

Masterarbeit

**Entwicklung eines modularen Smart Device zur Auftrennung
der Funktionalitäten von ‚Smart‘ und ‚Device‘**

Simon Schmieden
Matrikelnummer: 3030542
Angewandte Informatik (Master)



Fachgebiet Verteilte Systeme, Abteilung Informatik
Fakultät für Ingenieurwissenschaften
Universität Duisburg-Essen

21. April 2022

Erstgutachter: Prof. Dr.-Ing. Torben Weis
Zweitgutachter: Prof. Dr. Gregor Schiele
Zeitraum: 22. Oktober 2021 - 22. April 2022

Zusammenfassung

Wenn Produkthersteller den Schritt in den Bereich des Internet of Things machen wollen, müssen sie einige Hürden auf sich nehmen. Eine Menge wichtiger Entscheidungen sind zu treffen und eine Vielzahl unterschiedlicher Smart-Home-Systeme können diesen Schritt deutlich erschweren.

Das Ziel dieser Arbeit ist es, den Herstellern eine Lösung dafür zu bieten, leichter in die Entwicklung von Smart Devices einzusteigen. Um das zu erreichen, wurde sich von der bekannten Struktur eines Smart Device gelöst und die Möglichkeit betrachtet, die *smarten* Funktionen von den Funktionen des Device zu trennen. Die Trennung wurde mithilfe eines Prototyps explorativ umgesetzt. Entwickelt wurde eine Anwendung, in der zwei Smart Devices in zwei ‚Smart‘ und zwei ‚Device‘ aufgeteilt werden. Für die Kommunikation zwischen den einzelnen Komponenten werden serielle Schnittstellen verwendet. Um eine reibungslose Nutzung der Smart Devices zu gewährleisten, wurde aufbauend auf diesen Schnittstellen ein Kommunikationsprotokoll erschaffen. Das Protokoll beschreibt wichtige Abläufe in der Kommunikation zwischen Smart und Device und definiert eine nützliche Paket-Struktur.

Um zu untersuchen, inwieweit der umgesetzte Prototyp für die Anwendung in der Praxis geeignet ist, wurde sowohl das theoretische Design als auch die Umsetzung anschließend analysiert. Für die Analyse der Verzögerung, die durch die zusätzliche Indirektion entsteht, wurden Zeitmessungen durchgeführt.

Die Messungen zeigen, dass die Funktionsweise eines Smart Device durch die Trennung nicht eingeschränkt wird. Wird die Praxistauglichkeit serieller Schnittstellen aufgrund ihrer begrenzten Reichweite außer Acht gelassen, ergeben sich aus dem Beispiel dieser Arbeit einige Möglichkeiten, um Herstellern den Schritt in Richtung des Internet of Things zu erleichtern.

Inhaltsverzeichnis

1 Einleitung	1
2 Grundlagen	3
2.1 Smart Devices	4
2.2 Digital Twins	6
2.3 Smart Homes	7
2.3.1 HomeKit	8
2.3.2 AWS IoT	9
2.4 Vernetzung	10
2.5 Übertragungsmedien	11
2.5.1 Kabellose Kommunikation	12
2.5.2 Kabelgebundene Kommunikation	15
3 Verwandte Arbeiten	19
3.1 Smart Labs	19
3.2 Homebridge	23
3.3 Matter	23
4 Vorabanalyse	25
4.1 Eingliederung und Abgrenzung	25
4.2 Fragestellungen	26
4.3 Anforderungen	28
4.3.1 Anforderungen an Smart Devices	28
4.3.2 Anforderungen an die Auf trennung	29
5 Design	31
5.1 Architektur	31
5.2 Device	33
5.3 Smart	35
5.4 Digital Twin	35
5.4.1 Abbild auf Device	36
5.4.2 Abbild auf Smart	37
5.4.3 Auswahl	38
5.5 Kommunikation	38
5.5.1 Übertragungsmedium	39

5.5.2 Protokoll	40
6 Implementierung	57
6.1 Konfiguration	57
6.1.1 Komponenten	58
6.1.2 Aufbau	59
6.1.3 Frameworks und Sprachen	60
6.2 Ausführung	62
6.2.1 Smart	63
6.2.2 Device	74
6.2.3 Kommunikation	78
6.3 Fehlerbehandlung	83
6.3.1 Fehlererkennung	83
6.3.2 Fehlerbehebung	84
7 Evaluation	87
7.1 Messungen	87
7.1.1 Vorgehen	88
7.1.2 Erwartungen	90
7.1.3 Durchführung	94
7.1.4 Ergebnisse	94
7.2 Diskussion	99
7.2.1 Digital Twin	99
7.2.2 Ausgewählte Schnittstelle	100
7.2.3 Erkennung eines Device	100
7.2.4 Protokoll	101
7.2.5 Fehlerbehandlung	102
7.3 Fragestellungen	102
8 Fazit	105
8.1 Ausblick	106
Abkürzungen	107
Literatur	109
Weitere Quellen	113

Abbildungsverzeichnis

2.1	HomeKit-Architektur	8
2.2	AWS-Architektur	9
2.3	Vergleich der Topologien von UART, I ² C und SPI	15
3.1	Architektur des Living Lab Gateway	22
5.1	Architektur	32
5.2	Eigenschaften eines Kühlschranks	34
5.3	Mögliche Umsetzungen des Digital-Twin-Paradigmas	36
5.4	Aufbau einer Nachricht	40
5.5	Vergleich der Checksums	42
5.6	Beispielnachrichten	43
5.7	Erfolgreicher Nachrichtenaustausch	45
5.8	Startup-Sequenz	46
5.9	Fehlerfälle	49
5.10	Erkennung eines fehlerhaften State	51
5.11	Erkennung: Neuaufbau notwendig?	52
5.12	Neuaufbau ohne State-Invalidation	53
6.1	Aufbau des Test-Setups	58
6.2	Darstellung des Setups mit Fritzing	60
6.3	Einrichtungs- und Anmeldeprozess des Smart	63
6.4	Simplifizierter Aufbau des Smart	64
6.5	HAP Accessory Specification und Accessory Configuration Object	65
6.6	Registrierung der Bridge in der Home App	66
6.7	UML-Klassendiagramm: Device Factory	68
6.8	Eingebundene Devices in der Home App	70
6.9	HAP-Spezifikation: Switch	71
6.10	Vergleich: Target/Current Heating Cooling State	76
6.11	Nachrichtenaustausch	78
6.12	Für Nutzer sichtbarer Programmablauf	79
6.13	Vergleich: Nachrichtenaustausch von UART und I ² C	83
6.14	Aufgaben des Observer-Threads	84
7.1	Gemessene Konfigurationen	89

Abbildungsverzeichnis

7.2	Vergleich: Pakete von UART und I ² C	91
7.3	Effizienz der gesendeten Nachrichten nach Größe	92
7.4	Messergebnisse (Konfiguration 1 und 2)	94
7.5	Messergebnisse (Konfiguration 3 und 4)	95
7.6	Messergebnis (Konfiguration 5)	96
7.7	Sonderfall: State-C checksum korrekt, State fehlerhaft	101

Kapitel 1

Einleitung

Wer als Hersteller wettbewerbsfähig bleiben möchte, kann heutzutage schnell zu der Entscheidung gelangen, seine Produkte mit einer gewissen *Smartness* ausstatten zu wollen. Bereits die kleinsten und simpelsten Geräte können autonom arbeiten und von unterschiedlichen Orten aus gesteuert werden. Daher könnte angenommen werden, dass die Entwicklung solcher Geräte kein großes Problem darstellt.

Doch ein Hersteller, der in das Themengebiet des Internet of Things (IoT) einsteigen möchte, kann sich bereits nach einer kurzen Recherche mit einer Menge verschiedener Fragen konfrontiert sehen: „*Welche Smart-Home-Systeme sollen unterstützt werden?*“, „*Welche Übertragungsmethode soll genutzt werden?*“, „*Wie sollen Updates bereitgestellt werden?*“. Das ist lediglich ein Bruchteil der Fragen, die sich dieser Hersteller stellen muss.

Insbesondere für Unternehmen, deren Expertise nicht im informationstechnischen Bereich liegt, kann dieser Schritt eine große Herausforderung sein. Zusätzlich zu der Notwendigkeit für neue informatorische Fähigkeiten müssen Hersteller von Smart Devices beispielsweise viele Dinge beachten, die sich aus der heterogenen Zusammensetzung der Smart-Home-Systeme ergeben.

Es gibt eine Vielzahl unterschiedlicher IoT-Frameworks, die alle verschiedene, teilweise offene und teilweise auch geschlossene Application Programming Interfaces (APIs) anbieten. Alle Frameworks mit einem Gerät zu unterstützen, kann sehr aufwendig sein. Einzelne Systeme wie das AWS IoT oder das Apple HomeKit haben sich in der breiten Masse durchgesetzt. Bereits die Unterstützung dieser zwei Systeme erfordert ein großes Maß an Aufwand und Fachwissen.

Umso schwieriger wird es, sobald Komponenten ausfallen oder sich die APIs der Smart-Home-Systeme ändern. Um die Wartbarkeit, eine der wichtigsten Eigenschaften von Smart Devices, zu gewährleisten, muss sich der Hersteller Gedanken machen, wie er seine Geräte rechtzeitig, simpel und zuverlässig warten oder aktualisieren lassen kann.

Das alles sind für unerfahrene Hersteller große Hürden, die den Schritt in Richtung Smart Devices erschweren. In dieser Arbeit wird versucht, auch diesen Herstellern einen solchen Schritt zu ermöglichen, ohne dass sie über viel Erfahrung mit Smart Devices und eingebetteten Systemen verfügen müssen. Die Eigenschaften, die ein Gerät *smart* machen, sollen von den grundlegenden Funktionen des Gerätes getrennt und in separate

Komponenten ausgelagert werden. Aus einem Smart Device entsteht ein ‚Smart‘ und ein ‚Device‘. Diese Komponenten kommunizieren über geeignete physische Schnittstellen miteinander und tauschen notwendige Daten aus. So kann der Hersteller jegliche Verantwortung, die er bezüglich eines Smart Device hat, abgeben und sich weiterhin auf die Kernfunktionen seines Gerätes (‚Device‘) konzentrieren. Um die Funktionalitäten des ‚Smart‘ kann sich ein Unternehmen kümmern, welches über mehr Erfahrung und Fachwissen in dem Bereich verfügt. Der Device-Hersteller muss ausschließlich das von dem Smart vorgegebene Protokoll und seine API unterstützen.

Das Ziel dieser Arbeit ist es, diese Trennung in einem explorativen Ansatz vorzunehmen. Es gilt die Vor- und Nachteile, Limitierungen sowie Probleme dieser Herangehensweise zu analysieren. Es soll sich damit beschäftigt werden, ob diese Trennung sinnvoll ist, welche Übertragungsmedien geeignet sind und was für eine erfolgreiche Umsetzung beachtet werden muss. Ein großer Teil dieser Arbeit befasst sich daher mit dem Design und der Entwicklung eines Prototyps, bei dem die ‚Smart‘- und ‚Device‘-Funktionalitäten in unterschiedliche Hardware-Komponenten ausgelagert werden. Ein wichtiger Bestandteil ist dabei die Konstruktion eines geeigneten Kommunikationsprotokolls. Der vorgestellte Ansatz wird in einem abschließenden Schritt analysiert und bezüglich seiner Praxistauglichkeit diskutiert.

Die Arbeit ist folgendermaßen aufgebaut:

Kapitel 2: Grundlagen

Beschreibt für das Verständnis dieser Arbeit notwendige Konzepte.

Kapitel 3: Verwandte Arbeiten

Fasst Arbeiten zusammen, die sich einem ähnlichen Vorhaben widmen.

Kapitel 4: Vorabanalyse

Analysiert die vorgestellten Forschungen und Konzepte in Bezug auf ihren Nutzen in dieser Arbeit.

Kapitel 5: Design

Beschreibt Design-Entscheidungen für die Entwicklung eines Prototyps.

Kapitel 6: Implementierung

Erläutert den Entwicklungsprozess und die Umsetzung des Prototyps.

Kapitel 7: Evaluation

Evaluiert das Design und die Umsetzung der Anwendung anhand zuvor erarbeiteter Kriterien.

Kapitel 8: Fazit

Fasst die wichtigsten Aspekte dieser Arbeit zusammen und liefert ein abschließendes Fazit bezüglich der beobachteten Ergebnisse.

Kapitel 2

Grundlagen

Um die in dieser Arbeit aufgegriffenen Begriffe und Technologien verstehen und einzuordnen zu können, werden in diesem Kapitel relevante Konzepte vorgestellt.

Dabei wird zunächst auf verschiedene Definitionen von Smart Devices in der Literatur eingegangen. Anschließend werden die Eigenschaften betrachtet, die ihnen aus unterschiedlichen Quellen zugeschrieben werden. Da sich diese Arbeit intensiv mit Smart Devices und der Veränderung ihres bestehenden Designs beschäftigt, ist eine nähere Be trachtung ihrer Hauptmerkmale und Funktionen essenziell.

Die in diesem Kontext zusammengefassten Eigenschaften werden im späteren Verlauf der Arbeit mehrfach aufgegriffen.

Im Anschluss wird ein Überblick über die Definition sowie den Aufbau verschiedener Smart-Home-Systeme gegeben. Dieser Überblick soll dabei helfen, ein Verständnis dafür zu bekommen, wie die Systeme und Frameworks arbeiten, mit denen die Smart Devices verknüpft werden. Der Fokus liegt hier auf den Unterschieden zwischen den bestehenden Frameworks, ihren APIs und der Einbindung der Smart Devices in die Systeme.

Aufgrund der großen Anzahl verschiedener, individueller Komponenten ist ein wichtiger Bestandteil des IoT die Kommunikation zwischen ihnen. Das betrifft sowohl die klassischen Strukturen von Smart Home und Smart Device, als auch das explorative Vorhaben das ‚Smart‘ und das ‚Device‘ voneinander zu trennen. Daher beschäftigen sich die letzten beiden Abschnitte dieses Kapitels mit der Verknüpfung von Smart Devices. Im Fokus liegen dabei die Schwierigkeiten und Problemstellungen, die bei einer Vernetzung beachtet werden sollten.

Im letzten Abschnitt werden dazu einige Übertragungsmedien, die im IoT häufig verwendet werden, vorgestellt und miteinander verglichen.

Mithilfe der darin erarbeiteten Erkenntnisse werden diese Medien in den späteren Kapiteln bezüglich ihrer Eignung für den Anwendungszweck in dieser Arbeit diskutiert.

2.1 Smart Devices

Das Konzept der Smart Devices ist nicht so eindeutig, wie es zunächst erscheint. Im Kern ist die Idee, die diesen Geräten zugrunde liegt, in vielen unterschiedlichen Quellen ähnlich. Worin sie sich allerdings in vielen Fällen unterscheiden, sind die Begriffe und Eigenschaften, die diesen Geräten zugeschrieben werden.

Im Jahre 2005 hat sich Craig W. Thompson [33] an einer Definition versucht. Er beschreibt Smart Objects als Geräte, die dem Nutzer eine Menge Arbeit abnehmen sollen. Er erwähnt, dass die Komplexität durch den technischen Fortschritt weiter zunimmt. Thompson [33] sieht den Sinn von Smart Objects darin, die Nutzung von Geräten simpler und einsteigerfreundlicher zu machen. Die aufwendigen Einrichtungs- und Wartungsarbeiten, die mit den meisten elektronischen Geräten einhergehen, sind Dinge, um die sich ein solches Gerät eigenständig kümmern sollte [33]. Ein Smart Object sollte dem Nutzer die Arbeit erleichtern und nicht zusätzlich welche erzeugen.

Thompson [33] beschreibt bestimmte Anforderungen an ein solches Smart Device.

Es sollte:

- die Möglichkeit geben mit ihm kommunizieren zu können (*Communications*)
- eindeutig identifizierbar sein (*Identity and Kind*)
- einen Speicher haben und Zustände speichern können (*Memory and Status Tracking*)
- seine Umgebung wahrnehmen und in sie eingreifen können (*Sensing and Actuating*)
- eine logische Denkfähigkeit besitzen und lernfähig sein (*Reasoning and Learning*) [33]

Diese Anforderungen müssen nicht von jedem Smart Device erfüllt werden. Wenn ein Gerät die Anforderungen nicht erfüllt, sollte eine Entsorgung und anschließende Neuanschaffung nicht die Lösung sein [33]. Das Gerät sollte es ermöglichen diese und noch viele weitere Funktionen nachträglich hinzuzufügen [33]. Thompson verdeutlicht damit die Bedeutung der Erweiter- und Wartbarkeit von Smart Devices.

Er beschreibt zudem einige Fähigkeiten, über die ein zusammenhängendes System von Smart Devices verfügen könnte. Er nennt dabei unter anderem die Aspekte: *Controllability, Maintainability, Interoperability* und *Reliability* [33].

Thompsons Auffassung eines Smart Device umfasst also alleinstehend bereits eine Menge an Eigenschaften und Anforderungen. Wird die Gesamtheit der wissenschaftlichen Arbeiten betrachtet, ergeben sich deutlich mehr unterschiedliche Definitionen.

In einer Nachforschung von Silverio-Fernández, Renukappa und Suresh [30] wurde der Versuch unternommen, verschiedene Definitionen und Charakteristika von Smart Devices aus der Literatur zusammenzufassen und zu vergleichen. Anschließend wurde versucht eine allgemeingültige und klare Definition für Smart Devices bereitzustellen.

Dazu haben sie zunächst die beiden Stichwörter „Smart Device“ und „Mobile Device“ als hauptsächliche Bezeichner für diese Art von Geräten ermittelt [30]. Zusammen mit dem Begriff des „Internet of Things“ wurde anhand dieser Begriffe anschließend eine Literaturrecherche durchgeführt. Sie fasst die grundlegenden Merkmale zusammen, die diesen Objekten zugeschrieben werden. Aus den verschiedenen Quellen erarbeiteten sie die häufig genannten Charakteristika: *Connectivity, User-Interaction, Autonomy,*

Context-Awareness und *Mobility* [30]. Um zu ergründen, ob sie sich für eine exakte Definition von Smart Devices eignen würden, wurden diese Eigenschaften im Kontext des IoT näher betrachtet.

Die Charakteristika *User-Interaction* und *Mobility* wurden dabei aussortiert [30]. Sie werden zwar häufig genannt und sind auch in Bezug auf die Funktion einiger Smart Devices wichtig, sind aber nicht ausschlaggebend für eine genaue Definition. Sie kamen zu dem Schluss, dass der Begriff des „Mobile Device“, und damit zusammenhängend die *Mobility*, nicht mit der Definition des IoT kompatibel ist. Das IoT beschreibt physische Gegenstände, die miteinander verbunden sind und kommunizieren können. Dabei ist es unerheblich, ob das Gerät portabel ist oder nicht [30].

Die Zuordnung der *Mobility* entspringt nach Silverio-Fernández et al. [30] möglicherweise aus der Entwicklung von Geräten wie Smart Phones, Smart Watches, Tablets und Co. Sie können aufgrund ihrer Funktion und Eigenschaften auch als Smart Devices betrachtet werden. Das „Mobile Device“ bezeichne vielmehr eine Erweiterung bzw. eine Unterkategorie von Smart Devices, die zusätzlich die Eigenschaft erfüllen, handlich und portabel zu sein (Smart Mobile Device) [30].

Die *User-Interaction* ist ebenfalls ein großer Bestandteil vieler Smart Devices. Das Ändern von Parametern und das Erteilen von Anweisungen ermöglichen dem Nutzer viele Funktionen. Für das IoT bezieht sich die Vernetzung allerdings eher auf die Verbindung der Dinge untereinander [30]. So können beispielsweise auch Smart Devices existieren, mit denen der Nutzer nicht interagieren kann [30]. Sie sind mit anderen Geräten verbunden, können mit ihnen Daten austauschen, diese verarbeiten und bestimmte Funktionalitäten bereitstellen.

Als die drei Hauptmerkmale, die ein Gerät *smart* machen, stellen sie also die Merkmale *Connectivity*, *Autonomy* und *Context-Awareness* heraus [30].

Autonomy beschreibt hier die autonome Erledigung von Aufgaben und die Durchführung von Berechnungen, ohne die direkten Anweisungen eines Nutzers zu benötigen [30]. Diese Eigenschaft wird beispielsweise bei Thermostaten deutlich. Ihre Aufgabe ist es, Sensorwerte weiterzuverarbeiten, zu übermitteln und auf sie zu reagieren.

Connectivity bezieht sich auf die Fähigkeit des Gerätes, sich mit einem Netzwerk beliebiger Größe wie dem Internet oder einem lokalen Netz verbinden zu können [30]. Dazu sind kabellose oder kabelgebundene Kommunikationsschnittstellen nötig.

Das Konzept der *Context-Awareness* beschreibt die Fähigkeit, über Sensoren Informationen aus der Umgebung wahrzunehmen und für den Nutzer und das eigene System verwendbar zu machen.

Zusammenfassend beschreibt das Konzept des Smart Device also elektronische Geräte, die über Sensoren und/oder Aktuatoren verfügen, Berechnungen durchführen und mit Hilfe von Übertragungsmedien mit anderen Geräten oder einem Netzwerk kommunizieren können.

2.2 Digital Twins

Ein Konzept, welches im Kontext des IoT häufig in Erscheinung tritt, ist das der Digital Twins.

Da sich das Grundprinzip des IoT auf die Vernetzung von physischen Objekten bezieht, ist ein wichtiger Schritt in der Entwicklung die Frage nach einer passenden digitalen Repräsentation dieser Objekte. Um mit den Objekten interagieren und kommunizieren zu können, sollten sie über einige Informationen verfügen. Sie sollten wissen, welche Eigenschaften diese Geräte haben, wie sie sich bei der Interaktion verhalten und wie sich ihre Umgebung entsprechend verändern wird. Das alles kann durch geeignete Sensoren erfasst und in sinnvoller Form festgehalten werden. Für diesen Zweck werden Digital Twins eingesetzt. Das Konzept beschreibt die Erstellung eines digitalen Gegenstücks von einem physischen Objekt. Dabei wird das Objekt, sein aktueller Zustand und seine Aufgabe durch Eigenschaften und Funktionen eindeutig abgebildet.

Doch auch bei Digital Twins ist eine genaue Definition schwer zu finden.

Roberto Minerva, Gyu Myoung Lee und Noël Crespi [22] haben sich damit beschäftigt, unterschiedliche Definitionen und Eigenschaften von Digital Twins im Kontext des IoT zu sammeln und diese auf ein gesamtheitliches Konzept herunterzubrechen.

Nach ihnen konvergieren verschiedene Definitionen auf einem grundlegenden Level [22]. Alle Definitionen beschreiben, dass ein Digital Twin aus zwei Entitäten besteht, einer physischen und einer logischen. Werden jedoch die Eigenschaften dieser Entitäten betrachtet, ergeben sich einige Unterschiede. Sie sind häufig kontextabhängig und domänen spezifisch [22]. Bestimmte Anwendungsfälle benötigen grafische Repräsentationen und Parameter über die genaue Beschaffenheit des Objektes. Andere benötigen lediglich wenige Werte, um das Objekt abzubilden.

Minerva et al. [22] legen den Fokus dabei auf bestimmte Eigenschaften, die für einen Digital Twin gelten müssen. Die wichtigste Eigenschaft ist ihres Erachtens nach die *Identity*. Ebenso wie sein physisches Gegenstück muss ein logisches Objekt eindeutig identifiziert werden können [22]. Es sollte eindeutig auf das physische Objekt verweisen und von anderen Objekten unterschieden werden können [22]. Dazu kann eine ID verwendet werden. Kopien, die sich auf dasselbe physische Objekt beziehen, müssen dieselbe ID besitzen. Weitere wichtige Eigenschaften seien unter anderem: *Representativeness*, *Reflection* und *Replication* [22].

Die *Representativeness* bezieht sich darauf, dass es in vielen Fällen schwierig ist, das physische Objekt komplett und im kleinsten Detail zu repräsentieren. In den meisten Fällen ist das nicht nötig. Es ist ausreichend, wenn das logische Objekt die für den Einsatz zweck nötigen Eigenschaften und Verhaltensweisen des physischen Objektes abbildet [22]. Ein logisches Objekt muss also nicht alle möglichen Eigenschaften digitalisieren, sondern lediglich die für den Kontext wichtigen. Die Farbe des Lichtes in einem Kühlschrank ist beispielsweise nicht so wichtig wie die Frage, ob das Licht oder die Kühl funktion des Kühlschranks an- oder ausgeschaltet ist.

Reflection beschreibt die Eigenschaft, dass ein logisches Objekt den Zustand des phy-

sischen exakt erfassen und widerspiegeln muss [22]. Der Fokus liegt darauf, dass die gemessenen Daten dem Kontext entsprechend behandelt werden und den Zustand des Objektes in genau diesem Anwendungsfall abbilden.

Replication behandelt die Eigenschaft der Digital Twins, dass eine große Anzahl digitaler Kopien einer physischen Entität existieren kann, solange sie alle eindeutig auf diese zurückzuführen sind und ihren Zustand alle entsprechend gleich abbilden [22]. Dabei beschreiben Minerva et al. [22], dass es eine Master Replica geben muss, anhand derer eine Form der Synchronisierung zwischen den Kopien durchgeführt werden kann.

2.3 Smart Homes

Das Konzept des Smart Homes beschreibt die Vernetzung von Smart Devices im gesamten Haus. Um mit seinen Smart Devices, die an unterschiedlichen Orten im Haus verteilt sein können, einheitlich kommunizieren zu können, bedarf es einer zusätzlichen Architektur.

Wenn sie mit einer geeigneten Kommunikationsinfrastruktur ausgestattet sind, sind die meisten Smart Devices in der Lage ohne zusätzliche Software angesteuert zu werden. Da das allerdings bereits bei einer geringen Anzahl von Geräten aufwendig und komplex werden kann, wurden Systeme geschaffen, die diese Aufgabe zentral für eine große Anzahl von Geräten übernehmen können. Diese Systeme stellen eine Menge an Zusatzfunktionalitäten bereit. Sie kümmern sich beispielsweise um relevante Sicherheitsaspekte, Logging oder auch essenzielles Session-Management sowie Prozesse zum reibungslosen Hinzufügen neuer Geräte. Sowohl der Aufbau als auch die Funktionalitäten der Systeme sind allerdings nicht einheitlich vorgegeben.

Es existieren viele verschiedene Smart-Home-Frameworks. Sie sind oftmals unterschiedlich aufgebaut und ihre Protokolle sind inkompatibel zueinander. Anstatt sich an eine einheitliche Spezifikation von Smart Devices anzupassen und diese zu unterstützen, werden Smart Devices dazu gezwungen, jedes Smart-Home-Framework, welches sie unterstützen wollen, explizit zu bedienen und ihre Protokolle zu implementieren.

In einer Zusammenstellung von Ammar, Russello und Crispo [5] haben sie unterschiedliche IoT-Frameworks in Bezug auf ihre Security-Maßnahmen analysiert und verglichen. Dabei haben sie den Aufbau sowie die Protokolle dieser Frameworks genauer untersucht. Um die großen Abweichungen in den Ansätzen unterschiedlicher Anbieter hervorzuheben, wird im Folgenden der Aufbau von zwei dieser Frameworks näher betrachtet.

2.3.1 HomeKit

HomeKit ist ein von Apple entwickeltes IoT-Framework, welches sich auf die Verbindung von Smart Accessories innerhalb eines Hauses spezialisiert [5].

Die Funktionen dieses Frameworks können durch die hauseigene Home App auf iOS- oder macOS-kompatiblen Geräten genutzt werden. Durch sie erlangt ein Nutzer die Möglichkeit, unterstützte Geräte einzubinden, zu konfigurieren und zu steuern [5].

Mithilfe der Apple-internen Infrastruktur bietet das Framework zusätzlich wichtige Funktionen wie die Nutzer-Authentifizierung und weitere Security-Mechanismen. Ein simplifizierter Aufbau des Frameworks ist in Abbildung 2.1 dargestellt.

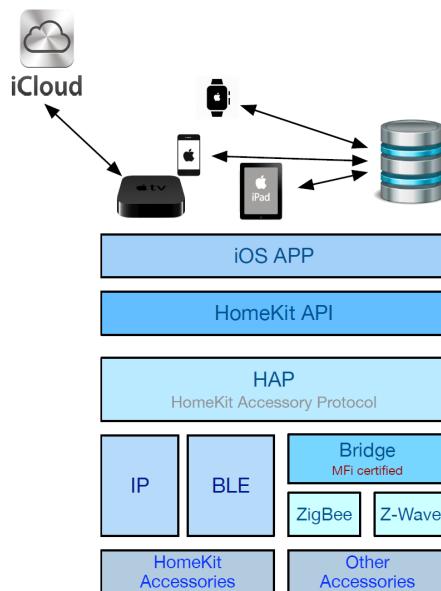


Abbildung 2.1: HomeKit-Architektur [5]

Auf der untersten Ebene liegen die Smart Devices. Im HomeKit-Kontext werden sie „HomeKit Accessories“ genannt. Um direkt in das Framework eingebunden zu werden, müssen diese Geräte bestimmte Anforderungen erfüllen. Die Anforderungen wurden von Apple in Spezifikationen zusammengefasst.

Erfüllt ein Gerät diese Spezifikationen nicht, so kann es dennoch mithilfe einer separaten HomeKit Bridge eingebunden werden [5]. Diese Bridge kann zwischen den Framework-eigenen Spezifikationen und dem Smart Device vermitteln. Dabei ist die Implementierung einer solchen Bridge abhängig von dem einzubindenden Smart Device und dem damit verbundenen Hersteller.

Ein Projekt, welches den Einsatz einer Bridge zur Einbindung von nicht unterstützten

Geräten nutzt, wird im späteren Verlauf dieser Arbeit vorgestellt.

Wie die Kommunikation zwischen den Devices und dem HomeKit-Framework auszusehen hat, wird durch das HomeKit Accessory Protocol (HAP) beschrieben [5]. Das Protokoll wird als zusätzliche Ebene auf die vorhandenen Protokolle aufgesetzt. Für den Datenaustausch zwischen den HomeKit-Anwendungen und den Geräten wird das JSON-Format verwendet.

Mit der HomeKit API wird Entwicklern eine zusätzliche API für die Entwicklung von separaten Anwendungen bereitgestellt. Alternativ kann die von Apple entwickelte iOS App genutzt werden.

Um weitere Funktionen wie beispielsweise die Nutzung der Dienste von außerhalb des eigenen Heimnetzes zu gewährleisten, ist die Einbindung eines Apple TVs, Homepods oder iPads notwendig [6]. Sie dienen als Gateways und leiten die Anfragen aus dem Internet in das eigene Heimnetz sowie an die entsprechenden Geräte weiter. Zudem ermöglichen sie weitere Funktionen wie die Erstellung von komplexen Szenen für die Heimautomation [6].

2.3.2 AWS IoT

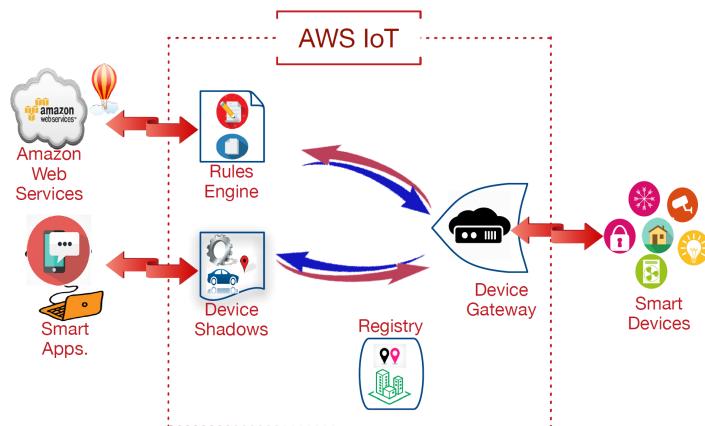


Abbildung 2.2: AWS-Architektur [5]

AWS IoT ist eine von Amazon entwickelte Cloud-basierte Umgebung für die Einbindung von IoT-Geräten [5]. Sie wurde explizit entwickelt, um Smart Devices einfach und sicher in das bestehende AWS-Cloud-System integrieren zu können. Dadurch können Funktionen wie die Nutzung der eigenen Sprachsteuerung oder Dienste wie die DynamoDB in Kombination mit den Smart Devices bereitgestellt werden [5].

Die Architektur des AWS IoT ist in Abbildung 2.2 zu sehen. Sie besteht aus vier Hauptkomponenten. An erster Stelle in der Kommunikation mit den Smart Devices steht das

Device Gateway. Ähnlich wie das Gateway in der HomeKit-Anwendung dient es als Vermittler zwischen den verbundenen Geräten und den Cloud-Diensten [5].

Für die Kommunikation zwischen den Geräten und dem Gateway wird das MQTT-Protokoll verwendet [5]. Es beruht auf einem Publish-And-Subscribe Prinzip. Das bedeutet, Nachrichten von den Devices werden als Broadcasts auf bestimmten Kanälen veröffentlicht. Diese Aufgabe übernimmt ein MQTT-Broker auf der Seite des Servers. Möchte ein Client wissen, ob auf einem Kanal etwas gesendet wurde, muss er diesen lediglich abonnieren und auf Nachrichten warten. Dadurch können viele Geräte ihre Statusänderungen und Parameter an das Gateway senden, ohne dass jedes Gerät direkt mit dem Gateway verbunden sein muss. Die einzige Verbindung, die bestehen muss, ist die zu einem dedizierten MQTT-Server.

Ein großer Vorteil ist hier, dass es unerheblich ist, von wem genau die Anfragen an die Smart Devices getätigt wurden. Sie empfangen Anfragen auf einem bestimmten Kanal und führen diese daraufhin aus. Die Funktion ist somit nicht auf die Nutzung der Cloud limitiert und kann auch von außen angesteuert werden [5].

An der letzten Stelle vor der AWS-Cloud befindet sich die Rules Engine. Sie verarbeitet die empfangenen Nachrichten und sendet sie an diverse Cloud-Dienste weiter [5]. Dort können weitere Funktionen wie die Verarbeitung, Abspeicherung und Analyse der Daten erfolgen. Mithilfe von Amazon Lambda können eigene Funktionen erstellt werden [5].

Die Registry kümmert sich um die Verwaltung der Geräte, ihrer zugehörigen Unique-IDs sowie ihrer Metadaten. Angelehnt an das Digital-Twin-Konzept, arbeiten die Clients auf einem Device-Shadow, welches das physische Gerät digital repräsentiert. Änderungen werden zunächst auf dem Device-Shadow durchgeführt [5]. Sie werden auf lokaler Ebene erfasst und gespeichert. Das geschieht auch, falls das Gerät Offline ist. Verbindet es sich erneut mit den AWS-Diensten, werden die Daten synchronisiert [5].

Diese Abstraktionsebene bietet den Vorteil, dass ein Entwickler mit einem standardisierten Objekt arbeiten kann, welches unabhängig von den genutzten Technologien gleich aufgebaut und angesprochen werden kann [5].

2.4 Vernetzung

Smart Devices müssen in der Lage sein, Anweisungen entgegenzunehmen und Informationen zu übermitteln. Aus diesem Grund ist die *Connectivity* ein wichtiger Faktor im IoT.

In einer Arbeit von S. Sujin Issac Samuel [29] wurden verschiedene Herausforderungen erarbeitet, die sich aus der Vernetzung von Smart Devices ergeben. Sie sollten sowohl bei dem Design des Gerätes als auch bei der Auswahl einer Kommunikationsschnittstelle berücksichtigt werden.

Als eine wichtige Herausforderung nennt Samuel unter anderem die *Interoperability* [29]. Sie beschreibt die Fähigkeit von zwei oder mehr Systemen, Informationen miteinander

auszutauschen und diese zu nutzen. Da sich das Gesamtsystem aus vielen unterschiedlichen Geräten zusammensetzt, ist es wichtig, dass alle Geräte verlässlich zusammenarbeiten.

Mit *Self-Management* erweitert Samuel seine Aufzählung um einen weiteren Punkt [29]. Es beschreibt die Fähigkeit des Gerätes, seine eigenen Funktionen kontrollieren sowie steuern zu können. Falls gewisse Funktionen nicht gewährleistet werden können, soll das Gerät gegebenenfalls Benachrichtigungen senden [29]. Dabei soll die Kontrolle der Sensoren und die Datenverarbeitung im Gerät selber stattfinden können. Die Maßnahmen, die ergriffen werden, sollen unabhängig von der Intervention eines Nutzers sein [29].

Die *Maintainability* ist eine Herausforderung, die sich besonders auf die Praxistauglichkeit der Geräte bezieht. Sie beschreibt die Fähigkeit des Systems, schnell an veränderte Gegebenheiten wie Protokolländerungen angepasst zu werden [29]. Diese Anpassbarkeit kann sich neben Änderungen in der direkten Umgebung ebenso auf andere Einflüsse wie Netzwerkprobleme oder defekte Komponenten beziehen [29].

Für ein Gerät in einem vernetzten Smart Home gilt es, diese Probleme zu erkennen und sich entsprechend anzupassen.

Ein weiterer wichtiger Aspekt, der zu beachten ist, ist das *Signaling* [29]. Es soll sicher gestellt werden, dass die gesendeten Daten beim Empfänger ankommen. Dazu ist eine verlässliche Datenübertragung nötig. Wenn sie nicht gewährleistet werden kann, müssen andere Maßnahmen ergriffen werden.

Abhängig davon, welche Aufgabe ein Gerät übernimmt und damit auch welche Daten übertragen werden müssen, ist die Betrachtung der *Bandwidth* zusätzlich nötig [29]. Da es sich bei diesen Geräten auch um mobile Geräte mit einer limitierten Stromversorgung handeln kann, muss in manchen Fällen auch die *Power-Consumption* beachtet werden [29].

Diese Herausforderungen ergeben sich für viele Devices, die im IoT-Kontext eingesetzt werden soll.

2.5 Übertragungsmedien

Entscheidend für die Kommunikation innerhalb des Smart Homes ist die Auswahl eines Übertragungsmediums. Bereits ohne eine Auftrennung der Funktionalitäten von ‚Smart‘ und ‚Device‘ stellt sich die Frage, welches Medium für die Kommunikation mit dem Gerät am besten geeignet ist.

Durch die Auftrennung kommt eine zusätzliche Indirektion und damit auch eine weitere Kommunikationsschnittstelle hinzu.

Hierbei gibt es viele unterschiedliche Technologien, die im IoT verwendet werden. Sie unterscheiden sich in Eigenschaften wie Reichweite, Datenrate, Störanfälligkeit, Stromverbrauch sowie in der Netzwerktopologie. Im Folgenden werden einige von ihnen genauer betrachtet und hinsichtlich ihrer Eignung im Kontext des IoT beurteilt.

2.5.1 Kabellose Kommunikation

Die kabellose Kommunikation kann in einem gewissen Rahmen eine standortunabhängige sowie einfache Einbindung von Geräten ermöglichen. Das ist im Hinblick auf das IoT und seine Grundidee, der Verknüpfung einer großen Anzahl von Geräten, ein wichtiger Punkt. In vielen Fällen wird eine kabellose Kommunikation bevorzugt. Allerdings kann das auch mit einigen Einschränkungen einhergehen.

Kabellose Kommunikation basiert auf der Emission elektromagnetischer Wellen. Aus diesem Grund gelten für sie dieselben Eigenschaften und Einschränkungen wie für viele Arten physikalischer Strahlung. Dazu zählen beispielsweise die Effekte *Scattering*, *Absorption* und *Diffraction* [21]. Diesen und weiteren Effekten sind Wellen in der Propagation durch die Luft ausgesetzt. Sie führen zu Störungen, verzerrten oder inkorrekteten Signalen und können viel Unsicherheit bezüglich der Verlässlichkeit der Übertragung mit sich bringen [21].

Zudem werden die zur Übertragung genutzten Wellen über bestimmte Frequenzen gesendet. Zur Einordnung dieser Frequenzen in spezifische Frequenzbereiche, die nicht vom Militär oder anderen Organisationen genutzt werden, wurden die ISM-Bänder (Industrial, Scientific and Medical Band) eingeführt. Diese Bänder definieren verschiedene lizenzzfreie Frequenzbereiche, die von jedem Hersteller und jeder Person frei genutzt werden dürfen. Ein großes Problem besteht darin, dass nur eine begrenzte Auswahl dieser Frequenzen existiert. Durch Eigenschaften wie die Reichweite und die Energieeffizienz wird diese Auswahl zusätzlich eingeschränkt. Daher gibt es einige Frequenzbereiche, die von vielen Technologien als Übertragungsfrequenzen genutzt werden.

Wenn viele unterschiedliche Geräte auf denselben oder nahen Frequenzen senden, können Kollisionen auftreten. So stören sich einige der Technologien gegenseitig in der Ausführung ihres Anwendungszwecks.

Die Nutzung von elektromagnetischen Wellen erzeugt allerdings ein weiteres Problem. Es muss damit gerechnet werden, dass die Daten von unterschiedlichen Orten innerhalb der Reichweite abgegriffen werden können. Demnach sind Security-Aspekte wie Verschlüsselung und Authentifizierung dort von besonderer Bedeutung.

Nachfolgend werden verschiedene kabellose Übertragungsmedien behandelt, die im IoT vielseitig eingesetzt werden.

RFID

Eine Technologie, die in den meisten Fällen stark in ihrer Reichweite limitiert ist, ist die Radio-Frequency Identification (RFID). Dabei werden Daten mithilfe von elektromagnetischen Wellen übertragen. Je nach der Art des Transponders kann die Reichweite dennoch zwischen wenigen Zentimetern und einigen Kilometern liegen [4] [25].

RFID erfüllt im IoT einen speziellen Zweck. Wie dem Namen bereits zu entnehmen ist,

ist diese Technologie dazu entwickelt worden, auf den Geräten gespeicherte Identifikationsinformationen wiederzugeben und auszusenden. Das erleichtert in vielen Fällen die kontaktlose Identifikation für bestimmte Zugriffskontrollen oder von Waren in der Logistik.

Ihr großer Vorteil ist der geringe Stromverbrauch. Passive Transponder benötigen oftmals keine zusätzliche Stromquelle. Sie nutzen die induktive Energie des Empfängers für das Senden ihrer Daten. Dadurch ermöglichen sie eine schnelle Identifikation von Dingen, die über keine eigene Energiequelle verfügen. Der Nachteil der passiven Transponder ist die geringe Reichweite von wenigen Zentimetern. Für eine ausführliche und stabile Datenübertragung, die über eine einfache Identifikation hinausgeht ist RFID allerdings nicht geeignet, da es keine aktive Kommunikation beider Gesprächsparteien vorsieht.

Bluetooth (LE)

Im Gegensatz zur RFID ist Bluetooth ein Standard, der explizit für Datenaustausch entwickelt wurde und weitreichend im IoT eingesetzt wird.

Er hat dabei den Vorteil, dass er für mobile Systeme geschaffen wurde und deshalb Eigenschaften wie den Stromverbrauch im besonderen Maße berücksichtigt. Deshalb wird dieser Standard heute vor allem für die Datenübertragung mit batteriebetriebenen mobilen Geräten eingesetzt. Bluetooth Low Energy (BLE) ist eine Erweiterung, die aus dem heraus Bestreben entstanden ist, den Stromverbrauch von Bluetooth weiter zu minimieren. Bei beiden Varianten werden Daten über einen elektromagnetischen Frequenzbereich im freien 2,4 GHz ISM-Band gesendet. Ebenso basieren beide auf einem Mesh-Netzwerk mit einer Master-Slave-Struktur. Das bedeutet, dass es einen aktiven Kommunikationspartner (Master) gibt, der Anfragen an die passiven Kommunikationspartner (Slaves) sendet, wenn eine Kommunikation initiiert werden soll. Der Vorteil dieses Prinzips liegt in der großen Kontrolle des Masters über den Datenverkehr mit verbundenen Geräten. Er muss sich keine Gedanken darüber machen, dass ungefragt Daten bei ihm ankommen, deren Kontext er bereits verworfen hat. Der große Nachteil ist allerdings, dass über diese Struktur weder die Kommunikation unter den Slaves noch ein aktives Senden an den Master ohne eine vorherige Anfrage möglich ist.

Doch auch trotz ihrer Ähnlichkeit sind die beiden Bluetooth-Varianten nicht miteinander kompatibel und können nicht für die Kommunikation mit der jeweils anderen Technik genutzt werden [12].

Das liegt daran, dass BLE ein anderes Frequenzsprungverfahren für die Übertragung der einzelnen Bits nutzt [36].

Zusammen mit Bluetooth hat auch BLE seit seiner Entwicklung einige Updates bekommen. Neben der Bereitstellung neuer Features verbessert die neuste Major Version (Bluetooth 5) auch Übertragungseigenschaften wie die Reichweite, die Datenrate und die Verlässlichkeit der Übertragung [35].

Die theoretisch erreichbare Übertragungsrate von Bluetooth 5 liegt bei 2 Mbps. Die erreichbare Reichweite liegt im Außenbereich bei ca. 200 m und in Innenräumen bei ca. 40 m [35].

BLE besitzt dabei unter anderem eine veränderte Mesh-Struktur [11].

Eine Eigenschaft, die bei der Nutzung von BLE beachtet werden muss, ist die Fehlerbehandlung. Wurde bei zwei aufeinanderfolgenden Paketen ein Fehler durch eine Cyclic Redundancy Check (CRC) erkannt, dann werden für diese Übertragung (Connection-Event) vom Master keine Pakete mehr angenommen. Erst wenn der Master eine neue Übertragung initiiert, wird ein neues Connection-Event erzeugt und die Kommunikation kann fortgeführt werden [12].

Wi-Fi (IEEE 802.11)

Wi-Fi ist der Markenbegriff für ein Wireless-Local-Area-Network (WLAN), welches dafür entwickelt wurde, das Internet auch auf mobilen Geräten nutzbar zu machen. Da Wi-Fi auf dem Data Link Layer des ISO/OSI-Referenzmodells dieselbe Adressierung verwendet wie das Ethernet, kann eine Verbindung zum bestehenden Netzwerk mithilfe von bestimmten Access-Points problemlos hergestellt werden.

Die Access-Points sind statische oder mobile Geräte, die jeweils wieder über bestimmte Medien (Ethernet, Wi-Fi, etc.) mit dem Internet verbunden sind. Solche Stationen werden auch Basic Service Sets (BSS) genannt. Sie können drahtlose Kommunikationsnetze über unterschiedliche ISM-Bänder bilden. Dabei zeichnet sich die Datenübertragung durch eine hohe Bandbreite und eine hohe Datenrate aus [17].

Im Laufe der Zeit entwickelten sich viele Varianten des IEEE 802.11 Standards. Durch viele Optimierungen können mit Wi-Fi bereits Übertragungsraten von bis zu 9,6 Gbit/s erreicht werden [15].

Mit der aktuellsten Version (Wi-Fi 6E) wurden die unterstützten ISM-Bänder von 2,4 GHz und 5 GHz um das 6 GHz-Band erweitert [24].

Im Vergleich zu Bluetooth und Co ist allerdings der Stromverbrauch zu beachten. Während Wi-Fi in einer bestimmten Version im Durchschnitt um die 100-350mA verbrauchte, benötigte Bluetooth lediglich 1-35mA [10].

Für eingebettete Systeme und Systeme, die eine längere Zeit von einer statischen Stromquelle getrennt sind, kann die Nutzung von Wi-Fi nachteilig sein.

Weitere beliebte Standards

Es gibt noch einige weitere Kommunikationsstandards, die im IoT gerne eingesetzt werden. Unter diesen sind beispielsweise IEEE 802.15.4 (Zigbee) oder LoRa.

Auch IEEE 802.15.4 nutzt unter anderem das 2,4 GHz ISM-Band und versucht den Stromverbrauch bei der Übertragung auf kurzen Distanzen zu minimieren.

LoRa konzentriert sich auf die Kommunikation zwischen Endgeräten und Gateways,

die über weite Distanzen miteinander kommunizieren müssen. Aufgrund ihrer niedrigen Grundfrequenzen und einiger Signal-Modulationen kann LoRa dazu genutzt werden, Nachrichten über mehrere Kilometer zu versenden und zu empfangen [27].

2.5.2 Kabelgebundene Kommunikation

Die kabelgebundene Kommunikation hat den großen Nachteil, dass sie nicht standortunabhängig stattfinden kann, da eine physische Verbindung zwischen zwei Geräten vorliegen muss. Diese Restriktion kann jedoch von Vorteil sein. Dieser liegt besonders in der Sicherheit des eigenen Netzes. Bei einem kabelgebundenen Netzwerk muss explizit ein physischer Zugriff erfolgen. Ein Angreifer muss sich also mithilfe eines Kabels physischen Zugriff zum Netzwerk verschaffen.

Ein weiterer Vorteil der Nutzung von kabelgebundenen Medien liegt in den Möglichkeiten, die einzelnen Signale voneinander zu isolieren und vor Störquellen zu schützen.

Dennoch überwiegen im Kontext des IoT oftmals die Nachteile gegenüber der kabellosen Kommunikation. Die Vorteile einer barrierefreien und standortunabhängige Kommunikation kann ein ausschlaggebendes Kriterium für den Endkunden sein.

Für die Kabelgebundene Kommunikation sind serielle Schnittstellen weit verbreitet. Einige davon werden von vielen Mikrocontrollern nativ unterstützt.

Im Laufe der Jahre wurden viele, teilweise sehr unterschiedliche serielle Standards entwickelt. Einige für den Kontext dieser Arbeit wichtige Standards werden in den nachfolgenden Abschnitten genauer besprochen. Ein Vergleich zwischen der Struktur von drei dieser Standards kann in Abbildung 2.3 eingesehen werden.

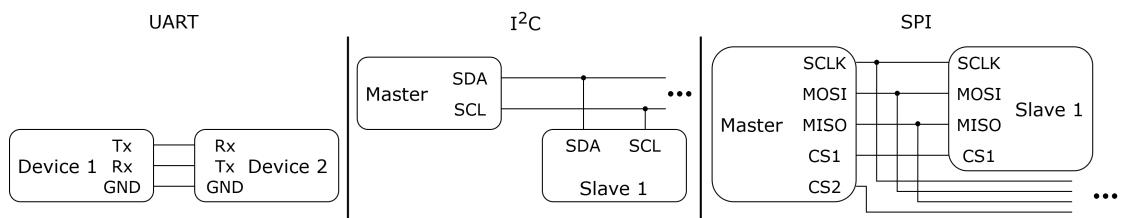


Abbildung 2.3: Vergleich der Topologien von UART, I²C und SPI

KNX

KNX ist ein serielles Kommunikationsprotokoll, welches explizit für den Einsatz in der Heimautomation entwickelt wurde [16].

Dabei können Geräte über unterschiedliche physische Übertragungsmedien miteinander verknüpft werden. Unterstützt wird unter anderem die Verbindung über die Stromleitung, Ethernet und Twisted-Pair-Kabeln [18].

Da KNX ein Protokoll beschreibt und nicht an ein Übertragungsmedium gebunden ist, kann es ebenso für die kabellose Übertragung genutzt werden [16]. Es wird in dieser Arbeit bei der kabelgebundenen Kommunikation aufgeführt, da es in vielen Fällen kabelgebunden eingesetzt wird.

Der große Vorteil dieses Protokolls liegt in der möglichen Nutzung von Stromleitungen, die über sehr große Distanzen verlegt sein können. So begünstigt es die verlässliche Kommunikation vieler, im ganzen Haus verteilter Geräte über eine bereits existierende Infrastruktur.

Über die Stromleitung sind Übertragungsgeschwindigkeiten von 1-2 Mbps erreichbar [8]. KNX unterstützt viele Topologien, die miteinander verknüpft werden können. Es ist nicht an eine Netzwerktopologie gebunden [18].

UART

Das Protokoll „Universal Asynchronous Receiver Transmitter (UART)“ basiert auf einem asynchronen Datenaustausch zwischen zwei Geräten.

Aufbauend auf dem Protokoll entwickelten sich UART-Schnittstellen. Sie erlauben eine Full-Duplex-Kommunikation zwischen den Schnittstellen.

Durch die Asynchronität und die Full-Duplex-Kapazität des Protokolls, können Benachrichtigungen und Änderungen von Zuständen ohne viel Aufwand in beide Richtungen übertragen werden. Aus diesem Grund wird es weitgehend in der Datenkommunikation und in Steuerungssystemen eingesetzt [9].

Ein korrekter Datenaustausch mit UART benötigt zwei Signaling-Lines [9]. Für jedes Gerät ergibt sich eine Line (Tx) zur Übertragung der eigenen Daten und eine Line (Rx) zum Empfang der vom anderen Gerät gesendeten Daten. Die Tx-Line des einen Gerätes wird mit der Rx-Line des anderen verbunden. So können beide Kommunikationspartner kollisionsfrei und unabhängig voneinander kommunizieren.

Damit beide Geräte die über die Leitungen gesendeten Signale verstehen und als Daten interpretieren können, müssen sich beide Parteien auf dieselbe Baudrate (Bitrate) geeinigt haben. Es muss sichergestellt werden, dass die Clocks beider Geräte akkurat genug sind und während der Übertragung keinen signifikanten Drift aufweisen [9].

Eine Übertragung (das Senden eines Wortes) wird mit einem Start-Bit angekündigt. Es repräsentiert dabei sowohl den Beginn der Datenübertragung als auch den Zeitpunkt,

an dem die Clocks der beiden Gesprächsparteien synchronisiert werden müssen. So können sich beide Parteien zu Beginn der Übertragung sicher sein, dass ihre Daten korrekt gesendet und empfangen werden können. Das Ende einer Nachricht wird durch ein oder mehrere Stop-Bits gekennzeichnet.

Die Konfiguration des Protokolls und der Aufbau der Nachrichten muss vor Beginn auf beiden Geräten abgestimmt werden. Andernfalls wird die Kommunikation fehlschlagen. Das Protokoll bietet die Möglichkeit, den Aufbau der gesendeten Pakete genau zu konfigurieren.

Konfigurierbar sind die Anzahl der Datenbits, die Anzahl an Stop-Bits sowie das hinzufügen zusätzlicher Parity-Bits, zur Erkennung von Fehlern in der Übertragung [9].

Die erreichbare Datenrate der Übertragung ist abhängig von der langsamsten Main-Clock der beiden Gesprächsparteien. Beide Parteien müssen in der Lage sein in derselben Datenrate empfangen und senden zu können.

I²C

Inter-Integrated Circuit (I²C) ist ein Protokoll, welches im Jahr 1982 von Philips entwickelt wurde [34]. Es basiert auf einem seriellen Datenbus mit einer Master-Slave-Topologie.

Der Standard ermöglicht die Kommunikation mehrerer Master mit mehreren Slaves. Die Datenübertragung findet in Form einer Half-Duplex-Kommunikation statt [34]. Das bedeutet, es existieren keine separaten Übertragungsleitungen zum Senden und Empfangen.

Master und Slaves sind über zwei Signaling-Lines miteinander verbunden [34]. Die Serial-Data-Line (SDA) dient der Übertragung der Daten zwischen Mاستern und Slaves. Die Serial-Clock-Line (SCL) gibt die Clock an.

Die Clock bestimmt die Geschwindigkeit und die Timings innerhalb der Datenübertragung. Da I²C ein synchrones Kommunikationsprotokoll ist, wird die Kommunikation vom jeweiligen Master synchronisiert. Dabei wird die Clock in der klassischen Umsetzung des Protokolls von einem Master vorgegeben. Diese Clock wird bei der Übertragung von dem Master auf der entsprechenden Line angelegt. Ist keine Übertragung im Gange, bleibt der Wert auf der SCL konstant auf HIGH [34].

Eine Übertragung kann nur von einem Master initiiert werden. Ein Slave ist nicht dazu in der Lage, eine Übertragung an den Master oder an andere Slaves einzuleiten.

Da die Nachrichten dabei über einen geteilten Bus gesendet werden, beschreibt das Protokoll ein eigenes Addressing-Scheme. Jeder Slave verfügt dabei über eine eindeutige Adresse auf dem Bus. Wenn ein Master Daten senden oder lesen will, muss er zunächst die Adresse des Slaves angeben, mit dem er interagieren will. So können alle anderen Slaves die Daten ignorieren, auch wenn sie über den gemeinsamen Bus empfangen werden. Der Slave mit der passenden Adresse antwortet auf die Anfrage mit einem ACK oder NACK, um die Bereitschaft für die Kommunikation zu bestätigen oder abzulehnen.

In derselben Nachricht wird zudem auch die Intention des Masters angegeben (Write/Read). Sendet der Master ein Write an den Slave, so wird der Inhalt der Nachricht an den Slave übertragen. Sendet er ein Read, wird das nächste vorbereitete Byte aufseiten des Slaves gelesen und kann von dem Master weiterverarbeitet werden.

Auf jedes empfangene Byte Daten antwortet der Slave mit einem ACK oder einem NACK. Je nach Modus kann das letzte Byte mit einem ACK oder NACK bestätigt werden. Wurden alle Bytes gelesen oder geschrieben, folgt ein End-Bit, welches das Ende der Übertragung angibt [34].

Anstatt Daten aktiv zu senden, stellt ein Slave Daten bereit, die der Master bei einer Anfrage lesen kann. Ein Slave ist nicht in der Lage von sich aus Daten an den Master zu senden. Je nach Anwendungszweck kann das ein großer Nachteil gegenüber Protokollen wie UART sein.

SPI

Das Protokoll „Serial Peripheral Interface (SPI)“ wurde von Motorola entwickelt [34]. Ähnlich wie I²C basiert es auf einem seriellen und synchronen Datenbus mit einer Master-Slave-Topologie.

SPI ermöglicht die Kommunikation zwischen einem Master und mehreren Slaves. Ein großer Unterschied zu I²C ist, dass die Kommunikation zwischen Master und Slaves in Full-Duplex durchgeführt wird. Das bedeutet, es existieren zwei separate Lines für Input und Output zwischen den Gesprächspartnern. Das ermöglicht eine nebenläufige Datenübertragung in beide Richtungen (Master ⇌ Slave).

Um SPI nutzen zu können, werden vier Signaling-Lines zwischen Master und Slaves benötigt [34]. Eine dieser Lines ist die Serial-Clock (SCLK), die ebenfalls vom Master generiert wird.

Damit die Full-Duplex-Kommunikation gewährleistet werden kann, werden zwei separate Lines für den Datenaustausch eingesetzt [34]. Dabei dient die Master-Output-Slave-Input-Line (MOSI) zur Übertragung der Daten vom Master zu einem der Slaves (Master → Slave). Die Master-Input-Slave-Output-Line (MISO) ist für die Übertragung in die Gegenrichtung zuständig (Master ← Slave). Die letzte Line ist die Slave-Select-Line, die auch Chip-Select (CS) genannt wird. Sie dient dem Master zur Adressierung sowie Aktivierung eines Slaves für die Initiierung einer Datenübertragung [34].

Jeder Slave muss mit dem Master über eine CS-Line verbunden sein. Die Adresse muss in diesem Fall nicht über den Bus gesendet werden. Dadurch entfällt ein großer Overhead für die Adressierung, wie es beispielsweise bei I²C der Fall ist. Für die Datenübertragung werden auf beiden Seiten Shift-Register eingesetzt. In jedem Clock-Cycle wird das nächste Byte des Masters in das Register des Slaves übertragen. Fordert der Master einen Read beim Slave an, dann werden die Daten aus dem Shift Register des Slave in das Register des Masters übermittelt.

Kapitel 3

Verwandte Arbeiten

Die Heterogenität und die Komplexität der verschiedenen Smart-Home-Frameworks machen es für Hersteller schwierig, in diesen Markt einzusteigen. Ein grundlegendes Problem, das viele Hersteller abschreckt, ist die Wartbarkeit.

Es stellt sich die Frage, wie diese Heterogenität für Hersteller und Nutzer reduziert werden kann.

In diesem Kapitel werden deshalb Arbeiten vorgestellt, die sich an einer Lösung für dieses Problem versucht haben oder mit ihrer Arbeit gute Ansätze geschaffen in diese Richtung geschaffen haben.

Dabei werden im ersten Abschnitt zunächst Arbeiten betrachtet, die weitere Abstraktionsebenen zwischen der Software und den Smart Devices einfügen.

Im zweiten und dritten Abschnitt werden daraufhin zwei Projekte beschrieben, die stellen sich auf verschiedene Arten dem Problem der *Interoperability* stellen und versuchen, diese im Hinblick auf Smart Devices bereitzustellen.

Die hier gewonnenen Erkenntnisse werden anschließend in den nachfolgenden Kapiteln für die Konzeption eines eigenen Prototyps aufgegriffen und eingearbeitet.

3.1 Smart Labs

In einer Arbeit von Salzman, Govaerts, Halimi und Gillet [28] wurde sich an Thompsons [33] Konzept eines Smart Device orientiert, um eine Spezifikation für Smart Devices innerhalb von Remote Labs zu entwickeln. Sie führten eine explizite Trennung der vom Internet ansteuerbaren Schnittstellen für die Geräte und die internen Funktionalitäten der Geräte selbst ein.

Auf Basis von Thompsons Überlegungen definierten Salzman et al. [28] eine generische Spezifikation, die von allen Smart Devices geteilt werden sollte. Das hat den Vorteil, dass die Smart Devices und ihre Funktionen für die Einbindung in Client-Anwendungen unerheblich sind. Smart Devices können im Kontext des Smart Labs standardisiert betrachtet werden. Unabhängig davon, um welches Gerät es sich genau handelt, können so für alle Devices externe Dienste und Funktionen wie *Logging* oder *Authentication* bereitgestellt werden [28]. Auf der Client-Seite wird es dadurch einfacher die Hauptfunktionen

des Remote Labs zu gewährleisten [28].

Um mit den entfernten Geräten auf eine ähnliche Weise arbeiten zu können wie mit ihren lokalen bzw. physischen Pendants, muss der Gesamt- oder Teilzustand des Systems Client-seitig abgebildet werden [28]. Dadurch wird eine barrierefreie Nutzung der Geräte in Echtzeit ermöglicht.

Jedes Smart Device stellt einen eigenen Server zur Verfügung. Der Zustand der Geräte in den Remote Labs kann sich durch unterschiedliche Ursachen (physische Schnittstelle oder Logik) ändern. Anstelle anderer HTTP-Dienste haben sie sich deshalb für die Verwendung von Websockets entschieden [28]. Dadurch sind sowohl Push- als auch Pull-Mechanismen für den Server nutzbar. Der Setup- und Integrationsaufwand dieser Server ist deutlich reduziert [28].

Angesprochen werden die Server über das HTTP-Protokoll. Die Daten werden in das JSON-Format serialisiert [28]. Die Kommunikation mit dem Smart Device erfolgt direkt und innerhalb des Labs, ohne dass eine separate Vermittlerinstanz benötigt wird.

Das Konzept des Gesamtsystems basiert auf Services. Ein Service repräsentiert einen Sensor oder Aktuator, der über die API mit dem Internet bzw. den Clients verbunden ist [28].

Services werden durch ihre Metadaten ausführlich beschrieben und können die zuvor genannten Zusatzfunktionen bereitstellen [28].

Die Metadaten können durch spezielle Metadaten-Services angefragt werden. Sendet ein Client beispielsweise die Anfrage `getSensorMetadata`, so erhält er von dem Smart Device ein JSON-Objekt mit einem Array aller Sensoren und ihrer Metadaten [28]. Unter diesen Daten sind unter anderem die IDs der Sensoren, ihre Namen, eine genaue Beschreibung, die verwendeten Datentypen und ihre gemessenen Werte [28].

Das Smart Device kann durch diese Abstraktion einheitlich angesprochen werden.

In diesem Ansatz wird somit versucht die Heterogenität der Smart Devices und ihrer Funktionen durch eine allgemeine Spezifikation und genaue Beschreibungen zu reduzieren. Allerdings wird hier davon ausgegangen, dass ein Device eine bestehende Internetverbindung besitzt, in der Lage ist einen stabilen Server zu implementieren und eine Vielzahl an Serialisierungen von JSON-Paketen in Echtzeit vornehmen kann. Das ist bei Geräten außerhalb eines solchen Labs nicht zwangsläufig der Fall. Die Arbeit betrachtet zudem lediglich die Spezifikation dieser Umsetzung. Die Entwicklung eines solchen Systems wurde dabei nicht vorgenommen.

Einen Schritt weiter sind Simeoni et al. [31] gegangen. Ihr Versuch, eine Abstraktionsebene zwischen den integrierenden Systemen und den Smart Devices zu schaffen, entstand ebenfalls aus der Notwendigkeit der Vernetzung vieler verschiedener Geräte und basiert auf einer ähnlichen Idee wie der vorherige Ansatz.

Sie nennen ihren Versuchsaufbau das „Living Lab“ [31]. Sie beschreiben das Problem der Heterogenität als eine Fragmentierung. Mit dieser Fragmentierung geht eine fehlende *Interoperability* vieler unterschiedlicher Technologien einher.

Um die *Interoperability* und eine gewisse Standardisierung ihres Systems zu adressieren, orientierten sie sich an dem vom World Wide Web Consortium (W3C) entwickelten

Standard ‚Web of Things (WoT)‘[31].

Der WoT-Standard beschreibt den Einsatz von bekannten und erfolgreichen Web-Paradigmen zur Reduktion der IoT-Fragmentierung und zur Erhöhung der *Flexibility* und *Interoperability* für IoT-Anwendungen [32]. Er bedient sich dabei größtenteils an dem Konzept der Digital Twins, also einer digitalen Abbildung der physischen oder abstrakten Dinge im Anwendungskontext.

Ähnlich wie in dem Ansatz von Salzman et al. [28] nutzen Simeoni et al. [31] für die Kommunikation HTTP und eine Form des JSON-Formaten. Auch ihr Ansatz beruht darauf, mithilfe von Services eine Abstraktionsebene zwischen der exakten Implementierung und den beschreibenden Funktionen eines Gerätes zu generieren.

Für die Beschreibung der Metadaten nutzen sie die Spezifikationen des Resource-Description-Framework, welches ebenfalls auf das W3C zurückzuführen ist.

Dieses Framework beschreibt ein Datenmodell für Metadaten und wird in der Arbeit von Simeoni et al. [31] zur Strukturdefinition der Smart Devices und zur Interaktion mit diesen innerhalb ihres Systems genutzt. Der Kern des WoT-Standards ist die Thing-Description. Bei Simeoni et al. [31] stellt sie eine Liste aller Geräte dar.

Es existieren einige Interaktionsweisen, die der Standard basierend auf der Thing-Description definiert. Sie liefern die Abstraktionsebene für die unterschiedlichen Devices. Es gibt: *Properties*, *Actions* und *Events* [32].

Properties sind lesbare und schreibbare Datenpunkte wie beispielsweise Sensorwerte (Read-Only), Konfigurationsparameter (Read-Write) oder Verarbeitungsergebnisse [32].

Actions sind aufrufbare Prozesse. Sie werden von Geräten bereitgestellt oder mit den Geräten durchgeführt [32]. Eine *Action* kann zum Beispiel das Kochen von Kaffee bei einer Kaffeemaschine oder das Ändern von Parametern über einen längeren Zeitraum darstellen.

Events sind asynchrone Interaktionen, die dazu in der Lage sind, beobachtete Veränderungen im System an Clients zu senden, ohne vorher eine Anfrage erhalten zu haben (Push-Semantik) [32]. Eine solche Änderung kann beispielsweise das Erreichen eines bestimmten Schwellenwertes oder die Veränderung von Sensorwerten durch ein Ereignis aus der Umgebung sein.

Mithilfe dieser Interaktionsmöglichkeiten soll semantische *Interoperability* von allen Devices und ihren Funktionsweisen sichergestellt werden.

Neben der Orientierung an diesem Framework ist ein großer Unterschied zu dem vorher behandelten Ansatz, dass dabei nicht vorausgesetzt wird, dass jedes Device über eine kabellose Infrastruktur verfügen muss. Ihr Ansatz basiert auf einem Bridge- oder Gateway-System („Living Lab Gateway“ [31]), welches die unterschiedlichen Geräte zentral einbindet und sich um ihre Verwaltung kümmert.

Das Gesamtsystem besteht aus den Geräten, der Bridge und einer Cloud-Infrastruktur mit angebundener Datenbank. Die Cloud-Infrastruktur basiert auf einer mit Docker-Containern bereitgestellten „Microservice Architecture“ [31].

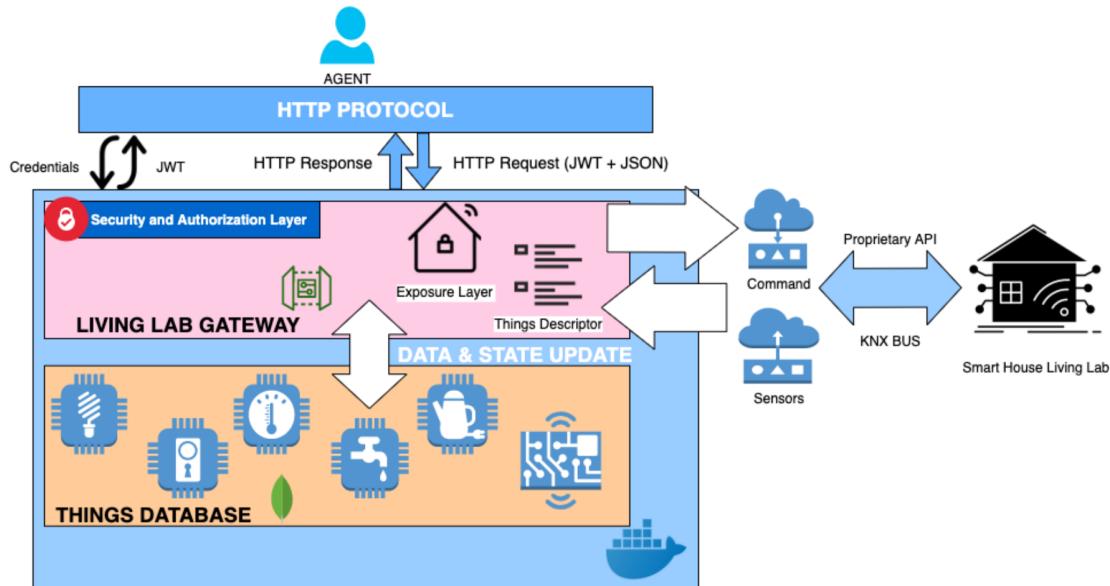


Abbildung 3.1: Architektur des Living Lab Gateway [31]

Die Bridge kann sowohl kabellose als auch kabelgebundene Geräte verwalten [31]. Kabelgebundene Geräte benötigen dafür eine KNX-Schnittstelle. Sie müssen bereits eine gewisse *Smartness* mit sich bringen.

Um den Verwaltungsaufwand und die Last auf der Bridge zu reduzieren, findet die Verarbeitung und die Speicherung der Daten in der separaten Cloud-Infrastruktur statt.

Die Bridge übernimmt weitestgehend die Aufgabe der Vorverarbeitung und Übermittlung der Daten an ein Cluster. Sie kümmert sich um alle nötigen Schritte, um die Kommunikation über die KNX-Schnittstellen zu gewährleisten [31]. Sie dient als Vermittler zwischen den Devices und der Logik des Systems, die in der Cloud integriert ist.

Der Aufbau des Systems ist in Abbildung 3.1 zu sehen. Die Cloud wird mithilfe einer Representational-State-Transfer-API (REST) angesteuert [31]. Ihre Hauptaufgaben sind das Abspeichern des Systemzustandes und der Gerätedaten in der Datenbank sowie die Bereitstellung von Funktionen wie *Security* und *Authentication* [31].

Für eine erfolgreiche Änderung eines Device wird zuerst der Thing-Descriptor von dem Gateway angefragt [31]. Die Antwort enthält eine Authentifizierungsanfrage. Der Client muss seine Nutzerdaten an das Gateway senden. Bei erfolgreicher Authentifizierung erhält der Client einen JWT-Token. Um unnötige Authentifizierungen zu verhindern, kann dieser mit jeder weiteren Nachricht versendet werden [31]. Der Client kann sich daraufhin eine Liste aller Devices geben lassen und Änderungsanfragen für einzelne Devices senden.

Zur Evaluierung ihrer Arbeit führten Simeoni et al. [31] anschließend Messungen durch. Sie wollten herauszufinden, inwieweit ihre Implementierung eine zusätzliche Verzögerung

im Vergleich zu einer direkten Kommunikation erzeugt. In ihrer Performance-Evaluation stellten sie leichte Varianzen der Verzögerungen fest, die bei einer direkten Verbindung nicht zu beobachten waren. Allerdings konnten sie diese bestimmten Testbedingungen und Ausreißern zuordnen. Sie ermittelten, dass ihr System im Durchschnitt keine signifikante zusätzliche Verzögerung erzeugt [31].

Das ist in gewissem Maße der hohen Geschwindigkeit der Datenbank und den Clustern sowie einer schnellen Infrastruktur des Netzwerks zu verdanken.

3.2 Homebridge

Eine populäre Entwicklung, welche sich mit der *Interoperability* von Smart Devices beschäftigt, ist das Open-Source-Projekt „Homebridge“. Diese Interoperabilität bezieht sich auf die Integration bestehender Smart Devices in den Kosmos des Apple HomeKit, auch wenn dieses nicht nativ unterstützt wird.

Homebridge ist ein auf NodeJS basierender Server, der im eigenen Heimnetzwerk laufen kann, um die Apple Homekit API zu emulieren [13]. Die Grundidee dieses Ansatzes ist die Nutzung von Plugins. Sie können von der Community erstellt werden, um viele unterschiedliche Third-Party Smart Devices und APIs zu unterstützen und in das Homekit-Framework zu integrieren [13].

Zum Zeitpunkt dieser Arbeit existieren ca. 3300 Pakete, die für die Hombebridge entwickelt wurden und sofort eingebunden werden können [23]. Darunter sind sowohl Pakete, die eine Unterstützung für die Kommunikation mit herstellerunabhängigen Smart Devices über separate Formate wie HTTP oder MQTT ermöglichen als auch Pakete, die explizit bestimmte herstellerspezifische Produkte und APIs einbinden [23].

Die Bridge muss auf einem Gerät eingerichtet werden, welches dauerhaft mit Strom- und Internetzugriff ausgestattet ist. Zum Einrichten wird ein gewisses technisches Grundwissen und eine Bereitschaft vorausgesetzt, den notwendigen Konfigurationsaufwand zu übernehmen. Wenn viele unterschiedliche zusätzliche Plugins eingebunden werden sollen, ist ein Plug-and-Play nicht ohne weiteres möglich.

3.3 Matter

Aus dem Bestreben heraus, die *Interoperability* zwischen Smart Devices und Smart-Home-Systemen (Plattformen) unterschiedlicher Hersteller bereitzustellen, haben sich einige Unternehmen der Branche für das Projekt „Matter“ zusammengeschlossen. Unter anderem durch die großen Vertreter Amazon, Apple, Comcast und Google formte sich im Jahr 2019 aus der ehemaligen „Zigbee Alliance“ die „Connectivity Standards Alliance“ (CSA) [1].

Zusammen wollen sie einen Standard erschaffen, der möglichst viele Frameworks und Heimautomationssysteme zusammenführt. Da die meisten großen Entwickler der Branche daran arbeiten, ist zu erwarten, dass der Standard weitreichend eingesetzt werden kann. Er soll die wesentlichen Charakteristika *Simplicity*, *Interoperability*, *Reliability* und *Security* implementieren [3].

Für die Kommunikation wird auf einen IP-basierten Ansatz gesetzt (Wi-Fi und Thread Network Layers). Der Prozess der Inbetriebnahme soll über Bluetooth LE durchgeführt werden [3].

Matter fokussiert sich dabei auf die Entwicklung eines Standards, der von allen unterstützt werden soll. Ein Hersteller müsste also lediglich einen universalen Standard einbinden und unterstützen. Das würde die Entwicklung neuer Devices vereinfachen und könnte auch viele Probleme der Endkunden lösen, die auf fragmentierte und inkompatible Geräte zurückzuführen sind.

Zum Zeitpunkt dieser Arbeit ist der Standard in der Endphase seiner Entwicklung. Laut CSA liegt der Fokus zunächst auf der Fertigstellung der Matter SDK. Erste Matter-basierte Geräte sollen im Jahr 2022 auf den Markt kommen [2].

Unternehmen, die Matter-Unterstützte Systeme und Devices entwickeln möchten, können sich bereits um eine Zertifizierung kümmern.

Kapitel 4

Voranalyse

Dieses Kapitel beschäftigt sich mit der Zusammenfassung und Beurteilung der in den vorherigen Kapiteln eingeleiteten Konzepte und Ideen. Aus den vorliegenden Arbeiten sollen für diese Arbeit notwendige Punkte herausgestellt werden. Es wird versucht, den Grundstein für die zu entwickelnde Anwendung zu legen.

Der erste Abschnitt beschäftigt sich daher zunächst mit der Eingliederung dieser Arbeit in den Kontext der zuvor beschriebenen Forschungen und Ansätze. Die vorherigen Ansätze werden von dem Vorhaben in dieser Arbeit abgegrenzt. Der zweite Abschnitt betrachtet bestimmte Fragestellungen, die sich in Bezug auf die zu entwickelnde Anwendung ergeben können. Sie sollen als Leitfaden für die Konstruktion einer Anwendung dienen. Das Ziel der Arbeit ist es, sich mit diesen Fragestellungen zu beschäftigen und sie im Verlauf dieser Arbeit zu beantworten.

In den letzten beiden Abschnitten werden Anforderungen erarbeitet, die für eine Umsetzung der beschriebenen Anwendung und die damit zusammenhängende Trennung gelten sollen. Die Anforderungen werden im nächsten Kapitel in die Design-Entscheidungen bezüglich der Umsetzung miteinbezogen und bei der Entwicklung berücksichtigt.

4.1 Eingliederung und Abgrenzung

Wie den zuvor behandelten Arbeiten zu entnehmen ist, ist die *Interoperability* von Smart Devices ein großes Problem, wenn es um ihre Integration in bestehende Systeme geht. Dabei wurde in einigen Fällen versucht, die *Interoperability* jeweils durch die Entwicklung eines eigenen Gesamtsystems zu gewährleisten.

Das ist beispielsweise im Falle der Remote Labs zu sehen. Es wurden neue Systeme erschaffen, die unterschiedliche, bereits existente Smart Devices einbinden und ihre APIs bedienen. Diese Arbeit versucht eine andere Herangehensweise. Da die meisten Endnutzer bereits bestehende Systeme und Smart-Home-Frameworks besitzen, ist der Ansatz, ein komplett neues System zu entwickeln, nicht zu empfehlen.

Es wird versucht, die Einbindung neuer Geräte an einer anderen Stelle vorzunehmen. Dabei wird ein ähnlicher Ansatz wie beim Homebridge-Projekt [13] gewählt. Das Projekt ist explizit für die Integration bestehender Smart Devices in das Apple HomeKit

System entwickelt worden. Da die Homebridge-Umsetzung sich an die Anwender richtet, fehlt allerdings eine Standardisierung für Geräte, die selbst keine Smart-Funktionalitäten bereitstellen können. Zudem ist das Projekt durch die Nutzung des Apple Frameworks limitiert.

Auch Matter [3] macht einen guten Schritt in Richtung *Interoperability*. Doch auch sie beschreiben einen neuen IP-Basierten Standard, in den sich Hersteller zunächst einarbeiten und sich mit möglicherweise für sie neuen Technologien auseinandersetzen müssen. In dieser Arbeit wird versucht, ein System zu entwickeln, dass unabhängig von den genutzten Frameworks erweiterbar ist und die Kommunikation mit den Geräten über simplere Technologien exploriert.

Besonders interessant ist dabei die Nutzung von Schnittstellen, die zuvor im IoT nur wenig Anwendungsfälle gefunden haben. Serielle Schnittstellen haben den Vorteil, dass sie in vielen Mikrocontrollern bereits integriert sind und für den Hersteller keinen großen Konfigurationsaufwand erzeugen. Da sich die Schnittstelle nicht verändert, muss sich der Hersteller zudem keine Gedanken über unterschiedliche Protokollversionen machen.

Die Verwendung dieser Technologien bietet einen wichtigen Anknüpfungspunkt, um mögliche Schwachstellen, Limitierungen und gegebenenfalls sinnvolle Anwendungsfälle zu erarbeiten.

4.2 Fragestellungen

Smart Devices kommunizieren entweder direkt mithilfe entsprechender Libraries oder über spezielle Gateways mit den Cloud-Diensten der Anbieter. Dafür wird von den Devices in den meisten Fällen die Nutzung kabelloser Übertragungsmedien vorausgesetzt. Hinzu kommt ein großer Verwaltungs- und Integrationsaufwand durch verschiedene Smart Frameworks, die der Hersteller mit diesem Gerät unterstützen will. Das Ziel dieser Arbeit ist es, herauszufinden, ob diese Abhängigkeit aufgelöst werden kann. Im Folgenden werden einige Fragestellungen vorgestellt, die sich im Hinblick auf das vorgestellte Ziel dieser Arbeit ergeben.

1. In welchen Fällen ist eine Trennung sinnvoll?

Eine Auftrennung kann einige Vorteile und auch einige Nachteile wie beispielsweise einen bestimmten Mehraufwand mit sich bringen. Smart Devices können in vielen verschiedenen Formen und mit unterschiedlichen Funktionen existieren. Daher gibt es Geräte, für die die Vorteile einer solchen Umsetzung minimal, der Mehraufwand jedoch sehr groß sein würde. Es gilt herauszufinden, in welchen Fällen der Nutzen die Kosten übersteigt.

2. Was muss bei der Entwicklung beachtet werden?

Da Smart Devices und Smart Homes sehr komplexe heterogene Systeme sind, ist ein Eingriff in eine dieser Architekturen nicht ohne weiteres möglich. Es existieren unterschiedliche Herangehensweisen an die Umsetzung der Trennung. Sie können Limitierungen mit sich bringen und zu bestimmten Problemen für die Anwendung führen. Damit keine Funktion eingeschränkt oder verändert wird, sollten einige Faktoren erarbeitet und bei der Entwicklung beachtet werden.

3. Welche Möglichkeiten ergeben sich?

Die zusätzliche Abstraktionsebene gibt dem Entwickler die Möglichkeit, weitere Verarbeitungen und zusätzliche Funktionalitäten bereitzustellen. Wie sehen diese aus? Sind ähnliche Umsetzungen, wie bei Salzman et al. [28] denkbar?

4. Welche Limitierungen entstehen und welche Auswirkungen haben sie?

Bei der Entwicklung eines bestimmten Ansatzes entstehen durch die Design-Entscheidungen gewisse Einschränkungen wie die Datenrate oder die Verzögerung. In der Arbeit von Simeoni et al. [31] wurde gezeigt, dass sie mit ihrer Cloud-basierten Lösung keine bemerkbare Verzögerung hervorrufen. Die Anbindung an die Cloud haben sie aufgrund vorteilhafter Performance und Speicherkapazität gewählt. Eine interessante Fragestellung ist, ob eine lokale Datenverarbeitung für diesen Einsatz nicht ausreichend ist und welche Verzögerung sie hervorruft. In dieser Arbeit sollen die Limitierungen dieser und möglicher weiteren Umsetzungen betrachtet und abgeschätzt werden. Die Auswahl des Übertragungsmediums ist hier ein entscheidender Faktor.

5. Welches Übertragungsmedium ist am besten geeignet?

In den vorherigen Kapiteln wird ersichtlich, dass eine Vielzahl an unterschiedlichen Medien zur Datenübertragung existiert. Eine wichtige Frage dieser Arbeit wird also sein, welches Übertragungsmedium für eine Trennung am besten geeignet ist. Da jede Technologie Vor- sowie Nachteile besitzt, wird diese Frage nicht absolut und eindeutig beantwortet werden können. Deshalb wird im Rahmen dieser Arbeit versucht, die Hauptaspekte der Technologien zu betrachten und ihre Eignung in unterschiedlichen Anwendungsfällen zu beurteilen.

6. Welche Verbindungen der Module sind möglich und nützlich?

Durch die zusätzliche Abstraktionsebene und einer Vermittlungsschicht, ist es möglich, verschiedene Kardinalitäten zu gewährleisten. Ähnlich wie beim Homekit-Ansatz, können auf Ebene des Smart möglicherweise eine Vielzahl von Geräten integriert werden. Das ist zum Beispiel sinnvoll, damit ein Hersteller nicht für jedes Device ein neues Smart herstellen oder kaufen muss. Das könnte beim Endkunden sogar eine Ersparnis mit sich bringen. Ein weiteres Konzept, welches im Rahmen dieser Abstraktion umsetzbar ist, ist die Verbindung von einem Device zu diversen Smarts. Limitierungen sind hierbei auf verbindungstechnischer Ebene zu verorten. Ob diese Kombinationen sinnvoll sind und welche Problematiken sich daraus ergeben, gilt es im Verlauf dieser Arbeit herauszufinden.

7. Welche Optimierungen dieses Ansatzes sind denkbar?

Diese Arbeit beschreibt die Entwicklung eines Prototyps. Dabei sollen die Möglichkeiten und Limitierungen erfasst werden, die bei der Umsetzung entstehen können. Dabei werden nicht immer optimale Entscheidungen getroffen. Es ist wichtig, sich damit zu beschäftigen, an welchen Stellen Optimierungsbedarf besteht und wie eine solche Optimierung aussehen kann.

4.3 Anforderungen

Damit die Problemstellungen dieser Arbeit genau erfasst werden können, ist es nötig, die Anforderungen an die geplante Umsetzung zu erarbeiten.

Da es bei dieser Umsetzung um die Erstellung eines Smart Device geht, sollten auch die allgemeinen Anforderungen an Smart Devices betrachtet werden. Die Einhaltung dieser Anforderungen muss sichergestellt sein. Ist das in der entwickelten Umsetzung nicht der Fall, das bedeutet, die Auftrennung des Smart Device erfüllt eine der Anforderungen nicht, dann ist das geplante Vorhaben gescheitert. Das aufgetrennte Smart Device muss in der Lage sein, alle Anforderungen an ein Smart Device zu erfüllen.

Sie werden im darauffolgenden Abschnitt durch weitere Anforderungen ergänzt, die explizit aus den Eigenschaften der Auftrennung entstehen. Sie sollten gegeben sein, um eine korrekte und unproblematische Nutzung des Gerätes zu gewährleisten. Wie diese Anforderungen genau aussehen, wird im folgenden Abschnitt mithilfe der bereits erwähnten Quellen erarbeitet.

4.3.1 Anforderungen an Smart Devices

Thompson [33] erwähnt in seiner Arbeit explizite Anforderungen, die er an Smart Devices hat und die für ihn ausschlaggebend für die Existenz einer *Smartness* sind. Er setzte mit seinem Konzept von Smart Devices den Grundstein für ihre Weiterentwicklung.

Da dieses Konzept innerhalb der Jahre auf viele unterschiedliche Arten in die Praxis umgesetzt wurde und sich weiterentwickelt hat, haben sich auch die Anforderungen an sie geändert. Er selbst erweiterte es mit möglichen Fähigkeiten, die Smart Devices und Smart-Home-Systeme bereitstellen können, um dem Nutzer eine bessere Erfahrung zu ermöglichen. Aus diesen Vorschlägen und Ideen haben sich im heutigen Kontext Anforderungen gebildet.

So ist beispielsweise die *Maintainability* heute mehr als ein Nice-To-Have, sondern stellt eine Grundanforderung an neue Smart Devices dar.

Auch Silverio-Fernandez et al. [30] haben in ihrer Arbeit drei wesentliche Eigenschaften von Smart Devices herausgestellt. Da diese Eigenschaften nach den Autoren das Konzept des Smart Device am genauesten definieren, können sie für den Zweck dieser Arbeit als Anforderungen an Smart Devices betrachtet werden. Die Eigenschaften der

Digital Twins nach Minerva et al. [22] gelten nicht für die allgemeine Definition von Smart Devices. Sie können nicht direkt als Anforderungen gesehen werden. Sie beziehen sich auf die technischen Aspekte der Smart Devices und sollten bei der Umsetzung eines Smart Device beachtet werden. Aus ihnen können jedoch auch andere allgemeingültige Anforderungen wie die Notwendigkeit einer für den Zweck passenden Datenrepräsentation gezogen werden.

Zusammenfassend ergeben sich folgende Anforderungen:

- Das Smart Device sollte einfach einzubinden sein.
(*Interoperability* und *Maintainability*) [29]
- Das Smart Device sollte eindeutig identifizierbar sein. (*Identity*) [33]
- Das Smart Device sollte über ein Netzwerk erreichbar sein und eine Kommunikation über geeignete Anfragen ermöglichen. (*Communications*) [33]
- Wenn der Nutzer einen Befehl gibt, sollte dieser in kurzer Zeit umgesetzt werden und eine Veränderung sollte beobachtbar sein. (*Responsiveness* und *Communications*) [33]
- Die Änderung der Umgebung und des Gerätes sollte mithilfe von geeigneten Repräsentationen dargestellt werden. (*Representativeness*) [22]
- Das Smart Device sollte in der Lage sein, seine Umgebung wahrzunehmen und auf Änderungen entsprechend reagieren zu können.
(*Context-Awareness* und *Autonomy*) [30]

4.3.2 Anforderungen an die Auftrennung

Die Auftrennung in ‚Smart‘ und ‚Device‘ führt zu einer Indirektion. Mit dieser Auftrennung gehen einige veränderte Eigenschaften wie erhöhte Verzögerungen und ein zusätzlicher Integrationsaufwand einher. Da das Bestreben der Auftrennung weitestgehend durch Vorteile für den Hersteller motiviert ist, sollte ein Endnutzer allerdings nur wenige Einschränkungen erfahren müssen. Um das zu gewährleisten, sollte die Umsetzung weitere Anforderungen erfüllen:

- Das Smart Device sollte sich genauso steuern und einbinden lassen wie ein klassisches Smart Device.
- Die Einbindung und Nutzung der Devices sollte für jede Art von Device gleich sein.
- Der zusätzliche Einrichtungsaufwand der Hardware-Komponenten sollte minimal sein.
(*Plug-and-Play*)
- Es sollte keine merkbare Verzögerung hinzukommen.
- Fehlerquellen, die durch die Auftrennung entstehen, sollten behandelt werden.
- Die Anzahl der möglichen verbundenen Geräte sollte nicht verringert werden.

Kapitel 5

Design

Wie zuvor erläutert, ist die *Interoperability* von Smart Devices sowohl für Endkunden als auch für Hersteller eine große Herausforderung. Um diesem Problem zu entgegnen, wird in dieser Arbeit versucht, die Abhängigkeiten bezüglich der Frameworks auf Ebene des Smart Device zu reduzieren. Es wird davon ausgegangen, dass Endkunden ihre bestehenden Systeme behalten wollen und wenig zusätzlichen Aufwand durch das in dieser Arbeit geplante Vorhaben in Kauf nehmen möchten.

Ziel ist es, die Funktionen eines Smart Device für die Hersteller voneinander abzugrenzen und separate Komponenten mit unterschiedlichen Aufgaben zu entwickeln. Die in dieser Arbeit vorgeschlagene Trennung beschreibt eine Komponente als das ‚Smart‘ und eine Komponente als das ‚Device‘. Im weiteren Verlauf dieser Arbeit werden diese Bezeichnungen für die einzelnen Komponenten mit den entsprechenden Funktionalitäten genutzt.

Die aus den vorherigen Arbeiten gewonnenen Erkenntnisse, Fragestellungen und Anforderungen sind dabei in die Konzeption einer Anwendung mit eingeflossen.

Die ersten drei Abschnitte dieses Kapitels beschreiben den Aufbau der Architektur und die Durchführung der Trennung.

Da ein wichtiger Bestandteil der Anwendung in der Datenrepräsentation liegt, beschäftigt sich der nächste Abschnitt mit der Beschreibung unterschiedlicher Repräsentationsweisen und der Entscheidung für eine von ihnen.

Der letzte Abschnitt beschreibt ausführlich die Entwicklung des Protokolls, welches für die Kommunikation zwischen Smart und Device genutzt werden soll.

Im nächsten Kapitel werden diese Überlegungen für die Umsetzung der Anwendung verwendet.

5.1 Architektur

Diese Arbeit sieht die Struktur der von den Kunden verwendeten Frameworks als gegeben an. Ein Nutzer ist nicht dazu bereit, sich ein weiteres Smart-Home-System einzurichten und es neben seinen bestehenden Anwendungen zusätzlich zu nutzen. Demnach werden

in der hier erstellten Architektur die bestehenden Smart-Home-Frameworks als unveränderlich und gegeben betrachtet.

Im Gegensatz zu vielen vorherigen Ansätzen wurde in dieser Arbeit auf der Ebene des Smart Device angesetzt. Die beschriebene Trennung wird durch die Aufteilung eines Smart Device in zwei Komponenten mit unterschiedlichen Funktionen vorgenommen.

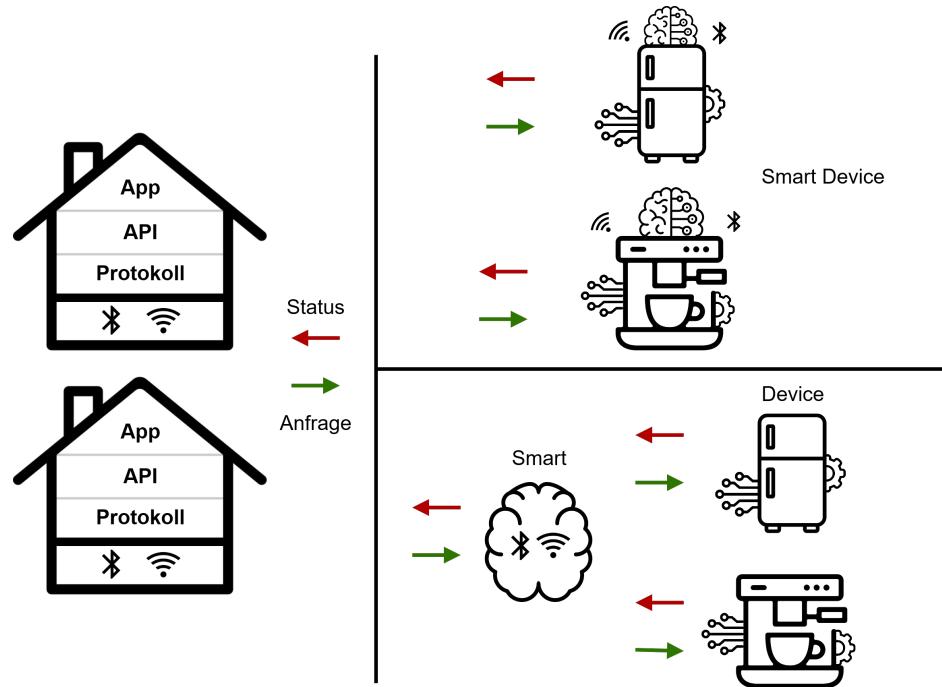


Abbildung 5.1: Architektur

Die Architektur dieses Systems ist in Abbildung 5.1 visualisiert.

Die Funktionen eines Smart Device werden auf ein „Smart“ und ein „Device“ aufgeteilt. Die Anfragen, die bei einem klassischen Smart Device direkt via kabelloser Datenübertragung an das Gerät gesendet werden, werden erst über ein dediziertes Smart gesendet und dann dem Device übergeben.

Das Smart muss dabei über die kabellosen Übertragungsmedien der unterstützten IoT-Frameworks verfügen. Es ist für die Kommunikation mit den verschiedenen Frameworks und die Weiterverarbeitung der Anfragen zuständig. Unterstützte Frameworks sowie deren APIs müssen explizit integriert und unterstützt werden.

Das Smart sendet die Anfragen daraufhin über einen weiteren Datenübertragungskanal an das entsprechende Device weiter. Das Device führt die Anfrage aus und sendet eine Antwort mit dem aktuellen Status zurück an das Smart, welches sie wieder an das passende Framework weiterleitet. So werden die Aufgaben des Smart Device in zwei ver-

schiedene Module aufgeteilt.

Um das zusätzliche Potenzial zu erforschen, welches sich aus dieser Trennung ergeben kann, ist das Smart dazu in der Lage, mehr als ein Device bedienen zu können. Hier wurde sich an dem Ansatz von Homebridge [13] orientiert. Das Smart repräsentiert in diesem Fall eine Bridge oder ein Gateway, welches mit vielen Devices verbunden sein kann. Ein Nutzer benötigt dadurch nicht für jedes Device ein neues Smart. So erübrigen sich beispielsweise die Anschaffungskosten und der Einrichtungsaufwand für weitere Smarts.

Es ist nicht auszuschließen, dass mehrere Smarts mit demselben Device verbunden sein können. Daher bietet die entworfene Architektur die Möglichkeit, auch mehrere Smarts mit demselben Device kommunizieren zu lassen.

Beide Fälle können in einem Anwendungsfall gleichzeitig vorliegen. In einem theoretischen Rahmen können mehrere Smarts jeweils mit mehreren Devices verbunden sein. Falls ein Nutzer zum Beispiel zwei Smarts besitzt, kann er jedes von ihnen mit denselben Devices verbinden und koppeln (2:1, 2:2, 2:3, ...).

Bei der Konfiguration von Smart und Device ergibt sich eine $M:N$ Relation. Wobei M die Anzahl der simultan existierenden Smarts beschreibt und N die Anzahl der möglichen Devices angibt, die mit jedem Smart verbunden sein können. M ist dabei limitiert durch die Anzahl der maximal einzubindenden Geräte in das entsprechende IoT-Framework. N ist limitiert durch die Rechenkapazität der Smarts.

Die Limitierung mit dem stärksten Einfluss ist allerdings durch die Anzahl an Kommunikationsschnittstellen gegeben, die sowohl von den Smarts als auch von den Devices zur Verfügung gestellt werden müssen.

5.2 Device

Die Devices sind in dieser Arbeit eingebaute Geräte von Herstellern, die über keine kabellosen Kommunikationsschnittstellen verfügen. Das Konzept ist zusätzlich erweiterbar durch die direkte Einbindung von Smart Devices, da sich die Kommunikation mit ihnen lediglich durch die Beschriftenheiten der Schnittstellen unterscheidet. Auf diese und weitere mögliche Erweiterungen wird am Ende der Arbeit näher eingegangen.

Das Device erfüllt die von dem Hersteller spezifizierten Anwendungsgebiete. Es beinhaltet Sensoren für die Datenerfassung und geeignete Aktuatoren für eine Interaktion. In Abbildung 5.2 wird ein Beispiel für ein solches Device und seine Eigenschaften gegeben. Das Device ist in diesem Fall ein Kühlschrank. Wichtige Eigenschaften eines Kühlschranks sind zum Beispiel die aktuelle oder gewünschte Temperatur.

Der Zustand eines Device kann aus seinen Eigenschaften abgeleitet werden. Ähnlich wie durch die *Representativeness* von Digital Twins [22] beschrieben wird, reichen wenige Eigenschaften dabei aus. Im abgebildeten Beispiel kann so der Zustand und die Arbeitsphase des Kühlschranks eindeutig erfasst werden. Die gewünschte Temperatur ist niedriger als die aktuell gemessene Temperatur. Zudem ist das Licht des Kühlschranks

angeschaltet. Das bedeutet, der Kühlschrank sollte sich in der Kühlphase befinden. Durch die zusätzliche Information, dass das Licht an ist, kann geschlossen werden, dass der Kühlschrank geöffnet ist. Diese Annahmen beruhen auf dem bekannten Wissen über die Funktionsweise eines Device.

Die Interaktion mit dem Device geschieht durch die Interaktion mit den Feldern (Eigenschaften), die es zur Verfügung stellt. Es existieren Eigenschaften, die durch Anfragen von außen veränderbar sind (Set) und Eigenschaften, von denen lediglich der aktuelle Status abgerufen werden kann (Get). Sie sind vergleichbar mit den *Properties*, wie sie im WoT [32] beschrieben werden. Die aktuelle Temperatur kann sich zum Beispiel durch die von Sensoren empfangenen Informationen ändern. Diesen Wert durch eine Anfrage explizit zu setzen, ergibt für die Funktionsweise des Gerätes keinen Sinn. Solche Werte werden als Read-Only angesehen.

Veränderbar sind Werte wie die gewünschte Zieltemperatur. Diese Werte können durch Anfragen von Außen (Set) gesetzt werden.

Für diese Arbeit wird angenommen, dass es sich bei den Devices um statische Geräte handelt, die über eine gesicherte Stromversorgung verfügen und ihre Position nicht häufig ändern. Dadurch führt die physische Verbindung von Smart und Device nicht zu Einschränkungen in der Portabilität.

Es wird angenommen, dass sich die Funktionsweise und damit die Interfaces der Devices nicht ändern. Die Fähigkeiten eines Kühlschranks ändern sich dann, wenn zusätzliche Funktionen durch die Integration neuer Hardware entstehen. Da die grundsätzliche Funktionalität bestehen bleibt, muss dieses Interface nur selten angepasst werden. Ein Device wird in dieser Arbeit unabhängig von seinen Funktionen und Eigenschaften standardisiert angesprochen und gesteuert. Wie die Kommunikation mit den Smarts im Detail aussieht, wird im Abschnitt ‚Kommunikation‘ ausführlich behandelt.

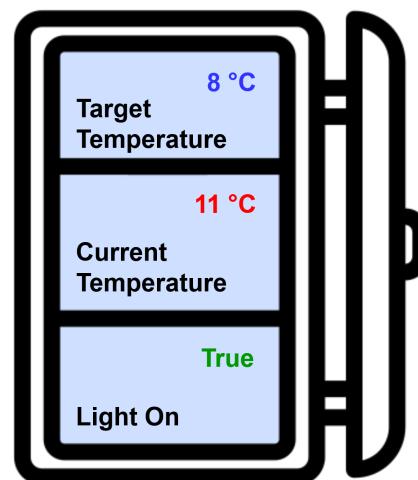


Abbildung 5.2: Eigenschaften eines Kühlschranks

5.3 Smart

Smarts sind separate Geräte, die mit einem oder mehreren kabellosen Übertragungsmedien ausgestattet sind, um die Kommunikation mit den IoT-Frameworks zu gewährleisten. Sie bilden die Abstraktionsebene zwischen den Frameworks und dem Device.

Das Smart ist über einen anderen Übertragungskanal mit einem oder mehreren Devices verbunden. Es verarbeitet die Nachrichten und leitet sie weiter.

Die Framework-spezifischen Anforderungen und Libraries werden hier abstrahiert, damit sich die Hersteller der Devices auf ihre Hauptkompetenzen konzentrieren können.

Viele Eigenschaften bezüglich der *Maintainability* sowie der Anpassung an Änderungen in den Frameworks werden von einem Smart übernommen. Ähnlich wie in dem Ansatz von Salzman et al. [28], werden die Daten auf dem Smart lokal gespeichert, um eine schnelle und konsistente Interaktion mit den Geräten zu gewährleisten.

In dieser Arbeit wird angenommen, dass ein Device über eine gesicherte Stromversorgung verfügt. Ein Smart kann sie ebenfalls nutzen und daraus seinen Strom beziehen.

5.4 Digital Twin

Um sinnvoll mit dem Device interagieren und seinen physischen Zustand in der Welt erfassen zu können, muss ein geeignetes digitales Abbild erstellt werden. Für die Datenrepräsentation wurde sich in an dem zuvor behandelten Konzept der Digital Twins orientiert.

Bei einem klassischen Smart Device ist es das einzige Gerät, welches direkt über die eigenen Sensordaten verfügt. Es ist dazu verpflichtet, diese Daten weiterzuverarbeiten und dem Framework zu übermitteln. Die Rohdaten der Sensoren sind für das darüberliegende Framework unbrauchbar. Bei einem klassischen Smart Device stellt sich nicht die Frage, an welcher Stelle ein Abbild am sinnvollsten ist.

Durch die Auf trennung in zwei unterschiedliche Geräte ergeben sich jedoch mehrere Möglichkeiten, diese Digitalisierung umzusetzen. Sie haben unterschiedliche Vor- und Nachteile und bringen sowohl zusätzliche Funktionen als auch Einschränkungen mit sich. Im Falle des ‚Living Lab‘ von Simeoni et al. [31], wurde dieses Abbild in einer Cloud vorgenommen. Der Vorteil einer lokalen Repräsentation liegt in der niedrigen Latenz und der simpleren Einrichtung des Smart. Bei der Nutzung einer Cloud-basierten Speicherung müssen im privaten Bereich zusätzliche Sicherheitsmaßnahmen vorgenommen werden. Daher ist eine lokale Repräsentation in dieser Arbeit sinnvoller.

Im Folgenden werden zwei dieser Möglichkeiten vorgestellt und voneinander abgegrenzt. Anschließend wird auf die für diese Arbeit gewählte Variante näher eingegangen und die getroffene Entscheidung begründet.

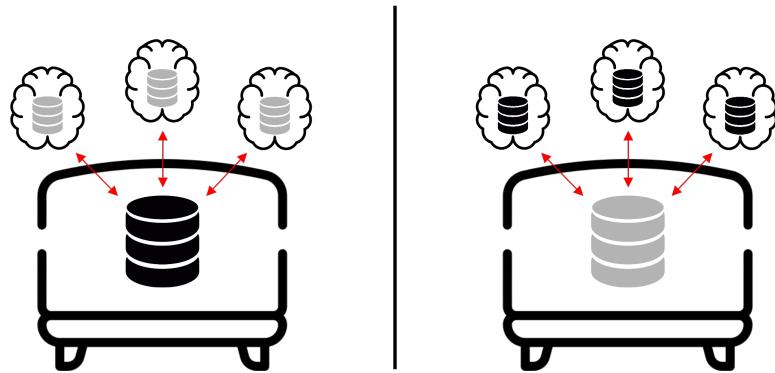


Abbildung 5.3: Mögliche Umsetzungen des Digital-Twin-Paradigmas

5.4.1 Abbild auf Device

Diese Variante ist auf der linken Seite der Abbildung 5.3 dargestellt. Die Speicherung des Zustandes wird direkt auf dem Device vorgenommen. Die Smarts besitzen zusätzliche lokale Kopien des Abbildes.

Das Abbild auf dem Device selbst vorzunehmen, ist die intuitive Variante. Sowohl die Sensoren als auch die Aktuatoren sind direkt mit dem Gerät verbunden oder in das Gerät integriert.

Um seinen Zustand entsprechend zu verändern, muss es dazu in der Lage sein, ihn genau zu erfassen. Bei einem Thermostat beispielsweise muss die aktuelle Temperatur an die gewünschte Temperatur angepasst werden. Dazu werden beide Temperaturen mehrmals in kleinen Schritten abgeglichen. Gegebenenfalls müssen Heiz- oder Kühlmechanismen aktiviert werden. So muss das Gerät auch ohne die Eigenschaften eines Smart Device eine gewisse *Autonomy* besitzen. Aufgrund der Nähe zum Objekt ist es sinnvoll, dass diese Anpassungen direkt im Gerät vorgenommen werden können.

Das Smart dient in dieser Variante als eine Art Vermittler zwischen dem Framework und dem Device. Es leitet sowohl Anweisungen von dem Framework als auch die Antworten von dem Device jeweils weiter.

In der Theorie ist es ausreichend, wenn das Smart alle Nachrichten lediglich weiterleitet. Der große Vorteil, das Abbild auf dem Device zu halten, liegt allerdings darin, dass sich die Eigenschaft der *Replication* zunutze gemacht werden kann. Nach Minerva et al. [22] können von einem Logical Object viele Kopien existieren, solange sie auf dasselbe physische Objekt zurückzuführen sind und seinen Zustand abbilden.

Die Master Replica befindet sich auf dem Device. Jedes angeschlossene Smart hält eine Kopie dieses Abbildes. Die Replicas auf den Smarts werden durch die Master Replica auf dem Device synchronisiert. Bei dieser Variante gibt es exakt eine Wahrheit über den Zustand des Gerätes. Sie liegt auf dem Device. Alle weiteren Kopien orientieren sich an dieser Wahrheit. Das trägt erheblich zu der Gesamtstabilität des Systems bei.

Mit geeigneten Synchronisierungsmechanismen wie zum Beispiel durch die Nutzung einer Checksum, kann so jederzeit sichergestellt werden, dass der abgebildete Zustand auf allen Geräten übereinstimmt.

Wird angenommen, dass jede Änderung auf dem Device von genau einem Smart initiiert werden muss, dann wäre kein zusätzliches Abbild auf dem Smart nötig. Sendet das Smart eine Anfrage, kann es anhand der Antwort oder durch einen möglichen Timeout Fehler in der Übertragung erkennen. Da sich der Zustand des Device jedoch auch autonom oder durch ein weiteres Smart ändern kann, reicht das Weiterleiten der Nachrichten nicht aus. Ohne eine Synchronisierung wird es dann schwierig, das Auftreten eines Fehlers festzustellen.

Durch die Master Replica und eine korrekte Synchronisierung können auch Fehler in der Übertragung erkannt werden, wenn ohne ein Smart keine Anfrage gesendet hat. Die genauen Mechanismen zur Fehlererkennung und Synchronisierung werden im Abschnitt „Kommunikation“ näher betrachtet.

Da die Wahrheit über den Zustand in diesem Fall in den Devices liegt, wird angenommen, dass das Device eine längere Laufzeit besitzt als das Smart. Es sollte an einer externen Stromquelle angeschlossen sein. Diese Variante ist sinnvoll für stationäre Geräte wie Kühlschränke oder Heizungen, deren Stromzufuhr sichergestellt ist.

Die Devices müssen sich darum kümmern, dass ihr Zustand abgespeichert und persistiert wird. Bei stationären Geräten ist das in den meisten Fällen gegeben, da ihr Anwendungszweck auch ohne die *Smartness* sichergestellt werden muss.

Ein Kühlschrank hat beispielsweise bereits Voreinstellungen und kann seine Funktion ausführen, ohne dass Anweisungen über das Smart gesendet werden müssen. Wird die Stromzufuhr nicht mehr gewährleistet, dann sollte das Gerät vonseiten des Herstellers aus dazu in der Lage sein, seinen Zustand entsprechend zu sichern. Schließt ein Nutzer ein Smart an, kann dieser den aktuellen Zustand des Gerätes abfragen und sein Replica synchronisieren. So bleibt das Gerät unabhängig von dem Smart funktionsfähig. Die *smarte* Komponente erweitert in dieser Variante das Device also um einige Zusatzfunktionen.

5.4.2 Abbild auf Smart

Diese Variante ist auf der rechten Seite der Abbildung 5.3 dargestellt. Der Zustand wird hier hauptsächlich auf dem Smart gespeichert. Um einige Funktionen gewährleisten zu können, wird eine lokale Kopie des Zustandes auf dem Device gehalten.

Die Wahrheit über den Zustand des Gerätes liegt aufseiten des Smart. Das bedeutet, dass alle Einstellungen und Eigenschaften des Gerätes auf dem Smart gespeichert werden. Daraus entstehen einige zusätzliche Funktionen, allerdings auch viele Einschränkungen. Dasselbe Konzept der *Replication* wie im anderen Fall ist in hier nicht anwendbar.

Bei einer 1:1 Relation zwischen Smart und Device ist eine Synchronisierung unproblematisch. Der Zustand des Smart kann als Wahrheit angesehen werden. Das Smart sendet

seinen Zustand kontinuierlich an das Device und übermittelt ihm dadurch seine Wahrheit.

Wird diese Relation jedoch durch die Existenz eines weiteren Smart aufgehoben, wird eine Synchronisierung schwieriger. In diesem Fall existiert mehr als eine Wahrheit. Im Fehlerfall kann es zu Abweichungen kommen.

Da die Smarts in keiner direkten Verbindung zueinander stehen, ist eine Synchronisierung nur über das Device möglich. Das Device kann seine Funktionalität auch bei dieser Variante ohne ein Smart ausführen. Da die Wahrheit allerdings von dem Smart ausgeht, wird sich der Zustand ändern, sobald ein Smart angeschlossen wird. Das Smart sendet dann seinen letzten gespeicherten Zustand an das Device. Das ermöglicht beispielsweise das Speichern der eigenen Konfiguration und das anschließende Übertragen auf andere Devices.

Wenn der Nutzer auf ein neues Device umsteigen möchte oder in eine andere Wohnung zieht, kann er sich dadurch eine erneute Konfiguration sparen. Dafür müssen alle Daten auf dem Smart persistent sein. Das kann bei vielen verbundenen Devices ein erheblicher Aufwand sein. Für eine optimale Ausführung sollte der Zustand nach jeder Änderung gespeichert werden. Dieser Vorgang kann weitere Verzögerungen mit sich bringen.

Es kann besonders dann problematisch werden, wenn die Stromzufuhr abrupt unterbrochen wird. Bei kleinen Datenmengen kann ein explizit dafür entwickelter Mechanismus die Daten vor dem Herunterfahren sichern. Je mehr Geräte und ihre Eigenschaften hinzukommen, desto unsicherer ist eine korrekte Persistenz.

5.4.3 Auswahl

Für diese Arbeit wurde die Variante gewählt, die Wahrheit über den aktuellen Zustand auf dem Device zu halten (Abbildung 5.3 links). Dadurch kommt es zu keinen Komplikationen darüber, in welchem Zustand sich das Device befindet. Das Smart muss wesentlich weniger Daten verwalten und speichern. Es übernimmt die wichtigen Aufgaben und muss sich nicht um komplexes Speichermanagement oder die damit verbundene Fehlerbehandlung kümmern. Durch das *Replication-Scheme* wird ein reibungsloses und konsistentes State-Management bereitgestellt.

Auch wenn die Zusatzfunktionen bei dieser Variante begrenzt sind, überwiegen hier die Vorteile.

5.5 Kommunikation

Um die Funktionsfähigkeit des getrennten Smart Devices zu gewährleisten, muss die reibungslose Kommunikation zwischen Smart und Device sichergestellt werden. Die zuvor behandelten Übertragungsmedien sind für diese Aufgabe jeweils mit einigen Vor- und Nachteilen verbunden. Aus diesem Grund beschäftigt sich der folgende Unterabschnitt

mit Überlegungen bezüglich der zu verwendenden Übertragungsmedien und ihrer Eigen-schaften.

Anschließend wird das in dieser Arbeit entwickelte Übertragungsprotokoll vorgestellt, welches für die Kommunikation genutzt werden soll. Der Fokus liegt dabei auf der Syntax und Semantik des Protokolls sowie der notwendigen Fehlerbehandlung, die in der Kommunikation eingesetzt wird.

5.5.1 Übertragungsmedium

Aufgrund der simplen Struktur wurde sich in dieser Arbeit für serielle Übertragungs-medien entschieden. Aus dem Kapitel ‚Grundlagen‘ kann jedoch entnommen werden, dass serielle Schnittstellen sehr unterschiedlich sein können. Sie besitzen inkompatible Topologien, kommunizieren mithilfe unterschiedlich aufgebauter Pakete und benötigen verschieden viel Konfigurationsaufwand.

Ein Kommunikationsmedium, welches für die Kommunikation zwischen Smart und Device eingesetzt werden soll, muss zudem einige Funktionen mit sich bringen. Der Zu-stand eines Smart Device kann sich aus unterschiedlichen Quellen ändern. Neben dem Senden einer Anfrage und dem Empfangen der Antwort können aufseiten des Smart auch Benachrichtigungen über Statusänderungen ankommen, ohne dass es zuvor eine Anfrage gesendet hat. Das wird besonders schnell ersichtlich bei der Betrachtung des Multi-Smart-Konzeptes, bei dem eine Anfrage von einem anderen Smart gesendet werden kann.

Wird die Eigenschaft der *Autonomy* betrachtet, können sich aus der Interaktion mit der Umgebung zusätzlich einige Zustandsänderungen ergeben.

Nicht jedes Übertragungsmedium ist für diese Betriebsweise geeignet. Der naheliegende Ansatz ist die Nutzung eines asynchronen Mediums, bei dem beide Seiten gleicherma-ßen dazu berechtigt sind, Nachrichten zu senden und die Kommunikation zu initiieren (UART). So können sowohl Anweisungen des Frameworks als auch Benachrichtigungen vonseiten des Device zu jedem Zeitpunkt gesendet und verarbeitet werden.

Technologien wie UART scheinen daher für die Anwendung in dieser Arbeit besser ge-eignet zu sein als Technologien wie I²C oder SPI, bei denen nur der Master eine Kom-munikation einleiten kann.

Es gibt allerdings Möglichkeiten, asynchrone Konzepte über Umwege mit synchronen und asymmetrischen Medien zu kombinieren. Das SPI-Protokoll zum Beispiel bietet durch einen Pull-Up-Resistor auf jeder Chip-Select-Line die Möglichkeit eine Übertragungsan-frage anzukündigen [34].

Der Master kann auf das Signal reagieren und die Daten von dem entsprechenden Slave anfragen.

Bei I²C ist das durch die Adressierung in den Paketen und die Open-Drain-Verbindung auf dem Bus nicht umsetzbar. Diese Funktionalität ist jedoch durch das Hinzufügen von separaten Signaling-Lines möglich. Sie können auf eine ähnliche Weise wie die Chip-

Select-Lines genutzt werden.

Neben dem beidseitig initiierbaren Senden, ermöglicht das auch zusätzliche Funktionen wie Sleep und Awake.

Hat sich der Zustand des Device geändert, kann das Device die Änderung in einem Buffer speichern. Anschließend sendet er ein Signal auf der Signaling-Line. Empfängt ein Smart dieses Signal, kann es sich sicher sein, dass das Device neue Daten bereitgestellt hat und diese abholen. Dadurch können die Stärken asymmetrischer Medien wie die zentrale Verknüpfung mehrerer Geräte auch in dieser Arbeit genutzt werden.

5.5.2 Protokoll

Das entwickelte Protokoll beschreibt die Struktur der zu sendenden Daten sowie mögliche Nachrichtenabfolgen und ihre Semantik. Es wird als zusätzliche Abstraktionsschicht auf das jeweilige Datenübertragungsprotokoll gesetzt und beinhaltet eine eigene Form der Adressierung. Es stellt zusätzliche Mechanismen zur Erkennung und zum Beheben von Übertragungsfehlern bereit. Für die Entwicklung eines Device wird vorausgesetzt, dass es dieses Protokoll mit einem unterstützten seriellen Übertragungsmedium implementiert.

Nachricht

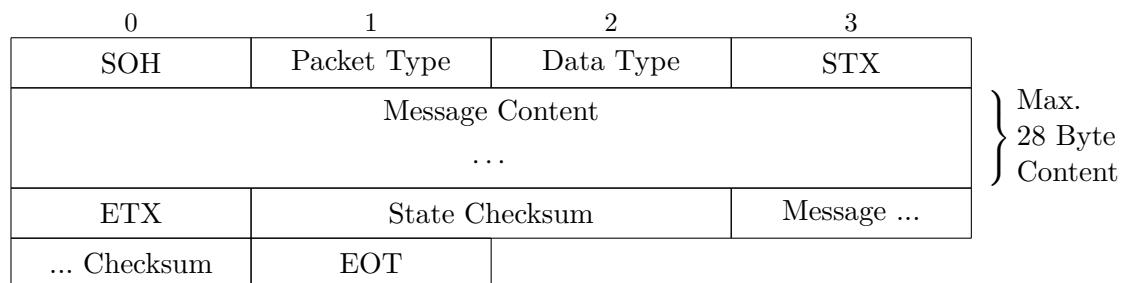


Abbildung 5.4: Aufbau einer Nachricht

Das Interface des Smart sollte von vielen Herstellern einfach zu implementieren sein. Die Kommunikation mit den Smart Devices besteht in den meisten Fällen aus Befehlen. Die Semantik dieser Befehle ist an Funktionsaufrufe angelehnt. Sie nehmen Parameter entgegen und geben nach einer bestimmten Verarbeitungszeit ein Ergebnis zurück.

Die Smart Devices unterscheiden sich größtenteils in ihren Attributen und in den zur Verfügung stehenden Funktionen. Für einen Kühlschrank könnte eine Anweisung beispielsweise so aussehen: „Setze die Temperatur auf 6 Grad“.

Damit die Anweisungen intuitiv und direkt zugeordnet werden können, wurde für das

Protokoll die textbasierte 7-Bit-Kodierung ASCII ausgewählt. Der ASCII bietet den Vorteil, dass er eine menschenlesbare Kodierung ermöglicht und Mechanismen wie Stuffing und Escaping bereits durch geeignete Steuerzeichen implementiert.

Jedes Zeichen innerhalb einer Nachricht ist ein Byte lang. Die Checksums werden jeweils mit zwei Zeichen dargestellt. Der Aufbau einer Nachricht kann in Abbildung 5.4 eingesehen werden.

Die Struktur eines Paketes wird durch die spezifischen ASCII-Steuerzeichen vorgegeben. Da jedes Paket eine einzelne abgeschlossene Nachricht darstellt, wurde sich gegen eine zeilenbasierte Form der Kommunikation entschieden. Stattdessen wird ein Paket durch die Zeichen für *Start of Header (SOH)* sowie *End of Transmission (EOT)* eingegrenzt. Nach dem *Start of Header* folgen zwei Header-Bytes, die den Inhalt der Nachricht näher beschreiben.

Der Inhalt der Nachricht wird von den Zeichen für *Start of Text (STX)* und *End of Text (ETX)* eingeschlossen. Zwischen den beiden Zeichen ETX und EOT befinden sich zwei Checksums, die für die Fehlererkennung verwendet werden. Sie sind in dieser Anwendung ebenfalls mit ASCII kodiert. Das muss gemacht werden, damit es zu keinen Komplikationen bei der Erkennung der Steuerzeichen kommt.

Da die Pakete in diesem Protokoll eine dynamische Größe haben, ist es wichtig, dass die Steuerzeichen eindeutig zuzuordnen sind. Sie geben den Anfang und das Ende eines Paketes an.

Die Nutzung von unkodierten Integers innerhalb eines Paketes kann dazu führen, dass sie zufällig den Wert eines Steuerzeichens annehmen. Um das zu verhindern, muss sichergestellt werden, dass die Checksum bestimmte Werte nicht erreichen kann. Bei Checksums bedeutet die Löschung bestimmter Werte, den Wertebereich des Endergebnisses zu verkleinern und somit eine unsichere Summe mit mehr Kollisionen zu erzeugen. Da eine 1-Byte Checksum bereits einen relativ kleinen Wertebereich hat, wurde eine andere Lösung gewählt. Die Checksum wird ebenfalls mit dem restlichen Paket kodiert. Dabei wurde sich an der Vorgabe der ASCII-Checksum orientiert. Die Checksum wird aus dem Dezimalsystem in das Hexadezimalsystem überführt. Anschließend werden die einzelnen Stellen dieser Zahl mit den entsprechenden ASCII-Zeichen ersetzt. So ist sichergestellt, dass die Checksum nur aus korrekten ASCII-Zeichen besteht. Im Vergleich zu der direkten Übertragung der Summe in einem einzigen Byte kommt durch diesen Vorgang ein weiteres Byte hinzu. Eine 1-Byte Checksum wird als eine 2-Byte Hexadezimalzahl übertragen.

	State	Checksum:	Checksum:	
		9 (0x09)	187 (0xBB)	Länge
1. Dezimal (variabel):		57	44	49 56 55
2. Dezimal (fix):		48 48 57		49 56 55
3. Hexadezimal:		48 57		66 66

Abbildung 5.5: Vergleich der Checksums

Die Umformung in das Hexadezimalsystem gegenüber der direkten Kodierung der Summe im Dezimalsystem hat den Vorteil, dass sie immer dieselbe Größe besitzt. Im Dezimalsystem kann die Summe zwischen 1 und 3 Byte groß sein. Daher ist es entweder notwendig, ein zusätzliches Trennzeichen für beide Summen in dem Paket einzuführen oder die Anzahl an Zeichen auf exakt 3 Byte festzulegen. Nimmt die Summe dann nur 1 Byte in Anspruch, können die restlichen Bytes mit 0 aufgefüllt werden.

Wie diese ASCII-kodierten Checksums im direkten Vergleich aussehen, ist in Abbildung 5.5 dargestellt. Wird der erste Fall betrachtet, können beide Checksums im Worst Case 7 Byte in Anspruch nehmen. Im zweiten Fall sind sie exakt 6 Byte groß. Die Nutzung der Hexadezimal-Repräsentation (Fall 3) limitiert die Größe einer Checksum auf exakt 2 Byte. Beide Checksums zusammen nehmen somit 4 Byte ein.

Die Funktionen der einzelnen Checksums werden in einem späteren Abschnitt genauer behandelt.

Beispiele für korrekt formatierte Nachrichten sind in Abbildung 5.6 zu sehen. Dabei wird zwischen zwei Typen von Nachrichten unterschieden: Die Anfragen (Cmd) und die Antworten (Reply). Der *Packet Type* beschreibt die Art der Nachricht. Durch ihn wird der Kontext der Nachricht und damit ihre Verarbeitungsweise auf Seiten des Empfängers festgelegt.

Eine Anfrage sollte nie von einem Device gesendet werden. Das Verhältnis zwischen Smart und Device ist asynchron. Das Device nimmt Anfragen entgegen, verarbeitet sie und sendet Antworten zurück an das Smart. Empfängt ein Smart eine Anweisung oder ein Device eine Antwort, so kann davon ausgegangen werden, dass ein Fehler aufgetreten ist.

Durch die Angabe des *Packet Type* stehen Nachrichten im Ablauf einer Kommunikation stellvertretend für die Nachrichten eines bestimmten Gerätes. Ein Smart sendet Anfragen und empfängt Antworten. Ein Device empfängt Anfragen und sendet Antworten.

SOH	c	s	STX	tar_temp,20.0	ETX	0x67	0x85	EOT	Cmd
SOH	r	ACK	STX	tar_temp,20.0	ETX	0x69	0x33	EOT	Reply

Abbildung 5.6: Beispielnachrichten

Das Feld im Header kann die Werte ‚c‘(Command) oder ‚r‘(Reply) annehmen. Der Typ einer Nachricht bestimmt die möglichen Werte des *Data Types*. Die Anfrage eines Smart kann entweder vom Typ ‚Get‘ oder vom Typ ‚Set‘ sein. Die Antwort eines Device kann ein Acknowledgement oder ein Negative Acknowledgement beinhalten. Dementsprechend kann das Feld *Data Type* im Header folgende Werte annehmen: ‚g‘(Get), ‚s‘(Set), ‚ACK‘(Acknowledgement) und ‚NAK‘(Negative Acknowledgement/NACK).

Das Feld liefert Zusatzinformationen über den Inhalt der Nachricht. Handelt es sich bei der Nachricht um eine Anfrage, muss zwischen zwei Arten von Anfragen unterschieden werden. Es gibt Anweisungen, die Änderungen am Zustand der Smart Devices hervorrufen sollen (Set) und es gibt Nachrichten, die den aktuellen Zustand anfragen (Get). Anstatt die Information über die Intention der Anfrage durch einen Präfix (*getTemperature*) in den Inhalt einzubauen, wurde sich dafür entschieden, diese Information in den Header auszulagern. Das hat den Vorteil, dass direkt ersichtlich ist, welcher Zweck mit dem Paket erreicht werden soll. Es vereinfacht die Verarbeitung aufseiten des Empfängers und ermöglicht die dynamische Nutzung des zweiten Header-Bytes. Im Kontext einer Antwort haben sowohl der *Data Type* als auch der *Content* eine andere Bedeutung. Das Feld *Data Type* repräsentiert eine positive oder negative Antwort auf die vorherige Anfrage des Smart.

Der Inhalt der Nachricht ist dabei eine Wiederholung des Inhaltes der vorherigen Anfrage. Ist die Antwort positiv, kann das Smart davon ausgehen, dass die Anweisung erfolgreich durchgeführt wurde. Ist die Antwort negativ, kann es davon ausgehen, dass die Anweisung nicht ausgeführt wurde und eine Fehlerbehandlung einleiten.

Durch die Binäre Auftrennung der Header-Felder, wäre es möglich, die Header-Informationen auch durch einzelne Bits anzugeben. Bei dem *Packet Type* könnte ‚Command‘ zum Beispiel durch eine 0 und ‚Reply‘ durch eine 1 repräsentiert werden. Der *Data Type* könnte so zwei verschiedene Aussagen haben (0 ⇒ Set und NAK / 1 ⇒ Get und ACK). Abhängig von dem *Packet Type* wäre der Header der Nachricht eindeutig zu bestimmen. Durch die Nutzung von ASCII und der damit verbundenen Konvention der Kodierung einzelner Bytes, müsste der Header entsprechend mit 6 weiteren Bits aufgefüllt werden. Dieser Vorgang wird auch Padding genannt. Der Header wäre in diesem Fall 1 Byte lang. Da die Ersparnis von einem Byte pro Nachricht sehr gering ist, wurde sich gegen die

kurze Form entschieden. Der Aufbau der Nachricht ist dadurch intuitiver und besser nachzuvollziehen.

Im *Content* der Nachricht befindet sich die Eigenschaft oder das Feld auf das sich die Nachricht bezieht. In dem Beispiel in 5.6 beziehen sich sowohl die Anfrage als auch die Antwort auf das Feld *Target Temperature*. Ist die Nachricht wie in diesem Fall eine Anfrage, die einen Wert des Device ändern soll, wird zudem der neue gewünschte Wert mitgesendet. Der Wert wird mit einem Komma hinter der Eigenschaft angekündigt. Welche Werte ein Feld entgegennimmt, ist durch das Device und die Eigenschaft vorgegeben. In dem dargestellten Beispiel kann die Eigenschaft *tar_temp* Float-Werte in einem definierten Bereich annehmen. Die explizite Schreibweise der Eigenschaften ist für jedes Device festgelegt. Um Ressourcen zu sparen, sollte die maximale Größe von 28 Bytes von keinem *Content* erreicht werden. Für eine eindeutige Identifikation einer bestimmten Eigenschaft ist es nicht notwendig, den Bezeichner jedes Feldes auszuschreiben. Daher wird in vielen Fällen eine Abkürzung gewählt. In dem genannten Beispiel wird *Target Temperature* durch *tar_temp* abgekürzt.

Adressierung

Die Adressierung wird in dieser Arbeit nicht in jedem einzelnen Paket vorgenommen. Es wird angenommen, dass sich die Adresse von einem Gerät zur Laufzeit des Smart nicht aktiv ändert. Die Adressierung wird für die Identifikation der Devices in einer eigenen Struktur genutzt und an die physische Schnittstelle gekoppelt, sobald eine Verbindung hergestellt wurde. Ändert sich die Adresse eines Device zur Laufzeit, dann ist das unproblematisch, da die Kopplung auf die alte ID einmalig vorgenommen wurde. Solange das Gerät an der physischen Schnittstelle nicht ausgetauscht wird oder ein Gerät mit derselben Adresse verbunden wird, bleibt die Adressierung korrekt. Diese Kopplung wird aufgehoben, sobald auf der physischen Verbindung für eine längere Zeit keine Kommunikation mehr festgestellt werden konnte (Timeout). Die Adressierung bleibt auch korrekt, wenn die Daten über Bus-basierte Medien wie I²C oder SPI gesendet werden. Die Kopplung geschieht dann nicht nur über die physische Schnittstelle, sondern zusätzlich über die logische Adresse innerhalb dieser Protokolle.

Nachrichtenaustausch

Alle Nachrichten zwischen Smart und Device werden asynchron übertragen. Das bedeutet, es wird nicht aktiv auf eine Antwort gewartet. Bei der Verwendung eines asymmetrischen und synchronen Protokolls wird die Asynchronität wie zuvor erläutert, durch zusätzliche Hilfsmittel erreicht. So können auch Update-Nachrichten an die Smarts gesendet werden, falls sich der Zustand des Device aus anderen Quellen heraus geändert hat. Ein Beispiel für einen erfolgreichen Nachrichtenaustausch ist in Abbildung 5.7 zu sehen.

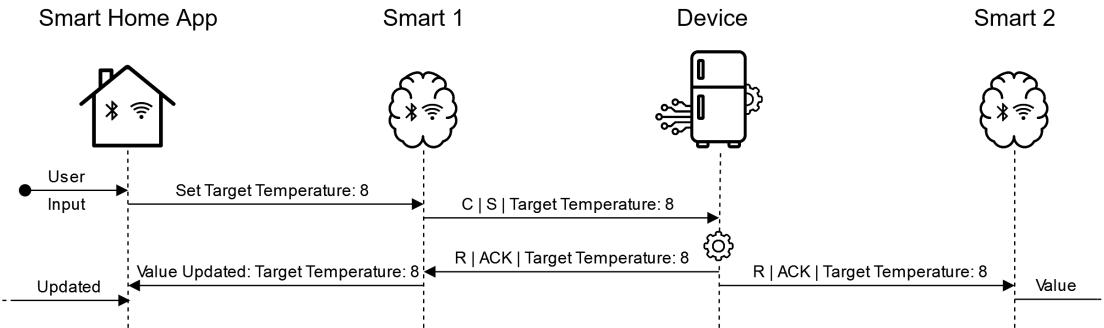


Abbildung 5.7: Erfolgreicher Nachrichtenaustausch

Zunächst wird eine Anfrage von der mobilen Smart-Home-Anwendung über einen kabellosen Übertragungskanal an das Smart gesendet. Anschließend wird die Anfrage in dem Smart verarbeitet und in das dem Protokoll entsprechende Format überführt. Das Paket wird dann an das Device weitergeleitet. Hat das Device die Anfrage verarbeitet, sendet es eine Antwort an alle verbundenen Smarts zurück. Es muss nicht darauf achten, welches Smart die Anfrage gesendet hat, da aus Gründen der Konsistenz eine Benachrichtigung an alle anderen Smarts gesendet wird.

Jedes Smart stellt eine eigene Instanz in der Smart-Home-Anwendung dar und agiert getrennt voneinander. Aus diesem Grund informiert es das Framework direkt über entsprechende Änderungen.

Eine Benachrichtigung wird auch gesendet, wenn sich eine Zustandsänderung aus dem Programmablauf des Device ergeben hat. Sie ist somit äquivalent zu der Antwort auf eine Anfrage. Hier wurde keine Unterscheidung gewählt, da es nicht wichtig ist, wodurch sich der Wert einer Eigenschaft geändert hat. Es wird kein zusätzlicher Aufwand benötigt, um dieselbe Semantik bereitzustellen. Bei einer Zustandsänderung aus dem Programmablauf entfallen die Anfragen von der App und dem Smart. Der restliche Ablauf bleibt gleich.

Eine Antwort enthält die Zustandsänderung sowie einen positiven oder negativen Indikator (ACK, NACK) bezüglich einer vorherigen Anfrage. Der Indikator gibt an, ob die Anfrage korrekt verarbeitet werden konnte und eine Zustandsänderung ausgelöst hat. Da der Inhalt der Anfrage auch bei der Benachrichtigung mitgesendet wird, kann das Smart auf seiner lokalen Repräsentation entsprechend dieselbe Zustandsänderung vornehmen.

Ist kein Fehler aufgetreten, sollten die Abbilder von Smart und Device nach Empfang und Anwendung der Zustandsänderung synchronisiert sein. Anschließend wird die Smart-Home-Anwendung über die Änderung informiert.

Da nicht zwischen Benachrichtigungen und Antworten unterschieden werden kann, wird die Anwendung auch dann informiert, wenn die Änderung bereits durch dasselbe Smart angefordert wurde. Dadurch kann zusätzlich sichergestellt werden, dass die Änderungen

konsistent übernommen werden.

Für die Abfrage der Daten vonseiten des Smart Homes wurde sich an Salzman et al. [28] orientiert. Sie schlagen die Nutzung einer lokalen Repräsentation vor, um eine schnelle Interaktion mit den Smart Devices zu gewährleisten.

Wenn die Konsistenz mit dem korrekten Abbild des Device sichergestellt ist, müssen keine Read-Anfragen an das Device weitergeleitet werden. Die Last auf Smart und Device wird reduziert und die Latenz dieser Anfragen zusätzlich verringert.

Startup-Sequenz

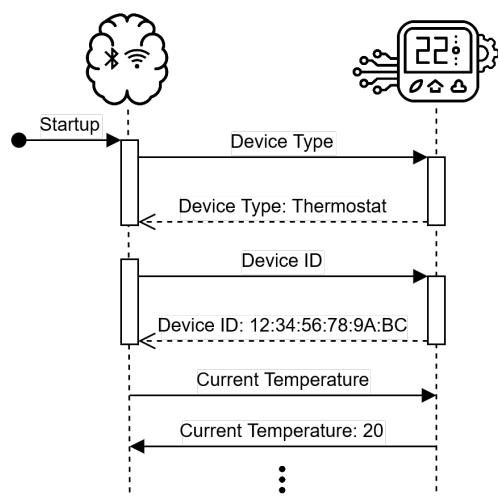


Abbildung 5.8: Startup-Sequenz

Da die Einbindung der Devices in die Smarts keinen weiteren Einrichtungsaufwand (Plug-and-Play) nach sich ziehen sollte, wird eine Startup-Sequenz eingeleitet, sobald das Smart an eine passende Stromversorgung angeschlossen wurde.

Diese Startup-Sequenz kümmert sich um das Erkennen sowie das Koppeln der angeschlossenen Devices. Sie läuft für jedes unterstützte Gerät gleich ab. Da das Smart seinen Strom von dem Device erhält, wird angenommen, dass das Device seinen Zustand für einen längeren Zeitraum hält. Das Smart muss daher bei jedem Neustart die aktuellen Zustände der Devices und des Gesamtsystems erfassen. Die Startup-Sequenz ist in Abbildung 5.8 nachzuvollziehen.

Nach dem Hochfahren sendet das Smart eine Anfrage bezüglich der ‚Device-ID‘ an alle verfügbaren Ports. Diese Anfrage hat zwei Aufgaben. Sie dient als eine Art Ping, um zu erfassen, auf welchem Port ein Device angeschlossen ist und als eine Art der Identifikation, mit der ein Device an einen Port gekoppelt werden kann. Jedes Device, welches

diese Anfrage empfängt, antwortet mit seiner *Device-ID*.

Die Anfrage nach der ID wird zuerst vorgenommen, um das Device aufseiten des Smart zu registrieren und an diesen Port zu binden. Jede weitere Nachricht auf diesem Port kann direkt einem Device zugeordnet werden.

Durch die ‚Device-ID‘ wird ein eindeutiger Bezeichner für die zu erzeugende Kopie der Device-Daten gegeben. Der Bezeichner ermöglicht die Erschaffung einer eindeutigen Abbildung zwischen der ID und der physischen Schnittstelle.

Das Smart nutzt diese ID für die eigene Organisation und die Persistenz der Device-Zustände. Da sie nur für eine interne Adressierung genutzt wird, kann sie von dem Device generiert werden.

Es ist sinnvoll, dass die ID an die Hardware des Device gebunden ist und im Kontext der verschiedenen Devices eindeutig ist. Eine Zertifizierung und damit zentrale Zuweisung der ID für unterstützte Geräte ist hier denkbar.

Nachdem eine erfolgreiche Kopplung mithilfe der *Device-ID* durchgeführt wurde, sendet das Smart eine Anfrage bezüglich des *Device-Type* an das entsprechende Device.

Der ‚Device-Type‘ bestimmt die Eigenschaften und Funktionen eines Device. Der von dem Device gesendete Typ muss von dem Protokoll und dem Smart unterstützt werden. Anhand des Typs weiß das Smart mit welcher Art von Gerät es kommuniziert. In der Abbildung 5.8 ist die Startup-Sequenz eines Gerätes vom Typ ‚Thermostat‘ zu sehen. Wurde das Device aufseiten des Smart registriert, wird der aktuelle Zustand des Device abgefragt. Wie in der Abbildung zu sehen ist, werden dazu nacheinander die Werte aller Attribute angefragt. Welche Attribute und Funktionen von einem Device bereitgestellt werden müssen, ist abhängig vom Typ des Device und wird in der Protokoll-Spezifikation definiert. Wenn der gesamte Zustand eines Device korrekt übermittelt und von dem Smart empfangen wurde, sind die Abbilder von Smart und Device erfolgreich synchronisiert. Das kann anhand der *State-C checksum*, die mit den Paketen versendet wird, erkannt werden. Falls die Replicas nicht übereinstimmen, muss eine geeignete Fehlerbehandlung vorgenommen werden. Auf diesen Vorgang wird im darauffolgenden Abschnitt detailliert eingegangen.

Die Startup-Sequenz wird in dieser Form durchgeführt, weil es für ein Device in den meisten Fällen nicht möglich ist, zu erkennen, ob ein Smart angeschlossen wurde. Da das Device mit einer kontinuierlichen Stromquelle ausgestattet ist und ohne das Smart bereits aktiv sein kann, ergibt eine Nachricht bei dem Hochfahren des Device keinen Sinn. Es ist möglich, dass das Smart zu diesem Zeitpunkt nicht angeschlossen ist und die Nachricht dementsprechend verpasst. Falls ein Fehlerfall eintritt und das Smart neu gestartet wird, stellt die Startup-Sequenz aufseiten des Smart sicher, dass alle Devices wieder korrekt eingebunden und synchronisiert werden.

Bei der Registrierung der Devices kann es lediglich problematisch werden, wenn ein Device mit einem anderen Device getauscht wird. Das muss jedoch geschehen, bevor ein Timeout auf diesem Port eingreifen kann. Eine Abmeldung des Device wird in diesem Fall nicht eingeleitet.

Es ist daher ratsam, dass ein Nutzer das Smart nach Änderungen an der hardwareseiti-

gen Konfiguration neu startet oder einige Sekunden wartet, bevor er ein neues Device auf diesem Port einbindet. Im Falle einer *1:1* Relation zwischen Smart und Device existiert dieses Problem nicht, da das Smart ohne eine Verbindung mit dem Device über keine Stromquelle verfügt. Ein Neustart bei einer Rekonfiguration ist zudem sinnvoll, weil die Startup-Sequenz nicht durchgeführt wird, wenn das Smart bereits mit einem anderen Device verbunden ist und von diesem Device entsprechend Strom bezieht.

Fehlererkennung

Durch die in dieser Arbeit eingeführte Indirektion ergeben sich weitere Fehlerquellen, mit denen ein klassisches Smart Device nicht oder begrenzt konfrontiert ist. Jeder zusätzliche Übertragungskanal erzeugt Unsicherheiten. Es können Fehler in der Übertragung auftreten. Da sich das aufgetrennte Smart Device in der Funktion nicht signifikant von einem klassischen Smart Device unterscheiden soll, müssen Fehler in der Übertragung erkannt und behoben werden.

Die Mechanismen für die Fehlererkennung der Smart-Home-Frameworks sind hier separat zu betrachten. Da die Kommunikation mit den meisten Smart Devices über kabellose Kanäle stattfindet, ist eine Fehlerbehandlung dort bereits integriert. Jedes Framework nutzt dafür seine eigenen Methoden. Es ist nicht ratsam, den Fehlerfall, der durch die Indirektion entsteht, an das Framework weiterzuleiten.

Eine Weiterleitung könnte beispielsweise realisiert werden, indem das Smart auf die Antwort des Device wartet. Bei einem Timeout oder einem NACK könnte ein Fehler an das Framework weitergegeben werden.

Die Anwendung dieser Methodik würde allerdings bedeuten, dass das Smart viele Zustände von vielen Nachrichten gleichzeitig halten müsste. Das stimmt nicht mit dem asynchronen Paradigma überein, welches in dieser Arbeit verwendet wird.

Zudem ist in diesem Fall keine einheitliche Fehlerbehandlung gewährleistet, da jedes Framework dabei anders verfahren kann. Es kann die Anfrage erneut senden oder einfach den aktuellen Zustand beibehalten. Um dem entgegenzuwirken, wurde sich dafür entschieden, dass die Fehlererkennung sowie -behandlung für das Framework verdeckt abläuft. Maßnahmen, die aufgrund von Fehlern in der Übertragung zwischen Framework und Smart entstehen, können in dieser Arbeit nicht beeinflusst werden und bleiben bestehen.

Empfängt das Smart eine Anweisung von dem Framework, gibt es eine positive Rückmeldung an das Framework zurück. Wenn anschließend ein Fehler auftritt, wird dieser von dem Smart behandelt. Dadurch kann sichergestellt werden, dass weder ein Nutzer noch das Framework mit neuen Fehlern konfrontiert wird.

Ein Fehlerfall in der Kommunikation zwischen Smart und Device kann unterschiedliche Formen annehmen. Durch Rauschen auf den Kanälen, Störungen aus der Umgebung und fehlerhafter Hardware können Übertragungsfehler auftreten.

Es kann zu fehlerhaften Bytes innerhalb der gesendeten Pakete kommen oder Pakete

können vollständig korrumptieren. Zudem ist es auf beiden Seiten möglich, dass Fehler in der Datenverarbeitung auftreten.

Im Kontext dieser Arbeit wird eine empfangene Nachricht entweder als korrekt oder fehlerhaft angesehen. Pakete mit fehlerhaften Bits oder ganzen Bit-Sequenzen sind fehlerhaft. Sind die Daten innerhalb der Pakete lesbar und die Fehler existieren nur in den Headern oder in den Steuerzeichen, werden sie ebenfalls als fehlerhaft behandelt. Welche Teile einer fehlerhaften Nachricht korrekt sind, ist für den Empfänger nicht festzustellen. Aus diesem Grund wird eine fehlerhafte Nachricht nicht weiterverarbeitet und entsprechend abgelehnt. Sie führt zu einem NACK auf dem Kanal, auf dem sie empfangen wurde. Ein NACK wird gesendet, sobald empfangene Daten fehlerhaft sind oder von dem Device aus anderen Gründen nicht verarbeitet werden können. Dieser Fall tritt auch ein, falls ein Befehl von dem Device nicht unterstützt bzw. implementiert ist.

Zur Erkennung von fehlerhaften Paketen wird die Paket-interne Checksum verwendet. Sie können direkt aussortiert werden.

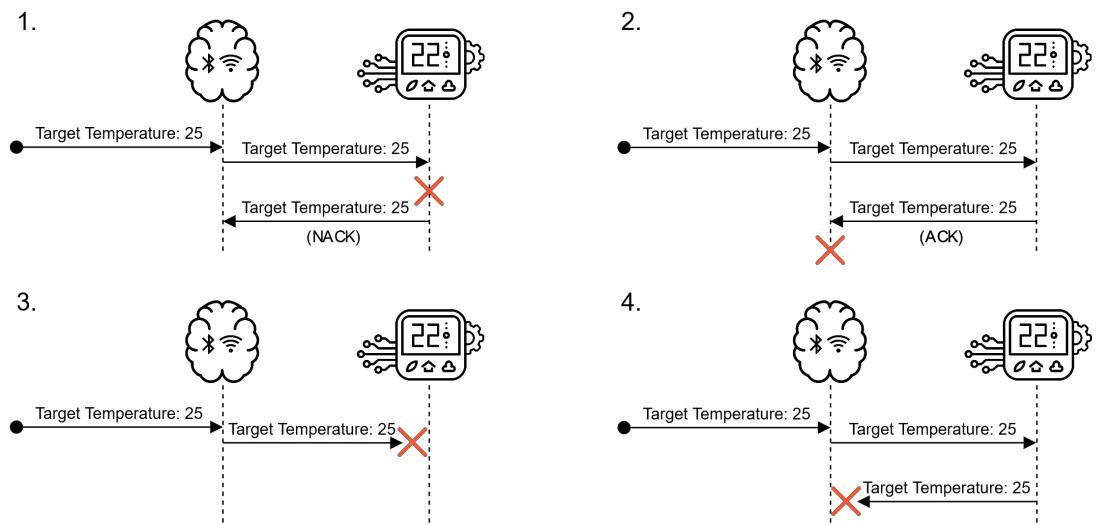


Abbildung 5.9: Fehlerfälle

Welche fehlerhaften Szenarien in der Übertragung auftreten können wird in Abbildung 5.9 verdeutlicht. Das erste Bild stellt dabei alle Fälle dar, bei denen eine fehlerhafte Übertragung durch das Device erkannt wird und durch ein NACK angezeigt wird. In Bild 2 wird der Fall dargestellt, bei dem die Antwort des Device fehlerhaft am Smart ankommt. Die Fehlerfälle in den Bildern 3 und 4 beschreiben das Verschwinden oder Über hören einer kompletten Nachricht in einem bestimmten Schritt der Übertragung. Das Verschwinden eines NACK ist gleichwertig zu betrachten mit dem Verschwinden der Anfrage, da in diesem Fall weder eine Funktion ausgeführt noch eine Antwort gesendet

wird. Bei der Benachrichtigung eines weiteren Smarts sind lediglich die Fälle interessant, bei denen eine Zustandsänderung auf dem Device erfolgt ist (2 & 4).

In allen anderen Fällen muss dieser Smart keine Fehlerbehandlung durchführen, da die Anfrage nicht aus seiner Instanz des Frameworks gekommen ist.

Für das sendende Smart sind alle Fälle relevant, da es dem Framework bereits eine positive Rückmeldung bezüglich der Anfrage gegeben hat.

Zusammenfassend ergeben sich aus den Beispielen die vier Fehlerfälle:

1. Negative Antwort des Device wird empfangen
2. Fehlerhafte Antwort des Device wird empfangen
3. Anfrage des Smart geht verloren
4. Positive Antwort des Device geht verloren

In den ersten beiden Fällen ist die Erkennung eines Fehlers simpel. Erhält das Smart ein NACK oder eine Nachricht mit einer fehlerhaften Checksum, kann es entsprechend reagieren und eine erneute Fehlerbehandlung einleiten.

Wie in den Fällen 3 und 4 zu sehen ist, besteht allerdings auch die Möglichkeit, dass Nachrichten verloren gehen oder anderweitig korrumptieren. Erhält das Smart eine negative Antwort, kann es davon ausgehen, dass seine Anfrage nicht erfolgreich verarbeitet und umgesetzt wurde. Empfängt es diese Antwort nicht oder nur teilweise, kann es sich nicht sicher sein, ob der Fehler bei der Anfrage oder der Antwort aufgetreten ist (Fehlerfälle 3 & 4). In diesem Fall kann das Smart nicht wissen, in welchem Zustand sich das Device befindet.

Eine Überlegung, um diesen Vorfall zu erkennen, wäre es, eine Zeitmessung aufseiten des Smart durchzuführen. Nach einer bestimmten Zeit würde das Smart keine Antwort mehr erwarten und eine Fehlerbehandlung einleiten können. Damit das einwandfrei funktionieren kann, müssten weitere Anpassungen gemacht werden. Das Smart müsste für jede gesendete Anfrage eine eigene Messung starten. Eine Nachricht müsste beispielsweise durch eine Sequence-Number eindeutig identifizierbar sein. Die Sequence-Number müsstepersistiert werden, um den richtigen Timer zurückzusetzen, falls eine Antwort auf diese Nachricht empfangen wird.

Diese Variante der Fehlererkennung wurde nicht gewählt, da der Mehraufwand für eine solche Umsetzung aufseiten des Smart zu hoch ist. Bei dem genannten Ansatz werden außerdem nur Fehler in Anfragen erkannt, die das Smart gesendet hat. Wird eine Benachrichtigung für eine Zustandsänderung nicht korrekt empfangen, so wird auch das nicht erkannt.

Um möglichst viele Fehlerfälle abzudecken, wurde sich in dieser Arbeit für die Nutzung einer zusätzlichen Checksum entschieden. Die *State-C checksum* enthält die Summe des gesamten Abbildes eines Device.

Sobald einer der Zustände sich geändert hat, ändert sich auch die Checksum. Das Device sendet sie in jedem Paket mit. Das Smart kann diese Checksum mit der Checksum seines eigenen lokalen Abbildes vergleichen. Sie wird auch in den Benachrichtigungen des

Device gesendet. Daher sind fehlerhafte Zustände für das Smart auch dann erkennbar, wenn von dem Smart zuvor keine Anfrage gesendet wurde. Empfängt ein Smart eine Änderungsbestätigung von einem Device, sollte der State nach der lokalen Durchführung dieser Änderung gleich sein. Das ist möglich, da die Anfrage, auf die sich die Bestätigung bezieht, in der Antwort mitgesendet wird. Weicht die lokal errechnete Checksum von der empfangenen ab, ist anzunehmen, dass ein Fehler in der Übertragung oder der Verarbeitung aufgetreten ist. Wie ein solcher fehlerhafter Nachrichtenaustausch mit korrekter Fehlererkennung aussehen kann, wird in Abbildung 5.10 veranschaulicht.

Die *State-C checksum* ist in erster Linie hilfreich, um festzustellen, ob eine Zustandsänderung auf dem Device erfolgt ist, ohne dass das Smart dies mitbekommen hat. Das bedeutet, sie richtet sich besonders an die Erkennung der vierten Fehlerkategorie.

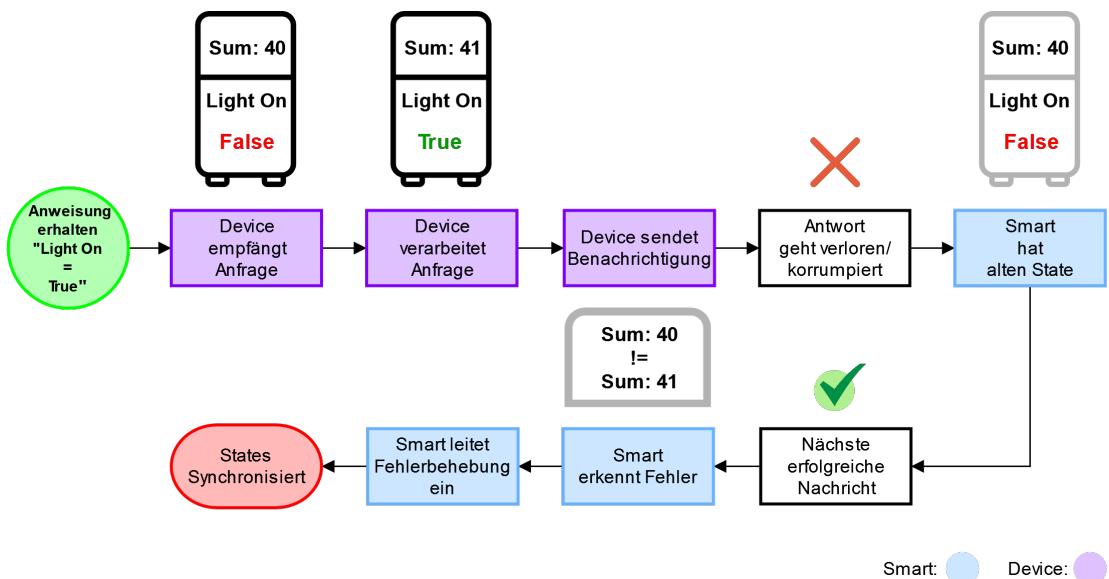


Abbildung 5.10: Erkennung eines fehlerhaften State

Ist die *State-C checksum* sinnvoll implementiert, kann sie allerdings auch bei der Erkennung der ersten und dritten Fehlerkategorie helfen. Im ersten Fehlerfall ist bei einem NACK keine Fehlerbehebung nötig, falls die Anfrage ursprünglich von einem anderen Smart gekommen ist. Das Smart kann die Antwort ignorieren, da die Anfrage zu keiner Zustandsänderung geführt hat. Ob das Smart die Anfrage gesendet hat, kann ebenfalls durch die *State-C checksum* erkannt werden.

Damit das funktioniert, muss das Smart eine empfangene Änderung von dem Framework direkt auf seinem lokalen Abbild ändern, noch bevor es die Anfrage an das Device weiterleitet.

Wird die Änderung von dem Device nicht durchgeführt, kann das durch die abweichende *State-C checksum* erkannt werden. Die beiden unterschiedlichen Fälle sind in Abbildung 5.11 dargestellt. Das Smart, welches die Anfrage nicht weitergeleitet hat, sollte nach einem empfangenen NACK dieselbe *State-C checksum* besitzen wie das Device, da sich sein Zustand nicht verändert hat.

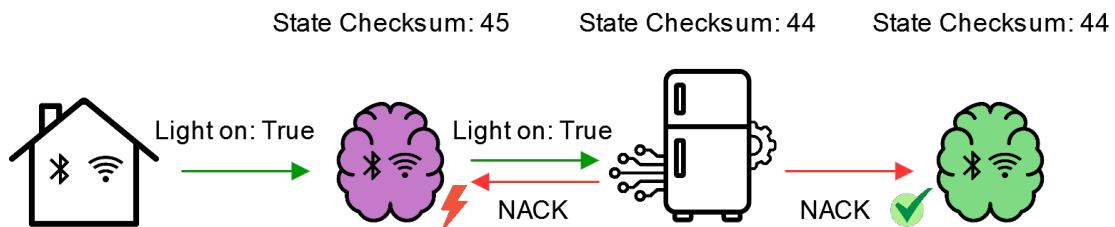


Abbildung 5.11: Erkennung: Neuaufbau notwendig?

Dieser Vorgang ist auch in der dritten Fehlerkategorie zu beobachten. Hat das Smart die Anfrage des Frameworks weitergeleitet, kann der Fehler bei der nächsten korrekt empfangenen Nachricht anhand der Checksum erkannt werden. Alle anderen Smarts werden den Verlust der Nachricht nicht bemerken.

Bei der zweiten Fehlerkategorie kann die *State-C checksum* nicht helfen, weil die Integrität der Daten nicht sichergestellt werden kann.

Diese Art der Fehlererkennung beruht auf der Annahme, dass zwischen Smart und Device kontinuierlich Nachrichten gesendet werden. Geht eine Antwort von einem Device verloren und es findet kein weiterer Nachrichtenaustausch statt, wird auch die Checksum bei der Erkennung nicht helfen.

Um dieses Problem anzugehen, hilft es ein weiteres Problem zu betrachten, welches sich aus der Kommunikation der beiden Geräte ergibt. Es ergibt sich aus der Frage, wann ein Device als verbunden gilt und wie dieser Zustand erkannt wird. Findet eine längere Zeit kein Nachrichtenaustausch statt, so ist es für das Smart unmöglich zu wissen, ob es noch mit dem Device verbunden ist oder nicht. Die Anfragen an ein Smart Device kommen von einem Anwender und liegen meist in einer niedrigen Frequenz vor. Es kommt häufig vor, dass ein Smart Device für mehrere Stunden keine Anweisungen empfängt.

Bezieht ein Smart seinen Strom aus einer anderen Quelle, so kann es sich nicht sicher sein, ob es noch mit dem Device verbunden ist. Daher ist hier die Einführung von Keep-Alive-Messages sinnvoll. Sie werden in einem regelmäßigen Intervall gesendet und geben Aufschluss darüber, ob eine fehlerfreie Verbindung mit dem Device vorliegt. Jede der Nachrichten sollte von den Devices beantwortet werden. Ist eine Verbindung fehlerhaft und es kann kein Datenaustausch stattfinden, kann dies durch eine Zeitmessung erkannt werden. Sie wird gestoppt, sobald eine Nachricht von dem Device empfangen wurde. Überschreitet diese Zeitmessung einen bestimmten Schwellenwert wie beispielsweise 10

Sekunden, kann die Verbindung als fehlerhaft angesehen und geschlossen werden. Dieser kontinuierliche Datenaustausch führt ebenfalls dazu, dass die zuvor genannte Fehlererkennung sichergestellt werden kann. Die *State-Checksums* können in den Keep-Alive-Messages mitgesendet werden.

Der Austausch der Keep-Alive-Messages beginnt nach der erfolgreichen Registrierung eines Device. Dadurch wird gewährleistet, dass die Zustände der beiden Abbilder (Smart und Device) in regelmäßigen Abständen synchronisiert werden. Wird eine Abweichung der Checksums festgestellt, kann eine Fehlerbehandlung eingeleitet und die Zustände erneut synchronisiert werden.

Fehlerbehebung

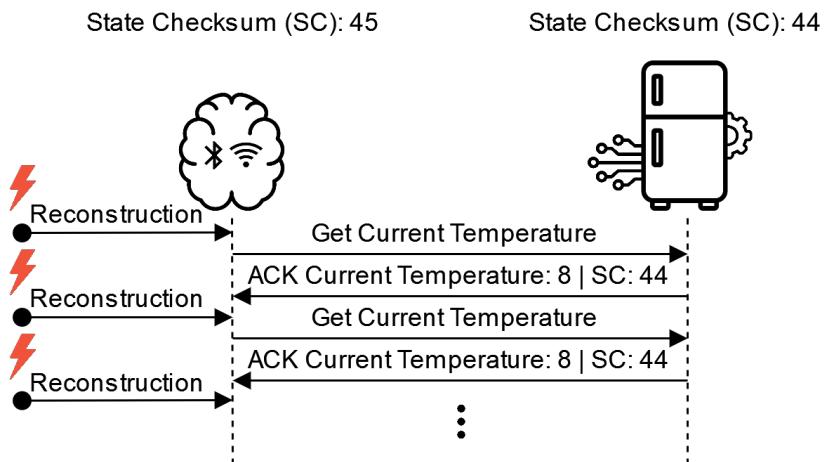


Abbildung 5.12: Neuaufbau ohne State-Invalidation

Wird ein fehlerhafter Zustand von dem Smart erkannt, wird der aktuell abgebildete lokale Zustand invalidiert. Anschließend wird das Abbild neu aufgebaut und der korrekte Zustand wiederhergestellt. Der Neuaufbau wird ähnlich wie der initiale Aufbau bei der Startup-Sequenz durchgeführt. Da anhand der *State-Checksum* nicht erkennbar ist, welcher Parameter über einen fehlerhaften Wert verfügt, wird jeder Parameter des Device einzeln neu angefragt.

Der Zustand auf dem Device wird dadurch nicht verändert. Der Zustand wird so lange als Invalid betrachtet, bis die *State-Checksum* des lokalen Abbildes erneut mit der empfangenen Checksum des Device übereinstimmt. Stimmt die empfangene Checksum wieder mit der lokalen Checksum auf dem Smart überein, ist der Zustand erfolgreich wiederhergestellt und das Smart kann seine Funktion im Normalzustand weiter durchführen.

Wurden alle Nachrichten empfangen und die Checksums stimmen nicht überein, wird

davon ausgegangen, dass nicht alle Parameter korrekt übertragen wurden. Um sicherzugehen, dass die Fehlerbehandlung, falls möglich, erfolgreich durchgeführt werden kann, versucht ein Retry-Mechanismus die Wiederherstellung des Zustandes in regelmäßigen Abständen. Ist eine Synchronisierung aufgrund einer stark verrauschten oder fehlerhaften Verbindung nicht möglich, wird die Verbindung getrennt.

Die Invalidierung des Zustandes ist hierbei ein wichtiger Schritt. Sie dient der Speicherung des fehlerhaften Zustandes für darauffolgende Nachrichten und führt zu einer anderen Verarbeitungsweise der Nachrichten aufseiten des Smart. Wird diese Invalidierung nicht vorgenommen, kann ein fehlerhafter Zustand zu einer starken Belastung des Kommunikationsmediums führen. In Abbildung 5.12 ist eine Fehlerbehandlung dargestellt, die ohne eine vorherige Invalidierung stattfindet. Ein fehlerhafter Zustand löst einen Neuaufbau des Abbildes aus. Dabei wird jeder Parameter des Device angefragt. Empfängt das Smart die erste Antwort mit dem ersten Parameter des Device, wird die empfangene *State-Checksum* mit einer großen Wahrscheinlichkeit noch nicht mit der des Smart übereinstimmen. Anschließend wird das Smart erneut einen fehlerhaften Zustand erkennen und das Abbild neu anfragen.

Diese Prozedur wird so lange durchgeführt, bis die fehlerhaften Parameter aufseiten des Smart übernommen wurden. Dabei kann es aufgrund der großen Anzahl der Nachrichten vorkommen, dass weder das Smart noch das Device mit der Verarbeitung vorankommen. Durch die Invalidierung kann sichergestellt werden, dass das Smart bei den weiteren empfangenen Nachrichten keine erneute Fehlerbehandlung einleitet.

Es wurde sich dafür entschieden, keine Form der Error-Correction in die Syntax der Pakete einzubauen. Durch diese könnten einige kleine Fehler korrigiert werden, die während der Übertragung auftreten. Error-Correction-Codes (ECCs) können genutzt werden, wenn eine Resend-Semantik in einer Kommunikation sehr aufwendig, kostenintensiv oder nicht möglich ist. Sie ermöglichen eine Korrektur aufseiten des Empfängers. Dadurch entfällt die Notwendigkeit, dass der Sender die Daten erneut senden muss. Die Anzahl an Bit-Fehlern, die mit ECCs korrigiert werden können ist jedoch begrenzt. Tritt ein Fehler auf, bei dem viele Bits in derselben Nachricht fehlerhaft sind, ist es möglich, dass dieser Fehler für den Empfänger nicht korrigierbar ist.

Die in dieser Arbeit gesendeten Nachrichten werden innerhalb der Kommunikationsprotokolle in einzelnen kleineren Paketen gesendet. Es ist nicht auszuschließen, dass an mehreren Punkten Fehler auftreten. In diesem Fall ist ein Resend oder, wie in dieser Arbeit, ein Neuaufbau des Abbildes unvermeidbar. Eine Erkennung eines fehlerhaften State ist daher auch bei der Verwendung von ECCs notwendig, um die korrekte Synchronisierung der Abbilder zu gewährleisten. Sie könnten zusätzlich eingesetzt werden, um kleine Fehler direkt und lokal zu beheben.

In dieser Arbeit wurde sich gegen die zusätzliche Nutzung von ECCs entschieden, da sie mit einem hohen Overhead einhergehen.

Ein Resend von einzelnen Nachrichten nach einem Timeout oder nach Empfang eines NACK wird ebenfalls nicht vorgenommen. Diese Mechanik ist in der asynchronen Umsetzung in dieser Arbeit nicht sinnvoll. Ähnlich wie bei der Erkennung von Fehlerfällen mithilfe einer Zeitmessung ist der zusätzliche Aufwand hier sehr hoch. Einem Nutzer gegenüber kann es zudem unintuitiv sein, wenn nach einer längeren Zeit die Anfrage neu gesendet wird. Nach einer bestimmten Zeit könnte ein Nutzer davon ausgehen, dass die Anfrage fehlgeschlagen ist und es erneut versuchen.

Abmeldung

Wird ein Device von dem Smart getrennt oder ist für eine längere Zeit keine Kommunikation mit diesem Device möglich, wird das Gerät abgemeldet. Eine Abmeldung bedeutet, dass das Gerät aus dem Kontext des Smart gelöscht wird. Um Ressourcen zu sparen wird das Abbild, also das logische Objekt, aufseiten des Smart ebenfalls verworfen. Bei einer erneuten Registrierung wird eine neue Kopie des Device erzeugt und ihr Zustand neu angefragt. Damit eine kurzzeitige Fehlerquelle ausgeschlossen werden kann, wird nach einer Abmeldung eines Device, auf demselben Port ein erneuter Verbindungsversuch initiiert. So kann ein Gerät auf derselben Schnittstelle gegebenenfalls wieder eingebunden werden.

Kapitel 6

Implementierung

Die in dem vorherigen Kapitel beschriebenen Design-Entscheidungen sind in die Entwicklung eines Prototyps eingeflossen. Dieser Prototyp exploriert die Auf trennung eines Smart Device in mehrere Smarts und mehrere Devices.

In diesem Kapitel wird die Implementierung des Prototyps detailliert beschrieben. Ein großer Fokus liegt dabei auf notwendigen Überlegungen, Schwierigkeiten und bestimmten Entscheidungen, die sich aus dem Entwicklungsprozess ergeben haben.

Der erste Abschnitt dieses Kapitels beschäftigt sich mit der hardware- und softwareseitigen Konfiguration der gesamten Anwendung. Im zweiten Abschnitt wird die Implementierung der Anwendung für den normalen Programmablauf beschrieben. Der letzte Abschnitt behandelt die Umsetzung der Fehlerbehandlung. Sie nimmt in dieser Arbeit einen wichtigen Teil der Anwendung ein, da durch die Trennung einige zusätzliche Fehlerfälle entstehen, die nicht von dem Smart-Home-System erkannt werden.

Die Implementierung wurde bewusst gewählt und wird von anderen Möglichkeiten abgegrenzt.

6.1 Konfiguration

Wie anhand der bekannten Arbeiten zu sehen ist, ist die Wahl der Komponenten und der Frameworks entscheidend für die gesamte Arbeit. Deutlich wird das bei der Betrachtung der unterschiedlichen Smart-Home-Frameworks und ihrer Zusammensetzung. [5]

Das Smart ist der Kern des in dieser Arbeit beschriebenen Vorhabens. Es muss sowohl die Frameworks unterstützen als auch mehrere Devices einbinden. Der Verwaltungs- sowie Rechenaufwand des Smart ist deutlich höher als der eines Device. Es ist also sehr wichtig, diese Komponente so auszuwählen, dass alle nötigen Frameworks implementiert werden können und eine ausreichende Rechenkapazität für die Verwaltung mehrerer Devices vorhanden ist.

6.1.1 Komponenten

Die Aufgabe des Device wird in dieser Umsetzung von einem Arduino UNO R3 übernommen. Der Arduino ist mit einem ATmega328P Mikrocontroller ausgestattet und simuliert in diesem Fall das statisch integrierte Device.

Die Funktionen des Device werden abstrahiert. Dazu wurde das Device mit einer zusätzlichen RGB-LED ausgestattet. Sie ermöglicht die visuelle Abstraktion der Device-Funktionalitäten und bietet einem Nutzer die Möglichkeit, Rückmeldungen über getätigte Eingaben zu erhalten.

Aufgrund der niedrigen Anforderungen an das Device ist der 16MHz Mikrocontroller des Arduino für die Funktionen des Device ausreichend. Die wesentliche Aufgabe des Device ist die Kommunikation mit dem Smart und das Ausführen einfacher Anweisungen. Für die Kommunikation ist der Arduino bereits mit mehreren seriellen Schnittstellen ausgestattet.

Das Smart wird in dieser Umsetzung durch eine Node MCU ESP32 repräsentiert. Der ESP32 Mikrocontroller ist mit einem leistungsstarken Dual-Core-Prozessor (240 Mhz) ausgestattet. Er wurde von Espressif explizit für den Einsatz im IoT entwickelt [20]. Aus diesem Grund bringt er notwendige Technologien wie Bluetooth und Wi-Fi bereits mit.

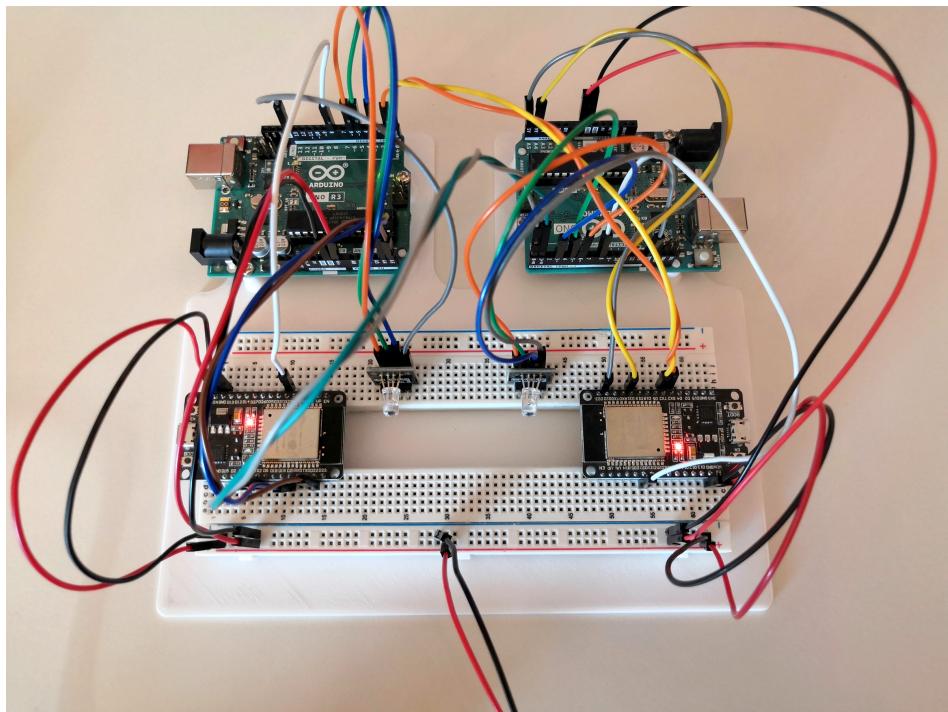


Abbildung 6.1: Aufbau des Test-Setups

6.1.2 Aufbau

Der Aufbau des Test-Setups ist in Abbildung 6.1 zu sehen.

Das Setup umfasst zwei ESP32-Module, zwei Arduino UNOs, zwei RGB-LEDs sowie notwendige Jumper-Kabel. Die ESP32-Module repräsentieren jeweils die Smarts. Die Arduinos simulieren die Devices.

Durch diese Konfiguration ist sichergestellt, dass mögliche Komplikationen testbar sind und unterschiedliche Praxisanwendungen untersucht werden können. Ein Arduino kann verschiedene Arten von Devices simulieren und implementiert daher unterschiedliche Funktionen.

Wie zuvor bereits erwähnt, ist jeder Arduino zusätzlich mit einer RGB-LED verbunden, um unterschiedliche Anwendungen repräsentieren zu können. Die LEDs sind auf externen Platinen angebracht und jeweils mit 1 kOhm Widerständen ausgestattet.

Verbunden sind alle Komponenten über ein Breadboard. Damit das Test-Setup portabel und kompakt gehalten wird, sind die Komponenten zusätzlich auf einer 3D-gedruckten Arbeitsplatte montiert. Sowie die LEDs als auch die ESP32-Module sind dabei direkt in das Breadboard eingelassen. Die Arduinos sind mit Schrauben und Nieten fest auf der Platte fixiert.

Alle Komponenten können Strom von einem konfigurierbaren Netzteil über eine der Leiterbahnen des Breadboards beziehen. Alternativ besteht die Möglichkeit, jede Komponente einzeln mithilfe von USB-Kabeln und Netzteilen über den integrierten Port mit Strom zu versorgen.

Für die Verkabelung werden Jumper-Kabel eingesetzt. Durch die Jumper-Kabel ist eine Änderung der Konfiguration und Verkabelung zu Testzwecken schnell umsetzbar. Beide Smarts sind jeweils mit beiden Devices über unterschiedliche serielle Schnittstellen verbunden. Dadurch können mehrere Technologien getestet werden.

Verwendet werden die seriellen Schnittstellen UART und I²C. Es wurden zwei unterschiedliche Technologien gewählt, da nicht jedes verwendete Gerät über mehrere Schnittstellen derselben Art verfügt. Daraus entsteht die Möglichkeit, verschiedene Kommunikationsparadigmen miteinander zu vergleichen. Da sich die symmetrische und asynchrone Kommunikation mithilfe von UART stark von der Master-Slave-Semantik von I²C unterscheidet, ist ein interessanter Ansatzpunkt diese beiden miteinander zu vergleichen. Es gilt herauszufinden, welche Vorkehrungen getroffen werden müssen, damit beide Technologien im Anwendungskontext erfolgreich genutzt werden können. Des Weiteren ist es interessant, welche Vor- und Nachteile die Nutzung für diesen Anwendungszweck mit sich bringt.

Der genaue Aufbau und die Verbindungen können in Abbildung 6.2 eingesehen werden. In dieser Abbildung ist das Gesamtsystem detailliert und anschaulich dargestellt. Sie wurde mithilfe des Open-Source Tool Fritzing¹ erstellt. Die parallel verlaufenden grünen und braunen Linien zeigen in dieser Darstellung die Verbindungen für die UART-

¹<https://fritzing.org/>

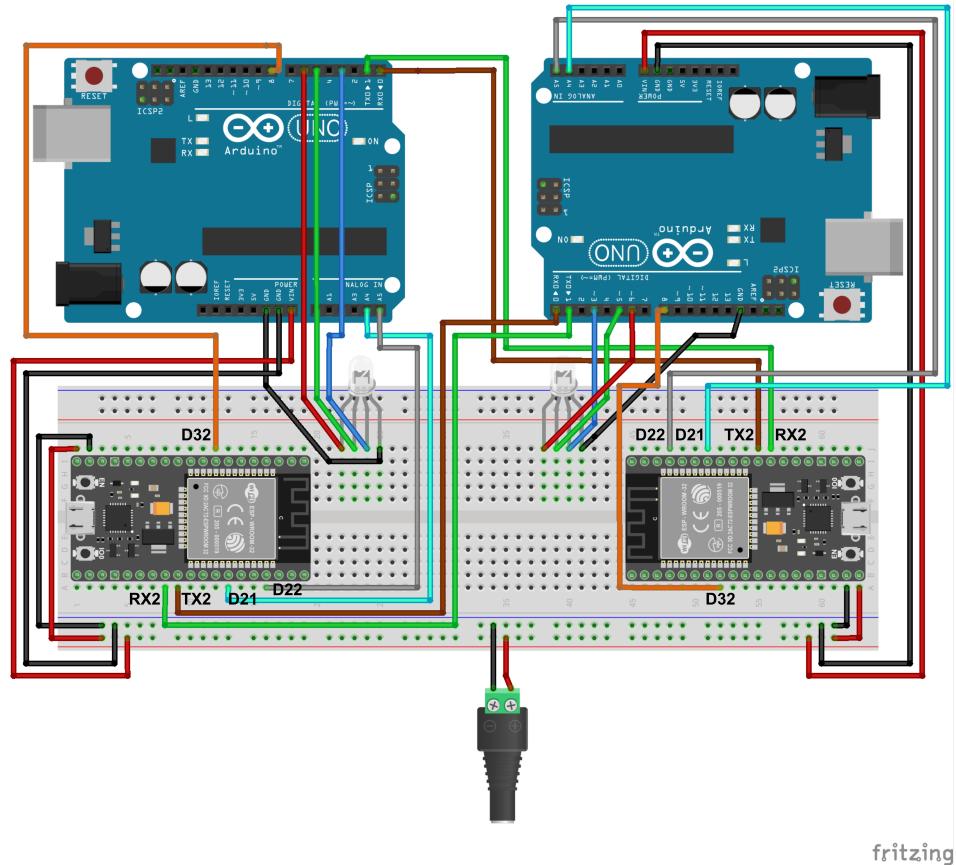


Abbildung 6.2: Darstellung des Setups mit Fritzing¹

Schnittstelle. Die grauen und türkisen Linien zeigen die Verbindungen für die I²C Schnittstelle.

In dem verwendeten Setup werden die Smarts jeweils mit derselben Software ausgestattet. Sie repräsentieren die gleichzeitige Nutzung von zwei Smarts, die sich im selben Netzwerk befinden und nicht eigenständig miteinander kommunizieren können. Die Synchronisierung geschieht über die Devices. Sie kommunizieren mit den Smart-Home-Frameworks über kabellose Übertragungskanäle. Die Devices unterscheiden sich in ihrer Funktionsweise und der Erkennung im Kontext der IoT-Frameworks.

6.1.3 Frameworks und Sprachen

Als IoT-Framework wurde in dieser Umsetzung das Apple HomeKit gewählt. Ein großer Vorteil gegenüber anderen Frameworks ist die einfache und native Nutzung der HomeKit Dienste durch die Integration des HAP-Standards [5].

Die nicht-kommerzielle Version des HAP [7] ist mit einem Developer Apple Account kostenfrei einsehbar.

Die große Anzahl der von dem HAP unterstützten Geräte ermöglicht es, diese Geräte mit einem niedrigen Konfigurationsaufwand einzubinden. Das Protokoll ist gut dokumentiert und beschreibt für jedes Gerät detailliert welche Parameter von einem Gerät angefragt und geändert werden können. Dabei wird kein separater Server benötigt, der sich um die Kommunikation mit dem Framework kümmert. Ein Nachteil des HomeKits ist dabei allerdings, dass für die Steuerung von außerhalb des eigenen Heimnetzes weitere Geräte notwendig sind. Das betrifft jedoch jedes Smart Device, welches in das HomeKit eingebunden wird und resultiert nicht aus der in dieser Arbeit entwickelten Trennung. Ein weiterer Vorteil des HomeKits ist die direkte Unterstützung von Bridge Devices.

Bridge Devices verwalten und integrieren mehrere unterschiedliche Geräte. Sie vermitteln zwischen den Devices und dem IoT-Framework. Das HomeKit bietet den Anwendern und Entwicklern die Möglichkeit, diese Bridge Devices separat einzubinden und zu nutzen. Dieser Aspekt wurde bereits bei der Konzeption des Homebridge-Projektes [13] genutzt. Das erleichtert das Vorhaben dieser Arbeit deutlich und hilft dabei die gewünschte Trennung durchzuführen. Anstatt dass ein Smart die unterschiedlichen Devices für das Framework lediglich simuliert, ist sich das Framework der Existenz der Bridge bewusst. Die Bridge wird dabei als ein explizites Gerät betrachtet und kann in der Home App eingesehen werden. Für einen Nutzer hat das den Vorteil, dass ihm Informationen wie die Versionsnummer oder die ID des Smart zur Verfügung stehen.

Ein sehr wichtiger Faktor für die Entwicklung und die Anwendung ist jedoch, dass die Bridge sich um die Einbindung der unterschiedlichen Geräte kümmern kann und nicht jedes Gerät separat in der Home App registriert werden muss. Existiert eine solche Bridge nicht oder werden andere Frameworks genutzt, so muss im schlimmsten Fall jedes neu angeschlossene Gerät einzeln in der jeweiligen App registriert werden. Dieser Vorgang kann teilweise aufwendig und störanfällig sein. Durch die Bridge fällt er für jedes an die Bridge angeschlossene Gerät weg.

Die Einbindung des HomeKit-Frameworks wurde so implementiert, dass zu einem späteren Zeitpunkt weitere Frameworks mit geringem Aufwand ergänzt werden können.

Für die Entwicklung der Anwendung wurde sowohl aufseiten des Smart als auch aufseiten des Device die Sprache C++ gewählt. Sie bietet den Vorteil, dass sie von den Gerät-Herstellern unterstützt wird und das Hochladen von Programmcode mithilfe der GNU-Toolchain ohne viel Aufwand ermöglicht. Dabei wurde sich explizit für die Sprache C++ gegenüber C entschieden, da sie über umfangreichere Standard-Libraries verfügt und für diese Arbeit sinnvolle Paradigmen wie Objektorientierung mit sich bringt.

Die Entwicklung des Device wurde mit der Arduino IDE durchgeführt, welche spezielle Regeln für die Code Strukturierung integriert und C sowie C++ unterstützt [26].

Bei den ESP32-Modulen besteht ebenfalls die Möglichkeit, die Arduino IDE für den Programmablauf zu nutzen. Dazu muss lediglich ein weiteres Paket in die Boardverwaltung der IDE eingefügt werden. Aufbauend auf dieser IDE wurden bereits Projekte geschaffen, die die Entwicklung von HomeKit Accessories auf ESP Mikrocontrollern ermöglichen.

„Arduino-Homekit“² von Mixiaoxiao ist ein Projekt, welches die Arduino IDE nutzt, um das HAP auf dem ESP32 und dem ESP8266 zu integrieren.

Die Entwicklung für den ESP32 wurde jedoch eingestellt, nachdem der Hersteller des Mikrocontrollers Espressif mit dem „ESP Apple HomeKit ADK“³ und dem „ESP HomeKit SDK“⁴ die Unterstützung des HAP und der HomeKit API von sich aus bereitgestellt hat. Anstatt die Arduino IDE zu nutzen, setzen beide Frameworks auf der hauseigenen Espressif ESP-IDF auf. Das ESP HomeKit SDK bietet bereits eine fertige Implementierung der HomeKit-Specification, die für die Entwicklung eines Smart Device genutzt werden kann. Sie beinhaltet die Mechanismen zum Wi-Fi-Provisioning für die Verbindung mit dem lokalen Heimnetzwerk sowie vorgefertigte Funktionen zum Definieren und Beschreiben von Accessories (Smart Objects im Kontext des HomeKits). Dabei bietet das Framework dem Entwickler viele Möglichkeiten, diese Accessories zu konfigurieren. Da das zugrundeliegende ESP-IDF zudem auf dem für eingebettete Systeme entwickelte Betriebssystem FreeRTOS aufbaut, wurde sich in dieser Arbeit für die Nutzung des ESP HomeKit SDK entschieden. Das FreeRTOS hat dabei den zusätzlichen Vorteil, dass bereits ein Scheduler und Mechanismen wie Multithreading implementiert sind und in der Anwendung genutzt werden können.

6.2 Ausführung

Jedes Smart wird als eine eigenständige Bridge in das HomeKit integriert und in der Home App angemeldet. Die Ausführungsreihenfolge des Anmeldeprozesses und der Registrierung der Devices ist in Abbildung 6.3 veranschaulicht.

Wie zuvor bereits erläutert, muss in der App lediglich die Bridge registriert und hinzugefügt werden. Der Rest kann durch das Smart erfolgen. Nach der Anmeldung der Bridge kann der im vorherigen Kapitel beschriebene Ablauf zur Erkennung der verbundenen Devices eingeleitet werden. Anschließend ist der korrekte Ist-Zustand hergestellt und die Funktionen des Smart Device können genutzt werden.

Welche Vorkehrungen getroffen wurden und was bei der Umsetzung beachtet wurde, wird im Folgenden ausführlich beschrieben.

Zur Umsetzung der Trennung wurden zwei verschiedene Devices mit unterschiedlichen Funktionalitäten gewählt. Auf einem Device werden mithilfe der RGB-LED die Aufgaben eines Lichtes übernommen. Das andere Device simuliert einen funktionstüchtigen Thermostat. Passende Repräsentationen der Zustände dieses Thermostats werden dabei ebenfalls mithilfe der LED dargestellt.

²<https://github.com/Mixiaoxiao/Arduino-HomeKit-ESP32>

³<https://github.com/espressif/esp-apple-homekit-adk>

⁴<https://github.com/espressif/esp-homekit-sdk>

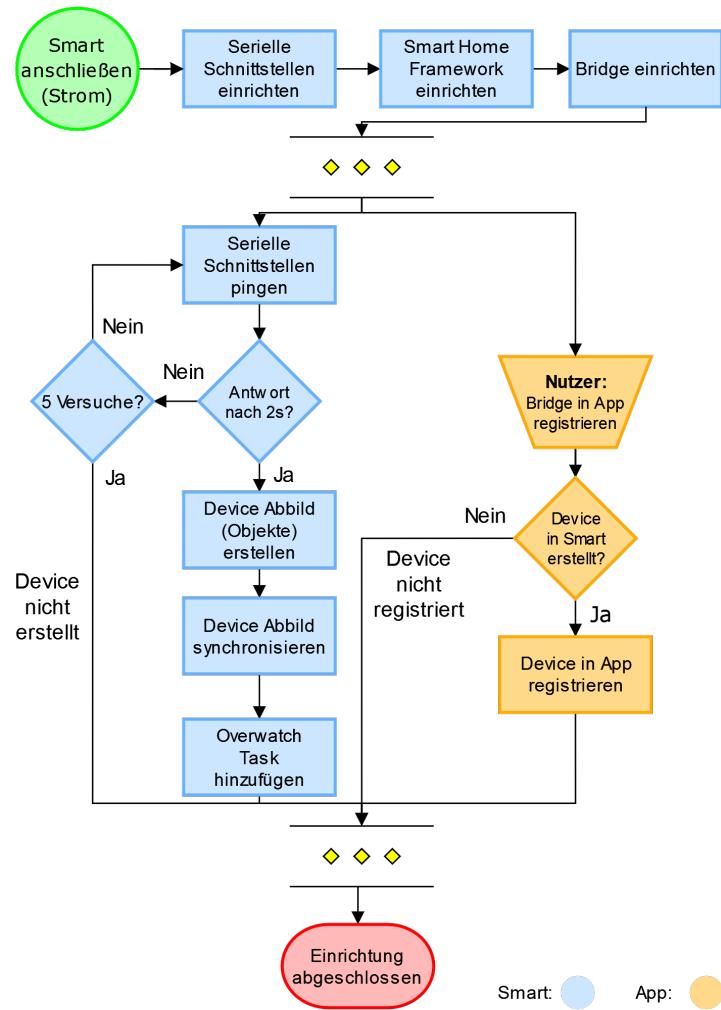


Abbildung 6.3: Einrichtungs- und Anmeldeprozess des Smart

6.2.1 Smart

Das Smart besteht aus vier Modulen *Main (Smart)*, *Smart Home Configuration*, *Device Factory* und dem *Serial Handler*. Ein vereinfachter Aufbau der Struktur dieses Systems ist in Abbildung 6.4 in Form eines Component-Diagramms dargestellt. Der Kern des Systems ist die *Main*. Die meisten Verwaltungsaufgaben werden in diesem Modul erledigt. Sie hält den aktuellen Zustand der Devices sowie des Gesamtsystems und verbindet die einzelnen Komponenten, um die Aufgaben des Smart korrekt ausführen zu können. Das Starten von Threads und Initiieren von Handlungsabläufen wird ebenfalls zentral in der *Main* vorgenommen. In der *Main* befinden sich die Callback-Funktionen für die

Benachrichtigungen der IoT-Frameworks, die zum Anstoßen von Zustandsänderungen der Devices nötig sind.

Das Modul *Smart Home Configuration* verwaltet die Framework-spezifischen Funktionen und Einrichtungsschritte für die Kommunikation mit den IoT-Frameworks. Sie hält eine Liste aller unterstützten Smart-Home-Frameworks und wird sowohl bei der Initialisierung des Systems als auch zur Anmeldung von neuen Geräten aufseiten der Frameworks eingesetzt.

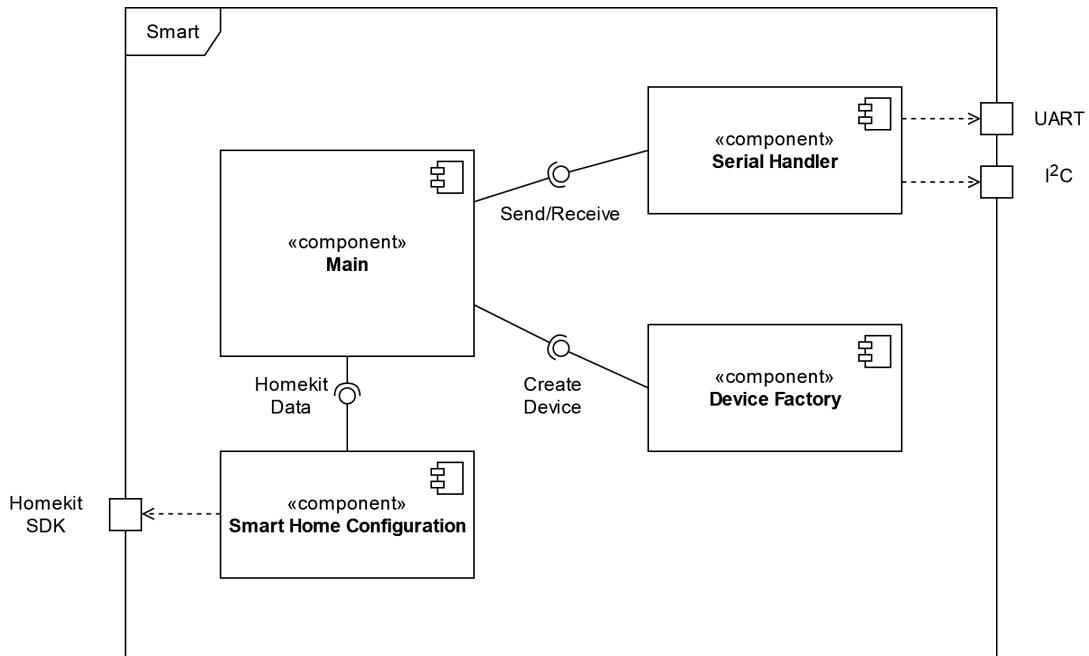


Abbildung 6.4: Simplifizierter Aufbau des Smart

Die *Device Factory* dient der dynamischen Erzeugung von Device-Abbildern zur Laufzeit. Dabei wurde sich dem Factory-Pattern bedient, um verschiedene Devices anhand des *Device-Type* erstellen zu können.

Der *Serial Handler* kümmert sich um die Einrichtung der Übertragungskanäle sowie das Formatieren und Senden der Nachrichten an die Devices. Die genaue Funktionsweise dieser Module im Kontext des Systemablaufs wird in den folgenden Abschnitten näher erläutert.

Property	Value	Accessory Configuration Object
UUID	0000003E-0000-8000-0026BB765291	
Type	public.hap.service.accessory-information	
Required Characteristics	"9.40 Firmware Revision" (page 177) "9.45 Identify" (page 180) "9.58 Manufacturer" (page 187) "9.59 Model" (page 187) "9.62 Name" (page 188) "9.87 Serial Number" (page 201)	<pre>hap_acc_cfg_t cfg = { .name = (char*) "Smart Bridge", .model = (char*) "SmartBridge01", .manufacturer = (char*) "Master's Thesis", .serial_num = (char*) hw_id, .fw_rev = (char*) "1.0.0", .hw_rev = NULL, .pv = (char*) "1.1.0", .cid = HAP_CID_BRIDGE, .identify_routine = accessory_identify, };</pre>
Optional Characteristics	"9.2 Accessory Flags" (page 158) "9.41 Hardware Revision" (page 178)	

Abbildung 6.5: HAP Accessory Specification und Accessory Configuration Object [7]
(Originalbild bearbeitet)

Einrichtung

Bei Start des Smart werden zunächst alle wichtigen Module für die Ausführung initialisiert und konfiguriert. Das beinhaltet sowohl die Konfiguration der Kommunikations-schnittstellen als auch aller unterstützten Smart-Home-Frameworks. Auf die Konfigurationsmöglichkeiten und die Eigenschaften der Schnittstellen wird in dem späteren Unterabschnitt ‚Kommunikation‘ näher eingegangen. Die Konfiguration der Smart-Home-Frameworks wird von dem Modul *Smart Home Configuration* übernommen. Dabei werden zunächst alle wichtigen Eigenschaften und Parameter für jedes Framework gesetzt. Im Falle des HomeKit-Frameworks wird dabei die Bridge konfiguriert. Sie wird mit einer ID versehen und bekommt Eigenschaften wie die Setup-Codes zum Einrichten für die Home App zugewiesen.

Da die Bridge im Kontext des HomeKit-Frameworks als eigenständiges Accessory gilt, sind alle Eigenschaften eines Accessory konfigurierbar. Die Liste an Meta-Informationen, die den Accessories mitgegeben werden können, ist in Abbildung 6.5 auf der linken Seite zu sehen. Auf der rechten Seite der Abbildung ist das Konfigurationsobjekt dargestellt, welches von der HomeKit SDK zur Verfügung gestellt wird. Aus diesem Konfigurations-objekt wird anschließend das Accessory Objekt erzeugt und der lokalen HAP-Datenbank des Frameworks hinzugefügt. Um jedes Smart eindeutig identifizieren zu können, ist in diesem Objekt die Angabe einer Seriennummer notwendig. Das Feld `identify_routine` im Konfigurationsobjekt verweist auf eine Funktion, die ausgeführt wird, wenn ein Accessory von diesem Typ erfolgreich in das HAP eingebunden wurde.

Als ID eines Smart wird die Wi-Fi-MAC-Adresse verwendet.

Das Wi-Fi kann mithilfe des Espressif-eigenen Wi-Fi-Provisioning-Verfahrens eingerichtet werden. Ist alles korrekt konfiguriert, wird der lokale HAP Core der HomeKit SDK gestartet und das Wi-Fi des Smart aktiviert. Zur Einrichtung des Wi-Fi muss die App

„ESP BLE Provisioning“ von Espressif auf einem mit Bluetooth und Wi-Fi ausgestatteten Smartphone heruntergeladen werden. Für die Verifikation des Wi-Fi kann ein zuvor generierter Quick-Response-Code (QR-Code) verwendet werden. Wurde das Gerät erfolgreich verifiziert, müssen die Wi-Fi-Eingangsdaten des Heimnetzes eingegeben werden. Die Daten werden an das Smart weitergeleitet. Anschließend kann das Gerät von der Home App gefunden und erkannt werden.

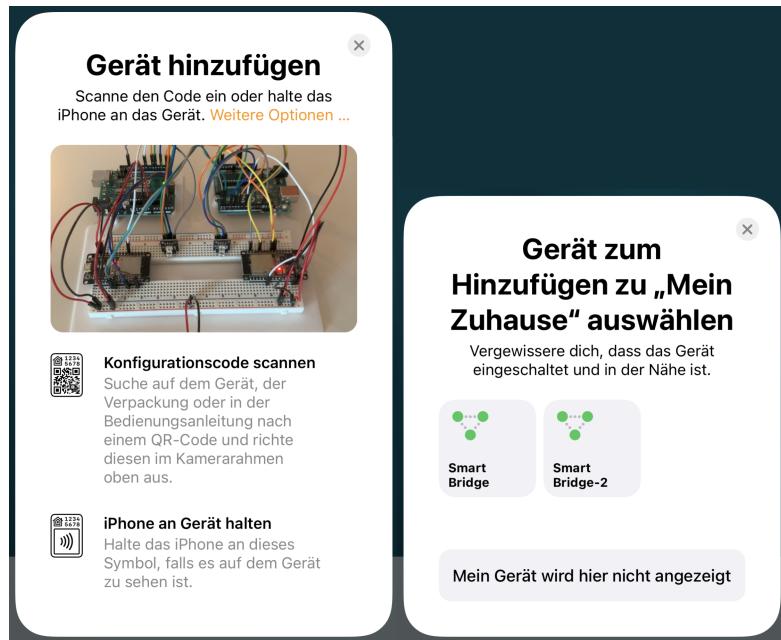


Abbildung 6.6: Registrierung der Bridge in der Home App (Originalbild bearbeitet)

Für die Kopplung mit dem Framework muss ein Setup-Code erstellt werden, der vom Nutzer eingegeben werden muss. Er ist für jedes Gerät individuell zu erstellen. In dieser Arbeit wurde der von der SDK vorgefertigte Beispielcode „111-22-333“ verwendet. Will ein Nutzer das Smart mit seiner Home App koppeln, muss dieser eingegeben werden. Alternativ bietet das SDK dem Nutzer auch hier die Möglichkeit, die Kopplung mithilfe eines QR-Codes schneller durchzuführen.

In Abbildung 6.6 werden beide Möglichkeiten zur Kopplung gezeigt. Der Schritt zur Einrichtung des Wi-Fi ist einmalig durchzuführen, solange das Smart nicht in den Werkszustand zurückgesetzt und sein Non-Volatile Storage (NVS) nicht gelöscht wird. Da lediglich das Smart über die notwendigen Technologien verfügt, entfallen diese zusätzlichen Schritte für die Einbindung der Devices. Die Devices werden von dem Smart direkt in die Home App und das Framework eingebunden, sobald sie von dem Smart erfasst und angemeldet wurden.

Wie in Abbildung 6.3 zu sehen ist, wird nach dem Einrichten der Bridge nach verbundenen Devices gesucht. Dazu werden Ping-Anfragen auf alle unterstützten Schnittstellen gesendet. Sie fragen die *Device-ID* eines Device an.

Wird auf einem Port innerhalb von zwei Sekunden keine Antwort empfangen, sendet das Smart die Anfrage erneut. Das geschieht auch auf dem jeweiligen Port, wenn auf den anderen Ports bereits eine Antwort angekommen ist. Das Senden der Pings endet nach 5 Versuchen oder wenn alle Ports mit einem Device belegt sind. Hier wird ein Resend vorgenommen, da noch keine aktive Kommunikation und kein Abbild des Device vorhanden ist. Die Mechanismen zur Fehlererkennung und Fehlerbehandlung können zu diesem Zeitpunkt noch nicht eingesetzt werden. Es wird angenommen, dass eine Verbindung keine erfolgreiche Kommunikation zulässt, wenn sie nach 5 Versuchen mit einer Verzögerung von 2 Sekunden noch nicht aufgebaut wurde. Die Grenze von 5 Versuchen wurde eingeführt, da es in der Anwendung sehr wahrscheinlich ist, dass ein Smart nicht alle seine Schnittstellen nutzen wird. Es wird verhindert, dass diese Ports dauerhaft einen Ping erhalten, auch wenn sie nicht genutzt werden. Dass auf einem Port keine Antwort empfangen wird, kann auch vorkommen, wenn ein Device erst nach dem Hochfahren des Smart angeschlossen wird. Falls keine Kommunikation möglich ist, kann ein Nutzer den Fehler beheben. Alternativ kann ein Gerät angeschlossen und das Smart daraufhin neu hochgefahren werden. Sofern auf dem Kanal keine unnatürlich starken Störungen vorhanden sind, wird auch dieses Device erkannt.

Anmeldung

Für die Anmeldung eines Device wurde sich an dem Ablauf aus dem Design-Kapitel orientiert. Sobald eine *Device-ID* von einem der Ports empfangen wurde, wird das Device auf dem Port registriert. Das Smart kann das Device mithilfe seiner ID an den Port binden. Jegliche Kommunikation, die über diesen Port empfangen wird, kann so dem richtigen Device zugeordnet werden.

Wie zuvor bereits erwähnt, wird angenommen, dass das an die Schnittstelle angeschlossene Device zur Laufzeit nicht gewechselt oder ausgetauscht wird. Wenn diese Möglichkeit ohne einen Neustart gegeben werden soll, müssen Änderungen an der Implementierung vorgenommen werden. Optimierungen bezüglich dieser und anderer Prozesse werden am Ende dieser Arbeit ausführlicher behandelt.

Um einen Überblick über die belegten Ports zu haben, hält das Smart eine Map. Die Map mit dem Namen `serial_mapping` bildet einen seriellen Port eindeutig auf eine *Device-ID* ab.

Wird eine Antwort mit einer ID auf einem Port empfangen, auf dem bereits ein Device registriert ist, wird dieser Wert überschrieben. In einer normalen Programmausführung tritt dieser Fall jedoch selten auf. Er kann vorkommen, falls durch den Resend-Mechanismus der Anfrage eine zweite Anfrage an dasselbe oder das danach angeschlossene Gerät gesendet wurde und die vorherige Antwort noch nicht verarbeitet wurde. Da

die *Device-ID* jedoch nichts über die Funktionsweise des Device aussagt, ist ein Fehlerfall hier zunächst nicht problematisch.

Wurde die *Device-ID* hinzugefügt, wird anschließend der *Device-Type* von dem jeweiligen Device angefragt. Wird diese Antwort empfangen und es existiert noch kein Device mit der *Device-ID*, wird ein Device-Objekt erstellt und die Synchronisierung des Abbildes eingeleitet. Um das Device nicht mit Anfragen zu überfordern, wird bei der Synchronisierung (Aufbau des State) jede Anfrage mit einem Abstand von 100 ms versendet.

Für die Erzeugung des Device-Objektes ist das Modul *Device Factory* zuständig. Für die Datenhaltung der Devices wurde sich der objektorientierten Prinzipien ‚Vererbung‘ und ‚Polymorphie‘ bedient. Das hat den Vorteil, dass mit jedem Device auf dieselbe Art interagiert werden kann und die Devices sich nur in Einzelheiten unterscheiden. Ein detaillierter Einblick in die Datenhaltung und die Gestaltung der Device-Objekte wird im nächsten Unterabschnitt gegeben.

Die *Device Factory* basiert auf dem Factory-Design-Pattern, das zu den Creational-Patterns gehört. Es ist darauf ausgelegt, dem Anwender eine allgemeine Schnittstelle für die Erzeugung verschiedener Objekte derselben Oberklasse zu geben und die darunterliegende Logik zu verschleiern. Sie wird angewandt, wenn die Unterklass für die Nutzung des Objektes nicht relevant ist, aber eine Unterscheidung zum Zeitpunkt der Erstellung notwendig ist.

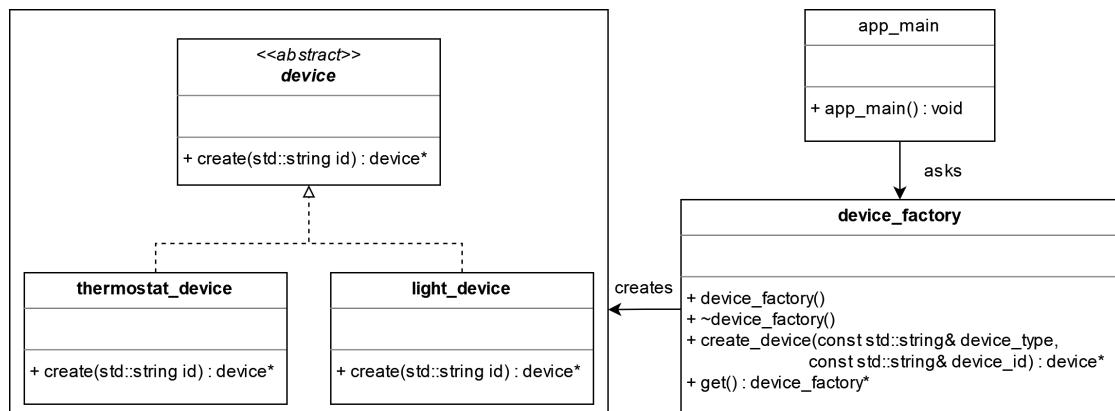


Abbildung 6.7: UML-Klassendiagramm: Device Factory

Da sich die Devices größtenteils in ihren Eigenschaften unterscheiden, wurde sich für die Anwendung dieses Patterns entschieden.

Ein UML-Diagramm von der Umsetzung des Patterns ist in Abbildung 6.7 zu sehen. Die Erstellung des Objektes wird in diesem Pattern von einer Funktion der Unterklasse selbst übernommen. Die Unterklasse enthält alle Informationen, um das Objekt genau zu beschreiben. Die Unterscheidung zwischen den einzelnen Unterklassen wird in dieser

Umsetzung mithilfe des *Device-Type* vorgenommen.

Zur Erzeugung der verschiedenen Objekte der Unterklassen kann in Java beispielsweise Reflection eingesetzt werden. Das ermöglicht die dynamische Erstellung von Objekten und das Ausführen von Funktionen durch die Angabe von Strings.

In C++ stehen einem diese Hilfsmittel nicht zur Verfügung. Aus diesem Grund wurde sich für die Verwaltung der möglichen Unterklassen ebenfalls einer Map bedient. Die Map bildet jeweils einen Namen (String) auf eine Funktion ab, welche sich in jeder Unterklass von `device` befindet. Diese Funktion ist die *Create-Funktion*. Sie erstellt ein neues Objekt der jeweiligen Unterklasse und gibt einen Pointer auf das erstellte Objekt der Oberklasse `device` zurück. Der Aufbau der Map ist in der Auflistung 6.1 nachzuvollziehen.

Auflistung 6.1: Hinzufügen der Create-Funktion

```
device_factory::device_factory() {
    add_device("light", &light_device::create);
    add_device("thermostat", &thermostat_device::create);
}
```

Für die Erstellung eines Objektes von einem bestimmten Typ muss ein Eintrag in dieser Map vorhanden sein.

Die Funktion *Create-Device*, die für die Generierung der Objekte zuständig ist, nimmt zwei Strings entgegen. Der *Device-Type* wird genutzt, um die passende *Create-Funktion* zu finden und das richtige Objekt zu erzeugen. Die *Device-ID* wird der Funktion zusätzlich mitgegeben, um diese bei der Erstellung des Objektes zu setzen. Der von dem Device gesendete *Device-Type* wird in dieser Funktion mit den Namen in der Map verglichen. Wurde die passende Funktion gefunden, wird sie ausgeführt. Anschließend wird ein Pointer auf das neu erstellte Device-Objekt zurückgegeben. Der Pointer auf das von der *Device Factory* generierte Objekt wird in dem Smart in `connected_devices` gespeichert.

Die Map `connected_devices` bildet die *Device-ID* auf den Pointer des jeweiligen Device ab. Um ein weiteres Mapping zu verhindern, wird zusätzlich auch der Serial-Port in dem Device gespeichert.

Wurde das Objekt für die lokale Repräsentation korrekt erstellt, kann das Device bei den Smart-Home-Frameworks angemeldet und eingerichtet werden. Anschließend wird jede Eigenschaft des Device einzeln von dem Device angefragt. Jedes Device-Objekt besitzt dafür eine Liste an Parametern, die im Protokoll mit einem bestimmten Namen hinterlegt sind. Damit die Prüfung der *State-C checksum*, wie bereits in dem Kapitel ‚Design‘ erläutert, nicht nach jeder Nachricht einen erneuten Neuaufbau des Zustandes einleitet, wird zum Zeitpunkt der Erstellung des Objektes das Flag `invalid_property_sum` auf den Wert ‚true‘ gesetzt. Das Senden der Nachrichten wird in einem externen Thread erledigt. Zusätzlich wird ein spezieller Observer-Thread gestartet, der über die gesamte Lebenszeit eines Device existiert und nur dann geschlossen wird, wenn das Objekt ge-

löscht wird. Dieser Thread ist sowohl für das Senden der Keep-Alive-Messages als auch für das kontinuierliche Überprüfen der `invalid_property_sum` zuständig. Die genaue Funktionsweise dieses Threads und weitere Mechanismen zur Fehlerbehandlung werden in einem späteren Abschnitt beschrieben.

Nach der erfolgreichen Einrichtung des Device und der Synchronisierung der Zustände, kann der normale Programmablauf ausgeführt werden.

Framework Device-Konfiguration

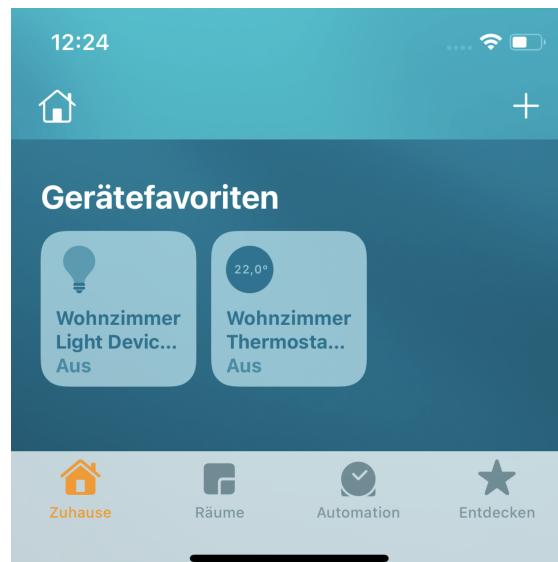


Abbildung 6.8: Eingebundene Devices in der Home App (Originalbild bearbeitet)

Die Anmeldung des Device wird von dem Smart vorgenommen, sobald das Device-Objekt erstellt wurde.

Wie erfolgreich eingebundene Devices in der Home App aussehen können, wird in Abbildung 6.8 gezeigt. Für die Anmeldung in das Smart-Home-Framework müssen unterschiedliche Parameter konfiguriert werden.

Jede Repräsentation eines Gerätes, welches im HomeKit-Framework angemeldet wird, wird als Accessory bezeichnet. Accessories im Kontext des HAP setzen sich aus Services und Characteristics zusammen [7]. Ein Accessory kann zum Beispiel ein Smart Light sein.

Services dienen der Gruppierung von Funktionalitäten eines Accessory und bieten so einen gewissen Ausführungskontext. Eine Lampe besitzt beispielsweise einen ‚Light Service‘. Bei komplexeren Geräten sind jedoch auch mehrere Services möglich, um die einzelnen Funktionen zu trennen.

Ein Service besitzt einen Typ mit einer einzigartigen UUID [7]. Characteristics sind Features, die Daten oder Verhaltensweisen eines Service repräsentieren. Jede Characteristic besitzt ebenfalls einen einzigartigen Typ, ein Format und Informationen über den Zugriff auf diese Characteristic [7].

Property	Value
UUID	00000049-0000-1000-8000-0026BB765291
Type	public.hap.service.switch
Required Characteristics	"9.70 On" (page 191)
Optional Characteristics	"9.62 Name" (page 188)

Abbildung 6.9: HAP-Spezifikation: Switch [7]

In Abbildung 6.9 ist ein Abschnitt aus dem HAP [7] zu sehen, in dem ein Binary Switch beschrieben wird. Der Service für einen Switch hat den Typ *public.hap.service.switch* [7]. Die Required Characteristic eines Switch ist „On“. Sie beschreibt die Zustände „On“ und „Off“ durch den Datentyp „Bool“. Die Zugriffsmöglichkeiten auf diese Characteristic sind als „Paired Read“, „Paired Write“ und „Notify“ gekennzeichnet [7]. Der Zustand der Characteristic kann also durch Anfragen gelesen und geschrieben werden. Zusätzlich ist es möglich, dass eine Benachrichtigung das Framework über eine Änderung der Characteristic informiert [7].

Für die Umsetzung in dieser Arbeit wurden die Accessories *Light Bulb* und *Thermostat* ausgewählt.

Da sich ein Accessory im Kontext des HomeKit nur durch seine Services auszeichnet, sind die Konfigurationsmöglichkeiten ähnlich wie bei der Erstellung der Bridge.

Der Name der angemeldeten Devices ist im Gegensatz zu der Bridge nicht statisch. Er wird bei der Generierung der Device-Objekte erzeugt und setzt sich zusammen aus dem *Device-Type* und einem Zählerwert. Der Zählerwert indiziert die Devices desselben Typs. Er erhöht sich für einen speziellen Typ nach der Generierung eines neuen Device. Der Name des ersten verbundenen Thermostats wird beispielsweise „Thermostat Device 0“ lauten.

Ein Service wird durch eine bestimmte Setup-Funktion des HomeKit SDK mithilfe des Bezeichners erstellt. Mit diesem Service können zusätzliche Daten verknüpft werden. Um das Ziel einer Anfrage aus der Home App einem bestimmten Device zuordnen zu können, wird in dieser Umsetzung die *Device-ID* mit dem Service verknüpft. Dadurch kann die *Device-ID* bei jeder Interaktion vonseiten des Frameworks mit dem jeweiligen Service mitgesendet werden. Die Interaktion mit dem Service wird über bestimmte Funktionen gewährleistet, die ebenfalls mit dem Service verbunden werden. Wie die meisten Implementierungen von IoT-Frameworks arbeitet das HomeKit SDK dabei mit

Callback-Funktionen. Sie werden aufgerufen, sobald Nachrichten von dem Frameworks empfangen werden. Da verschiedene Callback-Funktionen eines spezifischen Frameworks unterschiedliche Signatur besitzen, ist es leider nicht möglich diese weiter zu abstrahieren oder zu generalisieren. Für jedes Framework müssen alle notwendigen Callbacks einzeln implementiert werden.

Für die Nutzung des HomeKit sind zwei Callback-Funktionen relevant: „Read“ und „Write“. Sie repräsentieren die Anfragen „Paired Read“ und „Paired Write“ für alle Characteristics eines unterstützten Device. Anstatt diese Funktionen für einzelne Characteristics zu schreiben, wird bei jedem Service auf dieselben Callbacks verwiesen. Die Unterscheidung, auf welches Accessory und welche Characteristic sich die Anfrage bezieht, wird innerhalb der Funktion vorgenommen. Entscheidend ist dabei die zuvor konfigurierte *Device-ID* und die UUID der Characteristic, die der Funktion als Funktionsparameter übergeben werden.

Die Funktion „Read“ wird aufgerufen, wenn die Home App ihre Daten aktualisiert und den aktuellen Zustand der Smart Devices abrufen will. „Write“ wird aufgerufen, wenn über die Home App eine Anfrage zur Zustandsänderung einer Characteristic eingeleitet wurde. In der Implementierung des Smart heißen diese Funktionen `homekit_read_callback` sowie `homekit_write_callback`.

Datenhaltung

Die Datenhaltung des Device-Abbildes ist so strukturiert, dass jedes Device-Objekt über die gleiche Anzahl bestimmter Parameter verfügt, die für die Verwaltung und die Interaktion mit jedem Device benötigt werden. Gekapselt werden sie in einer abstrakten Oberklasse mit dem Namen `device`. Von ihr erben alle Device-Unterklassen und ergänzen sie durch gerätespezifische Eigenschaften. Die verwendete Klassenhierarchie ist ebenfalls in Abbildung 6.7 zu erkennen. Die Oberklasse `device` enthält dabei keine gerätespezifischen Eigenschaften, sondern dient der Speicherung von wichtigen Zusatzinformationen. Zu diesen Informationen gehören unter anderem:

- **Device-Name** - Name des Device, bestehend aus Typ + Index
- **Device-ID** - Von dem Device empfangene Device-ID
- **Serial-Port** - Serielle Schnittstelle auf dem die Device-ID empfangen wurde
- **Property-Names** - Aufzählung aller Eigenschaften eines Device
- **Accessory** - Accessory Objekt als Referenz für HomeKit SDK
- **Service** - Service Objekt als Referenz für HomeKit SDK
- **Invalid-Property-Sum** - Flag, welches eine fehlerhafte Übertragung signalisiert
- **Last-Communication-Timestamp** - Zeitpunkt der letzten empfangenen Nachricht
- **Checksum-Message-Counter** - Zähler für die Synchronisierung der Ping-Anfragen

Einige dieser Informationen und ihre Anwendungszwecke werden in den nachfolgenden Abschnitten genauer betrachtet.

Die spezialisierten Unterklassen wie `thermostat_device` erweitern die Device-Objekte um die notwendigen Eigenschaften für das Abbild des physischen Device. Für die Eigenschaften der Devices wurde sich an den Characteristics des HAP [7] orientiert. Sowohl die unterschiedlichen Characteristics eines Service als auch die Angaben bezüglich ihrer Datentypen wurden aus dem HAP übernommen. Erweitert sich die Anzahl der unterstützten Frameworks, so können weitere Eigenschaften hinzukommen. Es wird angenommen, dass sich die Datentypen der Frameworks nicht elementar unterscheiden. Falls notwendig kann zusätzlich eine Formatierung dieser Datentypen für die Kommunikation mit den Frameworks vorgenommen werden. Die Eigenschaften sind in den Objekten in Form von Member-Variablen hinterlegt.

Ein `thermostat_device` hat beispielsweise die Variablen `curr_temp` und `tar_temp`. Sie sind vom Typ `float` und stehen stellvertretend für die Characteristics *Current Temperature* und *Target Temperature*. Der Zugriff auf diese Variablen wird mithilfe von zwei standardisierten Funktionen sichergestellt. Die in der Oberklasse deklarierten Funktionen `get_property` sowie `set_property` müssen von jeder Unterklassie überschrieben und implementiert werden. Durch die Übergabe des jeweiligen Property-Namens (String) können die Eigenschaften eines Device einheitlich abgefragt und geändert werden. Die Signaturen dieser Funktionen finden sich in Auflistung 6.2 wieder.

Auflistung 6.2: Funktionssignaturen set/get_property

```
int set_property(std::string&, void*);  
void* get_property(std::string&);
```

Um den Wert einer Eigenschaft zu ändern, erhält die Funktion `set_property` den Namen einer Property sowie einen Pointer auf den gewünschten neuen Wert. Der Typ des übergebenen Wertes ist in dem Protokoll für jede Eigenschaft festgelegt.

Wird eine Eigenschaft mit dem richtigen Namen gefunden, dann wird ihr der übergebene Wert zugewiesen und die Funktion gibt eine positive Rückmeldung (0) zurück. Ist die Eigenschaft in dem Device unter dem angegebenen Namen nicht zu finden, wird eine negative Rückmeldung (-1) zurückgeben.

Das Abfragen einer Property (Eigenschaft) geschieht auf einem ähnlichen Weg. Die Funktion `get_property` nimmt den Namen einer Property entgegen, sucht diese und antwortet mit einem Pointer auf den Wert dieser Property. Wird keine Property mit diesem Namen gefunden, dann gibt die Funktion einen `nullptr` zurück.

Die Berechnung der *State-Checksum* eines Device wird ebenfalls in der Unterkasse realisiert. Dazu werden alle Eigenschaften miteinander addiert und als 8-Bit Unsigned Integer ausgegeben. Damit die Nachkommastellen der Variablen vom Typ `float` ebenfalls mit einbezogen werden, werden diese zuerst in einen `uint32_t` umgewandelt und mit 100 multipliziert. Da für die Temperaturangaben lediglich zwei Nachkommastellen relevant

sind, ist 100 hier ausreichend. Die Umwandlung in einen 8-Bit Integer ist vergleichbar mit einem Modulo von 256. Dieser Integer-Wert stellt die endgültige *State-Checksum* des Device dar.

6.2.2 Device

Die wichtigsten Aufgaben des Device sind es, die Anfragen des Smart entgegenzunehmen, sie zu verarbeiten, passende Funktionen zu simulieren und dem Smart sinnvolle Antworten zurückzugeben. Auf die Einzelheiten in der Implementierung der seriellen Schnittstellen sowie der Kommunikation mit dem Smart wird im Verlauf des nachfolgenden Abschnittes näher eingegangen.

In diesem Abschnitt werden der Aufbau des Device sowie die notwendigen Schritte betrachtet, um die Funktionen des Device sicherzustellen.

Das in dieser Arbeit entwickelte Device umfasst die Implementierung von zwei verschiedenen logischen Devices. Das bedeutet, der Arduino ist in der Lage, die Rolle von zwei Devices einzunehmen und ihre Funktionsweise zu simulieren. Für beide Funktionsmodi wurden unterschiedliche Verhaltensweisen implementiert. Die für die spezifischen Verhaltensweisen implementierten Vorkehrungen werden in den folgenden Unterabschnitten erläutert.

Auflistung 6.3: Set Mapping

```
const static arx::map<char*, bool (*)(char*)> set_map = {
    {"on", &set_on},
    {"brightness", &set_brightness},
    ...
};
```

Einrichtung

Als *Device-ID* wurde in dieser Implementierung die vom Hersteller des Mikrocontrollers hinterlegte Hardware-ID gewählt. Die ID ist im Kontext des Herstellers (Arduino) eindeutig und führt somit zu keiner Dopplung.

Das Umschalten zwischen den beiden logischen Devices ist durch das in dem Device hinterlegte Flag `light_device` mit dem Datentyp `bool` umgesetzt. Das Flag muss händisch vor dem Hochladen des kompilierten Programmcodes gesetzt werden, um die Funktionsweise des Device zu verändern. An wichtigen Stellen wird dieses Flag ausgewertet und ein anderer Programmablauf eingeleitet.

Für jede unterstützte Anfrage ist in dem Device ein Mapping auf eine geeignete Funktion hinterlegt. Um die jeweiligen Semantiken *Get* und *Set* voneinander zu trennen, wurde dieses Mapping mithilfe von zwei Maps realisiert. Eine verkürzte Darstellung

der *Set-Map* ist in Auflistung 6.3 nachzuvollziehen. Da die Arduino IDE viele C++ Standard-Bibliotheken nicht unterstützt, wurde mit „ArxContainer“⁵ eine externe Library eingebunden, die Typen wie Vector, Map und Array implementiert.

Light Device

Für die Funktionsweise des Lichtes sind vier Parameter relevant: `on`, `brightness`, `saturation` und `hue`. Sie wurden der Beschreibung eines *Light Bulb Service* aus dem HAP [7] entnommen. Für die Beschreibung der Lichteigenschaften wird das Format Hue-Saturation-Brightness (HSB) verwendet. Der Wert der Helligkeit wird durch einen ganzzahligen Prozentwert angegeben. Das Feld `brightness` ist daher vom Typ Integer und kann Werte zwischen 0 und 100 annehmen. Die angeschlossene RGB-LED wird über drei Pins für die Werte Rot (6), Grün (5) und Blau (3) angesteuert. Damit jeder dieser Werte mit einer Genauigkeit von 8-Bit einzeln konfiguriert werden kann, müssen die Pins eine hardwareseitige Pulse-Width Modulation (PWM) unterstützen. Der Arduino UNO R3 verfügt über 6 dieser speziellen GPIO-Pins. Falls nicht genügend Pins mit PWM-Unterstützung vorhanden sind, ist diese Semantik auch in Software umsetzbar. Darauf wurde in dieser Implementierung jedoch verzichtet. Da die LED nur die Werte R, G und B entgegennimmt, muss eine Konvertierung von dem HSB-Format in das RGB-Format durchgeführt werden. Dabei wurde sich an einer Ressourcen-freundlichen Implementierung von Kasper Kamperman [14] orientiert. Für die Umsetzung der Dim-Curve hat dieser eine Lookup-Table eingesetzt. Sie wird verwendet, damit die einzelnen Schritte der Helligkeit natürlich ineinander übergehen. Die Werte dieser Lookup-Table stammen aus einer Exponentialfunktion.

Um den Programmspeicher (SRAM) durch diese 256 Byte große Tabelle nicht zu belasten, wird sie im Flash-Speicher des Arduinos gehalten. Dazu wurde das Keyword `PROGMEM` bei der Initialisierung der Tabelle hinzugefügt.

Verändert sich ein Wert des Light Device, so werden die RGB-Werte errechnet und an die Pins übergeben. Das sollte anschließend zu einer Änderung des Lichtes der verbundenen LED führen. Solange sich die Werte auf den Pins nicht ändern, werden die aktuellen Werte von dem Arduino gehalten. Das Speichern eines komplexen State ist in dieser Anwendung nicht nötig.

Thermostat Device

Das HAP beschreibt für die Umsetzung eines Thermostats fünf Required Characteristics: *Current Heating Cooling State*, *Target Heating Cooling State*, *Current Temperature*, *Target Temperature* und *Temperature Display Units* [7].

⁵<https://github.com/hideakitai/ArxContainer>

In der Implementierung des Device wird jede dieser Characteristics von einer Property abgebildet.

Die Funktionsweise eines Thermostats geht über das zustandslose Entgegennehmen von Anfragen, wie es beispielsweise bei einem Light Device der Fall ist, hinaus. Aus diesem Grund sind in den Vorgaben des HAP bereits unterschiedliche Zustandseigenschaften integriert. Wichtig ist hierbei die Unterscheidung zwischen den aktuellen (Current) und den gewünschten (Target) Eigenschaften des Device.

Im Gegensatz zur Betrachtung einer Lampe kann nicht angenommen werden, dass eine Änderung der Temperatur sofort und ohne große Verzögerung durchgeführt wird. Es existiert also eine Indirektion zwischen dem, was ein Thermostat wahrnehmen und beeinflussen kann.

Die aktuellen Werte beschreiben den von vielen verschiedenen Faktoren abhängigen Zustand, der von dem System wahrgenommen und übermittelt werden kann. Die gewünschten zukünftigen Werte beschreiben die von dem Nutzer gegebenen Anweisungen. Abhängig von der Situation sind diese schnell, langsam oder gar nicht von dem System umsetzbar.

Es kann also eine Diskrepanz zwischen den Soll- und Ist-Zuständen existieren. Um diese Diskrepanz zu verringern, werden die Aktuatoren des Systems eingesetzt. Sie sollen die Umgebung beeinflussen. Bei einem Thermostat ist das zum Beispiel durch die Aktivierung einer Heiz- bzw. Kühlfunktion umgesetzt. Entsprechende Funktionsweisen werden in dieser Arbeit von dem Device simuliert.

	9.119 Target Heating Cooling State	9.32 Current Heating Cooling State
Valid Values	0 "Off" 1 "Heat. If the current temperature is below the target temperature then turn on heating." 2 "Cool. If the current temperature is above the target temperature then turn on cooling." 3 "Auto. Turn on heating or cooling to maintain temperature within the heating and cooling threshold of the target temperature." 4-255 "Reserved"	0 "Off." 1 "Heat. The Heater is currently on." 2 "Cool. Cooler is currently on." 3-255 "Reserved"

Abbildung 6.10: Vergleich: Target/Current Heating Cooling State
(vgl. [7])

Im Kontext des HAP besitzt ein Thermostat unterschiedliche Operationsmodi. Ein Überblick über die verschiedenen Operationsmodi und ihre Beschreibung wird auf der linken Seite der Abbildung 6.10 gegeben.

Sie werden als 8-Bit Unsigned-Integer angegeben. Durch Eingaben in der Home App kann ein Nutzer zwischen diesen Modi durchschalten. Der gewünschte Modus eines Nutzers wird in der Characteristic *Target Heating Cooling State* festgehalten. Wie in der Abbildung zu sehen ist, unterscheiden sich *Target* und *Current* deutlich voneinander.

Die Characteristic *Current Heating Cooling State* spiegelt im Kontext der Anwendung eher den Zustand der Aktuatoren wider. Die ersten drei Operationsmodi können im optimalen Fall mit einer leichten Verzögerung von einem Soll- in einen Ist-Zustand überführt werden. Der Modus „Auto“ jedoch ist kein Modus, der von den Aktuatoren unterstützt wird. Er beschreibt eine veränderte interne Verarbeitung des Zustandes, bei der die Aktuatoren vom System automatisch angesteuert werden, um die Temperatur in einem bestimmten Bereich zu halten. Befindet sich das Device im Modus „Off“, soll von dem Thermostat keine Temperaturanpassung vorgenommen werden.

Im Modus „Heat“ wird eine Erhöhung der Temperatur eingeleitet, sobald sie sich unter der gewünschten Temperatur befindet. Im Modus „Cool“ wird eine Verringerung der Temperatur eingeleitet, sobald sie sich über der gewünschten Temperatur befindet. Der Modus „Auto“ beschreibt die automatische Anpassung der Temperatur, wenn sich diese nicht in einem bestimmten, vorher definierten Bereich befinden. Um diesen Modus nutzen zu können, ist die Integration der Optional Characteristics *Cooling Threshold Temperature* und *Heating Threshold Temperature* notwendig, mit denen der Bereich konfiguriert werden kann.

Wird durch die Änderung des *Target Heating Cooling State* eine Abweichung von Soll- und Ist-Zustand wahrgenommen, dann ändert sich der aktuelle Modus (Current). Anschließend werden alle Smarts über diese Änderung informiert.

Um die Funktionen des Thermostats zu simulieren, wird der Wert der *Current Temperature* verändert. Zusätzlich wird eine weitere LED genutzt. Die LED repräsentiert dabei die Aktuatoren des Device. Wird die Temperatur erhöht, also die Funktion einer Heizung simuliert, leuchtet die LED in einem roten Licht auf. Wird eine Verringerung der Temperatur vorgenommen, leuchtet die LED entsprechend blau. Soll eine Veränderung der aktuellen Temperatur vorgenommen werden, so verändert sie sich alle 2,5 Sekunden um 0.5°C , bis die gewünschte Temperatur erreicht ist. Sind die Aktuatoren deaktiviert, ist das Licht der LED ausgeschaltet. Die neue Temperatur wird als *Current Temperature* an die verbundenen Smarts gesendet.

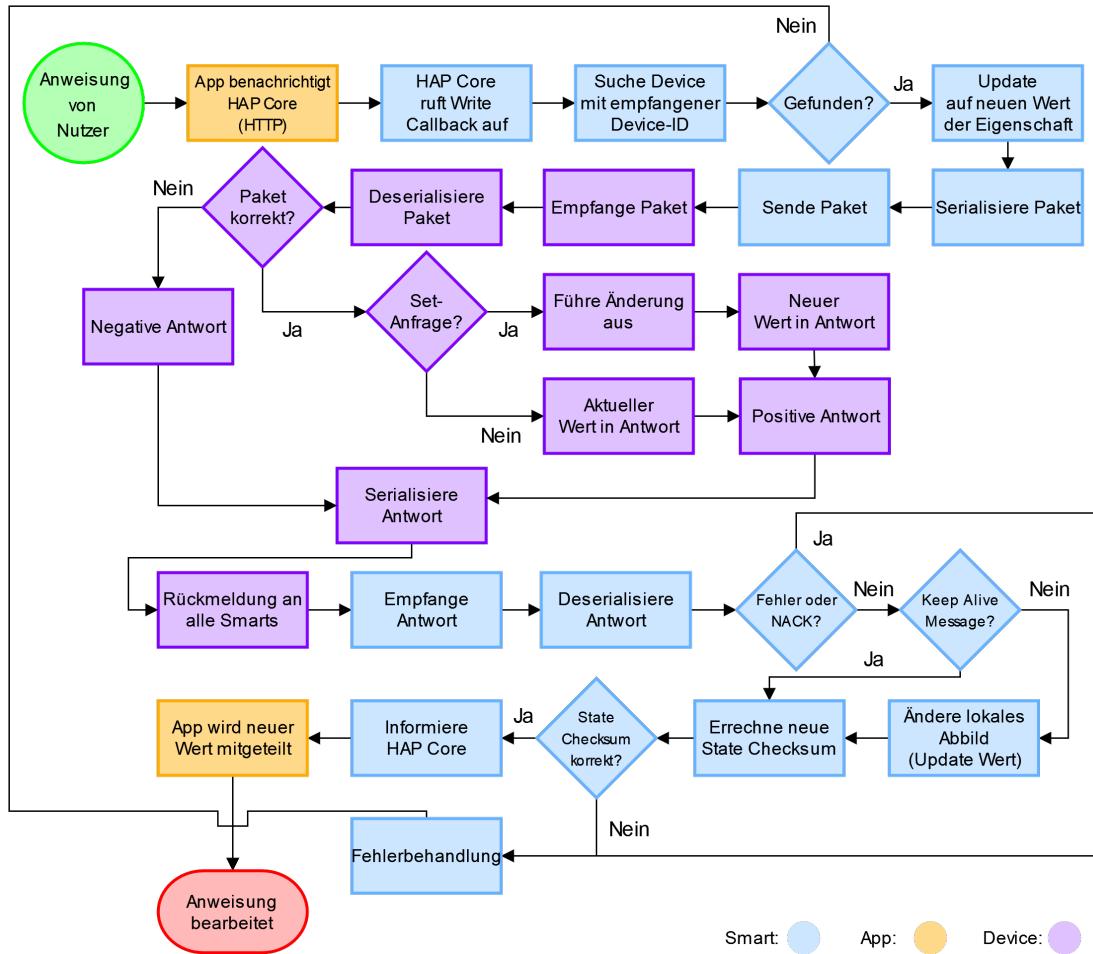


Abbildung 6.11: Nachrichtenaustausch

6.2.3 Kommunikation

Damit eine erfolgreiche Kommunikation zwischen Smart und Device sichergestellt werden kann, müssen auf beiden Seiten einige Schritte durchlaufen werden. Der vollständige Ablauf eines Datenaustauschs ist in Abbildung 6.11 in Form eines Flussdiagramms dargestellt. Der umgesetzte Kommunikationsablauf orientiert sich an den groben Vorgaben aus dem Kapitel „Design“. Ergänzt wird er durch das Hinzufügen zusätzlicher Schritte zur Serialisierung und Deserialisierung der Daten anhand des entwickelten Protokolls. In dem dargestellten Beispiel wird der Wert einer Eigenschaft verändert (Set). Die Abfragen der aktuellen Werte der Eigenschaften werden von dem Framework in regelmäßigen Abständen ohne Anfragen vom Nutzer eingeleitet (Get). Sie führen zu einem kürzeren

Programmablauf, da in diesem Fall lediglich die Werte des lokalen Abbildes an das Framework übermittelt werden.

Wie der dargestellte Programmablauf beim Ändern eines Wertes wahrgenommen werden kann, ist in Abbildung 6.12 zu sehen.

Dargestellt ist der Status desselben Light Device, welches über zwei verschiedene Smarts eingebunden wurde. Durch das Drücken auf eines der Lichter in der App soll das Licht angeschaltet werden. Wie im zweiten Teil des Bildes zu sehen ist, wird die Anweisung sofort von dem Framework übernommen und als ‚erfolgreich durchgeführt‘ betrachtet. Hat das Device die Anweisung verarbeitet, schaltet es die LED ein und sendet eine positive Antwort an die Smarts. Der letzte Teil des Bildes zeigt den Zustand, nachdem das zweite Smart die Änderung an das Framework weitergeleitet hat.

In den Abbildungen wird die Kommunikation zwischen den einzelnen Komponenten unabhängig von der verwendeten Übertragungstechnologie dargestellt.

Die zwei in der Anwendung integrierten Technologien besitzen jedoch sehr unterschiedliche Übertragungseigenschaften. Aus diesem Grund werden in den nachfolgenden Unterabschnitten notwendige Konfigurationsschritte vorgestellt, die in dieser Arbeit vorgenommen wurden, um einen korrekten und einheitlichen Datenaustausch zu gewährleisten.



Abbildung 6.12: Für Nutzer sichtbarer Programmablauf

Konfiguration

Die Konfiguration der Schnittstellen wird auf Seiten des Smart durch den *Serial Handler* vorgenommen. Beide Schnittstellen werden dabei mit einer Bitrate von 1 Mbit/s konfiguriert. Das dient der besseren Vergleichbarkeit der Ansätze im späteren Evaluations-Kapitel. Da beide jedoch auf unterschiedlichen Protokollen basieren, ist diese Einstellung nur begrenzt vergleichbar. Sie unterscheiden sich deutlich in der Formatierung der Pakete

und der damit verbundenen Anzahl an zusätzlichen Bits für bestimmte Kontrollstrukturen. Welche Auswirkungen diese Unterschiede haben und inwieweit die Technologien vergleichbar sind, wird im Kapitel ‚Evaluation‘ genauer betrachtet.

Die spezifischen Eigenschaften für das Senden der Nachrichten werden ebenfalls durch den *Serial Handler* abstrahiert. Das korrekte Übertragungsmedium wird anhand der für ein Device hinterlegten Schnittstelle erkannt. Zum Senden eines Paketes wird der Funktion ein Objekt der Klasse `serial_packet` und die ID der Schnittstelle übergeben. Die Daten werden anschließend serialisiert. Nach der Serialisierung werden die Daten über eine der seriellen Schnittstellen verschickt.

Serialisierung und Deserialisierung

Sowohl die Serialisierung als auch die Deserialisierung sind an das Übertragungsprotokoll gebunden. Um diese Funktionen zu nutzen, wird eine spezielle Protokoll-Klasse von dem *Serial Handler* eingebunden. Das Protokoll ist plattformunabhängig und muss auf beiden Seiten unterstützt werden. Daher steht dem Device eine ähnliche, leicht angepasste Version dieser Klasse zur Verfügung.

Zum Serialisieren dient die Funktion `marshal`. Sie nimmt ein Objekt der Klasse `serial_packet` entgegen und gibt einen Byte-Vector zurück. Notwendige Parameter wie *Data Type* und *Packet Type* sowie der Inhalt der Nachricht sollten in diesem Objekt gekapselt sein. Die Funktion fügt diese Informationen an die im Protokoll spezifizierten Stellen ein, ergänzt entsprechende Steuerzeichen und verwandelt das Paket in eine Sequenz von Bytes in einen Vector. Die Checksum der Nachricht errechnet sich dabei aus der Summe der restlichen Bytes. Da die Größe der Checksum ein Byte betragen soll, kann diese Summe in einem Bereich zwischen 0 und 255 liegen. Die Berechnung ist vergleichbar mit der Nutzung der Modulo-Funktion.

Die Deserialisierung wird durch die Funktion `unmarshal` implementiert. Sie ist das Gegenstück der Serialisierungs-Funktion. Sie nimmt einen Byte-Vector entgegen und gibt ein fertig formatiertes Objekt der Klasse `serial_packet` zurück. Neben dem Entpacken der Nachricht ist die Funktion auch dafür zuständig, die Korrektheit des empfangenen Paketes zu überprüfen. Ein falsch formatiertes Paket kann in bestimmten Fällen nicht sinnvoll entpackt oder gelesen werden. Aus diesem Grund werden zunächst alle Header und die Checksums auf Vollständigkeit und Korrektheit überprüft. Nur wenn das gesamte Paket überprüft wurde und die empfangene Checksum mit der lokal errechneten übereinstimmt, kann mit dem Entpacken fortgefahrene werden. Ist das nicht der Fall, wird ein Paket mit erkennbar falschen Werten zurückgeben.

UART

Für die Implementierung von UART auf dem Smart wurde ein in das ESP-IDF integrierter Treiber genutzt. Espressif stellt den Entwicklern eine simple API zum Einstellen der Übertragungsparameter sowie zum Senden und Empfangen von Daten über die physischen UART-Schnittstellen zur Verfügung. Dadurch entfällt beispielsweise die Arbeit mit hardwarespezifischen Registern und die manuelle Konfiguration von Interrupt Service Routines (ISRs).

Die in dieser Arbeit verwendete UART-Konfiguration zeichnet sich durch eine Baudrate von 1 Mbit/s, 8 Daten-Bits, keinem Paritäts-Bit und einem Stop-Bit aus. Die hardwareseitige Flow Control ist deaktiviert. Damit eine Kommunikation stattfinden kann, muss das UART-Modul bei allen Kommunikationspartnern gleich konfiguriert sein.

Zum Empfangen von Nachrichten besteht durch den Treiber zudem die Möglichkeit der Nutzung einer Message Queue. Diese Queue sammelt alle einkommenden Nachrichten und kann auf bestimmte Patterns reagieren. Anhand der erkannten Patterns können spezielle Funktionen ausgeführt werden. Dazu wird eine Handler Task gestartet, die auf die Events aus der ISR reagiert.

Auf dem Device wird die in der Arduino IDE integrierte Serial Library genutzt, um Daten mit UART zu versenden und zu empfangen. In jedem Zyklus der Main-Loop wird geprüft, ob sich Daten im internen Receive-Buffer der Serial Library befinden. Sind Daten vorhanden, werden sie in einen zusätzlichen Circular-Buffer eingefügt. Die Implementierung vom Circular-Buffer wurde aus einem vorherigen Projekt⁶ übernommen. Nach jedem Einfügen eines Bytes in den Buffer wird auf das Steuerzeichen geprüft, welches das Ende der Übertragung angibt. Der Buffer wird so lange gefüllt, bis das Zeichen erkannt wird. Wird dieses Zeichen durch einen Fehlerfall nicht empfangen, ist das erst bei der nächsten empfangenen Nachricht mit vorhandenen Steuerzeichen erkennbar. Wurde das Paket erfolgreich empfangen, wird es deserialisiert und die passende Funktion wird ausgeführt.

Da UART ein asynchrones Protokoll ist, können beide Gesprächspartner zu jeder Zeit Daten senden und bei Empfang der Daten entsprechend reagieren. Sobald das Smart eine Anfrage des Frameworks bekommt, kann es diese an das Device weiterleiten. Das Device bekommt eine Benachrichtigung über den Eingang der Daten. Durch die Asynchronität entsteht kein aktiver Warteprozess für das Smart. Wenn das Device die Daten verarbeitet hat und eine Funktion ausgeführt wurde, sendet es eine Antwort an das Smart. Wie bereits im Design-Kapitel erläutert, ist eine Antwort gleichzusetzen mit einer Benachrichtigung. Es ist unbedeutend, auf welche Anfrage sich die Antwort bezieht oder von welchem Smart die Anfrage kam. Das wird besonders dann deutlich, wenn mehrere Smarts mit einem Device verbunden sind.

⁶Projekt: Embedded Systems Bootcamp, unter der Leitung von Christopher Ringhofer
(Embedded Systems Department, Prof. Dr. Gregor Schiele)

Das Smart kann weiterarbeiten, während das Device die Anfrage bearbeitet. Die empfangene Antwort wird auf dem Smart in die Event-Queue eingefügt und auf die benötigten Steuerzeichen geprüft. Wird eine Antwort erkannt, kann sie deserialisiert und weiterverarbeitet werden.

I²C

Um I²C auf dem Smart zu realisieren, wurde ebenfalls ein von dem IDF bereitgestellter Treiber verwendet. I²C basiert auf einer Master-Slave-Topologie. Aus diesem Grund müssen die Zuständigkeiten von Master und Slave vor der Nutzung festgelegt werden. Das Smart übernimmt in dieser Arbeit die Aufgabe des Masters. Die Slaves werden von den Devices verkörpert. Notwendige Parameter müssen dabei von dem Master konfiguriert werden. So wird beispielsweise die Clock von dem Master bestimmt und auf einer separaten Verbindung mit den Slaves synchronisiert.

Zum Senden der Anfragen schickt das Smart ein I²C-Write mit dem serialisierten Paket an das jeweilige Device. Wie im Kapitel ‚Design‘ erwähnt wurde, verhindert die Master-Slave Topologie des Protokolls das aktive Senden einer Nachricht durch einen Slave. Ein Device muss jedoch dazu in der Lage sein, das Smart über Zustandsänderungen zu informieren. Zusätzlich kann ein Smart nach dem Senden einer Anfrage nicht wissen, wie lange ein Device benötigt, um sie zu bearbeiten. Wenn es die Antwort abholen will, kann es daher sein, dass sie noch nicht bereitgestellt wurde.

Um eine analoge Semantik wie bei UART zu erreichen, wurde sich in dieser Arbeit dafür entschieden, eine weitere Notification-Line einzufügen. Sie muss für jedes Device eingerichtet werden, welches mit einem Smart über I²C oder einem ähnlichen Protokoll verbunden ist. Auf dieser Notification-Line kann ein Device mit einem kurzen HIGH-LOW Impuls angeben, wenn Daten für das Smart bereitstehen. Die Daten können daraufhin von dem Smart angefragt werden. Vonseiten des Smart wird die Leitung mithilfe eines Pulldown-Registers auf LOW gehalten. So kann der Benachrichtigungs-Impuls des Device störungsfrei erkannt werden. Um das zu erreichen, wurde auf dem Smart ein Interrupt konfiguriert, welcher nur auf eine Rising Edge reagiert. Sobald das Signal von LOW auf HIGH steigt, wird der Interrupt ausgelöst und die passende Routine ausgeführt.

Wie sich diese asynchrone Variante von I²C im Vergleich zu der Implementierung von UART verhält und welche zusätzlichen Schritte durchlaufen werden müssen, ist in Abbildung 6.13 zu sehen.

Anstatt die Antwort wie bei der Verwendung von UART sofort zu senden, wird sie in einem Buffer gespeichert. Die Notification-Line wird für einen kurzen Zeitraum auf HIGH gesetzt. Die nächste Antwort wird im FIFO-Prinzip aus dem Buffer gelesen, wenn das Smart eine Read-Anfrage an das Device sendet. Es kann vorkommen, dass bereits neue Anfragen von dem Device verarbeitet wurden, ohne dass die Antworten der vorherigen Anfragen gelesen wurden. Aus diesem Grund ist der Buffer in der Lage mehrere Antworten zu halten. Bei jedem Read wird eine Antwort gelesen und aus dem Buffer gelöscht.

Inwiefern sich diese Zusatzschritte auf die Performance auswirken und ob die Nutzung einer solchen asymmetrischen Technologie im Vergleich zu einer symmetrischen Technologie wie UART Sinn ergibt, wird in dem nachfolgenden Kapitel ‚Evaluation‘ untersucht.

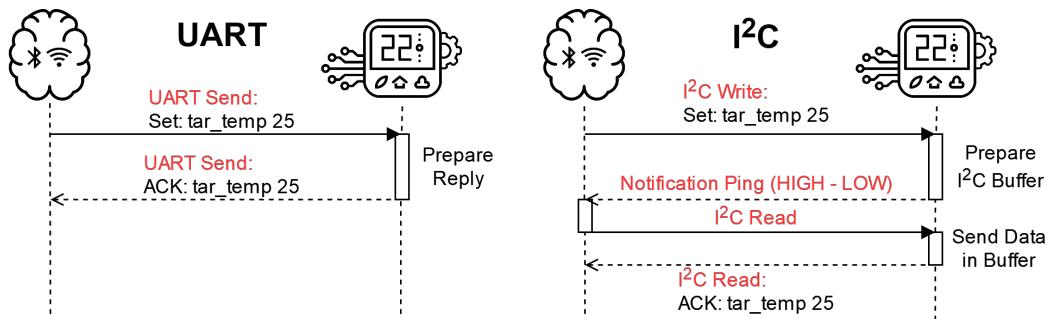


Abbildung 6.13: Vergleich: Nachrichtenaustausch von UART und I²C

6.3 Fehlerbehandlung

Der größte Teil der Fehlerbehandlung wird in dieser Arbeit von dem Smart übernommen. Fehler treten in den meisten Fällen bei der Übertragung zwischen Smart und Device auf. Da der Zustand eines Device als Wahrheit angesehen wird, ergibt es wenig Sinn, Fehler durch das Device zu berichtigen.

Im Kapitel ‚Design‘ wurde bereits deutlich, dass das Device in Bezug auf die Fehlererkennung eine informatorische Rolle einnimmt. Das ist auch in dieser Umsetzung der Fall. Es muss erkennen, ob eine Anfrage korrekt übertragen und formatiert wurde, damit es sie sinnvoll verarbeiten kann. Ist das nicht der Fall, so setzt es die Smarts darüber in Kenntnis und übermittelt ihre Informationen über den aktuellen Zustand (*State-Checksum*). Auf diese Weise ist das Device in die Fehlererkennung involviert. Es sollte Fehler erkennen, die sich in der Anfrage von einem Smart befinden.

6.3.1 Fehlererkennung

In der Kommunikation zwischen Smart und Device können vier verschiedene Fehlerfälle auftreten. Zur Erkennung der Fehlerfälle können vier Merkmale genutzt werden:

1. NACK empfangen und State-Checksum korrekt
2. NACK empfangen und State-Checksum fehlerhaft
3. Fehlerhafte Message-Checksum
4. Fehlerhafte State-Checksum

Die Fehlerfälle drei und vier aus dem Kapitel ‚Design‘ sind durch eine fehlerhafte *State-C checksum* direkt erkennbar (Merkmal 3). Dabei sollte der dritte Fehlerfall nur von dem Gerät erkannt werden, welches zuvor eine Anfrage gesendet hat. Das gleiche gilt für die Betrachtung des ersten Fehlerfalls.

Empfängt ein Smart ein NACK, bedeutet das, dass nach dieser Anfrage keine Zustandsänderung durchgeführt wurde. Das Smart, welches die Anfrage nicht gesendet hat, kann diese Antwort somit ignorieren. Daraus ergeben sich die Merkmale 1 und 2. Die *State-C checksum* verändert sich in dieser Umsetzung, sobald die Anfrage des Nutzers von dem Smart empfangen wurde. Das ermöglicht die oben genannte Unterscheidung in der Fehlererkennung.

Wird die Unterscheidung nicht vorgenommen, kann ein NACK zu einem unnötigen Neuaufbau des Abbildes auf jedem weiteren Smart führen.

Da bei einer fehlerhaften Message-Checksum nicht sichergestellt werden kann, dass die mitgesendete *State-C checksum* korrekt übermittelt wurde, muss in diesem Fall immer eine Fehlerbehebung durchgeführt werden. Das Smart kann nicht wissen ob die Nachricht zu einer Zustandsänderung geführt hat.

Wenn einer von den Fehlerfällen 2-4 festgestellt wurde, wird in dem entsprechenden Device-Objekt das Flag `invalid_property_sum` auf `true` gesetzt. Dieser Wert kann anschließend von dem Observer-Thread erkannt werden. Bei der Erkennung von Merkmal 1 ist kein Handeln notwendig.

6.3.2 Fehlerbehebung

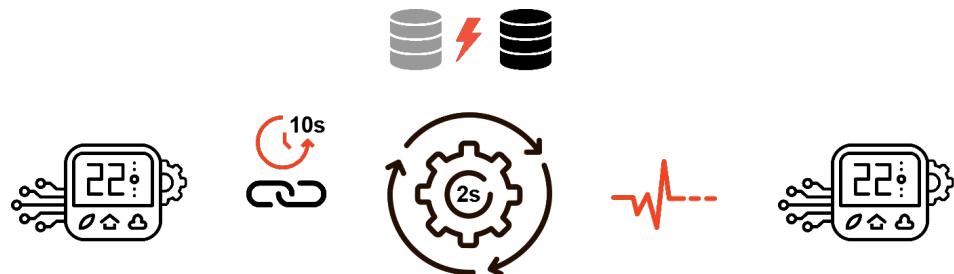


Abbildung 6.14: Aufgaben des Observer-Threads: Auf Timeout prüfen, State-C checksum prüfen, Keep-Alive-Messages senden

Die Fehlerbehebung wird von dem Observer-Thread des Device eingeleitet. Für jedes aktive Device existiert ein Observer-Thread. Die Aufgaben des Observer-Threads sind in Abbildung 6.14 anschaulich dargestellt.

Er übernimmt im Wesentlichen drei Aufgaben. In einem Intervall von 2 Sekunden sendet er Keep-Alive-Messages an das Device, und überprüft die Existenz sowie die Korrektheit der Kommunikation. Durch die Keep-Alive-Messages können auch fehlerhafte Zustände

erkannt werden, wenn der Nutzer nicht in regelmäßigen Abständen mit einem Device interagiert. Die Antworten der Keep-Alive-Messages werden unabhängig von dem Thread wie jede ankommende Nachricht behandelt. Sie führen zu keiner Statusänderung des Device, können jedoch eine Invalidierung des State hervorrufen. Um erkannte Fehler beheben zu können, überprüft der Observer-Thread zusätzlich im selben Intervall das `invalid_property_sum` Flag seines Device. Ist der State des Device fehlerhaft, wird das Senden der Keep-Alive-Messages ausgesetzt und jede Eigenschaft des Device neu angefragt. Wurde der State nach dem nächsten Intervall von 2 Sekunden nicht wiederhergestellt, wird angenommen, dass bei dem Neuaufbau erneut etwas schiefgegangen ist. Der Neuaufbau wird anschließend wieder initiiert.

Die dritte Aufgabe, die der Observer-Thread übernimmt, beschäftigt sich mit dem Fall, dass keine Kommunikation mehr mit dem Device möglich ist.

Der `last_communication_timestamp` gibt den letzten Zeitpunkt an, an dem eine Nachricht von dem Device korrekt empfangen wurde. Er wird auf den aktuellen Ausführungszeitpunkt gesetzt, sobald die Nachricht verarbeitet wurde.

Der Observer-Thread vergleicht den Timestamp mit dem aktuellen Ausführungszeitpunkt und kann so den Abstand zur letzten erfolgreichen Kommunikation errechnen.

Wenn die letzte erfolgreich empfangene Nachricht eines Device mehr als 10 Sekunden zurückliegt, wird angenommen, dass eine Kommunikation zu diesem Zeitpunkt nicht garantiert werden kann. Das Device wird daraufhin bei dem HAP Core abgemeldet und aus allen notwendigen Listen gelöscht. Auch das Device-Objekt wird verworfen. Für den Fall, dass das Device nur einen kurzzeitigen Ausfall hatte oder die Verbindung für eine kurze Zeit unterbrochen wurde, wird die Schnittstelle nach einem Disconnect mit Ping-Messages überprüft.

6.3.2.1 Besonderheit I²C

Da die Funktionsweise von I²C in dieser Arbeit um die Asynchronität erweitert wurde, sind neue Fehler zu beachten. Die Antwort auf eine Anfrage wird von dem Device in einen separaten Buffer geschrieben. Im Fehlerfall kann es vorkommen, dass die Notification-Line nicht korrekt verarbeitet wird. Wenn Störungen auf den Signalen vorliegen, kann es sein, dass Nachrichten von dem Smart nicht abgeholt werden. Es kann außerdem vorkommen, dass das Device bei einer neuen Verbindung mit einem neuen Smart noch alte Nachrichten von der letzten Verbindung in dem Buffer hält. Es ist nicht in der Lage, selbst zu entscheiden, wann diese Daten gelöscht werden sollten.

Damit veraltete Nachrichten aus dem Device entfernt werden können, wurde eine zusätzliche Flush-Semantik eingeführt. Sie wird im Fehlerfall und vor den initialen Pings ausgeführt und leert den Buffer des Device, sodass die Kommunikation korrekt stattfinden kann.

Beim Leeren werden so lange Daten vom Device angefragt, bis nur noch der Wert „0“ empfangen wird. Dieser Wert wird von dem Device gesendet, wenn der Buffer leer ist.

Ein besonderer Fall, bei dem keine Fehlerbehandlung eingeleitet wird, wenn Nachrichten verloren gehen, ist der Austausch von Ping-Messages. Eine Ping-Message führt zu keiner Zustandsänderung. Aus diesem Grund wird eine Fehlerbehandlung bei einem Fehlerfall nicht eingeleitet. Um den Buffer des Device trotzdem leeren zu können, wurde die Variable `checksum_message_counter` eingeführt. Er wird in jeder Ping-Message und der Antwort vom Device gesendet. Kommt eine veraltete Ping-Message beim Smart an, kann das Smart dies erkennen und einen Flush beim Device initiieren.

Kapitel 7

Evaluation

Dieses Kapitel beschäftigt sich mit der Evaluation des Gesamtkonzeptes dieser Arbeit sowie der im vorherigen Kapitel vorgestellten Umsetzung. Es soll die getätigten Designentscheidungen beurteilen und einen Eindruck davon geben, wo die Stärken und die Schwächen ihrer Implementierung liegen.

Die Idee der Auftrennung eines Smart Device in zwei separate Geräte entstand überwiegend aus dem Bestreben, den Herstellern einen großen Aufwand zu ersparen. Damit die Endanwender in der Umsetzung dieses Vorhabens nicht ausschließlich mit Kompromissen konfrontiert werden, wurden in dem Kapitel Vorabanalyse einige Anforderungen erarbeitet. Ein wichtiger Bestandteil der Evaluation ist die Betrachtung dieser Anforderungen und inwieweit sie in der Implementierung berücksichtigt wurden.

Dabei liegt der Fokus im ersten Abschnitt auf der Verzögerung, die durch die Auftrennung hinzukommt. Diese Verzögerung wird in unterschiedlichen Anwendungsfällen gemessen. Anhand der Messungen wird in diesem Abschnitt eine Aussage darüber getroffen, inwieweit die zusätzliche Verzögerung für den Anwender von Bedeutung sein kann.

Der darauffolgende Abschnitt dient der Beurteilung anderer Anforderungen und inwieweit sie bei der Umsetzung beachtet wurden. Es wird betrachtet, auf welche Weise bestimmte Teile der Anwendung umgesetzt wurden und welche Designentscheidungen gegebenenfalls zu anderen Ergebnissen führen können.

Der letzte Abschnitt dieses Kapitels beschäftigt sich mit den erarbeiteten Fragestellungen aus dem Kapitel Vorabanalyse. Es wird versucht, die Fragen anhand der entwickelten Anwendung zu beantworten und die Ergebnisse dieser Arbeit zu diskutieren.

7.1 Messungen

Die Messungen, die in diesem Abschnitt durchgeführt werden, beziehen sich auf die zusätzliche Verzögerung, die durch die Kommunikation zwischen Smart und Device entsteht.

Es soll gemessen werden, welche Verzögerung durch das Umleiten der Nachrichten entsteht und ob diese Verzögerung für den Anwender bemerkbar oder störend sein kann.

Dabei werden die beiden verwendeten Übertragungsprotokolle miteinander verglichen und voneinander abgegrenzt.

Um eine qualifizierte Aussage über die Verzögerung treffen zu können, ist es wichtig, die minimale sowie die maximale Verzögerung zu betrachten. Die Daten werden mit einer Geschwindigkeit von 1 Mbit/s übertragen. Beide verwendeten Geräte verfügen über weitaus höhere Taktraten. Aus diesem Grund wird angenommen, dass das Senden, Übertragen und Empfangen der Daten die meiste Zeit in Anspruch nimmt. Die Verarbeitungszeiten aufseiten des Device sollten durch unterschiedliche Nachrichten nur geringfügig voneinander abweichen. Als wesentlicher Faktor in der Betrachtung der Verzögerung wird daher die Nachrichtenlänge angesehen.

Für die Messung der minimalen Verzögerung wird die kürzeste Nachricht der aktuellen Implementierung betrachtet. Die kürzeste Nachricht entsteht durch eine Anfrage bezüglich der Eigenschaft `on` des Light Device. Das gesamte Paket inklusive Header und Checksums besitzt eine Länge von 14 Byte (10 Byte + #('on,1')).

Die maximale Verzögerung wird durch das Versenden der längsten Nachricht gemessen. Dazu wird eine Anfrage an das Device gesendet, die sich auf die Eigenschaft `tar_heat_cool_state` bezieht. Das Paket besitzt eine Gesamtlänge von 31 Byte (10 Byte + #('tar_heat_cool_state,1')).

Es ist hierbei ausreichend die Grenzwerte zu betrachten. Sie bilden den Rahmen an möglichen Verzögerungen, die von der Anwendung zu erwarten sind. Weitere Verzögerungen anderer Nachrichten können anhand dieses Rahmens nachträglich abgeschätzt werden.

7.1.1 Vorgehen

Gemessen wird an unterschiedlichen Punkten innerhalb der Anwendung, die jeweils andere Parameter betrachten, welche die Messungen beeinflussen können. Die Messungen werden ausschließlich aufseiten des Smart durchgeführt, da die dort gemessene Verzögerung in etwa die Verzögerung widerspiegelt, die der Anwender bemerken kann. Eine Betrachtung der Verzögerung jeder einzelnen Nachricht auf beiden Seiten hat in dem Kontext dieser Arbeit eine geringe Aussagekraft. Für die Betrachtung der Sendezeit ist die Übertragungsgeschwindigkeit des jeweiligen Übertragungsmediums ausreichend. Abweichungen, die aus der kabellosen Übertragung zwischen Smart und Smart-Home-Framework entstehen, existieren auch bei einem klassischen Smart Device und werden ebenfalls nicht betrachtet.

Die Durchführung der Messung geschieht über die Betrachtung eines dafür zugewiesenen Output-Pins. Dieser Pin wird auf HIGH gesetzt, sobald die Anfrage von dem Smart an das Device gesendet wurde. Empfängt das Smart die Antwort auf diese Anfrage, wird der Pin wieder auf LOW gesetzt. Gemessen wird die Länge des Zeitintervalls, bei dem der Wert dieses Pins HIGH beträgt. Um dieses Zeitintervall zu messen, wird ein Oszilloskop mit dem Output-Pin verbunden.

Abhängig davon, an welchen Punkten in der Anwendung diese Messung gestoppt wird,

können die Messungen erheblich voneinander abweichen. Werden zum Beispiel die Funktionen des HomeKit SDK in die Messungen miteinbezogen, kann eine zusätzliche Verzögerung beobachtet werden.

Für die Betrachtung der Gesamtverzögerung durch die in dieser Arbeit entwickelte Anwendung sollte die Verzögerung durch die Benachrichtigung des Frameworks miteinbezogen werden.

Wegen der Asynchronität des Nachrichtenaustausches wird nicht zwischen einer Anweisung unterschieden, die von dem Smart selbst an das Device gesendet wurde und einer, die lediglich als Benachrichtigung von dem Device empfangen wurde. Die empfangene Änderung muss passend formatiert und an das Framework weitergeleitet werden. Die Änderung wird anschließend im internen HAP Core angepasst. Das muss bei einem klassischen Smart Device nicht geschehen. Dieser Aufwand ist ein Zusatzaufwand und wird deshalb getrennt berücksichtigt.

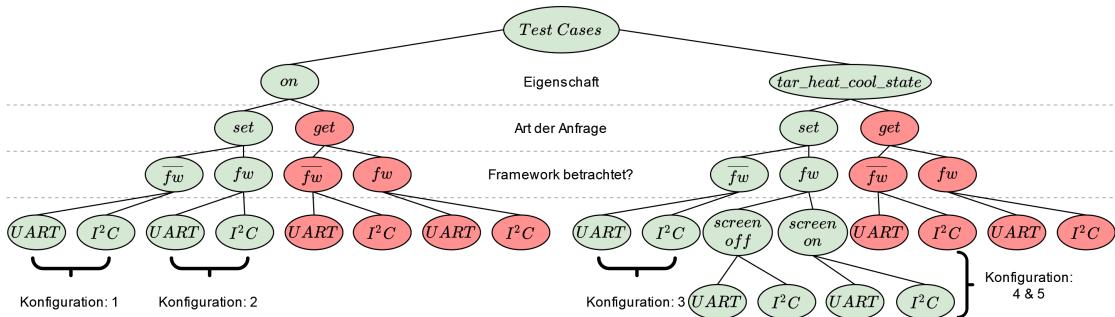


Abbildung 7.1: Gemessene Konfigurationen

Um die Messungen miteinander vergleichen zu können und weitere Erkenntnisse über die Ursachen möglicher Verzögerungen zu gewinnen, werden also unterschiedliche Messkonfigurationen verwendet. Dabei wird zusätzlich zwischen dem Senden von *Set*- und *Get*-Anfragen unterschieden. Zusammen mit den variablen Längen der Nachrichten ergeben sich die in Abbildung 7.1 nachvollziehbaren Konfigurationen.

Die Konfigurationen sind in einem Binary Tree angeordnet. Jede Abzweigung steht für die Nutzung eines bestimmten Konfigurationsparameters. In jeder Konfiguration werden die beiden Varianten UART und I²C miteinander verglichen. Um konstante Faktoren wie die Verarbeitungszeit des Frameworks zu erfassen und abzugrenzen, können die einzelnen Konfigurationen mit demselben Übertragungsmedium betrachtet werden. Zum Messen einer Konfiguration werden 25 Messungen durchgeführt.

Da *Get*-Anfragen in einem normalen Programmablauf nicht als einzelne Anfragen vorkommen, werden sie nicht ausführlich untersucht. Sie können nicht von einem Nutzer gesendet werden. Wichtig wird die Verzögerung dann, wenn ein Nutzer eine Zustandsänderung in der Home App anordnet. Diese werden als *Set*-Anfragen an das Device

weitergeleitet. *Get*-Anfragen werden an das Device gesendet, wenn eine Fehlerbehandlung eingeleitet wurde und der Zustand des Device neu synchronisiert wird. Da in diesem Fall alle Eigenschaften des Device nacheinander angefragt werden, wird der Neuaufbau als ein Ausnahmefall betrachtet und separat gemessen.

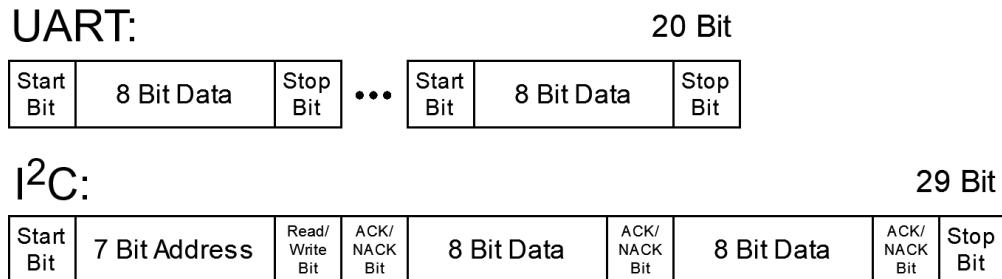
Wird die Länge der Pakete betrachtet, dann unterscheiden sich *Get* und *Set* durch 2-3 zusätzliche Byte zum Übermitteln des veränderten Wertes (*Set*). Sofern die Veränderung nicht dazu führt, dass eines der beiden Varianten weniger Pakete zum Versenden einer Nachricht benötigt, sollte keine veränderte Verzögerung aus der Übertragung festzustellen sein. Die *Get*-Anfragen können in Einzelfällen dazu genutzt werden, die Unterschiede in der Verarbeitungszeit zwischen *Get* und *Set* herauszustellen.

7.1.2 Erwartungen

Die Übertragungsrate der beiden Übertragungsmedien beträgt 1 Mbit/s. Das betrifft lediglich die Geschwindigkeit des Datenaustauschs über das Übertragungsmedium. Eine Nachricht mit einer Größe von 32 Byte benötigt so eine theoretische Übertragungsdauer von 0,256 ms. Es ist jedoch nicht zu erwarten, dass sich diese Geschwindigkeit auf die Schnittstellen innerhalb der beiden Geräte übertragen lässt. Je nach der Implementierung der internen Treiber wird eine Übertragung länger dauern als durch die Übertragungsrate vorgegeben.

Ein wichtiger Faktor in der Betrachtung dieser Arbeit ist besonders der zusätzliche Aufwand eines weiteren Protokolls, welches als eine zusätzliche Schicht über den Übertragungsprotokollen verwendet wird. Eine Nachricht wird als eine Anfrage auf der Anwendungsebene verpackt, in eine Byte-Sequenz formatiert und an das jeweilige Übertragungsmedium weitergegeben. Der implementierte Treiber versendet diese Nachricht anschließend dem Protokoll entsprechend mit zusätzlichen Informationen über den Übertragungskanal. Dort angekommen, wird sie von dem Treiber entpackt und muss zusätzlich auf der Anwendungsebene wieder in eine valide Nachricht umgewandelt werden. Aus diesem Grund ist zu erwarten, dass die Verzögerung, unabhängig von der durchzuführenden Aufgabe, mehrere Millisekunden dauern kann.

Es ist zudem zu erwarten, dass die zusätzlich eingeführte Asynchronität bei der Nutzung von I²C eine gewisse Verzögerung nach sich ziehen wird. Das Abhören der Notification-Line ist auf Seiten des Smart durch eine ISR realisiert. Wird ein Signal empfangen, dann wird ein Thread gestartet, der sich um das Abholen der Antwort beim Device kümmert. Diese Abfolge unterliegt der Ausführungsreihenfolge des internen Scheduler des Betriebssystems. Abhängig von der aktuellen Last des Smart kann hier die Verzögerung stark abweichen. Wie stark sich die Verzögerung auf die allgemeine Kommunikation auswirkt, ist nicht abzuschätzen. Sie sollte allerdings nicht durch die Größe der Nachrichten beeinflusst werden und abgesehen von den Abweichungen überwiegend konstant sein.


 Abbildung 7.2: Vergleich: Pakete von UART und I²C

Werden die unterschiedlichen Übertragungsmedien betrachtet, können bereits Aussagen bezüglich der Länge der einzelnen Nachrichten getroffen werden. Dabei ist es hilfreich sich die einzelnen Pakete der Protokolle anzusehen und diese zu vergleichen. In Abbildung 7.2 ist der Aufbau der Pakete von beiden seriellen Protokollen abgebildet. Die Pakete wurden nach den in dieser Arbeit verwendeten Konfigurationen aufgebaut.

Bei beiden Protokollen wird die Länge der Pakete verglichen, wenn zwei Bytes an Daten gesendet werden sollen. Durch die asynchrone Funktionsweise von UART, muss jedes Paket unabhängig voneinander gesendet werden. Vor und nach jedem zu sendenden Byte wird ein Start sowie eine Stop-Bit eingefügt. Ein Paket ist 10 Bit lang. So können die einzelnen Pakete voneinander abgegrenzt werden.

I²C hingegen bietet die Möglichkeit, mehrere zusammenhängende Daten (Byte) in einer Übertragung zu versenden. Bei aufeinanderfolgenden Bytes an Daten sparen sie so ein unnötiges Stop- und Start-Bit. Getrennt werden die einzelnen Pakete durch ACK-Bits vonseiten des Empfängers.

Ein großer Nachteil gegenüber UART ist jedoch die Notwendigkeit, die Adresse des Empfängers in jede Übertragung miteinzubeziehen. Unabhängig von der Länge der Daten entsteht dadurch für jede Übertragung ein zusätzlicher konstanter Overhead von 7 oder 10 Bit. In dieser Arbeit wurde eine Adressierung von 7 Bit gewählt.

Wird die Gesamtgröße einer Nachricht gesucht, ist die Anzahl an zu sendenden Daten-Bytes ein entscheidender Faktor.

Abhängig von der Anzahl an zu sendenden Bytes ergibt sich für die Länge einer Nachricht über UART die folgende Formel:

$$f(x) = 10x. \quad (7.1)$$

Für I²C ergibt sich die Formel:

$$f(x) = 9x + 11. \quad (7.2)$$

Wobei „x“ die Anzahl an Daten-Bytes und „f(x)“ die Länge der Nachricht in Bit angibt. Es fällt auf, dass der statische Overhead von I²C viel Platz einnimmt. Auch bei einer niedrigen Anzahl an Daten-Bytes wird ein hoher Overhead erzeugt. Wird das Senden

eines einzelnen Bytes an Daten betrachtet, ergibt sich bei der Nutzung von I²C beispielsweise eine Nachrichtengröße von 20 Bit. Bei UART entsteht durch den Overhead von 2 Bit für jedes gesendete Byte lediglich eine Nachrichtengröße von 10 Bit.

Das Verhältnis zwischen dem Overhead und den richtigen Daten wird in dieser Arbeit als Effizienz bezeichnet. Eine Nachricht ist effizienter als eine andere Nachricht, wenn für die Übertragung pro Byte an Daten weniger Overhead notwendig ist. Bei der Betrachtung der beiden Formeln (7.1) und (7.2) wird schnell ersichtlich, dass die Effizienz im Falle von I²C anfangs sehr niedrig ist. Da bei I²C pro Byte jedoch weniger Overhead anfällt als bei UART, wird die Effizienz bei einer steigenden Anzahl an Bytes höher. Um die Effizienz anhand der Formeln zu berechnen, müssen diese zunächst so umgeformt werden, dass sie die Anzahl an gesendeten Bits für ein Byte repräsentieren. Anschließend wird dieser Wert ins Verhältnis zu der Länge eines Bytes gesetzt.

Für UART ergibt sich so die Effizienz:

$$8 : \frac{10x}{x} = \frac{8}{10}. \quad (7.3)$$

Für I²C ergibt sich die Effizienz:

$$8 : \frac{9x + 11}{x} = \frac{8}{9 + \frac{11}{x}}. \quad (7.4)$$

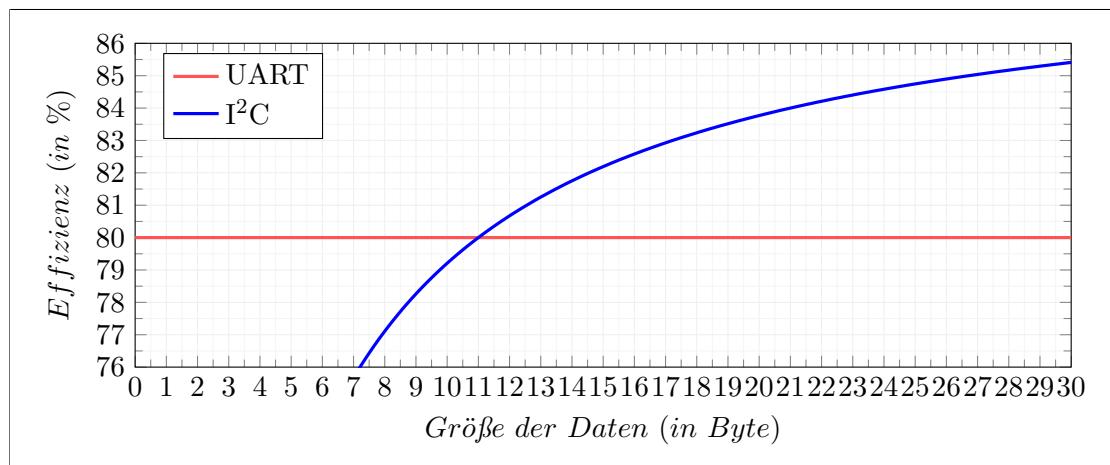


Abbildung 7.3: Effizienz der gesendeten Nachrichten nach Größe

Es ist zu erkennen, dass die Effizienz von UART konstant bei 80 % ist. Das bedeutet, dass sie sich durch die Nachrichtengröße nicht verändert. Das ist nachvollziehbar, da jedes Paket einzeln mit 2 Bits an Header-Informationen versendet wird. Der Verlauf der

Effizienz in Prozent ist in Abbildung 7.3 anschaulich dargestellt.

Wie zuvor bereits erwähnt, fällt auf, dass die Effizienz eines UART-Paketes bis zu einer Nachrichtengröße von 11 Byte höher ist als die Effizienz eines I²C Paketes. Bei einem Byte an Daten liegt die Effizienz von I²C bei 40 %. Ab einer Größe von 12 Byte übersteigt die Effizienz jedoch die von UART. Das bedeutet, dass ab diesem Punkt bei I²C pro gesendetem Byte mehr eigentliche Daten gesendet werden als es bei UART der Fall ist. Wird der Grenzwert der Effizienz von I²C betrachtet, nähert sie sich einer maximalen Effizienz von etwa 88,89 % an. Die Übertragung größerer Datenmengen mit I²C wird somit geringfügig schneller sein.

Da eine Nachricht in dieser Arbeit mindestens 14 Byte lang ist, wird für das Versenden dieser Nachricht mit I²C im Durchschnitt weniger Zeichen benötigt als mit UART. Über die exakte Verzögerung kann im Voraus keine direkte Aussage getroffen werden. Durch die oben genannten Informationen kann jedoch ein Geschwindigkeits-Unterschied zwischen den beiden Varianten abgeschätzt werden.

Die nachfolgende Gleichung beschäftigt sich mit dem prozentualen Unterschied der Anzahl an benötigten Bits zwischen I²C und UART in Abhängigkeit von der Nachrichtengröße. Dabei werden die Nachrichtengrößen 14 und 31 Byte betrachtet (Minimum/Maximum).

Normalisierter Abstand zwischen I²C und UART bei 14 Byte Nachrichtengröße:

$$\frac{10 * 14}{14} - \frac{9 * 14 + 11}{14} = \frac{3}{14} \approx 0,21 \text{ Bit.} \quad (7.5)$$

Normalisierter Abstand zwischen I²C und UART bei 31 Byte Nachrichtengröße:

$$\frac{10 * 31}{31} - \frac{9 * 31 + 11}{31} = \frac{20}{31} \approx 0,65 \text{ Bit.} \quad (7.6)$$

Unterschied der normalisierten Abstände:

$$\frac{20}{31} : \frac{3}{14} \approx 3.01. \quad (7.7)$$

Aus diesen Gleichungen kann geschlossen werden, dass der normalisierte Abstand zwischen I²C und UART bei einer Nachrichtengröße von 31 Byte um etwa das Dreifache größer ist als bei einer Nachrichtengröße von 14 Byte. Es ist also zu erwarten, dass die konstante Verzögerung, die durch die Notification-Line entsteht, durch eine steigende Nachrichtengröße verringert bzw. ausgeglichen wird. Die Verzögerung zwischen den Messungen mit 14 und 31 Byte sollte sich pro Byte etwa um den Faktor 3 verringern. Da sie allerdings nicht nur durch die Übertragung, sondern auch durch Faktoren wie die Bearbeitungszeit und die Notification-Line beeinflusst wird, wird der Unterschied geringfügig kleiner sein.

Zusammenfassend sind also unter anderem folgende Dinge zu erwarten:

- Eine Kommunikation wird mehrere Millisekunden benötigen.
- Die Verzögerung wird bei steigender Nachrichtenlänge größer.
- Bei der Nutzung von I²C wird eine zusätzliche Verzögerung durch die Notification-Line hinzukommen.
- UART wird bei kurzen Nachrichten eine geringere Verzögerung aufweisen als I²C.
- Der Abstand zwischen den Verzögerungen von I²C und UART wird sich mit steigender Nachrichtenlänge verringern.

7.1.3 Durchführung

Bei jeder Messkonfiguration werden die beiden Übertragungsmedien miteinander verglichen. Die Reihenfolge der Messungen ist zufällig durchgeführt worden. Das bedeutet, dass Unterschiede in bestimmten Durchgängen nicht vergleichbar sind. Die erste Messung von I²C einer bestimmten Messkonfiguration hat keine direkte zeitliche Relation zu der ersten Messung von UART innerhalb der gleichen Messkonfiguration.

Unter jeder Konfiguration sind die Mittelwerte des gesamten Messdurchlaufs protokolliert. Ausführlich behandelt werden lediglich die *Set*-Anfragen. Für die *Get*-Anfragen derselben Eigenschaften wurden in mehreren Messdurchläufen keine signifikanten Abweichungen zu den Konfigurationen mit *Set*-Anfragen (ohne die Betrachtung des Frameworks) festgestellt.

7.1.4 Ergebnisse

Die Ergebnisse der Messungen sind in den Abbildungen 7.4 bis 7.6 in Form von Graphen anschaulich dargestellt.

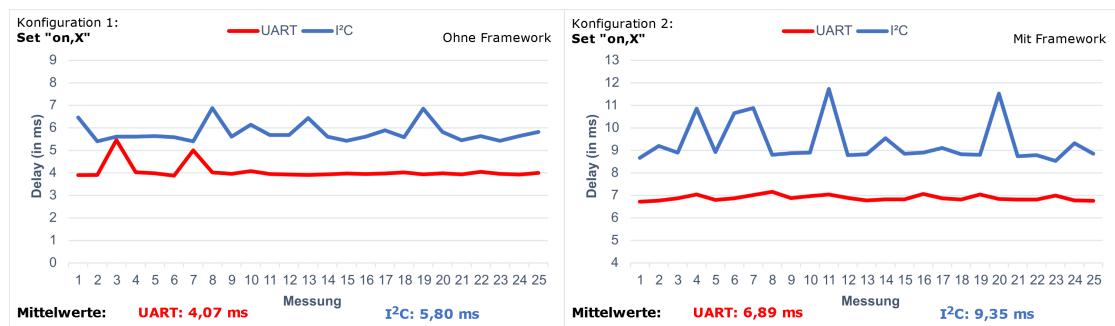


Abbildung 7.4: Messergebnisse (Konfiguration 1 und 2)

Aus den Messungen ist zu entnehmen, dass die Kommunikation über UART im Durchschnitt weniger Verzögerung aufweist als die Kommunikation über I²C. Diese Beobachtungen decken sich mit den zuvor genannten Erwartungen. Ein möglicher Grund dafür kann in der zusätzlich eingeführten Indirektion bei der Verwendung von I²C liegen.

Die kleinste gemessene Verzögerung, die durch das Weiterleiten der Anfragen entsteht, beträgt 3,88 ms. Diese Messung wurde über UART, durch eine *Set*-Anfrage an die Eigenschaft `on`, ohne die zusätzliche Betrachtung des Frameworks erreicht (Konfiguration 1, Abbildung 7.4). Die Größe einer gesendeten Nachricht beträgt 14 Byte in jede Richtung. Durch zwei fehlende Zeichen kann eine *Get*-Anfrage diese Verzögerung weiter unterbieten. Da *Get*-Anfragen jedoch nicht vom Nutzer getätigt werden können und keine signifikant niedrigeren Verzögerungen aufweisen als *Set*-Anfragen, werden sie hier nicht in die Betrachtung miteinbezogen.

Die größte gemessene Verzögerung unter der Betrachtung aller Konfigurationen beträgt 19,711 ms. Sie wird lediglich in einem Ausnahmefall erreicht. Dabei wurde eine *Set*-Anfrage an die Eigenschaft `tar_heat_cool_state` über I²C gesendet. Wird die Bearbeitungszeit des Frameworks miteinbezogen, können zusätzliche Abweichungen festgestellt werden. Das Ergebnis dieser Messungen ist in Abbildung 7.6 nachzuvollziehen. Diese Abweichungen sind abhängig davon, ob die Home App geöffnet oder geschlossen ist. Die Unterscheidung zeigt inkonsistente Messungen und wird aus diesem Grund als Sonderfall betrachtet. Der Sonderfall ist nur bei dem Thermostat zu beobachten. Ohne die Betrachtung des Frameworks ist bei I²C die maximale Verzögerung von 14,329 ms (Ausreißer) zu beobachten. Diese Verzögerung kann der Abbildung 7.5 entnommen werden.

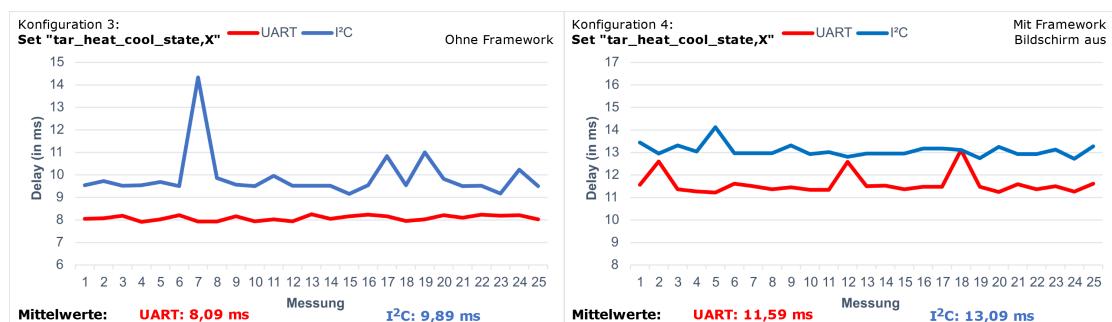


Abbildung 7.5: Messergebnisse (Konfiguration 3 und 4)

Bei der Betrachtung der Messungen fällt auf, dass ein großer Unterschied zwischen I²C und UART in der Konsistenz der Verzögerung liegt. Während UART weitgehend konstante Werte aufweist, ist bei I²C eine stärkere Varianz zu erkennen. Der Grund für diese Varianz kann ebenfalls in der Notification-Line liegen. Die Verarbeitung der Notification-Line ist aufseiten des Smart mit einer ISR und dem Starten eines neuen Threads verbunden. Je nachdem, an welchem Ausführungszeitpunkt sich das Smart befindet, kann

es also sein, dass zuvor andere Threads priorisiert werden.

Werden die Messungen nur durch die Länge der Nachrichten unterschieden, fällt auf, dass die Verzögerung in etwa mit dieser Länge korreliert. Wird also beispielsweise die UART-Konfiguration 1 (Abbildung 7.4) mit der UART-Konfiguration 3 (Abbildung 7.5) verglichen, unterscheiden sie sich größtenteils in ihrer Länge. Der Unterschied dieser Messungen liegt mit einem Faktor von etwa 1,99 unter dem Faktor 2,21, der sich nur aus der Betrachtung der Längen 14 und 31 ergibt. Diese Abweichung kommt zustande, weil in beiden Messungen die zusätzlichen konstanten Verzögerungen miteinbezogen werden, die durch weitere Verarbeitungsschritte entstehen. Das wird besonders dann deutlicher, wenn weitere Messungen miteinander verglichen werden, bei denen dieser zeitliche Overhead größer ist. Bei der zusätzlichen Betrachtung des Frameworks (UART-Konfiguration 2 und 4) ergibt sich lediglich ein Faktor von etwa 1.68. Aus diesen Beobachtungen kann also der theoretische zeitliche Einfluss des konstanten Aufwands abgeschätzt werden.

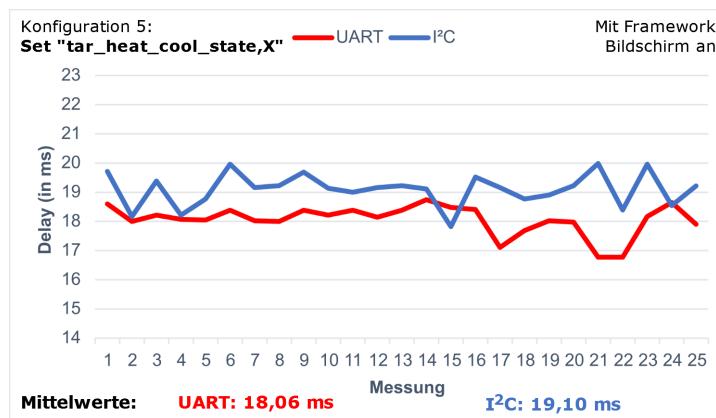


Abbildung 7.6: Messergebnis (Konfiguration 5)

Wie zuvor erwartet, lässt sich außerdem klar erkennen, dass der Unterschied zwischen beiden Varianten mit zunehmender Nachrichtenlänge kleiner wird. Dadurch, dass die Skalierung der Graphen gleich gewählt wurde, ist das mithilfe der Graphen schnell zu erfassen.

Für eine genauere Aussage bezüglich des Abstandes der Verzögerungen werden sich im Folgenden die Mittelwerte angesehen. Da die Verzögerung des Frameworks in etwa konstant sein sollte, werden für die Vergleichbarkeit beide Varianten miteinbezogen. Die letzte Konfiguration wird dabei nicht betrachtet.

Für das Senden von `on` ergibt sich der durchschnittliche Abstand der Messungen von UART und I²C aus folgender Gleichung:

$$\frac{5,80 - 4,07 + 9,35 - 6,89}{2} = \frac{419}{200} = 2,095 \text{ ms.} \quad (7.8)$$

Für `tar_heat_cool_state` ergibt sich der durchschnittliche Abstand:

$$\frac{9,89 - 8,09 + 13,09 - 11,59}{2} = \frac{33}{20} = 1,65 \text{ ms.} \quad (7.9)$$

Um zu überprüfen, inwieweit sich der Abstand der Verzögerungen von UART und I²C pro gesendetem Byte verringert hat, werden diese miteinander ins Verhältnis gesetzt:

$$\frac{2,095}{14} : \frac{1,65}{31} \approx 2,81. \quad (7.10)$$

Dabei beziehen sich die Abstände auf die Messungen des gesamten Nachrichtenverlaufs inklusive Anfrage und Antwort. Für den Abstand einer einzelnen Nachricht kann dieser durch den Faktor 2 dividiert werden. Aus Gründen der Verständlichkeit wurde darauf verzichtet. Das Ergebnis verändert sich dadurch nicht.

Der Unterschied in der gemessenen Verzögerung beim Senden einer 14 Byte langen Nachricht im Vergleich zu 31 Byte hat sich somit etwa um das 2,81-Fache verringert. Das entspricht ungefähr den zuvor genannten Erwartungen an die Messungen. Durch die konstanten Faktoren ist dieser Unterschied geringer als bei der Betrachtung der Übertragungsmedien errechnet wurde. Da eine Nachrichtengröße von 31 Byte in dieser Anwendung das Maximum darstellt, ist sie die maximale Verringerung des Unterschiedes, der zwischen UART und I²C zu erreichen ist.

Die oben gemessenen Verzögerungen betrachten den normalen Nachrichtenaustausch zwischen Smart und Device. Kommt es jedoch zu einem Fehlerfall und der State des Device wird invalidiert, muss der gesamte State neu angefragt werden. Dabei werden viele Nachrichten hintereinander verschickt. Das kann einen zusätzlichen Faktor in der Betrachtung der Verzögerung ausmachen. Um eine qualifizierte Aussage über die Verzögerung der Fehlerbehandlung zu treffen, muss ihre Funktionsweise betrachtet werden.

Alle Eigenschaften eines Device sind in einer Liste gespeichert. Sie werden immer in derselben Reihenfolge angefragt. Zur Verringerung der Last aufseiten des Device wird jede Anfrage mit einem Abstand von 100 ms versendet. Abhängig davon, an welcher Stelle innerhalb der Liste sich die nicht synchronisierte Eigenschaft befindet, kann die Verzögerung daher stark variieren. Es ist möglich, dass der State nach der ersten oder erst nach der letzten empfangenen Antwort wieder korrekt ist. Aus diesem Grund ist es wichtig, diese Grenzfälle zu betrachten. Die Verarbeitung aufseiten des Device geschieht immer im FIFO Prinzip. Das bedeutet, es ist anzunehmen, dass die erste empfangene Nachricht als Erstes eine Antwort erhält. Die Grenzwerte werden daher jeweils durch die ersten und die letzten angefragten Eigenschaften repräsentiert.

Der Best Case (B) wird zuerst abgefragt. Bei einem Light Device wird dieser Best Case durch die Eigenschaft `brightness` erreicht. Der Worst Case (W) wird durch die Eigenschaft `on` erreicht. Sie liegt an vierter Stelle der Liste, die einzeln nacheinander bearbeitet wird. Bei einem Thermostat Device ergibt sich der Best Case durch `curr_temp` und der Worst Case durch `temp_display_units`. Der Worst Case bei dem Thermostat Device wird als siebte Eigenschaft angefragt. Die für diesen Zweck durchgeführten Messungen

wurden stichprobenartig durchgeführt.

Die Ergebnisse der Messungen sind in Tabelle 7.1 aufgeführt. Sie zeigen die durchschnittlichen gemessenen Verzögerungen zum Wiederaufbau des korrekten State. Da die Erkennung eines fehlerhaften State an einen 2-Sekunden-Timer gebunden ist, können diese Ergebnisse von der Praxis abweichen. Je nachdem, zu welchem Zeitpunkt der State invalidiert wird, können im schlimmsten Fall bis zu 2 Sekunden Verzögerung hinzukommen. Die beobachteten Ergebnisse decken sich mit den Erwartungen, die sich aus den vorheri-

Protokoll	Light		Thermostat	
	brightness (B)	on (W)	curr_temp (B)	temp_display_units (W)
UART	9,31 ms	307,04 ms	9,53 ms	610,52 ms
I ² C	13,98 ms	309,68 ms	14,21 ms	611,30 ms

Tabelle 7.1: Gemessene durchschnittliche Verzögerung des State-Neuaufbaus

gen Messungen ergeben. Da die erste Nachricht direkt versendet wird und die Antworten unabhängig von diesem Prozess verarbeitet werden, hat die Wartezeit im Best Case keinen Einfluss auf die beobachtete Gesamtverzögerung.

Bei der Betrachtung des Worst Case macht sich die zusätzliche Wartezeit zwischen dem Absenden der Nachrichten allerdings deutlich bemerkbar. Beim Light Device werden vier Eigenschaften angefragt. Durch den Abstand von 100 ms zwischen jeder Nachricht kommt also eine zusätzliche Verzögerung von 300 ms hinzu. Da beim Thermostat sieben Eigenschaften angefragt werden, ergibt sich eine zusätzliche Verzögerung von 600 ms. Weil in dieser Arbeit nicht auf die Antwort der Anfragen gewartet wird, hat die Verzögerung der einzelnen Anfragen keinen zusätzlichen Einfluss auf die Gesamtverzögerung. Die Anfragen werden asynchron versendet, ohne dabei die Antworten zu berücksichtigen. Das bedeutet, der einzige Faktor für die zusätzliche Verzögerung ist die eingebaute Wartezeit.

Es ist anzunehmen, dass der Arduino dazu in der Lage ist, diese Anfragen deutlich schneller zu verarbeiten. Aufgrund einer maximal zu erwartenden Verzögerung von etwa 20 ms, kann eine Wartezeit von 25-30 ms theoretisch ausreichend sein. Ob das auch in der Praxis umsetzbar ist, sollte vorher jedoch getestet werden. Die Anwendung bietet hier einen großen Optimierungsspielraum. Die Wartezeit von 100 ms wurde vor der Durchführung der Messungen gewählt, um die Funktionalität der Anwendung sicherzustellen. Auch diese verhältnismäßig lange Wartezeit ist in der Praxis verkraftbar. Im Worst Case sind die Zustände nach etwa einer halben Sekunde wieder synchronisiert.

Für einen Nutzer wirkt sich diese Verzögerung kaum aus, da er durch die fehlende Ausführung der Funktion bereits nach dem Senden der Anfrage merken wird, dass etwas nicht funktioniert hat. Verzögert wird in diesem Fall die Aktualisierung der Anzeige in der App des Nutzers. Nach wenigen Sekunden (inkl. Observer-Thread-Verzögerung) sollte sich die Anzeige an den korrekten Zustand angepasst haben.

Für die normale Ausführung der Anwendung ohne die Betrachtung des Fehlerfalls sind die gemessenen Verzögerungen nicht bemerkbar. Einem Nutzer würde ein Unterschied von 20 ms an zusätzlicher Verzögerung nicht auffallen. In der Praxis wirken sich die Zeittintervalle für die Kommunikation mit der Home App über HTTP deutlich stärker aus. Die Verzögerung für die drahtlose Übertragung variiert spürbar und kann im schlimmsten Fall sogar bis zu einer Sekunde betragen. Dieser Faktor ist jedoch nicht zu ändern und steht in keinem direkten Zusammenhang mit dem Vorhaben der Arbeit.

7.2 Diskussion

In dieser Arbeit wurden viele Designentscheidungen getroffen. Die wenigsten davon sind unumstritten als die Besten zu qualifizieren. Jede Entscheidung hat ihre Vor- und Nachteile. Aus diesem Grund werden diese Entscheidungen im Folgenden genauer betrachtet sowie Alternativen diskutiert, die mit den umgesetzten Designentscheidungen verglichen werden.

7.2.1 Digital Twin

Wie im Design-Kapitel ausführlich erläutert wurde, kann eine andere Struktur bei der Nutzung des Digital-Twin-Konzeptes die Funktionsweise der Anwendung stark verändern. Für ein Single-Smart-Setup ergeben sich zusätzliche Funktionen. So kann die Möglichkeit geschaffen werden, bestimmte Einstellungen in dem Smart zu speichern und bei Wechsel eines Device alle Einstellungen direkt bei dem neuen Device zu übernehmen. Dafür muss die Anwendung und ihr Addressing-Scheme nur geringfügig angepasst werden. Problematisch wird dieser Ansatz, sobald mehrere Smarts mit einem Device kommunizieren. Für diesen Zweck ist die Digital-Twin-Umsetzung, für die sich in dieser Arbeit entschieden wurde, besser geeignet, da die Wahrheit konsistent in dem Device gehalten wird und mit diesem synchronisiert wird.

Ein weiteres Problem dieses Ansatzes liegt auf der Hardware-Seite. Es ist schwierig sicherzustellen, dass alle Zustände aller Devices konsistent abgespeichert werden, wenn es beispielsweise zu Stromausfällen kommt. Der ESP32 besitzt lediglich eine unausgereifte Mechanik, die bei einem Spannungsabfall wichtige Registerwerte sichert. Bei komplexen Strukturen kann nicht gewährleistet werden, dass sie ebenfalls gesichert werden. In durchgeführten Tests war eine erfolgreiche Speicherung bereits einzelner Werte nicht möglich. Das könnte allerdings bei einem anderen Prozessor besser gelöst sein.

Die aktuell umgesetzte Variante dient demnach lediglich als Vermittler. Zusätzliche Funktionalitäten des Smart sollten weitestgehend zustandslos sein. Eine dieser Funktionalitäten wäre beispielsweise ein Logging. Das könnte über eine separate Output-Line abrufbar gemacht werden.

7.2.2 Ausgewählte Schnittstelle

Die in dieser Arbeit verwendeten Schnittstellen sind I²C und UART. Sie wurden ausgewählt, weil ihr Konfigurationsaufwand gering ist. Sie dienen als Beispiel für die Nutzung serieller Schnittstellen im genannten Anwendungskontext. I²C kann ebenso gegen SPI ausgetauscht werden. Eine Erweiterung ist durchaus denkbar.

Mithilfe von SPI können deutlich höhere Übertragungsgeschwindigkeiten erreicht werden als mit UART und I²C [19].

Wenn die Übertragung mithilfe einer asymmetrischen Technologie mindestens doppelt so schnell durchgeführt wird, kann die Verzögerung durch die Notification-Line ausgeglichen werden. Damit das ausschlaggebend für die Verzögerung der Anwendung sein kann, sollte aufseiten des Device ein anderer Mikrocontroller eingesetzt werden. Die Verarbeitungsgeschwindigkeit des Arduinos limitiert sowohl die Antwortzeit des Device als auch die maximalen Datenraten der seriellen Schnittstellen.

7.2.3 Erkennung eines Device

Die Erkennung eines Device erfolgt in dieser Arbeit durch eine spezifische Startup-Sequenz. Sie wird ausgeführt, sobald das Smart hochgefahren oder ein Device durch einen Timeout vom Smart abgemeldet wurde. Diese Sequenz wurde zeitlich begrenzt, damit das Smart nicht unnötig Pings an Schnittstellen sendet, die von einem Nutzer nicht verwendet werden. Das Smart ist für die Benachrichtigung der Devices zuständig, da es in dieser Umsetzung Strom von diesem erhält. Problematisch ist der Fall, wenn ein weiteres Device zur Laufzeit angeschlossen wird und die Startup-Sequenz bereits durchgeführt wurde. Der in dieser Arbeit vorgeschlagene manuelle Neustart kann für einen Nutzer lästig sein. Als Alternative ist eine Benachrichtigung durch das Device denkbar. Dazu muss das Device das Wissen über die Verbindung erlangen und speichern. In diesem Fall kann das Device seine seriellen Schnittstellen dauerhaft anpingen, solange es nicht mit einem Smart verbunden ist. Erhält es von dem Smart eine Anfrage für eine *Device-ID*, kann es mit dem Ping aufhören. Auch hier ist zu beachten, dass das Hinzufügen eines Smart auf einer anderen Schnittstelle zur Laufzeit entweder nicht möglich sein wird oder nicht verbundene Schnittstellen dauerhaft Pings erhalten werden. Aus Gründen der Einfachheit wurde sich für die zeitlich begrenzte Startup-Sequenz aufseiten des Smart entschieden. Es ist durchaus möglich, dass ein dauerhafter Ping auf einem seriellen Port dennoch eine gute Lösung ist, die im negativen Fall kaum Einfluss auf die Leistung des Smart hat. Dadurch kann die Notwendigkeit eines Neustarts bei Konfigurationsänderungen wegfallen.

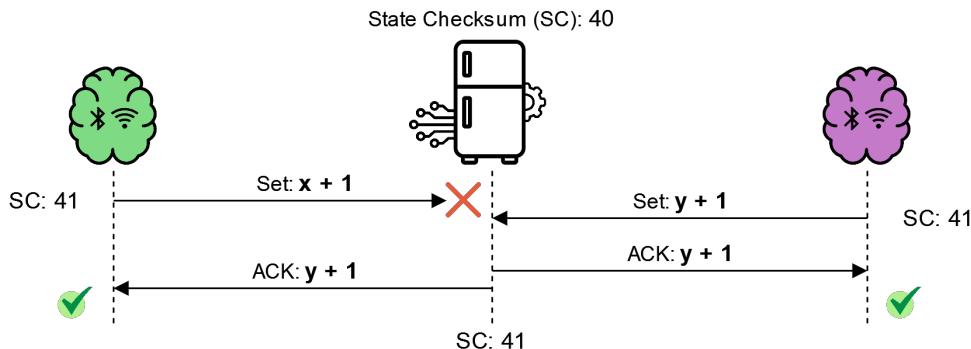


Abbildung 7.7: Sonderfall: State-Checksum korrekt, State fehlerhaft

7.2.4 Protokoll

Das entwickelte Protokoll ist nicht selbstbeschreibend. Das bedeutet, sobald sich etwas an dem Aufbau der Pakete oder dem Protokoll ändert, kann die Funktion nicht mehr gewährleistet werden.

Da jedes Zeichen eine feste Länge von 1 Byte besitzt, wäre es ausreichend, wenn die Selbstbeschreibung sich lediglich um die Beschreibung des Inhalts kümmern würde. Der Aufbau des Paketes könnte zu Beginn durch Bezeichner beschrieben werden (bspw. h|s|g). Kennt ein Device einen Bezeichner für ein Feld nicht, könnte es das Feld ignorieren.

Das Problem einer solchen Umsetzung besteht in dem großen Overhead für Fälle, bei denen sich der Aufbau der Pakete nicht verändert hat. Zudem benötigt jeder Client dieses Protokolls ein Mapping für die Bezeichner. Ändert sich das Mapping, so kann auch hier die Funktion nicht gewährleistet werden.

Aus diesem Grund ist es sinnvoll, das Protokoll so zu lassen, wie es in dem Kapitel „Design“ beschrieben wurde. Um Änderungen am Protokoll möglich zu machen, können stattdessen Versionsnummern zu Beginn der Kommunikation eingeführt werden. Da die Devices schwer zu updaten sind, sollte sich das Smart an die Version des Device anpassen.

Eine weitere Veränderung des Protokolls könnte vorgenommen werden, indem keine ganzen Namen, sondern einfache Bezeichner für die Eigenschaften der Devices verwendet werden. Dadurch wären die Nachrichten deutlich kürzer und würden eine statische Länge bekommen können. Das wäre umsetzbar, würde allerdings dazu führen, dass die Mappings auf den Smarts sehr unübersichtlich und schwer zu überblicken sind. Die aktuelle Umsetzung dient der Wartbarkeit und Lesbarkeit für Menschen. Durch die Messungen ist erkennbar, dass der Mehraufwand durch lange Nachrichten zu vernachlässigen ist. Die starke Verzögerung des Neuaufbaus kann anderweitig reduziert werden.

7.2.5 Fehlerbehandlung

Die Fehlerbehandlung, die in dieser Arbeit umgesetzt wurde, basiert auf einer Erkennung durch eine einfache 8-Bit Checksum. Bei der seriellen Datenübertragung ist diese Checksum durchaus ausreichend, da es durch die Serialität nicht möglich ist, dass Nachrichten in falscher Reihenfolge empfangen werden.

Bei der *State-C checksum* kann dieser Fall jedoch auftreten. Dazu müssen jedoch einige Gegebenheiten vorhanden sein.

In Abbildung 7.7 ist ein Beispiel dargestellt, wie sich der Wert einer anderen Eigenschaft ändern kann, sodass die *State-C checksum* dennoch den korrekten Wert anzeigt.

Die Anfrage des Smart muss komplett verloren gehen, ohne dass das Device nur einen Teil der Daten empfängt. Sobald ein Teil der Daten ankommt, wird das Device ein NACK als Antwort senden. Zusätzlich muss ein zweites Smart eine Anfrage mit einer Änderung senden, die den Zustand um denselben Betrag ändert. Diese Anfrage muss von dem Smart gesendet werden, noch bevor das erste Smart eine weitere Anfrage oder eine Keep-Alive Message an das Device sendet. Damit der beschriebene Fehlerfall also eintritt, muss ein zweites Smart innerhalb von 2 Sekunden eine weitere Anfrage senden. Da dies ein sehr spezieller Fall ist, wurde er als unwahrscheinlich eingestuft und rechtfertigt nicht den Mehraufwand einer Checksum, die auch die Reihenfolge der Werte mit einbezieht. Wenn dieser Fall behandelt werden soll, wäre die Umsetzung von Fletcher's Checksum oder einer CRC denkbar.

7.3 Fragestellungen

Die am Anfang dieser Arbeit vorgestellten Fragestellungen wurden innerhalb der einzelnen Kapitel ausgiebig behandelt. Welche Erkenntnisse aus der Bearbeitung der einzelnen Schritte gezogen werden können, ist im Folgenden aufgeführt:

1. In welchen Fällen ist eine Trennung sinnvoll?

Eine Auftrennung ist bei stationären Geräten sinnvoll, die ihren Standort nur selten ändern und bei denen eine physische Verbindung möglich ist. Großgeräte wie Heizungen oder Kühlschränke sind deutlich besser geeignet als Lampen, da diese meist in eigene Fassungen eingelassen werden und sich nicht leicht mit seriellen Schnittstellen ausrüsten lassen. Besonders einfach ist der Einsatz von Geräten, die bereits Mikrocontroller und Platinen mit seriellen Schnittstellen beinhalten. Der Mehraufwand ist hier gering. Die Verzögerung ist, unabhängig von dem verwendeten Gerät, in jedem normalen Anwendungsfall nicht bemerkbar.

2. Was muss bei der Entwicklung beachtet werden?

Wichtig sind besonders die Sicherstellung von Plug-And-Play, niedrigen Verzögerungen und einer funktionierenden Fehlerbehandlung.

3. Welche Möglichkeiten ergeben sich?

Zusätzliche Funktionen wie Logging oder die Speicherung von gewissen Konfigurationen sind einfach umzusetzen. Zudem kann ein Nutzer Geld sparen, indem er ein Smart für mehrere Devices verwendet.

4. Welche Limitierungen entstehen und welche Auswirkungen haben sie?

Aufseiten des Smart wird der Ansatz deutlich durch die Anzahl an physischen Schnittstellen limitiert. Werden alle genutzt, muss ein weiteres Smart hinzugefügt werden. Zudem ist es in einem praxisorientierten Szenario schwierig mehrere Geräte zu finden, die über einen seriellen Kanal mit demselben Gerät verbunden werden können. Das liegt daran, dass Schnittstellen wie UART oder I²C lediglich für kurze physische Verbindungen entwickelt wurden. Bei längeren Verbindungen wird die Fehlerrate steigen. Die in dieser Arbeit beschriebenen Anwendungsfälle (2Sx2D) werden daher eher als theoretische Szenarien betrachtet. Größere Daten wie Video-Streams sind mit dieser Trennung nur bedingt zu empfehlen. Dazu muss eine gewisse Übertragungsrate gewährleistet werden, die über die Möglichkeiten dieser Arbeit hinaus gehen. Mithilfe von SPI und einem anderen Device als einem Arduino ist dieser Anwendungsfall theoretisch umsetzbar. Für die Steuerung einer Remote-Cam wird eine direkte Verbindung mit einem Netzwerk jedoch weiterhin empfohlen.

5. Welches Übertragungsmedium ist am besten geeignet?

Die seriellen Schnittstellen, die in dieser Arbeit verwendet wurden, haben den Vorteil, dass sie von vielen Systemen bereits unterstützt werden. Für die Einrichtung ist wenig zusätzlicher Aufwand und wenig technisches Wissen nötig. Die Informationen können in das Protokoll eingebunden und dem Hersteller eines Device übermittelt werden. Es ist jedoch kein Leichtes, eine Antwort auf diese Frage zu geben. Die Intention dieser Arbeit bestand darin, eine einfache Struktur bereitzustellen, die simpel umsetzbar ist. Für diesen Zweck sind serielle Schnittstellen sehr gut geeignet. Im direkten Vergleich ist die asynchrone Arbeitsweise von UART für den Anwendungszweck am besten geeignet. Wird jedoch der zusätzliche Aufwand der Notification-Line und eine höhere Datenrate in Betracht gezogen, können auch asymmetrische Medien wie SPI oder I²C problemlos eingesetzt werden.

6. Welche Verbindungen der Module sind möglich und nützlich?

Es wurde gezeigt, dass es möglich ist mehrere Devices mit einem Smart zu verbinden. Die Anzahl der einzubindenden Devices in ein Smart wird am ehesten durch die Anzahl der seriellen Schnittstellen limitiert. In einem theoretischen Anwendungsfall ist die Einbindung von mehr als 30 Devices möglich. Für eine Bridge gibt das HAP [7] eine Maximalanzahl von 150 gleichzeitig verbundenen Accessories vor. Da diese Zahlen in der Praxis nicht erreicht werden sollten, sind diese Limitierungen hier nicht ausschlaggebend. Inwieweit die Funktionsfähigkeit durch das Smart bei dieser Menge an Geräten noch gewährleistet werden kann, wurde in dieser Arbeit nicht getestet.

Wird ein praxisnahes Szenario betrachtet, so wie es in der beschriebenen Anwendung der Fall ist, dann wird die Anzahl an seriell eingebundenen Geräten aufgrund der niedrigen Reichweite zwei bis drei Devices nicht überschreiten. Die ausführlich beschriebene Anwendung hat gezeigt, dass es unproblematisch für das Smart ist, zwei Devices gleichzeitig zu verwalten. Im Kontext der Smart-Home-Anwendungen ist der Nachrichtenverkehr nicht kontinuierlich vorhanden. Kommen mehrere Nachrichten hintereinander bei einem Device an, so wird die Geschwindigkeit eher durch ein Device limitiert.

Mehrere Smarts mit einem Device zu verbinden sollte in der Praxis sehr selten vorkommen, ist jedoch durchaus möglich. Wenn die Wahrheit in dem Device gehalten wird, ist die Konsistenz der Zustände mit geringem Aufwand zu gewährleisten. Bei der Verwendung dieser Variante ist die Anzahl an zusätzlichen Features jedoch stark begrenzt.

7. Welche Optimierungen dieses Ansatzes sind denkbar?

Da die entwickelte Anwendung lediglich einen Prototyp darstellt, mussten einige Designentscheidungen getroffen werden. Der Ansatz ist nicht fehlerfrei und hat besonders bei der Fehlerbehandlung Schwächen. Sowohl die Geschwindigkeit als auch die Zuverlässigkeit der Fehlerbehandlung kann weiter optimiert werden. Die Einführung weiterer Kommunikationsschnittstellen und die Verringerung des Protokoll-Overheads bieten gute Ansätze für Optimierungen.

Kapitel 8

Fazit

Das Ziel dieser Arbeit war es, Herstellern eine Möglichkeit zu bieten, möglichst einfach und ohne viel Erfahrung in den Markt der Smart Devices einzusteigen. Die Grundidee dabei war, die Funktionalitäten eines Smart Device auf die zwei separaten Geräte ‚Smart‘ und ‚Device‘ aufzuteilen. Der Hersteller eines Device kann sich auf seine Kernfunktionen konzentrieren. Um die Entwicklung des Gerätes mit den *smarten* Eigenschaften kann sich ein Hersteller kümmern, der über ausreichend Erfahrung in diesem Gebiet verfügt. Als die größten Probleme von heutigen Smart Devices wurden die *Interoperability* und die Heterogenität vieler Smart-Home-Systeme ausgemacht. Dadurch werden zusätzliche Faktoren wie die *Maintainability* für Hersteller erschwert.

Die Betrachtung einiger Forschungsarbeiten hat gute Ansätze hervorgebracht, die bei der Erstellung dieser Arbeit geholfen haben. In den Forschungsarbeiten lag der Fokus zuvor auf abgegrenzten Systemen wie Laboren, bei denen versucht wurde viele unterschiedliche Geräte zentral zu verwalten. Einige in diesen Ansätzen erschaffenen Ideen wie die Nutzung des Digital-Twin-Paradigmas waren auch für das geplante Vorhaben sehr hilfreich. Da diese Umsetzungen jedoch darauf basieren, neue Systeme für die Verwaltung der Devices zu entwickeln, mussten zusätzlich andere Arbeiten betrachtet werden. Der Homebridge-Ansatz verfolgte zwar eine sehr ähnliche Idee, limitiert sich jedoch durch die Nutzung eines Frameworks und richtet sich an die Anwender, anstatt die Hersteller. Inspiriert durch die Vorarbeit und die genannten Ansätze, wurde das Konzept für die Auftrennung eines Smart Device weiter ausgebaut. Um den strukturellen Rahmen der Arbeit abzugrenzen, wurden Fragestellungen und Anforderungen gesammelt, die als Leitfaden für das geplante Vorhaben dienen sollten. Dank der Anforderungen und der exakten Formulierung von Fragestellungen konnte anschließend das Design für einen Prototyp entwickelt werden.

Der Fokus lag in diesem Fall auf der Entwicklung eines geeigneten Übertragungsprotokolls und der Sicherstellung einer optimalen Kommunikation zwischen den neuerdings getrennten Geräten ‚Smart‘ und ‚Device‘.

Die in diesem Schritt gewonnenen Erkenntnissen haben bei der Entwicklung eines Prototyps geholfen. Der Prototyp exploriert die Auftrennung von Smart und Device in unterschiedliche Komponenten. Um möglichst viele Informationen über die Limitierungen und Möglichkeiten dieses Ansatzes zu erhalten, wurden sowohl zwei Smarts als auch

zwei Devices für diesen Prototyp entwickelt. Da das Übertragungsmedium ein wichtiger Faktor in der Kommunikation der Geräte darstellt, wurden zudem unterschiedliche Übertragungsmedien getestet.

In einem abschließenden Schritt wurde der entwickelte Prototyp anhand durchgeföhrter Messungen sowie den zugrundeliegenden Designentscheidungen analysiert. Die Ergebnisse der Messungen zeigten, dass die eingeföhrte Trennung keine signifikante zusätzlichen Verzögerung erzeugt. Abhängig von der Länge der von dem Smart weitergeleiteten Nachrichten übersteigt ein Nachrichtenaustausch in keiner Messung eine Verzögerung von 20 ms. Lediglich bei der Fehlerbehandlung war eine bemerkbare Verzögerung festzustellen. Es ist anzunehmen, dass die Umsetzung in diesem Fall durch kürzere Wartezeiten verbessert werden kann. In Bezug auf den Nutzen der Anwendung hat diese Verzögerung jedoch nur eine bedingte Aussagekraft. Die Fehlerbehandlung wird nicht aktiv von einem Nutzer eingeleitet. Sie stört die *Responsiveness* der Anwendung in seltenen Fällen und wird vom Nutzer möglicherweise nicht bemerkt.

Durch diese Arbeit wurde also gezeigt, dass eine Auf trennung eines Smart Device in ein ‚Smart‘ und ‚Device‘ einen validen Schritt in Richtung *Interoperability* bieten kann. Ein ausreichend leistungsstarkes Smart ist in der Lage mehrere unterschiedliche Devices zu verwalten. Serielle Schnittstellen sind durch ihre begrenzte Reichweite nicht perfekt, aber bieten deutlich Vorteile durch ihre simple Protokoll-Struktur. Die Verzögerung der einzelnen Ausführungsschritte ist zu vernachlässigen und der Einrichtungsaufwand neuer Devices wird sogar reduziert. Es wurde außerdem gezeigt, dass es mit einem ausgearbeiteten Replication Scheme möglich ist, mehrere Smarts mit einem Device kommunizieren zu lassen.

8.1 Ausblick

Die in dieser Arbeit durchgeföhrte Trennung und die damit zusammenhängende entwickelte Anwendung dienen als Proof-of-Concept.

Wird der Ansatz durch die Integration klassischer Smart Devices erweitert oder das System als Homebridge-Plugin umgesetzt, kann die in dieser Arbeit entwickelte Anwendung in der Praxis eingesetzt werden. Dadurch können sowohl Geräte von Herstellern, die sich an die Spezifikationen halten als auch andere Geräte von dem Kunden gleichermaßen genutzt werden. Da die positionsbezogene Limitierung aufgelöst wird, kann ein Multi-Smart-Setup in diesem Fall über das theoretische Konzept hinausgehen. Zudem wäre die Integration weiterer Smart-Home-Systeme und die Unterstützung weiterer physischer Schnittstellen denkbar.

Es bleibt jedoch abzuwarten, welche Rolle die CSA in der Entwicklung von Smart Devices spielen wird und ob eine Umsetzung wie sie in dieser Arbeit vorgenommen wurde, auch nach der Einföhrung von Matter noch Sinn ergibt.

Abkürzungen

API Application Programming Interface

BLE Bluetooth Low Energy

CRC Cyclic Redundancy Check

ECC Error-Correction-Code

HAP HomeKit Accessory Protocol

HSB Hue-Saturation-Brightness

I²C Inter-Integrated Circuit

IoT Internet of Things

ISR Interrupt Service Routine

PWM Pulse-Width Modulation

QR-Code Quick-Response-Code

RFID Radio-Frequency Identification

SPI Serial Peripheral Interface

UART Universal Asynchronous Receiver Transmitter

Literatur

- [1] Monica Alleven. *Zigbee Alliance morphs into CSA, introduces Matter*. [Online; letzter Zugriff: 10.04.2022]. URL: <https://www.fiercewireless.com/tech/zigbee-alliance-morphs-into-csa-introduces-matter>.
- [2] Connectivity Standards Alliance. *Connectivity Standards Alliance Matter Update*. [Online; letzter Zugriff: 10.04.2022]. URL: <https://csa-iot.org/newsroom/matter-march-update/>.
- [3] Connectivity Standards Alliance. *Matter - The Foundation for Connected Things*. [Online; letzter Zugriff: 10.04.2022]. URL: <https://csa-iot.org/all-solutions/matter/>.
- [4] Francesco Amato, Hakki M. Torun und Gregory D. Durgin. “RFID Backscattering in Long-Range Scenarios”. In: *IEEE Transactions on Wireless Communications* 17.4 (2018), S. 2718–2725. DOI: 10.1109/TWC.2018.2801803.
- [5] Mahmoud Ammar, Giovanni Russello und Bruno Crispo. “Internet of Things: A survey on the security of IoT frameworks”. In: *Journal of Information Security and Applications* 38 (2018), S. 8–27. ISSN: 2214-2126.
- [6] Apple. *Home Automation*. [Online; letzter Zugriff: 14.03.2022]. URL: <https://support.apple.com/en-us/HT208940>.
- [7] Apple. *HomeKit Accessory Protocol Specification (Non-Commercial Version)*. [Online; letzter Zugriff: 14.03.2022]. URL: <https://developer.apple.com/homekit/specification/>.
- [8] Prof. Aderemi Atayero, Adeyemi Alatishe und Y Ivanov. “Power Line Communication Technologies: Modeling and Simulation of PRIME Physical Layer”. In: Okt. 2012, S. 931–936. DOI: 10.13140/RG.2.2.33300.50561.
- [9] Yi-yuan Fang und Xue-jun Chen. “Design and Simulation of UART Serial Communication Module Based on VHDL”. In: *2011 3rd International Workshop on Intelligent Systems and Applications*. 2011, S. 1–4. DOI: 10.1109/ISA.2011.5873448.
- [10] E. Ferro und F. Potorti. “Bluetooth and Wi-Fi wireless protocols: a survey and a comparison”. In: *IEEE Wireless Communications* 12.1 (2005), S. 12–26. DOI: 10.1109/MWC.2005.1404569.

- [11] Muhammad Rizwan Ghori, Tat-Chee Wan und Gian Chand Sodhy. “Bluetooth Low Energy 5 Mesh Based Hospital Communication Network (B5MBHCN)”. In: *Advances in Cyber Security*. Springer Singapore, 2020, S. 247–261. ISBN: 978-981-15-2693-0.
- [12] Carles Gomez, Joaquim Oller und Josep Paradells. “Overview and Evaluation of Bluetooth Low Energy: An Emerging Low-Power Wireless Technology”. In: *Sensors* 12.9 (2012), S. 11734–11753. ISSN: 1424-8220. DOI: 10.3390/s120911734.
- [13] Homebridge. *Homebridge API Wiki*. [Online; letzter Zugriff: 14.03.2022]. URL: <https://developers.homebridge.io/#/>.
- [14] Kasper Kamperman. *Arduino Programming – HSB to RGB*. [Online; letzter Zugriff: 24.03.2022]. URL: <https://www.kasperkamperman.com/blog/arduino/arduino-programming-hsb-to-rgb/>.
- [15] Evgeny Khorov u. a. “A Tutorial on IEEE 802.11ax High Efficiency WLANs”. In: *IEEE Communications Surveys Tutorials* 21.1 (2019), S. 197–216. DOI: 10.1109/COMST.2018.2871099.
- [16] KNX. *KNX BASICS*. [Online; letzter Zugriff: 14.03.2022]. URL: https://www.knx.org/wAssets/docs/downloads/Marketing/Flyers/KNX-Basics/KNX-Basics_en.pdf.
- [17] Jin-Shyan Lee, Yu-Wei Su und Chung-Chou Shen. “A Comparative Study of Wireless Protocols: Bluetooth, UWB, ZigBee, and Wi-Fi”. In: *IECON 2007 - 33rd Annual Conference of the IEEE Industrial Electronics Society*. 2007, S. 46–51. DOI: 10.1109/IECON.2007.4460126.
- [18] Woo Suk Lee und Seung Ho Hong. “KNX — ZigBee gateway for home automation”. In: *2008 IEEE International Conference on Automation Science and Engineering*. 2008, S. 750–755. DOI: 10.1109/COASE.2008.4626433.
- [19] Frederic Leens. “An introduction to I2C and SPI protocols”. In: *IEEE Instrumentation Measurement Magazine* 12.1 (2009), S. 8–13. DOI: 10.1109/MIM.2009.4762946.
- [20] Alexander Maier, Andrew Sharp und Yuriy Vagapov. “Comparative analysis and practical implementation of the ESP32 microcontroller module for the internet of things”. In: *2017 Internet Technologies and Applications (ITA)*. 2017, S. 143–148. DOI: 10.1109/ITECHA.2017.8101926.
- [21] Yu Song Meng und Yee Hui Lee. “INVESTIGATIONS OF FOLIAGE EFFECT ON MODERN WIRELESS COMMUNICATION SYSTEMS: A REVIEW”. In: *Progress in Electromagnetics Research-pier* 105 (2010), S. 313–332.
- [22] Roberto Minerva, Gyu Myoung Lee und Noel Crespi. “Digital Twin in the IoT Context: A Survey on Technical Features, Scenarios, and Architectural Models”. In: *Proceedings of the IEEE PP* (Juni 2020), S. 1–40. DOI: 10.1109/JPROC.2020.2998530.

- [23] NPM. *Homebridge Plugins*. [Online; letzter Zugriff: 14.03.2022]. URL: <https://www.npmjs.com/search?q=keywords%3Ahomebridge-plugin>.
- [24] Edward J. Oughton u. a. “Revisiting Wireless Internet Connectivity: 5G vs Wi-Fi 6”. In: *Telecommunications Policy* 45.5 (2021), S. 102127. ISSN: 0308-5961. DOI: <https://doi.org/10.1016/j.telpol.2021.102127>.
- [25] Stevan Preradovic, Nemai C. Karmakar und Isaac Balbin. “RFID Transponders”. In: *IEEE Microwave Magazine* 9.5 (2008), S. 90–103. DOI: [10.1109/MMM.2008.927637](https://doi.org/10.1109/MMM.2008.927637).
- [26] Jack Purdum. “Beginning C for Arduino”. In: *Apress*. 2015.
- [27] Qahhar Muhammad Qadir. “Analysis of the Reliability of LoRa”. In: *IEEE Communications Letters* 25.3 (2021), S. 1037–1040. DOI: [10.1109/LCOMM.2020.3034865](https://doi.org/10.1109/LCOMM.2020.3034865).
- [28] Christophe Salzmann u. a. “The Smart Device specification for remote labs”. In: *Proceedings of 2015 12th International Conference on Remote Engineering and Virtual Instrumentation (REV)*. 2015, S. 199–208. DOI: [10.1109/REV.2015.7087292](https://doi.org/10.1109/REV.2015.7087292).
- [29] S. Sujin Issac Samuel. “A review of connectivity challenges in IoT-smart home”. In: *2016 3rd MEC International Conference on Big Data and Smart City (ICBDSC)*. 2016, S. 1–4. DOI: [10.1109/ICBDSC.2016.7460395](https://doi.org/10.1109/ICBDSC.2016.7460395).
- [30] Manuel Silverio-Fernandez, Suresh Renukappa und Subashini Suresh. “What is a smart device? - a conceptualisation within the paradigm of the internet of things”. In: *Visualization in Engineering* 6 (2018), S. 1–10.
- [31] Ezequiel Simeoni u. a. “A Secure and Scalable Smart Home Gateway to Bridge Technology Fragmentation”. In: *Sensors* 21.11 (2021). DOI: [10.3390/s21113587](https://doi.org/10.3390/s21113587).
- [32] Web of Things. *Web of Things Documentation*. [Online; letzter Zugriff: 14.03.2022]. URL: <https://www.w3.org/WoT/documentation/>.
- [33] Craig Thompson. “Smart Devices and Soft Controllers”. In: *Internet Computing, IEEE* 9 (Feb. 2005), S. 82–85. DOI: [10.1109/MIC.2005.22](https://doi.org/10.1109/MIC.2005.22).
- [34] Paolo Visconti u. a. “Features, operation principle and limits of SPI and I2C communication protocols for smart objects: A novel spi-based hybrid protocol especially suitable for IoT applications”. In: *International Journal on Smart Sensing and Intelligent Systems* 10 (Juni 2017), S. 262–295. DOI: [10.21307/ijssis-2017-211](https://doi.org/10.21307/ijssis-2017-211).
- [35] RF Wireless World. *Bluetooth 5.0 vs 4.2-Difference between Bluetooth 5.0 and 4.2*. [Online; letzter Zugriff: 14.03.2022]. URL: <https://www.rfwireless-world.com/Terminology/Bluetooth-5-vs-bluetooth-4-2.html>.
- [36] RF Wireless World. *Bluetooth vs BLE-difference between Bluetooth and BLE (Bluetooth Low Energy)*. [Online; letzter Zugriff: 14.03.2022]. URL: <https://www.rfwireless-world.com/Terminology/Bluetooth-vs-BLE.html>.

Weitere Quellen

Für einige Abbildungen in dieser Arbeit wurden Symbole von Flaticon.com verwendet.
Symbole von folgenden Autoren wurden genutzt:

- Freepik (Brain, House, Freezer, Settings, Heartbeat)
Nutzung: Abb. 5.1, 5.3, 5.7, 5.8, 5.9, 5.10, 5.11, 5.12, 6.13, 6.14, 7.7
- pojok d (Refrigerator)
Nutzung: Abb. 5.1, 5.7, 5.10, 5.11, 5.12, 7.7
- Good Ware (Open Fridge)
Nutzung: Abb. 5.2
- Kirill Kazachek (Dots)
Nutzung: Abb. 5.8, 5.12
- Stockio (Database)
Nutzung: Abb. 5.3, 6.14
- iconixar (Thermostat)
Nutzung: Abb. 5.8, 5.9, 6.13, 6.14
- Andrejs Kirma (Neuron)
Nutzung: Abb. 5.1, 5.7, 5.8, 5.9, 5.11, 5.12, 6.13, 6.14, 7.7
- netscript (Brain)
Nutzung: Abb. 5.1
- Smashicons (Link, Lightning)
Nutzung: Abb. 5.11, 5.12, 6.14
- Jane_Kelly (Coffee Machine)
Nutzung: Abb. 5.1
- Octopocto (Check Mark)
Nutzung: Abb. 5.10, 5.11, 7.7
- Pixel perfect (Touch, Close)
Nutzung: Abb. 5.9, 5.10, 6.12, 7.7
- Smartline (Timer)
Nutzung: Abb. 6.14
- SBTS2018 (Settings)
Nutzung: Abb. 6.14

Versicherung an Eides Statt

Ich versichere an Eides statt durch meine untenstehende Unterschrift,

- dass ich die vorliegende Arbeit - mit Ausnahme der Anleitung durch die Betreuer
- selbstständig ohne fremde Hilfe angefertigt habe und
- dass ich alle Stellen, die wörtlich oder annähernd wörtlich aus fremden Quellen entnommen sind, entsprechend als Zitate gekennzeichnet habe und
- dass ich ausschließlich die angegebenen Quellen (Literatur, Internetseiten, sonstige Hilfsmittel) verwendet habe und
- dass ich alle entsprechenden Angaben nach bestem Wissen und Gewissen vorgenommen habe, dass sie der Wahrheit entsprechen und dass ich nichts verschwiegen habe.

Mir ist bekannt, dass eine falsche Versicherung an Eides Statt nach §156 und §163 Abs. 1 des Strafgesetzbuches mit Freiheitsstrafe oder Geldstrafe bestraft wird.

Duisburg, 21. April 2022

(Ort, Datum)

(Vorname Nachname)