhören und mit einem *Rolling-Code* gegen das erneute Senden von Telegrammen gesichert. [3] [26] [27]

Wie bereits erwähnt, ist *EnOcean* für eine besonders energiearme Kommunikation entwickelt wurden. Es erlaubt den Betrieb mittels Batterie, aber auch durch das sogenannte *Energy Harvesting*, welches Energie aus der Umgebung des Geräts nutzt. So kann Storm beispielsweise von einer Solarzelle, der kinetischen Energie vom Drücken eines Schalters oder durch einen Temperaturunterschied erzeugt werden. [26]

Der Markt um die *EnOcean* Technologie fokussiert sich vorranging auf Sensorik, die mit einem geringen Energiebedarf batteriebetrieben oder mittels *Energy Harvesting* arbeiten. [3] [28] [29]

3.4.2 ZigBee

Der *ZigBee*-Standard ist ein weiterer in Europa weit verbreiteter *Smart Home* Funkstandard. Er wird von der Connectivity Standards Alliance[2] (früher *ZigBee* Alliance) entwickelt und verwaltet.

ZigBee ist für die Frequenzbänder 868 MHz und 2,4 GHz in Europa konzipiert. Jedoch werden nur noch *ZigBee*-Geräte für das 2,4 GHz Band hergestellt, sodass defacto auch nur dieses Frequenzband für *ZigBee* verwendet wird. Auf diesem Band funken allerdings auch viele andere Standards wie *Wi-Fi* und *Bluetooth*, sodass eine gegenseitige Störung das Resultat ist. Aufgrund der kürzeren Wellenlänge ist die Fähigkeit größere Reichweiten und Wände zu überwinden gemindert. [3]

3.4.3 Z-Wave

Ein weiterer wichtiger Standard ist *Z-Wave*. Es handelt sich um einen weit verbreiteten proprietären *Smart Home* Standard. [3] [30]

Für *Z-Wave* wird in Europa ausschließlich das 868 MHz Frequenzband verwendet, welches gute Übertragungseigenschaften im Vergleich zu 2,4 GHz bietet, wie schon in den vorangehenden Kapiteln erläutert. Die Reichweite einer Verbindung liegt bei ca. 30 Metern, jedoch kann die Reichweite über andere *Z-Wave* Geräte bis zu viermal erweitert werden, indem die Nachrichten weitergeleitet werden. [30]

Z-Wave ist mit *AES* (128 Bit) verschlüsselt und setzt Elliptic Curve Diffie-Hellman (*ECDH*) für die sichere Generierung von Sicherheitsschlüsseln während des Hinzufügens von neuen Geräten ein. [3] [31]

[2] Siehe https://csa-iot.org.

4 Analyse der Anforderungen an die DevBox

Nachdem wichtige Grundlagen und der Ausgangszustand geklärt wurden, wird sich dieses Kapitel mit der Analyse und Definition der Anforderungen an die DevBox anhand verschiedener Anwendungsfälle beschäftigen.

4.1 Anwendungsfälle

Um die genauen Anforderungen an die zu entwickelnde DevBox festzustellen, werden im Folgenden verschiedene Anwendungsfälle von den Entwicklern des Teams von ROS-E betrachtet.

Ein Anwendungsfall stammt von Lara Ziemert, die mit ihrer Masterarbeit (siehe [32]) eine Situationserkennung mithilfe eines künstlichen neuralen Netzes für ROS-E entwickelt hat. Dafür werden Ton und Bild vom Mikrofon bzw. der Kamera als Eingangskanäle für die Situationserkennung verwendet. Sie möchte nun, nach Abschluss ihrer Masterarbeit, weiter erforschen, ob und welche anderen Sensoren sich als Eingänge für die Situationserkennung eignen würden und will daher sowohl mit bereits in ROS-E eingebauten als auch mit neuen Sensoren uneingeschränkt experimentieren. Dazu gehört auch die Möglichkeit, die Sensorik in verschiedenen Umgebungen zu nutzen und zu testen. Zudem möchte sie die Sensoren unkompliziert mit ihrer bereits für ROS-E entwickelten Anwendung nutzen können. Da das Training des neuronalen Netzes sehr aufwändig und mit den Ressourcen von ROS-E nicht praktikabel ist, würde eine externe Recheneinheit für diese Prozesse sowie ein schneller und großer Speicher für die Trainingsdaten die weitere Forschung erleichtern.

Aus Gesprächen mit weiteren Mitgliedern des Kernteams um ROS-E hat sich ergeben, dass möglichst viele Hardwareschnittstellen und Bussysteme, mit denen eine Vielzahl an Sensoren und Aktoren kompatibel sind, unterstützt werden sollten. Im speziellen ist gewünscht, dass Hardwarekomponenten über *GPIO*, *I²C* und *SPI* angeschlossen werden können beziehungsweise Anschlüsse für diese Standards vorhanden sind. Des Weiteren wird die Errichtung einer gleichen oder zumindest zu der von ROS-E ähnlichen Softwareumgebung für die DevBox befürwortet.

Ein letzter, möglicher Anwendungsfall ergibt sich im Bereich der Lehre, zum Beispiel im Rahmen von Schülerlaboren oder Lehrveranstaltungen. In diesem Fall ist ein System gefordert, mit dem sich in kurzer Zeit und geringem Vorwissen einfache Schaltungen aufbauen, programmieren und verstehen lassen. Zudem wäre es gut, wenn Interaktionen die Neugier der Teilnehmer wecken und letztendlich zum eigenständigen Experimentieren anregen. Dafür ist wiederum eine große Flexibilität bei den möglichen Hardwarekomponenten und Verbindungen erforderlich.

4.2 Definition der Anforderungen

Nachdem verschiedene Anwendungsfälle für die DevBox betrachtet wurden, werden in diesem Abschnitt daraus abgeleitete Anforderungen definiert. Diese werden nach dem Design, der Hardware und Software gruppiert. Am Ende jedes Absatzes wird auf IDs der entsprechenden Anforderungen im Abschnitt *4.3* verwiesen, wo diese in einer tabellarischen Übersicht zusammengestellt sind.

4.2.1 Anforderungen an das Design

Auch wenn die Ästhetik der DevBox, die vor allem den Entwicklungsprozess unterstützen und eher zweitrangig betrachtet werden soll, gibt es dennoch Anforderungen an das Design bezüglich der Benutzung und Ergonomie.

Wie aus vorherigen Betrachtungen hervorgeht, soll es möglich sein, die DevBox zwischen verschiedenen Orten zu transportieren, wobei die DevBox dem Transport standhalten sollte. Daraus geht auch hervor, dass die DevBox verschließbar sein sollte, um angeschlossene Elektronik sicher und geschützt zu befördern. *(R1, R2)*

Da für die Verbindung von Hardwarekomponenten Kabel und gegebenenfalls auch Kleinkomponenten wie Widerstände notwendig sind, kann die DevBox eine Möglichkeit der Aufbewahrung für Sensoren und Aktoren sowie Kabel und kleinerer Bauelemente anbieten. *(R3)*

Eine weitere Anforderung, die nicht nur auf das Design oder die Bedienung beschränkt ist, sondern auch von der Hardware und Software entsprechend umgesetzt werden muss, betrifft das *Plug & Play* von Sensoren und Aktoren. Diese müssen nach dem Anschließen an der DevBox für den Benutzer in kurzer Zeit beziehungsweise ohne großen Konfigurationsaufwand verwendbar sein. *(R4, R5)*

4.2.2 Anforderungen an die Hardware

Eines der primären Ziele für die DevBox ist das flexible Einbinden von verschiedenen Hardwarekomponenten. Um diese Vielfalt anbieten zu können, muss die DevBox die gängige Hardwareschnittstelle *GPIO* und die Bussysteme *I^2C* und *SPI* bereitstellen, da viele Sensoren und Aktoren die genannten Bussysteme unterstützen oder direkt über *GPIO* angesteuert werden können. *(R6, R7, R8, R9, R10, R11, R12)*

Zur Unterstützung und Auslagerung der Verarbeitung von Eingangssignalen oder Steuerbefehlen sollte die DevBox einen Mikrocontroller beinhalten, der Aufgaben dieser Art übernehmen kann. Die Möglichkeit der Aktualisierung von Programmen des Mikrocontrollers über die zu entwickelnde DevBox kann angeboten werden, wodurch Zeit und Aufwand für das Umstecken von Kabelverbindungen gespart werden könnte. *(R13, R14)*

Für die notwendige Verkabelung der Sensoren und Aktoren untereinander, mit den Systemschnittstellen oder mit anderen Bauelementen, muss die DevBox eine *Steckplatine* (engl. Breadboard) beinhalten. Auf dieser können die verschiedenen Verbindungen leicht und schnell zusammengesteckt werden. Die *Steckplatine* soll ausreichend groß sein, sodass die meisten Sensoren und Aktoren genutzt werden können. *(R15, R16)*

Weitere Komponenten sollen über eine *USB*-Verbindung verfügbar gemacht werden. Insbesondere müssen *USB*-Adapter für die Funkstandards *EnOcean*, *ZigBee* und *Z-Wave* in die DevBox intergiert werden. Über diese soll die Kommunikation mit, für den jeweiligen Standard verfügbaren, *Smart Home* Geräten ermöglicht werden. Somit sollen vor allem Anwendungsfälle, die sich mit *Gebäudeautomation* und *Smart Home* befassen, ermöglicht werden. *(R17, R18, R19, R20, R21)*

Da Hardwarekomponenten, die nicht über einen Anschluss mit integrierter Spannungsversorgung angeschlossen sind, dennoch eine Versorgungsspannung benötigen, muss diese separat von der DevBox bereitgestellt werden. Dabei sollten gängige Spannungen wie 3,3 Volt und 5 Volt unterstützt werden. *(R22, R23)*

Die Kommunikation zwischen der DevBox und ROS-E muss sowohl über eine kabelgebundene Verbindung als auch drahtlos möglich sein. Somit kann je nach Anwendungsfall zwischen einer schnellen und robusten, jedoch auch auf einen Ort beschränkten, oder einer portablen und flexibleren, aber gegebenenfalls nicht so schnellen Verbindung gewählt werden. *(R24, R25, R26)*

4.2.3 Anforderungen an die Software

Aufbauend auf der Hardware der DevBox muss zusätzliche Software die Kommunikation mit der verbundenen ROS-E abwickeln. Die von ROS-E entsandten Steuerbefehle sowie Eingangssignale müssen von der DevBox-Software entsprechend übersetzt und transportiert werden. Dadurch werden diese Aufgaben dem Verwender abgenommen, sodass sich dieser mehr auf die eigentliche Anwendung konzentrieren kann. *(R27, R28)*

Für die Kommunikation mit Sensoren und Aktoren, die an die DevBox angeschlossen sind, muss eine geeignete Schnittstelle angeboten werden. Über diese muss das Abrufen von Sensorwerten sowie das Einstellen der Zustände von Aktoren möglich sein. Da in der Entwicklung verschiedene Programmiersprachen zum Einsatz kommen, sollte die Schnittstelle unabhängig davon angeboten werden und funktionieren. *(R30, R31)*

Es muss zudem eine geeignete Schnittstelle für die mit der DevBox verbundenen *Smart Home* Geräte geben. Über diese Schnittstelle muss es möglich sein, diese Geräte zu steuern und den Zustand der Geräte zu erfahren. Als grundlegende Technologie für die

Kommunikation zwischen ROS-E und *Smart Home* Geräten der DevBox muss *MQTT* verwendet werden. *(R32, R33, R29)*

Das Design, die Hardware und Software sollen zwar einfach zu benutzen sein, dennoch muss den potenziellen Benutzern eine gute Dokumentation zur Verfügung stehen, die einen einfachen und schnellen Einstieg in die Verwendung der Software (und Hardware) bietet. Die Dokumentation muss alle wichtigen Informationen für die Verwendung beinhalten und sollte beispielhafte Anleitungen für verschiedene Anwendungsfälle sowie passende Beispielprogramme (oder essenzielle Ausschnitte daraus) aufweisen. *(R34, R35, R36, R37)*

4.3 Tabellarische Übersicht der Anforderungen

In der folgenden Tabelle werden alle zuvor definierten Anforderungen angeführt. Es wird nach Kategorie (MUSS, KANN oder SOLL) sowie Art der Anforderung (*F* = funktional oder *NF* = nicht funktional) unterschieden. Jede Anforderung erhält eine fortlaufende, numerische ID, mit der die entsprechende Anforderung im Text referenziert wird.

ID	Komponente	Kategorie	Beschreibung	Art
R1	Die DevBox	SOLL	einem Transport standhalten.	*NF*
R2	Die DevBox	SOLL	verbundene Elektronik im Fall eines Transports vor Schäden beschützend einschließen.	*NF*
R3	Die DevBox	KANN	Räume zur Aufbewahrung von elektronischen Bauelementen enthalten.	*NF*
R4	Die DevBox	MUSS	angeschlossene Sensoren & Aktoren innerhalb von 30 Sekunden dem Benutzer verfügbar machen.	*NF*
R5	Die DevBox	SOLL	dem Benutzer ermöglichen, die Konfiguration von angeschlossenen Sensoren & Aktoren innerhalb von 5 Minuten abzuschließen.	*NF*
R6	Die DevBox	MUSS	über mindestens 20 *GPIO*-Anschlüsse zur freien Verwendung durch den Benutzer verfügen.	*NF*

R7	Die DevBox	MUSS	dem Benutzer ermöglichen, *GPIO*-Ausgänge zu steuern.	*F*
R8	Die DevBox	MUSS	dem Benutzer ermöglichen, *GPIO*-Eingänge auszulesen.	*F*
R9	Die DevBox	MUSS	über einen Anschluss für das *I²C*-Bussystem verfügen.	*F*
R10	Die DevBox	MUSS	dem Benutzer ermöglichen, mit Hardwarekomponenten, die per *I²C* angeschlossen sind und dies unterstützen, zu kommunizieren.	*F*
R11	Die DevBox	MUSS	über einen Anschluss für das *SPI*-Bussystem verfügen.	*F*
R12	Die DevBox	MUSS	dem Benutzer ermöglichen, mit Hardwarekomponenten, die per *SPI* angeschlossen sind und dies unterstützen, zu kommunizieren.	*F*
R13	Die DevBox	SOLL	mindestens einen Mikrocontroller zum Zweck der freien Programmierung durch den Benutzer beinhalten.	*F*
R14	Die DevBox	KANN	dem Benutzer ermöglichen, Programme auf angeschlossene Mikrocontrollern zu überspielen.	*F*
R15	Die DevBox	MUSS	eine *Steckplatine* beinhalten.	*F*
R16	Die *Steckplatine* der DevBox	SOLL	mindestens eine Fläche von 100 cm^2 abdecken.	*NF*
R17	Die DevBox	SOLL	über mindestens 4 *USB*-Anschlüsse zur freien Verwendung durch den Benutzer verfügen.	*F*

R18	Jeder *USB*-Anschluss der DevBox	SOLL	mindestens *USB* 2.0 unterstützen.	*NF*
R19	Die DevBox	MUSS	die Kommunikation mit kompatiblen *Smart Home* Geräten über den Funkstandard *EnOcean* ermöglichen.	*F*
R20	Die DevBox	MUSS	die Kommunikation mit kompatiblen *Smart Home* Geräten über den Funkstandard *ZigBee* ermöglichen.	*F*
R21	Die DevBox	MUSS	die Kommunikation mit kompatiblen *Smart Home* Geräten über den Funkstandard *Z-Wave* ermöglichen.	*F*
R22	Die DevBox	MUSS	die Möglichkeit bieten, Hardware-komponenten mit einer Spannung von 3,3 Volt zu versorgen.	*NF*
R23	Die DevBox	MUSS	die Möglichkeit bieten, Hardware-komponenten mit einer Spannung von 5 Volt zu versorgen.	*NF*
R24	Die DevBox	MUSS	die Kommunikation zwischen allen angeschlossenen Komponenten und einer verbundenen ROS-E ermöglichen.	*F*
R25	Die DevBox	MUSS	eine kabelgebundene Verbindung zu ROS-E zum Zweck der Kommunikation ermöglichen.	*NF*
R26	Die DevBox	MUSS	eine drahtlose Verbindung zu ROS-E zum Zweck der Kommunikation ermöglichen.	*NF*
R27	Die DevBox	MUSS	Steuersignale vom Benutzer, die für die angeschlossene Hardware bestimmt sind, entsprechend der	*F*

			Anschlussmethode übersetzen und zustellen.	
R28	Die DevBox	MUSS	Eingangssignale für den Benutzer, die von der angeschlossenen Hardware stammen, entsprechend der Anschlussmethode übersetzen und zustellen.	*F*
R29	Die DevBox	MUSS	die Kommunikation mit verbundenen *Smart Home* Geräten per *MQTT* gegenüber ROS-E ermöglichen.	*F*
R30	Die DevBox	MUSS	eine Programmierschnittstelle (*API*) für die Kommunikation mit angeschlossenen Geräten bereitstellen.	*F*
R31	Die DevBox *API*	SOLL	unabhängig von der verwendeten Programmiersprache benutzbar sein.	*NF*
R32	Die DevBox	MUSS	dem Benutzer den Zustand von verbundenen *Smart Home* Geräten bereitstellen.	*F*
R33	Die DevBox	MUSS	dem Benutzer ermöglichen, den Zustand von verbundenen *Smart Home* Geräten zu steuern.	*F*
R34	Für die DevBox	MUSS	dem Benutzer eine Dokumentation zur Verwendung der DevBox angeboten werden.	*NF*
R35	Die Dokumentation der DevBox	MUSS	alle wichtigen Informationen für die Verwendung der DevBox aus Benutzersicht enthalten.	*NF*
R36	Die Dokumentation der DevBox	SOLL	beispielhafte Anleitungen für konkrete Anwendungsfälle enthalten.	*NF*

R37	Die Dokumentation der DevBox	SOLL	Beispielprogramme für konkrete Schaltpläne mit der DevBox enthalten und erklären.	*NF*

Tabelle 1: Definition der Anforderungen an die DevBox

In diesem Kapitel wurden verschiedene Anforderungen an das Design, die Hardware und Software der DevBox basierend auf den Anwendungsfällen der primären Zielgruppe – die Entwickler von ROS-E – aufgestellt. In den weiteren Kapiteln wird erläutert, wie die DevBox mithilfe dieser Anforderungen konzipiert und umgesetzt wird.

5 Architektur- & Konzeptentwicklung

Für die soeben definierten Anforderungen soll in diesem Kapitel ein Konzept zur Umsetzung der DevBox erarbeitet werden. Im Folgenden wird zunächst die prinzipielle Architektur des geplanten Systems diskutiert, um aus dieser anschließend ein Konzept für die Hardware und Software der DevBox zu entwickeln.

5.1 Systemarchitektur

Aus den vorhergehenden Abschnitten geht bereits hervor, dass die DevBox verschiedene Schnittstellen und Gerätetypen unterstützen muss, damit das Prototyping verschiedener Ideen für ROS-E möglich ist. In der folgenden Abbildung 9 werden die diversen Verbindungen mit der DevBox dargestellt, die dafür notwendig sind.

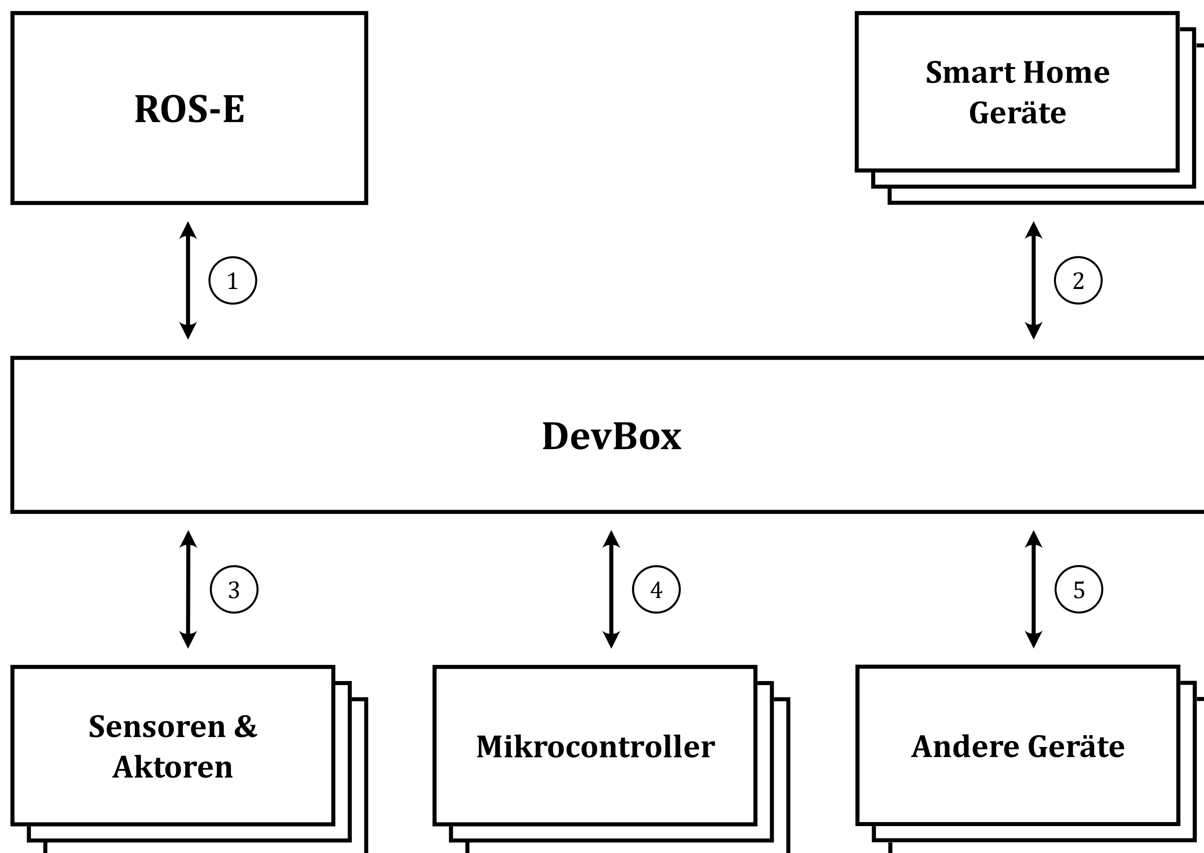

Abbildung 9: Übersicht über die Verbindungen der DevBox

Die Verbindung zwischen der DevBox und ROS-E (1) ist für die allgemeine Kommunikation mit der DevBox erforderlich und ermöglicht überhaupt die Verwendung der DevBox mit dem System und den darauf laufenden Anwendungen von ROS-E. Über diese Verbindung werden angeschlossene *Smart Home* Geräte und Hardwarekomponenten über die DevBox einer ROS-E zugänglich gemacht. Die Abbildung 9 zeigt darüber hinaus eine Verbindung mit *Smart Home* Geräten (2), wie es durch die

Anforderungen (siehe *R32* und *R33*) verlangt wird. Zudem ist die Anbindung von Sensoren und Aktoren (3) sowie Microcontroller (4) essenziell für die Entwicklung von Hardwarekomponenten für ROS-E. Im Folgenden werden verschiedene Alternativen für die einzelnen Verbindungen dieser Geräte und weiterer Geräte (5) diskutiert und ausgewählt. Als Ausgangspunkt für die folgenden Überlegungen dienen jeweils die Verbindungsmöglichkeiten von ROS-E, da das Ziel der DevBox vor allem das zugänglich machen dieser Schnittstellen ist.

5.1.1 Verbindung zu ROS-E

Die DevBox soll als Erweiterung an einer ROS-E angebunden werden können, sodass eine Verbindung zwischen der DevBox und ROS-E über eine bestehende Schnittstelle von ROS-E erfolgen muss. Als drahtlose Verbindungsmöglichkeiten werden *Bluetooth* und *Wi-Fi* von ROS-E angeboten. Für eine kabelgebundene Verbindung stehen *Ethernet* oder *USB* zur Verfügung. In diesem Abschnitt werden diese Schnittstellen bezüglich ihrer Eignung für diese Verbindung evaluiert, um eine kabelgebundene (vgl. *R25*) und drahtlose (vgl. *R26*) Schnittstelle auszuwählen. [1] [33]

Zunächst werden *Ethernet* und *USB* betrachtet. Mittels beider Standards sind je nach Version schnelle bis sehr schnelle Verbindungen der Größenordnung 1 Gbit/s möglich. Der typische Anwendungsfall von *Ethernet* ist die Anbindung von Computern an ein *Local Area Network* (*LAN*) und an das Internet. Das *USB*-Protokoll ist hingegen für die direkte Verbindung von Computern und *Peripheriegeräten* entwickelt worden. Allerdings können Computer auch per *Ethernet* direkt miteinander kommunizieren. Auf der physikalischen Verbindung aufbauend wird bei *Ethernet* fast ausschließlich das allgegenwärtige *Internetprotokoll* (*IP*) verwendet. Die gängigen Protokolle *TCP* und *UDP* sowie viele weitere können auf dieser Basis aufgesetzt werden. [34]

Nun werden *Bluetooth* und *Wi-Fi* betrachtet. Mittels *Bluetooth* ist vor allem die Verbindung zwischen *Peripheriegeräten* und Computern über kurze Distanzen möglich. Im Vergleich zu *Wi-Fi* ist *Bluetooth* um Größenordnungen langsamer, dafür aber energiesparender. Mit *Wi-Fi* werden schnelle und drahtlose Verbindungen, oft zur Anbindung an ein *LAN* oder das Internet, ermöglicht. [35]

Die Verbindung zwischen der DevBox und ROS-E muss in der Lage sein, die Kommunikation mit *Smart Home* Geräten und Hardwarekomponenten zu ermöglichen. Aus den Anforderungen ergibt sich bereits der Einsatz von *MQTT* für die Kommunikation mit *Smart Home* Geräten (vgl. *R29*). Neben *Smart Home* Geräten muss es auch möglich sein, angeschlossene Hardwarekomponenten über die Verbindung zwischen der DevBox und ROS-E anzusteuern (vgl. *R27* und *R28*). Das System von ROS-E abstrahiert Zugriffe auf die Hardware über das Austauschen von *ROS*-Nachrichten (vgl. Kapitel *2.2*). Eine Möglichkeit wäre also der Einsatz von *ROS* für die Kommunikation zwischen der DevBox

und ROS-E. Dies hätte den Vorteil, dass die DevBox mit derselben Technologie wie auch andere Hardwarekomponenten von ROS-E angesteuert wird. Das *MQTT*-Protokoll und der Austausch von *ROS*-Nachrichten erfordert eine *TCP*- oder UDP-Verbindung zwischen der DevBox und ROS-E. [36] [37] [38]

Sowohl *USB* als auch *Bluetooth* würden sich prinzipiell für eine Verbindung zwischen der DevBox und ROS-E eignen. Jedoch überwiegen die Vorteile für die Verwendung von *Ethernet* und *Wi-Fi*. Beide Standards ermöglichen eine schnelle Verbindung und unterstützen wichtige Protokolle wie *MQTT* und die Kommunikation via *ROS*. Daher wird *Ethernet* für die kabelgebundene und *Wi-Fi* für die drahtlose Verbindung zwischen der DevBox und ROS-E eingesetzt.

5.1.2 Verbindung zu Smart Home Geräten

Für die Anbindung von *Smart Home* Geräten müssen entsprechende Protokolle ausgewählt werden. Insbesondere werden folgend Funkprotokolle diskutiert, da die DevBox die Entwicklung unterstützen soll und nicht für die feste Anbindung in ein *Smart Home* über etwa ein kabelgebundenes System.

Wie zu Beginn dieser Arbeit erwähnt (siehe Kapitel 3.4), wurden die drei *Smart Home* Standards *EnOcean*, *Z-Wave* und *ZigBee* für ROS-E betrachtet und für die Integration in ROS-E ausgewählt. In der nachfolgende Tabelle 2 werden wichtige Eigenschaften der Standards vergleichend betrachtet. [3]

Kategorie	***EnOcean***	***ZigBee***	***Z-Wave***
Netz & Reichweite	Weit, Einsatz von Repeatern möglich [3]	Kurz, vermaschtes Netz erlaubt Weiterleitung [30]	Weit, vermaschtes Netz erlaubt Weiterleitung [30]
Robustheit	sehr robust [30]	weniger robust [3]	robust [3] [30]
Energiebedarf	sehr gering [26]	gering [39]	gering
Datensicherheit	*AES-128* [27]	*AES*-128 [40]	*AES*-128 + *ECDH* [31]
Marktanteil & Verbreitung	Geringer Anteil, Fokus auf Energy Harvesting [3] [28] [29]	Sehr großer Anteil, sehr viele verschiedene Komponenten [3]	Großer Anteil, viele verschiedene Komponenten. [3]

Tabelle 2: Vergleich von EnOcean, ZigBee und Z-Wave

Aus dem obigen Vergleich gehen *EnOcean* und *Z-Wave* bezüglich der Verbindungsqualität als am besten geeignet hervor. Der Marktanteil von *ZigBee* und *Z-Wave* überwiegt verglichen mit *EnOcean*. Alle Standards ermöglichen eine Absicherung der Übertragung, obgleich in unterschiedlicher Qualität. *EnOcean* zeichnet sich vor allem durch den sehr geringen bis nicht vorhandenen[3] Energiebedarf aus, bietet jedoch auch die meisten Produkte nur in diesem Bereich an.

Vor allem *EnOcean* und *Z-Wave* überzeugen in diesem Vergleich, jedoch kann der *ZigBee*-Standard aufgrund der vielen kompatiblen *Smart Home* Geräte nicht ignoriert werden. Daher werden alle drei Standards für eine möglichst große Kompatibilität mit *Smart Home* Geräten für die Anbindung eingesetzt (vgl. *R19*, *R20* und *R21*).

5.1.3 Verbindung zu Aktoren & Sensoren

Ein weiteres Ziel der DevBox ist die Anbindung von Aktoren und Sensoren. Dabei ist darauf zu achten, dass dieselben Schnittstellen zur Verfügung stehen, die auch von ROS-E angeboten werden. Dadurch wird prinzipiell gewährleistet, dass entwickelte Schaltungen auch mit ROS-E kompatibel sind und auch auf das System von ROS-E übertragbar sind.

Durch ROS-E wird die Anbindung von Aktoren und Sensoren grundsätzlich per *GPIO*, *I²C*, *SPI* und *UART* (Universal Asynchronous Receiver/Transmitter) unterstützt. Per *GPIO* können einfache Komponenten wie eine *LED* oder ein Taster angebunden werden. Hardwarekomponenten, die eine Datenanbindung benötigen können per *I²C*, *SPI* oder *UART* kommunizieren. Die folgende Tabelle stellt letztere vergleichend dar. [1] [33] [41]

Kategorie	***I²C***	***SPI***	***UART***
Maximale Anzahl von Geräten	bis zu 128 Geräte[4] [20]	Viele, abhängig von Chip Select-Leitungen [21]	Controller und Peripheral (2) [42]
Komplexität bei steigender Geräteanzahl	Einfach, da jedes Gerät einzeln adressierbar ist. [20]	Komplex, da jedes Gerät eine Chip Select-Leitung benötigt. [21]	N/A, da die Geräteanzahl nicht steigen kann [42]
Geschwindigkeit der Übertragung	schnell mit bis zu 5 Mbit/s [20]	potenziell sehr schnell, aber	moderat, aber abhängig von

[3] Im Sinne des *Energy Harvesting*.
[4] Es wird von 7-Bit-Adressen inklusive reservierter Adressen ausgegangen.

		abhängig von beteiligten Geräten [21] [43]	beteiligten Geräten [42]
Duplex	Halbduplex [20]	Vollduplex [21]	Vollduplex [42]
Anzahl der Leitungen	2 [20]	4 [21]	2 [42]
Anwendungen	Anbindung von Sensoren und Aktoren sowie anderen integrierten Schaltkreisen [20]	Anbindung von Sensoren [43]	Debugging, Anbindung von *Bluetooth* oder GPS [42] [43]

Abbildung 10: Vergleich von I²C, SPI und UART

Ähnlich wie bei *Smart Home* Protokollen hängt die Anschlussmethode meist von der entsprechenden Hardwarekomponente ab. Daher sollen mehrere Protokolle für diesen Zweck eingebunden werden. *I²C* und *SPI* unterstützen beide die Anbindung mehrerer Geräte und eine schnelle Kommunikation. Beide Protokolle werden von vielen Aktoren und Sensoren verwendet. *UART* ist im Vergleich langsamer und weniger für die Anbindung von Sensoren oder Aktoren verbreitet. Daher wird eine solche Verbindung via *GPIO* sowie *I²C* und *SPI* ermöglicht, wie es in den Anforderungen definiert wurde (vgl. *R6*, *R9* und *R11*). [41] [43]

5.1.4 Verbindung zu Microcontrollern

Neben einer Anbindung von Aktoren und Sensoren kann der Einsatz eines Microcontrollers (vgl. *R13*) hilfreich sein, um beispielsweise Logik oder Berechnungen auf diesen auszulagern oder zeitkritische Abläufe unabhängig von der DevBox oder ROS-E zu steuern. Nachfolgend werden die Möglichkeiten für die Verbindung zwischen der DevBox und Microcontrollern besprochen.

Die meisten Microcontroller verfügen über gängige Hardwareschnittstellen, um die Anbindung und Verwendung von Aktoren und Sensoren zu ermöglichen. Daher können dieselben Hardwareschnittstellen genutzt werden, mit denen Aktoren und Sensoren an die DevBox angebunden werden.

Zusätzlich zur Kommunikation ist auch das Überspielen von Programmen auf angeschlossene Microcontroller gefordert (vgl. *R14*). Bei vielen Microcontroller ist dies per *USB* und/oder eine *SPI*-Verbindung möglich. Daher wird zusätzlich eine *USB*-Schnittstelle für die Anbindung von Microcontroller vorgesehen (vgl. *R17*). [44]

5.1.5 Verbindung zu anderen Geräten

Weitere Geräte, die nicht zu den zuvor genannten Gerätegruppen gehören, sollen ebenfalls eine Möglichkeit für die Anbindung an die DevBox erhalten. Dafür soll die *USB*-Schnittstellen als eine weitverbreitete Verbindungsmöglichkeit für *Peripheriegeräte* eingesetzt werden.

5.2 Hardware der DevBox

Nachdem die verschiedenen Verbindungen der DevBox betrachtet und entsprechende Technologien ausgewählt wurden, wird in diesem Abschnitt ein Konzept für die Hardware der DevBox entwickelt. Die folgende Abbildung 11 fasst die Ergebnisse des vorangehenden Abschnitts noch einmal kurz zusammen.

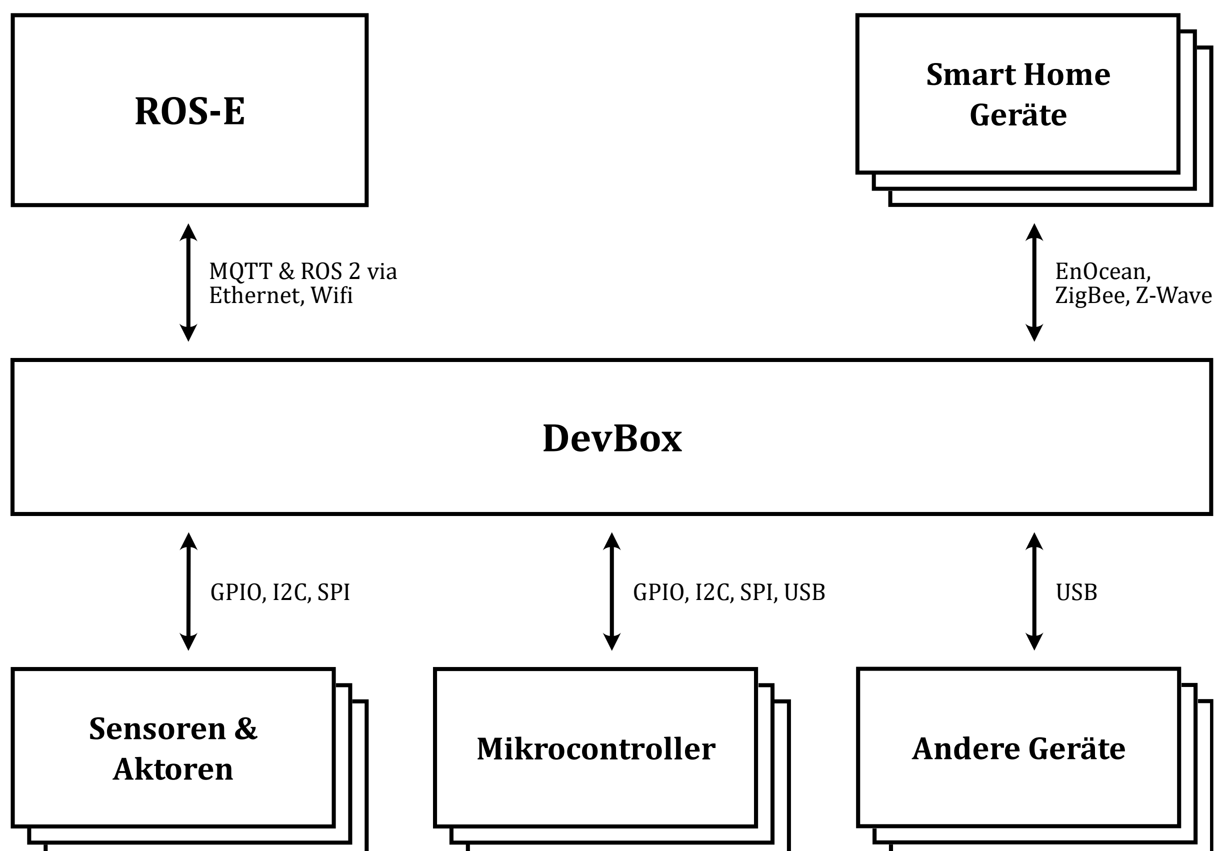

Abbildung 11: Konzept für die Verbindungen der DevBox

Basierend auf diesen Ergebnissen werden folgend Hardwarekomponenten für die DevBox ausgewählt. Anschließend wird der Aufbau der DevBox aus diesen Komponenten konzipiert.

5.2.1 Auswahl & Zusammenhang der Hardwarekomponenten

Die DevBox muss eine Vielzahl an Hardwareschnittstellen wie *GPIO*, *I²C*, *SPI*, *USB*, *Wi-Fi* und *Ethernet* unterstützen. Als Basis der DevBox muss daher ein Computersystem dienen, welches diese Schnittstellen vereint. Da diese Hardwareschnittstellen für die DevBox von ROS-E übernommen wurden, kann das Computersystem von ROS-E gleichermaßen für die DevBox verwendet werden. Somit wird eine der von ROS-E sehr ähnlichen Entwicklungsumgebung geschaffen. Daher wird ein Raspberry Pi 4 Model B (vgl. Kapitel 2.1) als zentrale Steuereinheit für die DevBox verwendet.

Neben den erwähnten Hardwareschnittstellen, müssen auch *EnOcean*, *ZigBee* und *Z-Wave* in die DevBox integriert werden. In dem zuvor erwähnten Praktikum zu diesem Thema wurden für diesen Zweck *USB*-Adapter für diese Funkstandards (vgl. *R19*, *R20* und *R21*) ausgewählt. Diese Adapter enthalten die notwendige Hardware zum Senden und Empfangen über das jeweilige Protokoll und können per *USB* angesprochen werden. [3]

Darüber hinaus werden ein Mikrocontroller (vgl. *R13*) und eine *Steckplatine* (vgl. *R15*) als Bestandteil der DevBox gefordert. Als Mikrocontroller wird ein Arduino eingesetzt, da dieser auch bei ROS-E verwendet wird und dessen Umgang den Entwicklern von ROS-E bekannt ist.

Der folgende Blockschaltplan in Abbildung 12 zeigt diese Komponenten und wie sie zusammenhängen.

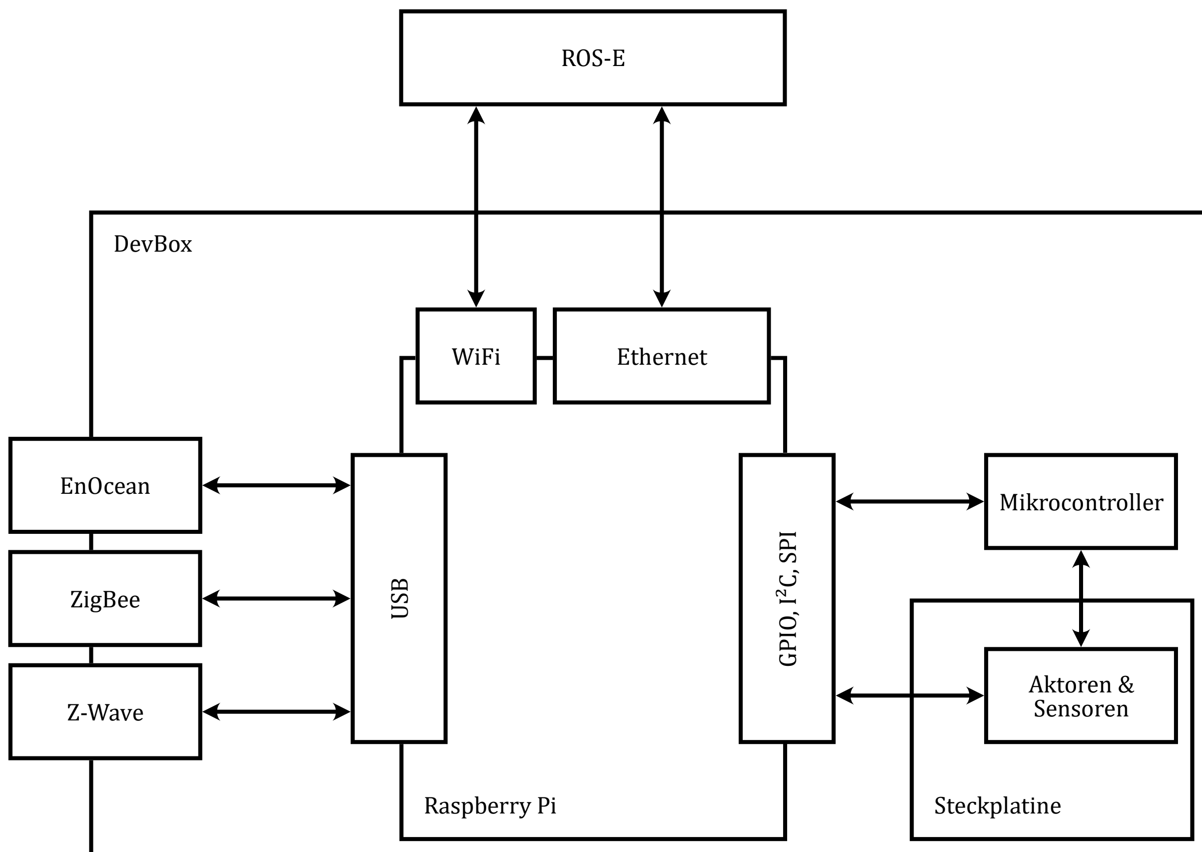

Abbildung 12: Zusammenhang der Hardwarekomponenten der DevBox

Der Raspberry Pi vereinigt als zentrale Steuereinheit alle wichtigen Schnittstellen. Die DevBox kann über *Wi-Fi* oder *Ethernet* mit einer ROS-E verbunden werden. Über *GPIO*, *I²C* und *SPI* kann mit dem Arduino als Mikrocontroller und Komponenten auf dem Breadboard zugegriffen werden. Die *USB*-Schnittstelle ermöglicht die Verwendung der Adapter für *ZigBee* und *Z-Wave* oder anderer *USB*-Geräte.

5.2.2 Aufbau & Design der DevBox

Als nächstes wird das Gehäuse und der innere Aufbau der DevBox konzipiert. In diesem Gehäuse werden die verschiedenen Hardwarekomponenten angeordnet. Dabei steht vor allem die Funktionalität und Zugänglichkeit für Verwender der DevBox im Mittelpunkt.

5.2.2.1 Entwurf des inneren Aufbaus

Wie aus den Anforderungen schon hervorgeht, beinhaltet die DevBox eine große *Steckplatine* (vgl. *R15* und *R16*), auf der verschiedene Sensoren, Aktoren und Schaltungen erprobt werden können. Die anderen Komponenten sollten gut erreichbar um die *Steckplatine* angeordnet sein, damit die zur Verfügung gestellten Schnittstellen einfach zu erreichen und zu benutzen sind. Mit diesen Überlegungen wurden mehrere Entwürfe für den Aufbau der DevBox erstellt. Im Folgenden wird der finale Entwurf vorgestellt.

Der Kern der DevBox bildet die *Steckplatine*, an die auf drei Seiten jeweils eine für die DevBox entworfene Platine angrenzt, wie es die folgende Abbildung 13 zeigt. Außerdem ist der Raspberry Pi und ein Arduino Nano zu erkennen. Jede Platine erlaubt den Zugriff auf bestimmte Schnittstellen des Raspberry Pi oder Arduino Nano. Die Platinen befinden sich nah an der *Steckplatine* und sind somit gut zugänglich für eine Verbindung mit Hardwarekomponenten, die sich auf der *Steckplatine* befinden.

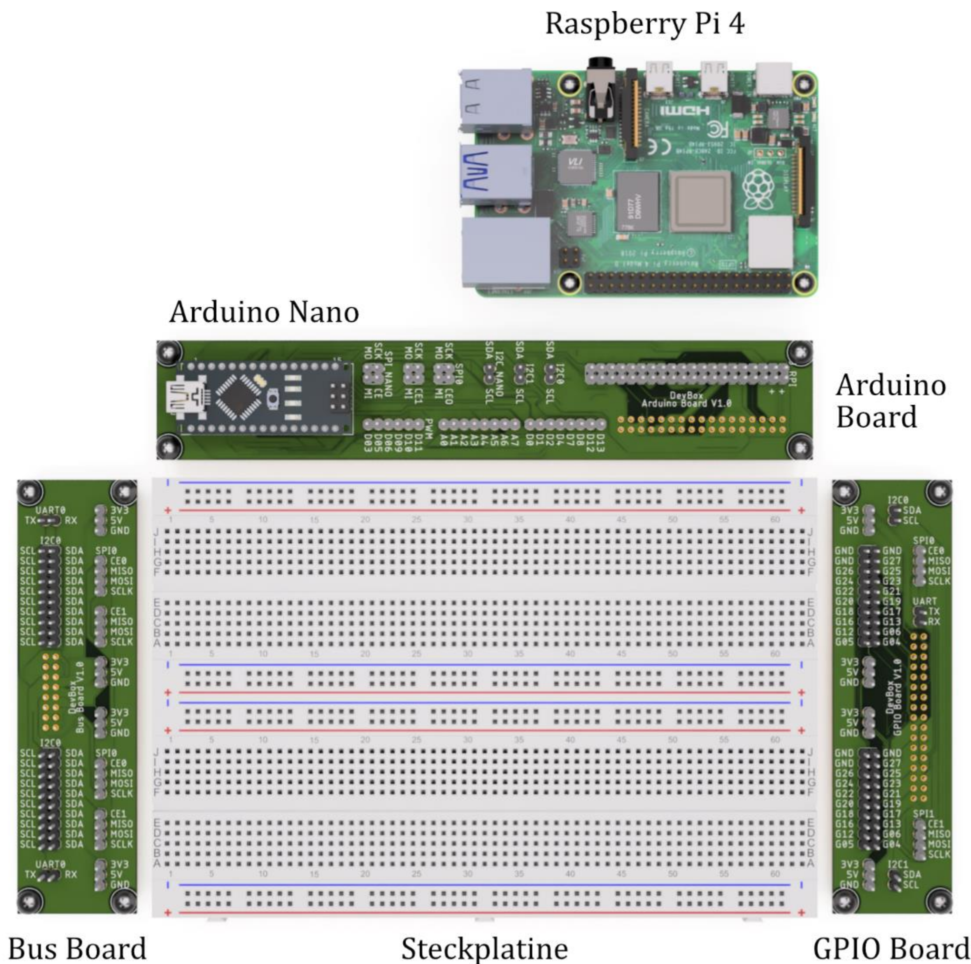

Abbildung 13: Entwurf für den inneren Aufbau der DevBox

Die dargestellte *Steckplatine* besteht aus zwei aneinander gesteckten *Steckplatine*n mit einer Länge von 165 mm sowie einer Breite von 55 mm und erreicht mit 181,5 cm^2 die geforderte Mindestgröße (siehe Anforderung R16).

Die Platine oberhalb der *Steckplatine* – beschriftet als „DevBox Arduino Board“ – führt den Arduino Nano und stellt dessen Anschlüsse über Stiftleisten bereit. Zudem kann über das Stecken einer Kabelverbindung auf der Platine der Arduino mit dem Raspberry Pi per *I²C* oder *SPI* verbunden werden.

Die linke Platine – beschriftet als „DevBox Bus Board“ – stellt Stiftleisten für verschiedene Kommunikationsschnittstellen wie *I²C*, *SPI* und *UART* bereit. Zu erkennen ist, dass die

Stiftleisten nach Funktion gruppiert sind und mehrere Pins mit derselben Signalleitung verbunden sind, um die Anbindung mehrerer *I²C* oder *SPI*-Teilnehmer zu vereinfachen.

Auf der rechten Platine – beschriftet als „DevBox *GPIO* Board“ – sind Stiftleisten für die *GPIO*-Pins des Raspberry Pi untergebracht. Diese wurden ebenfalls logisch gruppiert und mehrfach ausgeführt, um die Verwendung von möglichst jeder Position auf der *Steckplatine* zu erleichtern. Auf sowohl der linken als auch der rechten Platine sind Stiftleisten für die Spannungsversorgung vom Raspberry Pi vorhanden.

Die drei externen Platinen werden per Flachbandkabel mit dem Raspberry Pi verbunden. Dabei dient das „Arduino Board“ als Verteiler für die anderen Platinen. Die entsprechenden Stiftleisten für diese Kabelverbindungen sind auf den Unterseiten der Platinen angebracht, sodass die Kabel unterhalb der Platinen geführt werden können und bei der Verwendung nicht stören.

5.2.2.2 Entwurf des Gehäuses

Das Gehäuse der DevBox muss so entworfen werden, dass es die soeben konzipierte Anordnung der inneren Komponenten einfasst. Die Abbildung 14 zeigt zunächst die Basis der DevBox, auf der die *Steckplatine* mit den drei Platinen und der Raspberry Pi angebracht sind.

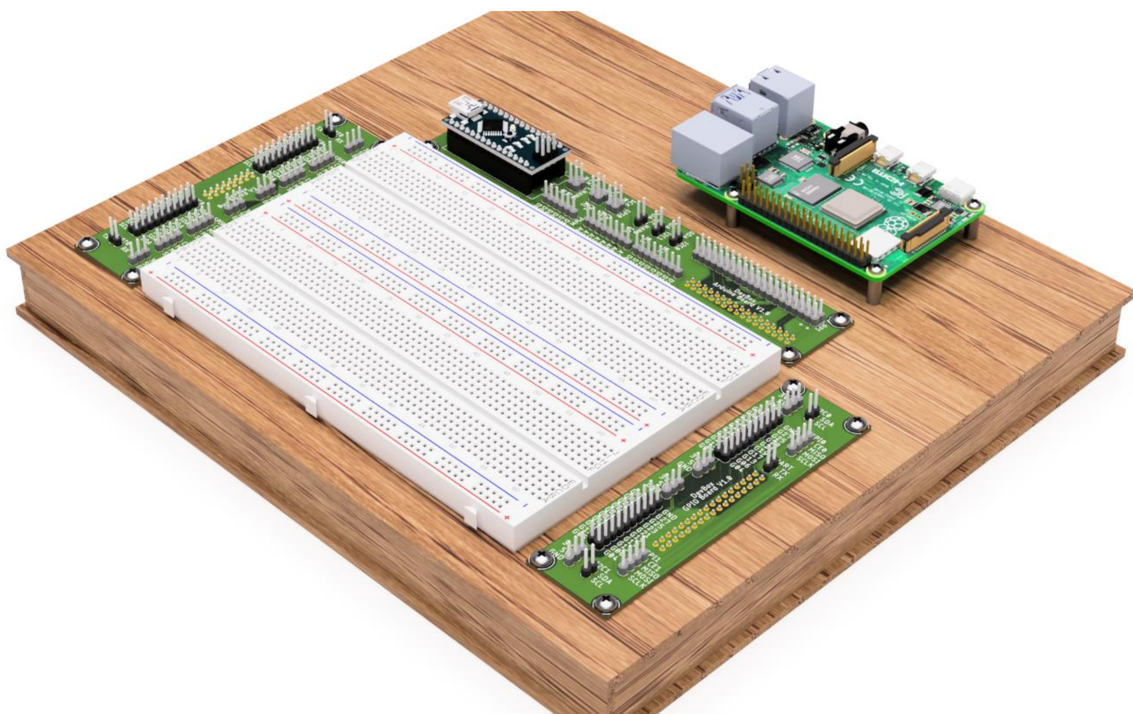

Abbildung 14: Entwurf für die Basis der DevBox

Die Platinen und der Raspberry Pi sollen mit Schrauben auf der oberen Platte angebracht werden, wohingegen die *Steckplatine* an der entsprechenden Stelle aufgeklebt wird. Die

Kabel für die Verbindung der Platinen werden über einen doppelten Boden in der Basis geführt. Der Zwischenraum der Basis kann auch für das Verstauen von Bauelementen verwendet werden (siehe Anforderung R3). Für den Transport und allgemeinen Schutz nach Anforderungen R1 und R2 können auf diese Basis noch Wände und eine Abdeckung aufgebaut werden, wie es in der folgenden Abbildung 15 zu sehen ist.

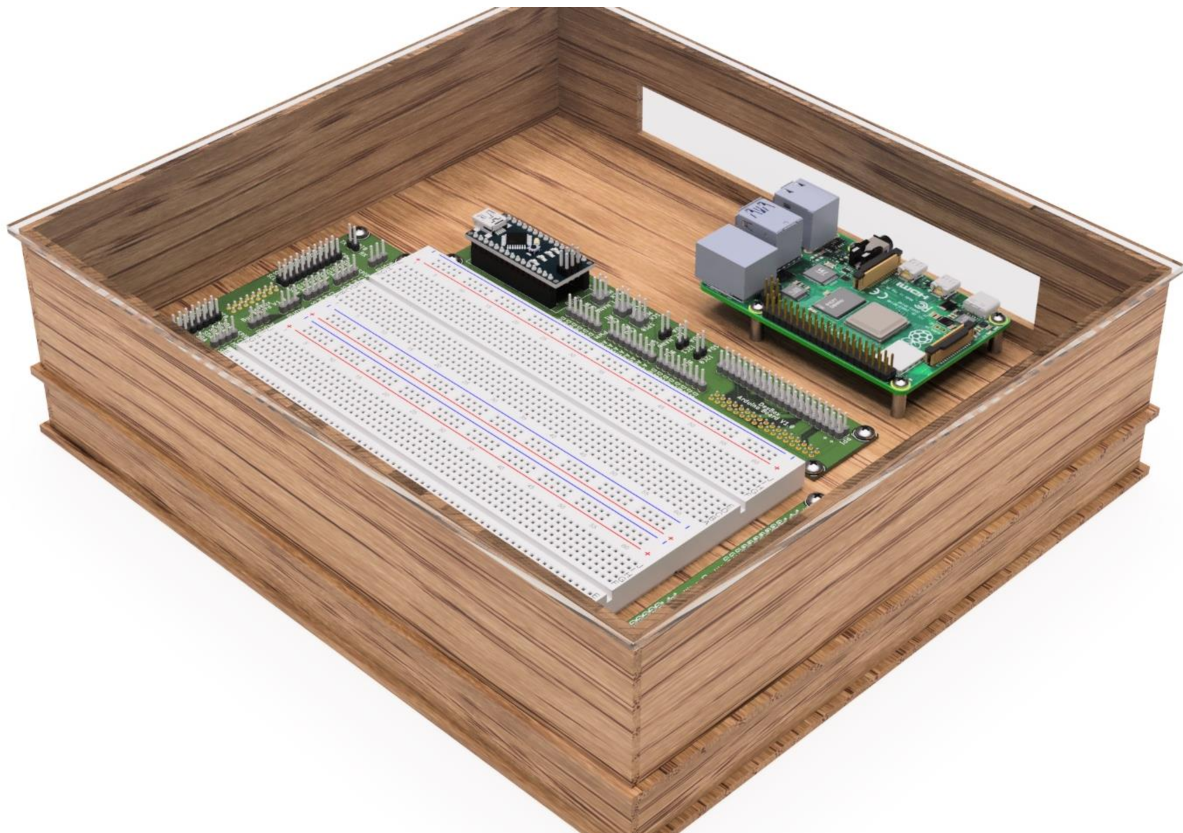

Abbildung 15: Entwurf für das Gehäuse der DevBox

Die Basis und die oberen Wände sollen aus Holz gefertigt werden, während für die Abdeckung Plexiglas eingesetzt werden soll, um auch im geschlossenen Zustand ein Blick ins Innere zu ermöglichen. Eine Auslassung der Wand in der Nähe des Raspberry Pi erlaubt das Anschließen von Kabel an diesen im geschlossenen Zustand. Mithilfe von Schrauben oder Magnetverschlüssen sollen die Wände und die Abdeckung mit der Basis verbunden werden.

5.2.3 Zusammenfassung der Materialien & Kosten

Für den Bau einer DevBox werden neben einem Raspberry Pi und einem Arduino Nano die drei entworfenen Platinen und die *Steckplatine* benötigt. Zudem werden Holz und Plexiglas für den Bau der Hülle verwendet. Für eine DevBox belaufen sich die Kosten damit auf ungefähr 200 €. Eine detaillierte Übersicht der Materialien und deren Kosten befindet sich unter *9.1 Materialliste für den Bau einer DevBox* im Anhang.

5.3 Software für die DevBox

Auf dem Entwurf für die Hardware der DevBox und der entwickelten Architektur für die Verbindungen der DevBox aufbauend wird in diesem Kapitel die Software für die DevBox konzipiert. Die wesentliche Aufgabe der Software ist die Kommunikation zwischen ROS-E, der DevBox und der daran angeschlossenen Geräte zu ermöglichen. In den folgenden Unterkapiteln wird zunächst das Konzept zur Ermöglichung dieser Kommunikation erläutert.

5.3.1 Kommunikation mit Hardwarekomponenten

Mithilfe von *ROS* wird der Zugriff von ROS-E auf die Hardware der DevBox ermöglicht. Auf *ROS* aufbauend werden für die verschiedenen Schnittstellen – *GPIO*, *I²C* und *SPI* – jeweils Knoten für diese Zugriffe entwickelt. Die folgende Abbildung 16 zeigt diese Knoten als „Interface Nodes“ und wie diese Knoten mit anderen Softwarekomponenten kommunizieren.

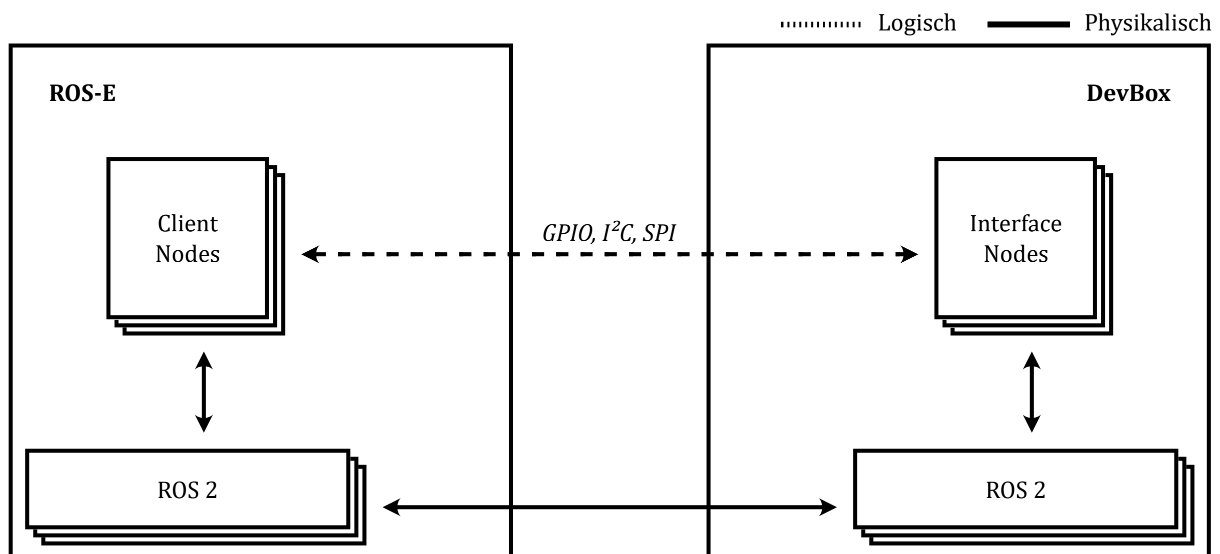

Abbildung 16: Softwarekomponenten für die Kommunikation über ROS

Die „Interface Nodes“ greifen auf die zuvor beschriebene physikalische Verbindung über den *ROS*-Protokollstapel zu. Auf der Seite von ROS-E kann über diese Verbindung mit den „Interface Nodes“ kommuniziert und so auf die angebotenen Funktionen zugegriffen werden. Für den Zugriff auf *GPIO*, *I²C* und *SPI* werden verschiedene *ROS*-Topics oder Services angeboten, die jeweils eine Teilfunktion implementieren. Die in der obigen Abbildung als „Client Nodes“ bezeichneten *ROS*-Knoten sind Knoten, die auf Topics und Services der „Interface Nodes“ auf Seiten der Verwender zugreifen. Im Folgenden werden die konzipierten Topics und Services erläutert. Zusätzlich werden analog die Programmierschnittstellen (*API*) der DevBox für die Erfüllung von Anforderung *R30* konzipiert.

5.3.1.1 General Purpose Input/Output

Für den Zugriff auf Hardware via *GPIO* ist in erster Linie ein *ROS*-Service zum Auslesen des an einem beliebigen Pin anliegenden Signals (vgl. R8) und ein Service zum Einstellen des Ausgangsignals eines beliebigen Pins (vgl. R7) notwendig. Zudem kann ein Service nützlich sein, mit dem das eingestellte Ausgangssignal abgefragt werden kann. Die folgende Tabelle 3 beschreibt diese Services.

Name	**Beschreibung**
`gpio/get_input`	Ein Aufruf an diesen Service liefert, unter Angabe eines *GPIO*-Pins, das Logiklevel des an den Pin anliegenden Signals, also *LOW* oder *HIGH*, zurück.
`gpio/get_output`	Ein Aufruf an diesen Service liefert, unter Angabe eines *GPIO*-Pins, das für die Ausgabe eingestellte Logiklevel, also *LOW* oder *HIGH*, zurück.
`gpio/set_output`	Ein Aufruf an diesen Service stellt, unter Angabe eines *GPIO*-Pins und dem einzustellenden Wert (*LOW* oder *HIGH*), den entsprechenden Pin auf diesen Wert ein.

Tabelle 3: ROS-Services für den Zugriff auf GPIO

In der Regel müssen *GPIO*-Pins für die Ausgabe oder Eingabe vor der entsprechenden Verwendung konfiguriert werden. Da ein Aufruf eines der beschriebenen Services bereits den Verwendungszweck des angegebenen Pins hinreichend definiert, wird diese Konfiguration, sofern notwendig, auch beim Aufruf des jeweiligen Service durchgeführt. Dadurch wird dem Verwender der Services eine weitere Konfiguration abgenommen.

Neben einfachen Schreib- und Lesezugriffen auf die *GPIO*-Pins, kann es für Anwendungen auch interessant sein, wann ein *GPIO*-Pin den Zustand wechselt. Das kann entweder von `LOW` zu `HIGH` (`RISING`) oder von `HIGH` zu `LOW` (`FALLING`) geschehen. Diese beiden Ereignisse werden mithilfe eines *ROS*-Topics veröffentlicht, sodass Anwendungen auf diese Ereignisse reagieren können, ohne laufend den aktuellen Zustand abfragen zu müssen. Die folgende Tabelle 4 beschreibt die dafür notwendigen *ROS*-Topics und Services. [19]

Name	Beschreibung
`gpio/events`	Auf diesem Topic werden Ereignisse im Fall von steigenden oder fallenden Flanken bei *GPIO*-Pins veröffentlicht. Jedes Ereignis gibt dabei an, um welche Art von Ereignis es sich handelt, den betroffenen *GPIO*-Pin sowie den Zeitpunkt, zu dem das Ereignis auftrat.
`gpio/subscribe`	Ein Aufruf dieses Service, unter Angabe eines *GPIO*-Pins und der Art der Flanke (`FALLING` oder `RISING`), konfiguriert den entsprechenden Pin für die Erkennung von Flanken.

Tabelle 4: ROS-Topic und Service für GPIO-Ereignisse

Basierend auf diesen *ROS*-Topics und Services kann die entsprechende *API* für die Benutzung durch Verwender entworfen werden. Das folgende Klassendiagramm in Abbildung 17 zeigt eine Vorlage für die Klasse `GpioClient`, die geeignete Methoden für das Ansprechen der *ROS*-Topics und Services für Zugriffe auf *GPIO* enthält.

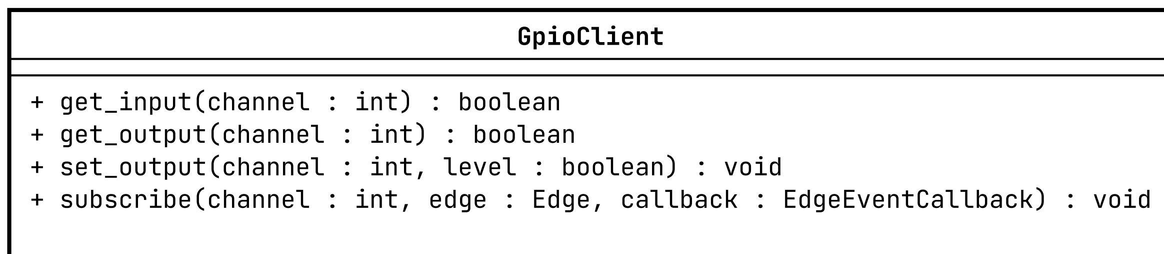

Abbildung 17: Klassendiagramm der GpioClient-Klasse

5.3.1.2 Inter-Integrated Circuit

Um die Kommunikation mit Hardwarekomponenten über *I²C* zu ermöglichen (vgl. R10), muss der Lese- und Schreibzugriff auf den entsprechenden *I²C*-Bus möglich sein. Diese Funktion wird jeweils durch einen *ROS*-Service übernommen. Die nachfolgende Tabelle 5 beschreibt diese Services. [20] [45]

Name	Beschreibung
`i2c/read`	Ein Aufruf dieses Service liefert, unter Angabe der Adresse eines Zielbusteilnehmers und einem Register, von diesem Teilnehmer per *I²C* aus dem Register empfangende Daten.
`i2c/write`	Ein Aufruf dieses Service sendet, unter Angabe der Adresse eines Zielbusteilnehmers, eines Registers und Daten, diese Daten per *I²C* an den Teilnehmer.

Tabelle 5: ROS 2 Services für die Kommunikation über I²C

Es wird davon ausgegangen, dass die DevBox als *I²C*-Controller auftritt und somit die Kommunikation steuert, sodass Lese- und Schreiboperationen möglich sind. Als Zielbusteilnehmer werden somit *I²C*-Targets bezeichnet, die mit dem *I²C*-Bus verbunden sind und dem durch die DevBox vorgegebenen Kommunikationsablauf folgen. [20]

Die folgende Abbildung 18 zeigt ein Klassendiagramm der `I2cClient`-Klasse, die als *API* für Verwender der *I²C*-Schnittstelle der DevBox dient. Die enthaltenen Methoden der Klasse vereinfachen den Zugriff auf die zuvor beschriebenen *ROS*-Services.

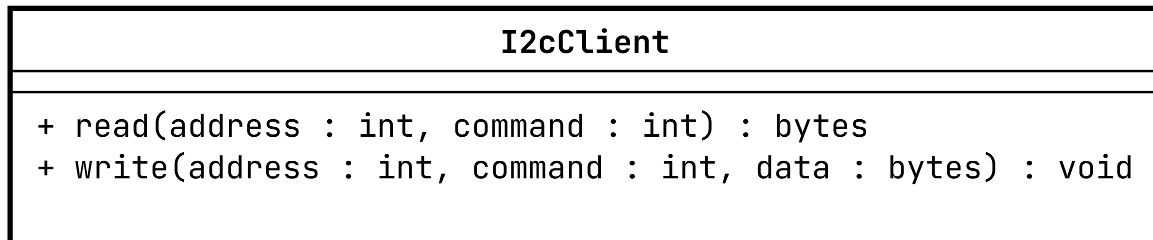

Abbildung 18: Klassendiagramm der I2cClient-Klasse

5.3.1.3 Serial Peripheral Interface

Die Kommunikation mit Hardwarekomponenten, die per *SPI* an die DevBox angeschlossen sind (vgl. R12), erfordert ebenfalls den Lese- und Schreibzugriff auf den *SPI*-Bus der DevBox. Für diesen Zweck wird je ein *ROS*-Service konzipiert, wie in der folgenden Tabelle 6 beschrieben. [21]

Name	Beschreibung
`spi/read`	Ein Aufruf dieses Service liefert, unter Angabe des *SPI*-Geräts, von diesem Gerät über *SPI* empfangende Daten.
`spi/write`	Ein Aufruf dieses Service sendet, unter Angabe des *SPI*-Geräts und Daten, diese Daten per *SPI* an das Gerät.

Tabelle 6: ROS-Services für die Kommunikation über SPI

Die *ROS*-Services sind so konzipiert, dass die DevBox als *SPI*-Controller auftritt und somit den Kommunikationsablauf des *SPI*-Busses steuert. Alle angeschlossenen *SPI*-Geräte befinden sich daher in der Rolle eines *SPI*-Peripherals und befolgen Steuerbefehle der DevBox. [21]

Für Verwender wird eine *API* in Form der `SpiClient`-Klasse entworfen, dessen Methoden die einfache Verwendung der oben beschriebenen *ROS*-Services erlaubt. Das Klassendiagramm in der folgenden Abbildung 19 zeigt diese Klasse.

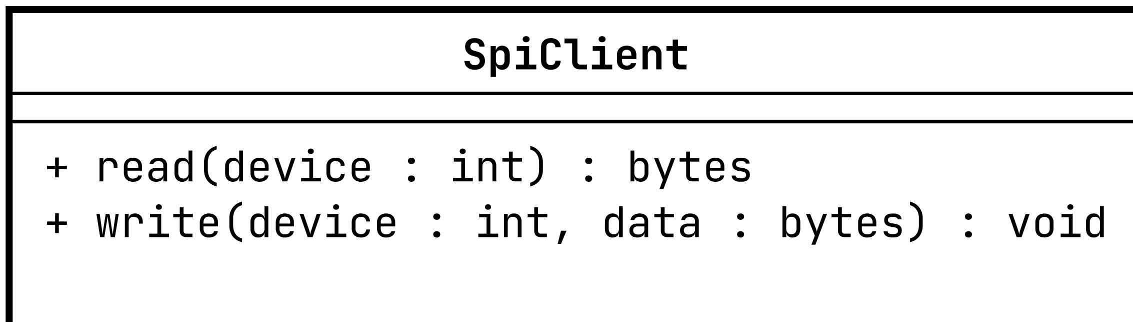

Abbildung 19: Klassendiagramm der SpiClient-Klasse

5.3.2 Kommunikation mit Smart Home Geräten

Die Kommunikation mit *Smart Home* Geräten, die via *EnOcean*, *ZigBee* oder *Z-Wave* mit der DevBox verbunden sind, wird mithilfe des *MQTT*-Protokolls ermöglicht. Dabei kommunizieren die Anwendungen nie direkt miteinander, sondern über einen *MQTT*-Broker. Die folgende Abbildung 20 zeigt das Konzept für einen *MQTT*-Broker und andere Softwarekomponenten, die für die Kommunikation notwendig sind.

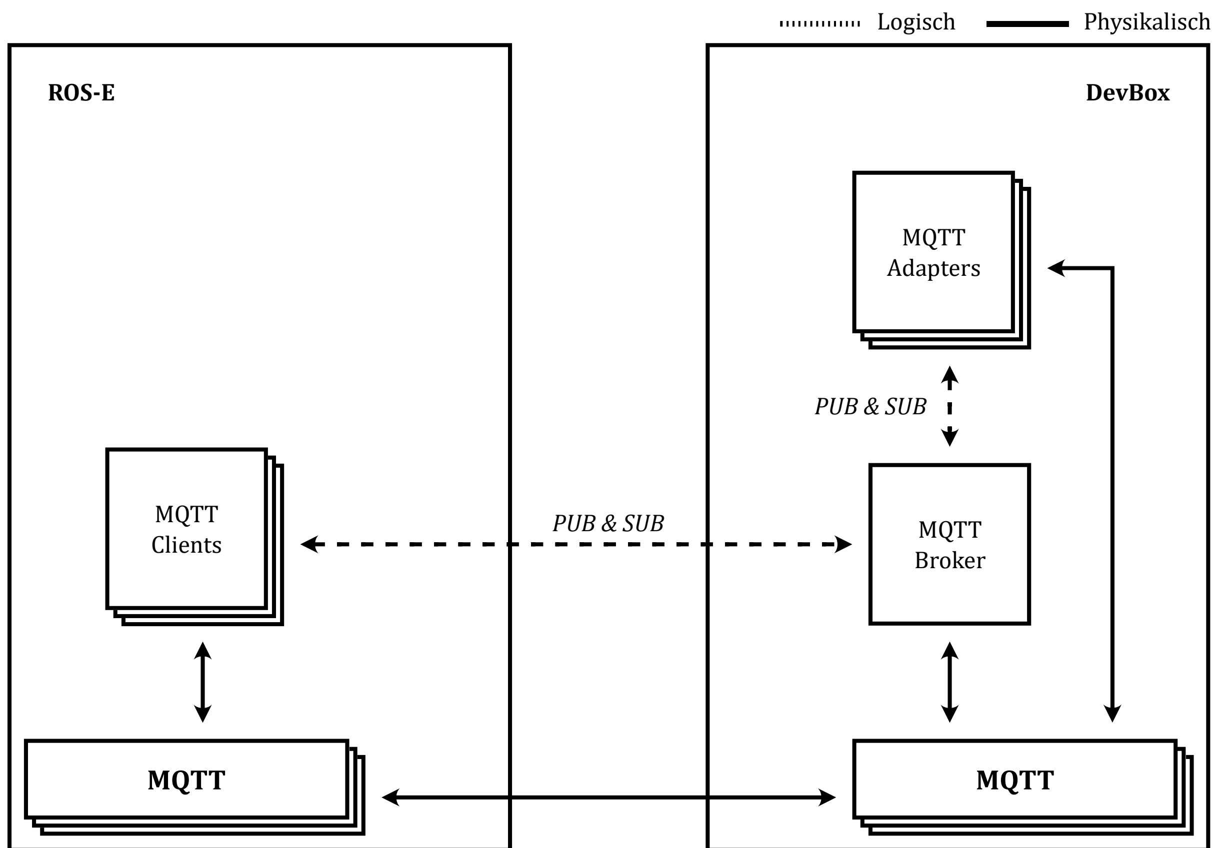

Abbildung 20: Softwarekomponenten für die Kommunikation über MQTT

Der *MQTT*-Broker wird von der DevBox den Verwendern bereitgestellt. Die Anbindung an die externen *Smart Home* Protokolle *EnOcean*, *ZigBee* und *Z-Wave* soll seitens der DevBox durch entsprechende Adapter („*MQTT*-Adapters") umgesetzt werden. Verwender der DevBox können über eigene *MQTT*-Clients auf ROS-E auf den *MQTT*-Broker der DevBox und damit auch auf die angebundenen *Smart Home* Geräte zugreifen. Die *MQTT*-Topics und der Aufbau der darüber ausgetauschten Nachrichten werden durch den Adapter des jeweiligen *Smart Home* Protokolls bestimmt.

5.3.2.1 Message Queuing Telemetry Transport

Damit ein *MQTT*-Client von außerhalb der DevBox – beispielsweise von ROS-E – eine Verbindung mit dem *MQTT*-Broker herstellen kann, muss dieser Client mindestens die *IP-Adresse* der DevBox und den *TCP*-Port für den Broker kennen. Da diese Informationen nicht trivial von außen zu ermitteln sind, wird ein zusätzlicher *ROS*-Service diese Informationen bereitstellen. Die folgende Tabelle 7 beschreibt diesen Service.

Name	Beschreibung
`mqtt/get_broker_info`	Ein Aufruf dieses Service liefert aktuelle Informationen zu dem *MQTT*-Broker der DevBox und beinhaltet insbesondere die *IP-Adresse* und den *TCP*-Port.

Tabelle 7: ROS-Service für MQTT-Informationen

Entsprechend des konzipierten *ROS*-Service wird mit der `MqttClient`-Klasse eine *API* für Verwender angeboten, mit dem Ziel, den Zugriff auf die *MQTT*-Schnittstelle zu vereinfachen. Die folgenden Abbildung 21 zeigt ein Klassendiagramm der Klasse.

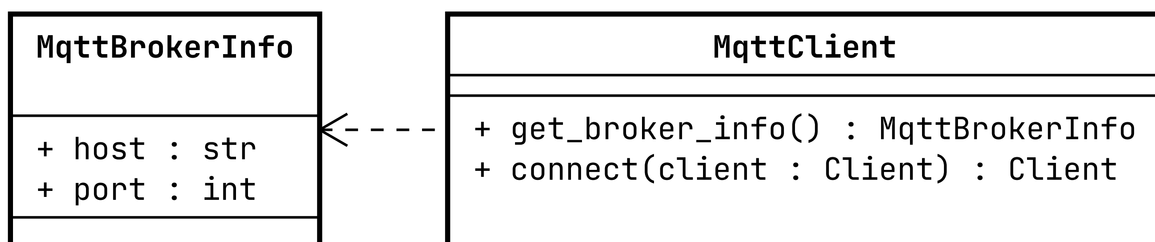

Abbildung 21: Klassendiagramm der MqttClient-Klasse

Das Klassendiagramm der Abbildung 21 zeigt darüber hinaus die Methode `connect`, die einen *MQTT*-Client (nicht die entworfene MqttClient-Klasse) mit dem *MQTT*-Broker der DevBox verbindet, um die Verwendung noch etwas einfacher zu machen.

5.3.3 Übergang von der DevBox zu ROS-E

Die DevBox soll vor allem die Evaluation und Entwicklung von Hardwarekomponenten für ROS-E unterstützen. Der nächste Schritt, nachdem ein zufriedenstellendes Ergebnis während dieser Entwicklung erreicht wurde, könnte der Einbau und die Verwendung mit ROS-E sein.

Um den Übergang von der DevBox zu ROS-E zu vereinfachen, sollte es möglich sein bereits entwickelte Software zumindest in großen Teilen wiederverwenden zu können. Aufgrund der bewusst geschaffenen Ähnlichkeit der Hardware von ROS-E und der DevBox (vgl. Kapitel 5.2.1 und 5.3.6), kann das Softwarekonzept für den Zugriff auf Hardwareschnittstellen gleichermaßen auf ROS-E angewendet werden. Dadurch wird die bestehende Lösung für den Zugriff auf den *I²C*-Bus (siehe Kapitel 2.4) ersetzt und um die weiteren Hardwareschnittstellen ergänzt. Im weiteren Verlauf wird jedoch, sofern nicht explizit ausgewiesen, nur die DevBox betrachtet.

5.3.4 Adressierung und Erkundung von DevBoxen

Eine DevBox kommuniziert mit einer ROS-E per *ROS* über ein *IP*-Netz. Beim Einsatz von mehreren DevBoxen im gleichen Netz muss zwischen den einzelnen DevBoxen unterschieden werden, um eine konkrete[5] DevBox gezielt ansteuern zu können. Jeder DevBox (und ROS-E) muss eine Adresse zugeordnet werden, die zumindest je Netz und Gerät einzigartig ist, um eine eindeutige Adressierung zu ermöglichen.

Eine der naheliegendsten Optionen wäre die Nutzung der *IP-Adresse*, die die DevBox verwendet, um mit ROS-E oder anderen Geräten im Netz zu kommunizieren, da diese Adresse bereits für die *IP*-Kommunikation die gewünschte Einzigartigkeit erfüllt. Jedoch ändert sich die *IP-Adresse* eines Geräts in der Regel mit dem Wechsel des Netzes. Zudem werden in den meisten Netzen *IP-Adressen* dynamisch per *DHCP* vergeben, sodass eine Adresse sich nach einiger Zeit ändern kann. Bei einer Änderung der *IP-Adresse* müsste auch die Adresse der DevBox angepasst werden, wodurch Benutzer der DevBox zur erneuten Konfiguration für diese DevBox gezwungen wären. Grundsätzlich können auch statische *IP-Adressen* vergeben werden, allerdings erfordert dies ebenfalls eine zusätzliche Konfiguration durch den Benutzer und ist eng an die Technologie des *IP*-Netzes gebunden. Die Adresse sollte nach den vorangehenden Überlegungen je DevBox unabhängig vom Netz sowie einzigartig sein und sich nicht ändern. Diese Eigenschaften werden von der *MAC-Adresse* erfüllt, die unter anderen bei *Wi-Fi*- und *Ethernet*-Geräten zum Einsatz kommt. [46] [47]

Damit Funktionen nur einer DevBox bezüglich des Zugriffs auf Hardwareschnittstellen und *Smart Home* Geräte angesprochen werden können, erhalten alle definierten ROS-Topics und Services ein Präfix in ihrem Namen. Das Präfix besteht aus der Adresse jener DevBox, die diese Topics und Services bereitstellt, sodass eine eindeutige Zuordnung zum Gerät durch die Adresse im Namen besteht.

Um eine DevBox möglichst schnell zu finden und diese im Anschluss zu adressieren, wird des Weiteren ein Mechanismus zur Erkundung von erreichbaren DevBoxen entwickelt. Die folgende Abbildung 22 zeigt für die Adressierung und Erkundung konzipierte Softwarekomponenten und deren Zusammenwirken.

[5] Durch eine gezielte Adressierung ist auch das Ansteuern mehrerer DevBoxen möglich.

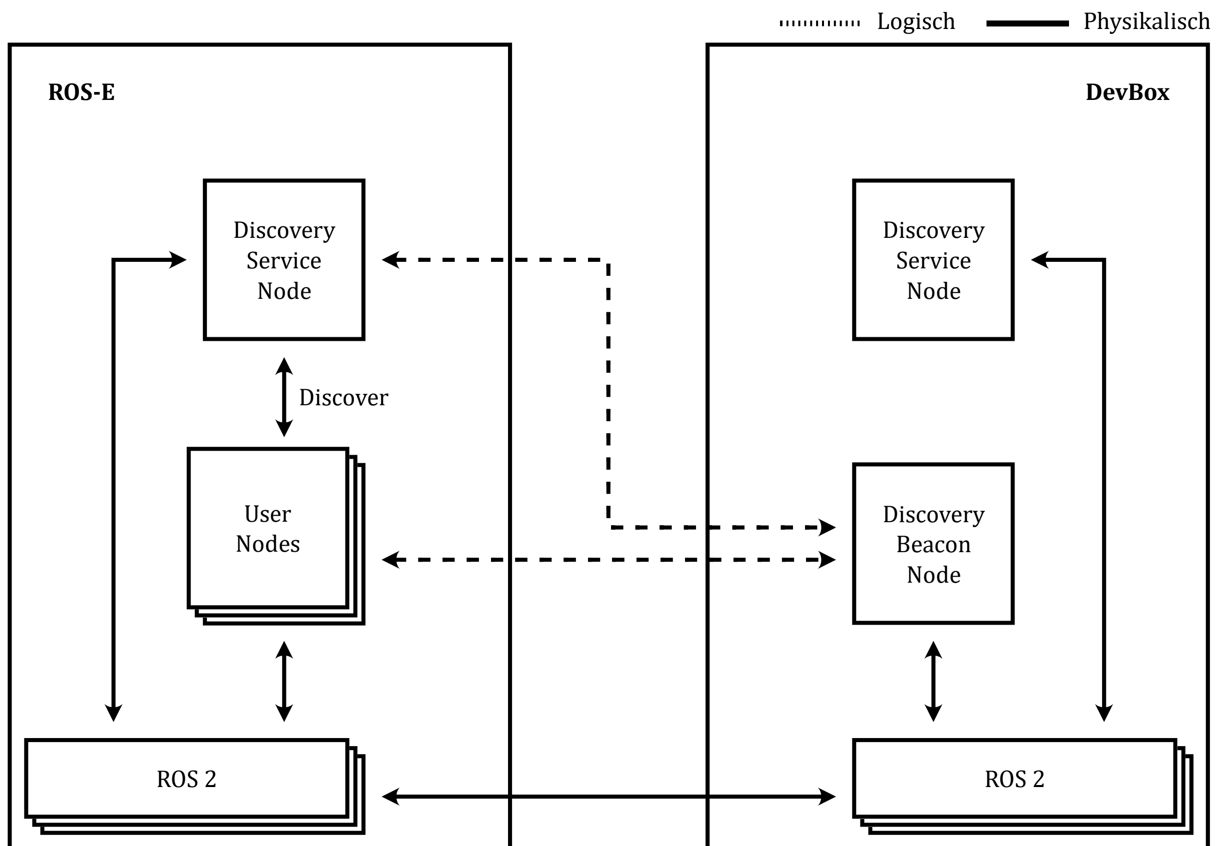

Abbildung 22: Softwarekomponenten für die Erkundung über ROS 2

Die Abbildung 22 zeigt auf beiden Seiten den Knoten „Discovery Service“. Dieser Knoten bietet einen *ROS*-Service, mit dem die Adresse der DevBox abgefragt werden kann und darüber Auskunft gibt, um welche Art von Gerät (eine DevBox oder ROS-E) es sich handelt (siehe Kapitel 5.3.3). Der Service wird in der nachfolgenden Tabelle 8 definiert.

Name	**Beschreibung**
`discovery/get_device_info`	Ein Aufruf dieses Service liefert Informationen über das jeweilige Gerät. Diese Informationen beinhalten insbesondere die *MAC-Adresse* des Geräts und ob es sich bei dem Gerät um eine DevBox oder eine ROS-E handelt.

Tabelle 8: ROS 2 Services für das Abfragen von Geräteinformationen

Damit eine Erkundung von DevBoxen ohne Vorwissen über die Adresse möglich ist, bedarf es noch einer weiteren Komponente. Der Knoten „Discovery Beacon“ in Abbildung 22 ist so konzipiert, dass dieser auf Erkundungsanfragen reagiert und mit den eigenen Geräteinformationen antwortet. Für das Austauschen von Erkundungsanfragen und -antworten wird je ein *ROS*-Topic definiert, wie es die folgende Tabelle 9 zeigt.

Name	Beschreibung
`/discovery/request`	Jede DevBox abonniert dieses Topic, um eine Erkundungsanfrage empfangen zu können. Durch eine Veröffentlichung auf dieses Topic wird ein neuer Erkundungsprozess gestartet.
`/discovery/response`	Eine DevBox, die zuvor eine Anfrage für den Erkundungsprozess erhalten hat, veröffentlicht auf diesem Topic eine entsprechende Antwort. Die Antwort beinhaltet insbesondere die Adresse der DevBox und dass es sich um eine DevBox handelt.

Tabelle 9: ROS-Topics für den Erkundungsprozess

Die Namen der Topics für den Erkundungsprozess in Tabelle 9 enthalten dabei kein Präfix mit einer Adresse, sodass ein Abonnement oder eine Veröffentlichung ohne die Kenntnis einer Adresse möglich ist. Um diesen Prozess zu vereinfachen, wird zusätzlich eine *ROS*-Action entwickelt. Die folgende Tabelle 10 definiert diese Action.

Name	Beschreibung
`discovery/discover`	Das Starten dieser Action, unter Angabe eines Timeouts, löst einen neuen Erkundungsprozess aus. Die Adresse und Art erkundeter Geräte werden sofort als Feedback übermittelt. Die Action endet nach dem Timeout und liefert alle erkundeten Geräte.

Tabelle 10: ROS-Action für den Erkundungsprozess

Um die exakte Funktionsweise und den Ablauf der Erkundung besser darzustellen, zeigt die folgende Abbildung 23 ein Flussdiagramm für diesen Prozess mit einer ROS-E und drei verschiedenen DevBoxen.

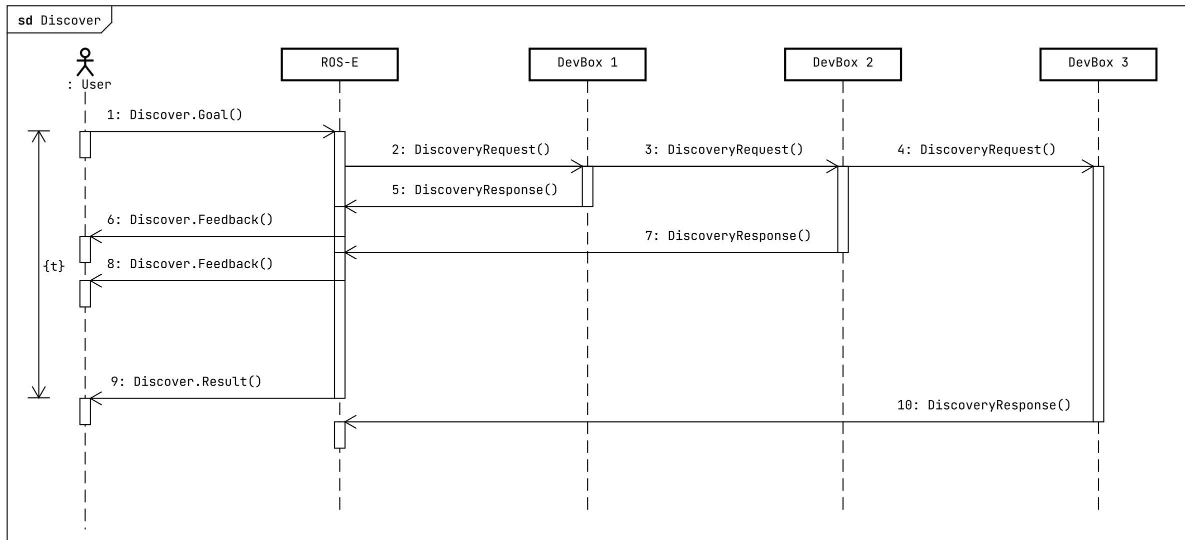

Abbildung 23: Flussdiagramm für den Entdeckungsprozess

Das Flussdiagramm in Abbildung 23 beginnt mit dem Start der Discover-Action (1). Anschließend wird auf dem Topic `/discovery/request` eine Erkundungsanfrage veröffentlicht, die von jeder DevBox empfangen wird (2, 3 und 4). Jede DevBox antwortet daraufhin, indem sie ihre Geräteinformationen auf dem Topic `/discovery/response` veröffentlicht (5, 7 und 10). Über das Topic empfangende Antworten werden direkt an den Nutzer als Feedback weitergeleitet (6 und 8). Nach einem vorgeschriebenen Timeout `t` endet die Action und liefert dem Nutzer alle im Zeitraum `t` gefundenen DevBoxen (9). Alle DevBoxen, die die Anfrage nicht empfangen haben oder keine Antwort rechtzeitig gesendet haben (10), werden ignoriert und gelten als nicht gefunden.

Wie bei anderen, zuvor definierten Schnittstellen der DevBox, wird auch für die Adressierung und Erkundung eine *API* in Form einer `DiscoveryClient`-Klasse konzipiert. Die nachfolgende Abbildung 24 zeigt ein entsprechendes Klassendiagramm.

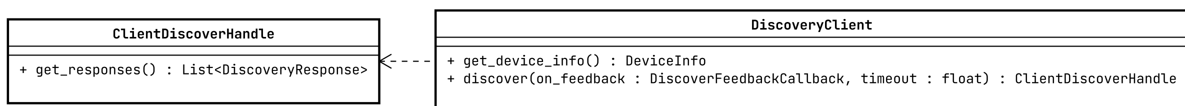

Abbildung 24: Klassendiagramm der DiscoveryClient-Klasse

Damit nicht jedes Mal erneut nach DevBoxen gesucht werden muss, soll zusätzlich eine Konsolenanwendung entwickelt werden, die das Erkunden von Geräten ermöglicht und eine erkundete DevBox als Standardverbindung speichern kann. Programme, die sich mit einer DevBox verbinden wollen, können dann stets auf diese Verbindung zurückgreifen, ohne das Programm jedes Mal anzupassen oder eine DevBox jedes Mal zu erkunden.

5.3.5 Programmierschnittstelle für die DevBox

In den vorherigen Kapiteln wurden bereits *API*s für die Verwender der DevBox konzipiert. Auf diesen Überlegungen aufbauend zeigt das Klassendiagramm der folgenden Abbildung 25 die Klasse `DevBoxClient`, die alle *API*s für Schnittstellen der DevBox zusammenfasst.

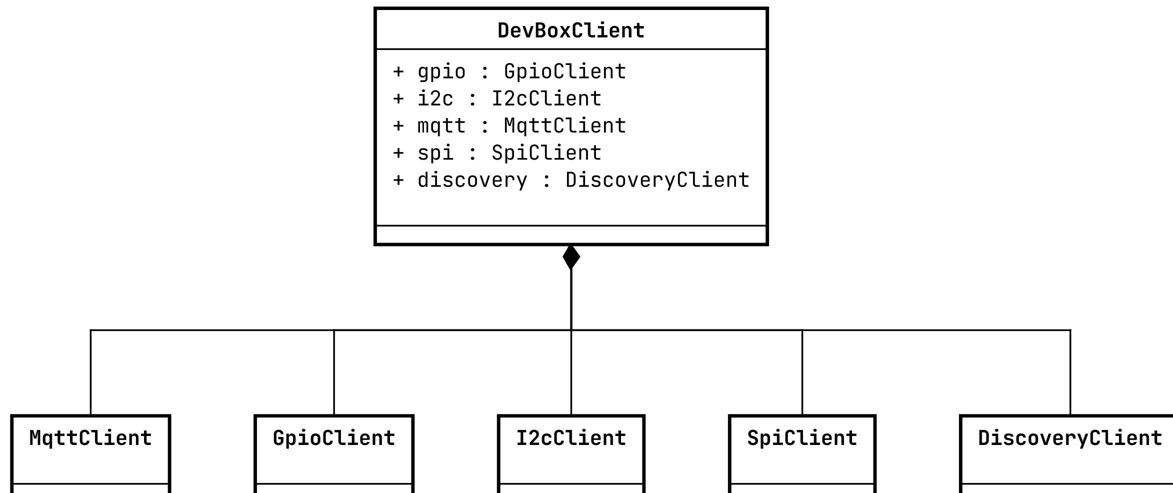

Abbildung 25: Klassendiagramm der DevBoxClient-Klasse

Die *API* deckt damit den Funktionsumfang, wie durch die zuvor genannten *ROS*-Topics und Services geboten, ab und ermöglicht einen einfachen Zugriff. Dadurch entfallen das Generieren sowie Interpretieren der entsprechenden *ROS*-Nachrichten auf der Seite des Verwenders. Es ist dennoch weiterhin möglich die plattformunabhängigen *ROS*-Topics und Services direkt zu verwenden.

5.3.6 Betriebssystem & weitere Software

Das Konzept beschrieb bisher die Hardware und zu entwickelnde Softwarekomponenten für die DevBox. Damit diese Software auch ausgeführt werden kann, sind ein Betriebssystem und entsprechende Laufzeitumgebungen notwendig. Das Konzept sieht die Verwendung von *ROS* 2 vor. Dementsprechend muss dies auf dem Raspberry Pi installiert sein. Um eine zumindest ähnliche Softwareumgebung wie der von ROS-E zu ermöglichen (vgl. Kapitel 4.1), wird als Betriebssystem ebenfalls Ubuntu 20.04 LTS und *ROS* Foxy verwendet. [1] [6]

6 Prototypische Umsetzung des Konzepts

In den vorherigen Kapiteln wurden die konzeptionellen Grundlagen für die Implementierung der zu entwickelnden DevBox für ROS-E erarbeitet. In diesem Kapitel wird die zunächst prototypische Umsetzung dieses Konzepts erläutert. Der Prototyp soll der Überprüfung und Erprobung des Konzepts dienen, bevor eine DevBox gebaut wird.

6.1 Hardwareprototyp

Basierend auf der Auswahl der Hardwarekomponenten für die DevBox in Kapitel 5.2.1 wurde ein Hardwareprototyp mit den essenziellen Komponenten zusammengestellt. Die folgende Abbildung 26 zeigt diesen prototypischen Aufbau für die DevBox.

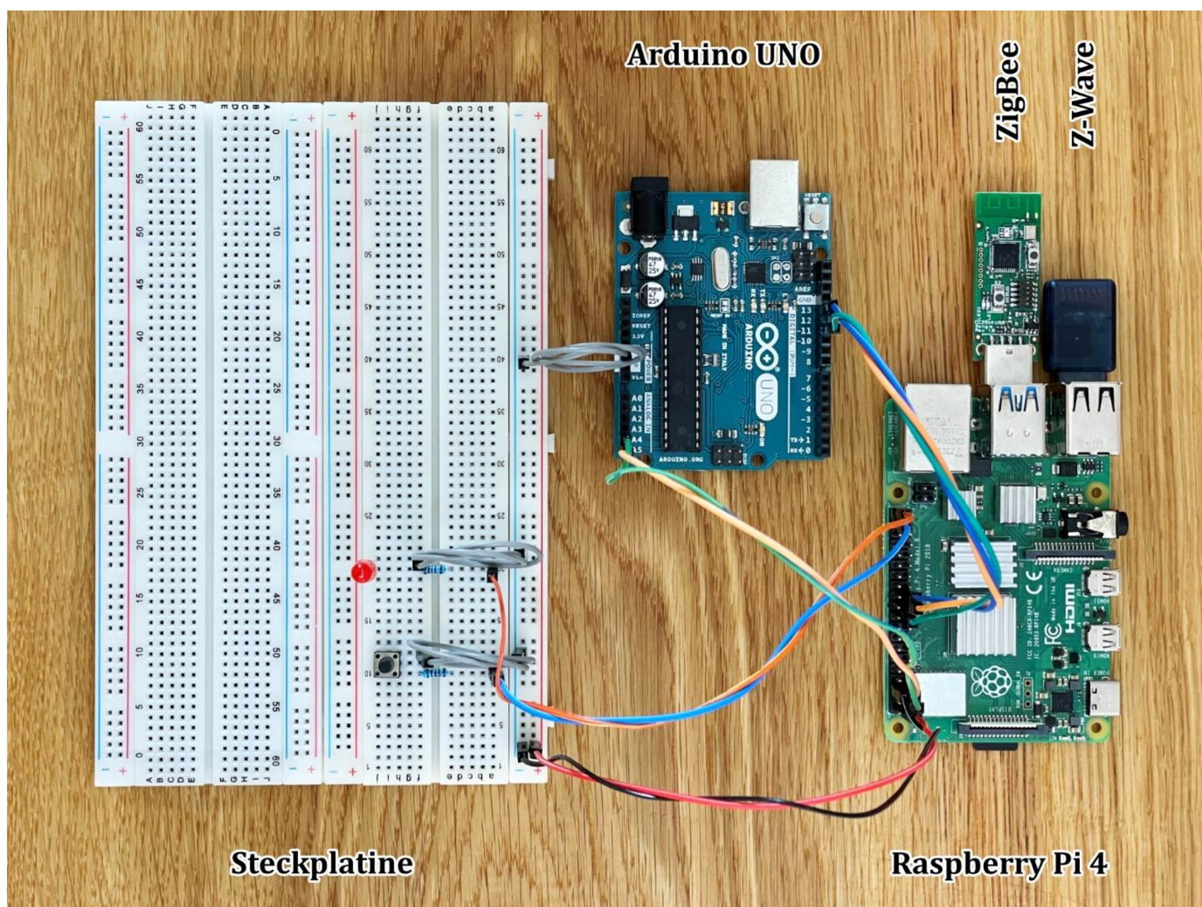

Abbildung 26: Aufbau des Hardwareprototypen für die DevBox

Der Aufbau besteht aus einem Raspberry Pi 4 (Model B) mit 8 GB an Arbeitsspeicher, an dem per *GPIO* eine *LED* und ein Taster sowie ein Arduino UNO per *I²C* und *SPI* angeschlossen sind. Zudem sind je ein *USB*-Adapter für *ZigBee* und *Z-Wave* angeschlossen. Ein *USB*-Adapter für *EnOcean* war während der Entwicklung nicht verfügbar. Die Implementierung für *EnOcean* soll daher zu geeigneter Zeit ergänzt werden.

Dieser einfache Aufbau weicht von der konzipierten DevBox ab, eignet sich jedoch gut für das Entwickeln und Testen der Software, da alle konzipierten Verbindungen der DevBox abgedeckt sind.

6.2 Software auf der DevBox

Mit dem soeben beschriebenen prototypischen Hardwareaufbau wurde die Software für die DevBox entwickelt und getestet. Der im Hardwareprototypen verwendete Raspberry Pi wurde mit Ubuntu 20.04 LTS und *ROS* 2 eingerichtet (vgl. Kapitel 5.3.6). Dieses Unterkapitel erläutert die Implementierung der Software auf der Seite der DevBox.

In der rechtsstehenden Abbildung 27 sind die Softwarepakete dargestellt, die die Implementierung der Software der DevBox enthalten. Das `base` Paket enthält grundlegende Funktionen wie die Adressierung und Erkundung von Geräten und erleichtert die Implementierung der anderen Pakete. Darauf bauen Pakete auf, die konkrete Schnittstellen wie *GPIO*, *I²C*, *SPI* und *MQTT* implementieren. Das `devbox` Paket fasst diese Pakete und deren Schnittstellen für die DevBox zusammen.

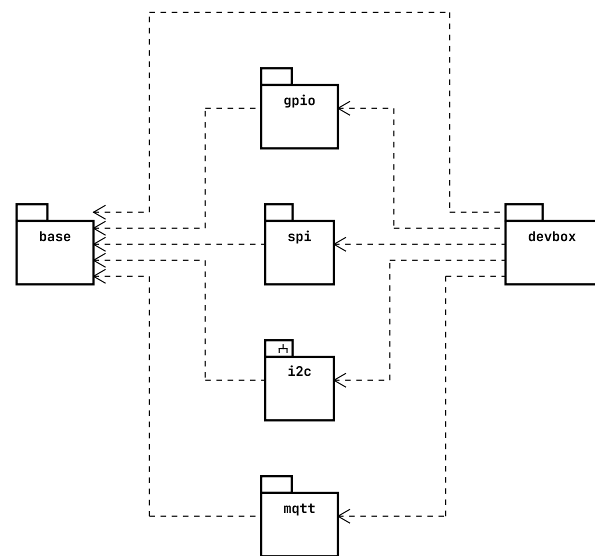

Abbildung 27: Übersicht und Abhängigkeiten der Softwarepakete (vereinfacht)

Im Folgenden wird genauer auf die Implementierung der Funktionen und Schnittstellen eingegangen. Dabei erfolgt die Umsetzung grundsätzlich als *ROS*-Knoten in Python[6] 3.

6.2.1 General Purpose Input/Output

Für die Implementierung der *GPIO*-Schnittstelle wurde ein *ROS*-Knoten, der die in Kapitel 5.3.1.1 konzipierten *ROS*-Services umsetzt, erstellt. Der Knoten wurde so entworfen, dass die *GPIO*-Operationen der verschiedenen Services durch eine Klasse `GpioManager` übernommen werden. In diesem Unterkapitel wird die Funktionsweise dieser Klasse kurz erläutert.

[6] https://www.python.org

Der `GpioManager` verwendet die Bibliothek RPi.GPIO [7] für den Zugriff auf die *GPIO*-Schnittstelle des Raspberry Pi 4 der DevBox. Die folgende Abbildung 28 zeigt einen Auszug aus dem Quellcode für die `GpioManager`-Klasse.

```
import RPi.GPIO as gpio


class GpioManager:

    def get_input(self, channel: int) → bool:
        gpio.setup(channel, gpio.IN)
        return bool(gpio.input(channel))

    def get_output(self, channel: int) → bool:
        gpio.setup(channel, gpio.OUT)
        return bool(gpio.input(channel)

    def set_output(self, channel: int, level: bool):
        gpio.setup(channel, gpio.OUT)
        gpio.output(channel, level)
```

Abbildung 28: Quellcode für die GpioManager-Klasse (Auszug)

Wie bereits zuvor konzipiert, erfolgt die notwendige Konfiguration des *GPIO*-Pins automatisch (in Abbildung 28 jeweils durch `gpio.setup(…)`). Darüber hinaus verfügt der GpioManager über Methoden (nicht abgebildet), um die *GPIO*-Schnittstelle zu initialisieren und ordnungsgemäß nach Verwendung zu bereinigen. Im Rahmen der Bearbeitungszeit konnte die im Konzept unter Kapitel 5.3.1.1 erdachte Schnittstelle für *GPIO*-Ereignisse nicht umgesetzt werden. Eine entsprechende Implementierung soll im Anschluss der `GpioManager`-Klasse hinzugefügt werden.

6.2.2 Inter-Integrated Circuit

Für die Kommunikation über den *I²C*-Bus wurden ebenfalls die in Kapitel 5.3.1.2 geplanten *ROS*-Services in einem Knoten implementiert. Die Steuerung des *I²C*-Busses wird dabei über die Python-Bibliothek `smbus2`[8] ermöglicht. Diese Bibliothek ist vorrangig für Steuerung eines *System Management Bus* (*SMBus*) konzipiert, unterstützt jedoch auch *I²C*, da SMBus auf *I²C* aufbaut und weitestgehend kompatibel ist. [48]

Der Raspberry Pi verfügt über zwei verschiedene *I²C*-Busse. Die Implementierung des *ROS*-Knoten für *I²C* wurde so entwickelt, dass der Bus als Parameter des Knoten angegeben werden kann. Jedoch wird standardmäßig der *I²C*-Bus mit der Nummer 1

[7] Siehe https://pypi.org/project/RPi.GPIO
[8] Siehe https://pypi.org/project/smbus2

eingestellt. Der andere Bus mit der Nummer 0 ist beim Raspberry Pi für *HAT*s reserviert. [49] [50]

6.2.3 Serial Peripheral Interface

Für die Möglichkeit mit *SPI*-Geräten zu kommunizieren wurde ein weiterer *ROS*-Knoten der DevBox-Software hinzugefügt, der die in Kapitel 5.3.1.3 konzipierten *ROS*-Services implementiert. Die Implementierung verwendet dabei die Python-Bibliothek `spidev`[9] für die Ansteuerung der *SPI*-Geräte.

Da jede Verbindung zu einem *SPI*-Gerät mit der `spidev`-Bibliothek einzeln geöffnet werden muss, wurde zusätzlich die Klasse `SpiManager` implementiert. Diese verwaltet Verbindungen zu *SPI*-Geräten und bietet Methoden an, mit denen neue Verbindungen geöffnet und bestehende geschlossen werden können. Die Klasse ist so entwickelt, dass bereits offene Verbindungen wiederverwendet werden und eine Schließung aller Verbindungen bei Beendigung des Knoten erfolgt. Die genaue Funktionsweise kann dem angehängten Quellcode entnommen werden.

6.2.4 Message Queuing Telemetry Transport

Für die Implementierung der *MQTT*-Schnittstelle wurde zunächst ein *MQTT*-Broker auf der DevBox eingerichtet. Für diesen Zweck hat sich Eclipse Mosquitto[10] in früheren Arbeiten zu *Smart Home* mit ROS-E bewährt und kommt deshalb auch für die DevBox zum Einsatz. [3]

Neben dem *MQTT*-Broker wird auch der im Kapitel 5.3.2.1 konzipierte Service `mqtt/get_broker_info` zum Abfragen der Brokerinformationen benötigt und wurde daher mit einem *ROS*-Knoten implementiert. Der Knoten wurde so entwickelt, dass die *IP-Adresse* der DevBox und der standardmäßige *MQTT*-Port 1883 als Verbindungsinformation zurückgegeben werden. Jedoch können beide Werte mithilfe von Parametern des Knoten geändert werden. [36]

6.2.5 EnOcean, ZigBee & Z-Wave

Das Konzept für die Anbindung von *Smart Home* Standards in Kapitel 5.3.2 sieht den Einsatz von *MQTT*-Adaptern vor. Basierend auf zuvor gemachten Erfahrungen mit *ZigBee* und *Z-Wave* wurden entsprechende Adapter für diese Standards ausgewählt. [3]

Für die Übersetzung der *ZigBee*-Kommunikation nach *MQTT* wird Zigbee2MQTT[11] verwendet. Die Anbindung von *Z-Wave* wird mittels ZWavejs2Mqtt[12] realisiert. Wie

---

[9] Siehe https://pypi.org/project/spidev
[10] Siehe https://mosquitto.org
[11] Siehe https://www.zigbee2mqtt.io
[12] Siehe https://zwave-js.github.io/zwavejs2mqtt

eingangs in Kapitel 6.1 erwähnt, soll die Umsetzung von *EnOcean* nachträglich ergänzt werden. Die Implementierung kann jedoch nach demselben Konzept wie bei *ZigBee* und *Z-Wave* mittels *MQTT*-Adapter wie `enocean-mqtt`[13] oder `enocean-mqtt-bridge`[14] erfolgen.

Sowohl Zigbee2MQTT als auch ZWavejs2Mqtt verfügen über ein Dashboard, welches als Webseite angeboten wird. Über dieses lassen sich verschiedene Einstellungen zu *ZigBee* bzw. *Z-Wave* und der *MQTT*-Verbindung machen. Zudem können über diese Webseite verbundene Geräte eingesehen oder neue Geräte hinzugefügt werden.

6.2.6 Adressierung & Erkundung

Für die Erkundung von DevBoxen wurden die verschiedenen *ROS*-Knoten, wie in Kapitel 5.3.4 konzipiert, implementiert. Die Knoten wurden jeweils so implementiert, dass die Adresse und die Art des Geräts (ROS-E oder DevBox) als Parameter angegeben werden können. Falls keine Adresse angegeben wird, wird die *MAC-Adresse* des Geräts verwendet. Die Bestimmung der *MAC-Adresse* und wie die Adresse als Präfix für die konzipierten *ROS*-Topics, Services und Actions verwendet wird, erläutern die folgenden Unterkapitel.

6.2.6.1 Bestimmung der Adresse

Für die Adressierung von Geräten wurde zunächst eine Funktion entwickelt, mit der die *MAC-Adresse* eines Geräts bestimmt werden kann. Die entsprechende Implementierung wird vereinfacht in der folgenden Abbildung 29 dargestellt.

```
def get_address(interface="eth0") → Optional[int]:
    path = Path("/sys/class/net") / interface / "address"

    with open(path, "r") as file:
        address = file.read().strip()

    return int(address.replace(":", ""), 16)
```

Abbildung 29: Quellcode für die Ermittlung der MAC-Adresse (vereinfacht)

Der abgebildete Quellcode ist ausschließlich für ein *Linux*-Betriebssystem ausgelegt, da die *MAC-Adresse* mithilfe des virtuellen Dateisystems `sysfs`[15], welches unter anderem den Zugriff auf Netzwerkeigenschaften ermöglicht, ausgelesen wird. Die *MAC-Adresse* des *Ethernet*-Adapters des Geräts wird beispielsweise von `/sys/class/net/eth0/address` gelesen. Anschließend wird die *MAC-Adresse*, die als Zeichenkette der Form

[13] Siehe https://github.com/embyt/enocean-mqtt
[14] Siehe https://github.com/rosenloecher-it/enocean-mqtt-bridge
[15] https://manpages.ubuntu.com/manpages/focal/man5/sysfs.5.html

`12:34:56:78:9A:BC` gelesen wurde, in eine Dezimalzahl [16] konvertiert und zurückgegeben. [51]

6.2.6.2 Kodierung von Adressen

Wie in Kapitel 5.3.4 konzipiert, wird die Adresse als Namensraum für alle gerätespezifischen *ROS*-Topics, Services und Actions verwendet. Dabei wird die Adresse nicht direkt als Dezimalzahl eingesetzt, sondern es wird mit einer alternativen Darstellung gearbeitet, die folgend erläutert wird.

Ursprünglich war der Einsatz der Base64-Kodierung geplant, mit dem Ziel die verfügbaren Zeichen eines *ROS*-Topic oder Service-Namen effizienter zu nutzen und gleichzeitig für Verwender einprägsamer zu sein. Jedoch sind für die Namen von *ROS*-Topics oder Services nur 62 verschiedene Zeichen in beliebiger Kombination zugelassen, sodass diese Kodierung nicht verwendet werden kann. Als Lösung wurde eine abgewandelte Base62-Kodierung entwickelt, welche eine Dezimalzahl zur Basis 62 konvertiert und als *ASCII*-Bytes darstellt. Eine entsprechende Dekodierung wurde ebenfalls implementiert. Da es sich immer um eine *MAC-Adresse* bestehend aus 48 Bits handelt, werden immer 9 *ASCII*-Zeichen[17] kodiert. [14] [52]

Der Algorithmus zur Kodierung und Dekodierung der Adresse wird in der folgenden Abbildung 30 dargestellt. Wie zu erkennen ist, handelt es sich jeweils um einen sehr einfach zu implementierenden Algorithmus.

[16] Eine Dezimalzahl eignet sich besser in der weiteren Verwendung der MAC-Adresse.
[17] Um 48 Bits in der Basis 62 darzustellen, werden mindestens $\lceil \log_{62}(2^{48}) \rceil = 9$ Zeichen benötigt.

```
def encode(value: int) → bytes:
    bytes_ = bytearray(9)

    for index in range(len(bytes_), 0, -1):
        value, remainder = divmod(value, len(ALPHABET))
        bytes_[index - 1] = ALPHABET[remainder]

    return bytes(bytes_)


def decode(bytes_: bytes) → int:
    value = 0

    for byte in bytes_:
        value *= len(ALPHABET)
        value += ALPHABET.index(byte)

    return value
```

Abbildung 30: Quellcode für die Kodierung und Dekodierung von Adressen

Die nachfolgende Abbildung 31 zeigt das verwendete Alphabet (`ALPHABET` in Abbildung 30) für die Base62-Kodierung. Dabei ist in der oberen Zeile jeweils der dezimale Wert und in der unteren Zeile der entsprechende Base62-Wert abgebildet.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u

21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41
v	w	x	y	z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P

42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61
Q	R	S	T	U	V	W	X	Y	Z	0	1	2	3	4	5	6	7	8	9

Abbildung 31: Alphabet für die Base62-Kodierung

Das Alphabet wurde so gewählt, dass für alle möglichen *MAC-Adressen* im Bereich von 0 bis $(2^{48} - 1)$ das jeweils erste Zeichen der kodierten Adresse ein Buchstabe ist, da ein Name für ein *ROS*-Topic oder Service nicht mit einer Zahl beginnen darf. [14]

6.3 Software für Verwender

Die zuvor besprochenen Implementierungen bezogen sich auf die Software, die auf der DevBox ausgeführt wird. In diesem Unterkapitel wird auf Software eingegangen, die auf

der Seite von ROS-E von Verwendern genutzt werden kann, um den Umgang zu vereinfachen.

Im Konzept (siehe Kapitel 5.3.5) wurde bereits die Programmierschnittstelle (*API*) für die Verwender der DevBox beschrieben. Diese wurde weitestgehend analog implementiert. Jedoch wurden ein paar Anpassungen vorgenommen, um die Umsetzung und Benutzung der *API* zu erleichtern. Folgend werden die wichtigsten Abweichungen beschrieben.

6.3.1 Basisklassen

Da es mehrere Clients für die diversen Schnittstellen der DevBox gibt, wurde die Basisklasse `InterfaceClient` implementiert, von der alle Clients für Schnittstellen erben. In diesem Zuge wurde auch eine Basisklasse für die `DevBoxClient`-Klasse erstellt, die als `DeviceClient` bezeichnet wird. Das folgende Klassendiagramm in Abbildung 32 zeigt, dass diese Klassen eine abstrakte Version der in Kapitel 5.3.5 konzipierten Struktur darstellen.

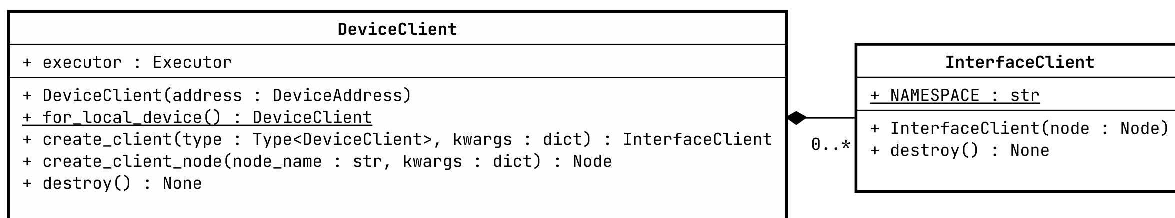

Abbildung 32: Klassendiagramm für DeviceClient und InterfaceClient

Die Klasse `DeviceClient` dient als Einstiegspunkt in die Verwendung von Schnittstellen, die jeweils über eine Unterklasse von `InterfaceClient` implementiert werden. Einem `InterfaceClient` wird beim Instanziieren ein *ROS*-Knoten übergeben, mit dem eine Kommunikation mit den *ROS*-Topics, Services und Actions möglich ist. Mithilfe der Methode `create_client_node` kann ein passender *ROS*-Knoten für ein `InterfaceClient` erzeugt werden, dessen Namensraum bereits die Adresse des Geräts (siehe Konstruktor von `DeviceClient`) beinhaltet. Weiter vereinfacht wird diese Methode von `create_client`, die basierend auf einer Unterklasse von `InterfaceClient`, den entsprechenden Client selbst erzeugt. Somit ist die Erstellung von Unterklassen einfach und benötigt dafür nur wenig sich wiederholenden Quellcode.

Der folgende Quellcode in Abbildung 33 demonstriert vereinfacht zwei Varianten, wie eine neue Instanz der `DevBoxClient`-Klasse (als Unterklasse von `DeviceClient`) angelegt werden kann, um damit anschließend die verschiedenen Schnittstellen ansteuern zu können.

```
# Verbindung mit einer bekannten Adresse in Base62.
devbox = DevBoxClient(DeviceAddress.decode_str("bjsRrxj2X"))

# Verbindung mit der lokalen18 DevBox.
devbox = DevBoxClient.for_local_device()
```

Abbildung 33: Quellcode zur Erzeugung einer DevBoxClient-Instanz

Die in Kapitel 5.3.4 konzipierte Möglichkeit, DevBoxen mithilfe einer Konsolenanwendung zu entdecken und eine DevBox als Standardverbindung festzulegen, konnte innerhalb der Bearbeitungszeit nicht umgesetzt werden. Durch eine nachträgliche Implementierung soll der Quellcode in Abbildung 33 durch eine weitere Methode ergänzt werden, die eine neue DevBox-Instanz für die gespeicherte Standardverbindung erzeugt.

6.3.2 Asynchrone Ausführung

Die im Konzept entwickelten APIs für die einzelnen Schnittstellen zeigen jeweils nur einfache Methodensignaturen, die von einer synchronen Ausführung ausgehen. Da die Methoden nur den Zugriff auf darunterliegende *ROS*-Topics, Services oder Actions abstrahieren, handelt es sich jedoch um asynchrone Operationen. Dementsprechend werden alle Methoden, die zuvor in Klassendiagrammen im Konzept definiert wurden, um eine zweite Methode mit dem Postfix `_async` ergänzt. Die Methoden führen dieselbe Operation aus, geben jedoch statt dem Ergebnis ein *ROS*-`Future`-Objekt zurück, mit dem auf die Fertigstellung der Operation gewartet werden kann, ohne die Ausführung anderer Operationen zu blockieren. Eine Visualisierung dieser Anpassung erfolgt im anschließenden Kapitel zum `MqttClient`. [53]

6.3.3 Ergänzungen zur MqttClient-Klasse

Nachfolgend ist eine überarbeitete Version des Klassendiagramms der `MqttClient`-Klasse in Abbildung 34 dargestellt, um die Anpassungen, die in den beiden vorherigen Kapiteln beschrieben wurden, beispielhaft zu visualisieren.

[18] Das lokale Gerät ist jenes, welches das Programm ausführt.

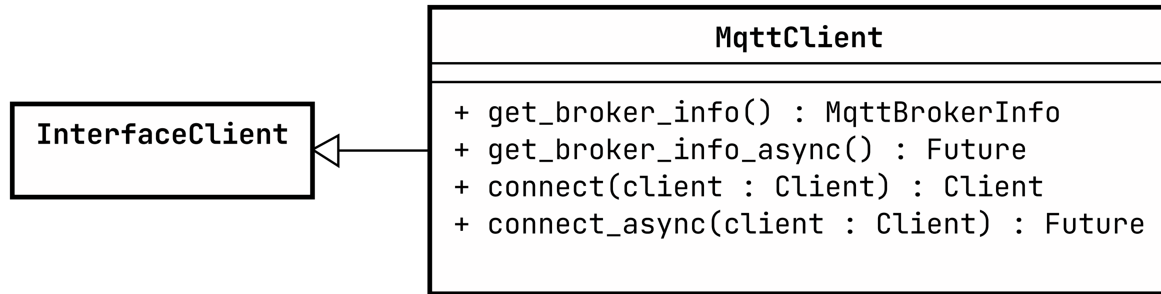

Abbildung 34: Klassendiagramm der MqttClient-Klasse (überarbeitet)

Das Konzept zur `MqttClient`-Klasse sah die Methode `connect` vor. Es wurde jedoch nicht weiter spezifiziert (vgl. Kapitel 5.3.2.1), um welchen `Client` es sich handelt. Im Rahmen der Implementierung wurde für den Client die Python-Bibliothek Eclipse Paho *MQTT*[19] gewählt, die die Verbindung mit dem *MQTT*-Broker herstellt.

6.3.4 „Hello World“-Programme & Dokumentation

Für die bis hierhin beschrieben Implementierung der Hardware und Software der DevBox wurden beispielhafte Anwendungen („Hello World“-Programme), wie von Anforderung R37 gefordert, angelegt. Diese zeigen, wie die *API* für die DevBox verwendet werden kann, um verschiedene Hardwareschnittstellen zu bedienen oder mit *Smart Home* Geräten zu kommunizieren.

Darüber hinaus wurden Methoden der DevBox *API* mittels Python Docstrings[20] im Quellcode dokumentiert. Jedoch konnte im Rahmen der Bearbeitungszeit dieser Bachelorarbeit keine zusammenhängende Dokumentation der DevBox und wie diese zu verwenden ist (vgl. R34, R35 und R36), erstellt werden.

[19] Siehe https://pypi.org/project/paho-mqtt
[20] Mehr unter https://peps.python.org/pep-0257

7 Tests & Ergebnisse

In diesem Kapitel werden die Softwaretests der DevBox erläutert und anschließend ausgewertet. Dafür werden zunächst der Testaufbau, die verschiedenen Testfälle und wie getestet wurde beschrieben.

7.1 Aufbau & Testfälle

Als Grundlage für alle Tests dient der Prototyp der DevBox, wie in Kapitel 6.1 vorgestellt. Diesem Aufbau wird eine ROS-E hinzugefügt, die je nach Test per *Wi-Fi* oder *Ethernet* mit der DevBox verbunden wird. Für das Testen der Anbindung von *Z-Wave* Geräten wird zusätzlich „The Button“ (dt. Taster) von FIBARO[21] eingesetzt und für *ZigBee* Lampen der Serie „TRÅDFRI“ von IKEA[22].

Für einen kleinen Teil der Funktionen der Software wurden automatisierte Tests angelegt. Darunter zählt beispielsweise die Kodierung von Geräteadressen. Der größte Teil an Funktionen wurde jedoch manuell getestet, da das Zusammenwirken mehrerer Programme und/oder Geräte untersucht wurde. Für diesen Zweck wurden die bereits entwickelten „Hello World“-Programme verwendet, die einzelne Testfälle darstellen.

7.1.1 Verbindung zwischen der DevBox und ROS-E

Mittels des Hardwareprototypen und einer ROS-E konnte die Verbindung zwischen beiden via *Ethernet* und *Wi-Fi* getestet werden. Über beide Verbindungsalternativen konnte erfolgreich kommuniziert werden.

7.1.2 Adressierung & Erkundung der DevBox

Die Kodierung von Geräteadressen wurde erfolgreich automatisiert getestet. Mithilfe einer entsprechend eingerichteten ROS-E konnte die prototypische DevBox erkundet werden. Auch die Adressierung der DevBox bzw. der angebotenen Funktionen dieser hat gut funktioniert.

7.1.3 Anbindung von Hardwarekomponenten

Die Steuerung von *GPIO*-Pins über die DevBox-Software konnte erfolgreich mittels der im Hardwareprototypen vorhandenen *LED* getestet werden. Auch die Verwendung eines *GPIO*-Anschlusses als Eingabe für den angeschlossenen Taster war erfolgreich. Jedoch konnte die Schnittstelle für *GPIO*-Ereignisse aufgrund der noch fehlenden Implementierung nicht getestet werden.

[21] https://www.fibaro.com/de/products/the-button
[22] https://ingka.page.link/C7DGxLLNVq9pRw5D9

Mithilfe des mit dem Prototyp verbundenen Arduino UNO wurde das Lesen und Schreiben über den *I²C*-Bus getestet. Auf dem Arduino wurde dazu ein Programm ausgeführt, welches das Schreiben von Daten und das spätere Auslesen derselben Daten erlaubt. Beide Operationen konnten erfolgreich über die DevBox durchgeführt werden.

Ähnlich wie *I²C* wurde auch die *SPI*-Schnittstelle mit dem Arduino als Gegenstück getestet. Jedoch konnten dabei keine zufriedenstellenden Ergebnisse erzielt werden. Im Rahmen der Bearbeitungszeit war es auch nicht möglich dieses Problem zu lösen. Eine mögliche Lösung könnte der Wechsel des Betriebssystems von Ubuntu zu Raspberry Pi OS [23] sein, welches die Hardwareschnittstellen des Raspberry Pi besser unterstützt. Zudem könnte auch das Testprogramm des Arduino die Fehlfunktion begründen.

7.1.4 Anbindung von Smart Home Geräten

Die Anbindung des oben erwähnten Tasters über *Z-Wave* erfolgte ohne Probleme über das entsprechende Dashboard von ZWavejs2Mqtt. Anschließend konnten in einem Test erfolgreich *MQTT*-Nachrichten beim Drücken des Tasters auf ROS-E über die DevBox empfangen werden. Die ordnungsgemäße Funktionalität von *ZigBee* konnte mithilfe der eingangs erwähnten Lampe gleichermaßen getestet werden. Die Funktionalität der Anbindung von *EnOcean* konnte aufgrund der noch ausstehenden Implementierung nicht innerhalb der Bearbeitungszeit getestet werden.

7.2 Auswertung der Ergebnisse

Die durchgeführten Tests zeigen, dass grundlegende Funktionen der DevBox funktionieren und verwendet werden können. Ein paar Funktionen konnten nicht innerhalb der Bearbeitungszeit implementiert bzw. getestet werden. Jedoch weist die aktuelle Implementierung die Möglichkeiten zur Steuerung von Hardware via *GPIO* und *I²C* sowie der Anbindung von *ZigBee* und *Z-Wave* Geräten auf.

[23] Das offizielle Betriebssystem für Raspberry Pi. Mehr unter https://www.raspberrypi.com/software.

8 Fazit

Nachdem nun alle Aspekte der Entwicklung der DevBox beginnend bei den Anforderungen bis hin zur Implementierung in den vorherigen Kapiteln betrachtet wurden, wird in diesem letzten Kapitel das Ergebnis der Arbeit zusammengefasst. Anschließend wird ein Blick auf noch ausstehende Entwicklungen und Anregungen für die Weiterentwicklung der DevBox geworfen.

8.1 Zusammenfassung

Das Ziel dieser Arbeit war die Entwicklung der DevBox, die eine flexible und prototypische Entwicklung von Hardwarekomponenten für den Roboter ROS-E unterstützt. Der Fokus lag dabei auf der Verbindung des Systems von ROS-E mit Aktoren, Sensoren und *Smart Home* Geräten, die an die DevBox angeschlossen sind.

Für die Realisierung dieses Ziels wurde sowohl die Hardware als auch die Software der DevBox entworfen. Es wurde ein Gehäuse für die DevBox entwickelt, welches die Hardware der DevBox schützend beinhaltet und einen Transport erlaubt. In das Gehäuse wurden speziell angepasste Leiterplatten integriert, die wichtige Hardwareschnittstellen – *GPIO*, *I²C* und *SPI* – in die unmittelbare Nähe einer großen *Steckplatine* für den flexiblen Zugriff durch Verwender der DevBox führen. Es wurde zudem eine Integration der in Europa wichtigsten *Smart Home* Funkstandards – *EnOcean*, *ZigBee* und *Z-Wave* – für die DevBox entworfen.

Jedoch konnte im Rahmen dieser Arbeit nur eine prototypische Umsetzung der geplanten DevBox erreicht werden, wobei grundlegende Funktionen der DevBox implementiert werden konnten.

8.2 Ausblick

Aus der Zusammenfassung geht hervor, dass die Entwicklung der DevBox noch nicht abgeschlossen ist. In diesem Unterkapitel werden kurz ausstehende Aufgaben besprochen und ein Blick auf Anregungen für Weiterentwicklungen geworfen.

In dieser Arbeit wurde das Konzept für den Bau einer DevBox entwickelt. Im Rahmen der Bearbeitungszeit konnten auch schon die konzipierten Leiterplatten (siehe Kapitel 5.2.2.1) beschafft werden, jedoch steht der Bau einer DevBox noch aus. Ebenso soll die Umsetzung für die Anbindung von *Smart Home* Geräten via *EnOcean* noch erfolgen.

Das Konzept für die Software der DevBox sieht zusätzliche Programme vor, die das Herstellen einer Verbindung zur DevBox vereinfachen. Die aktuelle Implementierung hat sich vor allem auf die Entwicklung von Kernkomponenten konzentriert, sodass diese Werkzeuge zu einem späteren Zeitpunkt nachgereicht werden.

Der entwickelte Quellcode enthält Kommentare und Hinweise, die Verwendern bei der Nutzung der DevBox unterstützen sollen. Auch in erstellten Beispielen zur Verwendung sind hilfreiche Erläuterungen eingebunden. Eine zusammenhängende Dokumentation für Verwender der DevBox kann in Zukunft ergänzt werden.

Nach der Fertigstellung einer DevBox ist der nächste Schritt der Einsatz für die Weiterentwicklung von ROS-E. Dabei sollen auch Eindrücke des Entwicklerteams von ROS-E bei der Verwendung der DevBox gesammelt werden, um diese gegebenenfalls zu verbessern. Neben der Weiterentwicklung von ROS-E wurden zu Beginn der Arbeit Lehrveranstaltungen als mögliche Einsatzorte der DevBox erwähnt. In weiteren Schritten könnte ein Konzept für die Verwendung der DevBox in der Lehre entwickelt werden, um ein Schülerlabor oder eine andere Form von Lehrveranstaltung zu realisieren.

Besonders im Bezug zur Lehre und in einem Umfeld, wo das Vorwissen über verschiedene Aktoren und Sensoren eventuell gering ausfällt, könnte eine interaktive Variante der Dokumentation entwickelt werden. Eine Anwendung könnte Schritt für Schritt durch verschiedene Tutorials führen und dabei die Verwendung von Hardwareschnittstellen und -komponenten unter Einbindung von interaktiven Elementen lehren. Die Anwendung könnte, je nach Zielgruppe oder Benutzer, spielerisch oder sachlich gestaltet werden.

9 Anhang

Die folgenden Unterkapitel enthalten zusätzliches und unterstützendes Material der Arbeit. Es wird bei Bedarf an entsprechender Stelle in der Arbeit auf Anhänge verwiesen.

9.1 Materialliste für den Bau einer DevBox

Die folgende Tabelle enthält alle Materialien sowie deren Kosten, die für den Bau einer DevBox benötigt werden. Die Kosten sind immer pro Stück angegeben. Für den Fall, dass das entsprechende Material nicht stückweise erworben werden kann, wird darauf und auf die zu erwerbende Menge hingewiesen.

Anzahl	Beschreibung	Stückpreis
1	Platine „Arduino Board“	10,03 €
1	Platine „Bus Board“	7,88 €
1	Platine „*GPIO* Board“	7,88 €
1	Raspberry Pi 4B (8 GB)	94,20 €
1	*ZigBee* CC2531 *USB*-Stick	11,90 €
1	Z-Wave.Me UZB *USB*-Stick	25,62 €
1	Arduino Nano	23,34 €
2	*Steckplatine* (55 x 165 mm)	5,12 €
1	Acrylglas (3 x 235 x 266 mm)	3,67 €
2	Holzplatte (3 x 235 x 266 mm)	0,93 €
4	Holzplatte (3 x 20 x 258 mm)	0,08 €
4	Holzplatte (3 x 50 x 258 mm)	0,19 €
1	DevBox	197,70 €

Tabelle 11: Materialien der DevBox und deren Kosten

Die Stückpreise für die Platinen beziehen sich jeweils auf eine Bestellung von fünf Platinen. Alle in Tabelle 11 aufgeführten Preise wurden im Jahr 2022 ermittelt. Für den Bau sind darüber hinaus Schrauben und Muttern notwendig, die dem Modell zur DevBox zu entnehmen sind und preislich nicht stark ins Gewicht fallen. Eine Abweichung der Kosten ist aufgrund von Mehrkosten, die beim Bau der DevBox entstehen möglich.

9.2 Digitale Anhänge

Dieser Bachelorarbeit ist eine SD-Karte mit einer digitalen Version dieser Arbeit als PDF beigelegt. Darüber hinaus sind folgend weitere digitale Anhänge beschrieben, die jedoch nicht auf der SD-Karte abgelegt sind.

9.2.1 Software & Quellcode

Der im Rahmen dieser Arbeit entwickelte Quellcode wird mittels Git verwaltet. Es folgt eine Auflistung aller Repositorys mit Quellcode in der folgenden Tabelle 12 nach den in Kapitel 6.2 beschriebenen Softwarepaketen.

Paket	**Link zum Repository**
Base	https://icampusnet.th-wildau.de/oskar.lorenz/ros2-riad-base
DevBox	https://icampusnet.th-wildau.de/oskar.lorenz/ros2-riad-devbox
GPIO	https://icampusnet.th-wildau.de/oskar.lorenz/ros2-riad-gpio
MQTT	https://icampusnet.th-wildau.de/oskar.lorenz/ros2-riad-mqtt
I²C	https://icampusnet.th-wildau.de/oskar.lorenz/ros2-riad-i2c
SPI	https://icampusnet.th-wildau.de/oskar.lorenz/ros2-riad-spi

Tabelle 12: Git Repositorys mit Quellcode dieser Arbeit

Alle Repositorys in Tabelle 12 folgen dem Branching-Modell `git-flow`[24]. Der aktuelle Entwicklungsstand befindet sich daher auf dem Branch `dev` oder `develop` und nicht auf dem `master` oder `main` Branch. Für den Zugriff auf die Repositorys ist ein separater Zugang erforderlich. Ein entsprechender Zugang kann bei Bedarf über iCampus Wildau[25] oder bei Prof. Dr. rer. nat. Janett Mohnke angefordert werden.

9.2.2 Modelle & Leiterplattendesigns

Das Design der DevBox und der vorgestellten Leiterplatten ist über ein Git Repository zugänglich. Unter https://icampusnet.th-wildau.de/oskar.lorenz/devbox ist das Repository zu finden. Ein PDF dieser Arbeit ist ebenfalls in diesem Repository zu finden. Ein Zugang kann ebenfalls über das iCampus Wildau oder Prof. Dr. rer. nat. Janette Mohnke angefordert werden.

[24] Mehr Informationen unter https://nvie.com/posts/a-successful-git-branching-model.
[25] Siehe https://icampus.th-wildau.de für Kontakt und Impressum.

Abkürzungen

Die folgenden, alphabetisch sortieren Abkürzungen wurden in dieser Arbeit verwendet. Darüber hinaus werden bekannte SI-Einheiten und Vorsätze für Maßeinheiten abgekürzt.

AAL	Ambient Assistant Living
API	Application Programming Interface
F	funktional
GPIO	General Purpose Input Output
HAT	Hardware Attached on Top
I^{2}C	Inter-Integrated Circuit
IEEE	Institute of Electrical and Electronics Engineers
IP(v4|v6)	Internet Protocol (Version 4|Version 6)
MAC	Media Access Control
MQTT	Message Queuing Telemetry Transport
NF	nicht funktional
ROS	Robot Operating System
SPI	Serial Peripheral Interface
TCP	Transmission Control Protocol
TH	Technische Hochschule
USB	Universal Serial Bus
UART	Universal Asynchronous Receiver/Transmitter
ASCII	American Standard Code for Information Interchange
OASIS	Organization for the Advancement of Structured Information Standards
ISO	International Organization for Standardization
AES	Advanced Encryption Standard
ECDH	Elliptic Curve Diffie-Hellman
DHCP	Dynamic Host Configuration Protocol
LED	Light Emitting Diode

Glossar

Die folgenden, alphabetisch sortierten Begriffe werden in dieser Arbeit verwendet und werden hier nochmal kurz erläutert.

Ambient Assisted Living	Beschreibt Konzepte und Lösungen zur Unterstützung des alltäglichen Lebens durch Technologie.
Bluetooth	Ein Funkstandard mit einer Frequenz von 2,4 GHz für die drahtlose Kommunikation auf kurzen Entfernungen und in Personal Area Networks.
DHCP	Mithilfe von *DHCP* werden unter anderem *IP-Adressen* in einem Netzwerk automatisch zugewiesen und an teilnehmende Geräte verteilt.
EnOcean	Ein Funkstandard für die Kommunikation mit *Smart Home* Geräten. Siehe Kapitel 3.4.1 für eine detaillierte Erklärung.
Ethernet	Eine kabelgebundene Technologie für die Übertragung von Daten, die vor allem beim Internetprotokoll zum Einsatz kommt und durch *IEEE* 802.3 standardisiert ist.
Gebäudeautomation	Die Planung und Technik für das Steuern und Regeln diverser Systeme in einem Gebäude.
HAT	Eine Leiterplatte, die auf einen Raspberry Pi aufgesteckt werden kann, um die Hardware und Funktionalität zu erweitern.
IP-Adresse	Eine Adresse, die ein Gerät im Internet oder in einem lokalen *IP*-Netzwerk identifiziert.
Linux	Eine Reihe an Betriebssystemen, die auf dem freien, Unix-ähnlichen *Linux*-Kernel basieren.
MAC-Adresse	Eine gerätespezifische Nummer, mit der Daten während einer Kommunikation den entsprechenden Geräten zugeordnet werden können.
Peripheriegerät	Ein externes Gerät, welches meist mit einem Computer verbunden wird, um dessen Funktionalität zu erweitern. Beispiele: Maus, Tastatur oder ein externer Speicher.

Plug & Play	Ein Plug & Play Gerät kann mit minimaler Interaktion mit dem Benutzer selbstständig eingerichtet werden.
Rolling-Code	Eine fortlaufende Nummer, die mit jeder Übertragung inkrementiert wird und somit das erneute Übertragen einer Nachricht unterbindet.
ROS (2)	Das Robot Operating System bietet Softwarebibliotheken und Programme, die bei der Entwicklung von Robotern unterstützen. Siehe Kapitel 3.1 für mehr Informationen.
ROS-E	Ein Roboter der vom RobotikLab Telematik der *TH* Wildau entwickelt wurde. Siehe Kapitel 2 für eine detaillierte Beschreibung.
Smart Home	Die Vernetzung und Automatisierung von Geräten in einem Haus für unter anderem mehr Komfort, Unterhaltung oder Sicherheit in diesem Haus.
Steckplatine	Auf einer Steckplatine (auch Breadboard oder Protoboard genannt) lassen sich prototypische elektrische Schaltungen durch das Stecken temporärer Kabelverbindungen erstellen und testen.
UART	Ein Protocol für den asynchronen und seriellen Austausch von Daten zwischen zwei Geräten.
USB	Der *USB*-Standard ermöglicht einen kabelgebundenen Datenaustausch zwischen kompatiblen Geräten. Zudem ist eine Stromversorgung per *USB* möglich.
Wi-Fi	Ein lokales Funknetz im 2,4 GHz und 5 GHz Bereich nach den Standards der *IEEE* 802.11-Familie.
ZigBee	Ein Funkstandard für die Kommunikation mit *Smart Home* Geräten. Siehe Kapitel 3.4.2 für eine detaillierte Erklärung.
Z-Wave	Ein Funkstandard für die Kommunikation mit *Smart Home* Geräten. Siehe Kapitel 3.4.3 für eine detaillierte Erklärung.

Abbildungen

Die folgenden, nach Erscheinung und mit Seitenzahl aufgelisteten Abbildungen wurden in dieser Arbeit verwendet. Es handelt sich stets um eigene Darstellungen, sofern keine andere Quelle angegeben ist.

Abbildung 1: Skizze zur Architektur der DevBox in Verwendung mit einer ROS-E 3

Abbildung 2: Hardwarekomponenten von ROS-E [RobotikLab, TH Wildau] 5

Abbildung 3: Blick in das Innere von ROS-E (rechts) 6

Abbildung 4: Blick in das Innere von ROS-E (Rückseite) 6

Abbildung 5: Blick in das Innere von ROS-E (links) 6

Abbildung 6: Beispiel zur Publish-Subscribe-Architektur von MQTT [heise.de] 9

Abbildung 7: I²C-Bus mit zwei I²C-Controllern und drei I²C-Targets 11

Abbildung 8: SPI-Bus mit einem SPI-Controller und SPI-Peripheral 11

Abbildung 9: Übersicht über die Verbindungen der DevBox 22

Abbildung 10: Vergleich von I²C, SPI und UART 26

Abbildung 11: Konzept für die Verbindungen der DevBox 27

Abbildung 12: Zusammenhang der Hardwarekomponenten der DevBox 29

Abbildung 13: Entwurf für den inneren Aufbau der DevBox 30

Abbildung 14: Entwurf für die Basis der DevBox 31

Abbildung 15: Entwurf für das Gehäuse der DevBox 32

Abbildung 16: Softwarekomponenten für die Kommunikation über ROS 33

Abbildung 17: Klassendiagramm der GpioClient-Klasse 35

Abbildung 18: Klassendiagramm der I2cClient-Klasse 36

Abbildung 19: Klassendiagramm der SpiClient-Klasse 37

Abbildung 20: Softwarekomponenten für die Kommunikation über MQTT 38

Abbildung 21: Klassendiagramm der MqttClient-Klasse 39

Abbildung 22: Softwarekomponenten für die Erkundung über ROS 2 41

Abbildung 23: Flussdiagramm für den Entdeckungsprozess 43

Abbildung 24: Klassendiagramm der DiscoveryClient-Klasse 43

Abbildung 25: Klassendiagramm der DevBoxClient-Klasse 44
Abbildung 26: Aufbau des Hardwareprototypen für die DevBox........ 45
Abbildung 27: Übersicht und Abhängigkeiten der Softwarepakete (vereinfacht) 46
Abbildung 28: Quellcode für die GpioManager-Klasse (Auszug)........ 47
Abbildung 29: Quellcode für die Ermittlung der MAC-Adresse (vereinfacht)........ 49
Abbildung 30: Quellcode für die Kodierung und Dekodierung von Adressen........ 51
Abbildung 31: Alphabet für die Base62-Kodierung 51
Abbildung 32: Klassendiagramm für DeviceClient und InterfaceClient........ 52
Abbildung 33: Quellcode zur Erzeugung einer DevBoxClient-Instanz 53
Abbildung 34: Klassendiagramm der MqttClient-Klasse (überarbeitet)........ 54

Tabellen

Die folgenden, nach Erscheinung und mit Seitenzahl aufgelisteten Tabellen wurden in dieser Arbeit verwendet. Es handelt sich stets um eigene Darstellungen, sofern keine andere Quelle angeben ist.

Tabelle 1: Definition der Anforderungen an die DevBox 21

Tabelle 2: Vergleich von EnOcean, ZigBee und Z-Wave 24

Tabelle 3: ROS-Services für den Zugriff auf GPIO 34

Tabelle 4: ROS-Topic und Service für GPIO-Ereignisse 35

Tabelle 5: ROS 2 Services für die Kommunikation über I²C 36

Tabelle 6: ROS-Services für die Kommunikation über SPI 37

Tabelle 7: ROS-Service für MQTT-Informationen 39

Tabelle 8: ROS 2 Services für das Abfragen von Geräteinformationen 41

Tabelle 9: ROS-Topics für den Erkundungsprozess 42

Tabelle 10: ROS-Action für den Erkundungsprozess 42

Tabelle 11: Materialien der DevBox und deren Kosten 59

Tabelle 12: Git Repositorys mit Quellcode dieser Arbeit 60

Quellen

Die Arbeit stützt sich auf die folgenden, nach dem *IEEE* Zitierstil angegebenen Literatur- und Internetquellen. Die Quellen sind entsprechend des ersten Verweises im Text sortiert. Ein Verweis im Text erfolgt durch Angabe der entsprechenden Nummer der Quelle in eckigen Klammern (siehe *Hinweise zum Lesen der Arbeit*).

[1] V. Schröter, „Prototypischer Bau eines Tischroboters als Plattform für zukünftige Entwicklungs- und Forschungsprojekte im Bereich des Ambient Assisted Livings und als Alternative zum NAO-Roboter," Technische Hochschule Wildau, 2019.

[2] iCampus Wildau, „ROS-E Tischroboter | iCampus Wildau," [Online]. Available: https://icampus.th-wildau.de/cms/de/roboticlab/ros-e-tischroboter. [Zugriff am 24. April 2022].

[3] O. Lorenz, „Praktikumsbericht," 2021.

[4] Innovation Hub 13, „Innofab_ Ideenwettbewerb Gewinner:innen 2022," Juni 2022. [Online]. Available: https://innohub13.de/innofab-gewinnerinnen-2022. [Zugriff am 27 August 2022].

[5] V. Schröter und O. Lorenz, „Home · Wiki · ROS-E / ROS-E Doku · GitLab," 2 August 2021. [Online]. Available: https://icampusnet.th-wildau.de/ros-e/doku/-/wikis/home. [Zugriff am 9 September 2022].

[6] V. Schröter und T. Kannenberg, „Setup Raspberry Pi 4 · Wiki · ROS-E / ROS-E Doku · GitLab," 27 April 2022. [Online]. Available: https://icampusnet.th-wildau.de/ros-e/doku/-/wikis/Setup/Setup-Raspberry-Pi-4. [Zugriff am 13 September 2022].

[7] V. Schröter, „ROS-E / Software / ROS2-Packages / ROS2 Hardware Interfaces · GitLab," 16 August 2021. [Online]. Available: https://icampusnet.th-wildau.de/ros-e/software/ros2-packages/hardware-interfaces. [Zugriff am 9 September 2022].

[8] Open Robotics, „ROS: Home," 2021. [Online]. Available: https://www.ros.org. [Zugriff am 29 August 2022].

[9] Open Robotics, „ROS: Why ROS?," 2021. [Online]. Available: https://www.ros.org/blog/why-ros. [Zugriff am 29 August 2022].

[10] Open Robotics, „Understanding nodes — ROS 2 Documentation: Humble documentation," 2022. [Online]. Available:

https://docs.ros.org/en/humble/Tutorials/Beginner-CLI-Tools/Understanding-ROS2-Nodes/Understanding-ROS2-Nodes.html. [Zugriff am 29 August 2022].

[11] Open Robotics, „Understanding topics — ROS 2 Documentation: Humble documentation," 2022. [Online]. Available: https://docs.ros.org/en/humble/Tutorials/Beginner-CLI-Tools/Understanding-ROS2-Topics/Understanding-ROS2-Topics.html. [Zugriff am 29 August 2022].

[12] Open Robotics, „Understanding services — ROS 2 Documentation: Humble documentation," 2022. [Online]. Available: https://docs.ros.org/en/humble/Tutorials/Beginner-CLI-Tools/Understanding-ROS2-Services/Understanding-ROS2-Services.html. [Zugriff am 29 August 2022].

[13] Open Robotics, „Understanding actions — ROS 2 Documentation: Humble documentation," 2022. [Online]. Available: https://docs.ros.org/en/humble/Tutorials/Beginner-CLI-Tools/Understanding-ROS2-Actions/Understanding-ROS2-Actions.html. [Zugriff am 29 August 2022].

[14] W. Woodall, „Topic and Service name mapping to DDS," Juni 2018. [Online]. Available: https://design.ros2.org/articles/topic_and_service_names.html. [Zugriff am 29 August 2022].

[15] Open Robotics, „Understanding parameters — ROS 2 Documentation: Humble documentation," 2022. [Online]. Available: https://docs.ros.org/en/humble/Tutorials/Beginner-CLI-Tools/Understanding-ROS2-Parameters/Understanding-ROS2-Parameters.html. [Zugriff am 29 August 2022].

[16] MQTT.org, „The Standard for IoT Messaging," 2022. [Online]. Available: https://mqtt.org. [Zugriff am 29 August 2022].

[17] M. Stal, „Kommunikation über MQTT," 1 Juli 2016. [Online]. Available: https://www.heise.de/blog/Kommunikation-ueber-das-Ethernet-Shield-mit-MQTT-3238975.html. [Zugriff am 29 August 2022].

[18] C. Götz, „MQTT: Protokoll für das Internet der Dinge," 15 April 2014. [Online]. Available: https://www.heise.de/ratgeber/MQTT-Protokoll-fuer-das-Internet-der-Dinge-2168152.html. [Zugriff am 29 August 2022].

[19] The kernel development community, „Introduction — The Linux Kernel documentation," 4 September 2022. [Online]. Available:

https://www.kernel.org/doc/html/latest/driver-api/gpio/intro.html. [Zugriff am 7 September 2022].

[20] NXP Semiconductors, „I2C-bus specification and user manual," 1 Oktober 2021. [Online]. Available: https://www.nxp.com/docs/en/user-guide/UM10204.pdf. [Zugriff am 17 Juli 2022].

[21] Motorola/Freescale/NXP, „SPI Block Guide," 14 Juli 2004. [Online]. Available: https://www.nxp.com/files-static/microcontrollers/doc/ref_manual/S12SPIV4.pdf. [Zugriff am 17 Juli 2022].

[22] S. C. Hill, J. Jelemensky und M. R. Heene, „Queued serial peripheral interface for use in a data processing system". USA Patent 4,816,996, 28 März 1989.

[23] M. Szczys, „Updating The Language Of SPI Pin Labels To Remove Casual References To Slavery | Hackaday," 29 Juni 2020. [Online]. Available: https://hackaday.com/2020/06/29/updating-the-language-of-spi-pin-labels-to-remove-casual-references-to-slavery. [Zugriff am 5 September 2022].

[24] Open Source Hardware Association, „A Resolution to Redefine SPI Signal Names – Open Source Hardware Association," 2022. [Online]. Available: https://www.oshwa.org/a-resolution-to-redefine-spi-signal-names. [Zugriff am 5 September 2022].

[25] EnOcean GmbH, „Funktechnologie - EnOcean," 2022. [Online]. Available: https://www.enocean.com/de/technologie/funktechnologie. [Zugriff am 11 September 2022].

[26] EnOcean GmbH, „EnOcean – The World of Energy Harvesting Wireless," 2020.

[27] EnOcean Alliance Inc., „Security of EnOcean Radio Networks," 15 Januar 2019. [Online]. Available: https://www.enocean-alliance.org/wp-content/uploads/2019/04/Security-of-EnOcean-Radio-Networks-v2_5.pdf. [Zugriff am 15 September 2022].

[28] C. Bertko, „Z-Wave, ZigBee + EnOcean: Du musst dich nicht entscheiden | SmartHome Blog," 13 Mai 2019. [Online]. Available: https://www.siio.de/z-wave-zigbee-enocean-du-musst-dich-nicht-entscheiden. [Zugriff am 12 September 2022].

[29] EnOcean Alliance Inc., „EnOcean Alliance - Produkte "Enabled by EnOcean",“ [Online]. Available: https://www.enocean-alliance.org/de/produkte. [Zugriff am 12 September 2022].

[30] C. Withanage, R. Ashok, C. Yuen und K. Otto, „A comparison of the popular home automation technologies,“ in *2014 IEEE Innovative Smart Grid Technologies - Asia (ISGT ASIA)*, 2014.

[31] Silicon Labs, „Z-Wave SDK 6.71: Introducing S2 Security,“ [Online]. Available: https://www.silabs.com/documents/public/presentations/PMP13827-2.pdf. [Zugriff am 3 Februar 2021].

[32] L. Zimert, „Konzeption und Implementierung eines Künstlichen Neuronalen Netzes zur Situationserkennung mit humanoiden Robotern anhand mehrerer Eingangskanäle,“ Technische Hochschule Wildau, 2022.

[33] Raspberry Pi Trading Ltd., „Raspberry Pi 4 Computer Model B,“ Januar 2021. [Online]. Available: https://datasheets.raspberrypi.com/rpi4/raspberry-pi-4-product-brief.pdf. [Zugriff am 24 Juli 2022].

[34] Apple Inc., Hewlett-Packard Inc., Intel Corporation, Microsoft Corporation, Renesas Corporation, STMicroelectronics, Texas Instruments, „Universal Serial Bus 3.2 Specification,“ 2022.

[35] J. G. Sponås, „Things You Should Know About Bluetooth Range,“ 7 February 2018. [Online]. Available: https://blog.nordicsemi.com/getconnected/things-you-should-know-about-bluetooth-range. [Zugriff am 15 September 2022].

[36] A. Banks, E. Briggs, K. Borgendale und R. Gupta, „MQTT Version 5.0,“ 7 März 2019. [Online]. Available: https://docs.oasis-open.org/mqtt/mqtt/v5.0/mqtt-v5.0.pdf. [Zugriff am 25 Juli 2022].

[37] D. Thomas, „ROS 2 middleware interface,“ September 2017. [Online]. Available: https://design.ros2.org/articles/ros_middleware_interface.html. [Zugriff am 25 Juli 2022].

[38] Object Management Group, „The Real-time Publish-Subscribe Protocol (RTPS) DDS Interoperability Wire Protocol Specification,“ September 2014. [Online]. Available: https://www.omg.org/spec/DDSI-RTPS/2.2/PDF. [Zugriff am 25 Juli 2022].

[39] C. Williams, „Green Power White Paper," 2017. [Online]. Available: https://zigbeealliance.org/wp-content/uploads/2019/11/Green-Power-White-Paper.pdf. [Zugriff am 15 September 2022].

[40] NXP Laboratories UK, Januar 2017. [Online]. Available: https://www.nxp.com/docs/en/supporting-information/MAXSECZBNETART.pdf. [Zugriff am 15 September 2022].

[41] y. „UART vs I2C vs SPI – Communication Protocols and Uses - Latest Open Tech From Seeed," 25 September 2019. [Online]. Available: https://www.seeedstudio.com/blog/2019/09/25/uart-vs-i2c-vs-spi-communication-protocols-and-uses. [Zugriff am 11 September 2022].

[42] E. Peňa und M. G. Legaspi, „UART: A Hardware Communication Protocol Understanding Universal Asynchronous Receiver/Transmitter," *Analog Dialogue,* Bd. 54, 2020.

[43] S. Mishra, N. K. Singh und V. Rousseau, „Chapter 10 - Sensor Interfaces," in *System on Chip Interfaces for Low Power Design*, Morgan Kaufmann, 2016, pp. 331-344.

[44] Atmel Corporation, „AVR910: In-System Programming," 2016. [Online]. Available: http://ww1.microchip.com/downloads/en/appnotes/atmel-0943-in-system-programming_applicationnote_avr910.pdf. [Zugriff am 15 September 2022].

[45] J. Valdez und J. Becker, „Understanding the I2C Bus," Texas Instruments, Juni 2015. [Online]. Available: https://www.ti.com/lit/an/slva704/slva704.pdf. [Zugriff am 17 Juli 2022].

[46] R. Droms, „Dynamic Host Configuration Protocol," 1997.

[47] „IEEE Standard for Local and Metropolitan Area Networks: Overview and Architecture," *IEEE Std 802-2014 (Revision to IEEE Std 802-2001),* pp. 1-74, 30 Juni 2014.

[48] S. Crump, „SMBus Compatibility With an I2C Device," 2009.

[49] „Raspberry Pi GPIO Pinout," 27 Oktober 2021. [Online]. Available: https://pinout.xyz. [Zugriff am 15 September 2022].

[50] J. Adams, P. Elwell und A. Scheller, „hats/designguide.md at master · raspberrypi/hats · GitHub," 9 Januar 2021. [Online]. Available:

https://github.com/raspberrypi/hats/blob/master/designguide.md. [Zugriff am 15 September 2022].

[51] The kernel development community, „ABI testing symbols — The Linux Kernel documentation," 4 September 2022. [Online]. Available: https://www.kernel.org/doc/html/latest/admin-guide/abi-testing.html. [Zugriff am 5 September 2022].

[52] S. Josefsson, „The Base16, Base32, and Base64 Data Encodings," 2006.

[53] Open Robotics, „Synchronous vs. asynchronous service clients — ROS 2 Documentation: Humble documentation," 2022. [Online]. Available: https://docs.ros.org/en/humble/How-To-Guides/Sync-Vs-Async.html. [Zugriff am 15 September 2022].

[54] Compaq, Hewlett-Packard, Intel, Lucent, Microsoft, NEC, Philips, „Universal Serial Bus Specification Revision 2.0," 2000.

[55] Raspberry Pi Ltd, „Raspberry Pi Documentation," 2022. [Online]. Available: https://www.raspberrypi.com/documentation/computers/raspberry-pi.html. [Zugriff am 24 Juli 2022].

[56] G. Ohland, „Der EnOcean-Funkstandard auf dem Prüfstand," *PC Magazin,* 1 März 2012.

Selbstständigkeitserklärung

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbstständig und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe; die aus fremden Werken wörtlich oder sinngemäß übernommenen Gedanken sind unter Angabe der Quellen gekennzeichnet.

____

Oskar Lorenz – Berlin, den 15.09.2022