HAW HAMBURG

BACHELOR THESIS
Maik Lorenz

Portierung des Smart-Home-Standards Matter nach RIOT OS

FAKULTÄT TECHNIK UND INFORMATIK
Department Informatik

Faculty of Engineering and Computer Science
Department Computer Science

HOCHSCHULE FÜR ANGEWANDTE WISSENSCHAFTEN HAMBURG
Hamburg University of Applied Sciences

Maik Lorenz

Portierung des Smart-Home-Standards Matter nach RIOT OS

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung
im Studiengang *Bachelor of Science Informatik Technischer Systeme*
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Thomas C. Schmidt
Zweitgutachter: Prof. Dr. Jan Sudeikat

Eingereicht am: 07.05.2024

Maik Lorenz

Thema der Arbeit

Portierung des Smart-Home-Standards Matter nach RIOT OS

Stichworte

RIOT OS, Matter, Smart Home, IoT

Kurzzusammenfassung

Mit dem Matter Standard (ehemals “Connected Home over IP”) wurde ein auf IPv6 aufbauendes Kommunikationsprotokoll entwickelt, das zum Ziel hat die Interoperabilität zwischen Smart Home Geräten zu verbessern und die bestehende Fragmentierung in diesem Bereich zu reduzieren. Im Rahmen dieser Arbeit wird das Matter-Protokoll unter Verwendung einer externen Library nach RIOT OS portiert, um Anwendungsentwicklern die Möglichkeit zu geben, Matter-kompatible Smart Home Geräte auf unterschiedlicher Hardware zu entwickeln und dabei die Vorteile von RIOT OS im Bereich des Internet of Things (IoT) zu nutzen. Zur Demonstration der erfolgreichen Portierung wird eine Demo-Anwendung entwickelt. Auf Basis der während des Projektes gemachten Erfahrungen werden zudem Vorschläge gemacht, wie das RIOT OS Ökosystem erweitert werden kann, um Matter in Zukunft noch besser zu unterstützen.

Maik Lorenz

Title of Thesis

Porting the Smart-Home-Standard Matter to RIOT OS

Keywords

RIOT OS, Matter, Smart Home, IoT

Abstract

With the Matter Standard (formerly "Connected Home over IP"), a communication protocol based on IPv6 was developed with the goal to improve the interoperability between smart home devices and reducing the existing fragmentation in this area. As part of this work, the Matter protocol is ported to RIOT OS using an external library to provide application developers the opportunity to develop Matter-compatible smart home devices on various hardware while taking advantage of RIOT OS in the area of the IoT. A demo application will be developed to demonstrate the success of the porting. Based on the gained experiences during the project, suggestions are made on how the RIOT OS ecosystem can be expanded to better support Matter in the future.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungen

6LoWPAN IPv6 over Low power Wireless Personal Area Network.

ACL Access Control List.

AP Access Point.

API Application Programming Interface.

BLE Bluetooth Low Energy.

BTP Bluetooth Transport Protocol.

CASE Certificate Authenticated Session Establishment.

CD Certification Declaration.

CLI Command Line Interface.

CoAP Constrained Application Protocol.

DAC Device Attestation Certificate.

DNS-SD DNS Based Service Discovery.

DRBG Deterministic Random Bit Generator.

GNRC Generic Network Stack.

HWRNG Hardware Random Number Generator.

ICMPv6 Internet Control Message Protocol for IPv6.

IMP Interaction Model Protocol.

IoT Internet of Things.

LwIP Lightweight TCP/IP Stack.

mDNS Multicast DNS.

MIC Message Integrity Check.

MRP Message Reliability Protocol.

MTD Memory Technology Device.

MTU Maximum Transmission Unit.

NOC Node Operational Credentials.

PAI Product Attestation Intermediate.

PASE Passcode-Authenticated Session Establishment.

RNG Random Number Generator.

RTC Real Time Clock.

SAUL Sensor Actuator Uber Layer.

SoC System on a Chip.

TCP Transmission Control Protocol.

UDP User Datagram Protocol.

VFS Virtual File System Layer.

WLAN Wireless Local Area Network.

WPAN Wireless Personal Area Network.

1 Einleitung

Das Smart Home als Teilbereich des IoT beschreibt die Vernetzung von Produkten aus dem privaten Bereich mit dem Ziel der Gebäudeautomation, der Programmierung "intelligenter" Funktionen und Fernsteuerung (z. B. mit Smartphone Apps). Einige Beispiele sind LED-Leuchten, Schalter, Thermostate, Temperatursensoren, Jalousien, Saugroboter oder sogar Kühlschränke. Für die Vernetzung der Geräte im Heimnetzwerk werden typischerweise Ethernet (`IEEE 802.3`) und WLAN (`IEEE 802.11`), aber auch ZigBee, Z-Wave, Bluetooth, Bluetooth Low Energy (BLE) oder KNX (Feldbus) eingesetzt. Darüber hinaus existieren Kommunikationsprotokolle wie z. B. CoAP, MQTT oder REST, über die IP-basierte Geräte kommunizieren können. Um den Anforderungen von Geräten mit begrenzten Ressourcen (Embedded Systems) wie z. B. batteriebetriebene, portable Lichtschalter oder Temperatursensoren gerecht zu werden, definiert `IEEE 802.15.4` einen Standard für Wireless Personal Area Networks (WPANs). Auf diesem baut auch die ZigBee-Spezifikation auf, mit der bereits versucht wurde, eine einheitliche Sprache für Smart Home Geräte zu schaffen. Der große Nachteil ist jedoch, dass ZigBee-Geräte nicht IP-basiert sind und somit nicht direkt mit anderen Smart Home Geräten kommunizieren können, die z. B. Ethernet oder WLAN verwenden.

Durch die Etablierung vieler Smart Home Produkte unterschiedlicher Hersteller, die verschiedene Link-Layer Technologien und Kommunikationsprotokolle verwenden, ist das Problem der **Fragmentierung** entstanden. Zum aktuellen Stand der Technik werden zur Lösung dieses Problems meist **Steuerzentralen** (auch bezeichnet als Hubs oder Gateways) eingesetzt, welche eine zentralisierte Kontrolle aller lokal angebundenen Geräte sowie die Erreichbarkeit über das Internet ermöglichen. Bekannte Beispiele für Steuerzentralen sind z. B. die Philips Hue Bridge, Apple HomePod oder die Open Source Software Home Assistant. Ein weiteres Problem besteht, wenn die Kommunikation zwischen Smart Home Geräten über die Server der Hersteller läuft, was neben Datenschutz-Bedenken auch zum Nachteil hat, dass die eigenen Geräte bei Ausfall der Server nicht mehr steuerbar sind [3].

Mit zunehmender Verbreitung des auf `IEEE 802.15.4` aufbauenden 6LoWPAN können IoT-Geräte mit begrenzten Ressourcen über IPv6 mit Ethernet- und WLAN-Geräten kommunizieren, was eine Interoperabilität auf dem *Internet Layer* (vgl. TCP/IP Modell) schafft. Um die zuvor beschriebenen Probleme im Bereich Smart Home zu lösen, wird jedoch auch eine einheitliche Sprache benötigt, die festlegt, wie die Geräte untereinander kommunizieren müssen. Hier gibt es bereits viele geeignete Kommunikationsprotokolle, allerdings scheinbar nicht genug Anreize für die Hersteller, sich auf ein Protokoll zu einigen.

Unter dem Dach der *Connectivity Standards Alliance* [4] (hervorgegangen aus der *ZigBee Alliance*) wurde das Projekt **Matter** ins Leben gerufen, welches dieses Problem durch Spezifikation eines Kommunikationsprotokolls auf Basis von IPv6 und den Link-Layer Technologien Ethernet, WLAN und 802.15.4 mit Thread [5] lösen möchte. Für die Inbetriebnahme neuer Geräte mit drahtloser Netzwerkschnittstelle wird BLE eingesetzt und ein darauf aufbauendes Bluetooth Transport Protocol (BTP) spezifiziert. Die Kommunikation Matter-fähiger Geräte kann innerhalb des lokalen Netzwerks erfolgen, was eine Unabhängigkeit zur öffentlichen Infrastruktur schafft.

1.1 Ziele der Arbeit

Das primäre Ziel dieser Arbeit ist die Erweiterung der unterstützten Hardware für die Entwicklung von Matter-fähigen Smart Home Geräten. Dafür wird das Open Source Betriebssystem RIOT OS verwendet, welches für den Betrieb auf IoT-Geräten ausgelegt ist und sich aufgrund seiner modularen Systemarchitektur, Netzwerkfähigkeiten, dem effizienten Ressourcenmanagement und vielen unterstützten Entwicklungsboards und CPU's sehr gut dafür eignet, die Erfüllung dieses Ziels zu unterstützen.

Zur Analyse der notwendigen Schritte für die Portierung von Matter nach RIOT OS wird eine Beispiel-Anwendung entwickelt, in der ein von der Matter-Spezifikation unterstützter Gerätetyp realisiert wird. Zur Evaluation der erfolgreichen Implementierung wird die Anwendung in einen Versuchsaufbau integriert, innerhalb dem das Gerät mit anderen Geräten über das Matter-Protokoll kommunizieren kann.

1.2 Gliederung der Arbeit

In 2 „Grundlagen“ werden zunächst verwandte Arbeiten beschrieben und die Grundlagen zu Matter anhand der Spezifikation und eines Versuchsaufbaus erklärt. Hier erfolgt auch eine Einführung in RIOT OS mit Beschreibung der für die Implementierung benötigten Module.

In 3 „Anforderungen & Analyse“ wird der Versuchsaufbau formal eingeführt, die konkreten Anforderungen erarbeitet, die sich aus der Matter-Spezifikation für die zu implementierenden Geräte ergeben und wie diese unter RIOT OS umgesetzt werden sollen.

Die Portierung von Matter nach RIOT OS und die Umsetzung des Versuchsaufbaus wird in 4 „Implementierung“ beschrieben und in 5 „Evaluation“, wie die implementierten Anforderungen getestet wurden. Zum Schluss werden die umgesetzten Ziele in 6 „Zusammenfassung und Ausblick“ reflektiert und weitere Schritte beschrieben, um die in dieser Arbeit umgesetzten Implementierungen weiterzuentwickeln.

2 Grundlagen

Vorab werden in 2.1 „Verwandte Arbeiten“ Beispiele beschrieben, in denen Matter bereits in andere Betriebssysteme oder SDK's integriert wurde. Für die umzusetzenden Schritte zur Portierung von Matter nach RIOT OS wird in 2.2 „Beschreibung der verwendeten Geräte“ ein Versuchsaufbau beschrieben, der im Rahmen dieser Arbeit implementiert wird. Die Grundlagen von Matter werden auf Basis des Versuchsaufbaus in Kapitel 2.3 beschrieben. In Kapitel 2.4 erfolgt eine Einführung in RIOT OS auf Basis der für diese Arbeit relevanten Module.

2.1 Verwandte Arbeiten

Die offizielle SDK von Matter [6] beinhaltet zum Zeitpunkt dieser Arbeit eine Implementierung der Version 1.2 der Matter-Spezifikation. Der Quellcode ist in C++ geschrieben und lässt sich mit dem Meta-Build-System GN [7] für die Ziel-Plattform konfigurieren und mit Ninja [8] zu statischen Bibliotheken kompilieren. Diese Implementierung ist bereits in die SDK's von Chip-Herstellern und Betriebssysteme integriert, die auf eingebetteten Systemen lauffähig sind. Eine Übersicht findet sich in [9, Guides/Platform Guides].

2.1.1 ESP-IDF von Espressif

Espressif unterstützt die Entwicklung von Matter-Endgeräten auf ESP32 SoC's mit Netzwerkanbindung über Wi-Fi und Thread sowie von Thread Border Routern und Matter Bridges (ZigBee und BLE Mesh). Die C++ Implementierung von Matter [6] ist als Submodul in *Espressif's SDK for Matter* [10] integriert und nutzt Softwarekomponenten des ESP-IDF (*Espressif IoT Development Framework*) [11] zur Integration der unterstützten Hardware (ESP32, ESP32-C, ESP32-S, ESP32-H).

Die erforderlichen Komponenten aus der ESP-IDF beinhalten unter anderem einen für Multi-Core Support modifizierten FreeRTOS Kernel, einen BLE Stack (Apache NimBLE) und einen Lightweight TCP/IP Stack (LwIP) (u. a. mit IPv4, IPv6, DHCP und mDNS Support) [11]. Für die Integration des LwIP Netzwerkstacks existieren *Networking API's* für Wi-Fi, Ethernet und Thread. Eine Anwendung kann unter Verwendung einer auf BSD-Sockets basierenden Application Programming Interface (API) Pakete über das Netzwerk senden und empfangen.

Um das Speichern und Auslesen von Daten aus internen oder angeschlossenen Speichermedien zu ermöglichen, existiert eine *Storage API*. Mit dieser lassen sich nicht-flüchtige Daten in einem Key-Value Store speichern (mit der NVS (*Non-volatile Storage Library*) Komponente). Das Modul VFS (Virtual Filesystem) bietet Schnittstellen, um Dateisysteme wie z. B. FAT oder SPIFFS einzubinden. Es existieren auch API's, um z. B. vom Hardware Random Number Generator (HWRNG) des ESP32 generierte Zufallszahlen oder die aktuelle Systemzeit unter Nutzung einer Real Time Clock (RTC) oder eines *High-Resolution Timers* abzufragen.

Ein Beispiel für die Entwicklung von Matter-Anwendungen in Rust auf Basis von ESP32 SoC's findet sich in [12], wo die Rust-Implementierung von Matter [13] mit den Komponenten aus ESP-IDF integriert werden.

2.1.2 Zephyr OS und nRF Connect SDK von Nordic Semiconductor

Nordic Semiconductor hat Matter mit einem Fork der Matter C++ SDK [6] in ihre *nRF Connect SDK* und in das Betriebssystem *Zephyr OS* integriert [14]. Dafür wurde ein "Integration Layer" entwickelt, der den BLE Stack aus Zephyr, den Wi-Fi Stack aus der nRF Connect SDK sowie den externen OpenThread Stack verwendet. Mit den Entwicklungsboards `nRF52840 DK`, `nRF5340 DK` und `nRF54L15 PDK` können "Matter over Thread" Geräte entwickelt werden.

2.1.3 Mbed-OS

Das Betriebssystem Mbed-OS [15] ist für den Einsatz auf eingebetteten Systemen mit Cortex-M Mikrocontrollern ausgelegt und hat Matter über einen "Special Intermediate Layer" mit dem eigenen BLE und Wi-Fi Stack integriert [9, Guides/Mbed-OS platform

overview]. Benötigte Komponenten des Matter C++ Quellcodes werden zu einer statischen Bibliothek kompiliert und mit CMake während des Mbed OS Build-Prozesses mit der Anwendung gelinkt.

2.2 Beschreibung der verwendeten Geräte

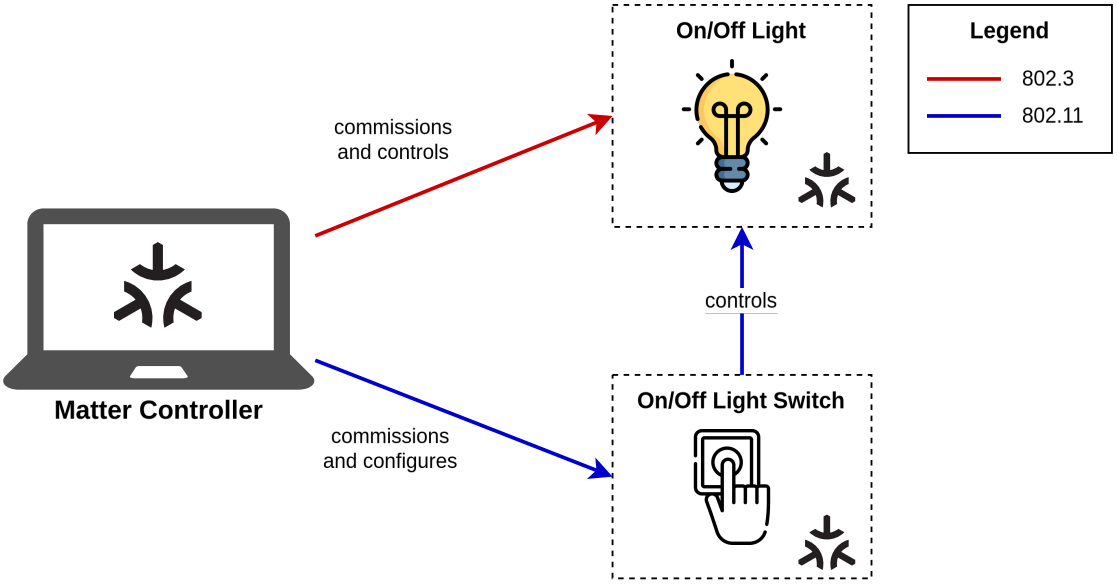

Abbildung 2.1: Versuchsaufbau bestehend aus einem Matter Controller, On/Off Light und On/Off Light Switch

Die Grundlagen zu Matter sowie die Anforderungen zur Portierung nach RIOT OS werden anhand des in Abbildung 2.1 dargestellten Versuchsaufbaus erarbeitet. Dieser besteht aus einem steuerbaren Licht mit Ein/Aus Funktion (**On/Off Light**) und einem Schalter (**On/Off Light Switch**), der das Licht über das Netzwerk ein- und ausschalten kann. Ein **Matter Controller** wird benötigt, um die Geräte in Betrieb zu nehmen (Commissioning) und zu konfigurieren. Zudem verfügt dieser auch über die Fähigkeit, mit den Geräten über Matter-Nachrichten zu interagieren und damit die Erfüllung der Anforderungen zu validieren.

2.2.1 On/Off Light

Das *On/Off Light* soll auf einem von RIOT OS unterstützten Entwicklungsboard implementiert werden. Daran ist eine LED angeschlossen, welche beim Empfang von Befehlen vom Matter Controller oder des On/Off Light Switch ein- bzw. ausgeschaltet werden soll. Zudem soll der aktuelle Zustand der LED ausgelesen werden können.

2.2.2 On/Off Light Switch

Der *On/Off Light Switch* soll auf einem von RIOT OS unterstützten Entwicklungsboard implementiert werden. Nach Kopplung mit dem On/Off Light können Kommandos über Matter gesendet werden, womit die Leuchte ein- bzw. ausgeschaltet und der aktuelle Zustand gelesen werden kann. Das Senden der Kommandos kann über ein eingebautes Terminal oder einen angeschlossenen Taster ausgelöst werden.

2.2.3 Matter Controller

Da die Geräte ohne Weiteres nicht in der Lage sind, über Matter miteinander zu kommunizieren, wird ein Controller benötigt zur Inbetriebnahme (Commissioning), Konfiguration und Steuerung der Geräte. Zudem sollen mit diesem die Anforderungen validiert werden.

2.3 Matter

Die Spezifikation des Matter-Protokolls ist aufgeteilt in die vier Dokumente **Core** [16], **Device Library** [17], **Application Cluster** [18] und **Standard Namespaces** [19] Specification. Die in dieser Arbeit beschriebenen Anforderungen beziehen sich auf die Version **1.2** der Matter-Spezifikation.

Core Specification Spezifiziert das Matter-Protokoll auf dem Application und Transport Layer und die grundlegenden Anforderungen, die alle Matter-zertifizierten Geräte erfüllen müssen. Dies beinhaltet unter anderem die unterstützten Technologien (Wi-Fi, Ethernet, Thread, BLE) und Transportprotokolle, das Datenmodell, Interaktionen, Adressierung der Geräte über IPv6, Commissioning-Verfahren, kryptografische Verfahren zur sicheren Authentifizierung etc.

Device Library Specification Hier sind die Anforderungen aller Gerätetypen spezifiziert, die zum Stand der Spezifikation von Matter unterstützt werden. Die gesamte Gerätefunktion ergibt sich aus der Zusammensetzung aus einem oder mehreren *Application Cluster*, die für den jeweiligen Gerätetyp unter den *Cluster Requirements* aufgelistet werden.

Application Cluster Specification Enthält die Spezifikation aller unterstützten *Application Cluster*, deren Attribute und unterstützen Interaktionen sowie ggf. weitere anwendungsspezifische Funktionen.

Standard Namespace Specification Spezifikation der Namespaces für das *Semantic Tag Feature*, das ermöglicht, Matter Cluster und Endpunkte zu beschreiben. Bspw. kann so der Standort (*Common Location Namespace*) oder die Position (siehe Beispiel in [19, S. 21]) beschrieben werden.

In den folgenden Kapiteln werden die Grundlagen des Matter-Protokolls anhand des in 2.2 „Beschreibung der verwendeten Geräte“ beschriebenen Versuchsaufbaus beschrieben. Dabei wird in Kapitel 2.3.3 zuerst das Datenmodell von Matter anhand des zu implementierenden Gerätetyps **On/Off Light** erklärt. Danach wird in 2.3.2 „Netzwerktopologie und Fabrics“ das Konzept von **Fabrics** beschrieben und wie **Nodes** miteinander kommunizieren können. In Kapitel 2.3.4 wird dann beschrieben, wie auf das Datenmodell von Nodes über die Ausführung von **Interaktionen** zugegriffen werden kann. Dies erfordert eine Kommunikation über das **Interaction Model Protocol**, das definiert wie Interaktionen mit dem in Kapitel 2.3.5 beschriebenen **Matter Message Format** kodiert werden. Um die Sicherheit bei der Übermittlung von Nachrichten zu gewährleisten, werden sichere (*Secure*) Sessions verwendet, welche in Kapitel 2.3.7 beschrieben werden.

2.3.1 Netzwerkarchitektur

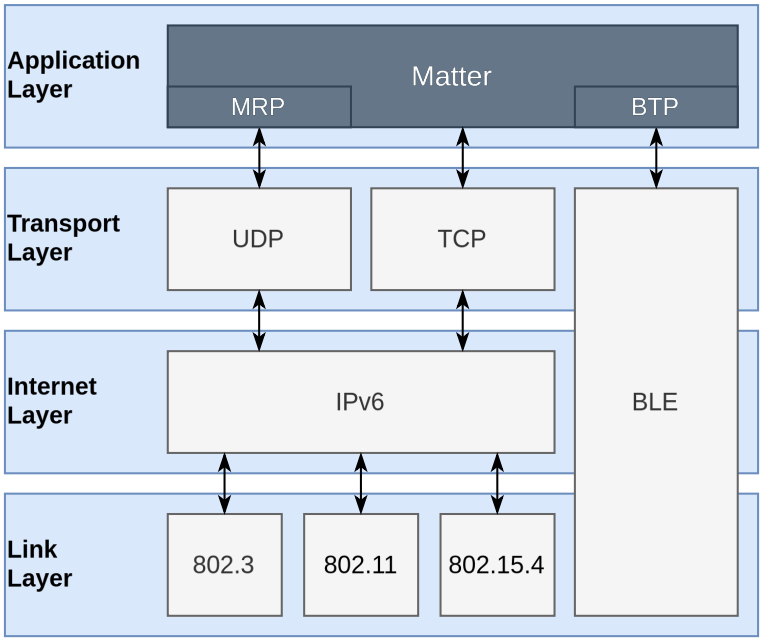

Abbildung 2.2: Einordnung von Matter in das TCP/IP Modell (Dunkel markiert: Teil des Matter-Stacks)

Wie in Abbildung 2.2 dargestellt, lässt sich das Matter-Protokoll im TCP/IP Modell auf dem *Application Layer* einordnen. Die Adressierung von Nodes zum Austausch von Nachrichten über die von Matter definierten Anwendungsprotokolle geschieht über IPv6. Als Transportprotokolle werden dabei UDP und TCP unterstützt. BLE wird nur für das Commissioning bei Geräten mit Wi-Fi oder Thread verwendet, um eine Konfiguration der Netzwerkschnittstelle durchzuführen (z. B. durch Senden der Zugangsdaten des Netzwerks). Um eine Verlässlichkeit bei der Datenübertragung zu gewährleisten, spezifiziert das Matter-Protokoll für UDP das Message Reliability Protocol (MRP) [16, S. 141] und bei Verwendung von BLE das Bluetooth Transport Protocol (BTP) [16, S. 203].

2.3.2 Netzwerktopologie und Fabrics

Eine Matter-Fabric beschreibt einen Verbund an Nodes, die einander vertrauen und miteinander über ihre Node-ID (Unicast) bzw. Group-ID (Multicast) kommunizieren können. Beim Hinzufügen eines neuen Gerätes (Commissioning) durch einen Commissioner tritt

das Gerät dessen Fabric bei und wird mit Node Operational Credentialss (NOCs) ausgestattet, welche die Basis des Vertrauens bilden. Matter Nodes können auch mehreren Fabrics angehören (*Multi-Admin Feature* [16, S. 869]), was den Vorteil hat, dass sich Geräte von mehreren Controllern (z. B. Smartphone Apps) steuern lassen. Durch die Verwendung von IPv6 können alle Nodes, die Wi-Fi, Ethernet oder Thread unterstützen, miteinander kommunizieren, solange sie Teil einer gemeinsamen Fabric sind. Auch Nicht-Matter-fähige Geräte, die z. B. nur ZigBee unterstützen, lassen sich über die Verwendung einer *Bridge* [16, S. 500] einbinden.

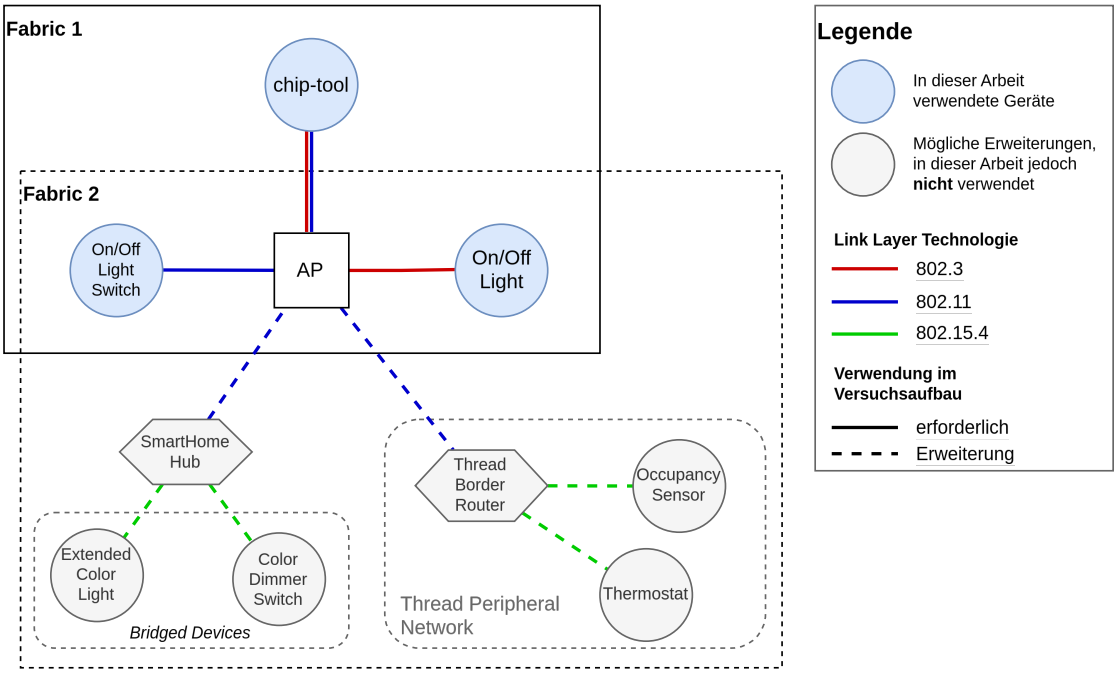

Abbildung 2.3: Netzwerktopologie des Versuchsaufbaus und möglichen Erweiterungen

Eine mögliche Netzwerktopologie in Matter wird in Abbildung 2.3 anhand des Versuchsaufbaus und möglichen Erweiterungen mit einer zweiten Fabric unter Beteiligung folgender Geräte dargestellt:

- **chip-tool**: Entwicklungs- und Testsoftware als CLI-Anwendung für Commissioning und Steuerung der Geräte in Fabric 1
- **On/Off Light**: Die für den Versuchsaufbau entwickelte Matter-Node
- **On/Off Light Switch**: Der für den Versuchsaufbau entwickelte Schalter, der mit dem On/Off Light gekoppelt wird

- **Access Point (AP)** mit Ethernet- und WLAN-Schnittstelle und Routingfähigkeiten
- **SmartHome Hub**: Fiktiver Matter-Controller, der die Einbindung von Nicht-Matter-fähigen ZigBee-Geräten sowie die Inbetriebnahme von Matter-Geräten über BLE unterstützt
- **Extended Color Light**: ZigBee Farblicht ohne Matter-Support
- **Color Dimmer Switch**: ZigBee Dimmschalter ohne Matter-Support zur Steuerung des Extended Color Light und des On/Off Light
- **Thread Border Router**: Routing zwischen Geräten des Thread Stub-Netzwerk und AP
- **Occupancy Sensor**: Bewegungssensor mit "Matter-over-Thread" Support
- **Thermostat**: Heizungstemperatur-Regler mit "Matter-over-Thread" Support

Im beschriebenen Beispiel befinden sich die in dieser Arbeit implementierten Geräte **chip-tool**, das **On/Off Light** und der **On/Off Light Switch** in der **Fabric 1**. Das On/Off Light lässt sich dabei über das chip-tool und den On/Off Light Switch steuern. Ein später hinzugefügter, fiktiver **SmartHome Hub**, der mit dem AP verbunden ist, bildet die Schaltzentrale der neuen **Fabric 2**. In diese wird das *On/Off Light* und der *On/Off Light Switch* durch ein erneutes Commissioning integriert. Die Kommunikation über *Matter Bridges* und Thread ist nicht Bestandteil des in dieser Arbeit umgesetzten Versuchsaufbaus, wird zur Vollständigkeit jedoch angedeutet. Dabei werden ein weiteres **Extended Color Light** und ein **Color Dimmer Switch** mit dem SmartHome Hub verbunden. Dieser emuliert die ZigBee-Geräte als Matter-Geräte, womit diese in die Fabric 2 eingebunden werden können. Die ebenfalls von Matter unterstützten Gerätetypen **Occupancy Sensor** und **Thermostat** werden über einen **Thread Border Router** nach dem Commissioning per BLE mit dem SmartHome Hub verbunden. Das Hinzufügen der neuen Fabric 2 hat keinen Einfluss auf die zuvor beschriebenen Funktionen innerhalb der Fabric 1, wurde jedoch so erweitert, dass das *On/Off Light* und der *On/Off Light Switch* nun auch innerhalb Fabric 2 bedient werden können.

Nodes, die Teil einer Fabric sind, können innerhalb dieser direkt über ihre link-lokalen IPv6-Adressen kommunizieren, die Anbindung an die öffentliche Infrastruktur ist also nicht notwendig. Die Kommunikation kann entweder direkt an eine einzelne Node gerichtet sein (Unicast) oder an eine Gruppe von Nodes über Multicast.

2.3.3 Datenmodell

Die grundlegenden Elemente des Datenmodells wie Endpunkte, Cluster, Attribute, Datentypen etc. sind im Detail in der *Matter Data Model Specification* [16, S. 355] beschrieben. Die Hierarchie von Elementen wird in 2.4 „Matter Datenmodell am Beispiel eines On/Off Light (gezeichnet anhand Vorlage aus [1])“ veranschaulicht. Die vollständigen Anforderungen an den Gerätetyp *On/Off Light* sind in [17, S. 31] beschrieben.

Node

Eine Node befindet sich in der Hierarchie des Matter-Datenmodells ganz oben und repräsentiert einen kompletten Gerätetyp und ist innerhalb einer Fabric über seine während des Commissioning zugewiesene Node ID eindeutig adressierbar. Eine Node besteht aus einem oder mehreren Endpunkten, die jeweils allgemeine oder anwendungsspezifische Funktionen erfüllen. Nodes können mehrere Rollen haben, von denen in dieser Arbeit folgende betrachtet werden:

- **Commissioner**: Eine Node, die ein Commissioning einer anderen Node durchführt (Beispiel: *SmartHome Hub* aus Abbildung 2.3)
- **Controller**: Eine Node, die dazu fähig ist, andere Nodes zu steuern (Beispiel: *On/Off Light Switch*)
- **Controlee**: Eine Node, die von anderen Nodes gesteuert werden kann (Beispiel: *On/Off Light*)

Eine Node mit der **Administrator**-Rolle ist zudem in der Lage, Aktionen auszuführen, die eine besondere Berechtigung benötigen. Hierzu zählen spezielle Cluster wie z. B. der **Access Control Cluster** [16, S. 486], in dem Zugriffsregeln erstellt werden können oder der **Binding Cluster** [16, S. 480], über den Endpunkte und Cluster von Nodes miteinander gekoppelt werden können.

Endpoints

Eine Matter Node besteht immer aus mindestens dem *Root Node* Endpunkt sowie weiteren Endpunkten, die anwendungsspezifische Funktionen implementieren. Jeder Endpunkt verfügt über einen Descriptor-Cluster [16, S. 477], der diesen Endpunkt beschreibt. Über

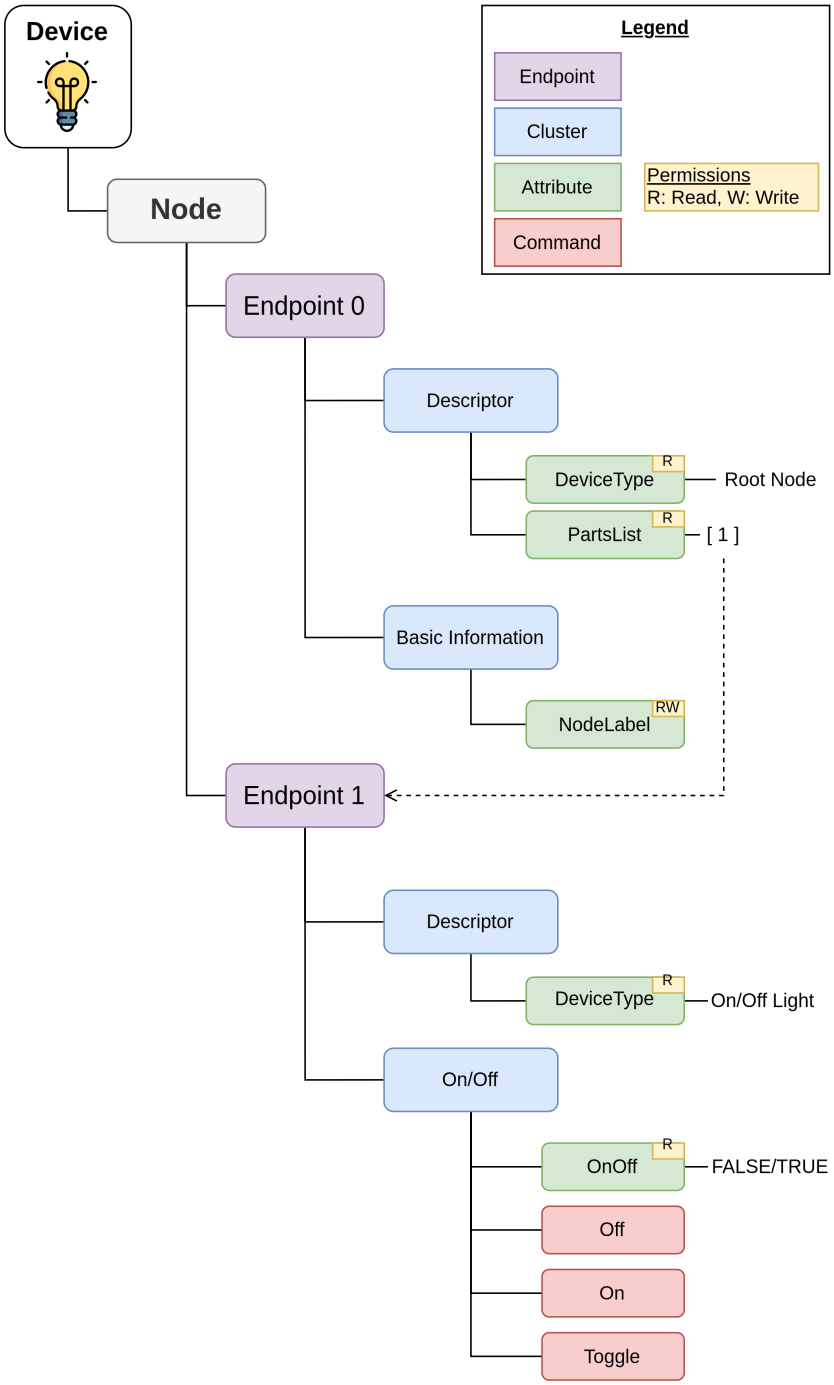

Abbildung 2.4: Matter Datenmodell am Beispiel eines On/Off Light (gezeichnet anhand Vorlage aus [1])

den Descriptor-Cluster des Root Node Endpunktes lässt sich das Gerät und seine Funktionen über das Auslesen von Attributen identifizieren. Das `DeviceTypeList` Attribut enthält eine Liste der unterstützten Gerätetypen und die `PartsList` eine Liste an weiteren Endpunkten, die auf der Node existieren. Beim Start am Root Node Endpunkt und Abfragen aller weiterer referenzierter Endpunkte und deren Cluster lässt sich somit die komplette Funktionalität bis ins Detail abfragen.

Cluster

Im Matter Datenmodell wird unterschieden zwischen Clustern für das *Service and Device Management* [16, ab S. 601], die im Root Node Endpunkt enthalten sind. Für gerätespezifische Cluster definiert die Application Cluster Specification [18] Attribute, Events, Kommandos und das Verhalten bei der Ausführung von Interaktionen. Bei Clustern gilt es noch zu unterscheiden zwischen *Server Cluster* und *Client Cluster*. Server Cluster sind *stateful* und repräsentieren ihren aktuellen Zustand über ihre Attribute. Client Cluster sind *stateless*, besitzen keine Attribute und können mit Server Cluster interagieren durch den Aufruf von Kommandos, dem Schreiben oder Lesen von Attributen.

2.3.4 Interaktionen

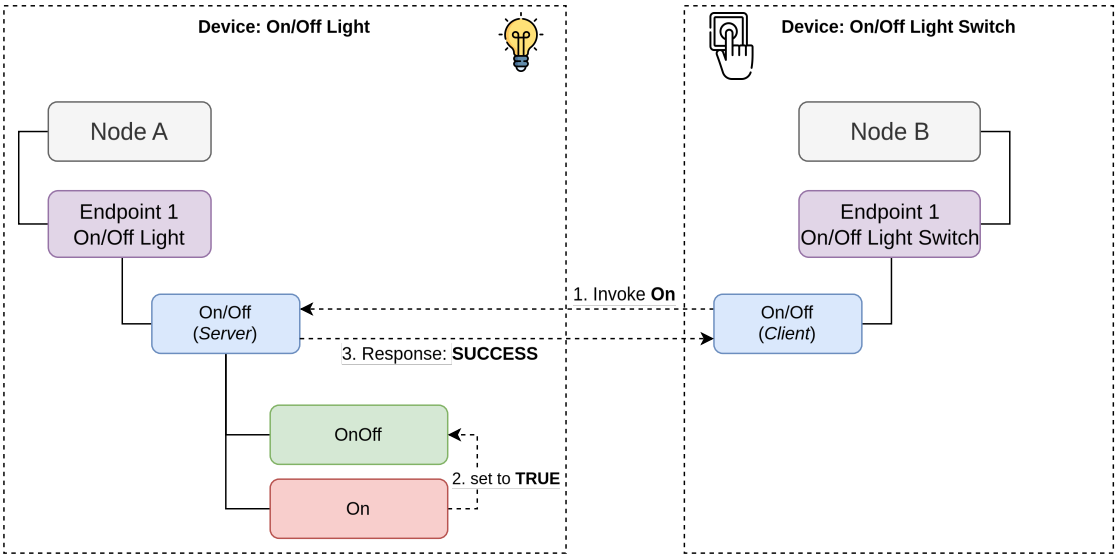

Abbildung 2.5: Matter Interaktionen zwischen On/Off Light Switch als Client und On/Off Light als Server (gezeichnet anhand Vorlage aus [1])

Um die Attribute des Datenmodells einer Node zu ändern, zu lesen oder Kommandos auszuführen, muss ein Client Cluster implementiert sein, welcher in der Lage ist, Interaktionen an einem Server Cluster auszuführen. Das grundlegende Prinzip von Interaktionen wird anhand des in Abbildung 2.5 gezeigten Beispiels erklärt. Dabei ist der On/Off Cluster als *Client* im Endpunkt 1 des On/Off Light Switch implementiert. Dieser führt am gekoppelten On/Off *Server* Cluster des On/Off Light die Invoke Interaction `On` aus, um das Licht einzuschalten. Nach Empfang des Kommandos ändert das On/Off Light das `OnOff` Attribut auf `TRUE` (Licht ist an) und sendet eine Antwort zurück zur Bestätigung der erfolgreichen Ausführung. Im Fehlerfall wird einer der Fehlercodes in [16, S. 464] zurückgesendet.

Alle unterstützten Interaktionstypen und der genaue Ablauf ist in der *Interaction Model Specification* [16, S. 425] spezifiziert. Die Interaktionstypen *Read* (Lesen von Attributen und Events), *Write* (Änderung von Attributen) sowie *Invoke* (Ausführen von Kommandos) sind für den Versuchsaufbau relevant.

Ablauf von Interaktionen

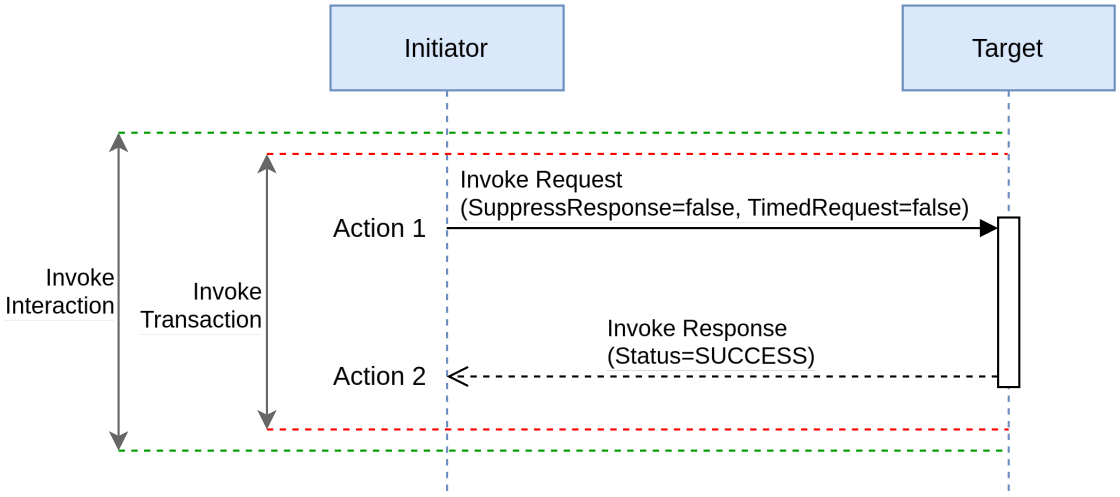

Abbildung 2.6: Sequenzdiagramm für das Funktionsprinzip einer Untimed Invoke Interaction

Eine Interaktion in Matter besteht immer aus einer oder mehreren Transaktionen, welche wiederum aus einer oder mehreren Aktionen besteht. Das Sequenzdiagramm in Abbil-

dung 2.6 zeigt beispielhaft den Ablauf einer *Untimed Invoke Interaction*, bestehend aus einer *Invoke Transaction* und der zwei Aktionen *Invoke Request* und *Invoke Response*.

Adressierung

In einer Interaktion muss immer der Pfad zum Cluster-Element einer Node oder Gruppe angegeben werden, welcher sich im Fall einer einzeln adressierten Node wie folgt zusammensetzt:

```
<node> <endpoint> <cluster> <attribute | command | event>
```

Diese Art der Adressierung erlaubt eine leichte Erweiterbarkeit bestehender und Unterstützung neuer Gerätetypen durch Spezifikation neuer Application Cluster.

Timed Interactions

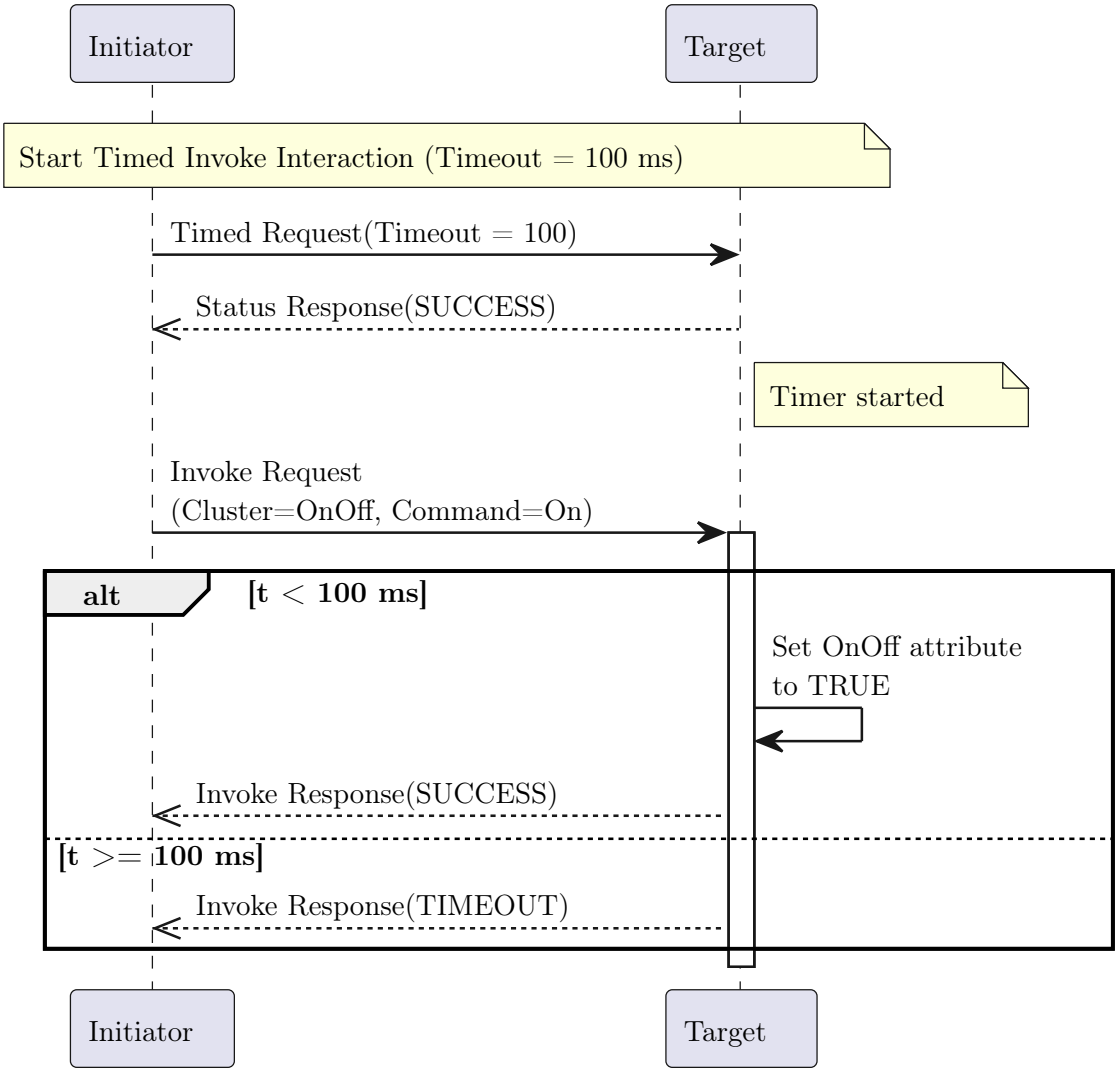

Abbildung 2.7: Sequenzdiagramm für das Funktionsprinzip einer Timed Invoke Interaction

Interaktionen vom Typ *Write* und *Invoke* können auch als *Timed Interactions* ausgeführt werden, womit Replay-Attacken verhindert werden sollen, bei denen ein Angreifer versucht, eine zuvor abgefangene Nachricht erneut zu senden. Bei *Timed Interactions* muss zuerst ein *Timed Request* gesendet werden, mit dem über ein Timeout angegeben wird, für wie lange die Interaktion gültig sein soll.

In dem in Abbildung 2.7 gezeigten Beispiel bestätigt die *Target*-Node den Timed Request mit einer Status Response und dem Status Code `SUCCESS` und startet einen Timer. Danach werden die Aktionen einer *Invoke Transaction* ausgeführt. Falls das Timeout noch nicht erreicht ist, wird der *Invoke Request* ausgeführt und bestätigt. Wenn der Invoke Request zu spät gesendet wurde, wird die gewünschte Interaktion nicht ausgeführt und der Fehlercode `TIMEOUT` zurückgesendet.

Interaction Model (IM) Protocol

Um die zuvor beschriebenen Aktionen einer Interaktion auszuführen, werden diese als *IM Protocol Messages* kodiert und zwischen beteiligten Nodes gesendet. In der *Interaction Model Encoding Specification* [16, S. 563] ist spezifiziert, wie die Payload dieser Nachrichten im *Matter TLV Format* kodiert wird. Bei jeder Interaktion wird im Paket-Header das Protokoll *Interaction Model* (`0x0001`) und der Aktionstyp als *Protocol Opcode* angegeben (z. B. ist der Opcode einer Invoke Request `0x08`).

2.3.5 Matter Message Format

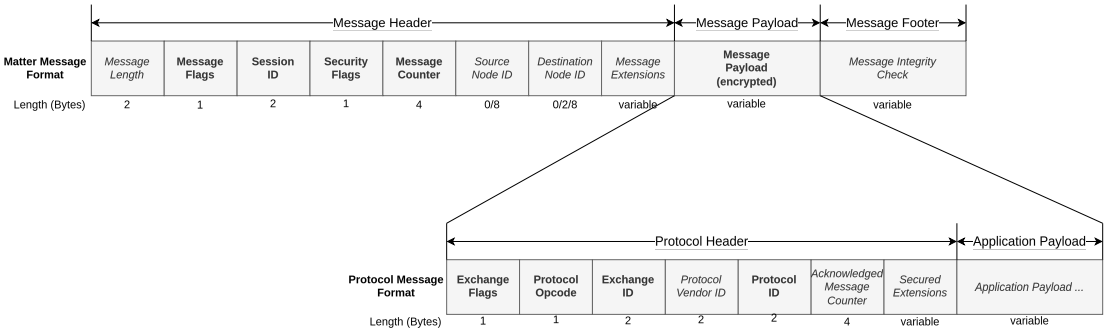

Abbildung 2.8: Matter Message Format (**Erforderliche** und *optionale* Felder)

Die Matter Core Spezifikation definiert in [16, S. 111] das *Matter Message Format*, das von Matter-Nodes unterstützt werden muss, um Nachrichten untereinander austauschen zu können. Eine Matter Message besteht aus einem **Message Header**, der **Message Payload** und einem **Message Footer**. Die Message Payload wiederum enthält Daten, die im **Protocol Message Format** kodiert sind.

Message Header

Der Message Header enthält u. a. Informationen über die Protokollversion, die im Paket enthaltenen Felder, den Message Counter (für die Erkennung von Duplikaten), den Typ und ID der assoziierten Session.

Message Payload

Eine Message Payload ist im *Protocol Message Format* kodiert. Der Inhalt und Aufbau ist abhängig vom verwendeten Protokoll und des Nachrichtentyps [16, S. 563] und wird wie folgt am Beispiel des `IM Protocol` anhand des in Abbildung 2.5 gezeigten Beispiels einer `InvokeRequestMessage` vom On/Off Light Switch an das On/Off Light beschrieben:

Durch `SuppressResponse = 0` wird angegeben, dass auf die Invoke Request eine Invoke Response zur Bestätigung erwartet wird. Mit `TimedRequest = 0` wird die Transaktion als *Untimed Transaction* ausgeführt. Die Liste an auszuführenden Kommandos wird im Feld `InvokeRequests` angegeben und enthält einen Eintrag, in dem der Pfad zum Cluster-Element `On` (`Command = 0x01`) des OnOff-Cluster (`Cluster = 0x0006`) an Endpunkt 1 (`Endpoint = 1`) angegeben wird. Für den Transport wird die Payload im Matter TLV (Type-Length-Value) Format kodiert, das im Detail in [16, S. 891] beschrieben ist.

Message Footer

Der Footer enthält nur ein Feld, den Message Integrity Check (MIC), welches bei sicheren Sessions einen Integritätscheck für die übermittelte Nachricht enthält. Damit kann der Empfänger feststellen, ob die Nachricht korrekt übertragen wurde.

2.3.6 Message Reliability Protocol

Da UDP als verbindungsloses Transportprotokoll keine zuverlässige Übertragung von Paketen garantiert, verwendet Matter das **Message Reliability Protocol (MRP)** [16, S. 141], das die korrekte Übertragung von Paketen wie folgt sicherstellt:

- Bei Timeout wird ein Paket bis zu 4 Mal [16, S. 151, `MRP_MAX_TRANSMISSIONS`] erneut versendet. Falls die Nachricht danach immer noch nicht bestätigt wurde, wird ein Fehler an die Applikation weitergegeben
- Identifikation eines Pakets und Duplikaterkennung anhand des *Message Counter* Feldes im Message Header
- Alle empfangenen Pakete müssen dem Sender bestätigt werden
- Es wird nur das zuerst empfangene Paket eines Message Counters an die Anwendung weitergegeben, Duplikate werden verworfen

2.3.7 Secure Sessions

Für den Austausch von Matter-Protokoll-Nachrichten zwischen Nodes muss zuerst ein sicherer Kanal (*Secure Session*) aufgebaut werden. Innerhalb einer Session können eine oder mehrere Interaktionen zwischen Nodes ausgeführt werden. An Sessions können auch mehrere Nodes beteiligt sein (Multicast), in dieser Arbeit wird jedoch nur die Unicast-Kommunikation zwischen zwei Nodes (Initiator und Target) betrachtet. Unicast Sessions beginnen immer mit einer ungesicherten *Session Establishment Phase*, für die zwei Varianten existieren. Für das Commissioning wird hier das Verfahren Passcode-Authenticated Session Establishment (PASE) verwendet. Für den Nachrichtenaustausch zwischen Nodes, die Teil einer Fabric sind, wird das Certificate Authenticated Session Establishment (CASE) verwendet, das die während des Commissioning erhaltenen NOC zur Authentifizierung verwendet. Am Ende des Session Establishment werden *Shared Keys* zwischen den Kommunikationsteilnehmern ausgetauscht, die während der darauf folgenden *Application Data Phase* zur Verschlüsselung und Entschlüsselung der Nachrichten dienen.

Passcode-Authenticated Session Establishment (PASE)

Zu Beginn des Commissioning wird eine sichere Session zwischen Commissioner und Commissionee mittels Passcode-Authenticated Session Establishment (PASE) [16, S. 156] aufgebaut. Dabei wird der in der Onboarding Payload kodierte Passcode als *Shared Secret* verwendet. Zur Initiierung der Session sendet der Commissioner den Passcode an den Commissionee. Wenn der Passcode korrekt ist, werden kryptografisch sichere Schlüssel ausgetauscht, die für die Kodierung und Dekodierung weiterer Nachrichten der Session verwendet werden.

Certificate-Authenticated Session Establishment (CASE)

Während dem Commissioning wurden vom Commissioner NOCs erstellt und an den Commissionee gesendet. Diese werden nach dem Commissioning von letzterem für die Authentifizierung innerhalb einer Fabric genutzt, um eine sichere Session zu anderen Nodes mit dem Certificate Authenticated Session Establishment (CASE) [16, S. 162] aufzubauen.

2.3.8 Discovery und Advertising

In Matter werden verschiedene Arten des Discovery definiert, welches das Ziel hat Commissionables, Commissioners oder Operational Nodes im Netzwerk aufzufinden. Nachfolgend sind die für diese Arbeit relevanten Verfahren beschrieben.

Commissionable Node Discovery [16, S. 87] Dient dem Auffinden von Nodes, bei denen ein Commissioning initiiert werden kann. Bei Commissionables, die bereits in das IP-Netzwerk eingebunden sind, erfolgt die Discovery nur über das Verfahren *IP Network Discovery* per DNS Based Service Discovery (DNS-SD), ansonsten über BLE oder Wi-Fi Soft AP. Der Status als Commissionable erfolgt durch das Advertising mit dem Servicetyp `_matterc._udp`. Zusätzlich müssen die für das Commissioning benötigten Daten [16, S. 90] als *Key-Value Pairs* gesendet werden.

Operational Node Discovery [16, S. 103] Beschreibt die Discovery von Nodes über DNS-SD, die bereits Teil einer Fabric sind. Für den Aufbau einer Verbindung wird die IPv6 Adresse oder der Hostname der adressierten Node benötigt, die über den aus der Fabric- und Node-ID konstruierten DNS-SD Instanznamen ermittelt werden kann. Eine Operational Node führt ein Advertising mit dem Servicetyp `_matter._tcp` durch. Optional können zusätzlich die in [16, S. 110] angegebenen Daten als *Key-Value Pairs* gesendet werden.

Für die Discovery werden die zuvor erwähnten Verfahren unterstützt:

IP Network Discovery Die Node ist bereits im IP-Netzwerk eingebunden und kann über das Senden von DNS-SD [20] Anfragen an die mDNS Multicast-Adresse `ff02::fb` und dem für mDNS spezifizierten Port `5353` [21] aufgefunden werden. Durch einen AAAA-Record wird die IPv6-Adresse und der Port mitgeteilt, an dem Matter-Protokoll-Nachrichten akzeptiert werden. Das Senden von A-Records (IPv4-Adresse)

ist optional [16, S. 86]. Zur eindeutigen Identifizierung eines gesuchten Gerätes im Netzwerk wird in einem TXT-Record ein *Discriminator* (optional auch die Vendor ID und Product ID) gesendet, welche auch in der Onboarding Payload kodiert sind.

Bluetooth Low Energy (BLE) Die Node führt ein Advertising über BLE durch und ist bereit für das Commissioning und die Netzwerkkonfiguration durch einen Commissioner

Wi-Fi Soft AP Die Node stellt einen *Wi-Fi* Access Point bereit für die Konfiguration der Wi-Fi Zugangsdaten und anschließendem Commissioning

2.3.9 Device Attestation

Durch den *Device Attestation* Prozess können Commissioner feststellen, ob es sich bei einem Gerät um ein Matter-zertifiziertes Produkt handelt, bevor ein erfolgreiches Commissioning durchgeführt werden kann. Commissionables müssen dafür das Device Attestation Certificate (DAC) und das zugehörige Private/Public Key Pair in der Firmware des Gerätes speichern [16, S. 282]. Das DAC muss die Vendor ID, Product ID und ein Gültigkeitsdatum enthalten und mit dem Format X.509v3 DER (siehe [22]) kompatibel sein. Die Matter-Spezifikation erlaubt auch das Commissioning nicht-zertifizierter Geräte in eine Fabric [16, S. 259], allerdings sollte der Nutzer über die damit einhergehenden Sicherheitsrisiken informiert werden. Damit ist das Verwenden von Testgeräten während der Entwicklungsphase möglich, wobei jedoch darauf geachtet werden muss, ob der verwendete Commissioner dies unterstützt.

2.3.10 Commissioning

Das Commissioning in Matter beschreibt den Prozess des Hinzufügens eines Commissionee in eine Fabric. Der Initiator dieses Prozesses ist der Commissioner, welcher bereits Mitglied und Administrator der Fabric sein muss. Bevor das Commissioning gestartet werden kann, werden Informationen vom Commissionee benötigt, die in einer Onboarding Payload kodiert sind. Danach wird der Commissionee mit einer der in 2.3.8 „Discovery und Advertising“ beschriebenen Verfahren zur *Commissionable Node Discovery* gesucht, das Commissioning wird gestartet und der Commissionee mit an die Fabric gebundenen NOCs ausgestattet, welche für die weitere sichere Authentifizierung der Node innerhalb der Fabric verwendet wird (siehe 2.3.7 „Secure Sessions“).

Onboarding Payload

In [16, S. 221] werden die Inhalte der Onboarding Payload spezifiziert. Diese kann entweder maschinell (z. B. in Form eines QR-Codes) oder textuell als *manuelle Onboarding Payload* kodiert und dem Commissioner bereitgestellt werden (z. B. durch Beilegen in der Verpackung des Gerätes oder Ausgabe in einem eingebauten Terminal oder Display).

Für die *Commissionable Node Discovery* wird in der Onboarding Payload u. a. die zu verwendende Technologie (BLE, Wi-Fi Soft AP oder On-Network sowie die Art des Commissioning spezifiziert. Der *Discriminator* (optional auch die *Vendor ID* und *Product ID*) dient dazu, den Commissionable über das Netzwerk eindeutig zu identifizieren (siehe auch 2.3.8). Der ebenfalls in der Onboarding Payload enthaltene *Passcode* dient als *Shared Secret* zum Aufbau einer sicheren Kommunikation mit PASE für das Commissioning (siehe 2.3.7).

2.4 RIOT OS

RIOT OS ist ein Open-Source Betriebssystem, das speziell für das IoT für den Einsatz auf kleinen System on a Chip (SoC) mit beschränkten Ressourcen entwickelt wurde. In [23] wurde auf die Anforderungen an ein Betriebssystem für IoT-Geräte bereits ausführlich eingegangen und RIOT OS als Lösung vorgestellt. Aufgrund der modularen Architektur ist RIOT OS auch darauf ausgelegt, externe Bibliotheken leicht zu integrieren. RIOT OS ist in C implementiert, unterstützt jedoch auch das Entwickeln von Anwendungen in C++ und Rust.

2.4.1 Systemarchitektur

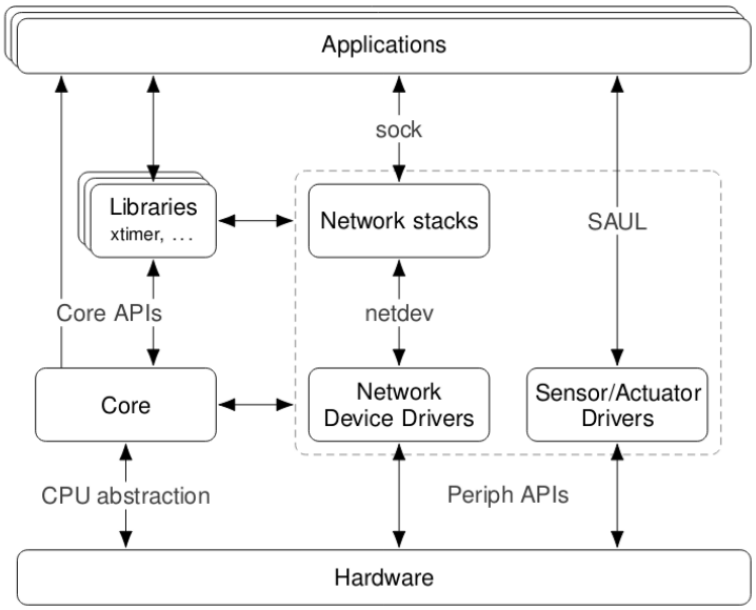

Abbildung 2.9: Systemarchitektur von RIOT OS (kopiert aus [2])

Zur Integration der Hardware bestehen Abstraktionen für verschiedene CPU's, Treiber für Netzwerkgeräte, Sensoren und Aktoren etc. Aufbauend auf den CPU-Abstraktionen bieten Libraries Funktionen wie z. B. Delays und das Ansteuern von GPIO's. Eine RIOT-Applikation kann mit dem Netzwerk-Stack über eine Implementierung der *RIOT Sock API* und mit Sensoren/Aktoren über den Sensor Actuator Uber Layer (SAUL) interagieren.

Das RIOT OS GitHub Repository [24] ist wie folgt aufgeteilt [25, Structure]:

- Kernel: `core`
- Platform-spezifischer Code: `cpu` und `boards`
- Gerätetreiber: `drivers`
- Bibliotheken und Netzwerk: `sys` (Interne System-Libraries) und `pkg` (Extern portierte Libraries)
- Anwendungsbeispiele und Tests: `examples` und `tests`

2.4.2 Build System

Anwendungen in RIOT müssen mindestens aus einem Makefile und Quellcode-Dateien mit einer `main`-Funktion bestehen. Im Makefile wird die Anwendung und ihre Abhängigkeiten wie folgt definiert:

- `APPLICATION`: Name der Anwendung
- `BOARD`: Standard-Board auf dem die Anwendung ausgeführt werden soll
- `RIOTBASE`: Absoluter Pfad zum RIOT OS Repository
- `USEMODULE`: Einbinden von RIOT Modulen
- `FEATURES_REQUIRED`: Definieren erforderlicher Features

Durch Aufruf des Befehls `make all` wird die Anwendung für das in der Variable `BOARD` angegebene Entwicklungsboard kompiliert.

2.4.3 Sock API und GNRC (Generic Network Stack)

Auf Basis von [26, 3.2 Overall Architecture] und der RIOT Dokumentation [25] wird nachfolgend die Netzwerkarchitektur von RIOT OS beschrieben, die darauf ausgelegt ist, verschiedene Link-Layer Technologien und Protokolle auf allen Ebenen des TCP/IP Modells zu unterstützen.

Intern erfolgt die Kommunikation zwischen den Schichten per (a)synchronem Message Passing über die `netapi`. Die Kommunikation mit einem auf der Zielhardware verfügbaren Netzwerkgerät erfolgt über die `netdev`-API, welche eine generische Schnittstelle bietet, um eine Integration verschiedener Link-Layer Technologien zu ermöglichen.

Für den *Application Layer* stellt die **Sock API** eine generische Schnittstelle zur Verfügung, mit der eine Anwendung oder Bibliothek Pakete unter Verwendung des Netzwerk-Stacks von RIOT OS senden und empfangen kann (vgl. POSIX Sockets). Zum Zeitpunkt dieser Arbeit werden UDP-, TCP-, DTLS- und Raw-IP-Sockets unterstützt. Die konkrete Implementierung der Sock API kann beim Kompilieren der Anwendung gewählt werden, indem das gewünschte Modul im `Makefile` eingebunden wird (z. B. über `USEMODULE += gnrc_sock_udp` für die Verwendung von UDP-Sockets.

2.4.4 SAUL Registry

Der Sensor Actuator Uber Layer (SAUL) bietet eine generische API, um Sensoren und Aktuatoren auslesen bzw. ansteuern zu können. Eine an einem Entwicklungsboard angeschlossene LED bspw. ist vom Aktuator-Typ *Switch* und wird bei Einbinden des `saul_default` Moduls automatisch in der SAUL Registry registriert. Aus der Applikation heraus lässt sich die LED dann in der Registry auffinden und durch Schreiben der Werte über die von SAUL genutzte, generische `Phydat`-Datenstruktur ansteuern.

2.4.5 shell

Das `shell`-Modul bietet die Möglichkeit, ein Command Line Interface (CLI) in die Anwendung zu integrieren. Je nach Einbinden weiterer Module werden automatisch Kommandos registriert (z. B. bei Einbinden des `saul_default` Moduls Befehle zum Auslesen von Sensoren und Ansteuern von Aktuatoren). Es lassen sich auch eigene Befehle in der Anwendung definieren, nach deren Eingabe selbst-definierte Funktionen aufgerufen werden.

2.4.6 ztimer

Das Modul `ztimer` stellt Timer-Funktionen zur Verfügung und schafft durch Abstraktion eine Unabhängigkeit zur CPU-spezifischen Initialisierung und API der verfügbaren Hardwaretimer. Damit lassen sich (a)synchrone Verzögerungen und Timeouts definieren, nach denen Funktionen aufgerufen werden (Callbacks).

2.4.7 VFS (Virtual File System) Layer

Mit dem `vfs`-Modul können Dateien und Verzeichnisse auf verschiedenen Speichergeräten und Dateisystemen gelesen und beschrieben werden (z. B. einer SD-Karte). Die API wurde nach Vorbild der POSIX API entworfen (open, close, read, write etc.). Durch Einbinden des `vfs_default` Moduls werden automatisch alle unterstützen und durch den Board-Treiber initialisierten Speichermedien eingebunden. Dynamische Speicherbereiche stehen unter dem Mountpoint `/nvm{n}` (wobei n = Anzahl an Mountpoints, beginnend bei 0) zur Verfügung. Durch Einbinden des `constfs`-Moduls wird ein unveränderlicher

interner Speicherbereich initialisiert, in dem zur Kompilierzeit bereitgestellte Daten gespeichert werden können.

2.4.8 Random Number Generator

Das Modul `random` bietet eine API für die Generierung von Zufallszahlen, wofür verschiedene Implementierungen existieren. Falls die Hardware dies unterstützt und ein RIOT-Treiber dafür implementiert ist, kann bspw. ein Hardware Random Number Generator (HWRNG) als Entropiequelle verwendet werden.

2.4.9 Rust in RIOT

Bei Entwicklung von Anwendungen in Rust unter RIOT OS muss im Makefile mit `FEATURES_REQUIRED += rust_target` angegeben werden, dass die CPU der Zielplatform die Kompilierung für Rust unterstützen muss. Im Build-System von RIOT wird Rust-Code als statische Bibliothek kompiliert und mit der RIOT-Anwendung gelinkt. Aus den C-Header-Dateien von allen eingebundenen Modulen werden dann für das `riot-sys` Crate Rust-Bindings generiert. Da die in `riot-sys` verfügbaren Funktionen für den Rust-Compiler "unsafe" sind und dadurch das Kompilieren nicht erlaubt, existiert das Crate `riot-wrappers`, das von der Anwendung eingebunden werden sollte, um RIOT-Module auf eine sichere Art zu verwenden. In `riot-wrappers` wird z. B. sichergestellt dass kein Zugriff auf nicht-initialisierte Variablen erlaubt ist und zugewiesener Speicher freigegeben wird, wenn Variablen ihren Gültigkeitsbereich verlassen.

Da Rust mit Cargo sein eigenes Paketmanagementsystem hat, müssen externe Libraries bei "Rust in RIOT"-Projekten nicht wie üblich in den `pkg`- oder `sys`-Ordner integriert werden, sondern können von *crates.io* (Rust Package Registry), von einem Git-Projekt oder aus einem lokalen Verzeichnis eingebunden werden.

3 Anforderungen & Analyse

Nachfolgend werden die Anforderungen analysiert, die sich anhand des in 2.2 „Beschreibung der verwendeten Geräte“ beschriebenen Versuchsaufbaus ergeben und wie dieser unter RIOT OS realisiert werden soll. Zunächst wird in Kapitel 3.1 eine externe Library ausgewählt, welche das Matter-Protokoll implementiert und für die Portierung nach RIOT OS verwendet werden soll. Danach wird die konkret verwendete Hardware und die unterstützten Funktionen der Geräte aus dem Versuchsaufbau in 3.2 „Realisierung des Versuchsaufbaus“ und 3.3 „Netzwerkanforderungen“ definiert.

In den darauf folgenden Kapiteln werden die Matter-spezifischen Anforderungen der zu implementierenden Geräte unter Berücksichtigung der von der Library und RIOT OS unterstützten Funktionalitäten beschrieben. Dabei wurden folgende Schritte ausgemacht:

- 3.4 „Datenmodell und Interaktionen“: Abbildung der physischen Geräte anhand der Matter-Spezifikation und Beschreibung der unterstützten Interaktionen
- 3.5 „Discovery und Advertising“: Verwendete Methode zum Auffinden als Matter-fähiges Gerät im Netzwerk
- 3.6 „Commissioning“: Aufbau der sicheren Kommunikation zum Gerät und Starten des Commissioning mit dem Ziel, das Gerät zur Fabric hinzuzufügen
- 3.7 „Device Attestation“: Methoden die Matter für die sichere Authentifizierung von Geräten nutzt, die das Matter-Protokoll unterstützen und welche Anforderungen sich daraus ergeben

Dabei wird auch betrachtet, welche der beschriebenen Anforderungen von der verwendeten Matter-Library unterstützt werden. In 3.8 „Integration von Matter in RIOT OS“ wird beschrieben, wie die Anforderungen unter RIOT OS umgesetzt werden sollen. Eine Zusammenfassung der Anforderungen findet sich in Kapitel 3.9.

3.1 Auswahl und Beschreibung der verwendeten Matter Library

Das erste Ziel bei der Verwendung einer externen Matter-Library ist das erfolgreiche Kompilieren und Linken nach Einbindung in eine RIOT OS Anwendung. Zu Beginn wurde versucht, die C++ Implementierung von Matter (**connectedhomeip** [6]) durch Erstellung eines neuen Package im RIOT OS Repository zu integrieren. Der benötigte Matter-spezifische Code muss dazu mit dem Meta-Build-System GN [7] konfiguriert, mit Ninja [8] zu einer statischen Bibliothek kompiliert und mit der Anwendung gelinkt werden. Aufgrund des komplexen Build-Systems und der vielen Konfigurationsmöglichkeiten konnte hier nach zwei Wochen noch kein Erfolg erzielt werden. Nach Anregung der Betreuer dieser Arbeit wurde versucht, die Rust-Implementierung von Matter zu verwenden, wobei innerhalb von drei Tagen schon das Ziel der erfolgreichen Kompilierung für eine minimale Demo-Anwendung erreicht werden konnte.

Für die Portierung von Matter nach RIOT OS im Rahmen dieser Arbeit wird deshalb die Rust-Library **rs-matter** gewählt, die auf GitHub unter der Apache 2.0 Lizenz veröffentlicht ist [13]. Zum Zeitpunkt dieser Arbeit befindet sich das Projekt noch im experimentellen Status und es werden folgende Funktionalitäten unterstützt:

- Commissioning über Ethernet
- Secure Session: PASE und CASE
- Interaktionen: `Invoke`, `Read` und `Write`
- Unterstützte Application Cluster: `On/Off`

3.2 Realisierung des Versuchsaufbaus

Nachfolgend wird die Hardware der im Versuchsaufbau verwendeten Geräte sowie deren unterstützten Funktionalitäten beschrieben.

3.2.1 On/Off Light

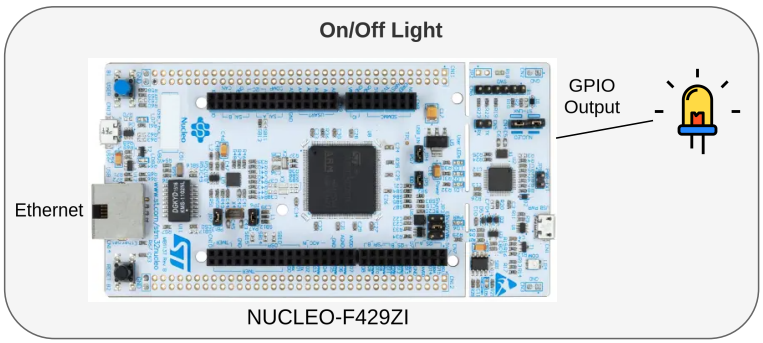

Abbildung 3.1: Verwendete Hardware des On/Off Light

Das *On/Off Light* wird als Beispiel-Anwendung in RIOT OS entwickelt und hardwareseitig über ein `NUCLEO-F429ZI` Entwicklungsboard (CPU: `STM32F4`) realisiert, welches über eine Ethernet-Schnittstelle in das Netzwerk eingebunden wird. Die auf dem Board verfügbaren LED's (rot, blau und grün) werden in der SAUL Registry (siehe Beschreibung in Kapitel 2.4.4) als Aktuator vom Typ `Switch` registriert. Der Zustand der blauen LED (ein/aus) soll nach Empfang der *Invoke Interactions* `On`, `Off` und `Toggle` von anderen Matter Nodes (im Versuchsaufbau durch den On/Off Light Switch und den Controller) über die SAUL Registry geändert werden können. Der aktuelle Zustand der LED wird im Attribut `OnOff` des `OnOff Cluster` gespeichert und kann mit *Read Interactions* ausgelesen werden.

3.2.2 On/Off Light Switch

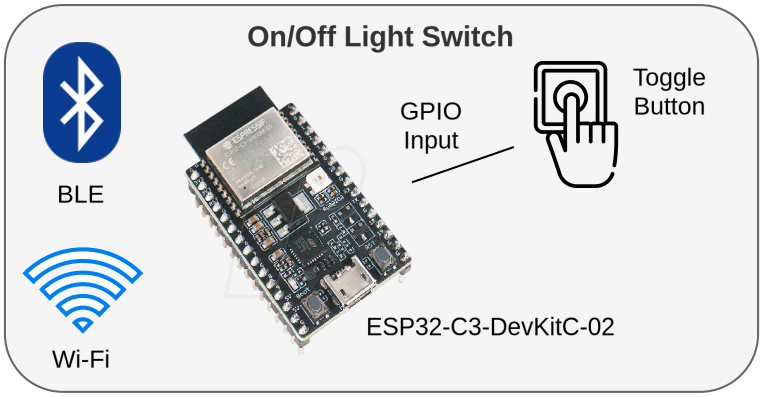

Abbildung 3.2: Verwendete Hardware des On/Off Light Switch

Der *On/Off Light Switch* soll auf dem Entwicklungsboard *ESP32-C3-DevKitC-02* (CPU: `ESP32-C3`) implementiert werden. Nach dem initialen Commissioning über BLE ist dieses per Wi-Fi in dieselbe Fabric wie das *On/Off Light* eingebunden. Mit dem *Matter Controller* wird der On/Off Light Switch an das On/Off Light gekoppelt, wodurch Interaktionen zwischen den Geräten ausgeführt werden können (siehe Beispiel in Abbildung 2.5). Das Senden der *Invoke Interactions* `Off`, `On` und `Toggle` an das On/Off Light soll durch Drücken eines angeschlossenen Tasters oder Ausführen von Befehlen in der eingebauten Shell ausgelöst werden können.

3.2.3 Matter Controller

Ein Controller wird benötigt, um das Commissioning der verwendeten Geräte, das Testen der entwickelten Anwendungen sowie die Konfiguration des On/Off Light Switch (Binding an das On/Off Light) zu unterstützen.

3.3 Netzwerkanforderungen

Nachfolgend werden die Anforderungen beschrieben, die die zu implementierenden Geräte als *IPv6 Node* erfüllen müssen.

3.3.1 IPv6 Adressierung

Da der GNRC Netzwerkstack von RIOT OS nur IPv6 unterstützt, wird sämtliche Kommunikation über IPv4-Adressen ignoriert, was laut Matter-Spezifikation auch erlaubt ist [16, S. 57]. Im Versuchsaufbau werden keine Gruppen an Nodes adressiert, weshalb lediglich eine Link-Local Adresse erforderlich ist, um die Kommunikation per Unicast zwischen den Nodes zu unterstützen.

3.3.2 Transportprotokoll

Als unterstütztes Transportprotokoll wird UDP verwendet. Matter schreibt für UDP als minimale Maximum Transmission Unit (MTU) 1280 Bytes vor (IPv6 Standard, siehe [27, S. 25]). Empfangene Pakete, die größer sind als 1280 Bytes, sollen von der Anwendung jedoch nicht verarbeitet werden [16, S. 119].

3.3.3 Multicast

Damit das On/Off Light die in Kapitel 2.3.8 beschriebenen Verfahren zur *IP Network Discovery* unterstützt, müssen von anderen Netzwerkteilnehmern gesendete DNS-SD Anfragen an die mDNS Multicast-Adresse (`ff02::fb`) empfangen und verarbeitet werden. Gruppen an Nodes werden in Matter über die aus der Fabric-ID und Gruppen-ID zusammengesetzten Multicast-Adresse adressiert. Für den Versuchsaufbau wird die Adressierung von Gruppen jedoch nicht benötigt und deshalb auch nicht näher betrachtet.

3.4 Datenmodell und Interaktionen

Nachfolgend werden die zu unterstützenden Elemente des Datenmodells (Endpunkte, Cluster, Features, Attribute und Kommandos) der verwendeten Geräte gemäß Matter-Spezifikation und der in [16, S. 358] beschriebenen *Conformance Level* ermittelt, abgesehen vom Descriptor-Cluster [16, S. 477], der für alle Endpunkte implementiert sein muss. Wie bereits in 2.3.3 „Endpoints“ beschrieben, müssen alle Matter-Nodes den *Root Node Endpunkt* unterstützen. Zusätzlich muss jeweils ein weiterer Endpunkt existieren, der die gerätespezifischen Funktionalitäten über den *On/Off Light* bzw. den *On/Off Light Switch* Endpunkt implementiert. Von den aufgeführten Clustern wird im Detail nur auf

den *On/Off Cluster* eingegangen, der dafür notwendig ist, die im Versuchsaufbau beschriebene Funktion zum ein- und ausschalten der LED am On/Off Light zu realisieren. Bei den weiteren Clustern wird auf die entsprechenden Stellen in der Spezifikation verwiesen.

3.4.1 Root Node Endpunkt

In der nachfolgenden Tabelle 3.1 sind die Anforderungen für den Root Node Endpunkt [17, S. 22] für die beiden Geräte zusammengefasst.

Cluster	**On/Off Light**	**On/Off Light Switch**	**rs-matter Support**
Basic Information [16, S. 601]	X	X	ja
Access Control [16, S. 486]	X	X	ja
Group Key Management [16, S. 610]	X	X	ja
General Commissioning [16, S. 670]	X	X	ja
Administrator Commissioning [16, S. 773]	X	X	ja
Network Commissioning [16, S. 648]	X	X	ja
Node Operational Credentials [16, S. 751]	X	X	ja
General Diagnostics [16, S. 685]	X	X	ja
Ethernet Network Diagnostics [16, S. 728]	X	-	ja
Wi-Fi Network Diagnostics [16, S. 721]	-	X	nein

Tabelle 3.1: Anforderungen an Server-Cluster des Root Node Endpunktes (X: erforderlich, –: nicht erforderlich)

3.4.2 On/Off Light Endpunkt

Der Endpunkt 1 des On/Off Light muss die gerätespezifischen Funktionen über folgende Cluster implementieren [17, S. 31]:

Cluster	Typ	rs-matter Support
Identify [18, S. 20]	Server	nein
Groups [18, S. 26]	Server	nein
Scenes [18, S. 34]	Server	nein
On/Off [18, S. 58]	Server	ja

Tabelle 3.2: Erforderliche Application Cluster des *On/Off Light* Endpunktes

Attribute

Für den On/Off Cluster muss mindestens das Attribut `OnOff` unterstützt werden, damit die am On/Off Light angeschlossene LED ein- und ausgeschaltet sowie der aktuelle Zustand gelesen werden kann. Aufgrund der Geräteart wird für das On/Off Light zudem das Feature `LT` (*Lighting Application*) spezifiziert [17, S. 32], was die Unterstützung weiterer nachfolgender Attribute erfordert, von denen rs-matter jedoch nur `OnOff` unterstützt.

OnOff Read-Only, Typ bool
: Aktueller Zustand (`TRUE`: an, `FALSE`: aus)

GlobalSceneControl Read-Only, Typ bool
: Muss auf `TRUE` gesetzt werden, wenn sich `OnOff` Attribut auf `TRUE` ändert oder bei Empfang des Kommandos `OnWithRecallGlobalScene`. Muss auf `FALSE` gesetzt werden nach Empfang des Kommandos `OffWithEffect`.

OnTime Read-Write, Typ uint16
: Dauer in 1/10 Sekunden der Einschaltzeit, wenn das Kommando `OnWithTimedOff` ausgeführt wird

OffWaitTime Read-Write, Typ uint16
: Dauer in 1/10 Sekunden, in der das Licht mindestens aus sein muss, bevor ein weiteres `OnWithTimedOff` Kommando ausgeführt werden kann

StartUpOnOff Read-Write, Typ Enum
: Zustand des Gerätes nach Einschalten der Stromversorgung (`0`=On, `1`=Off, `2`=Toggle)

3.4.3 On/Off Light Switch Endpunkt

Der Endpunkt 1 des On/Off Light Switch muss die gerätespezifischen Funktionen über folgende Cluster implementieren [17, S. 45]:

Cluster	Typ	rs-matter Support
Identify [18, S. 20]	Server	nein
Identify [18, S. 20]	Client	nein
On/Off [18, S. 58]	Client	nein

Tabelle 3.3: Erforderliche Application Cluster des *On/Off Light Switch* Endpunktes

3.4.4 Unterstützte Interaktionen

Die Interaktion mit den *Cluster Server* des On/Off Light geschieht durch *Client Cluster*, die auf anderen Nodes der Fabric implementiert sind (im Versuchsaufbau der On/Off Light Switch und der Matter Controller) und dadurch fähig sind Kommandos auszuführen, Attribute zu lesen und zu schreiben. Für das On/Off Light wurden folgende *Invoke Interactions* ausgemacht, die laut Matter Spezifikation [18, S. 64] implementiert sein müssen:

Name	Beschreibung	rs-matter Support
Off	Attribut `OnOff` auf `FALSE` setzen	ja
On	Attribut `OnOff` auf `TRUE` setzen	ja
Toggle	Attribut `OnOff` invertieren	ja
OffWithEffect	Licht ausschalten mit Effekt (`DelayedAllOff` oder `DyingLight`)	nein
OnWithRecallGlobalScene	Licht einschalten und globale Szene aufrufen	nein
OnWithTimedOff	Licht für in `OnTime`-Attribut angegebene Zeit ein- und danach wieder ausschalten	nein

Tabelle 3.4: Unterstützte *Invoke* Interaktionen des On/Off Cluster des On/Off Light Endpunktes

Zusätzlich müssen die in Kapitel 3.4.2 aufgelisteten Attribute je nach Berechtigung mit *Read Interactions* gelesen bzw. mit *Write Interactions* geändert werden können.

3.5 Discovery und Advertising

Da das im Versuchsaufbau verwendete On/Off Light über die Ethernet-Schnittstelle bereits in das Netzwerk eingebunden ist, wird das in 2.3.8 „Discovery und Advertising“ beschriebene Verfahren *IP Network Discovery* verwendet. Dabei muss das On/Off Light DNS-SD Antworten an die mDNS Multicast-Adresse `ff02::fb` senden und auf eingehende DNS-SD Anfragen per Unicast antworten.

Der On/Off Light Switch muss nach dem ersten Einschalten das Advertising über BLE und nach dem Commissioning genauso wie das On/Off Light das Verfahren *IP Network Discovery* unterstützen.

3.6 Commissioning

Um ein Commissioning durchführen zu können, müssen folgende Daten in der Firmware des jeweiligen Gerätes programmiert sein:

- **Discriminator** (12 Bit unsigned integer, zwischen `0` und `4095`) [16, S. 222]: Zur Identifikation des Gerätes über die *Commissionable Node Discovery*
- **Passcode** (27 Bit unsigned integer, zwischen `0x0000001` to `0x5F5E0FE`) [16, S. 222]: *Shared Secret* für den Aufbau einer sicheren Session mit PASE

Diese Daten müssen in einer **Onboarding Payload** kodiert sein, die für den Commissioner durch Ausgabe als Text-String in der Konsole nach Einschalten des Gerätes zur Verfügung gestellt werden muss. Anhand des **Discriminators** müssen die verwendeten Geräte über die *Commissionable Node Discovery* eindeutig identifiziert werden können. Es wird erwartet, dass der **Matter Controller** das Commissioning unter Angabe der **Onboarding Payload** erfolgreich durchführen kann.

3.7 Device Attestation

Während des Commissioning verifiziert der Commissioner, ob es sich beim On/Off Light um ein Matter-zertifiziertes Gerät handelt. Für ein erfolgreiches Commissioning müssen deshalb folgende Daten in der Firmware programmiert und während dem Betrieb der entwickelten Geräte ausgelesen werden können:

- Vendor ID: Eine der *Test Vendors #1-4* [16, S. 53]
- Product ID: Frei wählbar (außer `0x0000`, welche für die in [16, S. 53] beschriebenen, speziellen Anwendungsfälle reserviert ist)
- Certification Declaration (CD)
- Product Attestation Intermediate (PAI)
- DAC mit Private und Public Key Pair

3.8 Integration von Matter in RIOT OS

In diesem Kapitel wird beschrieben, wie Matter in RIOT OS durch die Verwendung der rs-matter Library integriert werden kann und wie diese die zuvor beschriebenen Anforderungen unterstützt bzw. einschränkt.

3.8.1 Systemarchitektur

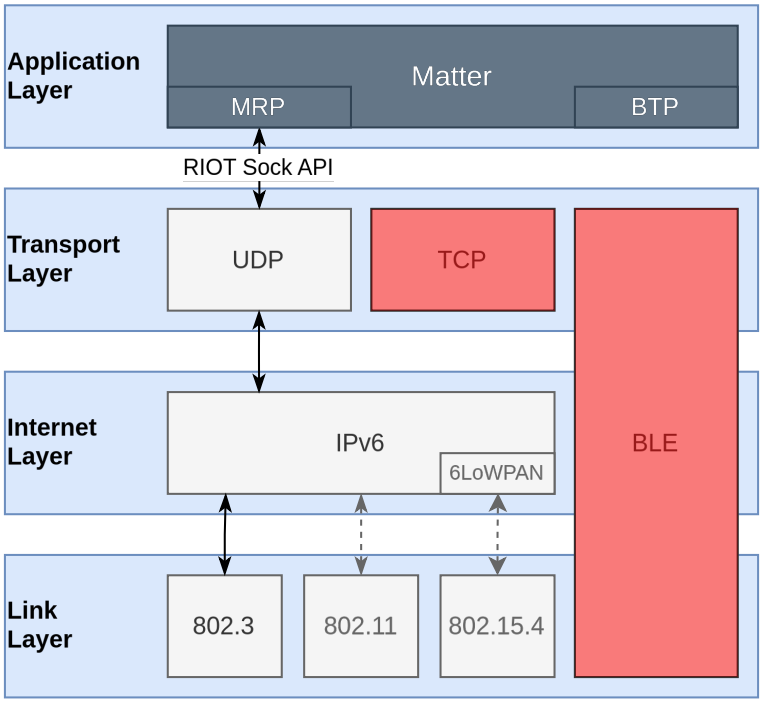

Abbildung 3.3: Integration von rs-matter in RIOT OS anhand TCP/IP Modell (Rot markiert: Aktuell noch nicht unterstützt von rs-matter)

Die zuvor beschriebene Library `rs-matter` muss im Rahmen der Implementierung in die Systemarchitektur von RIOT OS integriert werden, wie in Abbildung 3.3 dargestellt. Für die Kommunikation zwischen der Anwendung auf dem Application Layer und dem Netzwerkstack werden UDP-Sockets der in Kapitel 2.4.3 beschriebenen RIOT Sock API verwendet. Da mit der rs-matter Library wie im Kapitel 3.1 beschrieben nur das Commissioning über Ethernet (802.3) möglich ist, werden die weiteren für Matter zu unterstützenden Link-Layer Technologien 802.11 (Wi-Fi) und 802.15.4 über 6LoWPAN lediglich angedeutet.

3.8.2 Handling der vom On/Off Light empfangenen Interaktionen

Beim Empfang von Kommandos am `OnOff` Cluster Server des On/Off Light sollen diese von der Anwendung verarbeitet werden, um die angeschlossene LED anzusteuern. Dafür wird das in 2.4.4 „SAUL Registry“ beschriebene RIOT-Modul `saul` verwendet. Bei

Empfang einer *Invoke Interaction* zur Änderung des `OnOff`-Attributs wird die LED in der Registry gesucht und der neue Zustand beschrieben (`FALSE` → aus, `TRUE` → ein). Zusätzlich wird der aktuelle Zustand gespeichert, welcher dann bei *Read Interactions* des `OnOff`-Attributs des On/Off Cluster zurückgegeben wird.

3.8.3 RIOT Shell

Das RIOT-Modul `shell` soll für Diagnosezwecke und die Interaktion mit der Anwendung eingebunden werden. In der Anwendung sollen folgende Befehle unterstützt werden:

- `ifconfig`: Informationen über die Netzwerkschnittstelle abrufen, wie z. B. IPv6 Adressen, MTU
- `ps`: Informationen über aktuell laufende Threads mit Stack-Belegung
- `heap`: Anzeige des aktuell belegten Speichers auf dem Heap
- `saul`: Lesen und Schreiben des Zustands der angeschlossenen LED
- `vfs`: Lesen und Schreiben gespeicherter Dateien

3.9 Zusammenfassung der Anforderungen

Durch die Auswahl der rs-matter Library ergibt sich die wichtige Einschränkung für den Versuchsaufbau, dass zum Zeitpunkt dieser Arbeit unter RIOT OS nur die Implementierung des On/Off Light möglich ist. Der On/Off Light Switch und der Matter Controller sollen deshalb außerhalb RIOT OS und unter Verwendung der Matter C++ SDK [6] implementiert werden.

In Tabelle 3.5 „Funktionale Anforderungen zur Implementierung des On/Off Light unter RIOT OS“ sind die in den vorigen Kapiteln erarbeiteten Anforderungen an die durchzuführenden Arbeiten zur Portierung von Matter nach RIOT OS und zur Entwicklung der *On/Off Light* Anwendung zusammengefasst.

ID	Beschreibung
1	Das Gerät sendet in regelmäßigen Zeitabständen einen DNS-SD Response an die mDNS Multicast-Adresse `ff02::fb`, mindestens mit einem AAAA-Record (IPv6 Adresse) und einem TXT-Record mit Matter-spezifischen Informationen als *Key-Value Pairs*
2	Nach dem Einschalten ist das Gerät über die *Commissionable Node Discovery* per DNS-SD auffindbar
3	Der Passcode für das Commissioning wird nach dem Einschalten in der Konsole ausgegeben
4	Mit Eingabe des ausgegebenen Passcodes lässt sich das Commissioning erfolgreich durchführen
5	An Endpunkt 0 wird der Gerätetyp `Root Node` unterstützt
6	An Endpunkt 1 wird der Gerätetyp `On/Off Light` unterstützt
7	An Endpunkt 1 existiert der Cluster Server `On/Off` mit dem Attribut `OnOff` und den unterstützten Interaktionen `On`, `Off` und `Toggle`
8	Nach Empfang des Kommandos `On` am `On/Off Cluster` des Endpunkt 1 wird das `OnOff` Attribut auf `TRUE` gesetzt und die angeschlossene LED wird eingeschaltet
9	Nach Empfang des Kommandos `Off` am `On/Off Cluster` des Endpunkt 1 wird das `OnOff` Attribut auf `FALSE` gesetzt und die angeschlossene LED wird ausgeschaltet
10	Nach Empfang des Kommandos `Toggle` am `On/Off Cluster` des Endpunkt 1 wird das `OnOff` Attribut invertiert und der Zustand der angeschlossenen LED ändert sich
11	Der On/Off Light Switch ist in der Lage, die Interaktionen `On`, `Off` und `Toggle` am On/Off Light auszuführen

Tabelle 3.5: Funktionale Anforderungen zur Implementierung des On/Off Light unter RIOT OS

4 Implementierung

In der Implementationsphase wird der Versuchsaufbau in folgenden Schritten umgesetzt:

1. Modifikation der externen Matter-Library in einem Fork von **rs-matter** [28] mit den für RIOT OS benötigten Anpassungen (siehe Kapitel 4.2)
2. Integration von **rs-matter** in **riot-wrappers** [29] unter Verwendung des Standard-Netzwerkstacks von RIOT OS (GNRC) und der Sock API (siehe Kapitel 4.3 und Pull Request [30])
3. Implementierung des **On/Off Light** als Beispiel-Anwendung mit Rust-Support im **RIOT OS Repository** [24] (siehe Kapitel 4.4 und Pull Request [31])
4. Implementierung des **On/Off Light Switch** unter Verwendung der *Espressif's Matter SDK* [10] (siehe Kapitel 4.5)
5. Implementierung des **Matter Controller** unter Verwendung der *Matter C++ SDK* [6] (siehe Kapitel 4.6)

Die für diese Arbeit notwendigen Implementierungen sind auf dem GitHub Account (*maikerlab*) vom Autor dieser Arbeit veröffentlicht (vgl. Abbildung 4.1) und sollen durch Pull Requests in die originalen Repositories überführt werden.

4.1 Verwendete Software und Dependencies

Für die Implementierung wurde zusätzlich folgende Software verwendet:

- Rust Toolchain: `1.78.0-nightly (d18480b84 2024-03-04)`
- C2Rust [32]: `0.18.0 (2024-01-30)`

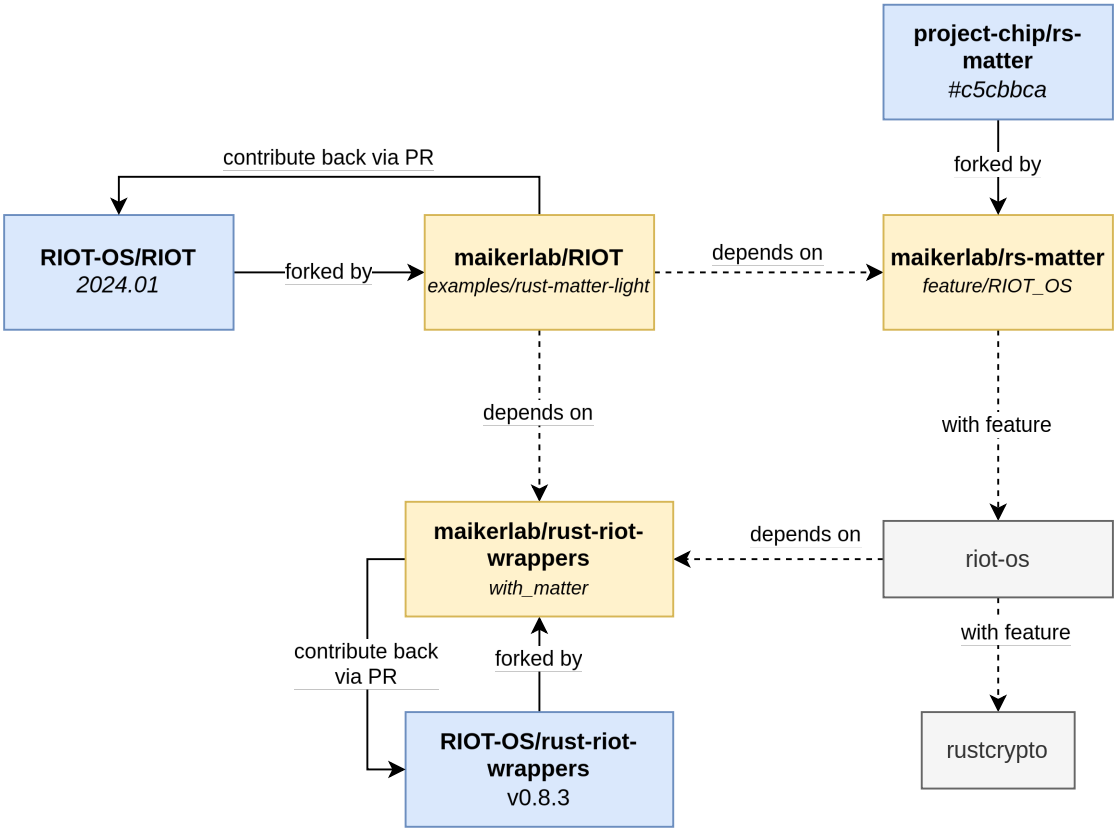

Abbildung 4.1: Projektstruktur (maikerlab: GitHub Account des Autors dieser Arbeit)

Alle verwendeten Rust-Libraries sind aus der `Cargo.toml` im Verzeichnis der entwickelten Demo-Anwendung ersichtlich oder durch Ausführung des Befehls `cargo tree`.

4.2 Anpassung der Matter Library für RIOT OS

Im Fork der *rs-matter* Library [28] werden die für RIOT OS benötigten Änderungen implementiert, welche beim Einbinden durch eine RIOT-Anwendung kompiliert werden müssen. Dafür wurde das Feature `riot-os` eingeführt, welches bei Aktivierung die benötigten Dependencies `riot-wrappers`, `embedded-hal-async`, `embedded-nal-async` sowie das bereits vorhandene Feature `rustcrypto` inkl. der Abhängigkeiten einbindet.

4.2.1 Deaktivieren des Senden von DNS-SD A-Records

Da IPv4 wie in 3.3.1 „IPv6 Adressierung“ beschrieben nicht unterstützt wird, wurde das Senden von DNS-SD A-Records deaktiviert.

4.2.2 Einbindung des ztimer Moduls

rs-matter verwendet für asynchrone Delays den `Timer` aus `embassy_time::timer`. Da eine Timer-Abstraktion in RIOT OS bereits existiert, wird für die Implementierung das Modul `ztimer` verwendet.

4.3 Integration von Matter in RIOT OS

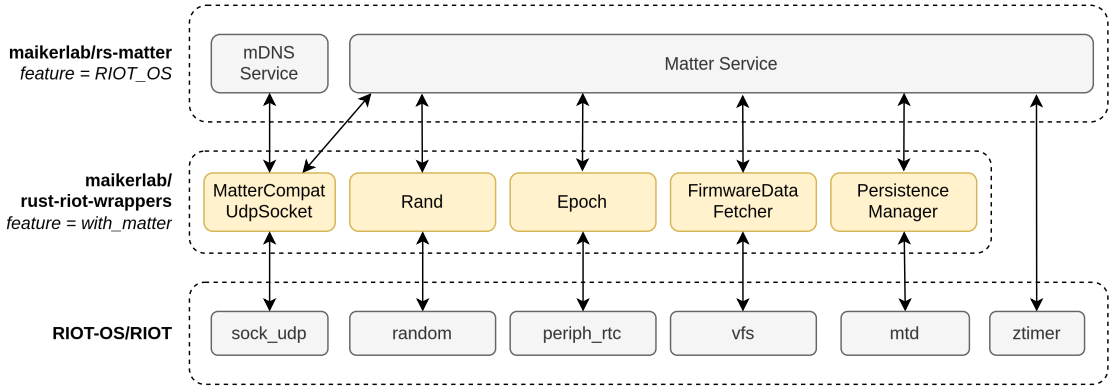

Abbildung 4.2: Kompatibilitätslayer für die Integration von Matter in RIOT OS (Gelb: Eigene Implementierung)

In Abbildung 4.2 sind die in rs-matter implementierten Matter- und mDNS-Service, die von RIOT OS genutzten Module sowie der zu entwickelnde *Kompatibilitätslayer* dargestellt.

MatterCompatUdpSocket Für die Nutzung der (UDP) Sock API von RIOT OS muss ein Wrapper entwickelt werden, welcher die Kompatibilität zu rs-matter herstellt (siehe 4.3.1 „UDP Sock API“)

Rand Nutzt das `random`-Modul als Entropie-Quelle für den in Matter spezifizierten Deterministic Random Bit Generator (DRBG) [16, S. 63] (siehe 4.3.3 „Random Number Generator“)

Epoch Stellt die aktuelle Systemzeit unter Nutzung einer RTC (Modul `periph_rtc`) zur Verfügung (siehe 4.3.4 „Ermittlung der aktuellen Systemzeit“)

FirmwareDataFetcher Stellt die für das Commissioning und Device Attestation benötigten Daten zur Verfügung (siehe 4.3.5 „Auslesen von Daten aus der Firmware“)

PersistenceManager Zuständig für das Speichern und Laden nicht-flüchtiger Daten in einem *Key-Value Store* (siehe 4.3.6 „Datenpersistierung“)

Der Kompatibilitätslayer wird in einem Fork von **riot-wrappers**, der Rust-Library für RIOT OS, in einem neuen Modul `matter` entwickelt, welches durch Aktivierung des Cargo-Features `with_matter` genutzt werden kann.

4.3.1 UDP Sock API

Für den Matter- und mDNS Service wird jeweils ein UDP-Socket an den Ports `5540` bzw. `5353` benötigt. Für die Implementierung wird der in riot-wrappers [29] im Modul `socket_embedded_nal_async_udp` verfügbare `UnconnectedUdpSocket` verwendet. In der Anwendung müssen dafür die Module `sock_udp`, `sock_aux_local` im Makefile eingebunden und in riot-wrappers das Feature `with_embedded_nal_async` aktiviert werden.

Problembeschreibung

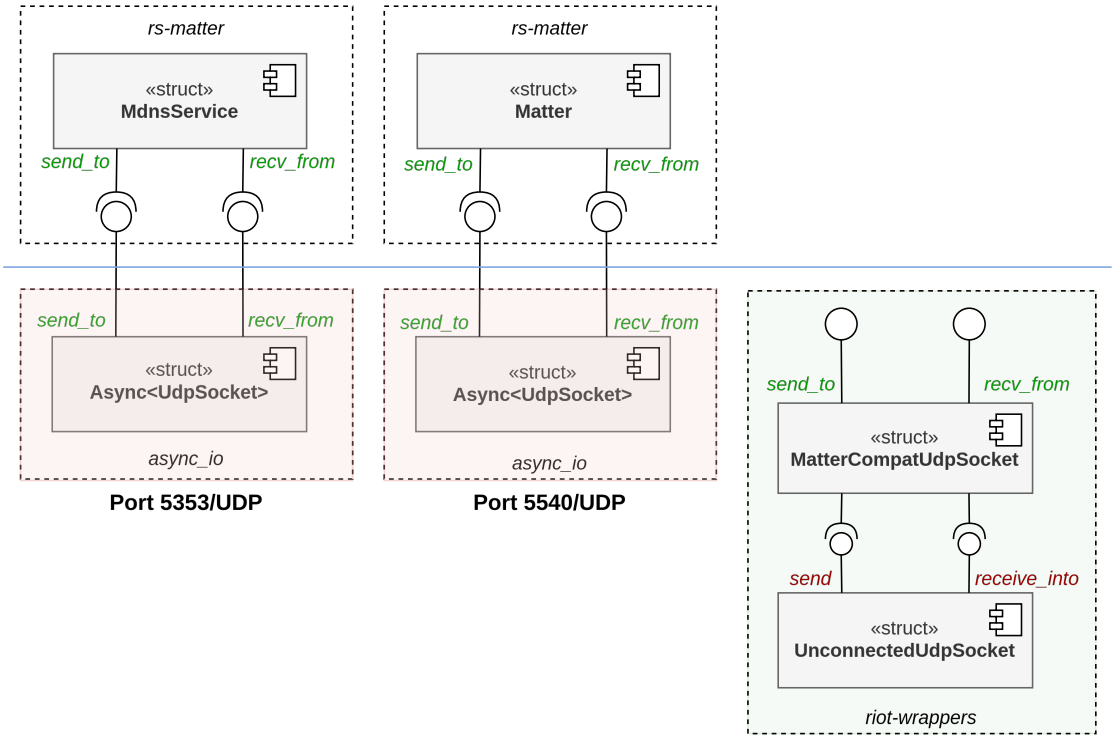

Abbildung 4.3: Async UDP-Socket aus `std::net` (rot) und RIOT Socket mit dem zu entwickelnden Wrapper (grün)

Mit den Traits `UdpSend` bzw. `UdpReceive` (siehe Listing A.1) wird in rs-matter spezifiziert, welche Schnittstellen ein verwendeter UDP-Socket implementieren muss.

In Abbildung 4.3 wird das erste von zwei Problemen dargestellt. Dieses besteht darin, dass der `UnconnectedUdpSocket` aus riot-wrappers die erforderlichen Schnittstellen

(Funktions-Signaturen) `send_to` und `recv_from` nicht implementiert. Es ist auch dargestellt, wie die Demo-Anwendung aus rs-matter [13, `examples/onoff_light`] den `Async`-Wrapper aus dem Crate `async_io` mit dem `UdpSocket` aus der Rust Standard-Library verwendet (rot markiert), um die erforderlichen Schnittstellen bereitzustellen. Für die Portierung nach RIOT OS wird die Entwicklung eines Wrappers (im Weiteren als `MatterCompatUdpSocket` bezeichnet) benötigt, welcher die Kompatibilität zwischen den Schnittstellen herstellt.

Der Hintergrund des zweiten Problems wird nachfolgend erklärt und bezieht sich auf die Funktionsweise des Rust-Compilers und die UDP-Socket Traits aus embedded-nal, welche der `UnconnectedUdpSocket` aus riot-wrappers implementiert. Der mDNS Service aus rs-matter erfüllt zwei Aufgaben: (i) Antwort auf eingehende DNS-SD Queries (Request-Response) und (ii) Senden von DNS-SD Responses an die mDNS Multicast-Adresse ("Fire and forget"). Der verwendete UDP-Socket muss also nicht nur ein klassisches Request-Response Schema unterstützen, sondern auch ermöglichen, dass jederzeit gesendet werden kann. In rs-matter wird auch diese Anforderung in den Traits `UdpSend` bzw. `UdpReceive` spezifiziert, indem eine *mutable shared reference* auf den UDP-Socket gefordert wird. Die Signatur der `run`-Funktionen des mDNS und Matter Service beinhalten zwei Argumente, in denen jeweils eine Referenz auf denselben Socket übergeben werden muss. Eine Übergabe von zwei *mutable shared references* auf den `UnconnectedUdpSocket` verbietet jedoch der Rust-Compiler. Eine *immutable shared reference* wäre erlaubt, allerdings werden dadurch die zuvor beschriebenen Anforderungen aus den Traits `UdpSend` bzw. `UdpReceive` nicht erfüllt und das Kompilieren ist auch nicht möglich.

Eine mögliche Lösung wäre ein "Splitting" des Sockets in zwei Hälften, jeweils für das Senden und Empfangen. Weitere Informationen über diese bereits vorgeschlagene Lösung und den Hintergrund findet sich im Pull Request 106 in embedded-nal [33]. Solange diese Funktion in riot-wrappers jedoch nicht unterstützt wird, muss der zu entwickelnde `MatterCompatUdpSocket` dies unterstützen, indem die Referenz auf den `UnconnectedUdpSocket` mit einem Mutex geschützt wird.

Implementierung des Wrappers

Der Wrapper (`MatterCompatUdpSocket`) ist *Owner* des `UnconnectedUdpSocket`. Da ein gleichzeitiges Aufrufen der `send_to` und `recv_from` Funktionen möglich ist und

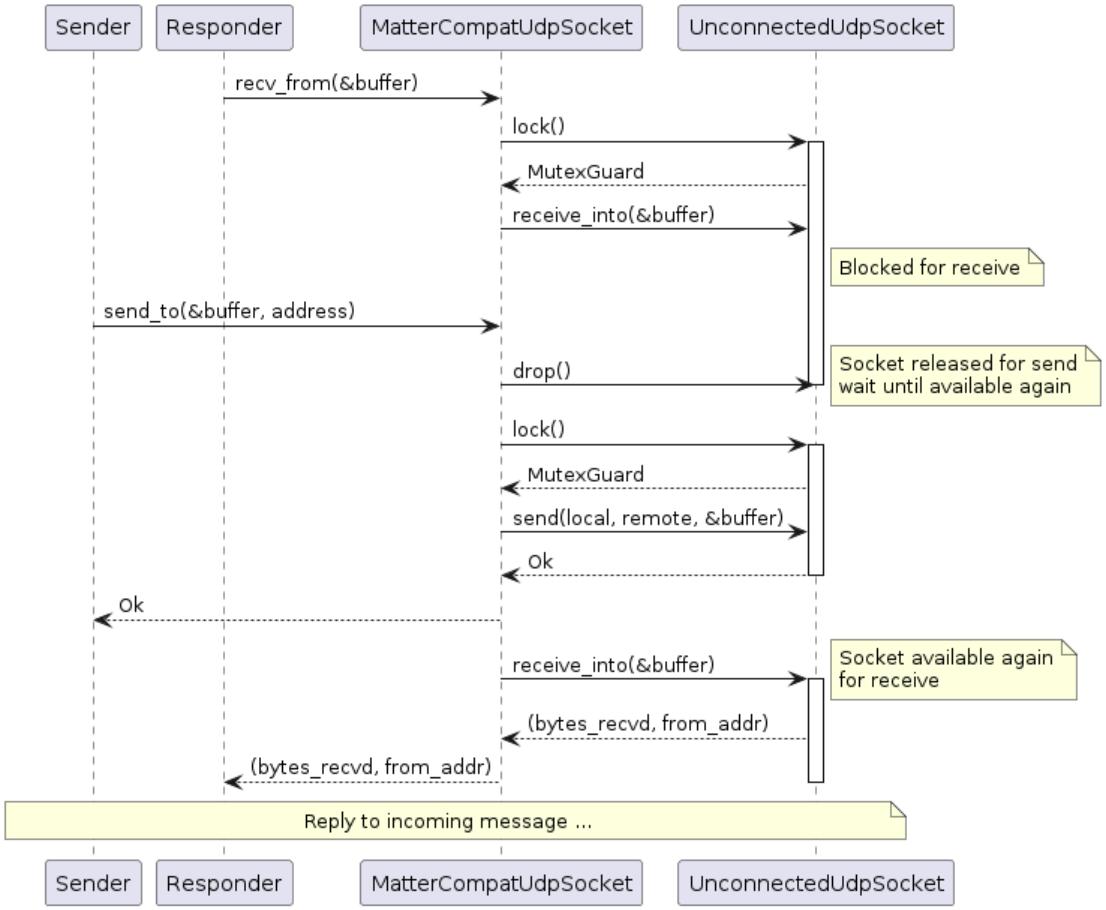

Abbildung 4.4: Koordination des entwickelten `MatterCompatUdpSocket` am Beispiel des mDNS Service

in der Signatur (siehe Traits in Listing A.1) eine *shared mutable reference* auf den Socket gefordert ist, verbietet Rust das Kompilieren, um Race Conditions zu verhindern.

Um die durch Einführung des Mutex möglich gemachten Deadlocks zu vermeiden und damit das abwechselnde Senden und Empfangen von Paketen zu ermöglichen, wird eine Koordination benötigt, welche in Abbildung 4.4 veranschaulicht wird. Der *Sender* des mDNS Service sendet dabei in regelmäßigen Abständen eine DNS-SD Response an die mDNS Multicast-Adresse und muss gleichzeitig in der Lage sein, auf eingehende DNS-SD Anfragen am gleichen Socket antworten zu können (vgl. 2.3.8 „Discovery und Advertising“). Wenn der *Responder* auf eingehende Pakete wartet, ist der Mutex des `UnconnectedUdpSocket` blockiert und der *Sender* kann in dieser Zeit nichts senden. Dieses Problem wird dadurch gelöst, dass die Funktion `send_to` die `recv_from` Funktion aus der *receive*-Schleife aufwecken kann, woraufhin der Mutex für den `UnconnectedUdpSocket` freigegeben wird. Nachdem das Paket gesendet wurde, wird die `recv_from` Funktion benachrichtigt, woraufhin diese wieder auf eingehende Pakete warten kann.

Die Koordination wird durch das Senden von asynchronen *Notifications* zwischen Sender und Empfänger umgesetzt (siehe Listing A.2). Der Empfänger blockiert dabei, bis entweder ein neues UDP-Paket oder eine `Notification` empfangen wurde. Wenn der Sender fertig ist, wird eine weitere Notification an den Empfänger gesendet, um zu signalisieren, dass der UDP-Socket wieder “frei” ist.

4.3.2 Multicast DNS

Damit DNS-SD Queries über mDNS empfangen werden können, muss der Multicast-Gruppe `ff02::fb` beigetreten werden. Da in riot-wrappers dafür bisher keine Funktion existiert, wird diese im Modul `gnrc::netapi` implementiert.

4.3.3 Random Number Generator

`rs-matter` erfordert die Definition einer Funktion, welche ein Byte-Array mit Zufallszahlen unter Verwendung eines Random Number Generator (RNG) befüllt, um die in [16, S. 63] beschriebenen kryptografischen Anforderungen von Matter zu erfüllen. In riot-wrappers wird dafür das `random`-Modul verwendet. Die Bewertung einer geeigneten Implementierung eines RNG ist nicht Bestandteil der Arbeit (eine Empfehlung findet sich in [34]), diese kann jedoch leicht durch Einbinden entsprechender Module im Makefile

der Anwendung gewählt werden (z. B. `periph_hwrng` oder `puf_sram`). Im Rahmen dieser Arbeit wird der HWRNG des Entwicklungsboards verwendet.

4.3.4 Ermittlung der aktuellen Systemzeit

Der *Matter Service* aus `rs-matter` erfordert die Implementierung einer Funktion, welche die aktuelle Systemzeit in Sekunden seit UNIX Epoche (01.01.1970 00:00:00) zurückgibt. Unter RIOT OS wird hierfür eine auf der Zielhardware verfügbare RTC verwendet, was durch `FEATURES_OPTIONAL += periph_rtc` im `Makefile` angegeben wird. Falls keine RTC vorhanden, wird die Systemzeit mit dem Modul `ztimer_sec` emuliert durch Addieren der Sekunden nach Boot plus RIOT Epoche (01.01.2000 00:00:00). Da die benötigten Funktionen des RTC-Moduls in riot-wrappers zum Zeitpunkt dieser Arbeit noch nicht existieren, werden die Funktionen aus riot-sys [35] direkt aufgerufen.

4.3.5 Auslesen von Daten aus der Firmware

Die für eine Matter-Anwendung spezifischen Daten für die Device Attestation und das Commissioning müssen in der Firmware des Gerätes gespeichert und vom Matter Service ausgelesen werden können. Durch Implementierung des `FirmwareDataFetcher` werden Funktionen bereitgestellt zum Auslesen der Daten unter Verwendung des RIOT `vfs`-Moduls. Bei Verwendung des `FirmwareDataFetcher` in einer Matter-Anwendung wird erwartet, dass nach Programmstart folgende Dateien im `constfs` Dateisystem existieren:

- `/const/passcode`: Passcode (27 Bit) für das Commissioning
- `/const/discriminator`: Discriminator (12 Bit) für die Discovery
- `/const/pai`: Product Attestation Intermediate (PAI) Certificate
- `/const/dac`: Device Attestation Certificate (DAC)
- `/const/dac_pubkey`: DAC Public Key
- `/const/dac_privkey`: DAC Private Key
- `/const/cd`: Certification Declaration (CD)

4.3.6 Datenpersistierung

Der Matter Service implementiert Funktionen zum Laden und Speichern von Fabric- und Access Control List (ACL)-Daten, sodass Konfigurationen nach einem Neustart des Gerätes erhalten bleiben. In rs-matter existiert im Modul `persist` eine fertige Implementierung eines *Persistence Managers*, welcher die Daten unter Verwendung der Rust Standard-Library im Dateisystem speichert und ausliest.

Für die Portierung nach RIOT OS soll diese Funktionalität durch Schreiben und Lesen der Daten aus einem Memory Technology Device (MTD) unterstützt werden. Dafür muss je nach verwendeter Speichertechnologie das passende `mtd`-Modul eingebunden werden. Zum Zeitpunkt dieser Arbeit existieren unter anderem MTD-Wrapper für internen Flash-Speicher, EEPROM's oder SD-Karten. Da zum Zeitpunkt dieser Arbeit keines der verfügbaren `mtd`-Module in riot-wrappers implementiert ist, wird die Erweiterung vorgeschlagen (siehe [36]) und vorerst lediglich das Struct `PersistenceManager` erstellt, mit der Möglichkeit zur Erweiterung sobald das Schreiben und Auslesen von MTDs unterstützt wird.

4.4 Implementierung des On/Off Light

Für die Entwicklung des *On/Off Light* wird die Demo-Anwendung aus `rs-matter` [13, `examples/onoff_light`], die unter Verwendung der Rust Standard-Library auf Linux und macOS ausgeführt werden kann, so angepasst dass sie unter RIOT OS lauffähig ist.

Die Anwendung wird in einem Fork von RIOT-OS im Ordner `examples/rust-matter-light` erstellt (siehe Pull Request [31]). Der Quellcode der Anwendung ist in einer Rust-Library geschrieben (Einstiegspunkt: `main`-Funktion in `src/lib.rs`), die nach dem Aufruf von `make all` kompiliert und mit dem Programm gelinkt wird. In der Datei `Cargo.toml` werden die benötigten Dependencies der Rust-Library definiert.

4.4.1 Implementierung des Datenmodells

Das in Kapitel 3.4 beschriebene Datenmodell des On/Off Light lässt sich durch Erstellung des Structs `Node` (aus `rs_matter::data_model::objects::node`) mit den unterstützen Gerätetypen, Endpunkten und Clustern initialisieren.

Für den Basic Information Cluster wurden folgende Werte initialisiert:

Attribut	Typ	Wert
Vendor ID	`u16`	`0xFFF1`
Product ID	`u16`	`0x8000`
Hardware Version	`u16`	`2`
Software Version	`u32`	`1`
Software Version String	`&str`	`1`
Serial Number	`&str`	`aabbccdd`
Device Name	`&str`	`OnOff Light`
Product Name	`&str`	`Light123`
Vendor Name	`&str`	`Vendor PQR`

Tabelle 4.1: Attributswerte des Basic Information Cluster

4.4.2 Onboarding Payload

Die Onboarding Payload enthält folgende für die Discovery und das Commissioning benötigten Werte, die bei Initialisierung des `Structs` CommissioningData (aus `rs_matter::core`) definiert werden müssen.

Name	Wert
Discriminator	`250`
Passcode	`123456`

Tabelle 4.2: Werte der Onboarding Payload des On/Off Light

4.4.3 Handling der Interaktionen

Für die Verarbeitung empfangener Interaktionen wird ein neues Struct `OnOffHandler` erstellt (siehe Listing A.3). Durch Implementierung des `Handler` Traits kann eine eigene Logik für das Handling der Interaktionen implementiert werden. Bei der Initialisierung des *Matter Service* wird der `OnOffHandler` dann als Argument übergeben.

4.4.4 Speichern firmwarespezifischer Daten

Für die in 4.3.5 „Auslesen von Daten aus der Firmware“ beschriebene Implementierung müssen die vom `FirmwareDataFetcher` erwarteten Daten für das Commissioning und die Device Attestation im `constfs` Dateisystem gespeichert werden. Dies wird bei Start der Anwendung durch Aufruf externer C-Funktionen aus dem `vfs`-Modul umgesetzt, indem zuerst das `constfs` Dateisystem initialisiert und danach die notwendigen Daten geschrieben werden. Die Existenz der erforderlichen Dateien und der korrekte Inhalt wurde mit dem Shell-Kommando `vfs` verifiziert.

```
> vfs ls /const
# passcode  4 B
# discriminator  2 B
# pai 463 B
# dac  492 B
# dac_pubkey  65 B
# dac_privkey  32 B
# cd  541 B
# total 7 files

> vfs r passcode 4 0
# 00000000: 40e2 0100
```

Listing 4.1: Anzeige der im `constfs` Dateisystem gespeicherten Daten und Auslesen des Passcodes

Aufgrund eines vermutlichen Fehlers in riot-wrappers (siehe erstellte Issue [37]) war das Auslesen der Daten mit dem `FirmwareDataFetcher` nicht möglich, weshalb die Daten in der On/Off Light Anwendung “hardcodiert” bereitgestellt wurden.

4.4.5 Datenpersistierung

Da wie in 4.3.6 „Datenpersistierung“ bereits beschrieben das Speichern von Daten während der Laufzeit noch nicht unterstützt werden kann, ist nach einem Geräteneustart ein erneutes Commissioning und Konfiguration des On/Off Light notwendig. Für die zukünftige Unterstützung der Datenpersistierung nach vollständiger Implementierung des

`PersistenceManager` in riot-wrappers kann das in der Anwendung definierte Feature `psm` durch Hinzufügen der Zeile `CARGO_OPTIONS += --features psm` aktiviert werden.

4.4.6 Programmablauf

Mit der angepassten Matter-Library (siehe Kapitel 4.2), dem in 4.3 „Integration von Matter in RIOT OS“ beschriebenen *Kompatibilitätslayer* sowie den applikationsspezifischen Initialisierungen kann die Anwendung nun gestartet werden. Die wesentlichen Schritte des Programmablaufs werden nachfolgend anhand Abbildung 4.5 beschrieben.

Initialisierung Zu Programmstart wird der globale Logger initialisiert, womit Infos, Warnungen und Fehlermeldungen der Matter-Library in der RIOT-Shell ausgegeben werden. Zudem wird das Virtual File System Layer (VFS)-Modul initialisiert und die in 3.7 „Device Attestation“ beschriebenen Daten aus dem Speicher gelesen. Durch Starten eines separaten Threads für die RIOT Shell können eingegebene Kommandos verarbeitet werden. In einem weiteren Thread für den Matter, mDNS Service und den PersistenceManager wird in *Init Network* zuerst die Link-Local IPv6-Adresse der Netzwerkschnittstelle für das Senden in DNS-SD Antworten abgefragt und der mDNS Multicast-Adresse beigetreten.

Start des mDNS Service In einem *async task* wird der UDP-Socket initialisiert, an den mDNS-Port `5353` gebunden und der *mDNS Service* gestartet

Start des Matter Service In einem *async task* werden die in 3.6 „Commissioning“ beschriebenen Daten, der in 4.4.3 „Handling der Interaktionen“ beschriebene *Interaction Handler*, sowie der UDP-Socket initialisiert, welcher an den mDNS-Port `5540` gebunden wird. Danach wird der *Matter Service* gestartet

Start des PersistenceManager Falls das `psm`-Feature aktiviert ist, wird der `PersistenceManager` in einem *async task* gestartet, welcher während der Laufzeit gespeicherte Einstellungen speichert und ausliest

4.4.7 Ermittlung und Optimierung der Speicherbelegung

Während des Tests der Implementierung trat ein Stack Overflow auf. Ursache dafür war, dass ein zu geringer Speicher der für den Stack reserviert wurde. Daraufhin wurde die Grö-

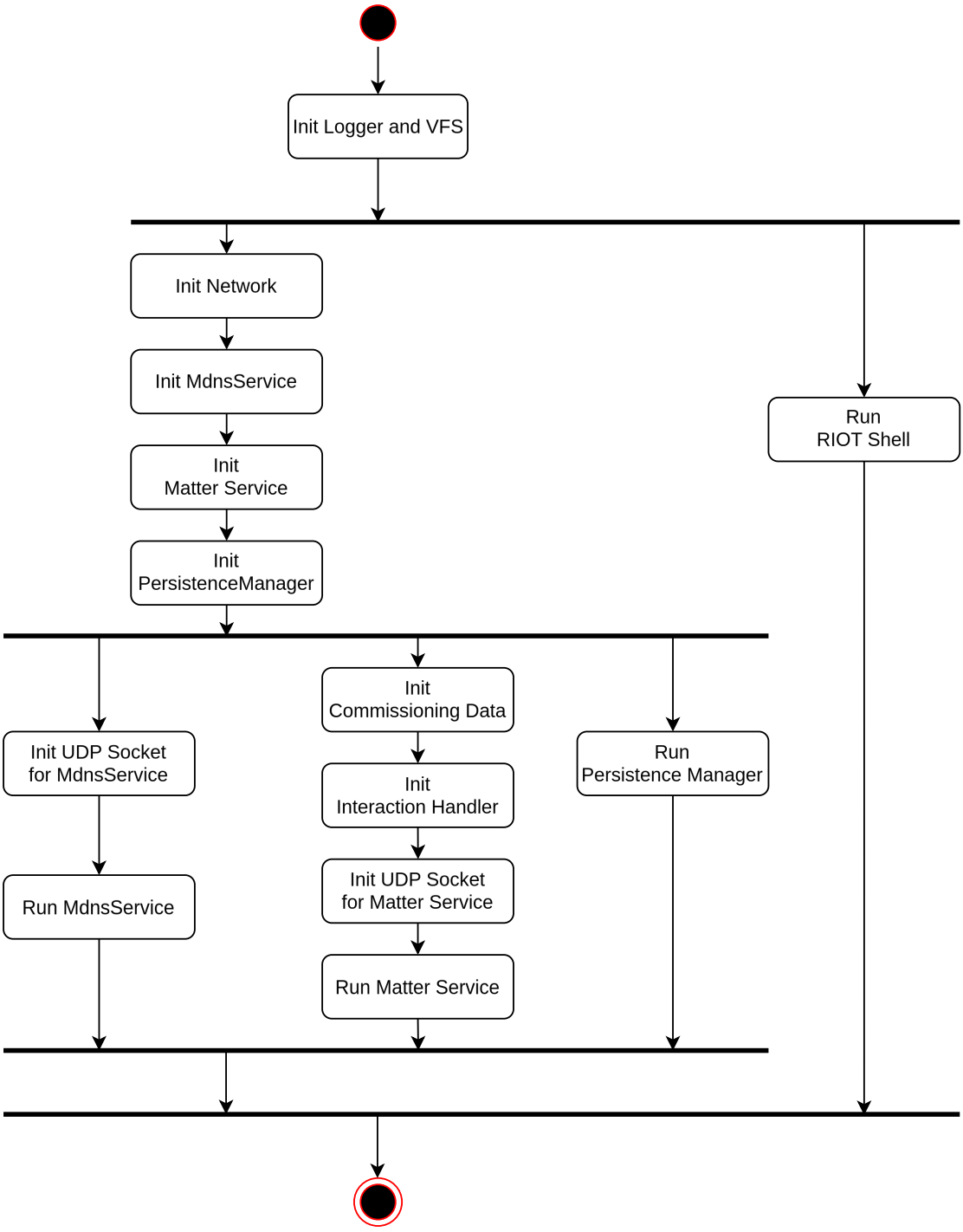

Abbildung 4.5: Programmablauf des On/Off Light

ße der im Programm verwendeten Datenstrukturen analysiert. Kritisch sind hier vor allem die Buffer für UDP-Pakete, die ca. 27 KiloByte belegen. Als Maßnahme wurde die Stackgröße im Makefile durch Überschreiben des Compilerflag `THREAD_STACKSIZE_MAIN` um 50 kB erhöht. Zudem wurden Datenstrukturen wo möglich im statischen Speicher angelegt.

Durch Ausführen des Befehls `make info-buildsize` wurde der benötigte Speicher für die verwendete Hardware ermittelt. Für das verwendete Board `nucleo-f429zi` werden mindestens **190 kB** RAM und **351 kB** ROM benötigt.

4.5 Implementierung des On/Off Light Switch

Da die `rs-matter` Library zum Zeitpunkt dieser Arbeit die Implementierung eines On/Off Light Switch nicht unterstützt, wird dafür eine fertig implementierte Demo-Anwendung aus der *Espressif Matter SDK* [10] verwendet. Durch Ausführen von Befehlen in der eingebauten Shell können die *Invoke Interactions* `Off`, `On` und `Toggle` an das On/Off Light gesendet werden.

Für die Implementierung des On/Off Light Switch wurde zuerst die Anleitung in [38] befolgt, um die Entwicklungsumgebung für *Espressif's SDK for Matter* einzurichten. Danach wurde die Beispiel-Anwendung aus [10, `examples/light_switch`] auf das Entwicklungsboard `ESP32-C3-DevKitC-02` programmiert (siehe Listing A.4).

4.6 Implementierung des Matter Controller

Als Implementierung des Matter Controller wird die CLI-Anwendung **chip-tool** aus dem GitHub-Repository der Referenzimplementierung von Matter [6, `examples/chip-tool`] verwendet. Folgende Befehle des Tools werden genutzt, um die korrekte Implementierung testen zu können:

- `interactive`: Starten einer interaktiven Session zur Ausführung mehrerer Interaktionen
- `discover`: Kommandos für das Auffinden von Matter-Nodes
- `pairing`: Durchführen des Commissioning

- `accesscontrol`: Verwalten von Berechtigungen an Nodes
- `binding`: Erstellen von Bindings zwischen Nodes
- `basicinformation`: Interaktionen mit dem *Basic Information Cluster* für das Auslesen von Geräteinformationen
- `descriptor`: Interaktionen mit dem *Descriptor Cluster* für das Auslesen von Endpunkt-Informationen
- `onoff`: Interaktionen mit dem *On/Off Cluster* des *On/Off Light*

In Listing A.5 wird gezeigt, wie die Entwicklungsumgebung der Matter SDK zuerst eingerichtet und das chip-tool gebaut wurde.

5 Evaluation

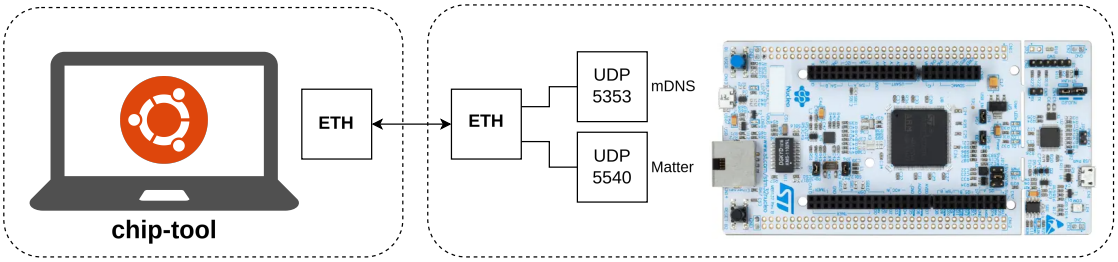

Abbildung 5.1: Versuchsaufbau zur Evaluation des On/Off Light

Für die Evaluation des unter RIOT OS entwickelten On/Off Light wird das Entwicklungsboard an die Ethernet-Schnittstelle des Entwicklungs-PCs angeschlossen. Auf diesem ist das **chip-tool** für das Betriebssystem Ubuntu (Version 22.04.3) installiert, welches für das Testen der in 3.9 „Zusammenfassung der Anforderungen“ beschriebenen Anforderungen verwendet wird. Für die Evaluation des Advertising- und Discovery-Verhaltens über mDNS wurde die CLI-Anwendung **avahi-browse** und **Wireshark** verwendet.

5.1 Initialisierung der Netzwerkschnittstelle und RTC

Nach Einschalten des On/Off Light wird mit dem Befehl `ifconfig` die Konfiguration der Ethernet-Schnittstelle, die Link-Local und Multicast-Adressen angezeigt. Anschließend wurde erfolgreich die Erreichbarkeit mit dem CLI-Tool `ping6` getestet.

Zudem wird die RTC auf die aktuelle Zeit eingestellt.

```
> rtc settime 2024-04-30 09:00:00
```

Listing 5.1: Einstellen der Systemzeit des RTC über die RIOT Shell

5.2 Verhalten nach dem Einschalten des Gerätes

Nach dem Flashen der On/Off Light Anwendung auf das Entwicklungsboard und Verbinden mit der Konsole erscheint der `Pairing Code` (Onboarding Payload) in textueller Form und als QR-Code, welcher von mobilen Apps für den Start des Commissioning gescannt werden kann.

Abbildung 5.2: Log-Ausgabe des On/Off Light nach dem Einschalten

5.3 Commissionable Node Discovery

Ein Commissioner muss das On/Off Light im Netzwerk finden können, indem er eine DNS-SD Anfrage an die Multicast-Gruppe `ff02::fb` sendet. Darauf wird eine DNS-SD Response mit dem Service-Typ `_matterc._udp` [16, S. 245] erwartet, wodurch das Gerät als Commissionable erkannt wird. Im TXT-Record werden die für das Commissioning benötigten Informationen mitgeteilt. Für das On/Off Light wird hier erwartet:

- `D = 250`: Discriminator

- `CM = 1`: Gerät befindet sich im Commissioning Mode und wartet auf Commissioning mit Passcode
- `VP` (optional): Vendor ID und Product ID
- `T = 0` (optional): TCP wird nicht unterstützt, nur UDP

Das On/Off Light kann als Commissionable mit `avahi-browse` und der `chip-tool` Anwendung über DNS-SD gefunden werden. Dabei werden auch die für den Commissioner notwendigen Informationen in einem TXT-Record gesendet.

```
$ avahi-browse _matterc._udp -r
= enx000ec6bf033f IPv6 8C00000000000000
 ↪  _matterc._udp        local
hostname = [riot-matter-demo.local]
address = [fe80::a431:56ff:fe90:18f4]
port = [5540]
txt = ["PI=" "PH=33" "SAI=300" "SII=5000" "VP=65521+32768"
 ↪  "DN=OnOff Light" "CM=1" "D=250"]

$ chip-tool discover commissionables
Discovered node:
Hostname: riot-matter-demo
IP Address #1: fe80::a431:56ff:fe90:18f4
Port: 5540
Mrp Interval idle: 5000 ms
Mrp Interval active: 300 ms
Mrp Active Threshold: not present
TCP Supported: 0
ICD: not present
Device Name: OnOff Light
Vendor ID: 65521
Product ID: 32768
Long Discriminator: 250
Pairing Hint: 33
Instance Name: 3BE684D404CD18C3
Commissioning Mode: 1
```

Listing 5.2: Commissionable Node Discovery Evaluation mit avahi-browse

Zudem wurde mit Wireshark aufgezeichnet, dass das On/Off Light alle 30 Sekunden eine DNS-SD Antwort an die mDNS Multicast-Adresse `ff02::fb` sendet.

5.4 Commissioning

Das Commissioning des On/Off Light wird mit Angabe der gewählten Node-ID `0x1000` und des in der Konsole ausgegebenen Pairing Code mit dem Befehl `chip-tool pairing code <node-id> <payload>` gestartet. Nach der *Commissionable Node Discovery* wird das Commissioning durchgeführt und mit Erfolg in der Konsole bestätigt.

```
$ chip-tool pairing code 0x1000 00876800071
CHIP:CTL: Discovered device to be commissioned over DNS-SD
...
CHIP:TOO: Device commissioning completed with success
```

Listing 5.3: Commissioning des On/Off Light mit dem chip-tool als Matter Controller

5.5 Operational Node Discovery

Auch nach dem Commissioning muss das On/Off Light über das Verfahren *IP Network Discovery* auffindbar sein, mit dem Unterschied dass der DNS-SD Servicetyp `_matter._tcp` lautet. Die Operational Node Discovery wurde mit avahi-browse und dem chip-tool evaluiert.

```
$ avahi-browse _matter._tcp -r
= enx083a885b3eb2 IPv6 AF4365025C0394AC-0000000000001000
 ↪  _matter._tcp         local
hostname = [riot-matter-demo.local]
address = [fe80::a431:56ff:fe90:18f4]
port = [5540]
txt = []

$ chip-tool interactive start
Fabric index 0x1 was retrieved from storage. Compressed
 ↪  FabricId 0xAF4365025C0394AC, FabricId 0x0000000000000001,
 ↪  NodeId 0x000000000001B669, VendorId 0xFFF1
$ discover resolve 0x1000 0xAF4365025C0394AC
Lookup started for AF4365025C0394AC-0000000000001000
NodeId Resolution: 4096 at
 ↪  UDP:[fe80::a431:56ff:fe90:18f4%enx083a885b3eb2]:5540
MRP retry interval (idle): 500ms
MRP retry interval (active): 300ms
Supports TCP: no
ICD is operating as: SIT
```

Listing 5.4: Operational Node Discovery mit avahi-browse

5.6 Binding des On/Off Light Switch an das On/Off Light

Um den *On/Off Light Switch* an das *On/Off Light* zu koppeln, muss dem ersteren zuerst Zugriffsrechte gewährt werden, um das Binding erstellen zu können. Dies geschieht durch Anlegen eines Eintrags im `ACL` Attribut des `Access Control` Cluster Server des *On/Off Light* mit den folgenden Daten:

- `Privilege = 5`: Admin Privileg
- `AuthMode = 2`: Authentifizierung per CASE
- `Subjects = 0x1001`: Node ID des On/Off Light Switch

- `Targets = 0`: Target ist ein Cluster (On/Off)

Nach Erteilen der Berechtigung kann das Binding des *On/Off Cluster* des On/Off Light Switch (Client) an den des On/Off Light (Server) durchgeführt werden. Dies geschieht durch das Anlegen eines Eintrags im `Binding`-Attribut des *Binding Cluster*:

- `Node = 0x1000`: Node ID des On/Off Light
- `Endpoint = 1`: Endpunkt mit dem Gerätetyp `On/Off Light`
- `Cluster = 6`: On/Off Cluster

Die zuvor beschriebenen Schritte wurden mit dem chip-tool wie folgt ausgeführt:

```
$ chip-tool accesscontrol write acl '[{"privilege": 5,
↪  "authMode": 2, "subjects": [ 112233, 4097 ], "targets":
↪  null}]' 0x1000 0x0
$ chip-tool binding write binding '[{"node":4096, "endpoint":1,
↪  "cluster":6}]' 0x1001 0x1
```

Listing 5.5: Erstellen des Bindings zwischen On/Off Light Switch und On/Off Light

5.7 Interaktionen

Nachfolgend werden die unterstützten Interaktionen (`Invoke`, `Read` und `Write`) des On/Off Light über den Austausch von Nachrichten im Interaction Model Protocol (IMP) getestet. Als Controller wird zuerst das `chip-tool` verwendet, das die Interaktionen an das On/Off Light sendet. Danach werden die Interaktionen vom On/Off Light Switch an den gekoppelten Server Cluster des On/Off Light gesendet. Dem On/Off Light wurde während des Commissioning die Node ID `0x1000` und dem On/Off Light Switch die Node ID `0x1001` zugewiesen, über die die Adressierung innerhalb der Fabric erfolgt.

5.7.1 Root Node Endpoint

Die unterstützen Cluster des Root Node Endpunktes wurden gemäß den Anforderungen in 3.1 „Anforderungen an Server-Cluster des Root Node Endpunktes (`X`: erforderlich, –: nicht erforderlich)“ durch Auslesen des Descriptor Clusters an Endpunkt 0 validiert. Das

Attribut `PartsList` enthält einen Eintrag mit der Endpunkt-ID `1` für den *On/Off Light Endpoint*. Zudem können die korrekten Basis-Geräteinformationen wie z. B. Vendor ID und Product ID über den *Basic Information Cluster* ausgelesen werden.

```
$ chip-tool descriptor read server-list 0x1000 0
CHIP:TOO:   ServerList: 10 entries
CHIP:TOO:     [1]: 29
CHIP:TOO:     [2]: 40
CHIP:TOO:     [3]: 48
CHIP:TOO:     [4]: 49
CHIP:TOO:     [5]: 60
CHIP:TOO:     [6]: 62
CHIP:TOO:     [7]: 31
CHIP:TOO:     [8]: 51
CHIP:TOO:     [9]: 55
CHIP:TOO:     [10]: 63

$ chip-tool descriptor read parts-list 0x1000 0
CHIP:TOO:   PartsList: 1 entries
CHIP:TOO:     [1]: 1

$ chip-tool basicinformation read vendor-id 0x1000 0
CHIP:TOO:   VendorID: 65521

$ chip-tool basicinformation read product-id 0x1000 0
CHIP:TOO:   ProductID: 32768
```

Listing 5.6: Auslesen der Geräteeigenschaften des On/Off Light durch Interaktion mit dem Root Node Endpunkt

Die Unterstützung von *Write Interactions* wurde durch Änderung des Attributs `NodeLabel` erfolgreich getestet.

```
$ chip-tool basicinformation write node-label "Hello RIOT"
↪  0x1000 0
$ chip-tool basicinformation read node-label 0x1000 0
CHIP:TOO:   NodeLabel: Hello RIOT
```

Listing 5.7: Test der Unterstützung von Write Interaktionen mit dem chip-tool

5.7.2 On/Off Light Endpoint

Das Attribut `DeviceTypeList` des Descriptor-Cluster an Endpunkt 1 enthält ein Element mit dem Wert `256 = 0x0100` (On/Off Light) und `ServerList` zwei Einträge mit den Werten `29` (Descriptor Cluster) und `6` (On/Off Cluster).

```
1  $ chip-tool descriptor read device-type-list 0x1000 1
2  CHIP:TOO:   DeviceTypeList: 1 entries
3  CHIP:TOO:     [1]: {
4          CHIP:TOO:       DeviceType: 256
5          CHIP:TOO:       Revision: 2
6          CHIP:TOO:      }
7
8  $ chip-tool descriptor read server-list 0x1000 1
9  CHIP:TOO:   ServerList: 2 entries
10 CHIP:TOO:     [1]: 29
11 CHIP:TOO:     [2]: 6
```

Listing 5.8: Interaktionen des chip-tool mit dem Descriptor Cluster des On/Off Light Endpoint

Die unterstützten Interaktionen (*Read* und *Invoke*) am On/Off Cluster können wie folgt mit dem `chip-tool` getestet werden:

```
$ chip-tool onoff read on-off 0x1000 1
Data = false

$ chip-tool onoff on 0x1000 1
status = 0x00 (SUCCESS)

$ chip-tool onoff read on-off 0x1000 1
Data = true

$ chip-tool onoff off 0x1000 1
status = 0x00 (SUCCESS)

$ chip-tool onoff toggle 0x1000 1
status = 0x00 (SUCCESS)
```

Listing 5.9: Interaktionen des chip-tool mit dem On/Off Light Endpoint

5.7.3 Interaktionen zwischen On/Off Light Switch und On/Off Light

Im Terminal des On/Off Light Switch wurden die Kommandos `Off`, `On` und `Toggle` im Format `matter esp client invoke <fabric_index> <node_id> <endpoint_id> <cluster_id> <command_id> <parameters>` an das gekoppelte On/Off Light gesendet und erfolgreich getestet.

```
> matter esp client invoke 0x1 0x1000 0x1 0x6 0x0
> matter esp client invoke 0x1 0x1000 0x1 0x6 0x1
> matter esp client invoke 0x1 0x1000 0x1 0x6 0x2
```

Listing 5.10: Interaktionen des On/Off Light Switch mit dem On/Off Light

5.8 Unterstützung weiterer Link Layer

Wegen mangelnder Unterstützung durch RIOT-Boards, die Wi-Fi sowie das Kompilieren von Rust-Code und der weiteren benötigten Module und Features unterstützen, konnte

die für Matter definierten Unterstützung des `IEEE 802.11` Link Layer nicht getestet werden. Die Unterstützung von BLE war nicht Bestandteil dieser Arbeit und wird zum aktuellen Zeitpunkt auch nicht von der `rs-matter` Library für das Commissioning unterstützt.

Allerdings konnte die *On/Off Light* Anwendung auf ein *nRF 52840 DK* Entwicklungsboard geflasht werden, welches über ein IEEE 802.15.4 Radio verfügt. In einem Versuchsaufbau mit einem Border Router (Beispiel aus RIOT OS Repository) konnte erfolgreich die automatische Konfiguration einer IPv6-Adresse mit Global Scope (Listing 5.11) und die Erreichbarkeit über 6LoWPAN per ICMPv6 Ping (Listing 5.12) getestet werden.

```
> ifconfig
  Iface  5  HWaddr: 1F:B9  Channel: 26  NID: 0x23  PHY: O-QPSK
                          ...
            Link type: wireless
            inet6 addr: fe80::5cef:e8ca:a044:9fb9  scope: link
             ↪  VAL
            inet6 addr: 2001:db8::5cef:e8ca:a044:9fb9  scope:
             ↪  global  VAL
            ...
```

Listing 5.11: Netzwerkkonfiguration des On/Off Light auf einem `nrf52840dk` Entwicklungsboard mit `IEEE 802.15.4` Radio

```
> ping 2001:db8::5cef:e8ca:a044:9fb9
  12 bytes from 2001:db8::5cef:e8ca:a044:9fb9: icmp_seq=0
   ↪  ttl=64 rssi=-168 dBm time=7.701 ms
```

Listing 5.12: Ping des On/Off Light durch einen 6LoWPAN Border Router

6 Zusammenfassung und Ausblick

Die in Kapitel 1.1 definierten Ziele für diese Arbeit werden nachfolgend anhand der umgesetzten Implementierung verglichen. Danach werden die benötigten Schritte für die Entwicklung zukünftiger Matter-Anwendungen unter RIOT OS auf Basis der gewonnenen Erkenntnisse zusammengefasst. Dazu werden Vorschläge gemacht, wie die durchgeführten Arbeiten besser in das Build-System von RIOT OS integriert und wie die verwendeten Rust-Libraries weiterentwickelt werden können, um eine nahtlose Integration von Matter in RIOT OS voranzutreiben.

6.1 Bewertung der umgesetzten Ziele

Das primäre Ziel der Portierung von Matter nach RIOT OS konnte durch die Entwicklung eines *Kompatibilitätslayers*, der Verwendung einer externen Matter-Library, bestehender RIOT-Module erfolgreich umgesetzt und durch die Entwicklung einer Demo-Anwendung validiert werden. Im Vergleich mit den in 2.1 „Verwandte Arbeiten“ beschriebenen Alternativen bietet RIOT OS als offenes, herstellerunabhängiges Betriebssystem den großen Vorteil, dass die Entwicklung von Matter-Anwendungen nicht nur auf CPU's bzw. Entwicklungsboards eines bestimmten Herstellers beschränkt ist. Auch wenn die Anwendung nur auf Native und einem Nucleo-Board getestet wurde, wird die in Kapitel 4.4 entwickelte Demo-Anwendung unter RIOT OS von 94 verschiedenen Boards unterstützt, unter anderem von den Herstellern Nordic Semiconductor, ST Microelectronics, Silicon Labs und Arduino.

Aufgrund der Entscheidung für Rust anstatt Verwendung der C++ Implementierung von Matter konnten schnell Fortschritte erzielt werden, auch dank des einfachen Paketmanagements, der Unterstützung bedingter Kompilierung für verschiedene Platformen mit *Cargo Features* sowie verschiedener CPU-Architekturen (Cross-Compilation). Der Nachteil ist jedoch, dass zum Zeitpunkt der Arbeit mit der rs-matter Library nur der Gerätetyp

On/Off Light implementiert werden kann. In C++ ließen sich sehr viel mehr Matter Features umsetzen und es bleibt abzuwarten, wie schnell diese in der Rust-Implementierung verfügbar sein werden. Zudem stellte sich heraus, dass viele Module und Packages aus RIOT OS noch nicht in riot-wrappers implementiert sind, was die Integration teilweise erschwerte.

Im Rahmen der Evaluation konnten die zum Zeitpunkt der Arbeiten unterstützten Funktionen der Matter Rust-Library erfolgreich unter RIOT OS getestet werden:

- Secure Session: PASE und CASE
- Commissionable und Operational Node Discovery über mDNS und DNS-SD
- *On-Network* Commissioning per Ethernet
- Unterstützte Endpunkte: Root Node und On/Off Light
- Unterstützte Application Cluster: On/Off
- `Invoke`, `Read` und `Write` Interaktionen

6.2 Weiterführende Arbeiten

Nachfolgend werden weitere Arbeiten vorgeschlagen, um die in dieser Arbeit umgesetzte Implementierungen weiterzuentwickeln.

6.2.1 Weiterentwicklung der Rust-Library für RIOT OS

(i) Um die Datenpersistierung in Matter-Anwendungen unter RIOT OS zu ermöglichen, wird vorgeschlagen die riot-wrappers Library so zu erweitern, dass das Speichern und Auslesen auf internen und externen Speichermedien (MTDs) während der Laufzeit ermöglicht wird (siehe erstellte Issue [36]). Eine weitere Lösung könnte ein neu zu entwickelndes RIOT-Modul sein für das Speichern von *Runtime Configurations*, wie in [39] bereits vorgeschlagen.

(ii) Nach Behebung des in [37] beschriebenen Fehlers kann der entwickelte `Firmware-DataFetcher` getestet und weiterentwickelt werden. Es sollte auch evaluiert werden, wie die Daten ggf. verschlüsselt gespeichert werden können.

(iii) Das in 4.3.1 „UDP Sock API“ beschriebene Problem, das die Entwicklung eines Wrappers für den UDP-Socket in Rust erforderte, kann gelöst werden, indem die Traits aus embedded-nal die Unterstützung des “Socket Splitting” erfordern (siehe Pull Request [33]). Danach müssen die aktualisierten Traits für den `UnconnectedUdpSocket` aus riot-wrappers implementiert werden (siehe erstellte Issue [40]) und der Wrapper wird damit obsolet.

(iv) In einem Fork der Matter-Library [28] wurde ein neues Feature eingeführt, mit dem die Library für RIOT OS kompiliert werden kann. Durch eine Weiterentwicklung von riot-wrappers wäre es jedoch möglich, dass die für RIOT OS spezifischen Implementierungen gar nicht mehr notwendig sind und somit immer die aktuelle Version von rs-matter eingebunden werden kann. Dies erfordert im Wesentlichen die Implementierung der Traits aus **embassy-time-driver** [41] und **embassy-time-queue-driver** [42] für RIOT OS unter Verwendung des `ztimer`-Moduls.

6.2.2 Demo-Anwendung On/Off Light in RIOT OS

Nachdem die im vorigen Kapitel 6.2.1 beschriebenen notwendigen Änderungen in riot-wrappers umgesetzt sind, kann in der entwickelten On/Off Light Demo-Anwendung die Datenpersistierung aktiviert werden, wodurch Matter-spezifische Einstellungen während der Laufzeit gespeichert werden. Zudem sollte eine sichere, anwenderfreundliche Möglichkeit geschaffen werden, mit der die für das Commissioning und Device Attestation notwendigen Daten in der Firmware gespeichert werden können.

6.2.3 Weiterentwicklung der Matter Rust-Library

Da sich die in dieser Arbeit verwendete Rust-Implementierung von Matter [13] auch noch im experimentellen Status befindet, ist in Zukunft mit Änderungen zu rechnen, darunter auch die Unterstützung von BLE, TCP, Thread und weiteren Gerätetypen. Dabei eventuell entstehende Inkompatibilitäten mit der RIOT-Implementierung müssen deshalb mit neuen Versionen der Library aufgelöst werden.

Literaturverzeichnis

[1] Google. Matter primer | google home developers. Abgerufen am 21.03.2024. [Online]. Available: https://developers.home.google.com/matter/primer

[2] RIOT OS. System overview - RIOT basics. Abgerufen am 29.03.2024. [Online]. Available: https://riot-os.github.io/riot-course/slides/03-riot-basics/#2

[3] R. E. Khoury. Nest outages prove that the smart home needs a local fallback. Abgerufen am 10.04.2024. [Online]. Available: https://www.androidpolice.com/2020/03/28/nest-outages-prove-that-the-smart-home-needs-a-local-fallback/

[4] Connectivity Standards Alliance Inc. Connectivity standards alliance. Abgerufen am 18.04.2024. [Online]. Available: https://csa-iot.org/

[5] Google. What is thread? | OpenThread. Abgerufen am 03.05.2024. [Online]. Available: https://openthread.io/guides/thread-primer

[6] Connectivity Standards Alliance Inc., "connectedhomeip," commit: d38a6496. [Online]. Available: https://github.com/project-chip/connectedhomeip

[7] Google, "GN - a meta-build system that generates build files for ninja." [Online]. Available: https://gn.googlesource.com/gn/

[8] ninja-build, "ninja: a small build system with a focus on speed." [Online]. Available: https://github.com/ninja-build/ninja

[9] Connectivity Standards Alliance Inc. Matter SDK documentation. Abgerufen am 12.04.2024. [Online]. Available: https://project-chip.github.io/connectedhomeip-doc/

[10] Espressif Systems, "esp-matter: Espressif's SDK for matter." [Online]. Available: https://github.com/espressif/esp-matter

[11] Espressif Systems. ESP IoT development framework. Abgerufen am 06.04.2024. [Online]. Available: https://www.espressif.com/en/products/sdks/esp-idf

[12] ivmarkov, "esp-idf-matter: Run rs-matter on espressif chips with ESP IDF." [Online]. Available: https://github.com/ivmarkov/esp-idf-matter

[13] Connectivity Standards Alliance Inc., "rs-matter: Rust implementation of the matter protocol," commit: c5cbbca. [Online]. Available: https://github.com/project-chip/rs-matter

[14] Nordic Semiconductor. Matter integration in the nRF connect SDK. Abgerufen am 12.04.2024. [Online]. Available: https://developer.nordicsemi.com/nRF_Connect_SDK/doc/latest/nrf/protocols/matter/overview/integration.html

[15] Arm Mbed. Mbed OS 6 documentation. Abgerufen am 12.04.2024. [Online]. Available: https://os.mbed.com/docs/mbed-os/v6.16/introduction/index.html

[16] Connectivity Standards Alliance Inc., "Matter core specification," version: 1.2. [Online]. Available: https://csa-iot.org/wp-content/uploads/2023/10/Matter-1.2-Core-Specification.pdf

[17] Connectivity Standards Alliance Inc., "Matter device library specification," version: 1.2. [Online]. Available: https://csa-iot.org/wp-content/uploads/2023/10/Matter-1.2-Device-Library-Specification.pdf

[18] Connectivity Standards Alliance Inc., "Matter application cluster specification," version: 1.2. [Online]. Available: https://csa-iot.org/wp-content/uploads/2023/10/Matter-1.2-Application-Cluster-Specification.pdf

[19] Connectivity Standards Alliance Inc., "Matter standard namespace specification," version: 1.2. [Online]. Available: https://csa-iot.org/wp-content/uploads/2023/10/Matter-1.2-Standard-Namespace-Specification.pdf

[20] S. Cheshire and M. Krochmal, "DNS-based service discovery," DOI 10.17487/RFC6763. [Online]. Available: https://datatracker.ietf.org/doc/rfc6763

[21] S. Cheshire and M. Krochmal, "Multicast DNS," DOI 10.17487/RFC6762. [Online]. Available: https://datatracker.ietf.org/doc/rfc6762

[22] S. Boeyen, S. Santesson, T. Polk, R. Housley, S. Farrell, and D. Cooper, "Internet x.509 public key infrastructure certificate and certificate revocation list (CRL) profile," DOI 10.17487/RFC5280 Num Pages: 151. [Online]. Available: https://datatracker.ietf.org/doc/rfc5280

[23] E. Baccelli, C. Gundogan, O. Hahm, P. Kietzmann, M. S. Lenders, H. Petersen, K. Schleiser, T. C. Schmidt, and M. Wahlisch, "RIOT: An open source operating system for low-end embedded devices in the IoT," vol. 5, no. 6, pp. 4428–4440. [Online]. Available: https://ieeexplore.ieee.org/document/8315125/

[24] RIOT OS. RIOT - the friendly OS for IoT. Abgerufen am 28.03.2024. [Online]. Available: https://github.com/RIOT-OS/RIOT

[25] RIOT OS. RIOT OS documentation. Abgerufen am 04.05.2024. [Online]. Available: https://doc.riot-os.org/index.html

[26] M. Lenders, P. Kietzmann, O. Hahm, H. Petersen, C. Gündoğan, E. Baccelli, K. Schleiser, T. C. Schmidt, and M. Wählisch, "Connecting the world of embedded mobiles: The RIOT approach to ubiquitous networking for the internet of things." [Online]. Available: http://arxiv.org/abs/1801.02833

[27] S. E. Deering and B. Hinden, "Internet protocol, version 6 (IPv6) specification," DOI 10.17487/RFC8200 Num Pages: 42. [Online]. Available: https://datatracker.ietf.org/doc/rfc8200

[28] M. Lorenz, "rs-matter: Rust implementation of the matter protocol," fork von project-chip/rs-matter. [Online]. Available: https://github.com/maikerlab/rs-matter/tree/feature/RIOT_OS

[29] RIOT OS, "rust-riot-wrappers: The 'riot-wrappers' crate, which enables high-level access to RIOT from the rust programming language." [Online]. Available: https://github.com/RIOT-OS/rust-riot-wrappers

[30] M. Lorenz, "New module for integrating matter into RIOT OS - pull request #92 in rust-riot-wrappers." [Online]. Available: https://github.com/RIOT-OS/rust-riot-wrappers/pull/92

[31] M. Lorenz, "Example: Matter on/off light - pull request #20456 in RIOT." [Online]. Available: https://github.com/RIOT-OS/RIOT/pull/20456

[32] Immunant, Inc., "c2rust: Migrate c code to rust," tag v0.18.0. [Online]. Available: https://github.com/immunant/c2rust

[33] ivmarkov, "UDP stack compatible with embassy-net; socket splitting - pull request #106 in embedded-nal." [Online]. Available: https://github.com/rust-embedded-community/embedded-nal/pull/106

[34] P. Kietzmann, T. C. Schmidt, and M. Wählisch, "A guideline on pseudorandom number generation (PRNG) in the IoT," vol. 54, no. 6, pp. 112:1–112:38, place: New York, NY, USA Publisher: ACM tex.theme: iot|nsec. [Online]. Available: https://dl.acm.org/doi/10.1145/3453159

[35] RIOT OS, "rust-riot-sys." [Online]. Available: https://github.com/RIOT-OS/rust-riot-sys

[36] M. Lorenz, "Support interfacing with MTD devices - issue #94 in rust-riot-wrappers." [Online]. Available: https://github.com/RIOT-OS/rust-riot-wrappers/issues/94

[37] M. Lorenz, "vfs: error when opening files from constfs filesystem - issue #93 in rust-riot-wrappers." [Online]. Available: https://github.com/RIOT-OS/rust-riot-wrappers/issues/93

[38] Espressif Systems. Espressif's SDK for matter - programming guide. Abgerufen am 19.03.2024. [Online]. Available: https://docs.espressif.com/projects/esp-matter/en/latest/esp32/

[39] L. J. Rosenow, "Runtime configuration of constrained devices via a shared operating system module." [Online]. Available: https://inet.haw-hamburg.de/thesis/completed/ba_lasse_rosenow.pdf/@@download/file/BA_Lasse_Rosenow.pdf

[40] M. Lorenz, "Extend API of UnconnectedUdpSocket - issue #91 in rust-riot-wrappers." [Online]. Available: https://github.com/RIOT-OS/rust-riot-wrappers/issues/91

[41] D. Nieuwenhuis, "embassy-time-driver." [Online]. Available: https://crates.io/crates/embassy-time-driver

[42] D. Nieuwenhuis, "embassy-time-queue-driver." [Online]. Available: https://crates.io/crates/embassy-time-queue-driver

[43] rust-embedded-community, "embedded-nal: An embedded network abstraction layer." [Online]. Available: https://github.com/rust-embedded-community/embedded-nal

[44] S. Klabnik and C. Nichols. The rust programming language. Abgerufen am 04.05.2024. [Online]. Available: https://doc.rust-lang.org/book/

A Anhang

A.1 Verwendete Hilfsmittel

In der Tabelle A.1 sind die im Rahmen der Bearbeitung des Themas der Bachelorarbeit verwendeten Werkzeuge und Hilfsmittel aufgelistet.

Tabelle A.1: Verwendete Hilfsmittel und Werkzeuge

Tool	Verwendung
LaTeX	Textsatz- und Layout-Werkzeug verwendet zur Erstellung dieses Dokuments
draw.io	Software verwendet zur Erstellung von Skizzen und Diagrammen
PlantUML	Online-Tool verwendet zur Erstellung von Aktivitäts- und Sequenzdiagrammen

A.2 Pull Requests und Issues auf GitHub

Tabelle A.2: Erstellte und relevante Pull Requests (PR) und Issues auf github.com

Repository	Typ	Beschreibung
rust-riot-wrappers [29]	PR #92	Entwickelter Kompatibilitätslayer für rs-matter
	Issue #91	Erweiterung des RIOT UDP Socket
	Issue #93	Fehler beim Lesen von Daten mit dem `vfs`-Modul
	Issue #94	Unterstützung des Lesen und Schreiben von MTDs in Rust
RIOT [24]	PR #20456	Demo-Anwendung On/Off Light
embedded-nal [43]	PR #106	Erweiterung der Traits für UDP-Sockets durch Unterstützung von “Socket Splitting”

A.3 Code Ausschnitte

```
pub trait UdpSend {
        async fn send_to(&mut self, data: &[u8], addr:
        ↪  SocketAddr) -> Result<(), Error>;
}

pub trait UdpReceive {
        async fn recv_from(&mut self, buffer: &mut [u8]) ->
        ↪  Result<(usize, SocketAddr), Error>;
}

pub async fn run<S, R>(&self, send: S, recv: R, udp_buffers:
↪  &mut UdpBuffers) -> Result<(), Error>
where
        S: UdpSend,
        R: UdpReceive,
{ ... }
```

Listing A.1: Definition der Traits `UdpSend` und `UdpReceive` aus rs-matter und Trait Bounds in der `run`-Funktion des `MdnsService` (Auszüge)

```rust
pub type Notification = Signal<NoopRawMutex, ()>;

async fn send_to(&mut self, data: &[u8], addr: SocketAddr) ->
↪  Result<(), Error> {
        self.release_socket_notification.signal(());
        // send packet...
        self.socket_released_notification.signal(());
}

async fn recv_from(&mut self, buffer: &mut [u8]) ->
↪  Result<(usize, SocketAddr), Error> {
        match select(
                self.release_socket_notification.wait(),
                sock.receive_into(buffer)
        ).await {
                // Handle received packet or wait for sender to
                ↪  be finished...
        }
}
```

Listing A.2: Auszüge aus der Implementierung des `MatterCompatUdpSocket` bzgl. Verwendung asynchroner `Notifications` für die Koordination

```
struct OnOffHandler {
        cluster: OnOffCluster,
        on: Cell<bool>,
}

impl Handler for OnOffHandler {
        fn read(&self, attr: &AttrDetails, encoder:
        ↪  AttrDataEncoder) -> Result<(), Error> {
                // Handle Read Interaction...
        }
        fn write(&self, attr: &AttrDetails, data: AttrData) ->
        ↪  Result<(), Error> {
                // Handle Write Interaction...
        }
        fn invoke(&self, exchange: &Exchange, cmd: &CmdDetails,
        ↪  data: &TLVElement, encoder: CmdDataEncoder) ->
        ↪  Result<(), Error> {
                // Handle Invoke Interaction...
        }
}

impl NonBlockingHandler for OnOffHandler {}
```

Listing A.3: Beispiel: Implementierung eines eigenen Handlers für den On/Off Cluster

```
$ source $HOME/esp/esp-idf/export.sh
$ source $HOME/esp/esp-matter/export.sh
$ cd $HOME/esp/esp-matter/examples/light_switch
$ idf.py set-target esp32c3
$ export ESP_MATTER_DEVICE_PATH="$HOME/esp/esp-matter
↪  /device_hal/device/esp32_devkit_c"
$ idf.py flash monitor
```

Listing A.4: Flashen des On/Off Light Switch auf einem ESP32-C3 Entwicklungsboard unter Verwendung der *Espressif's SDK for Matter* [10]

```
# Erstmaliges Einrichten der Matter SDK
$ git clone https://github.com/project-chip/connectedhomeip
 ↪  $HOME/connectedhomeip
$ cd $HOME/connectedhomeip
$ git submodule update --init

# Aktivieren der Entwicklungsumgebung
$ source scripts/activate.sh

# Installation des chip-tool unter 'out/linux-x64-chip-tool'
$ ./scripts/build/build_examples.py --target
 ↪  linux-x64-chip-tool build

$ ./out/linux-x64-chip-tool/chip-tool
Usage:
chip-tool cluster_name command_name [param1 param2 ...]
or:
chip-tool command_set_name command_name [param1 param2 ...]
```

Listing A.5: Kompilieren des `chip-tool` aus der C++ Matter SDK [6] unter Linux x86_64

Glossar

Commissionable Ein Gerät, das bereit ist einer Matter Fabric durch ein Commissioning beizutreten.

Commissionee Ein Gerät, bei der ein Commissioning zu einer Matter Node durchgeführt wird.

Commissioner Eine Matter Node, die ein Commissioning einer anderen Node durchführt.

Commissioning Hinzufügen eines Commissionable in eine Matter Fabric durch einen Commissioner durch das Zuweisen einer Operational Node ID und Node Operational Credentials (NOC).

Controller Eine Matter Node mit der Berechtigung, eine oder mehrere andere Nodes zu steuern.

Crate Kleinstmöglicher Teil an Code, den der Rust-Compiler kompilieren kann [44]. Ein Crate kann Module enthalten und wird meist entweder als Library oder ausführbare Anwendung veröffentlicht. Der Einstiegspunkt von Anwendungen ist die Datei `main.rs`. Libraries haben keinen Einstiegspunkt, sondern stellen ihre öffentlich deklarierten Elemente wie Funktionen und Datentypen anderen Libraries oder Anwendungen zur Verfügung.

Device Attestation Prozess zur Validierung eines Commissionee als Matter-zertifiziertes Gerät.

Node Eine über eine eindeutige ID innerhalb einer Matter Fabric adressierbare Entität, die das Matter-Protokoll unterstützt.

Trait Vgl. Interface in Java, das Polymorphismus in Rust ermöglicht und das Verhalten vorschreibt, das von implementierenden Typen unterstützt werden muss. Structs können eine oder mehrere Traits implementieren, um eine gewünschte Funktionalität zu unterstützen [44].

Trait Bound Schreibt in Rust für einen Typ vor, welches Verhalten er unterstützen muss.

Erklärung zur selbständigen Bearbeitung

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich gemacht.

Ort	Datum	Unterschrift im Original