Diplomarbeit

Höhere Technische Bundeslehranstalt Leonding
Abteilung für Informatik

Mesh-Netzwerk an der HTL Leonding

Eingereicht von:	**Tim Untersberger, 5BHIF**
	Stefan Waldl, 5BHIF
Datum:	**3. April 2020**
Betreuer:	**Thomas Stütz, Gerald Köck**
Projektpartner:	**HTBLA Leonding**

Gendererklärung

Zur besseren Lesbarkeit werden auf dieser Website personenbezogene Bezeichnungen, die sich zugleich auf Frauen und Männer beziehen, generell nur in der im Deutschen üblichen männlichen Form angeführt, also z.B. „Teilnehmer“ statt „TeilnehmerInnen“ oder „Teilnehmerinnen und Teilnehmer“.

Dies soll jedoch keinesfalls eine Geschlechterdiskriminierung oder eine Verletzung des Gleichheitsgrundsatzes zum Ausdruck bringen.

Gender Declaration

For better readability, personal names that refer to women and men at the same time are generally only given in the masculine form common in German, e.g. „Participants“ instead of „Participants“ or „Participants“.

However, this is in no way intended to express gender discrimination or a violation of the principle of equality.

Eidesstattliche Erklärung

Hiermit erkläre ich an Eides statt, dass ich die vorgelegte Diplomarbeit selbstständig und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe. Gedanken, die aus fremden Quellen direkt oder indirekt übernommen wurden, sind als solche gekennzeichnet.

Die Arbeit wurde bisher in gleicher oder ähnlicher Weise keiner anderen Prüfungsbehörde vorgelegt und auch noch nicht veröffentlicht.

Leonding, am 3. April 2020

Tim Untersberger, Stefan Waldl

Declaration of Academic Honesty

Hereby, I declare that I have composed the presented paper independently on my own and without any other resources than the ones indicated. All thoughts taken directly or indirectly from external sources are properly denoted as such.

This paper has neither been previously submitted to another authority nor has it been published yet.

Leonding, April 3, 2020

Tim Untersberger, Stefan Waldl

Zusammenfassung

An einer Schule sind Klassenräume der wichtigste Ort der Leistunserbringung, die Bedingungen in diesen Klassenräumen sollten für den Lernerfolg optimal sein, in der Realität gibt es oft Mängel (zB sauerstoffarme Luft), die leicht beseitigt werden könnten. Es ist daher Vorteilhaft die Bedingungen in den einzelnen Klassenräumen erfassen zu können, um einerseits Masnahmen zu ergreifen (öffnen der Fenster), andererseits auch um auswertbare Informationen für den Heizbetrieb(Temperatur) zu erhalten.

Ein Mesh-Netzwerk soll es daher ermöglichen die im gesamten Gebäude der HTL-Leonding verteilten Sensoren und Aktoren zu vernetzen.

Die Verwendung eines Mesh-Netzwerkes ermöglicht die einfache einbindung von zusätzlichen Sensoren und Aktoren, außerdem wird das Schulnetzwerk nicht beansprucht.

Im Rahmen dieser Arbeit wurde zunächst die Mesh-Infrastruktur erstellt (hardwarmäßige Konfiguration und softwaremäßiges Einbinden von Nodes in das Mesh-Netzwerk), weiters wurde als fachliche Anwendung ein IoT-Netzwerk implementiert, welches die Kommunikation zwischen den einzelnen Komponenten mittels eines Message Brokers ermöglicht. Die Visualisierung, als dritte Komponente, kann den aktuellen Zustand des IoT-Netzwerkes darstellen. Dabei werden folgende Fragen beantwortet: Welche Komponenten (Nodes) sind online, wann wird eine Message übermittelt, usw.

Zusätzlich wurde für die ESP32-Nodes ein Over-The-Air - Update (OTA) implementiert.

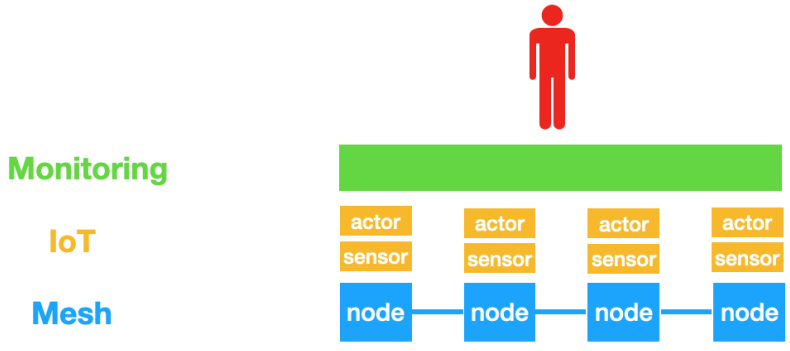

Abbildung 1: Struktur (Quelle: eigene Darstellung)

Abstract

At a school, classrooms are the most important place for the performance of pupils, the conditions in these classrooms should be optimal for learning success, in reality there are often shortcomings (e.g. low-oxygen air) that could easily be remedied. It is therefore advantageous to be able to record the conditions in the individual classrooms in order to take measures (open the window) on the one hand, and to obtain evaluable information for the heating mode (temperature) on the other.

A mesh network should therefore make it possible to connect the sensors and actuators distributed throughout the HTL-Leonding building.

The use of a mesh network enables simple integration of additional sensors and actuators, and the school network is not used.

As part of this work, the mesh infrastructure was initially created (hardware configuration and software integration of nodes into the mesh network), and an IoT network was implemented as an application, which enables communication between the individual components using a message broker. The visualization, as a third component, can represent the current state of the IoT network and answer following questions : Which components (nodes) are online, which messages (e.g. temperature values) are transmitted, etc.

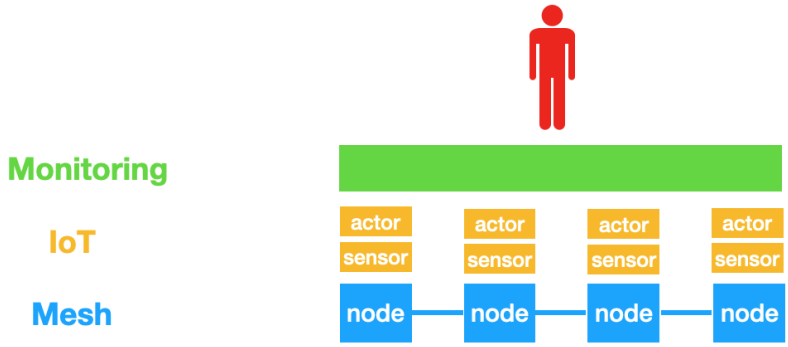

Figure 2: Structure (source: own source)

Autoren der Diplomarbeit

Tim Untersberger

Aufgabenbereich

OTA und Mesh Netzwerk

Name:	Tim Untersberger
Geburtsdatum:	15. April 2001
E-Mail:	timuntersberger2@gmail.com
Bildungsweg:	
2007 bis 2011	Volksschule Doppl
2011 bis 2015	NMS Hart
seit 2015	HTL Leonding, Informatik
Berufliche Erfahrung:	
Sommer 2016	Colour & Point, Softwareentwickler
Sommer 2018	Runtastic, Softwareentwickler
Sprachliche Kenntnisse:	
Deutsch	Muttersprache
Englisch	Fließend

Tabelle 1: Tim Untersberger

Stefan Waldl

Aufgabenbereich

OTA und Mesh Netzwerk

Name:	Stefan Waldl
Geburtsdatum:	26. März 2000
E-Mail:	waldl.stefan@gmail.com
Bildungsweg:	
2006 bis 2010	Volksschule 28
2010 bis 2015	BRG Ramsauerstraße
seit 2015	HTL Leonding, Informatik
Berufliche Erfahrung:	
Sommer 2017	CBCX-Betting Technologies, Softwareentwickler
Sommer 2018	CBCX-Betting Technologies, Softwareentwickler
Sprachliche Kenntnisse:	
Deutsch	Muttersprache
Englisch	Fließend

Tabelle 2: Stefan Waldl

Danksagung

An dieser Stelle möchten wir uns bei all denjenigen bedanken, die uns während der Planung und Durchführung dieser Diplomarbeit unterstützt und motiviert haben.

Ganz besonderen Dank an Herrn Prof. Thomas Stütz und Herrn Prof. Gerald Köck, welche uns bereits sehr früh unterstützt und unser Augenmerk auf die richtigen Technologien gelenkt haben, sowie uns auch betreut und unsere Arbeit begutachtet haben.

Bedanken wollen wir uns auch bei all unseren KorrekturleserInnen der Dokumente, welche uns für eine fehlerfreie und ordentliche Einreichung beiseite standen.

Inhaltsverzeichnis

1 Ausgangssituation und Zielsetzung **6**
- 1.1 Ausgangssituation . . . 6
- 1.2 Beschreibung des Problembereichs . . . 6
- 1.3 Aufgabenstellung . . . 7
- 1.4 Zielbestimmung . . . 8

2 Verwendete Technologien **9**
- 2.1 Message Queuing Telemetry Transport (MQTT) . . . 9
  - 2.1.1 QoS . . . 10
- 2.2 Docker . . . 11
  - 2.2.1 Docker vs. Virtuelle Maschinen . . . 12
  - 2.2.2 Begriffe . . . 13
- 2.3 Docker Compose . . . 13
  - 2.3.1 Docker Compose Beispiel . . . 14
- 2.4 ESP-IDF Toolchain . . . 16
  - 2.4.1 KConfig.projbuild . . . 17
  - 2.4.2 Kompilieren . . . 18
  - 2.4.3 Flashen . . . 18
- 2.5 Platform IO . . . 19
- 2.6 Utility lib Köck . . . 20

3 Ausgewählte Aspekte **22**
- 3.1 Verwendete Hardware . . . 22
  - 3.1.1 DHT22 . . . 22
  - 3.1.2 NodeMCU ESP32 . . . 24
- 3.2 Over The Air Update (OTA) . . . 26
  - 3.2.1 Partition Table . . . 28
  - 3.2.2 OTA Partition Table . . . 33
- 3.3 ESP-MESH . . . 34
  - 3.3.1 Wi-Fi vs Mesh . . . 34
- 3.4 Mesh Visualizer . . . 40
  - 3.4.1 Cytoscape . . . 41

3.5 Libraries . . . 45
3.5.1 ESP-IDF Libraries . . . 45
3.5.2 Eigene Libraries . . . 47
3.6 Erläuterung der Verwendung des Mesh-Netzwerks . . . 52
3.6.1 Übersicht . . . 52
3.6.2 Nodejs Setup . . . 53
3.6.3 ESP-IDF Setup . . . 54
3.6.4 Source Code . . . 55
3.6.5 Mosquitto . . . 56
3.6.6 Mesh Visualizer . . . 57
3.6.7 Partition Table . . . 60
3.6.8 Config . . . 60
3.6.9 Hardware . . . 64
3.6.10 Flashen . . . 65
3.6.11 Ergebnis . . . 65
3.6.12 Code . . . 66
3.7 ELF vs. Bin . . . 72
3.7.1 Bin . . . 72
3.7.2 ELF . . . 73
3.7.3 ELF-Header . . . 74
3.7.4 File Data . . . 76
3.7.5 Static vs. Dynamic binaries . . . 78
3.7.6 Fazit . . . 78

4 Resümee **79**
4.1 Tim Untersberger . . . 79
4.2 Stefan Waldl . . . 79

A Arbeitsaufteilung **98**

Kapitel 1

Ausgangssituation und Zielsetzung

1.1 Ausgangssituation

Die HTBLA-Leonding ist eine BHS im Raum Oberösterreich, welche für ihre gute Ausbildung und interessanten Projekte im Bereich der Informationstechnologie bekannt ist.

1.2 Beschreibung des Problembereichs

In den Klassenräumen fehlt es oft an Sauerstoff, manchmal ist auch die Temperatur nicht adäquat. Dieseh kann eine Reihe von unerwünschten Symptomen hervorrufen, wie zum Beispiel:

- Kopfschmerzen
- Ermüdung
- Schwindel
- Übelkeit
- mehr Probleme mit Asthma und Allergien

Die oben gennannten Symptome sind in einer Umgebung wie der Schule unbedingt zu vermeiden, da es ansonsten zu einem Konzentrationsverlust der Schüler führt.

Um zusätzlich eine Regelung der Heizung vornehmen zu können sind weitere Informationen der einzelnen Klassenräume, wie die Temperatur, notwendig. Weiters wird es rund um die Uhr ermöglicht, geöffnete Fenster zu lokalisieren und mit dieser Information Schäden an der Schule zu verhindern.

1.3 Aufgabenstellung

Um Sensoren in der Schule platzieren zu können, ist es erforderlich, zuerst eine Mesh-Infrastruktur für die Auswertung der Sensoren zu gestalten.

Eine Einbindung eines MQTT-Netzwerkes, welches direkt auf dem Mesh-Netzwerk aufsitzt ist ebenso notwendig. Dieses Mesh-Netzwerk ermöglicht den verschiedenen Knoten, welche mit den Sensoren und Aktoren verbunden sind, untereinander zu komunizieren und Informationen auszutauschen.

Um den Überblick über solch ein Netzwerk nicht zu verlieren, ist es ebenso wünschenswert eine Visualisierungs-Komponente zu gestallten, welche es ermöglicht den aktuellen Zustandes des Mesh-Netzwerkes zu inspizieren und den Benutzer nachvollziehen lässt wann Nachrichten versendet werden.

Zusätzlich dazu soll ein System entwickelt werden, welches es ESPs ermöglicht OTA Updates durchzuführen.

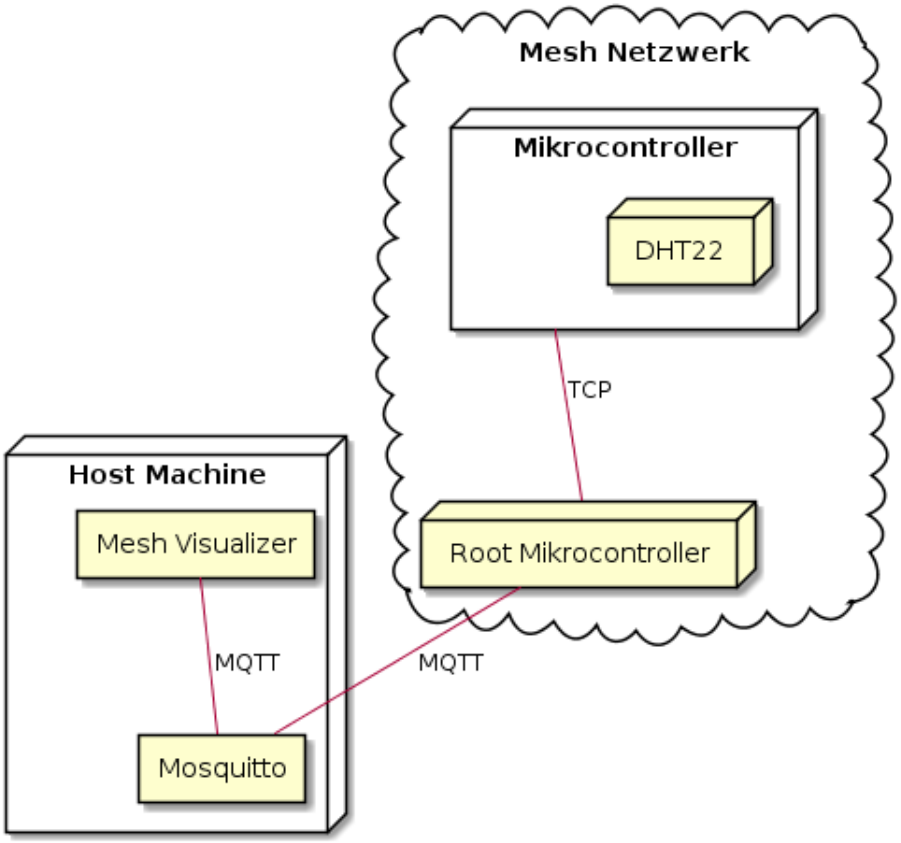

Abbildung 1.1: Systemarchitektur (Quelle: Eigene Darstellung)

1.4 Zielbestimmung

Das Ziel dieses System ist es die Schule, als Lernumgebung zu verbessern und den Schülern eine stress- und sorgenfreies Lernen zu ermöglichen.

Kapitel 2

Verwendete Technologien

2.1 Message Queuing Telemetry Transport (MQTT)

Im Laufe der Arbeit werden folgende Begriffe erwähnt, welche hier erklärt werden:

1. **Broker**:

   Ein Server, der alle Nachrichten von den Clients empfängt und diese dann an die entsprechenden Zielclients weiterleitet.

2. **Topic**:

   In MQTT verweist das Wort Topic auf eine UTF-8-Zeichenfolge, die der Broker zum Filtern von Nachrichten für jeden verbundenen Client verwendet. Das Topic besteht aus einer oder mehreren Topiclevels. Jedes Topiclevel ist durch einen Slash getrennt.

3. **Message**:

   Eine MQTT Message ist ein beliebiges Array von Bytes, welches an ein bestimmtes Topic gesendet wird.

   Wenn im Kapitel 3.5.2 von einer MQTT Message geschrieben wird, dann steht dies für die Struktur, welche in der selben Library definiert wird.

4. **Quality of Service (QoS)**:

   Das Quality of Service (QoS) Level ist eine Vereinbarung zwischen dem Absender und dem Empfänger einer Nachricht, die die Zustellgarantie für eine bestimmte Nachricht definiert.

2.1.1 QoS

In der nachstehenden Aufzählung werden die drei verschiedenen Arten von QoS näher erläutert, welche von der HiveMq Website[20] zusammengefasst wurden.

1. **QoS 0 - höchstens einmal**: Die kleinste QoS-Stufe ist 0. Dieser Service-Level garantiert eine bestmögliche Lieferung. Es gibt keine Liefergarantie. Der Empfänger bestätigt den Empfang der Nachricht nicht und die Nachricht wird vom Absender nicht gespeichert und erneut gesendet. QoS-Stufe 0 wird oft als „fire and forget“ bezeichnet und bietet die gleiche Garantie wie das zugrunde liegende TCP-Protokoll.

Abbildung 2.1: MQTT QoS 0 [20]

2. **QoS 1 - mindestens einmal**:

   QoS Level 1 garantiert, dass eine Nachricht mindestens einmal an den Empfänger übermittelt wird. Der Absender speichert die Nachricht, bis er vom Empfänger ein e Rückmeldung erhält, die den Empfang der Nachricht bestätigt. Es ist möglich, dass eine Nachricht mehrmals gesendet oder zugestellt wird.

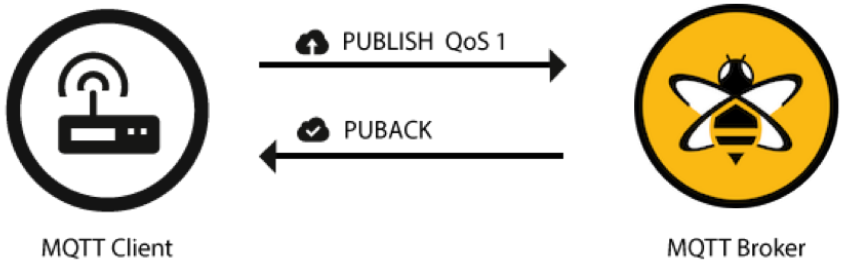

Abbildung 2.2: MQTT QoS 1 [20]

3. **QoS 2 - genau einmal**:

   QoS 2 ist das höchste Servicelevel in MQTT. Diese Stufe garantiert, dass jede Nachricht nur einmal von den vorgesehenen Empfängern empfangen wird. QoS 2 ist die sicherste und langsamste Servicequalität. Die Garantie wird durch mindestens zwei request/response flows zwischen dem Sender und dem Empfänger bereitgestellt.

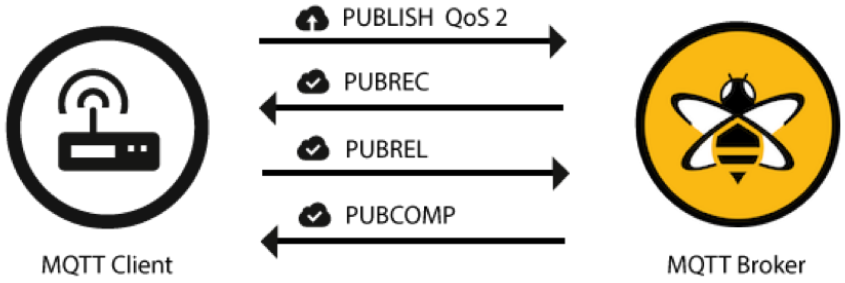

Abbildung 2.3: MQTT QoS 2 [20]

2.2 Docker

Docker ist ein Open-Source-Projekt zur Automatisierung der Bereitstellung von Anwendungen als tragbare Container, die überall ausgeführt werden können.

2.2.1 Docker vs. Virtuelle Maschinen

Virtuelle Maschinen

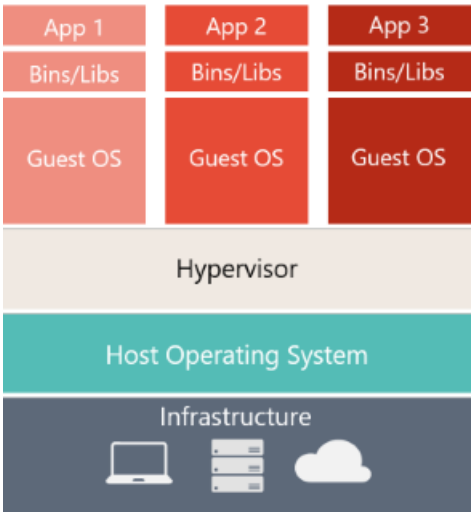

Abbildung 2.4: Docker Virtuelle Maschine (Quelle: eigene Darstellung)

Zu den virtuellen Maschinen gehören die Anwendung, die erforderlichen Bibliotheken oder Binärdateien sowie ein vollständiges Gastbetriebssystem. Die vollständige Virtualisierung erfordert mehr Ressourcen als die Containerisierung. [12]

Docker

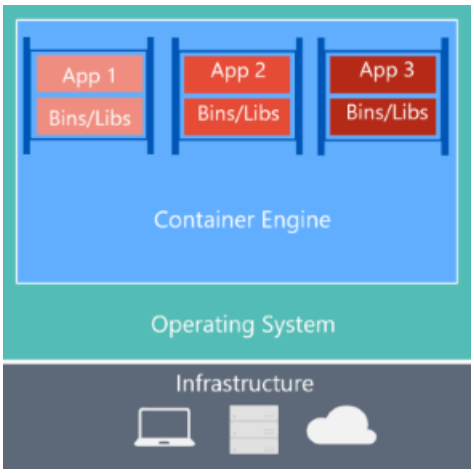

Abbildung 2.5: Docker (Quelle: eigene Darstellung)

Container enthalten die Anwendung und alle ihre Abhängigkeiten. Sie teilen den Betriebssystemkern jedoch mit anderen Containern und werden als isolierte Prozesse im Benutzerbereich des Host-Betriebssystems ausgeführt (außer in Hyper-V-Containern, in denen jeder Container innerhalb einer speziellen virtuellen Maschine pro Container ausgeführt wird). [12]

2.2.2 Begriffe

Im Laufe der Arbeit werden folgende Begriffe erwähnt:

1. **Container**:

   Ein Container ist eine Instanz eines Images. Er repräsentiert die Ausführung einer einzelnen Anwendung, eines Prozesses oder eines Dienstes.

2. **Image**:

   Ein Image ist ein Paket mit allen Abhängigkeiten und Informationen, die zum Erstellen eines Containers erforderlich sind. Ein Image enthält alle Abhängigkeiten (z. B. Frameworks) sowie die Bereitstellungs- und Ausführungskonfiguration, die von einer Container-Laufzeit verwendet werden soll.

3. **Volume**:

   Ein Volume ist ein beschreibbares Dateisystem, das der Container verwenden kann. Da Images schreibgeschützt sind, fügen Volumes eine beschreibbare Ebene über dem Image hinzu, sodass die Programme Zugriff auf ein beschreibbares Dateisystem haben.

4. **Network**:

   In einem Docker Network können Container auf sich gegenseitig mittels ihres Namens referenziern. Dies erleichtert das miteinander Arbeiten von verschiedenen Containern.

2.3 Docker Compose

Docker-compose ist ein Tool zum Definieren und Ausführen von Docker-Anwendungen mit mehreren Containern. Mit Compose wird eine YAML-Datei verwendet, um die Dienste von Anwendung zu konfigurieren. Anschließend werden alle Dienste mit einem einzigen Befehl erstellt und gestartet. [11]

2.3.1 Docker Compose Beispiel

```yaml
version: "3.5"
services:
  postgres:
    image: postgres:11.5
    ports:
      - 5432:5432
    volumes:
      - ./pgdata:/var/lib/postgresql/data
    networks:
      - postgres
    environment:
      POSTGRES_USER: user
      POSTGRES_PASSWORD: password
      POSTGRES_DB: db
  pgadmin:
    image: dpage/pgadmin4
    ports:
      - 5433:80
    environment:
      PGADMIN_DEFAULT_EMAIL: email
      PGADMIN_DEFAULT_PASSWORD: pw
    networks:
      - postgres
networks:
  postgres:
    driver: bridge
```

Abbildung 2.6: Docker Compose Beispiel (Quelle: eigene Darstellung)

In der obigen Abbildung kann man das docker-compose.yml file von dem OTA Server sehen.

Die Struktur eines docker-compose files ist im Normalfall immer gleich. Am Anfang definiert man, welche Version des Standards man verwendet und anschließend die verschiedenen Dienste. Der OTA Server benötigt nur 2 Dienste:

1. Postgres
2. PgAdmin4

In Abbildung 2.6 wird ein Netzwerk explizit erstellt. Es ist auch möglich, das Netwerk implizit zu erstellen, indem man den networks Bereich von ganz unten entfernt.

Postgres

PostgreSQL ist eine relationale Open Source Datenbank. Sie wird für die Speicherung von Informationen über die verschiedenen Firmwares, welche gerade auf dem Server hochgeladen sind, verwendet.

Die Datenbank benötigt irgendeinen Weg, um mit der Außenwelt kommunizieren zu können, deswegen wird der interne Port 5432 auf den externen Port 5432 geleitet.

Damit die Absicherung der Datenbank leicht verläuft, wird ein Volume für die Datenbank erstellt, worin sich die Daten der Datenbank befinden. In dieser docker-compose Datei wird das Volume implizit erstellt.

Man kann es auch explizit erstellen wie folgt:

```
volumes:
    pgdata:
```

Mithilfe von Umgebungsvariablen kann man die Standardwerte des Images überschreiben. Die Datenbank hat in der obigen Abbildung folgende Werte überschrieben:

- **POSTGRES_USER** auf *user*
- **POSTGRES_PASSWORD** auf *password*
- **POSTGRES_DB** auf *db*

PgAdmin4

PgAdmin4 ist eine web-basierte Benutzeroberfläche für Postgres.

Sie ist die PhpMyAdmin Oberfläche in der postgres Welt.

Die Software hört automatisch auf dem Port 80 zu, was für Entwicklungszwecke sehr unpraktisch ist, da Port 80 auf Ubuntu Adminrechte benötigt. Deswegen wird der interne Port 80 auf den externen Port 5433 umgeleitet.

Mit folgenden Umgebungsvariablen werden die default Logindaten für PgAdmin4 definiert:

- PGADMIN_DEFAULT_EMAIL
- PGADMIN_DEFAULT_PASSWORD

2.4 ESP-IDF Toolchain

Das Espressif IoT-Development-Framework (ESP-IDF) ist das offizielle Framework für die ESP32 Chips. [14]

Für die Entwicklung von Programmen für den ESP32 wurde eine eigene Toolchain namens ESP-IDF entwickelt.

Die Verwendung dieser Toolchain funktioniert nur in einem ESP-IDF konformen Projekt.

Ein Hello-World Template für ein ESP-IDF Projekt befindet sich im offiziellen Repository von ESP-IDF [15].

Hier ist es möglich, mit folgendem Befehl den ESP zu konfigurieren:

```
idf.py menuconfig
```

Dieser Befehl startet folgendes Konfigurationsprogramm:

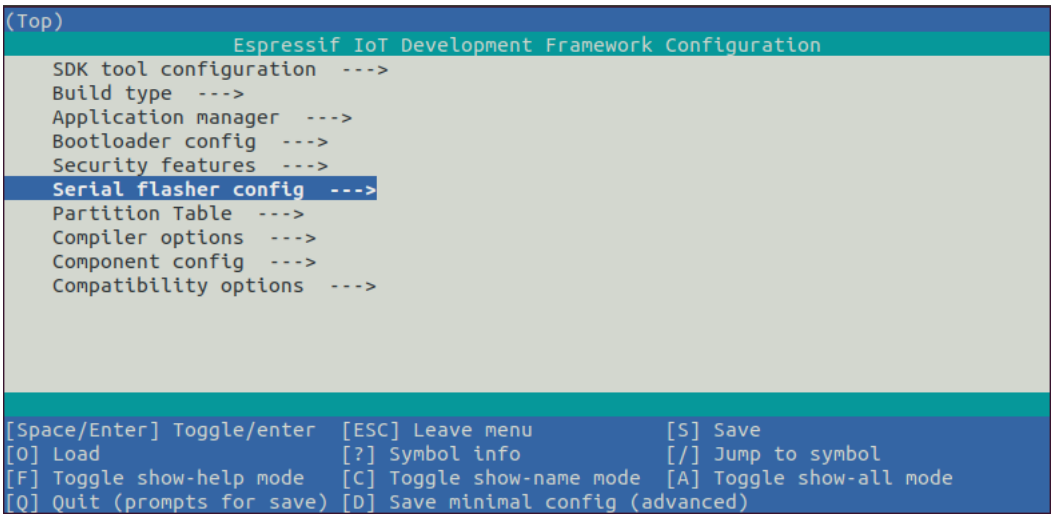

Abbildung 2.7: menuconfig [22]

Dieses Konfigurationsprogramm wird verwendet, um projektspezifische Variablen zu setzen, wie zB

- Netzwerkname
- Netzwerkpasswort
- Takt des Prozessors
- Parition Table Variante
- ... und viele mehr

2.4.1 KConfig.projbuild

Sind die von Espressif zur Verfügung gestellten Menüs nicht genug, so ist es möglich, eine **KConfig.projbuild** Datei anzulegen. Mit dieser Datei ist es möglich, neue Top-Level Menüeinträge und Untermenüeinträge zu generieren. Dies wird benötigt, da in der standarmäßigen **menuconfig** nur der Großteil der gebrauchten Konfigurations-Felder zur Verfügung gestellt werden.

Nachstehender Ausschnitt einer **KConfig.projbuild** Datei bietet einen Überblick über die Struktur dieser Datei.

KConfig.projbuild

```
menu "Example Configuration"

choice
    bool "Mesh AP Authentication Mode"
    default WIFI_AUTH_WPA2_PSK
    help
        Authentication mode.

    config WIFI_AUTH_OPEN
        bool "WIFI_AUTH_OPEN"
    config WIFI_AUTH_WPA_PSK
        bool "WIFI_AUTH_WPA_PSK"
    config WIFI_AUTH_WPA2_PSK
        bool "WIFI_AUTH_WPA2_PSK"
    config WIFI_AUTH_WPA_WPA2_PSK
        bool "WIFI_AUTH_WPA_WPA2_PSK"
endchoice

config MQTT_BROKER_URL
    string "MQTT broker url"
    default "mqtt://192.168.0.2"
    help
        Mqtt broker url.
endmenu
```

Der Name einer zusätzlichen Konfigurationsoption wird mithilfe des *name* Schlüsselwortes definiert, dieses nimmt eine beliebige Zeichenkette entgegen.

Diese Zeichenkette stellt den Namen der Option dar.

In einer Menuconfig kann zwischen folgenden Eingabetypen entschieden werden:

- **choice**

  Dieser Typ erwartet einen oder mehrere **config** Typen, welche die möglichen Optionen dieser Auswahl definieren.

- **config**

  Der **config** Typ ist ein textbasiertes Eingabefeld.

Der Defaultwert eines Eingabefeldes kann mittels des *default* Schlüsselwortes geändert werden.

Das Schlüsselwort *help* dient zur Vergabe einer Beschreibung für ein Eingabefeld.

2.4.2 Kompilieren

Damit die fertigen Espressif Beispiele auf einem Dual-Core ESP funktionieren, muss sich dieser im Single-Core Modus befinden.

Ist das Projekt richtig konfiguriert, kann es mit folgendem Befehl gebuildet werden:

```
idf.py build
```

Mit diesem Befehl wird die Anwendung kompiliert sowie der Bootloader und der Partition Table erstellt.

Sollten beim Kompilieren Fehler auftreten, so werden diese in der Konsole ausgegeben.

2.4.3 Flashen

Verläuft das Kompilieren fehlerfrei, kann das Programm mit diesem Befehl auf den ESP geflasht werden:

```
idf.py flash
```

Sollte es ein Problem beim Flashen geben, liegt dies häufig an einem defekten USB-Kabel.

IDF Monitor

Der IDF-Monitor ist ein serielles Terminalprogramm, das serielle Daten zum und vom seriellen Anschluss des Zielgeräts weiterleitet.

Ist das Programm bereits auf dem ESP, kann es mit folgendem Befehl auf dem IDF Monitor überprüft werden.

```
idf.py monitor
```

Mit dem nachstehenden Befehl kann das Programm gebuildet, geflasht und upgeloadet werden. Dies erspart den Aufwand, jeden Befehl einzeln einzugeben.

```
idf.py flash monitor
```

2.5 Platform IO

PlatformIO ist ein fortschrittliches und äußerst vielseitiges Ökosystem für die IoT-Entwicklung, das eine IDE, ein Build-System, einen Unified Debugger und einen Bibliotheksmanager umfasst. Es bietet Unterstützung für mehr als 550 Entwicklungsboards, mehr als 25 Entwicklungsplattformen und mehr als 10 nützliche Frameworks. Die PlatformIO IDE ist ein plattformübergreifendes Dienstprogramm für die schnelle berufliche Weiterentwicklung mit integriertem C / C++ - Intelligent Code Completion, Smart Code Linter und erweitertem Serial Port Monitor. PlatformIO kann auch in die gängigen IDEs und kontinuierlichen Integrationssysteme integriert werden, um die Zeit für die Bereitstellung von IoT-Anwendungen zu verkürzen.[25]

PlatformIO ist in reinem Python geschrieben und hängt nicht von zusätzlichen Bibliotheken / Tools eines Betriebssystems ab. So kann man mit PlatformIO auf Windows, Macintosh und Linux arbeiten.

2.6 Utility lib Köck

Dies ist Professor Köcks Library Sammlung, welche er persönlich verfasste. Sämtliche Technologien, welche durch die Libraries untersützt werden, sind in folgender Abbildung zu sehen.

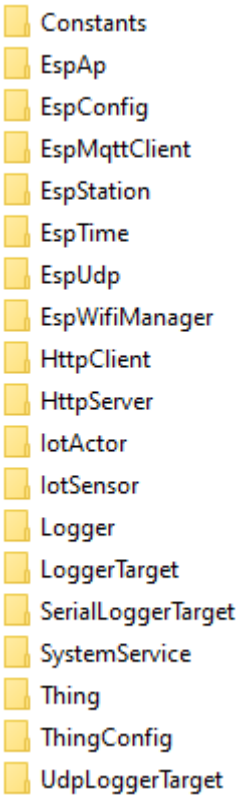

Abbildung 2.8: Technologien Utility lib (Quelle: eigene Darstellung)

Dazu sind folgende Sensor Libraries ebenfalls enthalten:

Abbildung 2.9: Sensoren und Aktoren Utility lib (Quelle: eigene Darstellung)

Im Laufe der Arbeit erfolgte ein Umstieg auf die offiziellen Espressif Libraries, da Professor Köcks Library Sammlung aufgrund fehlender Funktionalitäten für diese Diplomarbeit nicht mehr geeignet war.

Da die meisten Espressif Libraries Funktionalitäten aus der Programmiersprache C verwenden, welche nicht in C++ unterstützt werden und die Libraries von Professor Köck in C++ implementiert sind, ist es selten möglich diese gleichzeitig zu verwenden ohne, dass die Espressif Libraries angepasst werden müssen.

Um Kompatibilität herzustellen, ist es bei den meisten Anwendung notwendig, die Espressif Libraries anzupassen.

Kapitel 3

Ausgewählte Aspekte

3.1 Verwendete Hardware

3.1.1 DHT22

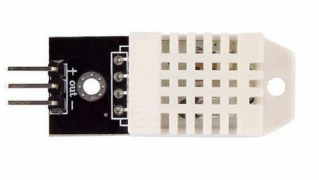

Abbildung 3.1: Bild DHT22 [9]

Der DHT22, oder auch AM2302, ist ein digitaler Temperatur und Feuchtigkeissensor, welcher ein digitales Signal der Sensordaten über den Daten-Pin ausgibt. Jedes Exemplar dieses Sensor Modells wurde in einer Kalibrierkammer kalibriert.

Dieser Kalibrierkoeffizient wird auf dem, am Mikrocontroller eingebauten, One-Time-Programmable - Speicher (OTP Memory 3.1.1) gespeichert.

Die technischen Daten des Sensors sind:

Model	DHT22
Power supply	3.3-6V DC
Output signal	digital signal via single-bus
Sensing element	Polymer capacitor
Operating range	humidity 0-100%RH; temperature -40~80Celsius
Accuracy	humidity +-2%RH(Max +-5%RH); temperature <+-0.5Celsius
Resolution or sensitivity	humidity 0.1%RH; temperature 0.1Celsius
Repeatability	humidity +-1%RH; temperature +-0.2Celsius
Humidity hysteresis	+-0.3%RH
Long-term Stability	+-0.5%RH/year
Sensing period	Average: 2s
Interchangeability	fully interchangeable
Dimensions	small size 14*18*5.5mm; big size 22*28*5mm

Abbildung 3.2: technische Daten Dht22 [10]

Zu den Vorteilen des DHT22 zählen:

- Kleiner Formfaktor (15,1mm x 25,1mm x 7,7mm)

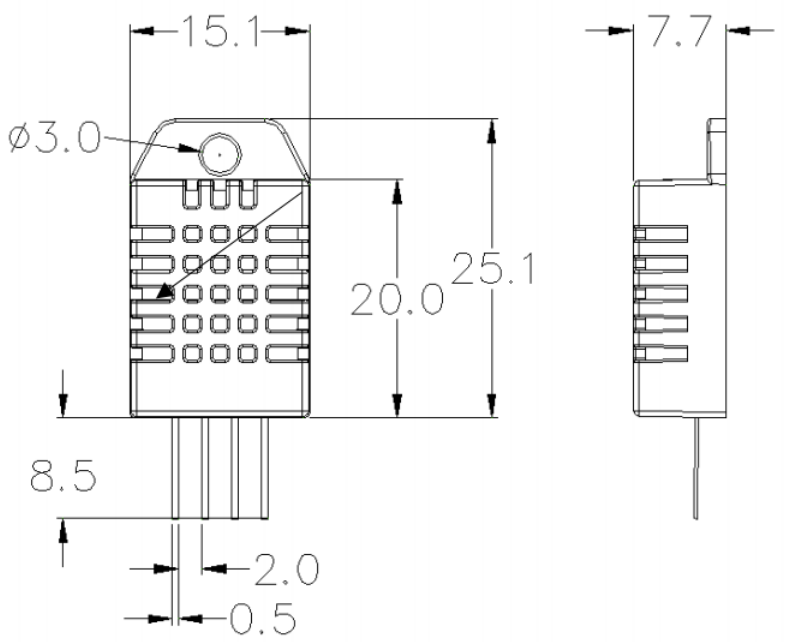

Abbildung 3.3: Formfaktor Dht22 [10]

- Niedriger Stromverbrauch (3,3-6 Volt)
- Lange Übertragungsdistanz (20 Meter)

Der Nachteil des Sensors ist:

- Wartezeit zwischen neuen Sensordaten (mindestens 2 Sekunden)

OTP Memory

OTP Speicher ist eine spezielle Form von nicht flüchtigem Speicher, der es genau einmal erlaubt, auf den Speicher zu schreiben. Wenn der Speicher einmal programmiert ist, behält er seine Informationen für immer, auch über Stromverlust.

Beispiele dafür sind:

- Boot code
- Encryption keys
- Konfigurationsparameter für analoge Sensoren

3.1.2 NodeMCU ESP32

Abbildung 3.4: Bild NodeMCU ESP32 [24]

Das NodeMCU ESP32 ist ein prototyping Board. Die wichtigste Komponente dieses ESPs ist das ESP-WROOM-32 3.1.2, es stellt das Gehirn dieses Mikrocontrollers dar.

ESP-WROOM-32

Das ESP-WROOM-32 ist ein generisches MCU Modul mit integriertem Wi-Fi, Bluetooth und Bluetooth low energy. So ermöglicht einem dieses Modul, sich beispielsweise mit Wlan-Routern oder einem Handy zu verbinden. Weiters ist es durch den niedrigen Schlafstrom von 5 µA gut für den Bateriebetrieb geeignet.

Categories	Items	Specifications
Certification	RF certification	FCC/CE-RED/IC/TELEC/KCC/SRRC/NCC
	Wi-Fi certification	Wi-Fi Alliance
	Bluetooth certification	BQB
	Green certification	RoHS/REACH
Test	Reliablity	HTOL/HTSL/uHAST/TCT/ESD
Wi-Fi	Protocols	802.11 b/g/n (802.11n up to 150 Mbps)
		A-MPDU and A-MSDU aggregation and 0.4 μs guard interval support
	Frequency range	2.4 GHz ~ 2.5 GHz
Bluetooth	Protocols	Bluetooth v4.2 BR/EDR and BLE specification
	Radio	NZIF receiver with –97 dBm sensitivity
		Class-1, class-2 and class-3 transmitter
		AFH
	Audio	CVSD and SBC
Hardware	Module interfaces	SD card, UART, SPI, SDIO, I^2C, LED PWM, Motor PWM, I^2S, IR, pulse counter, GPIO, capacitive touch sensor, ADC, DAC
	On-chip sensor	Hall sensor
	Integrated crystal	40 MHz crystal
	Integrated SPI flash	4 MB
	Operating voltage/Power supply	3.0 V ~ 3.6 V
	Operating current	Average: 80 mA
	Minimum current delivered by power supply	500 mA
	Recommended operating temperature range	–40 °C ~ +85 °C
	Package size	(18.00±0.10) mm × (25.50±0.10) mm × (3.10±0.10) mm
	Moisture sensitivity level (MSL)	Level 3

Abbildung 3.5: ESP32-WROOM-32 Spezifikationen[17]

3.2 Over The Air Update (OTA)

Problemstellung

Mikrocontroller befinden sich in einem laufendem System meist an ungünstigen Orten, an die man nur mit hohem Aufwand gelangt. Bis jetzt musste man den Computer physisch mit einem Kabel mit dem Mikrocontroller verbinden um neue Firmware auf den Kontroller zu spielen. OTA ermöglicht es, den Mikrocontroller über das Netzwerk mit neuer Firmware zu versorgen. Dabei muss der selektierte Microcomputer lediglich mit einem Wlan-Router verbunden sein und ein physisches Kabel wird überflüssig.

Wie OTA funktioniert

Bei dem OTA-Vorgang wird zuallererst die Config-Datei des jeweiligen Mikrocontrollers ausgelesen. In dieser Datei steht, welche Firmware der OTA Client herunterladen soll. Nun sendet der Mikrocontroller eine Anfrage mit dem in der Config-Datei eingetragen Namen der Firmware. Diese Anfrage dient dazu, den Timestamp an dem die Firmware das letzte Mal am Server geändert worden ist, herauszufinden.
Darauf antwortet der Server mit dem Timestamp.
Nun überprüft der Mikrocontroller den Timestamp der letzten Änderung des Servers mit dem Timestamp der letzten Änderung der Firmware, die gerade installiert ist.

Nun gibt es zwei Szenarien, die eintreten können:

- **Die Timestamps sind gleich**. Das bedeutet, dass der Server und der Mikrocontroller beide die gleiche Firmware besitzen und es besteht kein Grund diese vom Server herunterzuladen.

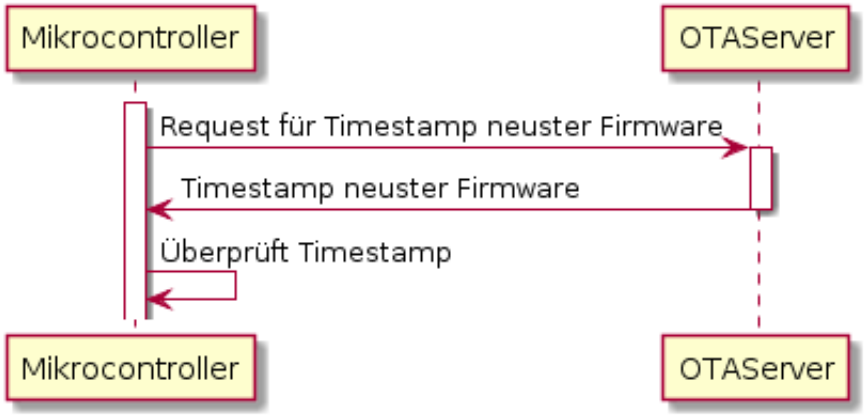

Abbildung 3.6: OTA Sequenz Timestamp gleich (Quelle: eigene Darstellung)

- **Die Timestamps sind unterschiedlich**. Das bedeutet, dass die Firmware am Server geändert wurde und der Mikrocontroller eine ältere Version besitzt.
  Da davon ausgegangen wird, dass der Server immer die letzte Version der Firmware besitzt, lädt der Mikrocontroller diese herunter und startet von der soeben heruntergeladenen Firmware neu.

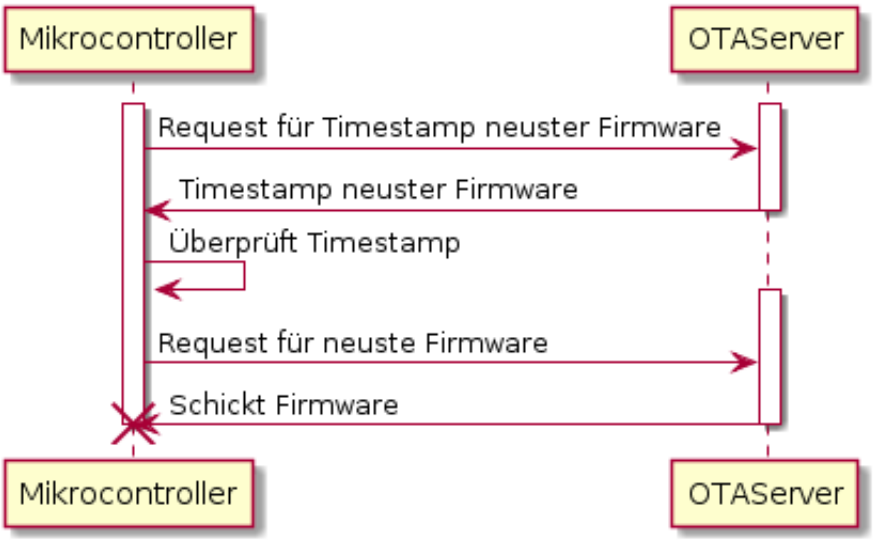

Abbildung 3.7: OTA Sequenz Timestamp anders (Quelle: eigene Darstellung)

Übersicht

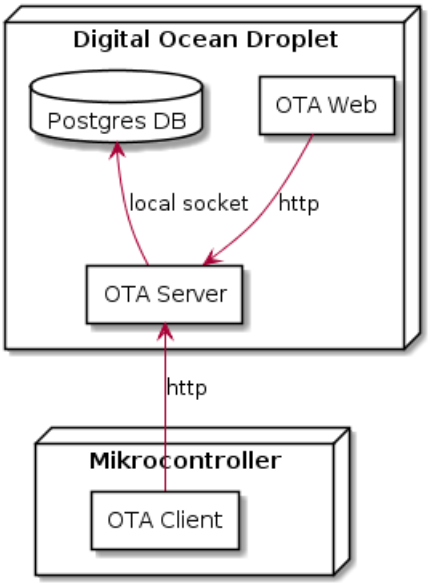

Abbildung 3.8: OTA Deployment (Quelle: eigene Darstellung)

Um OTA zu ermöglichen, ist auf einem *Digital Ocean Droplet* der OTA Server deployed.

Dieser stellt eine API zur Verwaltung von den existierenden Firmwares zur Verfügung.

Wenn eine Firmware geändert, entfernt oder eine neue hinzugefügt werden soll, geschieht dies über die OTA-Web Applikation.

Die OTA-Web Applikation stellt eine Benutzeroberfläche für die Verwaltung der Firmwares bereit.

Einem Mikrocontroller ist es nun ermöglicht, mithilfe des *OTA Servers* zu überprüfen, ob der Mikrocontroller die aktuelle Version seiner Firmware besitzt.

3.2.1 Partition Table

Die Partitionstabelle ist bei OTA mitunter eines der wichtigsten Themen. Konfiguriert man diese Tabelle nicht richtig, so würde das Programm nicht einmal starten. Um OTA zu ermöglichen, ist es notwendig, in der Partitionstabelle mindestens eine ausreichend große OTA-Partition zu vergeben.

Übersicht

Der Flash eines einzelnen ESP32 kann mehrere Apps sowie viele verschiedene Arten von Daten (Kalibrierungsdaten, Dateisysteme, Parameterspeicherung usw.) enthalten. Aus diesem Grund wird eine Partitionstabelle im Flash auf (Standardoffset) 0x8000 geflasht.

Die Länge der Partitionstabelle beträgt 0xC00 Byte (maximal 95 Partitionstabelleneinträge). Nach den Tabellendaten wird eine MD5-Prüfsumme angehängt, mit der die Integrität der Partitionstabelle überprüft wird. Wenn die Partitionstabelle aufgrund eines sicheren Starts signiert ist, wird die Signatur nach der Partitionstabelle angehängt.

Jeder Eintrag in der Partitionstabelle hat einen Namen (Bezeichnung), einen Typ (app, data, ota oder etwas anderes), einen SubType und den Offset im Flash, in dem die Partition geladen wird.

Custom Partition Tables

Bei komplexeren Anwendungen reicht die default Partitionstabelle nicht mehr aus und es muss eine angepasste Tabelle erstellt werden.

Wenn ausgewählt wird, dass eine benutzerdefinierte Partitionstabelle verwendet werden soll, muss manuell eine CSV-Datei erstellt werden, in welcher dann eine beliebige Anzahl von Partitionen mit folgenden Punkten eingetragen werden können:

- Name
- SubType
- Offset
- Size
- Flags

```
# Name,   Type, SubType,  Offset,   Size,  Flags
nvs,      data, nvs,      0x9000,  0x4000
otadata,  data, ota,      0xd000,  0x2000
phy_init, data, phy,      0xf000,  0x1000
factory,  app,  factory,  0x10000,  1M
ota_0,    app,  ota_0,    ,         1M
ota_1,    app,  ota_1,    ,         1M
nvs_key,  data, nvs_keys, ,        0x1000
```

Name Feld

Das Namensfeld kann beliebig gewählt werden. Es ist für den Mikrocontroller nicht von Bedeutung. Namen, die länger als 16 Zeichen sind, werden jedoch abgeschnitten.

Type Feld

Das Partitionstypfeld kann folgende Werte beinhalten:

1. App (0)
2. Data (1)
3. Eine beliebige Zahl zwischen 0 und 254

Die Typen 0x00-0x3F sind für esp-idf-Kernfunktionen reserviert.

Wenn eine Anwendung Daten speichern muss, muss ein benutzerdefinierter Partitionstyp im Bereich 0x40-0xFE hinzugefügt werden.

Der Bootloader ignoriert alle anderen Partitionstypen als App (0) und Data (1).

SubType

Das 8-Bit-SubType-Feld ist spezifisch für einen bestimmten Partitionstyp. Esp-idf gibt derzeit nur die Bedeutung des Subtypfelds für die Partitionstypen *App* und *Data* an (Stand 09.03.2020).

Wenn der Typ *App* ist, kann das Subtypfeld als

- *factory (0)*
- *ota_0 (0x10)* bis *ota_15 (0x1F)*
- *test (0x20)*

angegeben werden.

- *Factory (0)* ist die Standard-App-Partition. Der Bootloader führt die Factory-App aus, es sei denn, er sieht eine Partition vom Typ data / ota. In diesem Fall liest er diese Partition, um zu bestimmen, welches OTA-Image gestartet werden soll.
  - Weiters sei vermekt, dass OTA niemals die *factory* Partition aktualisiert.
  - Wenn die Flash-Nutzung in einem OTA-Projekt beibehalten werden soll, kann die Factory-Partition entfernt und stattdessen *ota_0* verwendet werden.
- *ota_0 (0x10)...ota_15 (0x1F)* sind die OTA-App-Slots. Sie verwenden dann die *ota_data* Partition, um zu konfigurieren, welchen App-Slot der Bootloader starten soll.
- Bei der Verwendung von OTA sollte eine Anwendung über mindestens zwei OTA-Applikationslots verfügen (*ota_0* und *ota_1*).

- *test (0x20)* ist ein reservierter Subtyp für werkseitige Testverfahren. Es wird als Fallback-Boot-Partition verwendet, wenn keine andere gültige App-Partition gefunden wird. Es ist auch möglich, den Bootloader so zu konfigurieren, dass er bei jedem Start einen general-purpose input/output(GPIO) Pin liest, und diese Partition zu starten, wenn der GPIO „low“ gehalten wird.
- Wenn der Typ *data* ist, kann das Subtypfeld mit folgenden Werten befüllt werden:
  - *ota (0)*
  - *phy (1)*
  - *nvs (2)*
  - *nvs_key (4)*
- *ota (0)* ist die OTA-Datenpartition, in der Informationen zur aktuell ausgewählten OTA-Anwendung gespeichert werden. Diese Partition sollte mindestens 0x2000 Bytes groß sein.
- *phy (1)* dient zum Speichern von PHY-Initialisierungsdaten. In der Standardkonfiguration wird die Phy-Partition nicht verwendet und die PHY-Initialisierungsdaten werden in der App selbst kompiliert. Daher wird diese Partition aus Platzgründen meist aus der Partitionstabelle entfernt.
- *nvs (2)* steht für die Non-Volatile Storage-API (**NVS**).
  - **NVS** wird zum Speichern von PHY-Kalibrierungsdaten pro Gerät verwendet (anders als Initialisierungsdaten).
  - **NVS** wird unter anderem zum Speichern von WiFi-Daten verwendet, wenn die Initialisierungsfunktion esp_wifi_set_storage (WIFI_STORAGE_FLASH) verwendet wird.
  - Die **NVS**-API kann auch für andere Anwendungsdaten verwendet werden.
  - Es wird dringend empfohlen, eine **NVS**-Partition von mindestens 0x3000 Byte aufzunehmen, da sonst einige unerwartete Fehler auftreten können.
  - Wenn die **NVS**-API zum Speichern vieler Daten verwendet wird, wird empfohlen, diese über den standardmäßig 0x6000 konfigurierten Bytes zu erhöhen.
- *nvs_keys (4)* ist die NVS-Key-Partition.
  - Sie wird zum Speichern von NVS encryption keys verwendet, wenn das NVS Encryption feature aktiviert ist.
  - Die Größe dieser Partition sollte 4096 Byte betragen (minimale Partitionsgröße).

Offset

Partitionen mit leeren Offsets beginnen automatisch nach der vorherigen Partition.

App-Partitionen müssen an Offsets sein, die auf 0x10000 (64 KB) ausgerichtet sind.

Wenn ein nicht ausgerichtetes Offset für eine App-Partition angegeben wird, wird ein Fehler zurückgegeben.

Größen und Offsets können als Dezimalzahlen, Hexadezimalzahlen mit dem Präfix 0x oder Größenmultiplikatoren K oder M (1024 und 1024 * 1024 Byte) angegeben werden.

Das Default-Offset für die erste Partition kann mithilfe der *CONFIG_PARTITION_TABLE_OFFSET* Konfigurationsvariable gesetzt werden. Diese wird nur respektiert, wenn die erste Partition kein Offset definiert.

Flags

Derzeit wird nur ein Flag unterstützt: encrypted. Wenn dieses Feld auf encrypted eingestellt ist, wird diese Partition verschlüsselt, wenn die Flash-Verschlüsselung aktiviert ist.

Partitionen vom App-Typ werden immer verschlüsselt unabhängig, ob die Flag gesetzt ist oder nicht.[18]

3.2.2 OTA Partition Table

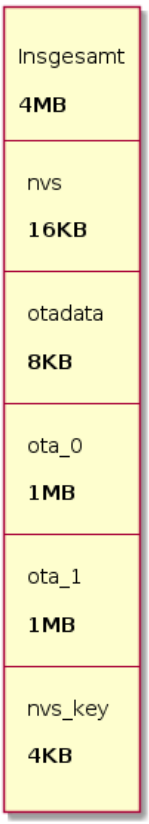

Abbildung 3.9: OTA Partition Table (Quelle: eigene Darstellung)

Der Partition Table des OTA Clients ist in Abbildung (3.9) zu sehen.

Insgesamt stehen allen Partitionen **4MB** zur Verfügung, da dies der gesamte Flash-Speicher des NodeMCU ESP32 (3.1.2) ist. Dieser enthält folgende Partitionen:

- Der **nvs**-Partition stehen **16KB** zur Verfügung, hier werden die notwendigen Config Daten gespeichert.
- Um OTA Updates durchführen zu können, ist auch eine **otadata** Partition notwendig, welcher in diesem Fall **8KB** zur Verfügung stehen. Hier wird gespeichert welche der beiden ota Partitionen gestartet werden muss.
- **ota_0** und **ota_1** sind mit jeweils **1MB** die beiden ota Partitionen. Hier wird die Firmware gespeichert, welche der Mikrocontroller dann ausführt.
- **nvs_key** erhält **4KB** Speicher. Da das NVS Filesystem aus Key-Value Paaren besteht, wird zusätzlich zur Value (**nvs**) auch ein Key gebraucht (**nvs_key**).

Auf eine **factory** Partition wurde verzichtet, um Speicher für die ota Partitionen zu sparen. Außerdem wird, wenn keine **factory** Partition existiert, die **ota** Partition mit dem niedrigsten Index gestartet. Hier ist das **ota_0**.

Die **phy_init** Partition wird ebenfalls weggelassen um Platz zu sparen. Weiters behandelt die **phy_init** Partition nur Ethernetdaten, welche in diesem Beispiel überflüssig sind.

3.3 ESP-MESH

Die folgenden Informationen sind eine Zusammenfassung von der offiziellen Espressif Website [16].

ESP-MESH ist ein Netzwerkprotokoll, das auf dem Wi-Fi-Protokoll basiert. Mit **ESP-MESH** können zahlreiche Geräte (im Folgenden als Knoten bezeichnet), die über einen großen physischen Bereich (sowohl drinnen als auch draußen) verteilt sind, unter einem einzigen WLAN (Wireless Local Area Network) miteinander verbunden werden. **ESP-MESH** ist selbstorganisierend und selbstheilend, was bedeutet, dass das Netzwerk autonom aufgebaut und gepflegt werden kann.

3.3.1 Wi-Fi vs Mesh

Wi-Fi

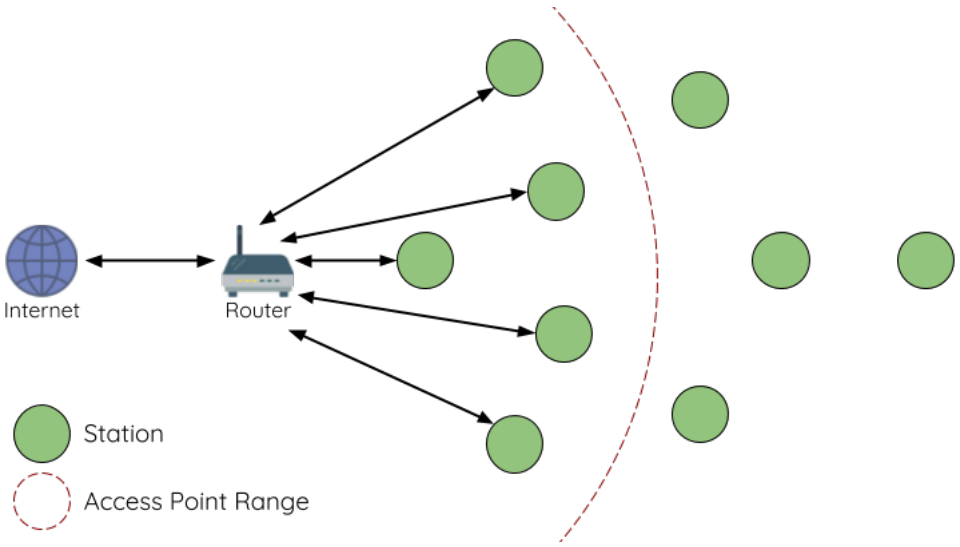

Abbildung 3.10: Wi-Fi Netzwerk Struktur [16]

Ein herkömmliches Wi-Fi-Netzwerk ist ein Punkt-zu-Mehrpunkt-Netzwerk, in dem ein einzelner zentraler Knoten, der als Access Point (AP) bezeichnet wird, direkt mit allen anderen Knoten verbunden ist, die als Stationen bezeichnet werden.

Der AP ist für die Schlichtung und Weiterleitung von Übertragungen zwischen den Stationen verantwortlich. Einige APs leiten auch Übertragungen von oder zu einem externen IP-Netzwerk über einen Router weiter.

Herkömmliche Wi-Fi-Netzwerke haben den Nachteil eines begrenzten Versorgungsbereichs, da jede Station in Reichweite sein muss, um eine direkte Verbindung mit dem AP herzustellen. Darüber hinaus sind herkömmliche Wi-Fi-Netzwerke anfällig für Überlastung, da die maximal zulässige Anzahl von Stationen im Netzwerk durch die Kapazität des AP begrenzt ist.

Mesh

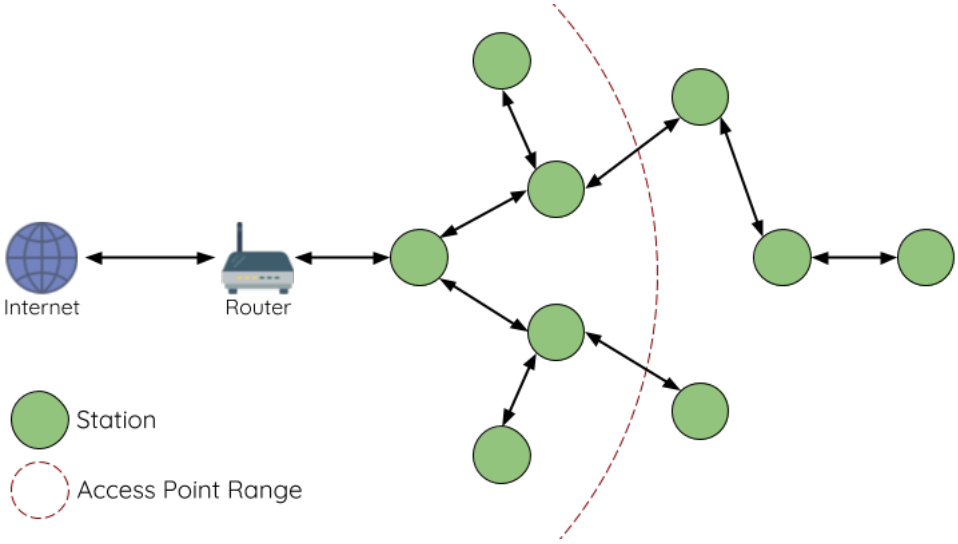

Abbildung 3.11: ESP-Netzwerk Struktur [16]

ESP-MESH unterscheidet sich von herkömmlichen Wi-Fi-Netzwerken darin, dass Knoten keine Verbindung zu einem zentralen Knoten herstellen müssen. Stattdessen dürfen Knoten eine Verbindung zu benachbarten Knoten herstellen.

Knoten sind gegenseitig für die Weiterleitung der Übertragungen verantwortlich. Dies ermöglicht es einem **ESP-MESH**-Netzwerk, einen viel größeren Versorgungsbereich zu haben, da Knoten immer noch mit verbunden sein können, ohne sich in Reichweite des zentralen Knotens befinden zu müssen. Ebenso ist **ESP-MESH** weniger anfällig für Überlastung, da die Anzahl der im Netzwerk zulässigen Knoten nicht mehr durch einen einzelnen zentralen Knoten begrenzt ist.

Um mit einem **ESP-MESH** Netzwerk umgehen zu können, ist es notwendig folgende Begriffe zu verstehen:

- **Node (Knoten)**

  Jedes Gerät, das Teil eines ESP-MESH-Netzwerks ist.

- **Root Node**

  Der **Root Node** ist der oberste Knoten eines Netzwerk.

- **Child Node**

  Ein **Child Node** ist ein Knoten, der sich auf der hierarchischen Ebene mindestens eine Stufe **unter** einem anderen Knoten befindet.

- **Parent Node**

  Ein **Parent Node** ist ein Knoten, der sich auf der hierarchischen Ebene mindestens eine Stufe **über** einem anderen Knoten befindet.

- **Intermediate Parent Node**

  Ein **Intermediate Parent Node** ist ein **Parent Node**, der gleichzeitig kein **Root Node** ist.

- **Descendant Node**

  Ein **Descendant Node** ist jeder Knoten, der durch wiederholtes Weitergehen von Eltern zu Kind erreichbar ist.

- **Sibling Node**

  Ein **Sibling Node** ist ein Knoten der sich mit einem anderen Knoten den selben **Parent Node** teilt.

- **Leaf Node**

  Ein **Leaf Node** ist ein **Node**, welchem es nicht erlaubt ist, **Child Nodes** zu besitzen.

- **Idle Node**

  Ein **Idle Node** ist ein Knoten, welcher sich noch nicht mit dem Netzwerk verbunden hat. Er wird jedoch versuchen, eine **Connection** mit einem **Parent Node** herzustellen.

- **Connection**

  Unter einer **Connection** versteht man eine herkömmliche Wi-Fi Verbindung.

- **Upstream Connection**

  Darunter wird eine **Connection** von einem **Child Node** zu einem **Parent Node** verstanden.

- **Downstream Connection**

  Darunter wird eine **Connection** von einem **Parent Node** zu einem **Child Node** verstanden. Es ist das Gegenteil zu einer **Upstream Connection**.

- **Wireless Hop**

  Dies ist der Teil des Pfades zwischen Ursprungs- und Zielknoten, der einer einzelnen drahtlosen Verbindung entspricht. Ein Datenpaket, das eine einzelne Verbindung durchläuft, wird als *Single-Hop* bezeichnet, während das Durchlaufen mehrerer Verbindungen als *Multi-Hop* bezeichnet wird.

- **Subnetwork**

  Ein **Subnetwork** ist die Unterteilung eines **ESP-MESH-Netzwerks**, das aus einem Knoten und allen seinen untergeordneten Knoten besteht. Daher besteht das Subnetz des Stammknotens aus allen Knoten in einem **ESP-MESH-Netzwerks**.

- **media access control(MAC) Address**

  Eine **MAC Addresse** ist eine Adresse, welche eine Netzwerkkarte eindeutig bestimmen kann.

In folgender Abbildung ist ein **ESP-MESH** Netzwerk, mit gekennzeichneten Node Typen dargestellt.

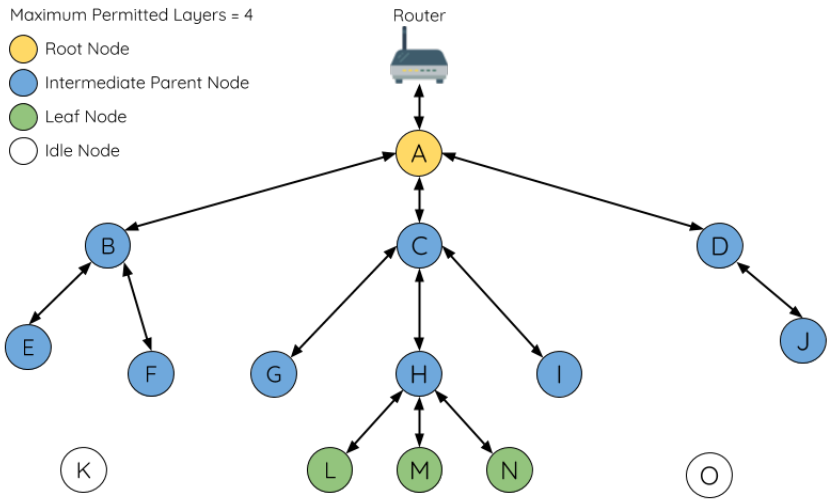

Abbildung 3.12: Node Typen [16]

Connection eines Idle Nodes

Wenn ein **Idle Node** nur einen möglichen **Parent Node** in Reichweite erkennt, so verbindet er sich zu diesem. Tritt der Fall ein, dass mehrere Möglichkeiten für einen **Parent Node** existieren, so hängt die Wahl des **Parent Nodes** von folgenden Faktoren ab:

- auf der wievielten Ebene sitzt der **Parent Node**
- die Anzahl der **Child Nodes** eines **Parent Nodes**

Der **Idle Node** bevorzugt jedes Mal den **Parent Node** auf der niedrigsten Ebene, so wird die Anzahl der verschiedenen Ebenen im **ESP-MESH** Netzwerk möglichst niedrig gehalten.

Wenn sich mehrere mögliche **Parent Nodes** auf der gleichen Ebene befinden, verbindet sich der **Idle Node** zu dem mit den wenigsten **Child Nodes**.

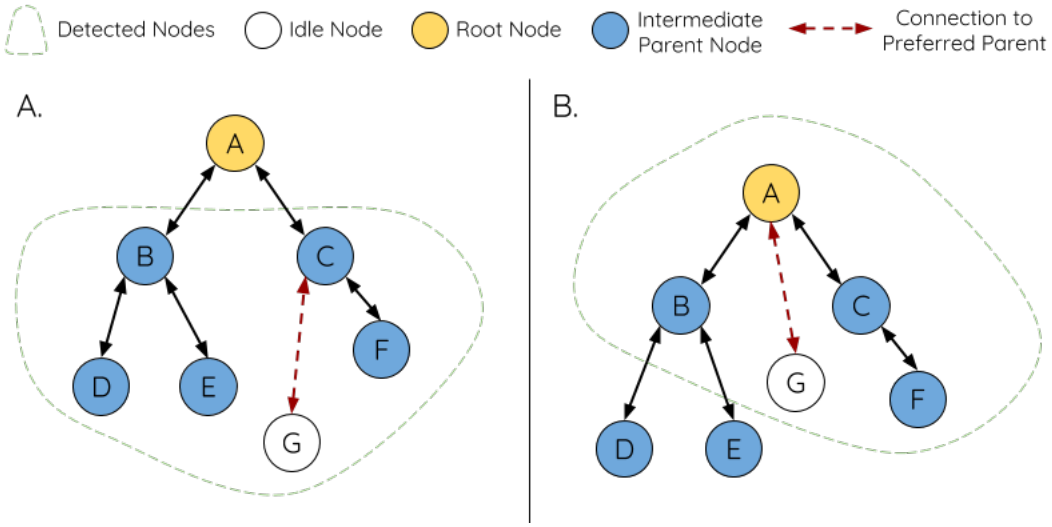

Abbildung 3.13: Bevorzuget Parent Node [16]

Node Versagen

Wenn ein Knoten versagt, können zwei Situationen entstehen:

- Der Knoten ist ein **Root Node**:

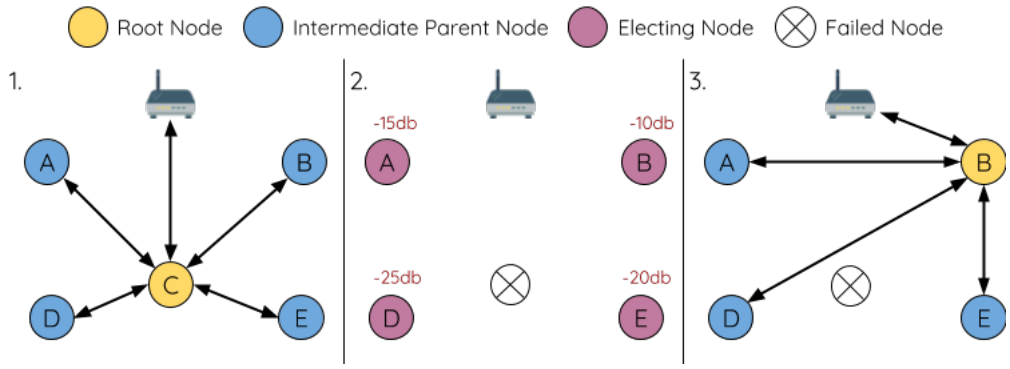

Abbildung 3.14: Root Node Versagen[16]

Wenn der **Root Node** versagt, wird dies von seinen **Child Nodes** sofort erkannt. Die **Child Nodes** werden zuerst mehrmals versuchen, sich erneut mit dem ehemaligen **Root Node** zu verbinden, wenn dies fehlschlägt, wird es zwischen den **Child Nodes** eine neue **Root Noode** Wahl geben. Mittels dem Received Signal Strength Indicator (RSSI) ermittelt nun jeder **Child Node** die Signalstärke des Routers und der **Child Node** mit der stärksten Verbindung wird der neue **Root Node**, zu dem sich die **Child Nodes** verbinden.

- Der Knoten ist ein **Intermediate Parent Node**:

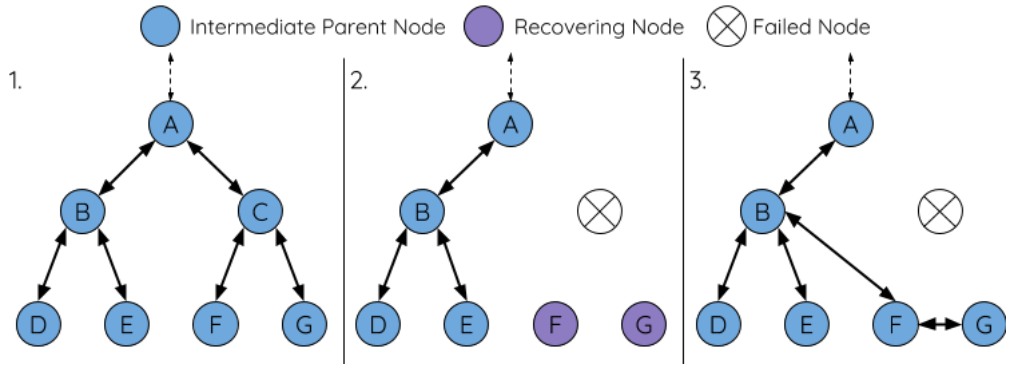

Abbildung 3.15: Intermediate Parent Node Versagen[16]

Wenn ein **Intermediate Parent Node** versagt, versuchen dessen **Child Nodes** sich mehrmals neu zu verbinden. Nach mehrfachen fehlgeschlagen Versuchen werden die **Child Nodes** beginnen, nach neuen potentiellen **Parent Nodes** zu suchen. Dies geschieht nach dem gleichen Prinzip wie beim **Root Node Versagen**.

Jeder **Child Node** wird selbständig einen neuen **Parent Node** aussuchen. Findet ein Knoten keinen neuen **Parent Node**, so wird dieser Knoten ein **Idle Node**.

3.4 Mesh Visualizer

Die in der nachstehenden Abbildung dargestellte Website dient zur Visualisierung eines Mesh-Netzwerks.

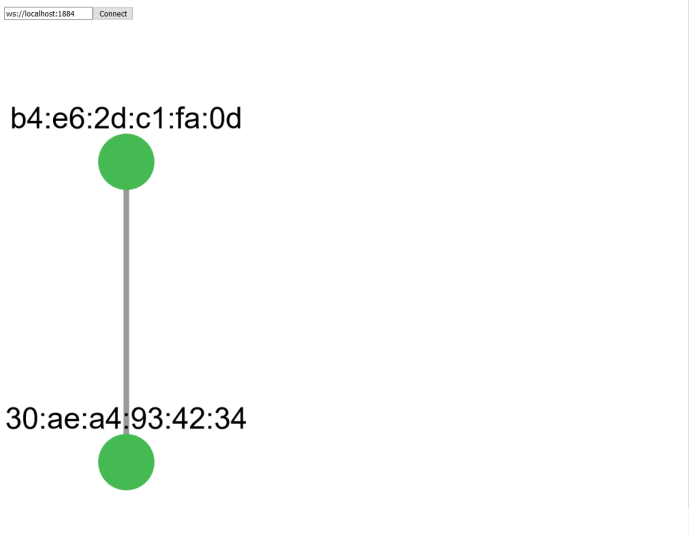

Abbildung 3.16: Mesh Visualizer (Quelle: eigene Darstellung)

Sie verwendet die Graphen Library *cytoscapejs* für die Darstellung der Knoten. Weitere Informationen sind dem Kapitel 3.4.1 zu entnehmen.

Die Benutzung in der Praxis ist im Kapitel 3.6.6 näher beschrieben.

3.4.1 Cytoscape

Cytoscape ist eine in Javascript geschriebene Open-Source-Library für Graphentheorie. Mit Cytoscape ist es möglich, auf einfacher Art und Weise umfangreiche interaktive Diagramme anzuzeigen und bearbeiten. [6]

Cytoscape wird in dieser Arbeit zur Visualisiserung des Mesh-Netzwerks verwendet.

Die Distribution dieser Library verläuft über die Website von *Node Packager Manager (NPM)* [23], welcher der zentrale Ort für alle veröffentlichten Javascript packages ist.

Für die Visualisierung des Mesh-Netzwerks existieren mehrere Möglichkeiten.

Für die Entwicklung der Website kamen folgende Algorithmen in Frage:

- Dagre (Kapitel 3.4.1)
- Euler (Kapitel 3.4.1)
- Spread (Kapitel 3.4.1)

Letztendlich wurde auf den Dagre Algorithmus zurückgegriffen.

Weitere Informationen daüber wie jeder Algorithmus funktioniert und warum *Dagre* gewählt wurde, sind in den jeweiligen Kapiteln zu finden.

Dagre

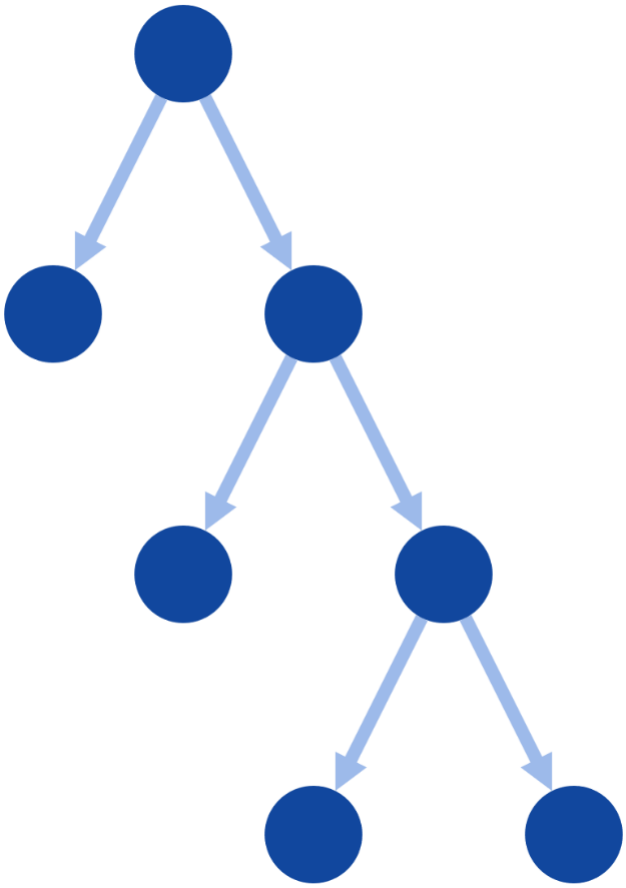

Abbildung 3.17: Cytoscape Dagre Beispiel [4]

Dagre ist ein Algorithmus zur Ordnung von Knoten in einem Diagramm. Das resultierende Diagramm ist ein baumförmiges Netzwerk. Ein Beispiel zu diesem Diagramm ist in Abbildung 3.17 dargestellt.

Der Mesh Visualizer benutzt diesen Algorithmus, da er der einzige baumartige Algorithmus der Kandidaten ist. Ein Mesh-Netzwerk ist baumförmig aufgebaut, deswegen ist es auch wichtig, dass die Knoten in einer baumförmigen Art dargestellt werden.

Euler

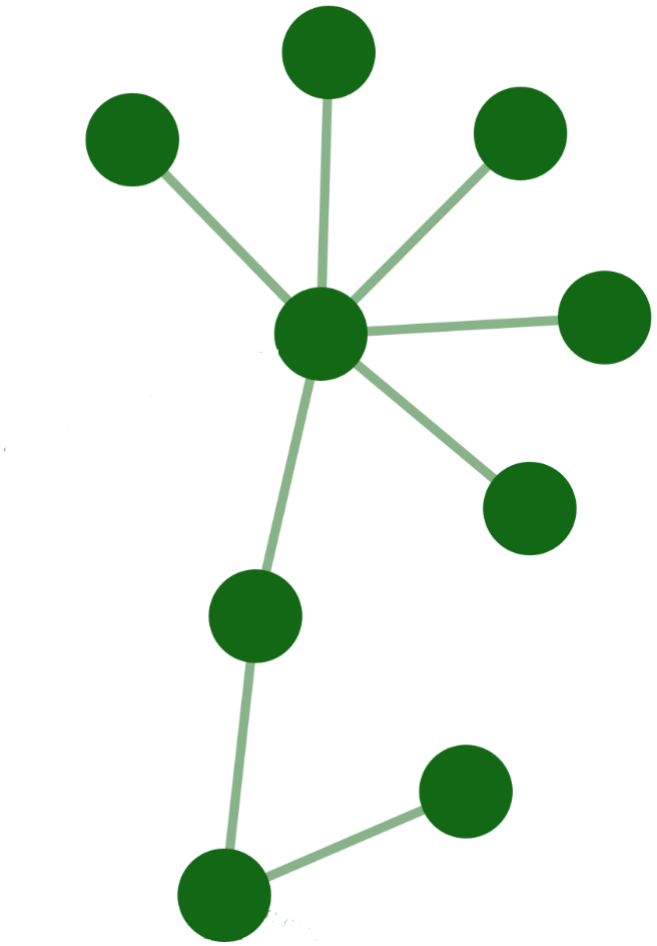

Abbildung 3.18: Cytoscape Euler Beispiel [5]

Euler ist ein Algorithmus, welcher ein physikbasiertes Diagramm erstellt. Dieser Algorithmus war leider nicht für die Visualisierung des Mesh-Netzwerks geeignet, da der Status quo einen Memory Leak hatte und ein Warten auf die Behebung dieses Problems nicht realistisch war.

Spread

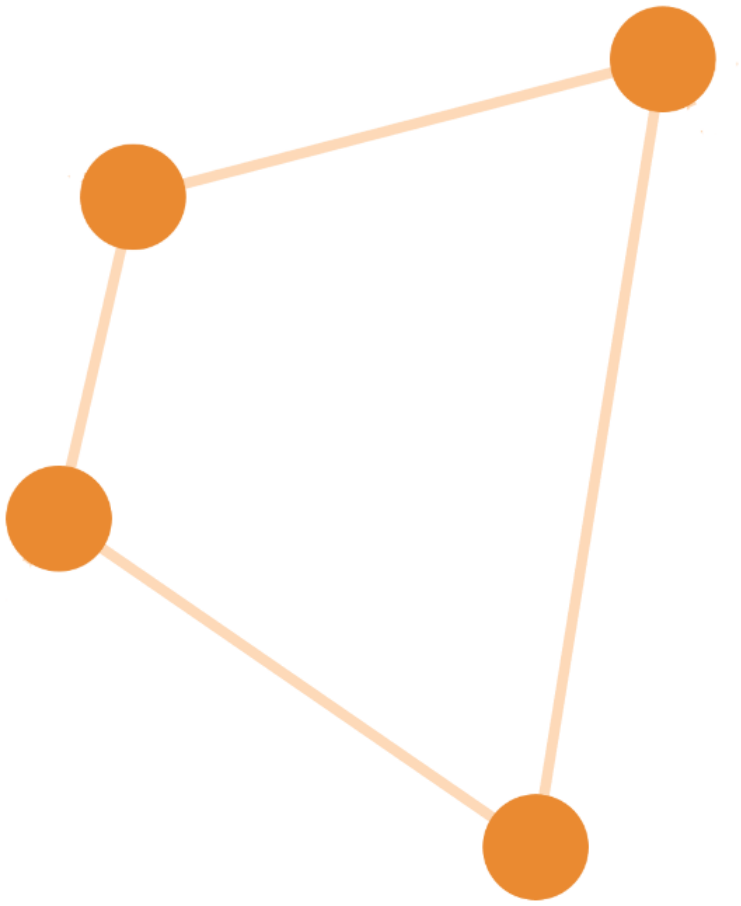

Abbildung 3.19: Cytoscape Spread Beispiel [7]

Der Spread Algorithmus ist wie Euler (Kapitel 3.4.1) physikbasiert. Dazu kommt auch noch, dass dieser versucht, die Knoten so weit wie möglich auseinander zu halten. Letztendlich wurde aus visuellen Gründen dieser Kandidat nicht gewählt.

3.5 Libraries

3.5.1 ESP-IDF Libraries

freertos

Diese Library ist notwendig, um auf einem ESP Code parallel laufen lassen zu können.

Mit folgender Funktion wird ein Task angelegt:

```
static inline IRAM_ATTR BaseType_t xTaskCreate(
        TaskFunction_t pvTaskCode,
        const char * const pcName,
        const uint32_t usStackDepth,
        void * const pvParameters,
        UBaseType_t uxPriority,
        TaskHandle_t * const pvCreatedTask)
```

Wie zu sehen ist, werden der Funktion folgende Parameter übergeben:

- **pvTaskCode**

  Hier wird die Addresse der Funktion eingetragen, welche im Task laufen soll.

- **pcName**

  Dieser Parameter ist der beschreibende Name des Tasks. Er wird verwendet, um debugging zu erleichtern.

- **usStackDepth**

  Diese Variable definiert die Größe des Task-Stacks. Die Zahl, die hier übergeben wird, entspricht der Anzahl der Variablen, die der Task zugleich halten kann, nicht der Anzahl der Bytes.

- **pvParameters**

  Dies ist der Pointer einer Variablen, die an den Task übergeben wird.

- **uxPriority**

  uxPriority stellt die Priorität des Tasks dar. Systeme mit MPU-Unterstützung können optional Tasks in einem privilegierten (System-) Modus erstellen, indem das Bit *portPRIVILEGE_BIT* des Prioritätsparameters gesetzt wird.

- **pvCreatedTask**

  pvCreatedTask wird verwendet, um ein *handle* zurückzugeben, mit dem auf den erstellten Task verwiesen werden kann, um den Task beispielsweise zu löschen.

[28]

Dht22

In der **Dht22** Library [19] sind insgesamt sechs öffentliche Funktionen definiert, wie in der folgenden Abbildung zu sehen ist:

```
void dht22_set_gpio(int gpio);
void dht22_handle_error(int response);
int dht22_read();
float dht22_get_humidity();
float dht22_get_temperature();
```

Abbildung 3.20: DHT22 Funktionen (Quelle: eigene Darstellung)

- **void dht22_set_gpio(int gpio)**:

  Dieser Funktion wird der Index des ESP Pins, mit dem data Pin verbunden ist mitgegeben. Dieser Index wird anschließend intern vermerkt, damit der ESP weiß auf welchem Pin er die Sensordaten des DHT22 erhalten wird.

  Diese Funktion sollte aufgerufen werden, bevor mit dem DHT22 versucht wird, Daten zu messen, da dies sonst fehlschlagen wird.

- **int dht22_read()**:

  Um diese Funktion aufzurufen, ist kein Parameter notwendig.

  Sie dient dazu, die Luftfeuchtigkeit und Tempereatur zu messen und schreibt diese in interne Variablen, auf die Benutzer keinen Zugriff haben.

  Der Rückgabewert der **dht22_read** Funktion ist ein Fehlercode. Dieser Fehlercode gibt Auskunft über den Verlauf der Funktion.

- **void dht22_handle_error(int response)**:

  Als Parameter nimmt die **dht22_handle_error** Funktion Fehlercodes entgegen.

  Das Ziel dieser Funktion ist es, den Fehlercode zu umschreiben und auf die Console zu loggen.

- **float dht22_get_humidity()**:

  Diese Funktion gibt den Wert der internen Luftfeuchtigkeits-Variablen, in welche **dht22_read()** die Luftfeuchtigkeits-Daten des Sensors schreibt, in Gleitkommadarstellung zurück.

- **float dht22_get_temperature()**:

  Diese Funktion gibt den Wert der internen Temperatur Variablen, in welche **dht22_read()** die Temperatur-Daten des Sensors schreibt, in Gleitkommadarstellung zurück.

3.5.2 Eigene Libraries

Im Laufe dieser Arbeit sind mehrere wiederverwendbare Libraries entstanden.

Das Application Programming Interface (API) dieser Libraries und wie man diese benutzt wird in diesem Kapitel erklärt.

MQTT

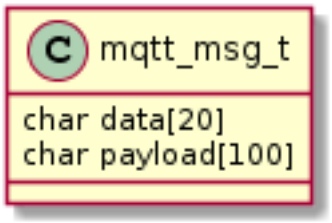

Abbildung 3.21: Mqtt Message Klassendiagramm (Quelle: eigene Darstellung)

Die MQTT Library definiert einen eigenen Struct namens *mqtt_msg_t*, welcher in Abbildung 3.21 als Klassendiagramm zu sehen ist.

Dieser besitzt die folgenden Felder:

- **topic**, ein 20 Zeichen (1 Byte) langer Buffer.
- **payload**, ein 100 Zeichen langer Buffer.

```
#define MQTT_MESSAGE_TOPIC_SIZE (20)
#define MQTT_MESSAGE_PAYLOAD_SIZE (100)

typedef struct {
    char topic[MQTT_MESSAGE_TOPIC_SIZE];
    char payload[MQTT_MESSAGE_PAYLOAD_SIZE];
} mqtt_msg_t;
```

Abbildung 3.22: Mqtt Message (Quelle: eigene Darstellung)

In der obigen Abbildung (3.22) ist die selbe Struktur im Source Code dargestellt.

Hier ist zu sehen, dass die Längen der jeweiligen Buffers mithilfe einer *preprocessor directive* definiert wurden.

Eine *preprocessor directive* ist in der Programmiersprache C ein Weg, wie man Makros definieren kann.

Makrodefinitionen sind keine Variablen und können nicht wie Variablen von ihrem Programmcode geändert werden. Im Allgemeinen wird diese Syntax verwendet, wenn Konstanten erstellt werden sollen, die Zahlen, Zeichenfolgen oder Ausdrücke darstellen.

Diese Library stellt insgesamt 2 öffentliche Funktionen zur Verfügung, welche in der nachstehenden Abbildung (3.23) zu sehen sind.

```
void mqtt_publish_msg(mqtt_msg_t *msg);
void mqtt_start();
```

Abbildung 3.23: Mqtt Funktionen (Quelle: eigene Darstellung)

- **void mqtt_publish_msg(mqtt_msg_t *msg)**:

  Als einziges Argument wird die Addresse einer MQTT Message erwartet.

  Diese wird anschließend synchron an den definierten MQTT Broker gesendet.

- **void mqtt_start()**:

  Nach dem Aufrufen dieser Funktion wird versucht, eine Verbindung mit dem definierten MQTT Broker herzustellen.

Hypertext Transfer Protocol (HTTP)

```
esp_err_t http_request(esp_http_client_method_t method, const char* url);
char* http_get_response_body();
int http_get_response_body_size();
```

Abbildung 3.24: HTTP Funktionen(Quelle: eigene Darstellung)

Die HTTP Library definiert und implementiert drei verschiedene Funktionen, welche in Abbildung 3.24 zu sehen sind.

- **esp_err_t http_request(esp_http_client_method_t method, const char* url)**:

  Diese Funktion nimmt folgende zwei Argumente entgegen:

  – method

    Dieses Argument erwartet einen in Abbildung 3.25 dargestellten Enum-Wert.

```
/**
 * @brief HTTP method
 */
typedef enum {
    HTTP_METHOD_GET = 0,    /*!< HTTP GET Method */
    HTTP_METHOD_POST,       /*!< HTTP POST Method */
    HTTP_METHOD_PUT,        /*!< HTTP PUT Method */
    HTTP_METHOD_PATCH,      /*!< HTTP PATCH Method */
    HTTP_METHOD_DELETE,     /*!< HTTP DELETE Method */
    HTTP_METHOD_HEAD,       /*!< HTTP HEAD Method */
    HTTP_METHOD_NOTIFY,     /*!< HTTP NOTIFY Method */
    HTTP_METHOD_SUBSCRIBE,  /*!< HTTP SUBSCRIBE Method */
    HTTP_METHOD_UNSUBSCRIBE,/*!< HTTP UNSUBSCRIBE Method */
    HTTP_METHOD_OPTIONS,    /*!< HTTP OPTIONS Method */
    HTTP_METHOD_MAX,
} esp_http_client_method_t;
```

Abbildung 3.25: HTTP Client Methoden[21]

Die HTTP Library unterstützt ausschließlich die *HTTP_METHOD_GET* Methode, da dies den minimalen Anforderungen des Mesh-Netzwerks entspricht.

– url

Durch dieses Argument wird definiert an welche Adresse dieser HTTP Request verschickt wird.

Die URL kann auch Query Parameter beinhalten.

In der folgenden URL ist ein Query Parameter mit einem Schlüssel namens *hello* und einem Wert von *world* definiert.

`https://www.github.com?hello=world`

Nach dem Aufrufen dieser Funktion wird ein synchroner HTTP Request verschickt und anschließend ein Fehlercode zurückgegeben.

- **char* http_get_response_body()**:

  Diese Funktion gibt die Addresse des Body des letzten HTTP Request zurück.

- **int http_get_response_body_size()**:

  Diese Funktion gibt die Länge des Body des letzten HTTP Request zurück.

Mesh

Die Mesh Library wurde um das *mesh_cmd_t* Konstrukt, welches in Abbildung 3.26 dargestellt wird, gebaut. Es existiert eine globale Warteschlange von Commands, welche nach dem First-In-First-Out (FIFO) Prinizip versendet werden. Die Warteschlange wird nach dem Starten des Mesh-Netzwerkes periodisch entleert und jeder Command in der Warteschlange versendet.

```
typedef struct {
    mesh_cmd_type_t type;
    bool is_broadcasted;
    bool send_to_self;
    uint8_t payload[MESH_CMD_PAYLOAD_SIZE];
    uint8_t from[6];
} mesh_cmd_t;
```

Abbildung 3.26: Mesh Command Type (Quelle: eigene Darstellung)

Ein *mesh_cmd* besteht hauptsächlich aus dem *type* und einem *payload*. Dazu werden noch verschiedene Metadaten über diesen Command gespeichert.

Es werden folgende Metadaten gespeichert:

- wenn **is_broadcasted** gesetzt ist, dann verteilt ein Knoten den neuen Command an alle Unter-Knoten.
- ist **send_to_self** gesetzt, wird der versendete Command auch an sich selber verschickt.
- das **from** Feld gibt an, welcher Knoten diesen Command konstruiert ist.

Es werden die Funktion, welche in folgender Abbildung (3.27) zu sehen sind, zur Verfügung gestellt.

```
typedef void (*mesh_cmd_cb)(mesh_cmd_t *);
typedef void (*mesh_connected_cb)(bool);

void mesh_create_network_interface();
void mesh_queue_cmd(mesh_cmd_t *cmd);
void mesh_on_cmd(mesh_cmd_cb);
void mesh_on_connected(mesh_connected_cb);
void mesh_set_cmd_payload(mesh_cmd_t *cmd, uint8_t *payload);
void mesh_init();
void mesh_start();
```

Abbildung 3.27: Mesh Funktionen (Quelle: eigene Darstellung)

In der obigen Abbildung sind verschiedene Funktionstypen dargestellt, welche das jeweilige Interface eines Callbacks beschreiben.

Es werden folgende Typen definiert:

- **void mesh_cmd_cb(mesh_cmd_t* cmd)**:

  Dieser Callback wird aufgerufen, wenn der Knoten, auf dem dieses Programm läuft, einen neuen Command erhält. Ihm wird die Address dieses Commands übergeben.

- **void mesh_connected_cb(bool is_root)**:

  Wenn sich der Knoten, auf welchem dieses Programm läuft, mit dem Mesh-Netzwerk verbindet, wird diese Funktion aufgerufen. Sie bekommt als Argument mit, ob dieser Knoten der Root Knoten ist.

- **void mesh_create_network_interface()**:

  Diese Funktion erstellt die Netzwerkschnittstelle des Mesh-Netzwerks, welche für die Verbindung mit dem Wi-Fi benötigt wird. Daher ist es wichtig, dass diese Funktion aufgerufen wird bevor das Wi-Fi initialisiert wird.

- **void mesh_queue_cmd(mesh_cmd_t *cmd)**:

  Die *mesh_queue_cmd* Funktion nimmt die Adresse eines Mesh Commands als einiziges Argument an, welches an die Warteschlange angehängt wird.

- **void mesh_on_cmd(mesh_cmd_cb)**:

  Mithilfe dieser Funktion kann die Funktion geändert werden, welche aufgerufen wird, wenn dieser Knoten einen neuen Command erhält.

  Sie bekommt eine Funktion vom Typen *mesh_cmd_cb*, welche am Anfang dieses Kapitels beschrieben wurde, als Argument.

- **void mesh_on_connected(mesh_connected_cb)**:

  Diese Funktion setzt die globale Referenz auf eine Funktion, welche aufgerufen wird, wenn dieser Knoten sich mit einen Mesh-Netzwerk verbindet.

  Sie bekommt eine Funktion vom Typen *mesh_connected_cb*, welche am Anfang dieses Kapitels beschrieben wurde, als Argument.

- **void mesh_set_cmd_payload(mesh_cmd_t *cmd, uint8_t *payload)**:

  Dies ist eine Hilfsfunktion, welche den Inhalt des Payload Buffers des übergebenen Commands auf den übergebenen Payload setzt.

- **void mesh_init()**:

  Diese Funktion muss nach der *mesh_create_network_interface()* Funktion aufgerufen werden. Sie initialisiert das Mesh-Netzwerk, was bedeutet, dass sie vor jeder anderen Funktion aufgerufen werden muss.

3.6 Erläuterung der Verwendung des Mesh-Netzwerks

3.6.1 Übersicht

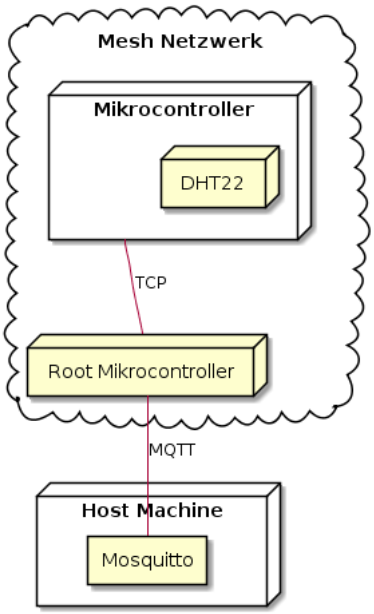

Abbildung 3.28: Deployment Diagramm (Quelle: eigene Darstellung)

In diesem kleinen Beispiel gibt es insgesamt 2 Mikrocontroller:

- Ein **Mikrocontroller**, welcher mit einem DHT22 verbunden ist und regelmäßig die Sensordaten weiterschickt.
- Ein **Root Mikrocontroller**, welcher für das Weiterleiten der Nachrichten zuständig ist und mit einem MQTT-Broker (**Mosquitto**) verbunden ist.

Der Root Mikrocontroller ist mit einem MQTT Broker (**Mosquitto**) verbunden.

Der **Mosquitto** Server läuft auf einer beliebigen **Host Machine** (Bsp.: Digital Ocean Droplet).

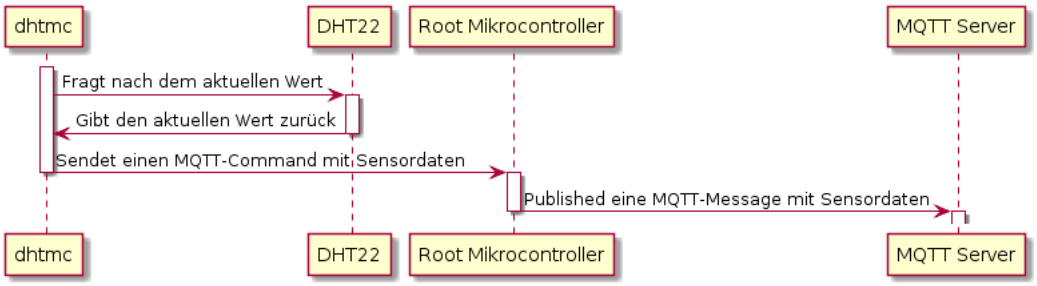

Abbildung 3.29: Sequence Diagramm (Quelle: eigene Darstellung)

Die Abbildung 3.29 zeigt den Verlauf einer Iteration dieses Beispieles. Der **dhtmc** (der **Mikrocontroller** des Deployment Diagramms in Abbildung 3.28) liest die aktuellen Werte des DHT22, welche er dann im Mesh-Netzwerk mittels einem MQTT Command (3.5.2) verschickt.

Wenn der **Root Mikrocontroller** diesen neuen MQTT Command bekommt, schickt dieser die MQTT Nachricht an den angegebenen **MQTT Server** weiter.

3.6.2 Nodejs Setup

Der offiziellen Website von NPM [27] ist der Installationsvorgang für Nodejs zu entnehmen.

Für den **Mesh Visualizer** (Kapitel 3.4) und **OTA Server** (Kapitel 3.2) wird eine Installation von Nodejs 10 benötigt.

3.6.3 ESP-IDF Setup

Die folgenden Anweisungen setzen ein Ubuntu-Betriebssystem voraus. Es wird nicht garantiert, dass dies unter Alternativen wie zum Beispiel dem **Windows Subsystem for Linux (WSL) 3.6.3** funktioniert.

Dependencies

Zuallererst müssen die Dependencies von ESP-IDF installiert werden. Dafür gibt es den folgenden shell command.

```
sudo apt-get install git wget flex bison gperf python python-pip
python-setuptools cmake ninja-build ccache libffi-dev libssl-dev
```

Libraries

Damit man die ESP-IDF Libraries benutzen kann, muss man diese zuerst herunterladen.

```
mkdir $HOME/esp &&
cd $HOME/esp &&
git clone --recursive https://github.com/espressif/esp-idf.git
```

Mit den obigen Befehlen erstellt man zuerst einen neuen Ordner im Homeverzeichnis des aktuell eingeloggten Benutzers und wechselt in diesen. Anschließend wird das offizielle Repository von ESP-IDF heruntergeladen.

Nachdem das Repository erfolgreich gecloned wurde, muss man die Libraries installieren. Dies erfolgt durch folgende Bash Befehle.

```
cd $HOME/esp/esp-idf &&
./install.sh
```

Mit diesen Befehlen wird das Arbeitsverzeichnis auf das vorher installierte Repository gesetzt und die **install.sh** Datei von ESP-IDF ausgeführt.

Dies kann einige Minuten dauern.

Umgebungsvariablen

Damit die Toolchain nun verwendet werden kann, müssen noch ein paar Umgebungsvariablen definiert werden.

```
export IDF_PATH=$HOME/esp/esp-idf
```

Die *IDF_PATH* Variable gibt den Pfad des Repositories von ESP-IDF an.

```
. $HOME/esp/esp-idf/export.sh
```

Danach müssen noch weitere Umgebungsvariablen von ESP-IDF selbst gesetzt werden, dafür muss man den oben angeführten Befehl ausführen.

Diese Variablen werden nur für die aktuelle Shell-Session gesetzt, deswegen wäre es sinnvoll, diese Befehle in die **.bashrc** Datei im Homeverzeichnis einzutragen.

```
export IDF_PATH=$HOME/esp/esp-idf
. $HOME/esp/esp-idf/export.sh
```

Es ist auch möglich, eine eigene Funktion dafür zu definieren, wenn man die Toolchain nur bei Bedarf benutzen möchte.

```
function loadEspIdf() {
    export IDF_PATH=$HOME/esp/esp-idf
    . $HOME/esp/esp-idf/export.sh
}
```

Anschließend muss der Terminal neu gestartet werden, um die **.bashrc** Datei ausführen zu können.

WSL

WSL unterstützt bis zum Stand vom 29.03.2020 das Linux USB Interface nicht. Dies bedeutet, dass für eine fehlerfreie Nutzung der ESP-IDF-Toolchain nicht garantiert wird.

3.6.4 Source Code

Der Source Code der für dieses Beispiel benötigt wird, lebt im folgenden Repository.

```
https://github.com/TimUntersberger/Diplomarbeit
```

Für die erfolgreiche Absolvierung des Beispieles ist es notwendig, das Repository zu clonen.

```
git clone https://github.com/TimUntersberger/Diplomarbeit
```

Nachdem der Befehl fertig ausgeführt ist, befinden sich mehrere Unterordner in dem neu erstellten Ordner namens *Diplomarbeit*. Die einzigen relevanten Ordner für das Beispiel sind **Dht22Example** und **MeshVisualizer**.

Nach Bedarf können die anderen gelöscht werden.

Die zwei Dateien, welche in Kapitel 3.6.5 beschrieben werden, befinden sich in dem Unterordner **MeshVisualizer**.

Genauere Anweisungen zu **MeshVisualizer** befinden sich in dem Kapitel 3.6.6.

Die Struktur und wie der Code von **Dht22Example** benutzt werden kann, wird im Kapitel 3.6.12 beschrieben.

3.6.5 Mosquitto

mosquitto.conf

```
listener 1883
protocol mqtt

listener 1884
protocol websockets
```

Die **mosquitto.conf** Datei konfiguriert den Mosquitto Broker so, dass dieser auf zwei Ports Daten erwartet.

1. 1883
2. 1884

Auf dem Port **1883** hört ein Websocket Server zu, welcher für die Nutzung von dem **Mesh Visualizer** 3.4 konfiguriert wurde.

Der Port 1883 ist wie üblich eine MQTT-Schnittstelle.

docker-compose.yml

```
version: '3'
services:
  mosquitto:
    image: eclipse-mosquitto
    ports:
      - '1883:1883'
      - '1884:1884'
    volumes:
        - ./mosquitto.conf:/mosquitto/config/mosquitto.conf
```

Diese **docker-compose.yml** Datei verwendet die Version 3 von docker-compose. Insgesamt wird nur ein einziger Service benötigt für dieses Beispiel. Der Name des Services ist *mosquitto* und benützt das offizielle Image von eclipse namens *eclipse-mosquitto*.

Wie schon bei der **mosquitto.conf** Datei erwähnt, benötigt Mosquitto zwei offene Ports. Diese werden hier mit den selben äußeren Ports verbunden.

Die Konfigurationsdatei wird mittels eines Volumens in den Container injected.

In der folgenden Abbildung (3.30) ist zu sehen, wie man den Mosquitto nun startet.

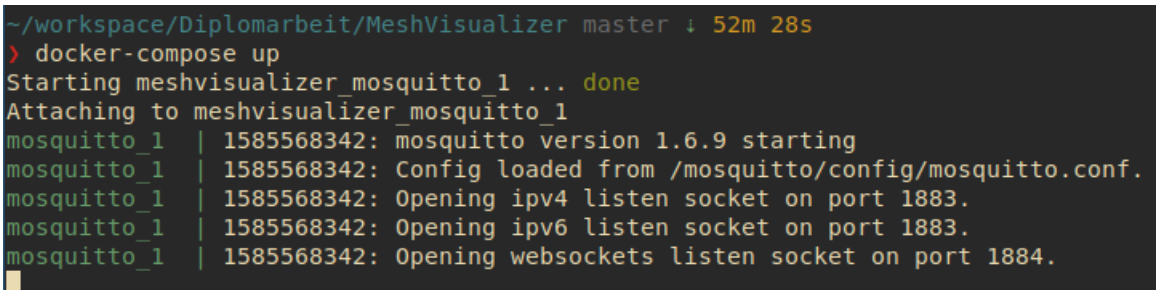

Abbildung 3.30: Example Mosquitto Start (Quelle: eigene Darstellung)

3.6.6 Mesh Visualizer

Nach der erfolgreichen Absolvierung der in Kapitel 3.6.4 beschriebenen Anweisungen befindet sich nun der Unterordner namens **MeshVisualizer** im Ordner Diplomarbeit.

Bevor der **MeshVisualizer** gestartet werden kann, muss das Arbeitsverzeichnis auf den Pfad des **MeshVisualizer** Ordners gesetzt werden.

Anschließend ist es notwendig, die Dependencies des Projekts zu installieren.

Dies erfolgt durch den in Abbildung 3.31 angeführten Command.

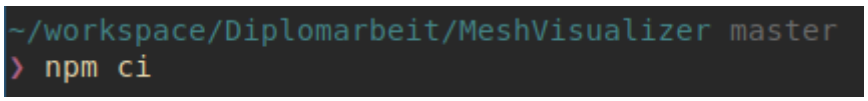

Abbildung 3.31: Example Mesh Visualizer Installation Command (Quelle: eigene Darstellung)

In der nachstehenden Abbildung (3.32) wird das Ergebnis visualisiert.

```
> deasync@0.1.19 install /home/tim/workspace/Diplomarbeit/MeshVisualizer/node_modules/deasync
> node ./build.js

`linux-x64-node-12` exists; testing
Binary is fine; exiting

> core-js@2.6.11 postinstall /home/tim/workspace/Diplomarbeit/MeshVisualizer/node_modules/core-js
> node -e "try{require('./postinstall')}catch(e){}"

Thank you for using core-js ( https://github.com/zloirock/core-js ) for polyfilling JavaScript standard

The project needs your help! Please consider supporting of core-js on Open Collective or Patreon:
> https://opencollective.com/core-js
> https://www.patreon.com/zloirock

Also, the author of core-js ( https://github.com/zloirock ) is looking for a good job -)


> fsevents@1.2.12 install /home/tim/workspace/Diplomarbeit/MeshVisualizer/node_modules/fsevents
> node-gyp rebuild

make: Entering directory '/home/tim/workspace/Diplomarbeit/MeshVisualizer/node_modules/fsevents/build'
  SOLINK_MODULE(target) Release/obj.target/.node
  COPY Release/.node
make: Leaving directory '/home/tim/workspace/Diplomarbeit/MeshVisualizer/node_modules/fsevents/build'

> parcel@1.12.4 postinstall /home/tim/workspace/Diplomarbeit/MeshVisualizer/node_modules/parcel
> node -e "console.log('\u001b[35m\u001b[1mLove Parcel? You can now donate to our open collective:\u001

Love Parcel? You can now donate to our open collective:
 > https://opencollective.com/parcel/donate
added 794 packages in 20.791s
```

Abbildung 3.32: Example Mesh Visualizer Installation Output(Quelle: eigene Darstellung)

Das Starten des Visualizers geschieht wie in der nachstehenden Abbildung (3.33).

```
~/workspace/Diplomarbeit/MeshVisualizer master 21s
❯ npm run start

> MeshVisualizer@1.0.0 start /home/tim/workspace/Diplomarbeit/MeshVisualizer
> parcel index.html

Server running at http://localhost:1234
✨  Built in 12.85s.
```

Abbildung 3.33: Example Mesh Visualizer Start (Quelle: eigene Darstellung)

Nachdem der Server auf dem Port *1234* läuft, ist nun die Website, welche in Abbildung 3.34 zu sehen ist, auf der URL *http://localhost:1234/* verfügbar.

Abbildung 3.34: Example Mesh Visualizer Website (Quelle: eigene Darstellung)

Auf der Website ist momentan nur ein Eingabefeld zusammen mit einem Knopf direkt daneben zu sehen.

Durch das Eingabefeld wird die URL des Mosquittos gesetzt. Die Definition des verwendeten Protokolls darf nicht außer Acht gelassen werden.

Da Javascript im Browser nicht in der Lage ist, mit dem MQTT Protokoll zu kommunizieren, muss man sich mit dem vorher definierten Websocket Port (1884) des Brokers verbinden.

Beispiel

```
ws://localhost:1884
```

Mit dem Drücken des *connect* Knopfes wird versucht, eine Verbindung mit dem Broker herzustellen.

Die Benutzung der Website wird in Kapitel 3.6.11 genauer erläutert.

3.6.7 Partition Table

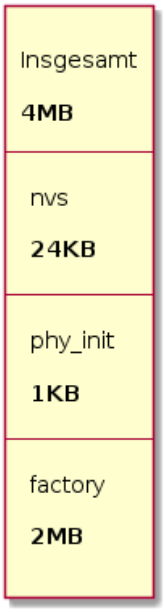

Abbildung 3.35: Example Partiton Table (Quelle: eigene Darstellung)

Der verwendete Mikrocontroller besitzt insgesamt **4MB** an Speicher.

Für das Programm sind **2MB** reserviert, da der default Wert von **1MB** für dieses Beispiel nicht ausreichend ist.

Mehr Informationen zu *Partition Tables* sind in Kapitel 3.2.1 zu finden.

3.6.8 Config

Die Konfiguration eines ESP-IDF Projekts wird im Normalfall mittels der menuconfig gelöst, wie in Kapitel 2.4 erwähnt wurde.

Nach dem Öffnen der menuconfig sind die folgenden Optionen abgebildet.

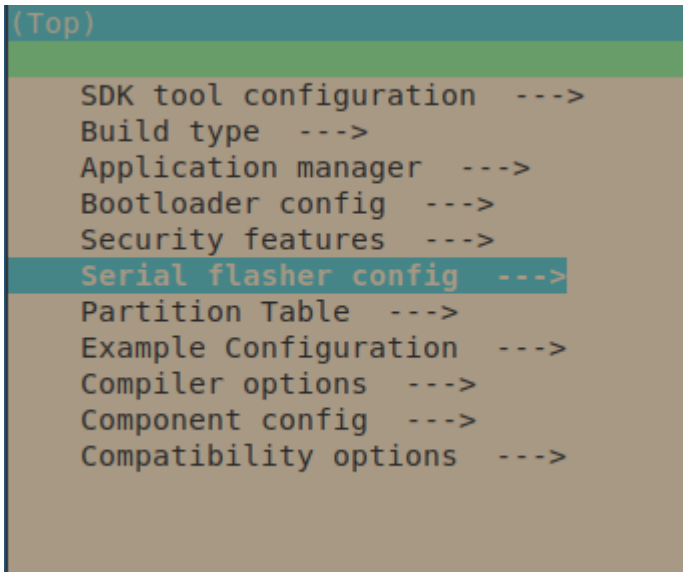

Abbildung 3.36: Example Config Übersicht (Quelle: eigene Darstellung)

Die folgenden Optionen werden im Laufe dieses Kapitels erklärt:

1. Example Configuration (Kapitel 3.6.8)
2. Serial flasher config (Kapitel 3.6.8)
3. Partition Table (Kapitel 3.6.8)

Nachdem das Programm fertig konfiguriert ist, kann die menuconfig geschlossen werden.

Example Configuration

In der **Example Configuration** Option wird das Mesh-Netzwerk und der MQTT-Client konfiguriert.

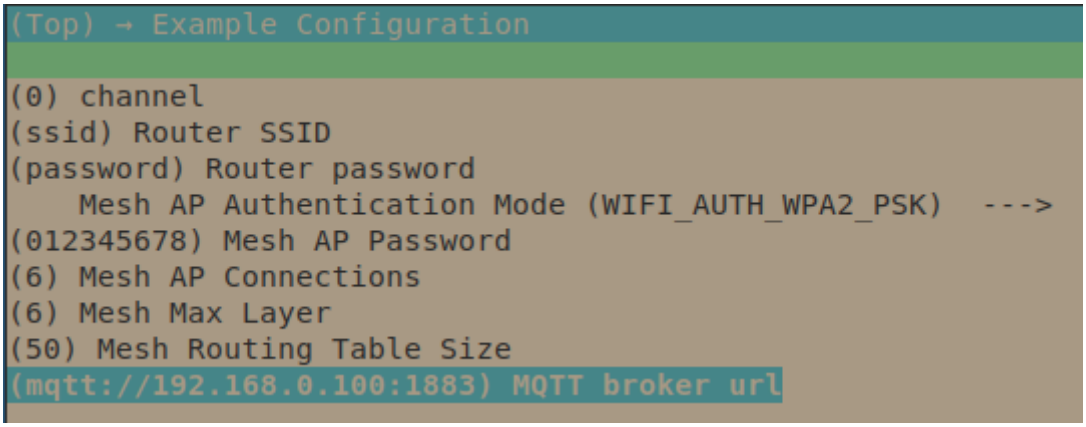

Abbildung 3.37: Example Config Example Configuration (Quelle: eigene Darstellung)

Folgende Werte müssen angepasst werden:

- **Router SSID**
  Die SSID des WLAN Routers, womit sich der Mikrocontroller verbinden soll.
- **Router password**
  Das Passwort des WLAN Routers, womit sich der Mikrocontroller verbinden soll.
- **Mesh AP Password**
  Das Mesh-Netzwerk interne Passwort. Das Passwort muss mindestens acht Zeichen lang sein und mindestens eine Zahl beinhalten.
- **MQTT broker url**
  Die URL des MQTT Brokers. Hier ist es wichtig, dass nicht localhost eingegeben wird, da dieses Programm auf dem esp läuft (dies ist am Anfang der Arbeit passiert).
  Das Schema muss der URL im Beispiel unten in Abbildung 3.37 folgen.

Serial flasher config

In der **Serial flasher config** Option wird der Serial Flasher konfiguriert.

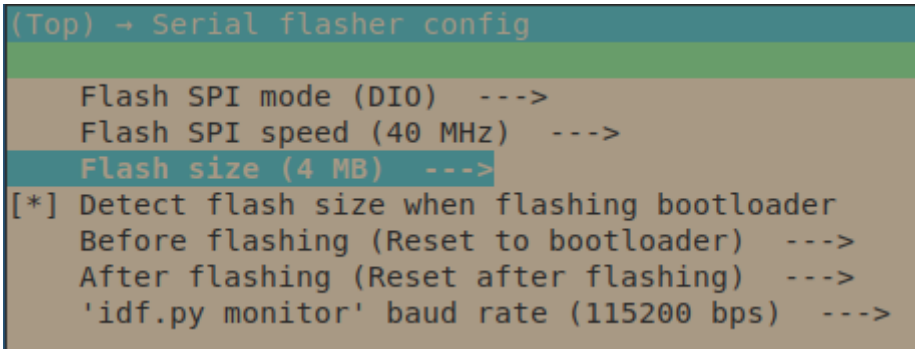

Abbildung 3.38: Example Config Serial Flasher Config (Quelle: eigene Darstellung)

Hier muss nur die **Flash size** angepasst werden. Die Größe hängt vom jeweiligen Modell des ESPs ab.

Partition Table

In der **Partition Table** Option wird der verwendete *Partition Table* konfiguriert.

Abbildung 3.39: Example Config Partition Table (Quelle: eigene Darstellung)

In dieser Konfiguration muss der Typ des **Partition Table** Felds auf *Custom partition table CSV* gesetzt werden.

3.6.9 Hardware

DHT22 mit ESP32 verbinden

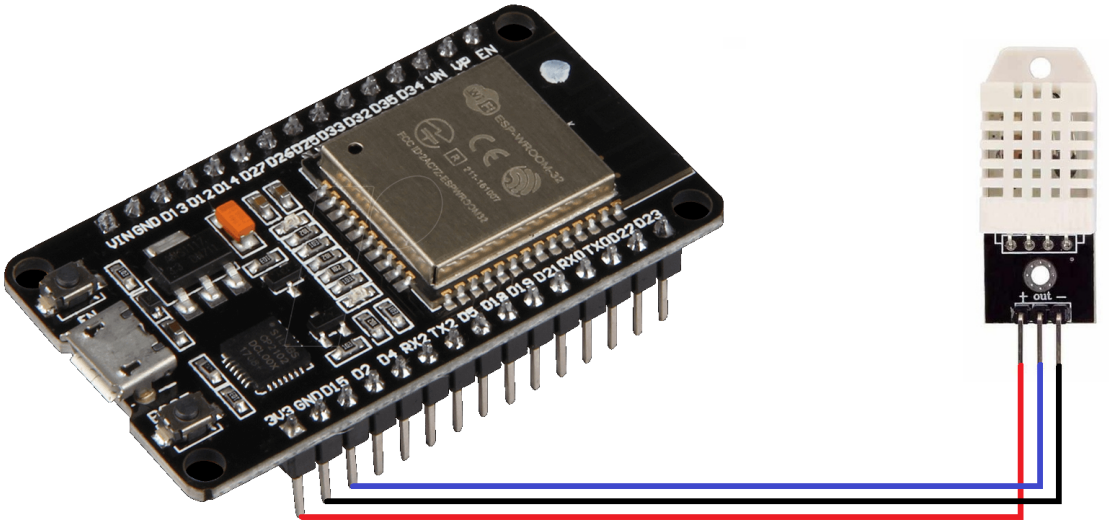

Abbildung 3.40: Verbindung ESP32-DHT22 [9, 24]

Um die zwei in Abbildung 3.40 dargestellten Module miteinander zu verbinden, sind drei weiblich zu weiblich Dupont-Kabel notwendig.

Dabei ist die Farbe der Kabel egal, es wird jedoch empfohlen, für den GND Pin immer ein schwarzes und für den 3.3V Pin immer ein rotes Kabel zu verwenden, da ansonsten die Übersicht verloren geht.

Nun werden folgende Pins miteinander verbuden:

- Der - Pin des DHT22 mit dem GND Pin des Mikrocontrollers (in Abbildung 3.40 schwarz markiert)
- Der + Pin des DHT22 mit dem 3.3V Pin des Mikrocontrollers (in Abbildung 3.40 rot markiert)
- Der out Pin des DHT22 mit einem beliebigem data Pin des Mikrocontrollers (in Abbildung 3.40 blau markiert)

ESP32 mit Computer verbinden

Um den ESP mit dem Computer zu verbinden, wird ein gewöhnliches USB zu Micro-USB Kabel verwendet.

Das Micro-USB-Ende des Kabels wird in den seriellen Ausgang des Mikrocontrollers gesteckt und das USB-Ende des Kabels in einen beliebigen USB-Port des gewünschten Computer.

3.6.10 Flashen

Bevor das Programm auf den ESPs laufen kann, muss es hochgeladen werden.

Das Hochladen bzw. Flashen eines Programms verläuft wie in Kapitel 2.4 beschrieben.

3.6.11 Ergebnis

Mesh Visualizer

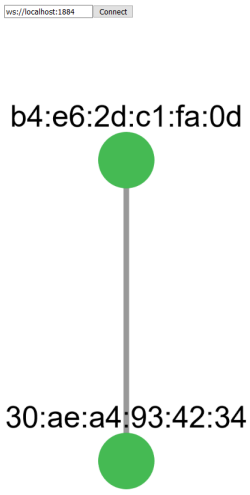

Abbildung 3.41: Beispiel Ergebnis Mesh Visualizer (Quelle: eigene Darstellung)

In Abbildung 3.41 sieht man, wie sich die Mesh Visualizer Website verändert hat.

In dieser Abbildung sind zwei Knoten mit jeweils einer MAC Addresse zu sehen.

Die MAC Addressen stehen für den jeweiligen ESP, der zu diesem Netzwerk verbunden ist.

Die Knoten sind in einer baumartigen Struktur abgebildet. Der Knoten an der Spitz des Diagramms ist der **Root Knoten**.

Mqtt Fx

Wie in der nachstehenden Abbildung zu sehen ist, wird regelmäßig die aktuelle Temperatur und Luftfeuchtigkeit an den MQTT Broker gesendet.

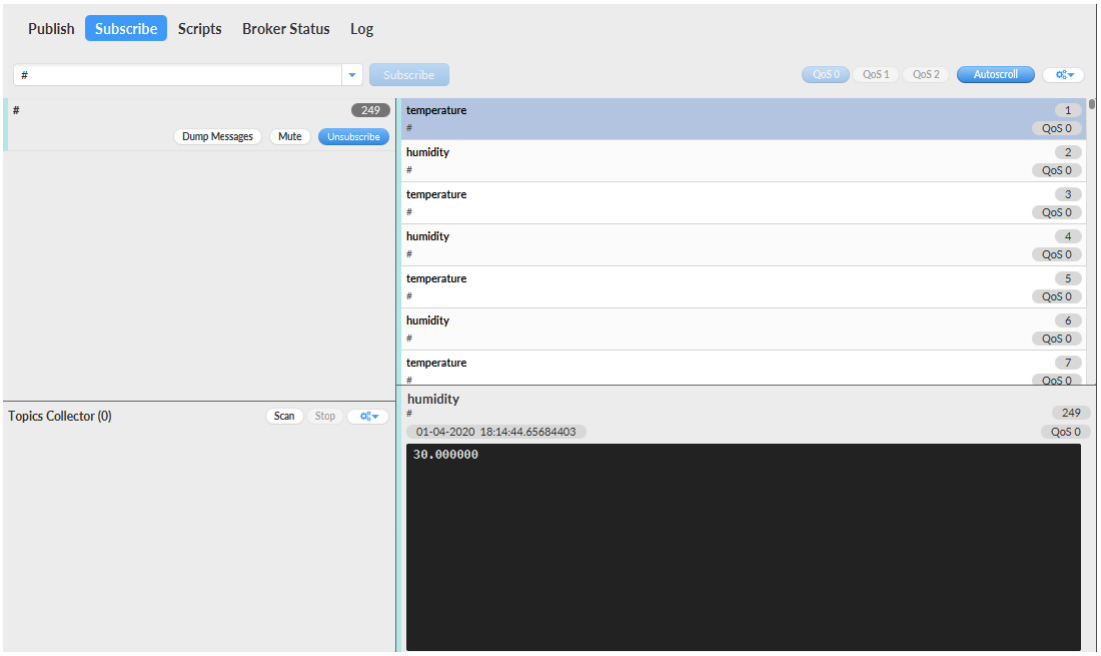

Abbildung 3.42: Beispiel Ergebnis Mqtt (Quelle: eigene Darstellung)

Der MQTT Client, welcher in Abbildung 3.42 zu sehen ist, wurde von hivemq entwickelt und dient zur Kommunizierung mit einem MQTT-Broker.

3.6.12 Code

Struktur

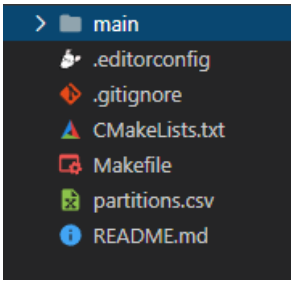

Abbildung 3.43: Beispiel Code Struktur Übersicht (Quelle: eigene Darstellung)

Die Übersicht in Abbildung 3.43 zeigt einen kleinen Ausschnitt der Struktur.

Der Stammordner beinhaltet die Konfigurations Dateien sowie den **main** Ordner, welcher den Source Code beinhaltet.

Die **.editorconfig** Datei ist dazu da, die Formatierung des Source Codes in allen Text-Editoren uniform zu halten.

In der **partitions.csv** Datei befindet sich die Beschreibung des in Kapitel 3.6.8 erwähnten Partition Tables.

Die nachstehende Abbildung zeigt die Struktur des **main** Ordners.

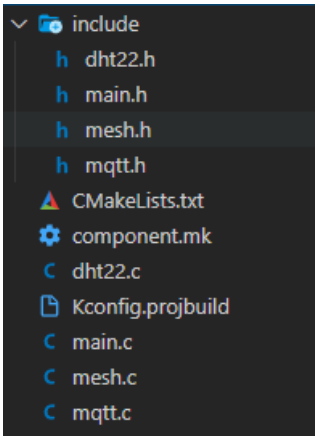

Abbildung 3.44: Example Code Struktur Main (Quelle: eigene Darstellung)

Dieser beinhaltet nicht nur den Source Code sondern auch die *menuconfig* Datei, welche in Kapitel 2.4 genauer beschrieben ist.

Informationen über die einzelnen Libraries, die im Laufe dieser Arbeit enstanden sind (wie zum Beispiel *mqtt.h*), befinden sich im Kapitel 3.5.2.

Setup

```
void app_main(void)
{
    ESP_ERROR_CHECK(nvs_flash_init());
    ESP_ERROR_CHECK(esp_netif_init());
    ESP_ERROR_CHECK(esp_event_loop_create_default());

    mesh_create_network_interface();

    wifi_init();
    mesh_init();

    mesh_on_cmd(&on_cmd);
    mesh_on_connected(&on_connected);

    mesh_start();
}
```

Abbildung 3.45: Example Code Setup (Quelle: eigene Darstellung)

Die in Abbildung 3.45 dargestellte Funktion *app_main* ist für die Initialisierung des Programms zuständig.

Sie initialisiert folgende Komponenten:

- nvs flasher
- netif
- event loop
- wifi
- mesh

Anschließend startet sie das Mesh-Netzwerk.

Wifi Setup

```
void wifi_init(){
    wifi_init_config_t config = WIFI_INIT_CONFIG_DEFAULT();
    ESP_ERROR_CHECK(esp_wifi_init(&config));
    ESP_ERROR_CHECK(esp_event_handler_register(IP_EVENT, IP_EVENT_STA_GOT_IP, &ip_event_handler, NULL));
    ESP_ERROR_CHECK(esp_wifi_set_storage(WIFI_STORAGE_FLASH));
    ESP_ERROR_CHECK(esp_wifi_start());
}
```

Abbildung 3.46: Example Code Setup Wifi (Quelle: eigene Darstellung)

In Abbildung 3.46 ist eine Funktion namens *wifi_init* zu sehen. Diese ist für die Initialisierung des Wifis zuständig.

Die Funktion verwendet die default Werte von ESP-IDF für die Initialisierung des Wifis. Es wird nicht nur das wifi gestartet sondern auch ein *Callback* für das *IP_EVENT_STA_GOT_IP* Event registriert.

Der *Callback* wird in Kapitel 3.6.12 beschrieben.

IP Event Handler

```
void ip_event_handler(void *arg, esp_event_base_t event_base,
                      int32_t event_id, void *event_data)
{
    ip_event_got_ip_t *event = (ip_event_got_ip_t *) event_data;

    ESP_LOGI(TAG, "SUCCESSFULLY CONNECTED TO ROUTER");
    ESP_LOGI(TAG, "MY IP:" IPSTR, IP2STR(&event->ip_info.ip));
    is_connected_to_router = true;
    mqtt_start();
}
```

Abbildung 3.47: Example Code Event Handler (Quelle: eigene Darstellung)

Die Funktion *ip_event_handler*, welche in Abbildung 3.47 zu sehen ist, startet den MQTT-Client, nachdem sie die globale Variable *is_connected_to_router* setzt.

is_connected_to_router steht für den aktuellen Status des Wifis. Normalerweise ist dieser immer auf *false* gesetzt, da nur der *Root Mikrocontroller* eine Verbindung mit dem externen Netz aufbauen kann.

Command Callback

Der *Command Callback* wird aufgerufen, wenn ein neuer *Command* an diesen Knoten verschickt wird.

Was ein *Command* ist und wie er funktioniert, ist in Kapitel 3.5.2 beschrieben.

Der Callback reagiert nur auf neue Commands, wenn der aktuelle Knoten der *Root Knoten* ist.

Das Beispiel Programm reagiert auf folgende Command Typen:

- **MQTT**
- **ADD NODE**
- **REMOVE NODE**

ADD_NODE und **REMOVE_NODE** sind Commands, die automatisch von der Mesh Library verschickt werden.

Das Programm reagiert auf diese beiden Commands und schickt sie dann über MQTT im JSON Format weiter.

Dies ist wichtig für den *Mesh Visualizer* (Kapitel 3.4).

Erreicht ein MQTT Command den Root Knoten, so versendet dieser mithilfe der MQTT Library (Kapitel 3.5.2) den Payload, welcher die MQTT Message beinhaltet.

```
mqtt_msg_t *mqtt_msg = (mqtt_msg_t*) cmd->payload;
mqtt_publish_msg(mqtt_msg);
```

Abbildung 3.48: Example Code Command Callback 1 (Quelle: eigene Darstellung)

DHT22 Task

```
void DHT_task(void *pvParameter)
{
    while (1)
    {
        int ret = dht22_read();
        float humidity = dht22_get_humidity();
        float temperature = dht22_get_temperature();

        dht22_handle_error(ret);

        ESP_LOGI(TAG, "Humidity: %.1f\n", humidity);
        ESP_LOGI(TAG, "Temperature: %.1f\n", temperature);

        mqtt_msg_t msg = {0};

        snprintf(msg.topic, MQTT_MESSAGE_TOPIC_SIZE, "temperature");
        snprintf(msg.payload, MQTT_MESSAGE_PAYLOAD_SIZE, "%f", temperature);

        mqtt_publish_msg(&msg);

        snprintf(msg.topic, MQTT_MESSAGE_TOPIC_SIZE, "humidity");
        snprintf(msg.payload, MQTT_MESSAGE_PAYLOAD_SIZE, "%f", humidity);

        mqtt_publish_msg(&msg);

        vTaskDelay(3000 / portTICK_RATE_MS);
    }
}
```

Abbildung 3.49: Example Code DHT22 Task (Quelle: eigene Darstellung)

DHT_task ist eine Funktion, welche als Task (Kapitel 3.5.1) gestartet wird.

Sie ist für das regelmäßige Einlesen der aktuellen DHT22 Werte und das Verschicken dieser an den spezifizierten MQTT Broker zuständig.

Ungefähr alle drei Sekunden läuft dieser Task die in Abbildung 3.29 zu sehende Sequenz durch.

```
int ret = dht22_read();

if (ret != ESP_OK)
{
    dht22_handle_error(ret);
    continue;
}

float humidity = dht22_get_humidity();
float temperature = dht22_get_temperature();

ESP_LOGI(TAG, "Humidity: %.1f\n", humidity);
ESP_LOGI(TAG, "Temperature: %.1f\n", temperature);
```

Abbildung 3.50: Example Code DHT22 Task 1 (Quelle: eigene Darstellung)

Am Anfang einer Iteration werden die aktuellen Werte des DHT22 Sensors gelesen. Funktioniert etwas beim Einlesen der Werte nicht, so wird dies mithilfe der *dht22_handle_error* Funktion behandelt und die aktuelle Iteration übersprungen.

Wenn das Einlesen erfolgreich ist, holt sich das Programm die gespeicherte Feuchtigkeit und Temperatur (Kapitel 3.5.1) und gibt diese aus.

```
mqtt_msg_t msg = {0};

snprintf(msg.topic, MQTT_MESSAGE_TOPIC_SIZE, "temperature");
snprintf(msg.payload, MQTT_MESSAGE_PAYLOAD_SIZE, "%f", temperature);

mqtt_publish_msg(&msg);

snprintf(msg.topic, MQTT_MESSAGE_TOPIC_SIZE, "humidity");
snprintf(msg.payload, MQTT_MESSAGE_PAYLOAD_SIZE, "%f", humidity);

mqtt_publish_msg(&msg);
```

Abbildung 3.51: Example Code DHT22 Task 2 (Quelle: eigene Darstellung)

Anschließend wird die Temperatur und Feuchtigkeit in eine MQTT Message umgewandelt und verschickt.

3.7 ELF vs. Bin

Platform IO generiert beim Kompilieren eines Projektes zwei Firmware Dateien. Eine Firmware Datei ist im *ELF* Format und die andere im *Bin* Format.

3.7.1 Bin

Die Dateierweiterung .bin wird am häufigsten mit komprimierten Binärdateien verknüpft. Diese Dateien werden von vielen verschiedenen Computeranwendungen und für eine Vielzahl von Zwecken verwendet. Die Erweiterung .bin wird häufig für CD- und DVD-Backup-Image-Dateien verwendet.

In einigen Fällen werden die BIN-Dateien im einfachen Binärformat gespeichert und können mit einem Texteditor geöffnet werden. Es gibt jedoch einige BIN-Dateien, die von bestimmten Computeranwendungen erstellt werden und mit der Software geöffnet werden müssen, mit der sie erstellt wurden, oder mit einer kompatiblen Softwareanwendung.[2]

Hintergrund

Grundsätzlich sind Binärdateien als solche daran erkennbar, dass der Dateiinhalt, mit einem üblichen Texteditor angezeigt, keine oder überwiegend keine lesbaren Zeichen enthält. Der Versuch, eine Binärdatei als Textdatei zu interpretieren (beispielsweise durch Öffnen mit einem Texteditor), ergibt dann unleserlichen bzw. unsinnigen Text. Für die meisten der heute verwendeten 8-Bit-Zeichensätze gilt: nicht lesbare Steuerzeichen umfassen Zeichen mit ASCII-Werten von 0 bis 31, lesbare Zeichen die mit Werten von 32 bis 126. Die Lesbarkeit von Zeichen mit Werten ab 127 ist abhängig vom verwendeten Zeichensatz. Textdateien können auch gewisse Steuerzeichen enthalten, ohne dass sie deshalb als Binärdatei gelten; dazu gehören Steuerzeichen für Zeilenvorschub, Wagenrücklauf, Seitenumbruch (Seitenvorschub) und Tabulatorzeichen.

Weil Binärdateien alle möglichen Bit-Kombinationen nutzen, bieten sie eine höhere Informationsdichte als Textdateien. Deshalb benötigen sie meist weniger Speicherplatz auf Massenspeichern und lassen sich schneller laden und speichern. Ferner lassen sich darin verschiedene Objekttypen (beispielsweise Text mit Bildern) relativ einfach ablegen.

Binärformate werden beim Austausch über verschiedene Plattformen hinweg (beispielsweise Windows, Macintosh, Linux) nicht beschädigt, da die jeweiligen Softwarekomponenten nicht versuchen, die Dateien für die Zielplattform zu konvertieren. Andererseits wird der systemübergreifende Datenaustausch erschwert, da Binärdateien häufig Daten in einem systemabhängigen Format enthalten. (Beispielsweise Zahlen im Big- oder Little-Endian-Format.) Die Spezifikation des Dateiformats einer Binärdatei legt fest, wie mit der Datei zu verfahren ist. Zum Lesen, Bearbeiten und Speichern binärer Datenformate benötigt man im Allgemeinen spezielle, auf das Dateiformat abgestimmte Editoren (beispielsweise Textverarbeitung für Office-Texte, ein Bildbearbeitungsprogramm für Fotos, regedit für die Windows-Registrierungsdatenbank).

Zu beachten ist, dass man unter einer Binärdatei bzw. unter Binärformat nicht Daten versteht, die nur aus den (sichtbaren) Zeichen „0“ und „1“ aufgebaut sind – wie die Namensanalogie zu Hex(adezimal)datei nahelegen könnte. Binärdatei bedeutet auch nicht, dass die Daten nur aus binären „0“ und „1“ bestehen – weil das auch bei Text-Zeichensätzen der Fall ist. Auch ist eine Datei, die von einem Textverarbeitungsprogramm erzeugt wurde, meist (abhängig vom Dateiformat) keine reine Textdatei im engeren Sinn, sondern eine Binärdatei, in der zum Beispiel Formatangaben und andere Steuerzeichen nicht mit einem lesbaren Zeichensatz codiert sind. Solche Dateien, zum Beispiel im Rich-Text-Format, sind insofern eine Mischform aus Text- und Binärdatei.[3]

3.7.2 ELF

Das Folgende ist eine Zusammenfassung der wichtigsten Aspekte und Eckpunkte der Website [13]

ELF ist die Abkürzung für Executable and Linkable Format und definiert die Struktur für Binärdateien, Bibliotheken und Core-Dateien. Die formale Spezifikation ermöglicht es dem Betriebssystem, die zugrunde liegenden Maschinenanweisungen korrekt zu interpretieren. ELF-Dateien sind normalerweise die Ausgabe eines Compilers oder Linkers und haben ein Binärformat.

Ein häufiges Missverständnis ist, dass ELF-Dateien nur für Binärdateien oder ausführbare Dateien bestimmt sind. Es ist jedoch möglich, sie für Teilstücke (Objektcode) verwendet zu können. Ein weiteres Beispiel sind gemeinsam genutzte Bibliotheken oder sogar Core-Dumps (Core- oder a.out-Dateien). Die ELF-Spezifikation wird auch unter Linux für den Kernel selbst und die Linux-Kernelmodule verwendet.

Struktur

Aufgrund des erweiterbaren Designs von ELF-Dateien unterscheidet sich die Struktur je nach Datei. Eine ELF-Datei besteht aus:

- ELF-Header
- Dateidaten

Mit dem Befehl readelf können wir uns die Struktur einer Datei ansehen und sie sieht ungefähr so aus:

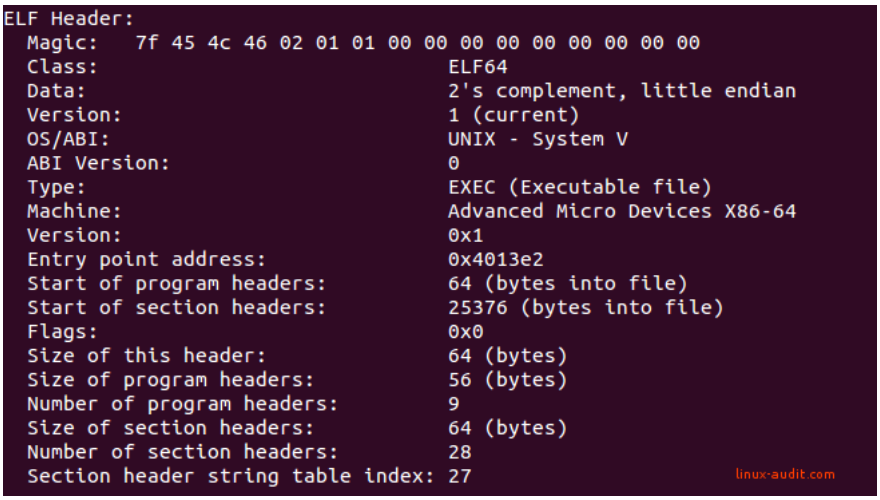

Abbildung 3.52: Details einer Elf-binary [8]

3.7.3 ELF-Header

Wie in Abbildung 3.52 zu sehen ist, beginnt der ELF-Header mit etwas Magic. Diese ELF-Header-Magic liefert Informationen über die Datei. Die ersten 4 hexadezimalen Teile definieren, dass dies eine ELF-Datei ist (45 = E, 4c = L, 46 = F), der der Wert 7f vorangestellt ist.

Dieser ELF-Header ist obligatorisch. Es stellt sicher, dass Daten während der Verknüpfung oder Ausführung korrekt interpretiert werden. Um die innere Funktionsweise einer ELF-Datei besser zu verstehen, ist es hilfreich zu wissen, dass diese Header-Informationen verwendet werden.

Class

Nach der ELF-Typdeklaration ist ein Class-feld definiert. Dieser Wert bestimmt die Architektur für die Datei. Es kann sich um eine 32-Bit- (= 01) oder 64-Bit- (= 02) Architektur handeln. Magic zeigt eine 02, die vom Befehl readelf als ELF64-Datei übersetzt wird, mit anderen Worten, eine ELF-Datei, die die 64-Bit-Architektur verwendet.

Data

Der nächste Teil ist das Data-Feld. Es kennt zwei Optionen: 01 für LSB (Least Significant Bit), auch als Little-Endian bekannt. Dann gibt es den Wert 02 für MSB (Most Significant Bit, Big-Endian). Dieser spezielle Wert hilft dabei, die verbleibenden Objekte in der Datei korrekt zu interpretieren. Dies ist wichtig, da verschiedene Prozessortypen unterschiedlich mit den eingehenden Anweisungen und Datenstrukturen umgehen. In diesem Fall wird LSB verwendet, was für Prozessoren vom Typ AMD64 üblich ist.

Version

Als nächstes folgt eine weitere “01“ in der Magic, die die Versionsnummer ist. Derzeit gibt es nur einen Versionstyp: dies ist momentan der Wert ”01”.

OS / ABI

Jedes Betriebssystem hat eine große Überlappung in gemeinsamen Funktionen. Darüber hinaus hat jedes von ihnen spezifische oder zumindest geringfügige Unterschiede. Die Definition des richtigen Sets erfolgt über eine Application Binary Interface (ABI). Auf diese Weise wissen sowohl das Betriebssystem als auch die Anwendungen, was zu erwarten ist, und die Funktionen werden korrekt weitergeleitet. Diese beiden Felder beschreiben, für was ABI verwendet wird und die zugehörige Version. In diesem Fall ist der Wert 00, was bedeutet, dass keine bestimmte Erweiterung verwendet wird. Die Ausgabe zeigt dies als System V.

ABI-Version

Bei Bedarf kann eine Version für das ABI angegeben werden.

Machine

Den Maschinentyp (AMD64) finden wir auch im Header.

Type

Das Typfeld gibt an, wozu die Datei dient. Es gibt einige gängige Dateitypen.

- CORE (Wert 4)
- DYN (Shared Object File) für Bibliotheken (Wert 3)
- EXEC (ausführbare Datei) für Binärdateien (Wert 2)
- REL (verschiebbare Datei), bevor sie in eine ausführbare Datei gelinkt wird (Wert 1)

3.7.4 File Data

Neben dem ELF-Header bestehen ELF-Dateien aus drei Teilen.

- Program Headers oder Segments (9)
- Section Headers oder Sections (28)
- Data

Außerdem ist es gut zu wissen, dass ELF zwei sich ergänzende „Ansichten“ hat. Eine Benutzeroberfläche muss für den Linker verwendet werden, um die Ausführung zu ermöglichen (segments), die andere zum Kategorisieren von Anweisungen und Daten (sections). Je nach Ziel werden also die zugehörigen Headertypen verwendet.

Programm-Header

Eine ELF-Datei besteht aus null oder mehr Segmenten und beschreibt, wie ein process/memory image für die Laufzeitausführung erstellt wird. Wenn der Kernel diese Segmente sieht, verwendet er sie, um sie mithilfe des Systemaufrufs mmap (2) dem virtuellen Adressraum zuzuordnen. Mit anderen Worten, es konvertiert vordefinierte Anweisungen in ein Speicherbild. Wenn eine ELF-Datei eine normale Binärdatei ist, sind diese Programmheader erforderlich. Andernfalls wird es einfach nicht ausgeführt. Diese Header mit der zugrunde liegenden Datenstruktur werden verwendet, um einen Prozess zu bilden. Dieser Vorgang ist für shared libraries ähnlich.

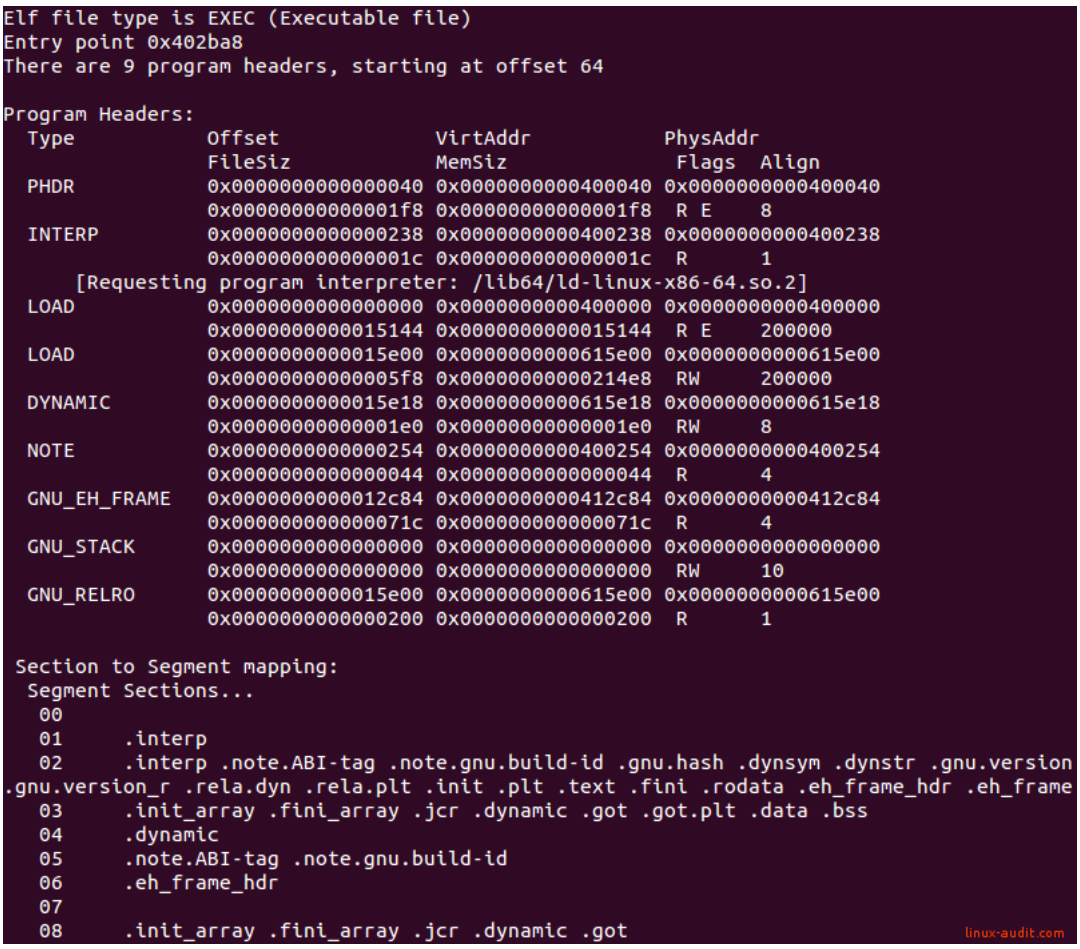

Abbildung 3.53: Programm Header einer Elf binary [26]

GNU_EH_FRAME

Dies ist eine sorted queue, die vom GNU C-Compiler (gcc) verwendet wird. Es speichert Ausnahmebehandlungsroutinen. Wenn also etwas schief geht, kann es diesen Bereich verwenden, um richtig damit umzugehen.

GNU_STACK

Dieser Header wird zum Speichern von stack-Informationen verwendet. Der stack ist ein Puffer oder eine Arbeitsstelle, an der Elemente wie lokale Variablen gespeichert werden. Dies geschieht bei LIFO (Last In, First Out). Beim Starten einer Prozessfunktion wird ein Baustein reserviert. Wenn die Funktion beendet ist, wird sie wieder als frei markiert. Der interessante Teil ist nun, dass ein Stack nicht ausführbar sein sollte, da dies zu Sicherheitslücken führen kann. Durch Manipulation des Speichers könnte man auf diesen ausführbaren Stack verweisen und beabsichtigte Anweisungen ausführen.

Wenn das Segment GNU_STACK nicht verfügbar ist, wird normalerweise ein executable stack verwendet. Die Tools scanelf und execstack sind zwei Beispiele, um die stack-Details anzuzeigen.

```
# scanelf -e /bin/ps
 TYPE   STK/REL/PTL FILE
ET_EXEC RW- R-- RW- /bin/ps

# execstack -q /bin/ps
- /bin/ps
```

Abbildung 3.54: Beispiel GNU_STACK [1]

3.7.5 Static vs. Dynamic binaries

Beim Umgang mit ELF-Binärdateien ist es gut zu wissen, dass es zwei Typen gibt und wie sie verknüpft sind. Der Typ ist entweder statisch oder dynamisch und bezieht sich auf die verwendeten Bibliotheken. Zu Optimierungszwecken stellen wir häufig fest, dass Binärdateien „dynamisch“ sind, was bedeutet, dass externe Komponenten für die ordnungsgemäße Ausführung erforderlich sind. Häufig handelt es sich bei diesen externen Komponenten um normale Bibliotheken, die allgemeine Funktionen wie das Öffnen von Dateien oder das Erstellen eines Netzwerk-Sockets enthalten. In statischen Binärdateien sind dagegen alle Bibliotheken enthalten. Dadurch werden sie größer und dennoch tragbarer (z. B. wenn sie auf einem anderen System verwendet werden).

3.7.6 Fazit

ELF-Dateien dienen zur Ausführung oder zum Verknüpfen. Abhängig vom primären Ziel enthält es die erforderlichen Segmente oder Abschnitte. Segmente werden vom Kernel angeschaut und auf den Speicher gemappt (mithilfe von mmap). Abschnitte werden vom Linker angeschaut, um ausführbaren Code oder freigegebene Objekte zu erstellen.

Der ELF-Dateityp ist sehr flexibel und bietet Unterstützung für mehrere CPU-Typen, Maschinenarchitekturen und Betriebssysteme. Es ist auch sehr erweiterbar: Jede Datei ist je nach den erforderlichen Teilen unterschiedlich aufgebaut.

Header bilden einen wichtigen Teil der Datei und beschreiben genau den Inhalt einer ELF-Datei. Mit den richtigen Tools erhalten Sie ein grundlegendes Verständnis des Zwecks der Datei. Von dort aus können Sie die Binärdateien weiter überprüfen. Dies kann durch Bestimmen der zugehörigen Funktionen oder der in der Datei gespeicherten Zeichenfolgen erfolgen.

Kapitel 4

Resümee

4.1 Tim Untersberger

Im Laufe dieser Arbeit habe ich das Arbeiten mit Mikrocontrollern näher kennengelernt und meine C/C++ verbessern dürfen.

Da die Entwicklung mit Mikrocontrollern neu für mich war, erwies sich der Beginn der Arbeit als holprig. Durch mein kontinuierlich wachsendes Verständnis über das IOT-Ökosystem machte mir das Programmieren mehr und mehr Spaß.

Da ich bis jetzt hauptsächlich Projekte alleine entwickelt habe, fiel mir die Zusammenarbeit mit einem Partner anfangs eher schwer. Dank fortwährender Kommunikation konnten wir diese Hürde als Team meistern.

Durch diese Arbeit habe ich wichtige Grundsätze für die Zukunft meiner Softwareentwickler Karriere gelernt.

4.2 Stefan Waldl

Beim Arbeiten an dieser Diplomarbeit konnte ich viel für mein Leben lernen. Der größte Punkt, was das betrifft, war das Arbeiten mit einem Partner. Man lernt, wie wichtig es ist einfach aber trotzdem prägnant zu kommunizieren, da sonst bei einem größeren Projekt wie diesem leicht Missverständnise auftreten können.

Anfangs war es sehr mühsam, auf Mikrocontrollern zu programmieren, da sich diese ganz anders verhalten als herkömmliche Computer. Doch als ich mich an die Eigenheiten der Mikrocontrollern gewöhnt hatte, war es mir möglich, in einen produktiven Workflow zu gelangen.

Im Großen und Ganzen bin ich froh, mein Know-How um die Welt der Mikrocontroller erweitern haben zu können.

Literaturverzeichnis

[1] Beispiel GNU_STACK.
https://linux-audit.com/elf-binaries-on-linux-understanding-and-analysis/.

Abgerufen am 16.2.2020.

[2] Bin File.org.
https://file.org/extension/bin.
Abgerufen am 15.2.2020.

[3] Bin Wikipedia.org.
https://de.wikipedia.org/wiki/Bin\protect\unhbox\voidb@x\bgroup\U@
D1ex{\setbox\z@\hbox{\char127}\dimen@-.45ex\advance\dimen@\ht\z@}\
accent127\fontdimen5\font\U@Da\egrouprdatei.
Abgerufen am 15.2.2020.

[4] Cytoscape Dagre.
https://cytoscape.org/cytoscape.js-dagre/.
Abgerufen am 1.4.2020.

[5] Cytoscape Euler.
https://cytoscape.org/cytoscape.js-euler/.
Abgerufen am 1.4.2020.

[6] Cytoscape Intro.
https://js.cytoscape.org/.
Abgerufen am 1.4.2020.

[7] Cytoscape Spread.
https://cytoscape.org/cytoscape.js-spread/.
Abgerufen am 1.4.2020.

[8] Details einer Elf binary.
https://assets.linux-audit.com/wp-content/uploads/2015/08/
elf-header-linux-binary.png.
Abgerufen am 16.2.2020.

[9] DHT22 Bild.
https://5.imimg.com/data5/MT/CE/MY-43948449/
dht22-temperature-500x500.jpg.
Abgerufen am 30.3.2020.

[10] DHT22 Bild.
https://www.sparkfun.com/datasheets/Sensors/Temperature/DHT22.pdf.
Abgerufen am 30.3.2020.

[11] Docker-compose Beschreibung.
https://docs.docker.com/compose/.
Abgerufen am 2.2.2020.

[12] Docker Definiert.
https://docs.microsoft.com/en-us/dotnet/architecture/microservices/
container-docker-introduction/docker-defined.
Abgerufen am 1.4.2020.

[13] ELF.
https://linux-audit.com/elf-binaries-on-linux-understanding-and-analysis/.

Abgerufen am 16.2.2020.

[14] ESP-IDF Github.
https://github.com/espressif/esp-idf.
Abgerufen am 2.4.2020.

[15] ESP-IDF Github Template.
https://github.com/espressif/esp-idf/tree/master/examples/
get-started/hello_world.
Abgerufen am 2.4.2020.

[16] ESP-MESH.
https://docs.espressif.com/projects/esp-idf/en/latest/esp32/
api-guides/mesh.html.
Abgerufen am 1.4.2020.

[17] ESP32-WROOM-32 Spezifikatoinen.
https://www.espressif.com/sites/default/files/documentation/
esp32-wroom-32_datasheet_en.pdf.
Abgerufen am 30.3.2020.

[18] Espressif Partitionstabelle.
https://docs.espressif.com/projects/esp-idf/en/latest/api-guides/
partition-tables.html.
Abgerufen am 9.3.2020.

[19] Github DHT22 Library.
https://github.com/gosouth/DHT22.
Abgerufen am 2.4.2020.

[20] HiveMq Website.
https://www.hivemq.com/blog/mqtt-essentials-part-6-mqtt-quality-of-service-levels/.

Abgerufen am 1.4.2020.

[21] Http Methoden.
https://github.com/espressif/esp-idf/blob/master/components/esp_http_client/include/esp_http_client.h#L75.
Abgerufen am 1.4.2020.

[22] menuconfig.
https://docs.espressif.com/projects/esp-idf/en/latest/esp32/get-started/index.html.
Abgerufen am 31.3.2020.

[23] Node Package Manager.
https://docs.npmjs.com/about-npm/.
Abgerufen am 2.4.2020.

[24] NodeMCU ESP32 Bild.
http://anleitung.joy-it.net/?goods=nodemcu-esp32.
Abgerufen am 30.3.2020.

[25] PlatformIO About us.
https://www.linkedin.com/company/platformio/.
Abgerufen am 3.1.2020.

[26] Programm Header einer Elf binary.
https://assets.linux-audit.com/wp-content/uploads/2015/08/elf-program-headers-segments.png.
Abgerufen am 16.2.2020.

[27] Wie installiert man Nodejs.
https://docs.npmjs.com/downloading-and-installing-node-js-and-npm.
Abgerufen am 2.4.2020.

[28] xTaskCreate Definition.
http://web.ist.utl.pt/~ist11993/FRTOS-API/group___tasks.html.
Abgerufen am 1.4.2020.

Abbildungsverzeichnis

1 Struktur (Quelle: eigene Darstellung) 1
2 Structure (source: own source) 1

1.1 Systemarchitektur (Quelle: Eigene Darstellung) 7

2.1 MQTT QoS 0 [20] 10
2.2 MQTT QoS 1 [20] 10
2.3 MQTT QoS 2 [20] 11
2.4 Docker Virtuelle Maschine (Quelle: eigene Darstellung) 12
2.5 Docker (Quelle: eigene Darstellung) 12
2.6 Docker Compose Beispiel (Quelle: eigene Darstellung) 14
2.7 menuconfig [22] 16
2.8 Technologien Utility lib (Quelle: eigene Darstellung) 20
2.9 Sensoren und Aktoren Utility lib (Quelle: eigene Darstellung) 21

3.1 Bild DHT22 [9] 22
3.2 technische Daten Dht22 [10] 23
3.3 Formfaktor Dht22 [10] 23
3.4 Bild NodeMCU ESP32 [24] 24
3.5 ESP32-WROOM-32 Spezifikationen[17] 25
3.6 OTA Sequenz Timestamp gleich (Quelle: eigene Darstellung) 26
3.7 OTA Sequenz Timestamp anders (Quelle: eigene Darstellung) 27
3.8 OTA Deployment (Quelle: eigene Darstellung) 28
3.9 OTA Partition Table (Quelle: eigene Darstellung) 33
3.10 Wi-Fi Netzwerk Struktur [16] 34
3.11 ESP-Netzwerk Struktur [16] 35
3.12 Node Typen [16] 37
3.13 Bevorzuget Parent Node [16] 38
3.14 Root Node Versagen[16] 39
3.15 Intermediate Parent Node Versagen[16] 39
3.16 Mesh Visualizer (Quelle: eigene Darstellung) 40
3.17 Cytoscape Dagre Beispiel [4] 42
3.18 Cytoscape Euler Beispiel [5] 43
3.19 Cytoscape Spread Beispiel [7] 44

3.20 DHT22 Funktionen (Quelle: eigene Darstellung) 46
3.21 Mqtt Message Klassendiagramm (Quelle: eigene Darstellung) 47
3.22 Mqtt Message (Quelle: eigene Darstellung) 47
3.23 Mqtt Funktionen (Quelle: eigene Darstellung) 48
3.24 HTTP Funktionen(Quelle: eigene Darstellung) 48
3.25 HTTP Client Methoden[21] 49
3.26 Mesh Command Type (Quelle: eigene Darstellung) 50
3.27 Mesh Funktionen (Quelle: eigene Darstellung) 51
3.28 Deployment Diagramm (Quelle: eigene Darstellung) 52
3.29 Sequence Diagramm (Quelle: eigene Darstellung) 53
3.30 Example Mosquitto Start (Quelle: eigene Darstellung) 57
3.31 Example Mesh Visualizer Installation Command (Quelle: eigene Darstellung) 57
3.32 Example Mesh Visualizer Installation Output(Quelle: eigene Darstellung) 58
3.33 Example Mesh Visualizer Start (Quelle: eigene Darstellung) 58
3.34 Example Mesh Visualizer Website (Quelle: eigene Darstellung) 59
3.35 Example Partiton Table (Quelle: eigene Darstellung) 60
3.36 Example Config Übersicht (Quelle: eigene Darstellung) 61
3.37 Example Config Example Configuration (Quelle: eigene Darstellung) . . . 62
3.38 Example Config Serial Flasher Config (Quelle: eigene Darstellung) 63
3.39 Example Config Partition Table (Quelle: eigene Darstellung) 63
3.40 Verbindung ESP32-DHT22 [9, 24] 64
3.41 Beispiel Ergebnis Mesh Visualizer (Quelle: eigene Darstellung) 65
3.42 Beispiel Ergebnis Mqtt (Quelle: eigene Darstellung) 66
3.43 Beispiel Code Struktur Übersicht (Quelle: eigene Darstellung) 66
3.44 Example Code Struktur Main (Quelle: eigene Darstellung) 67
3.45 Example Code Setup (Quelle: eigene Darstellung) 68
3.46 Example Code Setup Wifi (Quelle: eigene Darstellung) 68
3.47 Example Code Event Handler (Quelle: eigene Darstellung) 69
3.48 Example Code Command Callback 1 (Quelle: eigene Darstellung) 70
3.49 Example Code DHT22 Task (Quelle: eigene Darstellung) 70
3.50 Example Code DHT22 Task 1 (Quelle: eigene Darstellung) 71
3.51 Example Code DHT22 Task 2 (Quelle: eigene Darstellung) 71
3.52 Details einer Elf-binary [8] 74
3.53 Programm Header einer Elf binary [26] 77
3.54 Beispiel GNU_STACK [1] 78

Tabellenverzeichnis

1 Tim Untersberger 1
2 Stefan Waldl 2

Protokolle

12. August 2019

Anwesende

Gerald Köck, Stefan Waldl, Tim Untersberger

Ort

Leonding

Inhalt

Folgende Themen wurden angesprochen:

- OTA Versionierung
- Fritzbox Mesh-Netzwerk
- Balancing System für Mesh-Netzwerk

15. Oktober 2019

Anwesende

Gerald Köck, Stefan Waldl, Tim Untersberger

Ort

Leonding

Inhalt

Folgende Themen wurden angesprochen:

- EspWifiManager Implementierung überarbeiten.
- OTA Server mehrere Firmwares
- Neues GitHub Repository für die Diplomarbeit

19. November 2019

Anwesende

Gerald Köck, Stefan Waldl, Tim Untersberger

Ort

Leonding

Inhalt

Folgende Themen wurden angesprochen:

- Wechseln auf die ESP-IDF Toolchain
- Partition Table Probleme
- Monorepo
- OTA Improvements
- ESP Konfiguration automatisieren

21. November 2019

Anwesende

Gerald Köck, Stefan Waldl, Tim Untersberger

Ort

Leonding

Inhalt

Folgende Themen wurden angesprochen:

- OTA Improvements
- File embedding
- Fertiges OTA herzeigen
- Fallback Plan für Abstürze

3. Dezember 2019

Anwesende

Gerald Köck, Stefan Waldl, Tim Untersberger

Ort

Leonding

Inhalt

Folgende Themen wurden angesprochen:

- ESP Startup Optimisierungen
- OTA Web Improvements
- Code Review von OTA Client
- ESP Konfiguration auf OTA Web eventuell auslagern

10. Dezember 2019

Anwesende

Gerald Köck, Stefan Waldl

Ort

Leonding

Inhalt

Folgende Themen wurden angesprochen:

- PHY
- ESP Client Logging
- Partitiontable Offset

14. Jänner 2020

Anwesende

Gerald Köck, Stefan Waldl, Tim Untersberger

Ort

Leonding

Inhalt

Folgende Themen wurden angesprochen:

- Komplett auf ESP-IDF Toolchain umsteigen
- docker-compose files schreiben
- Überlegungen für die Theoretische Arbeit

21. Jänner 2020

Anwesende

Gerald Köck, Tim Untersberger

Ort

Leonding

Inhalt

Folgende Themen wurden angesprochen:

- Dockerfile für das Mesh-Netzwerk verbessern

10. März 2020

Anwesende

Gerald Köck, Tim Untersberger, Stefan Waldl

Ort

Leonding

Inhalt

Folgende Themen wurden angesprochen:

- Dht22 Beispiel
- Mesh Alternativen

12. März 2020

Anwesende

Thomas Stütz, Tim Untersberger, Stefan Waldl

Ort

Leonding

Inhalt

Folgende Themen wurden angesprochen:

- Visualisierung des Mesh-Netzwerks
- Inhaltsverzeichnis finalisieren
- Tutorial für das DHT22 Beispiel

19. März 2020

Anwesende

Thomas Stütz, Tim Untersberger, Stefan Waldl

Ort

Discord

Inhalt

Folgende Themen wurden angesprochen:

- Fertiger Mesh Visualizer
- Alternativen zu Cytoscape besprochen
- Mehr Diagramme für die Arbeit

26. März 2020

Anwesende

Thomas Stütz, Tim Untersberger, Stefan Waldl

Ort

Discord

Inhalt

Folgende Themen wurden angesprochen:

- Verbesserung der Diagramme

Anhang A

Arbeitsaufteilung

Tim Untersberger

- Zusammenfassung
- Autoren der Diplomarbeit
  - Tim Untersberger
- Danksagung
- Verwendete Technologien
  - Message Queuing Telemetry Transport (MQTT)
    - QoS
  - Docker
    - Docker vs. Virtuelle Maschinen
    - Begriffe
  - Docker Compose
    - Docker Compose Beispiel
      - Postgres
      - PgAdmin4
- Ausgewählte Aspekte
  - Mesh Visualizer
    - Cytoscape
      - Dagre

        - Euler
        - Spread
- Libraries
    * Eigene Libraries
        - MQTT
        - Hypertext Transfer Protocol (HTTP)
        - Mesh
- Erläuterung der Verwendung des Mesh-Netzwerkes
    * Übersicht
    * Nodejs Setup
    * ESP-IDF Setup
        - Dependencies
        - Libraries
        - Umgebungsvariablen
        - WSL
    * Source Code
    * Mosquitto
    * Mesh Visualizer
    * Partition Table
    * Config
        - Example Configuration
        - Serial flasher config
        - Partition Table
    * Flashen
    * Ergebnis
        - Mesh Visualizer
        - Mqtt Fx
    * Code
        - Struktur

            - Setup
            - Wifi Setup
            - IP Event Handler
            - Command Callback
            - DHT22 Task
- Resümee
    - Tim Untersberger

Stefan Waldl

- Zusammenfassung
- Autoren der Diplomarbeit
    - Stefan Waldl
- Danksagung
- Ausgangssituation und Zielsetzung
    - Ausgangssituation
    - Beschreibung des Problembereichs
    - Aufgabenstellung
    - Zielbestimmung
- Verwendete Technologien
- ESP-IDF Toolchain
    - KConfig.projbuild
    - Kompilieren
    - Flashen
        * IDF Monitor
    - Platform IO
    - Utility lib Köck
- Ausgewählte Aspekte
    - Verwendete Hardware
        * DHT22

        - OTP Memory
    - NodeMCU ESP32
        - ESP-WROOM-32
- Over The Air Update (OTA)
    - Problemstellung
    - Wie OTA funktioniert
        - Übersicht
    - Partition Table
        - Übersicht
        - Custom Partition Tables
        - Name Feld
        - Type Feld
        - SubType
        - Flags
    - OTA Partition Table
    - ESP-MESH
        - Wi-Fi vs Mesh
- Libraries
    - ESP-IDF Libraries
        - freertos
        - Dht22
- Erläuterung der Verwendung des Mesh-Netzwerkes
    - Hardware
        - DHT22 mit ESP32 verbinden
        - ESP32 mit Computer verbinden
- ELF vs. Bin
    - Bin
        - Hintergrund
    - ELF

        - Struktur
    - ELF-Header
        - Class
        - Data
        - Version
        - OS / ABI
        - ABI-Version
        - Machine
        - Type
    - File Data
        - Programm-Header
        - GNUEHFRAME
        - GNUSTACK
    - Static vs. Dynamic binaries
    - Fazit
- Resümee
  - Stefan Waldl