Hochschule für Technik
und Wirtschaft Berlin
University of Applied Sciences

Entwicklung eines Systems zur Verwaltung und Anzeige von Raumbelegungsplänen an der HTW – Implementierung mittels E-Paper-Displays und ESP32

Bachelorarbeit

Name des Studiengangs

Angewandte Informatik

Fachbereich 4

vorgelegt von
Rico Flemming

Datum:
Berlin, 10.02.2025

Erstgutachter: Prof. Dr. Alexander Huhn
Zweitgutachter: Prof. Dr.-Ing. Thomas Schwotzer

Kurzbeschreibung

Diese Bachelorarbeit behandelt die Entwicklung eines digitalen Systems zur Anzeige von Raumbelegungsplänen an der HTW. Ziel ist es, eine energieeffiziente und wartungsarme Lösung zu schaffen, die Raumpläne automatisiert abruft und auf E-Paper-Displays anzeigt. Der Mikrocontroller ESP32 bildet das Herzstück des Systems, das über Webscraping aktuelle Belegungsdaten erfasst und diese via MQTT an die Anzeigeeinheiten überträgt.
Zusätzlich werden Over-the-Air-Updates (OTA) zur Fernwartung integriert sowie ein WLAN-Fingerprinting-Ansatz getestet, um die Raumzuordnung automatisiert vorzunehmen.
Herausforderungen wie begrenzter Speicher des ESP32, Energieeffizienz und die Genauigkeit des Fingerprinting-Ansatzes wurden analysiert und teilweise optimiert. Die Ergebnisse zeigen eine stabile Kommunikation zwischen Server und ESP32 sowie eine effiziente Nutzung der Batteriekapazität durch den Deep-Sleep-Modus. Die Arbeit bietet eine skalierbare Grundlage für zukünftige Weiterentwicklungen in der digitalen Raumverwaltung.

Inhalt

1. Einführung

1.1 Motivation und Problemstellung

Die Verwaltung und Anzeige von Raumbelegungsplänen stellt in vielen Bildungseinrichtungen eine zentrale Herausforderung dar. Klassische Methoden, wie das Aushängen von gedruckten Plänen, erfordern manuellen Aufwand und sind fehleranfällig. Änderungen im Belegungsplan müssen oft kurzfristig umgesetzt werden, was mit herkömmlichen Mitteln weder flexibel noch effizient möglich ist. Dies kann dazu führen, dass Studierende oder Lehrende nicht rechtzeitig über Raumänderungen informiert werden und es zu organisatorischen Problemen kommt.

Ein digitales Raumbelegungssystem kann diese Herausforderungen lösen, indem es Raumpläne zentral verwaltet und in Echtzeit aktualisiert. Ziel dieses Projekts ist die Entwicklung eines energieeffizienten, IoT-basierten Anzeigesystems, das Raumbelegungsinformationen automatisiert bereitstellt. Hierfür werden stromsparende Mikrocontroller (ESP32) und E-Paper-Displays eingesetzt, um eine flexible Lösung zu realisieren.

Um die Funktionalität und Effizienz dieses Ansatzes zu überprüfen, werden folgende Konzepte getestet:

- Automatisierte Datenbeschaffung durch Webscraping: Die Raumbelegungsdaten sollen direkt von der Universitätsplattform abgerufen und verarbeitet werden.
- Drahtlose Kommunikation über MQTT: Die Verteilung der Daten erfolgt über ein leichtgewichtiges Nachrichtenprotokoll, das speziell für IoT-Systeme optimiert ist.
- Energieeffiziente Anzeige auf E-Paper-Displays: Diese Technologie ermöglicht eine dauerhafte Anzeige ohne kontinuierlichen Stromverbrauch.
- Over-the-Air-Updates (OTA): Die Firmware des ESP32 soll remote aktualisiert werden können, um Wartungsaufwand zu minimieren.
- WLAN-Fingerprinting zur automatisierten Raumzuordnung: Die Geräte sollen sich selbstständig dem richtigen Raum zuweisen können, ohne manuelle Konfiguration.

Durch die Umsetzung dieses Systems soll getestet werden, ob eine zuverlässige und wartungsarme Lösung zur digitalen Raumverwaltung realisierbar ist. Dabei stehen insbesondere die Energieeffizienz, die Stabilität der Kommunikation und die Benutzerfreundlichkeit im Fokus der Untersuchung.

1.2 Zielsetzung

Das Ziel dieser Arbeit ist die Entwicklung eines energieeffizienten, skalierbaren und wartbaren Systems zur digitalen Anzeige von Raumbelegungsplänen. Das System soll Raumpläne aus einer universitären Plattform automatisch abrufen, aufbereiten und auf drahtlosen Displays darstellen. Besonderer Wert wird auf die folgenden Aspekte gelegt:

- **Energieeffizienz:** Einsatz von stromsparenden Technologien wie dem ESP32-Mikrocontroller und E-Paper-Displays.
- **Zentralisierte Steuerung:** Verwaltung und Verteilung der Daten über einen Server, der sowohl die Datenintegration als auch Over-the-Air-Updates (OTA) ermöglicht.
- **Benutzerfreundlichkeit und Wartbarkeit:** Einfache Konfiguration und Updates durch modularen Aufbau und den Einsatz bewährter Technologien wie MQTT und Node-RED.

1.3 Aufbau der Arbeit

Die Arbeit gliedert sich in die folgenden Kapitel:

1. **Grundlagen:** Technologische und methodische Grundlagen, die für das Verständnis des Systems notwendig sind, werden erläutert.
2. **Anforderungserhebung und Analyse:** Funktionale und nicht-funktionale Anforderungen werden analysiert und definiert.
3. **Konzeption und Entwurf:** Beschreibung der Systemarchitektur, des Softwaredesigns und der Datenflüsse.
4. **Implementierung:** Realisierung der Hardware- und Softwarekomponenten sowie der Server- und Client-Funktionalitäten.
5. **Validierung:** Tests und Analysen zur Überprüfung der Funktionalität und Stabilität des Systems.
6. **Abschluss und Fazit:** Zusammenfassung der Ergebnisse, Limitationen und Ausblick auf mögliche Weiterentwicklungen.

2. Grundlagen

2.1 ESP32-Mikrocontroller

Der ESP32 ist ein moderner Mikrocontroller von Espressif Systems, der speziell für Anwendungen im Bereich des Internet of Things (IoT) entwickelt wurde. Er zeichnet sich durch eine Vielzahl von Funktionen aus, die ihn ideal für drahtlose Kommunikationssysteme und ressourcenschonende Anwendungen machen. Der Mikrocontroller basiert auf einer Dual-Core-CPU der Tensilica Xtensa LX6-Architektur, die mit einer Taktrate von bis zu 240 MHz arbeitet. Dies ermöglicht die gleichzeitige Ausführung mehrerer Prozesse, wie z. B. die Verarbeitung von Daten und die Steuerung externer Geräte.

Mit einem integrierten RAM von bis zu 520 KB und der Unterstützung für externen Flash-Speicher von bis zu 16 MB bietet der ESP32 potenziell genügend Speicher für komplexe Aufgaben. Seine integrierten Kommunikationsmodule umfassen sowohl WLAN (802.11 b/g/n) als auch Bluetooth (Classic und Low Energy), wodurch er vielseitig in drahtlosen Netzwerken eingesetzt werden kann. Zudem verfügt der ESP32 über eine Vielzahl von Schnittstellen wie GPIO, SPI, I²C und UART, die die einfache Anbindung externer Komponenten wie Sensoren und Displays ermöglichen.

Besonders hervorzuheben ist die Energieeffizienz des ESP32. Durch seinen Deep-Sleep-Modus kann der Stromverbrauch auf unter 10 µA reduziert werden, was ihn ideal für batteriebetriebene Anwendungen macht. Darüber hinaus verfügt der Mikrocontroller über einen Ultra-Low-Power-Co-Prozessor, der einfache Aufgaben im Energiesparmodus ausführen kann, ohne die Hauptkerne zu aktivieren. Diese Kombination aus hoher Leistung, flexiblen Kommunikationsmöglichkeiten und Energieeffizienz macht den ESP32 zu einem zentralen Bestandteil vieler IoT-Projekte und bildet die Grundlage für das digitale Raumbelegungssystem dieser Arbeit (vgl. Espressif Systems, 2023a).

2.2 E-Paper-Technologie

E-Paper-Displays, auch bekannt als elektronische Papierdisplays, basieren auf elektrophoretischer Technologie, bei der elektrisch geladene Pigmentpartikel in einer Flüssigkeit bewegt werden, um Texte und Bilder darzustellen. Diese Technologie ahmt das Aussehen von Papier nach und bietet eine hervorragende Lesbarkeit, auch bei direkter Sonneneinstrahlung. Im Gegensatz zu LCD- oder OLED-Displays benötigen E-Paper-Displays keine Hintergrundbeleuchtung, was sie Stromsparender macht (vgl. Koubek, 2007: 7-8).

Ein wesentlicher Vorteil von E-Paper-Displays ist ihr niedriger Energieverbrauch. Strom wird nur benötigt, wenn der dargestellte Inhalt aktualisiert wird. Einmal dargestellte Informationen bleiben ohne zusätzliche Energiezufuhr sichtbar. Dies macht E-Paper-Displays ideal für batteriebetriebene Anwendungen, bei denen Energieeffizienz entscheidend ist. Typische Einsatzbereiche sind E-Reader, digitale Preisschilder und IoT-Anzeigen. (vgl. E Ink Corporation, 2024).

Im Rahmen dieser Arbeit wird ein 7,5-Zoll-E-Paper-Display verwendet, das über eine SPI-Schnittstelle mit dem ESP32 verbunden ist. Das Display bietet eine Auflösung von 800 x 480 Pixeln, was eine klare und gut lesbare Darstellung von Raumplänen ermöglicht. Seine Kombination aus hervorragender Lesbarkeit, Energieeffizienz und einfacher Ansteuerung macht es zu einer idealen Wahl für dieses Projekt (vgl. Waveshare, 2024).

2.3 Node-RED

Node-RED ist eine Flow-basierte Entwicklungsumgebung, die speziell für IoT-Anwendungen und die Integration verschiedener Geräte und Dienste entwickelt wurde. Sie bietet eine grafische Benutzeroberfläche, mit der Entwickler komplexe Datenflüsse visuell modellieren und steuern können. Dies erleichtert nicht nur die Entwicklung, sondern auch die Wartung und Anpassung von Systemen.

In diesem Projekt wird Node-RED eingesetzt, um die Over-the-Air-Updates (OTA) und die Koordination der MQTT-Kommunikation zu verwalten. Node-RED empfängt die Nachrichten vom MQTT-Broker, verarbeitet sie und leitet sie an die entsprechenden ESP32-Geräte weiter. Zusätzlich können durch Node-RED Überwachungsfunktionen implementiert werden, die den Status des Systems in Echtzeit anzeigen. Diese Kombination aus Flexibilität und Benutzerfreundlichkeit macht Node-RED zu einem unverzichtbaren Werkzeug für die Serverimplementierung (vgl. OpenJS Foundation, 2024).

2.4 Python

Python ist eine universelle Programmiersprache, die sich durch ihre einfache Syntax, ihre hohe Lesbarkeit und ihre Vielseitigkeit auszeichnet. Sie wurde in den frühen 1990er Jahren von Guido van Rossum entwickelt und hat sich seitdem zu einer der beliebtesten Programmiersprachen weltweit entwickelt. Python wird in einer Vielzahl von Bereichen eingesetzt, darunter Webentwicklung, Datenanalyse und IoT-Anwendungen (vgl. Lutz, 2009: 9-10).

Ein entscheidender Vorteil von Python ist die umfangreiche Standardbibliothek, die Module für nahezu jeden Anwendungsfall bietet. So können Entwickler problemlos auf Funktionen zur Dateiverwaltung, Netzwerkkommunikation oder Datenverarbeitung zugreifen, ohne zusätzliche Bibliotheken installieren zu müssen. Darüber hinaus ist Python plattformunabhängig, wodurch der gleiche Code auf verschiedenen Betriebssystemen wie Windows, macOS oder Linux ausgeführt werden kann (vgl. Lutz, 2009: 3-4).

Neben der Standardbibliothek bietet Python ein großes Ökosystem an Drittanbieter-Bibliotheken, die die Entwicklung spezialisierter Anwendungen erleichtern. Für dieses Projekt sind insbesondere zwei Bibliotheken von zentraler Bedeutung: BeautifulSoup und paho-mqtt. BeautifulSoup ermöglicht das Webscraping und wird genutzt, um Raumbelegungsdaten von der Universitätsplattform zu extrahieren (vgl. Richardson, 2024). Die paho-mqtt-Bibliothek implementiert das MQTT-Protokoll und sorgt für eine effiziente Kommunikation zwischen dem Server und den ESP32-Geräten (vgl. Eclipse Foundation, 2024).

Die Integration von Python in diesem Projekt erfolgt auf Serverseite. Hier wird Python verwendet, um die extrahierten Raumbelegungsdaten zu verarbeiten, in ein maschinenlesbares Format wie JSON zu konvertieren und über den MQTT-Broker an die ESP32-Geräte zu senden. Dank seiner einfachen Syntax und der leistungsstarken Bibliotheken ist Python besonders geeignet, um schnell und effizient serverseitige Prozesse zu implementieren. In Kombination mit der Flexibilität und der Unterstützung für moderne Technologien wie MQTT trägt Python maßgeblich zur Funktionalität und Effizienz des Systems bei (vgl. Lutz, 2009: 17).

2.5 MicroPython

MicroPython ist eine speziell für Mikrocontroller entwickelte Implementierung der Programmiersprache Python. Ihr Ziel ist es, die bekannte und leicht verständliche Syntax von Python auf Geräten mit begrenzten Ressourcen verfügbar zu machen. Mit MicroPython können Entwickler schnell und effizient Anwendungen für Mikrocontroller schreiben, ohne auf die Verwendung einer Low-Level-Programmiersprache wie C angewiesen zu sein.

Ein zentrales Merkmal von MicroPython ist seine Ressourcenschonung. Es wurde speziell darauf ausgelegt, mit wenig Speicherplatz und begrenzten Rechenressourcen zu arbeiten. So benötigt es lediglich etwa 256 KB Flash und 16 KB RAM, was es ideal für Mikrocontroller wie den ESP32 macht. Gleichzeitig bietet MicroPython eine abgespeckte Version der Python-Standardbibliothek, die jedoch für die meisten Anwendungen im Mikrocontroller-Bereich ausreichend ist. Neben allgemeinen Modulen wie „math“ oder „json“ stellt MicroPython hardwarebezogene Module wie

„machine“ bereit, die GPIOs, SPI, I²C und andere Schnittstellen ansteuern können (vgl. George, 2023).

Ein großer Vorteil von MicroPython ist der interaktive REPL-Modus (Read-Eval-Print-Loop). Über diesen können Entwickler direkt auf den Mikrocontroller zugreifen, Befehle ausführen und die Ergebnisse in Echtzeit überprüfen. Dies beschleunigt den Entwicklungsprozess erheblich, da Änderungen schnell getestet werden können, ohne die Firmware jedes Mal komplett neu flashen zu müssen. Darüber hinaus unterstützt MicroPython auch die Energiesparmodi moderner Mikrocontroller. Beim ESP32 ermöglicht dies die einfache Nutzung des Deep-Sleep-Modus, wodurch der Stromverbrauch reduziert und die Batterielaufzeit verlängert werden kann (vgl. George, 2023).

Im Rahmen dieses Raumbelegungssystems wird MicroPython verwendet, um die Firmware des ESP32 zu realisieren. Es steuert dabei die Kommunikation mit dem Server über MQTT, die Verarbeitung der empfangenen Daten und die Ansteuerung des E-Paper-Displays über die SPI-Schnittstelle. Die einfache Syntax von MicroPython erleichtert die Entwicklung und macht es möglich, komplexe Funktionen wie die parallele Ausführung von Aufgaben mit `uasyncio` effizient umzusetzen. Aufgrund seiner Ressourcenoptimierung und der Unterstützung moderner Hardwarefunktionen ist MicroPython die ideale Wahl für dieses Projekt (vgl. George, 2023).

2.6 BeautifulSoup

BeautifulSoup ist eine in Python entwickelte Bibliothek, die speziell für das Extrahieren und Verarbeiten von Daten aus HTML- und XML-Dokumenten entwickelt wurde. Sie ermöglicht es, komplexe Webseiten einfach zu analysieren und relevante Informationen gezielt auszulesen. Die Bibliothek bietet eine intuitive API, mit der sich DOM-Elemente wie Tags, Attribute und Textinhalte durchsuchen und manipulieren lassen (vgl. Richardson, 2024).

Im Rahmen dieser Arbeit wird BeautifulSoup verwendet, um Raumbelegungsdaten von der universitären Plattform zu extrahieren. Diese Daten liegen meist in unstrukturierter Form vor und müssen vor ihrer weiteren Verarbeitung bereinigt und in ein maschinenlesbares Format wie JSON umgewandelt werden. BeautifulSoup übernimmt dabei die Aufgabe, die HTML-Struktur der Webseite zu analysieren, relevante Inhalte zu extrahieren und diese für den Server bereitzustellen. Durch diesen automatisierten Ansatz wird der Prozess der Datenerhebung erheblich beschleunigt und fehleranfällige manuelle Eingriffe vermieden (vgl. Richardson, 2024).

2.7 MQTT-Kommunikation

Das Message Queuing Telemetry Transport (MQTT)-Protokoll ist ein leichtgewichtiges Kommunikationsprotokoll, das speziell für IoT-Anwendungen mit begrenzten Ressourcen entwickelt wurde. MQTT arbeitet nach dem Publish-Subscribe-Modell, bei dem ein zentraler Broker Nachrichten zwischen Publishern und Subscribern vermittelt. Diese Architektur ermöglicht eine flexible und skalierbare Kommunikation zwischen zahlreichen Geräten (Usmani, 2021:1-2).

Die Hauptvorteile von MQTT liegen in seiner Effizienz und Zuverlässigkeit. Es minimiert den Overhead bei der Datenübertragung, was besonders in Netzwerken mit begrenzter Bandbreite von Vorteil ist. Außerdem bietet das Protokoll verschiedene QoS-Level (Quality of Service), die sicherstellen, dass Nachrichten zuverlässig und ohne Datenverlust übertragen werden. Dies macht MQTT besonders geeignet für Anwendungen, bei denen Echtzeitkommunikation und hohe Zuverlässigkeit erforderlich sind (Usmani, 2021:2).

In diesem Projekt wird MQTT verwendet, um die Kommunikation zwischen dem Server und den ESP32-Geräten zu realisieren. Der Server fungiert als Publisher, der Raumbelegungsdaten und Over-the-Air-Updates bereitstellt. Die ESP32-Geräte abonnieren die entsprechenden Topics und empfangen die Nachrichten. Dieser Ansatz ermöglicht eine effiziente und zentralisierte Datenverwaltung, die sowohl die Aktualität der Informationen als auch die Skalierbarkeit des Systems gewährleistet.

2.8 Over-the-Air-Update (OTA)

Over-the-Air-Updates (OTA) sind eine essenzielle Technologie in der modernen IoT-Entwicklung, die es ermöglicht, Firmware und Software drahtlos zu aktualisieren, ohne dass physischer Zugriff auf das Gerät erforderlich ist. Dies reduziert Wartungskosten, erleichtert die Bereitstellung neuer Funktionen und verbessert die Sicherheit durch zeitnahe Updates. OTA ist besonders in verteilten Netzwerken mit vielen Endgeräten von Vorteil, da es eine skalierbare Lösung zur Aktualisierung von Firmware über WLAN oder Mobilfunknetze bietet (vgl. Fasolo, E., 2022).

Funktionsweise von OTA-Updates

OTA-Updates basieren auf einem Client-Server-Modell, bei dem das Gerät regelmäßig mit einem Server kommuniziert, um zu prüfen, ob neue Firmware-Versionen verfügbar sind. Der Update-Prozess läuft typischerweise in mehreren Schritten ab:

1. **Versionsprüfung:** Das Gerät sendet eine Anfrage an den Server, um zu prüfen, ob eine neue Firmware-Version existiert.
2. **Herunterladen der neuen Firmware:** Falls ein Update verfügbar ist, wird die neue Version in einen separaten Speicherbereich geladen.
3. **Validierung der Firmware:** Das Gerät überprüft, ob die Firmware vollständig und fehlerfrei heruntergeladen wurde (z. B. durch Hash- oder Checksum-Prüfung).
4. **Installation und Neustart:** Nach erfolgreicher Validierung wird die neue Firmware aktiviert und das Gerät neu gestartet.

Um sicherzustellen, dass ein fehlerhaftes Update nicht zu einem irreparablen Zustand führt, nutzen viele OTA-Systeme einen **Fallback-Mechanismus**, bei dem das Gerät auf die vorherige Version zurückkehrt, falls die neue Firmware nicht korrekt funktioniert (vgl. Espressif Systems, 2023b).

3. Anforderungserhebung und Analyse

Die Entwicklung eines digitalen Raumbelegungssystems erfordert eine sorgfältige Erhebung der Anforderungen, um sicherzustellen, dass das System die gewünschten Funktionen erfüllt und gleichzeitig robust, effizient und benutzerfreundlich ist. Die Anforderungen lassen sich in funktionale und nicht-funktionale Kategorien unterteilen, wobei die funktionalen Anforderungen die unmittelbaren Aufgaben des Systems beschreiben und die nicht-funktionalen Anforderungen die Qualitätsmerkmale und Rahmenbedingungen festlegen.

3.1 Funktionale Anforderungen

Die funktionalen Anforderungen beschreiben die zentralen Aufgaben des Systems und legen fest, welche Funktionen es ausführen muss, um seinen Zweck zu erfüllen. Im Kontext des digitalen Raumbelegungssystems sind dies:

1. **Anzeige von Raumbelegungsplänen:**
   Das System soll die aktuellen Belegungspläne eines Raums auf einem E-Paper-Display anzeigen. Die Anzeige soll eine tabellarische Struktur mit Wochentagen und Zeitblöcken enthalten, die den Belegungsstatus für jede Stunde darstellt.

2. **Automatische Datenaktualisierung:**
   Das System soll automatisch die aktuellen Raumbelegungsdaten von einer zentralen Serverplattform abrufen. Änderungen, die auf der Universitätsplattform vorgenommen werden, sollen zeitnah auf dem E-Paper-Display sichtbar sein.

3. **Datenabruf über MQTT:**
   Die Kommunikation zwischen Server und ESP32-Geräten soll über das MQTT-Protokoll erfolgen. Dabei sendet der ESP32 eine Anfrage mit dem Raumnamen und empfängt die entsprechenden Daten des Servers.

4. **Over-the-Air-Updates (OTA):**
   Das System soll es ermöglichen, die Firmware der ESP32-Geräte drahtlos zu aktualisieren. Dies reduziert den Wartungsaufwand und erlaubt es, neue Funktionen oder Fehlerbehebungen schnell bereitzustellen.

5. **Energieoptimierung:**
   Das System soll inaktive Perioden erkennen und den ESP32 in den Deep-Sleep-Modus

versetzen, um Strom zu sparen. Das Display bleibt währenddessen unverändert, da es keinen Strom für die Anzeige benötigt.

6. **Benutzerkonfiguration:**
   Die ESP32-Geräte sollen einfach konfigurierbar sein, z. B. durch die Eingabe von WLAN-Zugangsdaten oder das Orten des Raumes über WLAN-Fingerprinting und damit das Erkennen des Raumnamens. Diese Konfiguration soll zentral gesteuert werden können, um die Einrichtung mehrerer Geräte zu erleichtern.

3.2 Nicht-funktionale Anforderungen

Neben den unmittelbaren Funktionen des Systems sind auch nicht-funktionale Anforderungen entscheidend, da sie die Qualität und Effizienz des Systems bestimmen. Die wichtigsten nicht-funktionalen Anforderungen sind:

1. **Energieeffizienz:**
   Das System soll so konzipiert sein, dass es mit minimalem Energieverbrauch betrieben werden kann. Dies umfasst den Einsatz von Deep-Sleep-Modi des ESP32, die Verwendung eines E-Paper-Displays mit geringem Strombedarf und optimierte Kommunikationszyklen.

2. **Skalierbarkeit:**
   Das System soll erweiterbar sein, sodass mehrere ESP32-Geräte in unterschiedlichen Räumen gleichzeitig betrieben werden können. Der Server und die Kommunikationsstruktur sollen problemlos mit einer wachsenden Anzahl von Geräten umgehen können.

3. **Zuverlässigkeit:**
   Die Anzeige der Raumbelegungspläne soll stets korrekt und aktuell sein. Kommunikationsunterbrechungen oder Serverfehler dürfen keine dauerhaften Inkonsistenzen verursachen. Mechanismen zur Wiederherstellung, wie automatische Neustarts des ESP32 oder Wiederholungsversuche bei fehlgeschlagener Datenübertragung, sind erforderlich.

4. **Benutzerfreundlichkeit:**
   Die Einrichtung und Bedienung des Systems soll einfach sein. Endbenutzer sollen keine tiefgehenden technischen Kenntnisse benötigen, um das System zu nutzen. Ein intuitives Konfigurationsinterface sowie klar strukturierte Anzeigen auf dem Display sind essenziell.

5. **Wartbarkeit:**
   Die Software des Systems soll modular aufgebaut sein, um zukünftige Erweiterungen oder

Fehlerbehebungen zu erleichtern. OTA-Updates und zentrale Konfigurationsmöglichkeiten tragen zur einfachen Wartung bei.

6. **Robustheit:**

   Das System soll auch unter schwierigen Bedingungen, wie instabilen WLAN-Verbindungen oder kurzzeitigen Serverausfällen, zuverlässig arbeiten. Es sollen Mechanismen integriert werden, um die Funktionalität in solchen Szenarien zu gewährleisten.

7. **Anpassbarkeit:**

   Das System soll so flexibel sein, dass es leicht an spezifische Anforderungen oder neue Datenquellen angepasst werden kann. Dies schließt die Unterstützung für alternative Plattformen oder unterschiedliche Displaygrößen ein.

4. Konzeption und Entwurf

Das System wurde so konzipiert, dass es die Anforderungen an Energieeffizienz, Skalierbarkeit und Benutzerfreundlichkeit erfüllt. Die Kombination aus leistungsfähiger Hardware, modularer Firmware und einem zentralen Server bildet die Grundlage für ein robustes und flexibles digitales Raumbelegungssystem. In diesem Kapitel werden die Systemarchitektur, das Softwaredesign, das Serverkonzept und der Datenfluss näher beschrieben.

4.1 Systemarchitektur

Die Hardware- und Softwarestruktur des Systems wurde für eine modulare Arbeitsweise entwickelt, um die verschiedenen Komponenten optimal miteinander zu verknüpfen. Die Hauptbestandteile der Systemarchitektur sind die ESP32-Geräte mit ihrer Firmware und der zentrale Server, der die Daten bereitstellt und verwaltet.

4.1.1 Hardwarestruktur

1. **ESP32 Driver Board von Waveshare:**
   Der ESP32 Mikrocontroller steuert die gesamte Funktionalität des Systems, darunter die Kommunikation mit dem Server, die Datenverarbeitung und die Steuerung des E-Paper-Displays. Das Waveshare Driver Board erleichtert die Integration des ESP32 mit dem E-Paper-Display durch optimierte SPI-Schnittstellen.

2. **E-Paper-Display:**
   Das 7,5-Zoll-E-Paper-Display wird über SPI angesteuert und dient zur dauerhaften Anzeige der Raumbelegungspläne. Es benötigt nur Energie während der Aktualisierung, wodurch eine hohe Energieeffizienz erreicht wird.

3. **TP4056 Lithium-Ionen-Ladecontroller:**
   Der TP4056-Charge-Controller stellt die optimale Energieverwaltung sicher. Er ermöglicht das Laden der 4000-mAh-Lithium-Ionen-Batterie über einen Micro-USB-Anschluss und schützt diese vor Überladung und Tiefentladung.

4. **Zentrale Energieversorgung:**
   Die Lithium-Ionen-Batterie versorgt das gesamte System. Die Kombination aus den stromsparenden Eigenschaften des ESP32 und des E-Paper-Displays gewährleistet eine lange Betriebsdauer.

4.1.2 Softwarestruktur

Die Firmware des ESP32 wird in MicroPython entwickelt und soll modular aufgebaut sein. Sie umfasst Funktionen zur WLAN-Verbindung, Datenverarbeitung und Darstellung sowie Mechanismen für Over-the-Air-Updates (OTA). Ebenfalls wird noch eine Schnittstelle zum WLAN-Fingerprinting zur Standortbestimmung erstellt. Die Softwarestruktur ist darauf ausgelegt, eine hohe Stabilität und parallele Verarbeitung sicherzustellen.

Der zentrale Server ist für das Webscraping, die Datenaufbereitung und die OTA-Verwaltung zuständig. Dies wird durch den Einsatz von Python, BeautifulSoup und Node-RED realisiert. Ebenfalls läuft auf ihm die WLAN-Fingerprinting API und Datenbank.

4.2 Softwaredesign

Die Firmware des ESP32 ist modular aufgebaut, was die Wartung und Erweiterbarkeit des Systems erleichtert. Die wichtigsten Module und Konzepte sind:

1. **Modularer Aufbau:**
    - **WLAN-Verbindung:** Das Modul wifi_manager.py verbindet den ESP32 mit dem WLAN und stellt sicher, dass Verbindungsabbrüche automatisch behoben werden.
    - **Datenverarbeitung:** Das Modul file_helper.py speichert empfangene Daten in JSON-Dateien, die von anderen Modulen verarbeitet werden können.
    - **Anzeige:** Das Modul view.py sorgt für die Darstellung der Daten auf dem E-Paper-Display in Form einer übersichtlichen Tabelle.
    - **OTA-Updates:** Das Modul boot.py überwacht eingehende MQTT-Nachrichten und ermöglicht Firmware-Updates, ohne das Gerät physisch zu manipulieren.
    - **WLAN-Fingerprinting:** Das Modul wifi_localization.py scannt die Umgebung nach WLAN-Signalen, sendet die gesammelten Daten an den Server und erhält als Antwort die vermutete Raumnummer.
2. **Einsatz von „uasyncio“**:
   Um die parallele Verarbeitung von Aufgaben zu ermöglichen, wird die Event-basierte Programmierung mit „uasyncio“ genutzt.
    - **Hauptsteuerung:** Das Modul main.py verarbeitet die empfangenen Raumbelegungsdaten und aktualisiert das Display.

- **OTA-Mechanismus:** Ein separater Task überwacht kontinuierlich eingehende MQTT-Nachrichten und verarbeitet OTA-Updates.
  Durch den nicht-blockierenden Event-Loop bleiben alle Systemfunktionen reaktionsfähig, und die Aufgaben können gleichzeitig ausgeführt werden.

4.3 Serverkonzept

Der Server spielt eine zentrale Rolle im System, da er die Datenverarbeitung und die Verteilung der Informationen an die ESP32-Geräte übernimmt. Er kombiniert mehrere Technologien und Werkzeuge, um diese Aufgaben effizient zu erfüllen:

1. **Webscraping mit BeautifulSoup:**
   Ein Python-Skript verwendet BeautifulSoup, um die Raumbelegungsdaten von der Universitätsplattform zu extrahieren. Dabei werden die HTML-Inhalte analysiert, relevante Informationen gefiltert und in einer strukturierten JSON-Datei gespeichert. Diese Daten enthalten die Belegungszeiten, Räume und Veranstaltungsarten.

2. **Node-RED für OTA-Updates:**
   Auf dem Server läuft eine Node-RED-Instanz, die MQTT-Nachrichten verwaltet und OTA-Updates an die ESP32-Geräte ermöglicht. Firmware-Dateien können über Node-RED an den ESP32 gesendet und automatisch installiert werden, wodurch eine zentrale Wartung des Systems möglich ist.

3. **MQTT-Kommunikation:**
   Der Server agiert als Publisher im MQTT-Netzwerk. Er sendet die JSON-Daten mit den Raumbelegungsplänen an die ESP32-Geräte und verarbeitet gleichzeitig Anfragen, die von den Geräten initiiert werden.

4. **WLAN-Fingerprinting zur Raumzuordnung:**
   Der Server verarbeitet die WLAN-Daten, die von den ESP32-Geräten gesendet werden. Diese werden per HTTP Post Request an die API eines bestehenden Projekts gesendet und die WLAN-Fingerprinting API schickt dann eine „response“ mit der vermuteten Raumnummer zurück (vgl. Völkers, 2024).

4.4 Datenfluss

Der Datenfluss beschreibt, wie Informationen von der Universitätsplattform zum E-Paper-Display gelangen und welche Schritte dabei durchlaufen werden:

1. **Standortbestimmung mit WLAN-Fingerprinting (ESP32)**
   Der ESP32 scannt nach verfügbaren WLAN-Netzwerken und erfasst SSID(Netzwerknamen), BSSID (MAC-Adressen der Access Points), Signalstärken (RSSI-Werte). Diese Daten werden an den Server per HTTP-Request gesendet. Die Raumzuordnung wird dem ESP32 zurückgesendet und in „config['room_name']“ gespeichert. Dadurch weiß das Gerät automatisch, zu welchem Raum es gehört, ohne manuelle Konfiguration.

2. **ESP32 fordert Raumbelegungsdaten an (MQTT-Request)**
   Der ESP32 sendet über MQTT eine Anfrage an den Server, um die Raumbelegungsdaten für seinen Standort zu erhalten. Diese Anfrage wird über das Topic „esp32/request_room_schedule“ gesendet. Der Server empfängt die Anfrage und startet die Datenverarbeitung.

3. **Webscraping:**
   Der Server ruft die aktuellen Raumbelegungsdaten von der Universitätsplattform ab. Diese werden mithilfe von BeautifulSoup analysiert und als JSON-Datei gespeichert.

4. **Datenverteilung:**
   Über MQTT sendet der Server die JSON-Daten an die ESP32-Geräte. Jede Nachricht enthält spezifische Informationen für den angefragten Raum.

5. **Datenverarbeitung:**
   Die empfangenen JSON-Daten werden vom ESP32 gespeichert und von view.py gelesen. Das Modul formatiert die Daten und bereitet sie für die Anzeige auf dem E-Paper-Display vor.

6. **Anzeige:**
   Die Raumbelegungspläne werden auf dem E-Paper-Display dargestellt. Dabei werden Wochentage, Zeitblöcke und Belegungsstatus in einer tabellarischen Ansicht angezeigt.

7. **Hintergrundprozesse:**
   Parallel zur Anzeige laufen Prozesse zur Statusüberwachung, die den Verbindungsstatus und den freien Speicherplatz prüfen. Gleichzeitig wird nach OTA-Updates geprüft, die bei Bedarf automatisch durchgeführt werden.

5. Implementierung

5.1 Hardware-Setup

Für den Aufbau des Systems wurden die in der Systemarchitektur beschriebenen Hardwarekomponenten gemäß den dort definierten Anforderungen miteinander verbunden. Die Hardware wurde zwischen 2 Plastikscheiben montiert, die dem Design der aktuellen Raumschilder der HTW gleichen, um Stabilität, Zugänglichkeit und Kompatibilität zu gewährleisten.

Der ESP32 ist bereits auf dem Waveshare Driver Board installiert und über die SPI-Schnittstelle mit dem 7,5-Zoll-E-Paper-Display verbunden. Die Verbindung der GPIO-Pins erfolgte nach den Vorgaben des Treibers. Zusätzlich wurde der TP4056-Ladecontroller integriert, der die Energieversorgung der Hardware durch eine 4000-mAh-Lithium-Ionen-Batterie sicherstellt. Alle Verbindungen wurden mit Steckverbindern realisiert, um Wartungsfreundlichkeit zu gewährleisten.

Vor der Softwareimplementierung wurde die Hardware getestet:

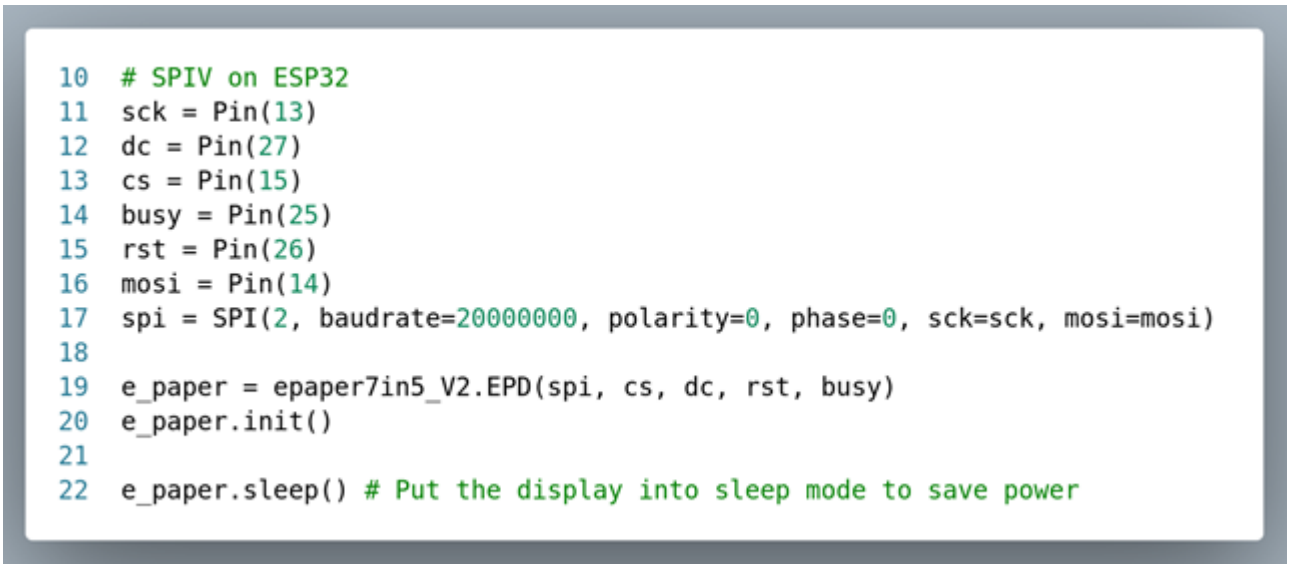
1. E-Paper-Display: Es wurde geprüft, ob das Display korrekt initialisiert werden kann und Inhalte zuverlässig anzeigt, dafür gab es bereits Testsoftware auf dem ESP32, sowie einen angepasster Treiber für das E-Paper-Display.
2. Kommunikation: Die SPI-Kommunikation zwischen dem ESP32 und dem Display sowie WLAN wurden getestet, um sicherzustellen, dass die Datenübertragung stabil ist.
3. Stromversorgung: Der Ladecontroller und die Batterie wurden auf ordnungsgemäße Funktion überprüft, durch das Laden des Akkus über den Ladecontroller, sowie das Einschalten über den Ein- und Ausschalter.

Mit diesen Tests wurde sichergestellt, dass die Hardware korrekt arbeitet und die Anforderungen der Software erfüllt.

5.2 Firmware-Programmierung

Die Firmware des ESP32 wurde in MicroPython entwickelt und modular aufgebaut, um die unterschiedlichen Funktionen des Systems effizient zu trennen. Ziel der Firmware ist es, die Hardware zu initialisieren, Daten vom Server abzurufen, diese zu verarbeiten und auf dem E-Paper-Display anzuzeigen sowie Over-the-Air-Updates (OTA) zu ermöglichen. Der Einsatz von „uasyncio“ erlaubt eine parallele und nicht-blockierende Verarbeitung aller Aufgaben. Im Folgenden werden die wichtigsten Module und Funktionen detailliert beschrieben.

5.2.1 Initialisierung der Hardware



```
10  # SPIV on ESP32
11  sck = Pin(13)
12  dc = Pin(27)
13  cs = Pin(15)
14  busy = Pin(25)
15  rst = Pin(26)
16  mosi = Pin(14)
17  spi = SPI(2, baudrate=20000000, polarity=0, phase=0, sck=sck, mosi=mosi)
18
19  e_paper = epaper7in5_V2.EPD(spi, cs, dc, rst, busy)
20  e_paper.init()
21
22  e_paper.sleep() # Put the display into sleep mode to save power
```

Abbildung 1 view.py Hardware Konfiguration

In Abbildung 1 wird die SPI-Schnittstelle definiert, die zur Kommunikation mit dem Display notwendig ist. Dies umfasst die Pins für den Takt (SCK) die Datenleitung (MOSI), den Chip-Select (CS), den Daten/Steuerungs-Pin (DC), den Reset-Pin (RST) sowie den Busy-Pin. Die SPI-Verbindung wird mit einer Baudrate von 20Mhz eingerichtet. Mit „e_paper.init()“ wird der Display initialisiert, damit dieser in keinem undefinierten Zustand mehr ist und danach mit „e_paper.sleep()“ in den Stromsparmodus versetzt.

5.2.2 Over-the-Air-Update (OTA)

Die OTA-Funktionalität wird über ein Projekt von Till Giesa realisiert, wobei noch Anpassungen erfolgen für dieses Projekt. Ohne Änderungen der Funktionalität übernommen wurden dabei folgende Dateien: config.py, simply.py, file_helper.py, status_helper.py und wifi_manger.py. Diese werden deshalb nur kurz erläutert, jedoch keine genaue Codebetrachtung durchgeführt.

Bei der config.py handelt es sich um die Konfigurationsdaten für den ESP32. Darin enthalten sind eine CLIENT_ID, ein Alias für das Gerät, SSID und Passwort für das WLAN, die Host Adresse und Port für den MQTT-Server, der MQTT-User und Passwort, sowie neu hinzugefügt der „ROOM_NAME“.

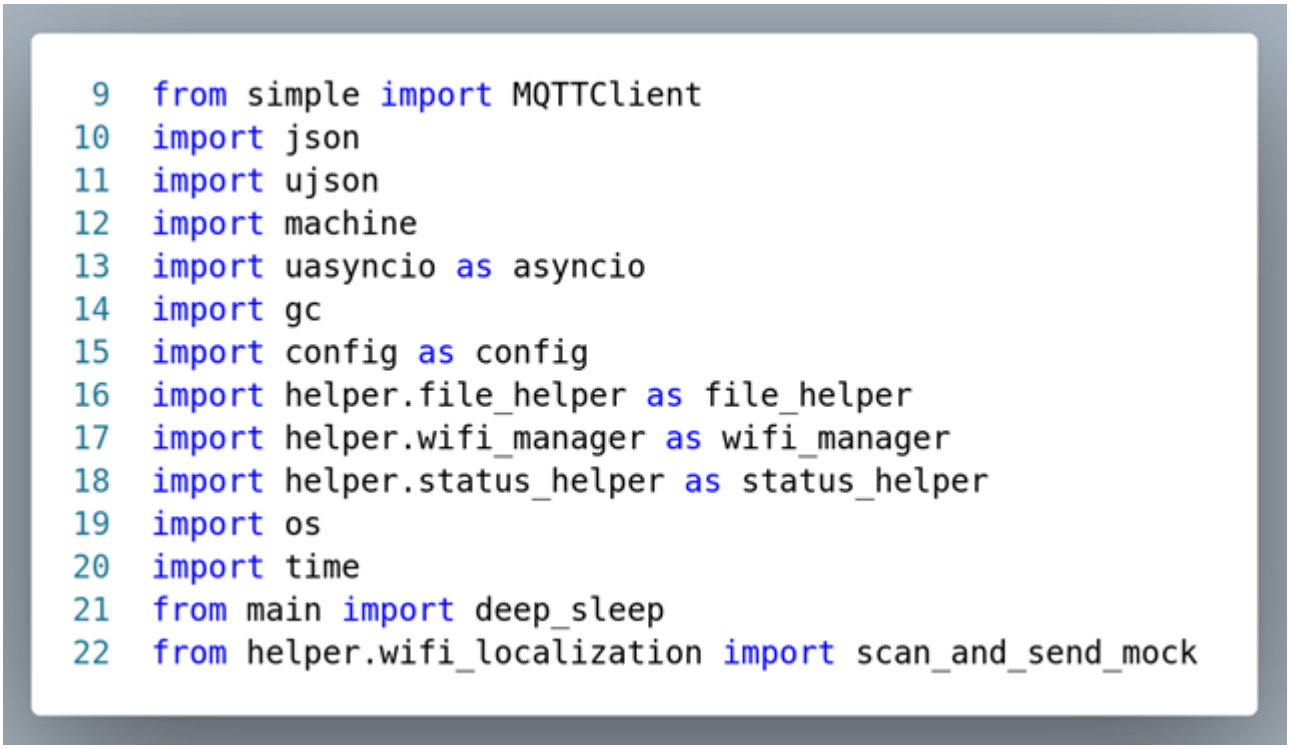
Bei der simply.py handelt es sich um eine minimalistische und ressourcenschonende MQTT-Bibliothek für MicroPython. Sie bietet dem ESP32 damit die Funktionen sich zum MQTT-Broker zu verbinden, Nachrichten zu veröffentlichen (Publish) und sich auf Themen zu abonnieren (Subscribe) und eingehende Nachrichten zu verarbeiten.

Die file_helper.py enthält Hilfsmethoden zum Managen von Dateien und die status_helper.py ist für die Verwaltung von Status-Updates des ESP32 über MQTT zuständig.

Der wifi_manager.py stellt die Funktionalität zur WLAN-Verbindung bereit, dabei werden die fehlende Eingabe von WLAN SSID und Password abgefangen, sowie bis zu 120 Versuche sich mit dem WLAN zu verbinden.

5.2.2.1 Boot.py

Als nächstes wird die boot.py beschrieben, diese steuert die komplette OTA-Funktionalität und ruft die main.py für weitere eigene Programmabläufe aus. Sie kümmert sich dabei um die Initialisierung des WLAN, die Verbindung zum MQTT-Broker, die Verwaltung von Nachrichten über MQTT und das Starten der main.py.



```
 9  from simple import MQTTClient
10  import json
11  import ujson
12  import machine
13  import uasyncio as asyncio
14  import gc
15  import config as config
16  import helper.file_helper as file_helper
17  import helper.wifi_manager as wifi_manager
18  import helper.status_helper as status_helper
19  import os
20  import time
21  from main import deep_sleep
22  from helper.wifi_localization import scan_and_send_mock
```

Abbildung 2 boot.py import

In der Abb. 2 sind die Imports der boot.py zusehen. Ein Großteil wurde bereits zuvor beschrieben, weswegen nur die fehlenden und neu hinzugefügten Imports hier erwähnt werden. Als erstes haben wir die „json“ und „ujson“, beide werden für das Parsen und Serialisieren von JSON-Nachrichten verwendet. „machine“ ist für die Steuerung der Hardware-Funktion nötig. In diesem Fall für den Reset des ESP32. Für das asynchrone Ausführen von Programmcode wird „uasyncio“ verwendet. Bei „gc“ handelt es sich um den Garbage Collector, dieser wird für die Kontrolle des Speicherverbrauchs benötigt. „os“ ist für die Überprüfung von vorhanden sein der Dateien nötig und „time“ zur Zeitmessung. „deep_sleep wird verwendet, um den ESP32 nach einer bestimmten Zeit oder bei Fehlern in den Deep-Sleep-Modus zu versetzen, um Energie zu sparen. „scan_and_send_mock“ ermöglicht es, WLAN-Netzwerke zu scannen und die erfassten Daten an den Server zu senden, um den aktuellen Raum zu bestimmen. Dies ist aktuell eine Mock Variante, eine funktionierende befindet sich auch im „helper.wifi_localization“, mehr dazu später im Kapitel aufgetretene Probleme.

```
18  run_main = True
19
20  try:
21      import main
22
23  except ImportError or ModuleNotFoundError:
24      run_main = False
25  gc.enable()
26  global mqtt_client
27  last_message_time = None
28  LOG_FILE = "error_log.txt"
29  MAX_LOG_ENTRIES = 31
30  MAX_RETRIES = 3  # Maximum number of attempts for scan_and_send
```

Abbildung 3 boot.py Globale Variablen

In Abb. 3 sind die globalen Variablen zu sehen. Mit „run_main“ wird festgehalten, ob die main.py ausgeführt werden soll. Sollte es beim Import zu Fehlern kommen wird damit verhindert, dass der ESP32 abstürzt und es können noch OTA-Updates erfolgen. „mqtt_client“ dient zur Verwaltung der MQTT-Client-Verbindung (vgl. Giesa, 2024).

Neu hinzugefügt für das Projekt ist „last_message_time“. In dieser Variable wird die Zeit der letzten MQTT-Nachricht gespeichert. „LOG_FILE“ speichert den Pfad zur Log-Datei, in der Fehlermeldungen mit Zeitstempel protokolliert werden. “MAX_LOG_ENTRIES“ definiert die maximale Anzahl an Einträgen in der Fehler-Log-Datei und „MAX_RETRIES“ gibt an, wie oft der ESP32 versuchen soll, eine WLAN-Fingerprinting-Anfrage durchzuführen, bevor er aufgibt.

```
32  # ------------ Begin config ---------------#
33  config = {
34      "ssid": config.WIFI_SSID,
35      "wifi_pw": config.WIFI_PASSWD,
36      "host": config.HOST,
37      "host_port": config.HOST_PORT,
38      "client_id": config.CLIENT_ID,
39      "alias": config.ALIAS,
40      "room_name": "" ,
41      "topic_sub_all": "update/all",
42      "topic_sub_device": "update/"+str(config.CLIENT_ID),
43      "topic_publish_log": "log",
44      "topic_table_display_answer": "esp32/schedule/" + config.ROOM_NAME,
45      "topic_table_display_request": "esp32/request_room_schedule",
46      "mqtt_user": config.MQTT_USER,
47      "mqtt_pw": config.MQTT_PW,
48      "api_route": "http://141.45.212.246:8000/measurements/predict"
49
50  }
51
52  ignored_files = ["boot.py", "/helper/helper_methods.py", "/helper/wifi_manager.py",
53                   "/helper/git_helper.py", "ca.crt", "config.py", "pymakr.conf", ".gitignore"]
54  # ------------ End config ---------------#
```

Abbildung 4 boot.py Konfiguration

Abb. 4 zeigt den Config-Abschnitt. Er lädt verschiedene Parameter, die in der config.py definiert werden. Dabei handelt es sich um die WLAN-Daten mit der SSID und Passwort, Informationen für die MQTT-Broker-Verbindung wie Host, Port, Benutzername und Passwort. Außerdem Geräteeinstellungen wie die „client_id“ und der „alias“. Dazu noch die Topics auf denen abonniert bzw. veröffentlicht wird. (vgl. Giesa 2024). Für das Projekt neu dazugekommen sind Topics zum Erfragen der Raumplandaten, bzw. für den Erhalt dieser, sowie die „api_route“ für das WLAN-Fingerprinting.

```python
# Callback method for incoming mqtt messages
def sub_cb(topic, msg):
    global last_message_time
    if msg is not None:
        try:
            msg_json = json.loads(msg)
            # Reset the timer when a message is received
            last_message_time = time.time()
            if msg_json["method"] == "update":
                for file in msg_json["files"]:
                    if file_helper.isFileOnBlacklist(ignored_files, file["file_name"]):
                        log("File is on Blacklist")
                        continue
                    result = file_helper.write_filename_and_folders(
                        file["file_name"], file["file_content"])
                    log(str(result))
            elif msg_json["method"] == "delete":
                if not file_helper.isFileOnBlacklist(ignored_files, msg_json["file"]):
                    result = file_helper.delete_file(msg_json["file"])
                    log(str(result))
            elif msg_json["method"] == "get_tree":
                result = file_helper.list_files_and_folders()
                log(str(result))
            elif msg_json["method"] == "get_free_space":
                result = file_helper.get_free_disk_space()
                log("Free diskspace: "+str(result))
            elif msg_json["method"] == "reset":
                machine.lightsleep(5000)
                machine.reset()
            elif msg_json["method"] == "room_schedule":
                print("Raumplan empfangen und gespeichert.")
                room_schedule = json.loads(msg)
                file_helper.save_json_to_file(room_schedule)

            else:
                print("Method:", msg_json.get("method", "Kein method-Feld vorhanden"))
        except Exception as e:
            print(f"Fehler beim Verarbeiten der Nachricht: {e}")
        gc.collect()  # Clean up memory
```

Abbildung 5 boot.py sub_cb(topic,msg)

In der Abb. 5 wird die Callback-Funktion dargestellt. Diese kümmert sich um das unterschiedliche Verhalten des ESP32 bei eingehenden MQTT-Nachrichten. So werden Dateien vom Broker heruntergeladen und auf dem ESP32 gespeichert, wenn eine Nachricht mit der „method“ update empfangen wird. „delete“ löst ein Löschen von Dateien auf dem ESP32 aus. „get_tree“ schickt die Dateistruktur des ESP32 zurück und „get_free_space“ den verfügbaren Speicherplatz. „reset“ führt einen Neustart des ESP32 aus. (vgl. Giesa, 2024) In „room_schedule“ wird der Raumplan empfangen und als room_schedule.json mithilfe des „file_helper“ gespeichert.

```
91  # Connect to mqtt broker
92  def connect_to_mqtt_and_subscribe_topic():
93      global mqtt_client
94      mqtt_client = MQTTClient(
95          config['client_id'],
96          config['host'],
97          config['host_port'],
98          config['mqtt_user'],
99          config['mqtt_pw'],
100         keepalive=60)
101     mqtt_client.set_callback(sub_cb)
102     mqtt_client.connect()
103
104     mqtt_client.subscribe(config['topic_sub_all'], qos=1)
105     mqtt_client.subscribe(config['topic_sub_device'], qos=1)
106     log("Subscribing to topic:", "esp32/schedule/" + config["room_name"])
107     mqtt_client.subscribe(config['topic_table_display_answer'], qos=1)
108
109     log("Connected to MQTT  Broker")
```

Abbildung 6 boot.py connect_to_mqtt_and_subscribe_topic()

In Abb. 6 Ist die Funktion zur Herstellung der Verbindung zum MQTT-Broker zu sehen, sowie das Abonnieren von mehreren MQTT-Themen auch Topics genannt. Dadurch können Nachrichten empfangen werden. Es werden allgemeine Nachrichten („topic_sub_all“), Gerät spezifische Nachrichten(„topic_sub_device“) (vgl. Giesa, 2024) und die Raumdaten empfangen(„topic_table_display_asnwer“).

```python
async def wait_for_reset():
    global last_message_time
    start_time = time.time()  # Store the start time

    while True:
        elapsed_time = time.time() - start_time  # Time since start
        remaining_time = 300 - int(elapsed_time)

        if last_message_time is not None:
            # If a message was received, check time since last message
            last_elapsed = time.time() - last_message_time
            remaining_last_msg = 300 - int(last_elapsed)

            if remaining_last_msg <= 0:
                print("Kein Update in den letzten 5 Minuten seit letzter Nachricht. Neustart des ESP32...")
                machine.reset()
            else:
                print(f"Countdown (seit letzter Nachricht): {remaining_last_msg} Sekunden bis zum Neustart.")

        else:
            # If **no message has ever** been received, check total time
            print(f"Countdown (seit Start): {remaining_time} Sekunden bis zum Deep Sleep.")

            if last_message_time is not None:
                print("Neue Nachricht empfangen! Reset des Deep-Sleep-Timers.")
                start_time = time.time()  # Timer zurücksetzen
            elif remaining_time <= 0:
                print("Kein Update in den letzten 5 Minuten. Prüfe erneut, ob eine Nachricht empfangen wurde...")

                # **Double check before deep sleep**
                if last_message_time is None:
                    print("Immer noch keine Nachricht. Gehe in Deep Sleep...")
                    deep_sleep(8, 0, 7, 55)
                else:
                    print("Neue Nachricht empfangen! Reset des Deep-Sleep-Timers.")
                    start_time = time.time()  # Reset timer because a message was received

        await asyncio.sleep(1)
```

Abbildung 7 boot.py wait_for_reset()

In Abb. 7 wird die Funktion „wait_for_reset()“ gezeigt, die für die Überwachung eingehender Nachrichten und das automatische Zurücksetzen bzw. den Deep-Sleep-Modus des ESP32 verantwortlich ist.
Diese Funktion sorgt dafür, dass der ESP32 nach 5 Minuten Inaktivität entweder neu startet oder in den Deep-Sleep-Modus wechselt, um Energie zu sparen und eine zuverlässige Systemfunktionalität zu gewährleisten.

Beim Aufruf der Funktion wird der Startzeitpunkt („start_time“) erfasst, um die Gesamtdauer der Inaktivität zu messen. Falls der ESP32 eine MQTT-Nachricht empfangen hat, wird die Zeit seit der letzten Nachricht („last_message_time“) überprüft. Falls seit der letzten Nachricht mehr als 5 Minuten vergangen sind, wird der ESP32 automatisch neu gestartet („machine.reset()“).

Falls keine Nachricht seit dem letzten Start oder Neustart empfangen wurde, überprüft die Funktion die verstrichene Zeit seit dem Start. Falls 300 Sekunden („remaining_time <= 0“) abgelaufen sind, prüft das System ein zweites Mal, ob in der Zwischenzeit eine Nachricht empfangen wurde.

Falls der ESP32 keine Daten empfängt, geht er in den Deep Sleep, um den Stromverbrauch zu minimieren, so wird unnötiger Stromverbrauch vermieden, falls der Server Probleme hat.

```
159 def request_room_schedule():
160     try:
161         room_request = config['room_name']
162         print(f"Fordere Raumplan für '{room_request}' an...")
163         mqtt_client.publish(config['topic_table_display_request'],room_request,qos=0)
164     except Exception as e:
165         raise Exception(f"Fehler beim Senden der Raumplan-Anfrage: {e}")
```

Abbildung 8 boot.py request_room_schedule()

In Abb. 8 wird die Funktion „request_room_schedule()“ dargestellt, die für das Anfordern eines Raumbelegungsplans über das MQTT-Protokoll verantwortlich ist. Diese Funktion ermöglicht es dem ESP32, gezielt einen Raumplan für den in der Konfiguration hinterlegten Raum („config['room_name']“) vom Server anzufordern.

Beim Aufruf der Funktion wird zunächst der „room_name “ aus der Konfigurationsdatei ausgelesen und in der Variable „room_request“ gespeichert. Anschließend gibt die Funktion eine Bestätigungsmeldung über die serielle Konsole aus („print(f"Fordere Raumplan für '{room_request}' an..."“)), um den Status der Anfrage sichtbar zu machen.

Die eigentliche Kommunikation mit dem MQTT-Broker erfolgt über „mqtt_client.publish()“, wobei die Anfrage auf dem spezifischen Topic „esp32/request_room_schedule“ veröffentlicht wird. Dieses Topic wurde eigens für die Raumplan-Anforderungen eingeführt, sodass der Server eine gezielte Antwort mit den entsprechenden Belegungsdaten zurücksenden kann.

```
135  # Print and public via mqtt
136  def log(msg):
137      global mqtt_client
138      message = ujson.dumps({
139              "device_id": config['client_id'],
140              "alias": config['alias'],
141              "message": msg
142      })
143
144      print(msg)
145      mqtt_client.publish(config['topic_publish_log'], message)
```

Abbildung 9 boot.py log(msg)

In Abb. 9 wird die Funktion „log()“ dargestellt, die für das Protokollieren und Veröffentlichen von Systemnachrichten über MQTT zuständig ist. Diese Funktion ermöglicht es dem ESP32, Statusmeldungen und Debug-Informationen sowohl in der seriellen Konsole als auch über das MQTT-Protokoll an den Server zu senden.

Beim Aufruf der Funktion wird die übergebene Nachricht („msg“) zusammen mit Geräteinformationen als JSON-Objekt formatiert. Dazu werden der „client_id“ und der alias aus der Konfigurationsdatei (config.py) ausgelesen und mit der Nachricht in einem JSON-String („ujson.dumps()“) zusammengefasst.

Anschließend wird die Nachricht auf zwei Wegen veröffentlicht. Einmal die serielle Ausgabe: Die Nachricht wird über „print(msg)“ in der Konsole ausgegeben, um eine direkte Überprüfung während der Laufzeit zu ermöglichen, sowie über MQTT-Publishing. Die formatierte JSON-Nachricht wird dabei über „mqtt_client.publish()“ auf dem Topic „log“ veröffentlicht, sodass der Server sie empfangen und ggf. weiterverarbeiten kann.

Durch diese Implementierung ist sichergestellt, dass wichtige Systemereignisse, Debug-Informationen oder Fehlerberichte nicht nur lokal sichtbar, sondern auch zentral über den MQTT-Broker abrufbar sind. Dies erleichtert die Überwachung und Wartung des Systems erheblich (vgl. Giesa, 2024).

```python
# Function to store errors with timestamp and automatically delete old entries
def log_error(error_message):
    try:
        timestamp = time.localtime()
        formatted_time = "{:04d}-{:02d}-{:02d} {:02d}:{:02d}:{:02d}".format(
            timestamp[0], timestamp[1], timestamp[2],  #Year, Month, Day
            timestamp[3], timestamp[4], timestamp[5]   # Hour, Minute, Second
        )

        log_entries = [] # Ensure it is a list

        # Check if the file exists
        if LOG_FILE in os.listdir():
            with open(LOG_FILE, "r") as file:
                log_entries = file.read().splitlines()

        # Add new log message
        log_entries.append(f"[{formatted_time}] {error_message}")

        # Limit the number of entries to MAX_LOG_ENTRIES (remove oldest)
        if len(log_entries) > MAX_LOG_ENTRIES:
            log_entries = log_entries[-MAX_LOG_ENTRIES:]  # Keep only the last 31 entries

        # Save logs back to the file
        with open(LOG_FILE, "w") as file:
            for line in log_entries:
                file.write(line + "\n")  #Write each line individually

        print(f"Fehler protokolliert: [{formatted_time}] {error_message}")

    except Exception as e:
        print(f"Fehler beim Speichern in Log-Datei: {e}")
```

Abbildung 10 boot.py log_error(error_message)

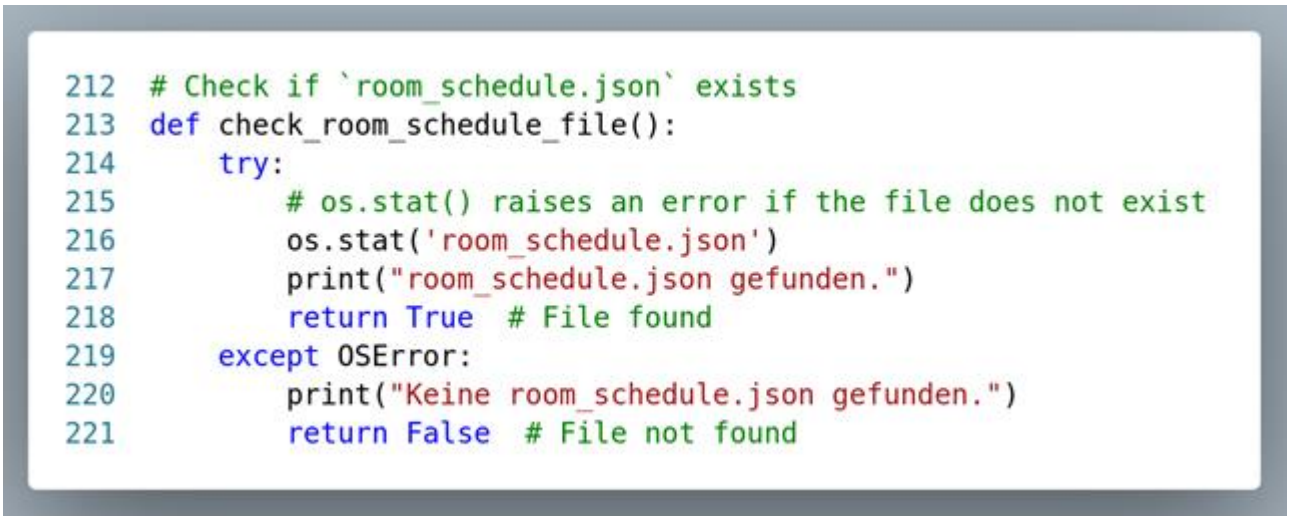
In Abb. 10 wird die Funktion log_error() gezeigt, die für die Protokollierung von Fehlern und die automatische Verwaltung der Log-Datei zuständig ist.
Diese Funktion stellt sicher, dass Fehlermeldungen mit einem Zeitstempel versehen und in einer lokalen Log-Datei gespeichert werden. Sie wurde auch neu für das Projekt hinzugefügt, um Fehler zu protokollieren die auftreten, während keine Verbindung zum Server besteht.

Dabei wird zunächst ein Zeitstempel im Format YYYY-MM-DD HH:MM:SS generiert, um den Zeitpunkt des Fehlers eindeutig zu erfassen. Anschließend wird überprüft, ob die Log-Datei bereits existiert und falls ja, werden die bestehenden Einträge ausgelesen. Die neue Fehlermeldung wird dann zur Liste der Log-Einträge hinzugefügt.

Um eine übersichtliche Fehlerhistorie zu gewährleisten, ist die Anzahl der gespeicherten Einträge auf MAX_LOG_ENTRIES (31) begrenzt. Falls die Anzahl überschritten wird, werden die ältesten Einträge gelöscht, sodass nur die letzten 31 Fehler protokolliert bleiben.

Nach der Aktualisierung der Einträge wird die Datei neu geschrieben, indem jede Zeile einzeln gespeichert wird. Falls ein Fehler beim Speichern auftritt, wird eine Fehlermeldung in der Konsole ausgegeben.

Diese Funktion ist essenziell für die Fehleranalyse und Wartung des Systems, da sie eine detaillierte Nachverfolgung von Problemen ermöglicht, auch wenn keine Verbindung zum Internet besteht.



```
212  # Check if `room_schedule.json` exists
213  def check_room_schedule_file():
214      try:
215          # os.stat() raises an error if the file does not exist
216          os.stat('room_schedule.json')
217          print("room_schedule.json gefunden.")
218          return True  # File found
219      except OSError:
220          print("Keine room_schedule.json gefunden.")
221          return False  # File not found
```

Abbildung 11 boot.py check_room_schedule_file()

In Abb. 11 wird die Funktion „check_room_schedule_file()“ dargestellt, die für die Überprüfung der Existenz der Datei „room_schedule.json“ im Dateisystem des ESP32 verantwortlich ist. Diese Funktion ist auch neu für das Projekt hinzugefügt worden. Diese Datei enthält die zuletzt empfangenen Raumbelegungsdaten und wird benötigt, um die Anzeige auf dem E-Paper-Display auch ohne aktive Verbindung zum Server zu ermöglichen.

Beim Aufruf der Funktion wird die Methode „os.stat('room_schedule.json')“ verwendet, um zu prüfen, ob die Datei existiert. Sollte die Datei nicht vorhanden sein, löst „os.stat()“ eine OSError-Exception aus, die im except-Block abgefangen wird.

Je nach Ergebnis gibt die Funktion eine entsprechende Meldung aus. Falls die Datei vorhanden ist, wird "room_schedule.json gefunden." ausgegeben und die Funktion gibt „True“ zurück. Falls die Datei nicht existiert, wird "Keine room_schedule.json gefunden." ausgegeben und die Funktion gibt „False“ zurück.

Diese Implementierung ist essenziell für die Steuerlogik des ESP32, da sie entscheidet, ob das System mit bereits gespeicherten Daten arbeiten kann oder ob eine neue Anfrage an den Server gesendet werden muss. So wird sichergestellt, dass die Anzeige auch nach einem Neustart oder einer unterbrochenen Verbindung weiterhin funktioniert, sowie mit dem begrenzten Arbeitsspeicher

des ESP32 auskommt.

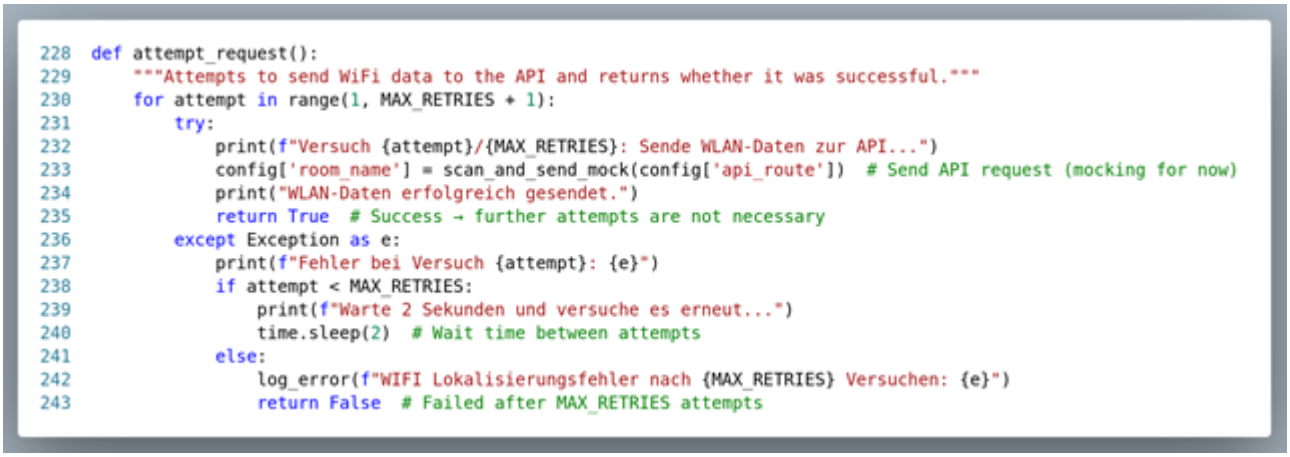
```
158  # Start der Verbindung zu WiFi und MQTT
159  def setup_wifi_and_mqtt():
160      wifi_manager.do_connect(config['ssid'], config['wifi_pw'])
161      connect_to_mqtt_and_subscribe_topic()
```

Abbildung 12 boot.py setupt_wifi_and_mqtt()

In Abb. 12 wird die Funktion „setup_wifi_and_mqtt()“ dargestellt, die für den Aufbau der WLAN- und MQTT-Verbindung auf dem ESP32 verantwortlich ist. Diese Funktion stellt sicher, dass der Mikrocontroller eine stabile Netzwerkverbindung aufbaut und anschließend eine Verbindung zum MQTT-Broker herstellt, um Daten zu empfangen und zu senden.

Beim Aufruf der Funktion werden zwei wesentliche Schritte ausgeführt, einmal die WLAN-Verbindung herstellen, dabei wird die Funktion „wifi_manager.do_connect()“ mit den in der config.py definierten Zugangsdaten (SSID und Passwort) aufgerufen. Falls eine Verbindung bereits besteht, wird diese weiter genutzt. Andernfalls wird die WLAN-Schnittstelle aktiviert und die Verbindung hergestellt. Danach kommt es zur Verbindung mit dem MQTT-Broker und Topics werden abonniert. Dabei wird sobald die WLAN-Verbindung aktiv ist, „connect_to_mqtt_and_subscribe_topic()“ aufgerufen. Diese Funktion baut die MQTT-Verbindung auf und abonniert relevante Topics, darunter „esp32/schedule/{room_name}“, um Raumbelegungspläne zu empfangen.

Durch diese Implementierung wird sichergestellt, dass der ESP32 automatisch eine Netzwerkverbindung herstellt und sich mit dem MQTT-Broker synchronisiert, um neue Daten oder Befehle zu empfangen (vgl. Giesa, 2024).



```
228 def attempt_request():
229     """Attempts to send WiFi data to the API and returns whether it was successful."""
230     for attempt in range(1, MAX_RETRIES + 1):
231         try:
232             print(f"Versuch {attempt}/{MAX_RETRIES}: Sende WLAN-Daten zur API...")
233             config['room_name'] = scan_and_send_mock(config['api_route'])  # Send API request (mocking for now)
234             print("WLAN-Daten erfolgreich gesendet.")
235             return True  # Success → further attempts are not necessary
236         except Exception as e:
237             print(f"Fehler bei Versuch {attempt}: {e}")
238             if attempt < MAX_RETRIES:
239                 print(f"Warte 2 Sekunden und versuche es erneut...")
240                 time.sleep(2)  # Wait time between attempts
241             else:
242                 log_error(f"WIFI Lokalisierungsfehler nach {MAX_RETRIES} Versuchen: {e}")
243                 return False  # Failed after MAX_RETRIES attempts
```

Abbildung 13 boot.py attempt_request()

In Abb. 13 wird die Funktion „attempt_request()“ gezeigt, die für die mehrfache Übertragung von WLAN-Scan-Daten an eine API zur Standortbestimmung verantwortlich ist.
Da der bereits laufende Server auf die erste Anfrage häufig mit einem 500er HTTP Error Code reagiert , sorgt diese Funktion dafür, dass die Daten bis zu MAX_RETRIES Mal erneut gesendet werden, falls ein Fehler auftritt.

Die Funktion führt eine Schleife von 1 bis „MAX_RETRIES“ aus, um mehrfach eine Verbindung zur API herzustellen und eine zuverlässige Raumzuordnung zu gewährleisten. Dabei wird „scan_and_send_mock(config['api_route'])“ ausgeführt, wodurch der ESP32 eine Liste der verfügbaren WLAN-Netzwerke scannt und die erfassten Daten an die vorgegebene API-Adresse („config['api_route']“) übermittelt. Die vom Server zurückgesendete Raumzuordnung wird anschließend in „config['room_name']“ gespeichert, sodass das Gerät seine aktuelle Position innerhalb des Systems bestimmen und entsprechend die zugehörigen Raumbelegungsdaten abrufen kann. Falls die Daten erfolgreich gesendet wurden, gibt die Funktion „True“ zurück und beendet die Schleife vorzeitig. Dadurch werden weitere unnötige Versuche vermieden. Falls ein Fehler auftritt, wird eine Fehlermeldung mit „print()“ ausgegeben. Falls es sich nicht um den letzten Versuch handelt, wartet das System 2 Sekunden („time.sleep(2)“), bevor der nächste Versuch startet.

Falls nach „MAX_RETRIES“ Versuchen immer noch kein Erfolg erzielt wurde, wird der Fehler mit „log_error()“ protokolliert. Die Funktion gibt dann „False“ zurück, um anzuzeigen, dass die WLAN-Daten nicht erfolgreich gesendet wurden.

Diese Funktion gewährleistet, dass der ESP32 möglichst zuverlässig seinen aktuellen Raumstandort bestimmt und sich selbstständig korrigieren kann, falls die API nicht sofort antwortet.

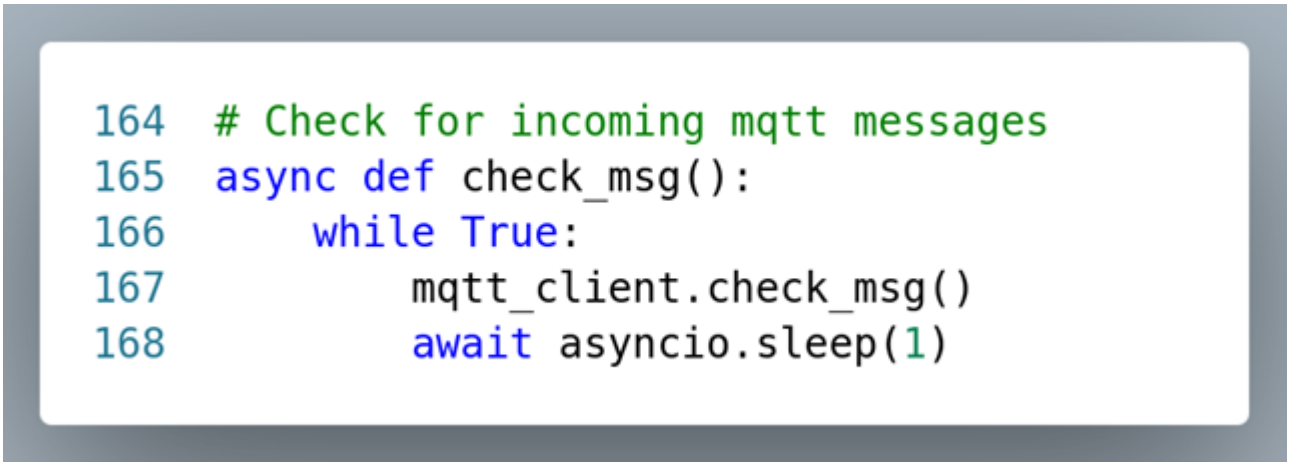


Abbildung 14 boot.py check_msg()

In Abb. 14 wird die Funktion „check_msg() dargestellt," die für das regelmäßige Überprüfen eingehender MQTT-Nachrichten verantwortlich ist. Diese Funktion läuft asynchron in einer Endlosschleife und stellt sicher, dass der ESP32 jederzeit auf neue Nachrichten vom MQTT-Broker reagieren kann.

Die Implementierung erfolgt mit „uasyncio", wodurch „check_msg()" nicht-blockierend arbeitet und parallel zu anderen Aufgaben ausgeführt werden kann. Der Ablauf gestaltet sich wie folgt: Zunächst wird kontinuierlich überprüft, ob neue MQTT-Nachrichten eingegangen sind. Dazu ruft die Funktion „mqtt_client.check_msg()" regelmäßig den Nachrichtenspeicher des MQTT-Clients ab. Sobald eine Nachricht empfangen wird, erfolgt die automatische Weiterleitung an die Callback-Funktion „sub_cb()", welche die Verarbeitung der Nachricht übernimmt. Um eine übermäßige Belastung des Prozessors zu vermeiden, wird nach jeder Überprüfung eine kurze Pause durch den Befehl „await asyncio.sleep(1)" eingelegt. Diese Verzögerung stellt sicher, dass das System effizient arbeitet, ohne unnötige Rechenressourcen zu beanspruchen, während gleichzeitig eine schnelle Reaktion auf eingehende Nachrichten gewährleistet bleibt.

Diese Implementierung sorgt für den reibungslosen Betrieb des ESP32, da sie eine kontinuierliche Kommunikation mit dem MQTT-Broker gewährleistet. In Kombination mit „sub_cb()" ermöglicht „check_msg()" die Echtzeit-Verarbeitung von Nachrichten wie Raumplan-Updates, OTA-Befehlen oder Neustart-Kommandos (vgl. Giesa, 2024).

```
try:
    # Check if room_schedule.json exists
    if check_room_schedule_file():
         # If the file exists, start the main function
        main.main()
        print("Main gestartet.")
    else:
        # If the file does not exist, try to establish a connection
        try:
            setup_wifi_and_mqtt()
        except Exception as e:
            log_error(f"WLAN/MQTT-Verbindungsfehler: {e}")
            deep_sleep(8, 00, 7, 55)
        if attempt_request():
            # Try to request the room schedule
            request_room_schedule()
        else:
            # If still not possible after 3 attempts, enter deep sleep
            print("Maximale Anzahl an Versuchen erreicht. Gehe in den Deep-Sleep-Modus...")
            deep_sleep(8, 00, 7, 55)  # Deep-Sleep aus main.py nutzen
        # Start asynchronous tasks
        asyncio.create_task(status_helper.send_status(mqtt_client))
        asyncio.run(asyncio.gather(check_msg(),wait_for_reset()))

except Exception as e:
    log_error(f"Schwerwiegender Fehler: {e}")
    machine.reset()
finally:
     asyncio.new_event_loop()
```

Abbildung 15 boot.py Programmablauf

In Abb. 15 wird der Hauptablauf des Programms dargestellt. Diese Schleife steuert die Verbindung zum WLAN/MQTT-Broker, die Standortbestimmung, die Anfrage nach Raumbelegungsdaten sowie die Verwaltung von Hintergrundprozessen.

Das System verfolgt dabei eine strukturierte Reihenfolge, um sicherzustellen, dass Verbindungsabbrüche oder fehlgeschlagene API-Anfragen erkannt und behandelt werden, ohne dass der ESP32 in einem fehlerhaften Zustand verbleibt.

Zunächst überprüft das System, ob die Datei „room_schedule.json“ vorhanden ist. Falls die Datei existiert, wird die main.py ausgeführt, die für die Anzeige des Raumplans auf dem E-Paper-Display verantwortlich ist. Falls die Datei jedoch nicht existiert, wird versucht eine Netzwerkverbindung über „setup_wifi_and_mqtt()“ herzustellen. Falls diese Verbindung fehlschlägt, wird der Fehler protokolliert und der ESP32 in den Deep-Sleep-Modus versetzt, um Energie zu sparen.

Anschließend führt das Programm einen WLAN-Scan zur Standortbestimmung (WLAN-Fingerprinting) aus, um den aktuellen Raum des ESP32 zu ermitteln. Falls die Standortbestimmung erfolgreich ist („attempt_request()“ gibt „True“ zurück), wird über MQTT eine Anfrage an den

Server gesendet („request_room_schedule()“), um die aktuellen Raumbelegungsdaten abzurufen. Falls die Standortbestimmung dreimal fehlschlägt, wechselt das System in den Deep-Sleep-Modus, um unnötigen Stromverbrauch zu vermeiden.

Nach der Initialisierung werden mehrere asynchrone Prozesse („asyncio.create_task()“) gestartet, darunter „status_helper.send_status(mqtt_client)“ zur Überwachung des Gerätestatus, „check_msg()“, um eingehende MQTT-Nachrichten kontinuierlich zu verarbeiten,„wait_for_reset()“, das überprüft, ob eine neue Nachricht empfangen wurde, und bei längerer Inaktivität einen automatischen Neustart oder Deep-Sleep auslöst.

Diese nicht-blockierenden asyncio-Tasks ermöglichen es dem ESP32 parallel Nachrichten zu empfangen, den Status zu überwachen und sich bei Bedarf selbstständig zurückzusetzen. Falls ein unerwarteter Fehler auftritt, wird dieser in einer Log-Datei gespeichert („log_error()“), und das System führt einen Neustart über „machine.reset()“ durch, um eine dauerhafte Fehlfunktion zu verhindern.

Diese Art der Implementierung über 2 unterschiedliche Betriebsmodi war nötig, da es bei gleichzeitiger Ausführung des gesamten Code zu Problemen mit dem Arbeitsspeicher kam, da der Framebuffer 48kb benötigt, die bei gleichzeitiger Verwendung mit der OTA-Funktionalitäten nicht mehr verfügbar waren und unterscheidet sich an dieser Stelle auch vom Code von Till Giesa (vgl. Giesa, 2024).

5.2.3 Datenverarbeitung und Anzeige

Dieses Kapitel behandelt die Verarbeitung der empfangenen Raumbelegungsdaten und deren anschließende Darstellung auf dem E-Paper-Display. Nachdem der ESP32 über MQTT neue Raumbelegungsdaten erhalten hat, werden diese zunächst lokal gespeichert, anschließend verarbeitet und schließlich auf dem Display ausgegeben.

Zunächst werden die empfangenen Daten aus der JSON-Datei („room_schedule.json“) geladen und analysiert. Anschließend erfolgt die grafische Darstellung in tabellarischer Form auf dem E-Paper-Display, sodass die Raumbelegung übersichtlich nach Wochentagen, Zeitblöcken und Belegungsstatus geordnet wird. Um die Speicherressourcen optimal zu nutzen, werden veraltete Daten gezielt entfernt, sodass stets nur die aktuellen Belegungspläne gespeichert bleiben. Darüber hinaus stellt das System sicher, dass die boot.py den jeweils richtigen Betriebsmodus wählt, abhängig davon, ob eine gültige Belegungsdatei vorhanden ist oder eine neue Anfrage an den Server erforderlich wird.

Nach Abschluss der Darstellung wird der ESP32 in den Deep-Sleep-Modus versetzt, um den Energieverbrauch zu minimieren und eine möglichst lange Betriebsdauer zu gewährleisten. Durch diese optimierte Architektur wird sichergestellt, dass das System zuverlässig arbeitet, ohne unnötige Ressourcen zu beanspruchen, und gleichzeitig die Aktualität der angezeigten Informationen gewährleistet bleibt.

Die Implementierung basiert auf zwei zentralen Modulen. Einmal der main.py, diese ist verantwortlich für das Laden und Löschen der JSON-Daten sowie für die Steuerung der Anzeige. Das zweite Modul ist die view.py, sie ist zuständig für das Zeichnen der Raumbelegungsdaten als Tabelle auf dem E-Paper-Display. Im Folgenden werden die einzelnen Komponenten und deren Funktionsweise detailliert beschrieben.

5.2.3.1 Main.py

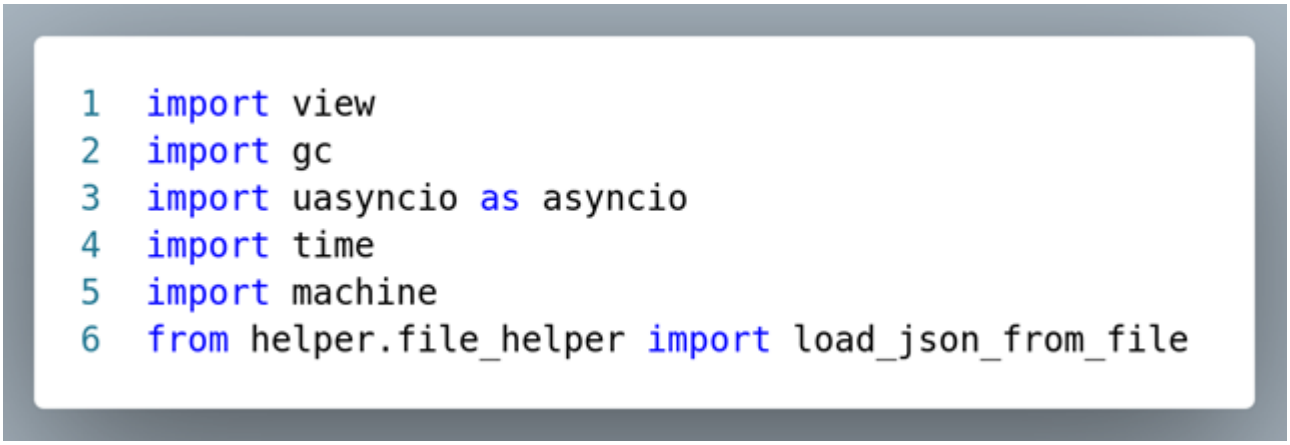
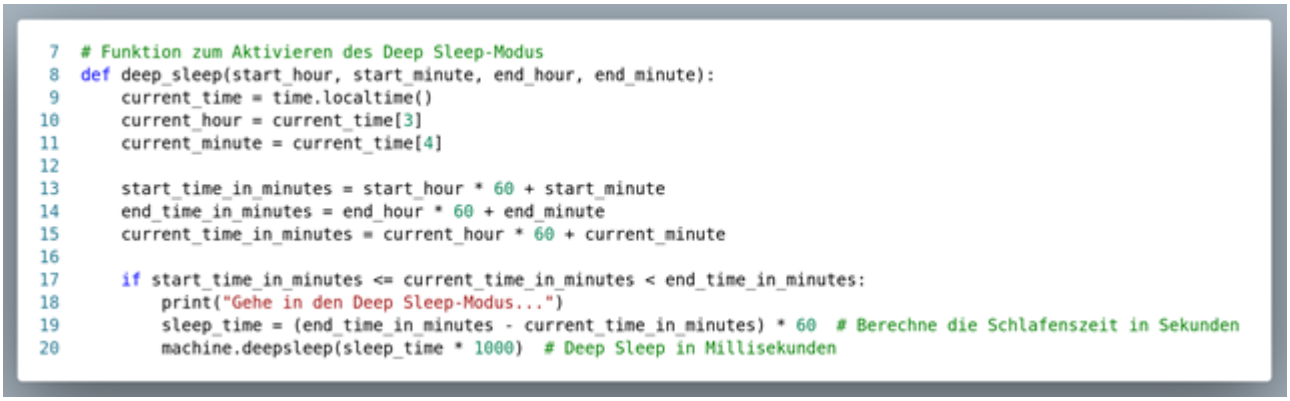


Abbildung 16 main.py imports

In Abb. 16 werden die Imports der main.py dargestellt.
Das „view“-Modul ist für die Darstellung der Raumbelegungsdaten auf dem E-Paper-Display zuständig. Es übernimmt die Formatierung und das Zeichnen der Tabellenstruktur. Das „gc“-Modul sorgt für die Speicherverwaltung, indem es ungenutzte Speicherbereiche freigibt, um die Leistung des ESP32 zu optimieren. Mit „time“ werden Zeitstempel und Wartezeiten verwaltet, während das „machine“-Modul hardwarebezogene Funktionen wie den Deep-Sleep-Modus bereitstellt. Zusätzlich wird mit „load_json_from_file()“ aus dem „file_helper“-Modul die zuletzt gespeicherte Raumbelegungsdatei (room_schedule.json) geladen, die anschließend für die Anzeige verarbeitet wird (vgl. Flemming, 2024).



```python
# Funktion zum Aktivieren des Deep Sleep-Modus
def deep_sleep(start_hour, start_minute, end_hour, end_minute):
    current_time = time.localtime()
    current_hour = current_time[3]
    current_minute = current_time[4]

    start_time_in_minutes = start_hour * 60 + start_minute
    end_time_in_minutes = end_hour * 60 + end_minute
    current_time_in_minutes = current_hour * 60 + current_minute

    if start_time_in_minutes <= current_time_in_minutes < end_time_in_minutes:
        print("Gehe in den Deep Sleep-Modus...")
        sleep_time = (end_time_in_minutes - current_time_in_minutes) * 60  # Berechne die Schlafenszeit in Sekunden
        machine.deepsleep(sleep_time * 1000)  # Deep Sleep in Millisekunden
```

Abbildung 17 main.py deep_sleep(start_hour, start_minute, end_hour, end_minute)

In Abb. 17 wird die Funktion „deep_sleep()“ dargestellt, die für das energiesparende Umschalten des ESP32 in den Deep-Sleep-Modus verantwortlich ist. Der Deep-Sleep-Modus reduziert den Stromverbrauch erheblich, indem der Mikrocontroller für eine definierte Zeitspanne in einen niedrigen Energiezustand versetzt wird, bevor er automatisch neu startet.

Die Funktion berechnet zunächst die aktuelle Zeit aus „time.localtime() “ und wandelt die Start- und Endzeiten des definierten Ruhezeitraums in Minuten um. Anschließend wird überprüft, ob die aktuelle Zeit innerhalb dieses Intervalls liegt. Falls dies der Fall ist, wird die verbleibende Schlafenszeit („sleep_time“) berechnet und der ESP32 mit „machine.deepsleep(sleep_time * 1000)“ in den Tiefschlafmodus versetzt.

Dieser Mechanismus ist besonders wichtig, um die Batterielaufzeit zu maximieren, da der ESP32 nur dann aktiv bleibt, wenn eine Datenverarbeitung erforderlich ist. Während des Deep-Sleep-Modus werden alle nicht benötigten Funktionen deaktiviert, wodurch der Stromverbrauch auf ein Minimum reduziert wird (vgl. Flemming, 2024).

```python
# Hauptprogramm
def main():
    while True:
        room_schedule = None
        try:
            room_schedule = load_json_from_file()
        except Exception:
            print("Keine room_schedule.json.")
            pass

        if room_schedule and "data" in room_schedule:
            gc.collect
            view.draw_table_with_time(room_schedule)  # Tabelle mit den neuen JSON-Daten anzeigen
            try:
                import os
                os.remove('room_schedule.json')  # Datei löschen
                print("Room schedule JSON-Datei wurde nach der Anzeige gelöscht.")
            except OSError as e:
                print(f"Fehler beim Löschen der JSON-Datei: {e}")



        # Überprüfen, ob der ESP32 in den Deep Sleep-Modus versetzt werden soll
        deep_sleep(8, 00, 7, 55)
        print("Noch keine Deep Sleep Zeit.")
        time.sleep(1)  # Kurze Pause, um die Schleife nicht zu überlasten

```

Abbildung 18 main.py main()

In Abb. 18 wird die Funktion „main()“ dargestellt, die für die Datenverarbeitung und Anzeige auf dem E-Paper-Display des ESP32 verantwortlich ist. Diese Funktion überprüft regelmäßig, ob eine neue Raumbelegungsdatei („room_schedule.json“) vorhanden ist, und aktualisiert bei Bedarf die Anzeige. Zu Beginn wird versucht die JSON-Datei zu laden. Falls diese existiert und gültige Daten enthält, wird die Speicherverwaltung („gc.collect“) aktiviert, um Speicherplatz freizugeben, bevor die Funktion „view.draw_table_with_time(room_schedule)“ aufgerufen wird, um die Raumbelegungsdaten als Tabelle auf dem E-Paper-Display darzustellen. Nach der erfolgreichen Anzeige wird die Datei gelöscht, um Speicherplatz zu sparen und sicherzustellen, dass bei der nächsten Aktualisierung neue Daten abgerufen werden.

Anschließend überprüft die Funktion, ob der ESP32 in den Deep-Sleep-Modus versetzt werden soll. Dazu wird die Funktion „deep_sleep(8, 00, 7, 55)“ aufgerufen, die festlegt, ob der ESP32 innerhalb des definierten Zeitraums schlafen soll. Falls der aktuelle Zeitpunkt nicht innerhalb der Schlafenszeit liegt, wird "Noch keine Deep Sleep Zeit." ausgegeben und das System bleibt aktiv. Eine kurze Wartezeit („time.sleep(1)“) sorgt dafür, dass die Schleife nicht unnötig Prozessorleistung verbraucht und das System effizient arbeitet (vgl. Flemming, 2024).

5.2.3.2 View.py

Abbildung 19 view.py Imports

In Abb. 19 sind die Imports für die Steuerung des E-Paper-Displays zu sehen. Diese ermöglichen die grafische Darstellung der Raumbelegungspläne und stellen die notwendigen Hardware-Schnittstellen bereit. „gc“ importiert den Garbage Collector, der dazu dient, unbenutzten Speicher freizugeben und die Speicherverwaltung zu optimieren, um den begrenzten Speicher des ESP32 effizient zu nutzen. „framebuf“ bietet eine Framebuffer-Schnittstelle, die es ermöglicht, Grafiken oder Text darzustellen, bevor sie auf das E-Paper-Display geschrieben werden. „Writer“ stellt eine Schriftverarbeitungsschnittstelle bereit, die es erlaubt, Texte in verschiedenen Schriftarten auf das Display zu rendern (Hinch, 2016–2018). „machine.Pin “ ermöglicht die Konfiguration und Steuerung der GPIO-Pins, die für die Ansteuerung des E-Paper-Displays notwendig sind und „machine.SPI“ stellt die Schnittstelle zur seriellen Kommunikation (Serial Peripheral Interface) bereit, die für die Datenübertragung zwischen ESP32 und E-Paper-Display genutzt wird. „minecraft10“ importiert eine benutzerdefinierte Schriftart, die für die Darstellung der Texte auf dem E-Paper-Display verwendet wird. „epaper7in5_V2“ enthält die Treiberbibliothek für das 7,5-Zoll E-Paper-Display. Diese Bibliothek ermöglicht es, Daten auf das Display zu schreiben, den Bildschirm zu löschen oder in den Sleep-Modus zu versetzen, um Energie zu sparen. (vgl. tanahy, 2020).

```
24  # Define the display width and height
25  w = 800
26  h = 480
```

Abbildung 20 Globale Variablen

In Abb. 20 werden die Definitionen der Breite und Höhe des E-Paper-Displays dargestellt. Diese Parameter legen die Grundstruktur der Anzeige fest und definieren den Zeichenbereich, innerhalb dessen der ESP32 die Raumbelegungspläne darstellt. “w“ legt die Breite des E-Paper-Displays in Pixeln fest. „h“ definiert die Höhe des Displays in Pixeln. Das 7,5-Zoll-Display verwendet eine Auflösung von 800 × 480 Pixeln. Diese Werte bestimmen den verfügbaren Anzeigebereich, in dem Text und Tabellen gezeichnet werden können (vgl. Flemming, 2024).

```
29  # A dummy display is required to access the values.
30  class DummyDisplay(framebuf.FrameBuffer):
31      def __init__(self, buffer, width, height, mode):
32          self.width = width
33          self.height = height
34          self.buffer = buffer
35          self.mode = mode
36          super().__init__(self.buffer, self.width, self.height, self.mode)
```

Abbildung 21 view.py DummyDisplay(framebuf.FrameBuffer)

In Abb. 21 wird die Klasse DummyDisplay dargestellt, die für die Verarbeitung von Framebuffer-Daten auf dem E-Paper-Display benötigt wird. Diese Klasse dient als Hilfsklasse, um an die Pixeldaten des Displays zu gelangen und bestimmte Zeichenoperationen auszuführen (vgl. Flemming, 2024).

Die Klasse erbt von „framebuf.FrameBuffer“, wodurch sie die Funktionalität zur Pixelmanipulation und Zeichenwiedergabe erhält. „__init__(self, buffer, width, height, mode)“ initialisiert die Framebuffer-Instanz mit einem Byte-Array („buffer“), das als Speicher für die Bilddaten dient. Die Breite („width“) und Höhe („height“) werden definiert, um den exakten Anzeigebereich zu bestimmen. „mode“ gibt den Farbmodus der Darstellung an (z. B. Schwarz-Weiß für das E-Paper-

Display). „super().__init__()“ ruft den Konstruktor der FrameBuffer-Klasse auf, um die Vererbung korrekt zu initialisieren (vgl. Flemming, 2024).

Diese Klasse ist notwendig, weil der ESP32 beim Zeichnen von Grafiken auf einem Framebuffer arbeiten muss, bevor die Daten an das Display gesendet werden. Der DummyDisplay ermöglicht es, Zwischenwerte und Pixeldaten zu manipulieren, bevor sie an das E-Paper-Display übertragen werden.

Durch die Nutzung dieser Klasse kann das Layout der Tabellenansicht optimiert und Schriftarten oder andere Zeichenoperationen effizient im Speicher verarbeitet werden, bevor die endgültige Darstellung auf dem E-Paper-Display erfolgt (vgl. Flemming, 2024).

```
38  def load_font(font_name):
39      try:
40          # Dynamically load the font module using __import__
41          font_module = __import__(f'fonts.{font_name}')
42          return getattr(font_module, font_name)
43      except ImportError:
44          print(f"Schriftart {font_name} konnte nicht geladen werden.")
45          return None
```

Abbildung 22 view.py load_font(font_name)

In Abb. 22 wird die Funktion „load_font()“ gezeigt, die für das dynamische Laden von Schriftarten auf dem E-Paper-Display verantwortlich ist. Die Funktion nutzt „__import__()“, um das angegebene Schriftart-Modul (font_name) aus dem „fonts“-Verzeichnis dynamisch zu importieren. Dadurch kann das System zur Laufzeit unterschiedliche Schriftarten laden, ohne dass jede Schriftart manuell eingebunden werden muss. “return getattr(font_module, font_name)“ extrahiert nach dem erfolgreichen Import die tatsächliche Schriftart-Klasse oder -Instanz aus dem Modul. Dadurch erhält der ESP32 Zugriff auf die Schriftart, die anschließend für die Textdarstellung auf dem E-Paper-Display genutzt wird. Fehlermanagement wird mit „except ImportError“ durchgeführt. Falls die gewünschte Schriftart nicht gefunden oder geladen werden kann, wird eine Fehlermeldung ausgegeben. Die Funktion gibt in diesem Fall „None“ zurück, um sicherzustellen, dass das System weiterhin stabil bleibt, auch wenn die Schriftart fehlt (vgl. Flemming, 2024).

```
47  # Time slots only for full hours (for the table)
48  def generate_full_hour_time_slots():
49      time_slots = ["vor 8:00"]
50      for hour in range(8, 20):
51          time_slots.append(f"{hour}:00")
52      time_slots.append("ab 20:00")
53      return time_slots
```

Abbildung 23 view.py generate_full_hour_time_slots()

In Abb. 23 wird die Funktion „generate_full_hour_time_slots() “ gezeigt, die für die Erstellung von Zeitslots in Stundenschritten zuständig ist.
Diese Zeitslots werden verwendet, um den Stundenplan tabellarisch auf dem E-Paper-Display darzustellen. Der erste Eintrag definiert einen vorgelagerten Zeitslot für Veranstaltungen vor 8:00 Uhr. („time_slots = ["vor 8:00"]“). „for hour in range(8, 20): time_slots.append(f"{hour}:00")“ erstellt eine Liste von vollen Stunden von 08:00 bis 19:00 Uhr. Jede Stunde wird im Format "HH:00" gespeichert. „time_slots.append("ab 20:00")“ ergänzt einen abschließenden Zeitslot für Veranstaltungen nach 20:00 Uhr.

Die generierte Liste dient als Grundlage für die tabellarische Darstellung des Stundenplans, indem die Zeitslots als Zeilenüberschriften für die Belegungstabellen auf dem E-Paper-Display genutzt werden.

Durch diese dynamische Erstellung der Zeitslots kann die Darstellung des Stundenplans flexibel angepasst werden, falls sich die Unterrichtszeiten ändern oder erweitert werden.

```
55  # Time slots for events (for the subjects)
56  def generate_event_time_slots():
57      time_slots = ["vor 8:00"]
58      for hour in range(8, 20):
59          for minute in [0, 15, 30, 45]:
60              time_slots.append(f"{hour}:{minute:02d}")
61      time_slots.append("ab 20:00")
62      return time_slots
```

Abbildung 24 view.py generate_event_time_slots()

In Abb. 24 wird die Funktion „generate_event_time_slots()“ dargestellt, die für die Erstellung von detaillierten Zeitslots in 15-Minuten-Schritten zuständig ist.
Diese feineren Zeitslots werden benötigt, um Veranstaltungen mit exakter Start- und Endzeit im Stundenplan darzustellen. „time_slots = ["vor 8:00"]“ definiert einen vorgelagerten Zeitslot für Veranstaltungen, die vor 08:00 Uhr stattfinden. „for hour in range(8, 20)“ erstellt Stunden-Zeitslots von 08:00 bis 19:45 Uhr. „for minute in [0, 15, 30, 45]“ unterteilt jede Stunde in vier Zeitslots (00, 15, 30 und 45 Minuten), sodass Veranstaltungen mit einer höheren Genauigkeit dargestellt werden können. Das Format „{hour}:{minute:02d}“ stellt sicher, dass einheitliche zweistellige Minutenangaben (z. B. "8:00", "8:15", "8:30", "8:45") erzeugt werden. „time_slots.append("ab 20:00")“ ergänzt einen abschließenden Zeitslot für Veranstaltungen nach 20:00 Uhr.

Diese detaillierte Zeitslot-Struktur ermöglicht es, Veranstaltungen präzise in den Stundenplan einzuordnen, indem Beginn- und Endzeiten mit 15-Minuten-Abständen exakt dargestellt werden. Dadurch kann der ESP32 die Raumbelegungspläne auf dem E-Paper-Display besonders übersichtlich und genau präsentieren.

```python
#Helper function to add 45 minutes to a time in the format 'HH:MM'.
def adjust_time_by_45_minutes(time_str):
    hour, minute = map(int, time_str.split(':'))
    minute += 45

    if minute >= 60:
        minute -= 60
        hour += 1

    # Ensure hours remain within the range of 0 to 23
    if hour >= 24:
        hour -= 24

    # Return the new time in 'HH:MM' format
    return f"{hour:02}:{minute:02}"
```

Abbildung 25 view.py adjust_time_by_45_minutes(time_str)

In Abb. 25 wird die Funktion „adjust_time_by_45_minutes()“ dargestellt, die zur Korrektur der Zeitanzeige auf dem E-Paper-Display dient. Diese Anpassung wurde erforderlich, da die Fächer um genau 45 Minuten verschoben waren, ohne dass die Ursache zunächst ersichtlich war.

Obwohl der Header-Bereich bereits in die Berechnungen einbezogen wurde, ergab sich eine fehlerhafte Platzierung der Stundenblöcke, die dazu führte, dass Veranstaltungen zu früh oder zu spät in der Tabelle angezeigt wurden.

Die Funktion verschiebt jede Zeit nach vorne um 45 Minuten und stellt somit sicher , dass die Fächer an den korrekten Positionen im Stundenplan erscheinen. Die Stunden- und Minutenwerte werden extrahiert. Falls die Minutenanzahl 60 überschreitet, wird eine zusätzliche Stunde hinzugefügt und die Minuten entsprechend angepasst. Das dient der Sicherstellung, dass die Anzeige korrekt zweistellig formatiert ist. Diese Funktion sorgt nun dafür, dass die Veranstaltungsblöcke genau an den vorgesehenen Zeitslots ausgerichtet sind und keine unerwarteten Verschiebungen mehr auftreten.

```python
#Splits text to fit within the available space in a cell.
def wrap_text_in_cell(text, max_width, writer):
    lines = []
    words = text.split()
    current_line = ""

    for word in words:
         # Check if adding the next word would exceed the width
        if writer.stringlen(current_line + word) <= max_width:
            current_line += word + " "
        else:
            lines.append(current_line)
            current_line = word + " "

    if current_line:
        lines.append(current_line)

    return lines
```

Abbildung 26 view.py wrap_text_in_cell(text, max_width, writer)

In der Abb. 26 wird die Funktion „wrap_text_in_cell()“ dargestellt, die zur automatischen Anpassung von Text innerhalb einer Tabellenzelle dient.
Da der Platz in den einzelnen Zellen auf dem E-Paper-Display begrenzt ist, sorgt diese Funktion dafür, dass Texte nicht abgeschnitten werden, sondern bei Bedarf mehrzeilig dargestellt werden. Dabei wird der Eingabetext wird mit „text.split()“ in eine Liste von Wörtern umgewandelt. Eine Schleife durchläuft alle Wörter und prüft mit „writer.stringlen()“, ob das Hinzufügen eines weiteren Wortes die maximale Zellenbreite („max_width“) überschreiten würde. Falls die maximale Breite überschritten wird, wird die aktuelle Zeile in die Liste „lines“ aufgenommen. Der Inhalt der Liste beginnt dann eine neue Zeile. Dabei wird nach Worten getrennt. Falls nach der Schleife noch ein Resttext in „current_line“ vorhanden ist, wird dieser als letzte Zeile hinzugefügt. Die Funktion gibt eine Liste von Textzeilen zurück, die anschließend in einer Tabellenzelle dargestellt werden.

```
 99  def draw_table_head(
100      time_column_width,
101      cell_width,
102      header_height,
103      num_days,
104      dummy_display,writer
105  ):
106      # Draw the table header
107      header_labels = ["Zeit"] + ["Montag", "Dienstag", "Mittwoch", "Donnerstag", "Freitag"]
108      x_positions = [0] + [time_column_width + i * cell_width for i in range(num_days)]
109      for i, label in enumerate(header_labels):
110          writer.set_textpos(dummy_display, 5, x_positions[i] + 5)
111          writer.printstring(label)
112      dummy_display.hline(0, header_height - 1, e_paper.width, 0)  # Horizontale Linie unten
```

Abbildung 27 view.py draw_table_head(time_column_width, cell_width,header_height,num_days,dummy_display,writer)

In Abb. 27 wird die Funktion „draw_table_head()“ dargestellt, die für das Zeichnen des Tabellenkopfs in der Raumbelegungsanzeige des ESP32 auf dem E-Paper-Display zuständig ist. Der Tabellenkopf enthält die Zeitspalte und die Wochentage, die als Spaltenüberschriften für die Belegungstabelle dienen. Die Liste „header_labels“ enthält die Titel der einzelnen Spalten, beginnend mit "Zeit" für die erste Spalte und gefolgt von den Wochentagen Montag bis Freitag.

Die X-Koordinaten („x_positions“) werden berechnet, um die Startpositionen der einzelnen Spalten festzulegen. Die erste Spalte (Zeit) beginnt bei 0. Die übrigen Spalten werden gleichmäßig verteilt, wobei jede Spalte um die Breite der Tageszellen („cell_width“) versetzt wird. Mithilfe von „writer.set_textpos()“ werden die Überschriften der Wochentage an den berechneten Positionen auf das E-Paper-Display geschrieben. Eine horizontale Linie („dummy_display.hline()“) wird unter den Spaltenüberschriften gezeichnet, um den Tabellenkopf optisch vom restlichen Stundenplan abzugrenzen.

Die Funktion draw_table_head() sorgt dafür, dass die Spalten korrekt ausgerichtet und gut lesbar sind. Durch die exakte Berechnung der Positionen wird sichergestellt, dass die Spaltenüberschriften korrekt oberhalb der jeweiligen Zeitslots stehen.

```
114  def draw_table_lines(
115      time_column_width,
116      cell_width,cell_height,
117      header_height,
118      total_table_height,
119      num_days,time_slots,
120      dummy_display,
121      writer
122  ):
123      # Draw the table (full-hour slots)
124      for i, time_slot in enumerate(time_slots):
125          y = header_height + i * cell_height  # Position below the header
126          writer.set_textpos(dummy_display, y + 5, 5)
127          writer.printstring(time_slot)
128          dummy_display.hline(0, y, e_paper.width, 0)  # Horizontal line
129
130          # Vertical lines for the weekdays
131          for day_index in range(num_days + 1):
132              x = time_column_width + day_index * cell_width
133              dummy_display.vline(x, header_height, total_table_height, 0)
```

Abbildung 28 view.py draw_table_lines(time_column_width, cell_width,cell_height,header_height,total_table_height,num_days,time_slots,dummy_display, writer)

In Abb. 28 wird die Funktion „draw_table_lines()“ dargestellt, die für das Zeichnen der Tabellenstruktur auf dem E-Paper-Display zuständig ist.

Diese Funktion sorgt dafür, dass die Zeiteinteilung der Raumbelegungsanzeige visuell strukturiert wird, indem sie horizontale und vertikale Linien zur Unterteilung der Tabelle erstellt. Die Funktion iteriert über die „time_slots“, also die Stundenblöcke, die in der Tabelle dargestellt werden. Für jede volle Stunde wird eine horizontale Linie („hline“) gezeichnet, die sich über die gesamte Breite des Displays erstreckt. Die Y-Koordinate („y“) wird anhand der Tabellenhöhe und der Zeilenanzahl berechnet, sodass die Linien exakt an den richtigen Positionen erscheinen. Jede Stunde wird mithilfe von „writer.set_textpos()“ an der linken Seite der Tabelle als Zeiten-Beschriftung eingefügt. Dies sorgt dafür, dass der Nutzer sofort erkennt, welcher Zeitblock zu welcher Stunde gehört. Die Spalten für die Wochentage werden mit vertikalen Linien („vline“) getrennt. Die X-Koordinaten („x“) werden so berechnet, dass die Linien exakt an den Spaltenübergängen der Wochentage erscheinen. Dabei werden „time_column_width“ und „cell_width“ genutzt, um die richtigen

Startpunkte für jede Spalte zu bestimmen. Die Funktion „draw_table_lines()“ stellt sicher, dass die Tabellenstruktur sauber und visuell klar getrennt ist. Durch die horizontale und vertikale Rasterung wird der Stundenplan klar in Zeitblöcke und Wochentage unterteilt, sodass eine intuitive Lesbarkeit und schnelle Orientierung möglich sind.

```python
def draw_schedules(
    time_column_width,
    cell_height,
    cell_width,
    header_height,
    fifteen_minute_slot_height,
    fifteen_minute_slots,
    dummy_display,
    writer,
    json_data
    ):
    # Subjects in the 15-minute slots (within the full-hour slots)
    days = ["Montag", "Dienstag", "Mittwoch", "Donnerstag", "Freitag"]
    for day_index, day in enumerate(days):
        if day not in json_data.get("data", {}):
            continue

        for entry in json_data["data"][day]:
            start_time = entry.get("start_time")
            end_time = entry.get("end_time")
            text = entry.get("text", "")

            # Adjust times by adding 45 minutes
            start_time_adjusted = adjust_time_by_45_minutes(start_time)
            end_time_adjusted = adjust_time_by_45_minutes(end_time)

            if start_time_adjusted not in fifteen_minute_slots or end_time not in fifteen_minute_slots:
                continue

            start_index = fifteen_minute_slots.index(start_time_adjusted)
            end_index = fifteen_minute_slots.index(end_time_adjusted)

             # Calculate the subject position in the 15-minute grid
            x = time_column_width + day_index * cell_width

            # Calculate the subject position within the 15-minute grid
            y_start = header_height + start_index * fifteen_minute_slot_height
            y_end = header_height + end_index * fifteen_minute_slot_height
            block_height = y_end - y_start

             # Always start subjects at the beginning of the time slot (align start time to full slot)
            dummy_display.fill_rect(x, y_start, cell_width, block_height, 0)  # Schwarzer Block

             # Line wrapping for subject text within cells
            text_lines = wrap_text_in_cell(text, cell_width, writer)
            line_height = cell_height // len(text_lines)  # Höhe pro Zeile für den Text
            current_y = y_start + 5
            for line in text_lines:
                writer.set_textpos(dummy_display, current_y, x + 5)
                writer.printstring(line)
                current_y += line_height

            # Display start and end times
            writer.set_textpos(dummy_display, current_y, x + 5)
            writer.printstring(f"{start_time} - {end_time}", invert=True)
```

Abbildung 29 view.py draw_schedules(time_column_width, cell_height, cell_width,header_height, fifteen_minute_slot_height, fifteen_minute_slots, dummy_display, writer, json_data)

In Abb. 29 wird die Funktion „draw_schedules()“ gezeigt, die für das Einzeichnen der Raumbelegungszeiten in die Tabellenstruktur des E-Paper-Displays verantwortlich ist.

Sie sorgt dafür, dass Veranstaltungen und Fächer an den richtigen Zeitpunkten innerhalb des Stundenplans dargestellt werden. Die Funktion verarbeitet die Raumbelegungsdaten für jeden Wochentag. Falls für einen bestimmten Tag keine Daten im „json_data“ vorhanden sind, wird der Tag übersprungen. Für jeden Eintrag werden die Startzeit („start_time“), Endzeit („end_time“) und der Name des Faches oder Veranstaltungstitel („text“) ausgelesen. Die Zeiten werden durch „adjust_time_by_45_minutes()“ angepasst, um sicherzustellen, dass die Belegung exakt im Zeitschema des Stundenplans dargestellt wird, da wie schon erwähnt sie alle aus unbekannten Gründen sonst um 45 Minuten verschoben sind. Danach werden die X-Koordinate („x“) anhand des Wochentags berechnet und die Y-Koordinaten („y_start und y_end“) werden basierend auf den 15-Minuten-Zeitslots bestimmt. Dann werden die Veranstaltungen als schwarze Rechtecke („fill_rect()“) dargestellt, um eine visuelle Abgrenzung im Stundenplan zu schaffen. Die Blockhöhe („block_height“) wird anhand der Dauer der Veranstaltung berechnet. Der Veranstaltungsname wird mit „wrap_text_in_cell()“ umbrochen, um sicherzustellen, dass der Text innerhalb des verfügbaren Platzes bleibt. Die einzelnen Zeilen werden innerhalb des Blocks vertikal verteilt, damit die Schrift lesbar bleibt. Die Start- und Endzeit der Veranstaltung werden unterhalb des Fachnamens in den jeweiligen Zellen ausgegeben, um eine bessere Übersichtlichkeit zu gewährleisten.

Die „draw_schedules()“-Funktion sorgt dafür, dass der Stundenplan klar strukturiert und gut lesbar ist. Durch die exakte Platzierung der Veranstaltungen innerhalb der Tabelle wird sichergestellt, dass alle Kurse und Raumbelegungen übersichtlich auf dem E-Paper-Display erscheinen. Zusätzlich verbessert die automatische Textanpassung und Umbruchberechnung die Darstellung, sodass auch längere Veranstaltungsnamen korrekt in den Zellen angezeigt werden.

```python
def draw_table_with_time(json_data):
    e_paper.init()
    e_paper.clear()  # Clear the display
    print("Neue Tabellendaten empfangen. Aktualisiere Anzeige...")

    # Generate full-hour time slots (for the table)
    time_slots = generate_full_hour_time_slots()

    # Generate 15-minute time slots (for the subjects)
    fifteen_minute_slots = generate_event_time_slots()

    # E-Paper dimensions
    time_column_width = 80
    num_days = 5
    header_height = e_paper.height // 20  # Header height (5% of the display height)
    total_table_height = e_paper.height - header_height   # Remaining space for the table
    cell_width = (e_paper.width - time_column_width) // num_days
    cell_height = total_table_height // len(time_slots)  # Height of each full-hour slot

    # Calculate the 15-minute slot heights
    fifteen_minute_slot_height = cell_height // 4  # Jede Stunde in 4 15-Minuten-Slots unterteilen

    # Create the complete framebuffer for the table
    framebuffer = bytearray(e_paper.height * e_paper.width // 8)
    dummy_display = DummyDisplay(framebuffer, e_paper.width, e_paper.height, framebuf.MONO_HLSB)
    dummy_display.fill(1)  # White background

    font = load_font("myfont")
    if font is None:
        print("Fehler: Schriftart konnte nicht geladen werden.")
        font = minecraft10 # Backup font
    writer = Writer(dummy_display, font)

    draw_table_head(
        time_column_width,
        cell_width,
        header_height,
        num_days,
        dummy_display,
        writer
        )
    draw_table_lines(
        time_column_width,
        cell_width,
        cell_height,
        header_height,
        total_table_height,
        num_days,
        time_slots,
        dummy_display,
        writer
    )
    draw_schedules(
        time_column_width,
        cell_height,
        cell_width,
        header_height,
        fifteen_minute_slot_height,
        fifteen_minute_slots,
        dummy_display,
        writer,
        json_data
    )

    # Sende den Framebuffer an das Display
    e_paper.display_frame(framebuffer)

    print("Display erfolgreich aktualisiert.")
    e_paper.sleep()
    gc.collect()

```

Abbildung 30 view.py draw_table_with_time(json_data)

In Abb. 30 wird die Funktion „draw_table_with_time()“ gezeigt, die die gesamte Darstellung des Raumbelegungsplans auf dem E-Paper-Display steuert.
Sie ruft mehrere Unterfunktionen auf, um die Struktur der Tabelle zu zeichnen, den Tabellenkopf zu setzen, die Rasterlinien zu erstellen und die Raumbelegungen korrekt einzutragen.

Das E-Paper-Display wird initialisiert („e_paper.init()“) und anschließend das aktuell angezeigte Bild gelöscht (e_paper.clear()), um eine neue Anzeige vorzubereiten.

Die Zeiten für volle Stunden („generate_full_hour_time_slots()“) und die 15-Minuten-Slots („generate_event_time_slots()“) werden erstellt. Diese Zeitslots dienen als Grundlage für die tabellarische Darstellung der Raumbelegungen. Spaltenbreiten und Zeilenhöhen werden auf Basis der E-Paper-Auflösung bestimmt. Die Header-Höhe („header_height“) wird auf 5% der gesamten Bildschirmhöhe gesetzt. Die Höhe der Tabellenzellen wird basierend auf der Anzahl der Stunden- und 15-Minuten-Blöcke berechnet. Ein Framebuffer („dummy_display“) wird angelegt, um die komplette Darstellung vor dem Zeichnen auf dem Display zu speichern. Der Hintergrund wird weiß gefüllt („dummy_display.fill(1)“). Falls die primäre Schriftart nicht gefunden wird, wird eine Ersatzschrift („minecraft10“) verwendet. Danach zeichnet „draw_table_head()“ den Tabellenkopf mit den Wochentagen und der Zeitspalte. Daraufhin erstellt „draw_table_lines()“ die horizontale und vertikale Rasterstruktur für die Stunden- und Tageszellen und anschließend fügt „draw_schedules()“ die Raumbelegungsblöcke in die Tabelle ein, basierend auf den empfangenen JSON-Daten. Der Framebuffer wird auf das E-Paper-Display übertragen („e_paper.display_frame(framebuffer)“), um die finale Darstellung zu aktualisieren.

Nach der Aktualisierung wird das E-Paper-Display in den Sleep-Modus versetzt („e_paper.sleep()“), um Energie zu sparen. Der Garbage Collector („gc.collect()“) wird aufgerufen, um Speicher freizugeben und eine effiziente Nutzung der ESP32-Ressourcen sicherzustellen.

Die Funktion „draw_table_with_time()“ steuert die gesamte visuelle Darstellung des Stundenplans auf dem ESP32.
Durch den modularen Aufbau mit Unterfunktionen bleibt der Code übersichtlich und ist gut wartbar.

5.3 Server-Programmierung

Der Server des digitalen Raumbelegungssystems übernimmt eine zentrale Rolle in der gesamten Architektur. Er ist verantwortlich für die Verarbeitung von Anfragen der ESP32-Geräte, das Abrufen und Analysieren der aktuellen Raumbelegungspläne sowie die Bereitstellung dieser Daten über das MQTT-Protokoll. Darüber hinaus stellt der Server eine Over-the-Air (OTA)-Update-Funktionalität bereit, die über Node-RED gesteuert wird, um Firmware-Updates an die ESP32-Geräte zu verteilen.

Dieses Kapitel wird sich dabei nur mit dem neuen Code beschäftigen. Der Code von der OTA-Funktionalität wurde nicht verändert und entspricht damit dem Code von Till Giesas Projekt (vgl. Giesa, 2024). Genauso wie der Code vom WLAN-Fingerprinting bereits auf einem Server im HTW-Netzwerk läuft und auch nicht hier behandelt wird (vgl. Völkers 2024).

5.3.1 Schedule_server.py

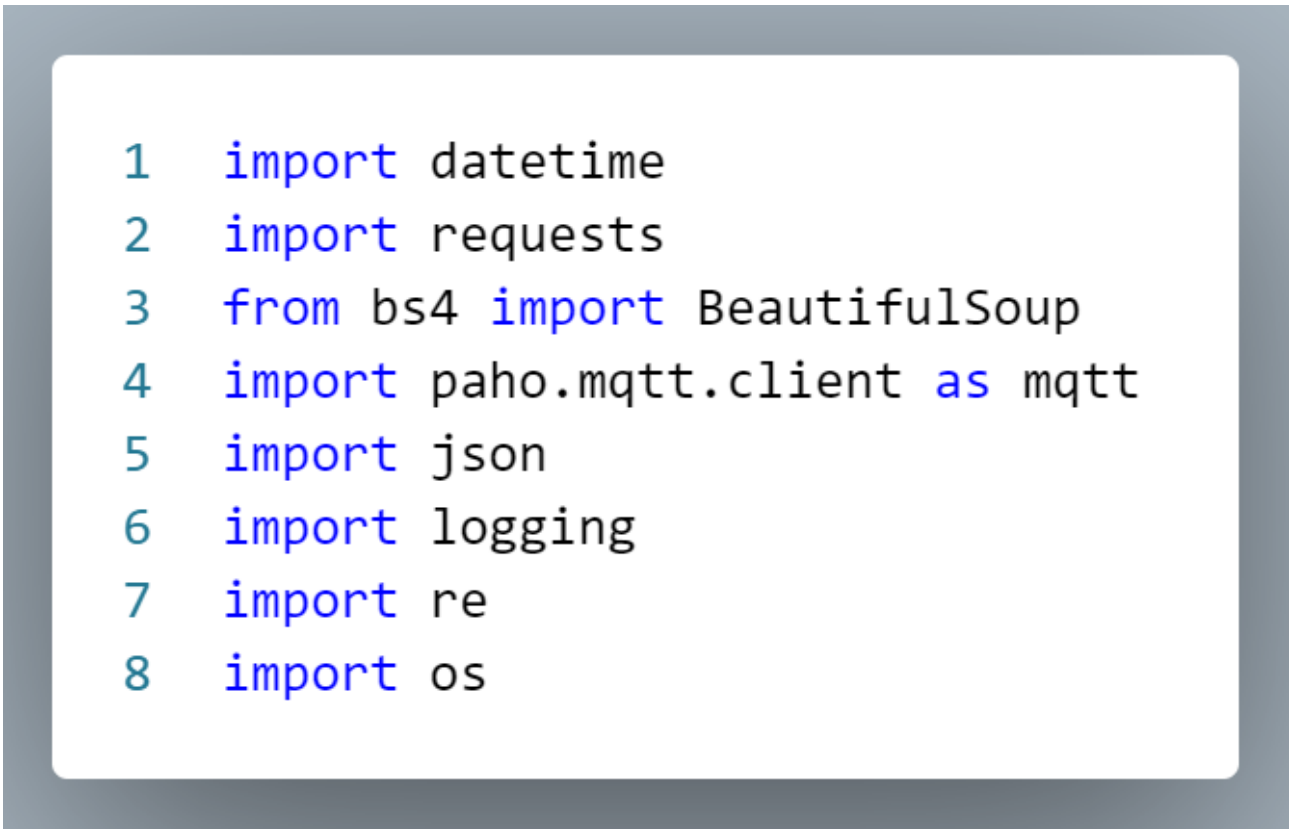


Abbildung 31 schedule_server.py Imports

In Abb. 31 sind die Importe des Servers dargestellt, die für die Verarbeitung und Bereitstellung der Raumbelegungsdaten erforderlich sind. Diese Bibliotheken ermöglichen die Kommunikation mit

der Universitätsplattform, das Extrahieren relevanter Informationen, die Speicherung der Daten sowie die Interaktion mit den ESP32-Geräten über MQTT.

Das Modul „datetime“ dient der Verwaltung von Datum und Uhrzeit. Es wird genutzt, um beispielsweise die aktuelle Kalenderwoche zu berechnen oder Zeitstempel für Log-Dateien zu generieren. Dies ist essenziell für die korrekte Zuordnung der Raumbelegungspläne, da sich diese wöchentlich ändern.

Mit „requests“ wird die HTTP-Kommunikation mit der Universitätsplattform ermöglicht. Der Server sendet eine GET-Anfrage an die Webseite, um die aktuellen Raumbelegungsdaten abzurufen. Die erhaltenen HTML-Daten bilden die Grundlage für das nachfolgende Webscraping.

Das Modul „BeautifulSoup“ aus der „bs4“-Bibliothek wird für das Parsing und Extrahieren der relevanten Daten aus dem HTML-Code verwendet. Dabei werden spezifische HTML-Elemente, wie Tabellenzellen („<td>“) oder Links („<a>“), gefiltert, um die relevanten Informationen zu den Raumbelegungen zu extrahieren.

Für die Kommunikation zwischen Server und ESP32-Geräten kommt das MQTT-Protokoll zum Einsatz, das durch das Modul „paho.mqtt.client“ bereitgestellt wird. Der Server agiert als MQTT-Broker, empfängt Anfragen von ESP32-Geräten und sendet die zugehörigen Raumbelegungspläne als JSON-Daten über das Netzwerk zurück.

Das „json“-Modul wird verwendet, um die extrahierten Raumpläne in einer strukturierten JSON-Datei zu speichern. Diese Datei dient als Zwischenspeicher, um die Daten schnell abrufen zu können, ohne erneut eine Anfrage an die Universitätsplattform senden zu müssen.

Das Logging-Modul („logging“) sorgt für eine detaillierte Protokollierung aller wichtigen Ereignisse und Fehler. Dadurch können Fehlermeldungen, erfolgreiche MQTT-Kommunikationen oder Webscraping-Probleme dokumentiert werden, um eine spätere Fehleranalyse und Wartung des Systems zu erleichtern.

Das Modul „re“ für reguläre Ausdrücke verwendet. Dies ermöglicht die extraktion spezifischer Informationen, wie z. B. Uhrzeiten, Raumnummern oder Gruppenkennungen aus den HTML-Daten. Durch den Einsatz regulärer Ausdrücke kann der Server die Struktur der Belegungspläne effizient analysieren und formatierte Informationen extrahieren.

Zum Schluss wird „os“ importiert, um mit Betriebssystem-Funktionen zu arbeiten, in diesem Fall, um Zugriff auf Datei- und Verzeichnispfade zu haben.

Diese Module bilden die technische Grundlage des Servers und ermöglichen eine automatisierte, effiziente und zuverlässige Verarbeitung der Raumbelegungsdaten.

```
11  # Logging configuration
12  logging.basicConfig(level=logging.INFO, format='%(asctime)s - %(levelname)s - %(message)s')
13
14  SCRIPT_DIR = os.path.dirname(os.path.abspath(__file__))
15
16  # MQTT configuration
17  MQTT_USERNAME = "mobilelab"
18  MQTT_PASSWORD = "mobilelab"
19  MQTT_PORT = 1883
20  ROOM_JSON_FILE = "room_data.json"
21  JSON_OUTPUT_FILE = os.path.join(SCRIPT_DIR, "schedule_server_data.json")
```

Abbildung 32 schedule_server.py Globale Variabeln

In Abb. 32 sind die Globalen Variablen für das Logging und die MQTT-Kommunikation des Servers dargestellt. Diese sind die Einstellungen für die Überwachung des Serverstatus, die Speicherung von Fehlern und Ereignissen sowie die sichere Kommunikation mit den ESP32-Geräten über das MQTT-Protokoll.

Die Logging-Konfiguration („logging.basicConfig()“) definiert das Protokollierungsniveau und das Format der Log-Einträge. Das Logging-Level „INFO“ sorgt dafür, dass alle wichtigen Informations-, Warn- und Fehlermeldungen erfasst werden. Das Format '%(asctime)s - %(levelname)s - %(message)s' stellt sicher, dass jeder Log-Eintrag mit einem Zeitstempel versehen ist, um eine nachvollziehbare Ereigniskette zu erhalten. Dies erleichtert die Fehlersuche und Wartung, da sämtliche Aktivitäten des Servers detailliert dokumentiert werden.

„SCRIPT_DIR = os.path.dirname(os.path.abspath(__file__))“ wird verwendet, um das Verzeichnis des aktuellen Skripts zu bestimmen. Sie stellt sicher, dass das Skript unabhängig vom aktuellen Arbeitsverzeichnis zuverlässig auf Dateien und Verzeichnisse zugreifen kann.

Für die MQTT-Kommunikation sind mehrere zentrale Konfigurationswerte definiert. „MQTT_USERNAME“ und „MQTT_PASSWORD“ enthalten die Anmeldedaten für den MQTT-Broker, um eine sichere Authentifizierung der ESP32-Geräte zu ermöglichen. „MQTT_PORT = 1883“ gibt an, dass Port 1883 für die Kommunikation genutzt wird, da dies der Standard-Port für unverschlüsselte MQTT-Verbindungen ist. Die Datei „ROOM_JSON_FILE“ speichert die Zuordnung von Raumnummern zu Raum-IDs, die für die Anfragen der ESP32-Geräte benötigt

werden. Das „JSON_OUTPUT_FILE“ speichert die extrahierten Stundenplandaten aus dem Web Scraping, um diese anschließend über MQTT an die ESP32-Geräte zu senden.

Diese Konfiguration stellt sicher, dass der Server zuverlässig arbeitet, MQTT-Daten sicher verarbeitet und eine strukturierte Protokollierung für Fehleranalysen bereitstellt.

```
20  # Load ROOM_DICT
21  def load_room_dict(json_file):
22      try:
23          with open(json_file, 'r', encoding='utf-8') as file:
24              return {entry["room_number"]: entry["room_id"] for entry in json.load(file)}
25      except Exception as e:
26          logging.error(f"Fehler beim Laden der JSON-Datei: {e}")
27          return {}
```

Abbildung 33 schedule_server.py load_room_dict(json_file)

In Abb. 33 wird die Funktion „load_room_dict()“ gezeigt, die für das Laden der Raumnummern-zu-ID-Zuordnung aus einer JSON-Datei verantwortlich ist. Diese Zuordnung ist essenziell, da die Universitätsplattform für die Abfrage von Raumplänen interne Raum-IDs anstelle von Raumnummern verwendet. Die Funktion öffnet die JSON-Datei („json_file“), die zuvor durch das Hilfsprogramm zur Raumextraktion erstellt wurde, und liest die Zuordnung zwischen Raumnummern und Raum-IDs aus. Dabei wird ein Dictionary („dict“) erstellt, in dem die Raumnummern als Schlüssel („room_number“) und die dazugehörigen Raum-IDs als Werte („room_id“) gespeichert werden. Falls beim Öffnen oder Verarbeiten der JSON-Datei ein Fehler auftritt, wird dieser über „logging.error()“ protokolliert, und die Funktion gibt ein leeres Dictionary {} zurück. Dies verhindert, dass der Server mit fehlerhaften oder unvollständigen Daten arbeitet und ermöglicht eine einfache Fehleranalyse. Diese Funktion stellt sicher, dass der Server die empfangenen MQTT-Anfragen den korrekten Räumen zuordnen kann, um die passenden Stundenpläne bereitzustellen.

```
30  ROOM_DICT = load_room_dict(ROOM_JSON_FILE)
```

Abbildung 34 schedule_server.py ROOM_DICT

In Abb. 34 wird die Initialisierung der Raum-zu-ID-Zuordnung („ROOM_DICT“) dargestellt. Diese Variable enthält ein Dictionary, dass die Raumnummern den zugehörigen internen Raum-IDs

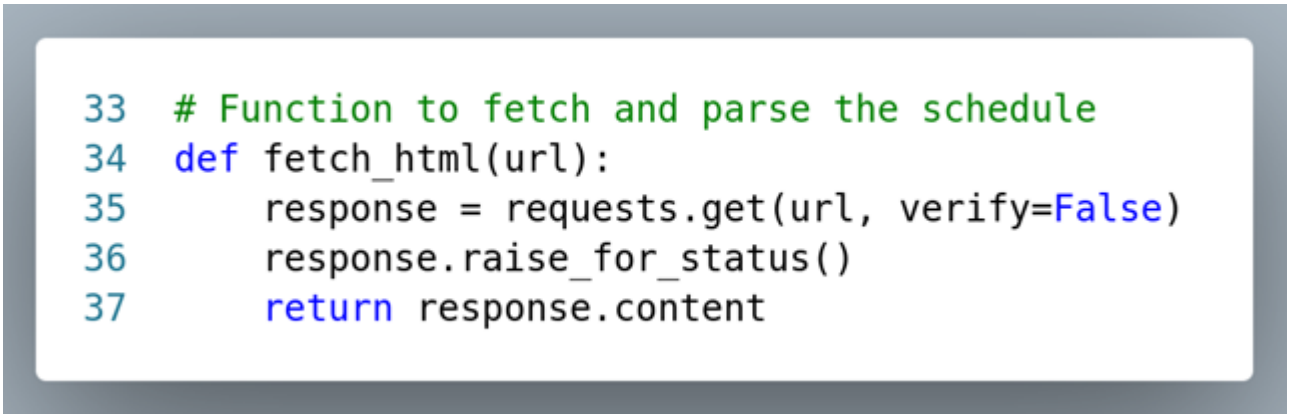


Abbildung 35 schedule_server.py fetch_html (url)

zuordnet. Die Werte für „ROOM_DICT“ werden durch einen Aufruf der Funktion „load_room_dict(ROOM_JSON_FILE)“ aus einer JSON-Datei geladen.

In Abb. 35 wird die Funktion „fetch_html() “ dargestellt, die für das Abrufen der Stundenplandaten von der Universitätsplattform über eine HTTP-Anfrage zuständig ist.
Diese Funktion bildet die erste Stufe des Webscraping-Prozesses, indem sie die Webseite mit den Raumplänen abruft und den HTML-Inhalt zur weiteren Verarbeitung zurückgibt.

Die Funktion verwendet das „requests.get()“-Modul, um eine HTTP-GET-Anfrage an die übergebene URL („url“) zu senden. Dabei wird das Argument „verify=False“ genutzt, um SSL-Zertifikatsüberprüfungen zu deaktivieren, da der Server ein selbstsignierte Zertifikate verwendet, das als unsicher eingestuft wird.

Nach der erfolgreichen Verbindung überprüft die Funktion mit „response.raise_for_status()“, ob die Anfrage korrekt verarbeitet wurde. Falls der Server eine Fehlermeldung (z. B. 404 oder 500) zurückgibt, wird eine Exception ausgelöst, die eine weitere Verarbeitung verhindert.

Falls keine Fehler auftreten, wird der gesamte HTML-Quellcode der Webseite als „response.content“ zurückgegeben, sodass die Daten anschließend mit „BeautifulSoup“ weiterverarbeitet werden können.

Diese Funktion stellt sicher, dass die aktuellen Raumbelegungsdaten zuverlässig abgerufen werden, bevor sie weiter analysiert und verarbeitet werden.

```python
def parse_schedule(html):
    soup = BeautifulSoup(html, 'html.parser')
    table = soup.find('table', {'border': '1'})
    if not table:
        raise ValueError("Keine Tabelle gefunden.")

    schedule = {}
    for cell in table.find_all('td', class_='plan2'):
        title_tag = cell.find('a', title=True)
        title = title_tag['title'] if title_tag else None
        link = title_tag['href'] if title_tag else None

        notiz_rows = cell.find_all('td', class_='notiz')
        if not notiz_rows or len(notiz_rows) < 2:
            continue

        time_info = notiz_rows[0].get_text(strip=True)
        match = re.match(r"(\w+),\s*(\d{2}:\d{2})\s*-\s*(\d{2}:\d{2})\s*,\s*(.*?),\s*(.*)", time_info)
        if not match:
            continue

        day, start_time, end_time, turnus, group = match.groups()
        hour_type = notiz_rows[1].get_text(strip=True)

        entry = {
            "text": title,
            "link": link,
            "start_time": start_time,
            "end_time": end_time,
            "type": hour_type,
            "turnus": turnus,
            "group": group
        }

        if day not in schedule:
            schedule[day] = []
        schedule[day].append(entry)

    return schedule
```

Abbildung 36 schedule_server.py parse_schedule(html)

In Abb. 36 wird die Funktion „parse_schedule()“ gezeigt, die für das Extrahieren und Strukturieren der Raumbelegungsdaten aus dem HTML-Quelltext verantwortlich ist.
Diese Funktion nutzt „BeautifulSoup“, um den zuvor abgerufenen HTML-Code der Universitätsplattform zu analysieren und die relevanten Daten in einer strukturierten Form zu speichern. Zuerst wird der übergebene HTML-Quellcode mit BeautifulSoup geparst, sodass die Daten gezielt durchsucht werden können. Daraufhin wird die Tabelle mit den Stundenplandaten anhand des HTML-Attributes „border="1"“ identifiziert. Falls keine passende Tabelle gefunden wird, löst die Funktion eine Fehlermeldung („ValueError“) aus, um ungültige Anfragen frühzeitig abzufangen. Danach sucht die Funktion gezielt nach Tabellenzellen („<td>“) mit der Klasse "plan2", die die Veranstaltungsinformationen enthalten. Falls die Tabellenzelle einen Hyperlink („<a>“) mit einem title-Attribut enthält, wird der Name der Veranstaltung („title“) und der zugehörige Link („href“) extrahiert. Die Veranstaltungszeiten und Zusatzinformationen werden aus

Tabellenzellen mit der Klasse „notiz“ ausgelesen. Die erste Notiz-Zelle enthält Informationen zum Tag, Uhrzeit und Turnus, die mit einem regulären Ausdruck („re.match()“) extrahiert werden. Falls ein Match gefunden wird, werden folgende Werte gespeichert: Tag („day“), Startzeit („start_time“), Endzeit („end_time“), Turnus („turnus“) (z. B. wöchentlich, zweiwöchentlich), Gruppe („group“).

Falls der Tag („day“) noch nicht im „schedule“-Dictionary vorhanden ist, wird er als leere Liste initialisiert. Die extrahierten Veranstaltungsdetails werden als neues Dictionary („entry“) gespeichert und dem jeweiligen Tag hinzugefügt. Die gesammelten Veranstaltungsdaten werden als verschachtelte Datenstruktur zurückgegeben, in der jeder Wochentag eine Liste mit Veranstaltungen und deren Details enthält.

```
81  # Function to save JSON data
82  def save_json_data(data, filename):
83      try:
84          with open(filename, 'w', encoding='utf-8') as file:
85              json.dump(data, file, ensure_ascii=False, indent=4)
86          logging.info(f"Daten erfolgreich in '{filename}' gespeichert.")
87      except Exception as e:
88          logging.error(f"Fehler beim Speichern der JSON-Daten: {e}")
```

Abbildung 37 schedule_server.py save_json_data (data,filename)

In Abb. 37 wird die Funktion „save_json_data()“ dargestellt, die für das Speichern der extrahierten Raumbelegungsdaten in einer JSON-Datei verantwortlich ist.
Diese Funktion stellt sicher, dass die Stundenplandaten nach dem Webscraping dauerhaft gespeichert werden, sodass sie für die spätere Verarbeitung und Verteilung über MQTT zur Verfügung stehen. Die Funktion erwartet zwei Parameter. „data“ die zu speichernden Daten (in diesem Fall ein Dictionary mit Raumbelegungsinformationen). „filename“ der Dateiname, unter dem die JSON-Daten gespeichert werden. Die Datei wird im Schreibmodus („w“) geöffnet und mit UTF-8-Kodierung („encoding='utf-8'“) gespeichert, um Sonderzeichen korrekt darzustellen. Die übergebenen Daten werden mit „json.dump()“ in eine formatierte JSON-Datei geschrieben. Die Option „ensure_ascii=False“ sorgt dafür, dass Umlaute und Sonderzeichen korrekt gespeichert werden. Mit „indent=4“ wird das JSON-Format lesbar strukturiert, indem Einrückungen für eine bessere Übersicht hinzugefügt werden. Falls die Speicherung erfolgreich ist, wird eine Log-Nachricht („logging.info()“) generiert, die den Dateinamen der gespeicherten Datei enthält. Dadurch wird dokumentiert, dass der Speichervorgang erfolgreich abgeschlossen wurde. Falls ein

Fehler während der Speicherung auftritt, wird dieser abgefangen („except Exception as e“) und eine Fehlermeldung („logging.error()“) im Log-System gespeichert. Dies verhindert, dass das Programm unbehandelt abstürzt, und ermöglicht eine schnelle Identifikation von Problemen.

```
91  # Function to send MQTT data
92  def send_mqtt_data(broker, topic, message):
93      try:
94          # Add the method `room_schedule` to the message
95          enriched_message = json.dumps({
96              "method": "room_schedule",
97              "data": json.loads(message)  # The original schedule data
98          }, ensure_ascii=False)
99
100         client = mqtt.Client()
101         client.username_pw_set(username=MQTT_USERNAME, password=MQTT_PASSWORD)
102         client.connect(broker, MQTT_PORT, 60)
103         client.publish(topic, enriched_message)
104         client.disconnect()
105         logging.info(f"Nachricht erfolgreich an Topic '{topic}' gesendet: {enriched_message}")
106     except Exception as e:
107         logging.error(f"Fehler beim Senden der MQTT-Nachricht: {e}")
```

Abbildung 38 schedule_server.py send_mqtt_dataq (broker, topic, message)

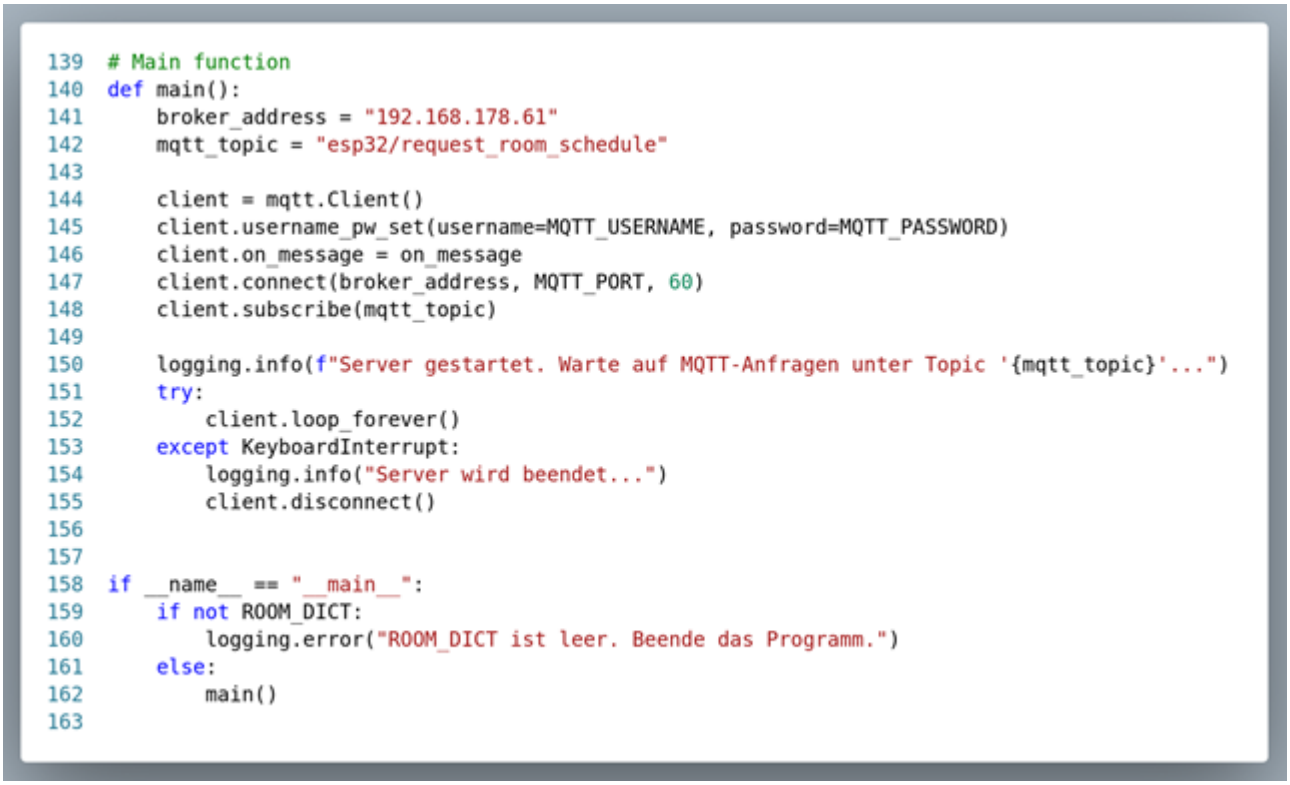
In Abb. 38 wird die Funktion „send_mqtt_data()“ dargestellt, die für die Kommunikation zwischen dem Server und den ESP32-Geräten über das MQTT-Protokoll verantwortlich ist.
Diese Funktion ermöglicht es dem Server, die zuvor extrahierten Stundenplandaten gezielt an die angeforderten Geräte zu senden. Die übermittelten Stundenplandaten („message“) werden in eine erweiterte JSON-Struktur umgewandelt. Dabei wird das Feld "method": "room_schedule" hinzugefügt, um die Nachricht als Stundenplan-Update zu kennzeichnen. Dies geschieht durch „json.dumps()“, wobei das ursprüngliche „message“-Objekt in das neue „data“-Feld eingefügt wird. Ein neuer MQTT-Client („mqtt.Client()“) wird erstellt. Die Authentifizierungsdaten („MQTT_USERNAME“, „MQTT_PASSWORD“) werden gesetzt, um eine sichere Verbindung mit dem Broker herzustellen. Der Client stellt eine Verbindung zum angegebenen Broker („broker“) über den zuvor definierten Port („MQTT_PORT“) her. Die formatierte JSON-Nachricht wird an das angegebene MQTT-Topic („topic“) gesendet. Anschließend trennt der Client die Verbindung, um Systemressourcen zu sparen. Falls die Nachricht erfolgreich gesendet wurde, wird dies mit „logging.info()“ protokolliert. Falls ein Fehler auftritt, wird dieser mit „logging.error()“ erfasst, um eine spätere Fehlersuche zu erleichtern.

```python
# MQTT callback
def on_message(client, userdata, message):
    room_number = message.payload.decode('utf-8')
    logging.info(f"Raumanfrage erhalten: {room_number}")

    room_id = ROOM_DICT.get(room_number)
    if not room_id:
        logging.warning(f"Raum '{room_number}' ist nicht im Dictionary definiert.")
        return

    # Determine current year and calendar week
    current_year, current_week, _ = datetime.date.today().isocalendar()
    week_str = f"{current_week}_{current_year}"
    print(week_str)
    url = f"https://lsf.htw-berlin.de/qisserver/rds?state=wplan&raum.rgid={room_id}&week={week_str}&act=Raum&pool=Raum&show=plan&P.vx=mittel&fil=plu&P.subc=plan"

    try:
        html = fetch_html(url)
        schedule = parse_schedule(html)

        save_json_data(schedule, JSON_OUTPUT_FILE)

        schedule_json = json.dumps(schedule, ensure_ascii=False)
        mqtt_topic = f"esp32/schedule/{room_number}"
        send_mqtt_data(client._host, mqtt_topic, schedule_json)
    except Exception as e:
        logging.error(f"Fehler bei der Verarbeitung der Tabelle: {e}")
```

Abbildung 39 schedule_server.py on_message (client,userdata,message)

In Abb. 39 wird die Funktion „on_message()“ dargestellt, die als MQTT-Callback dient und für die Verarbeitung von Raumanfragen der ESP32-Geräte verantwortlich ist.
Diese Funktion wird immer dann aufgerufen, wenn der Server über MQTT eine Nachricht auf dem entsprechenden Topic empfängt. Die empfangene Nachricht („message.payload“) enthält die Raumnummer, für die der Belegungsplan angefordert wird. Die Nachricht wird mit „decode('utf-8')“ in eine lesbare Zeichenkette umgewandelt und die Anfrage wird im Log-System („logging.info()“) protokolliert, um eine Nachverfolgbarkeit der Anfragen zu ermöglichen. Danach sucht Funktion in „ROOM_DICT“, ob die empfangene Raumnummer („room_number“) eine zugehörige Raum-ID („room_id“) besitzt. Falls keine Zuordnung gefunden wird, wird eine Warnung („logging.warning()“) ausgegeben, und die Funktion beendet sich vorzeitig. Falls eine zugehörige Raum-ID („room_id“) existiert, ruft die Funktion mit „datetime.date.today().isocalendar()“ die aktuelle Kalenderwoche und das Jahr ab. Diese Werte werden für die dynamische Erstellung der URL zur Raumplanabfrage benötigt. Die generierte URL enthält die Raum-ID und die aktuelle Kalenderwoche, sodass der Server den tagesaktuellen Stundenplan abrufen kann. Diese ruft er über „fetch_html(url)“ ab und werden dann mit „parse_schedule(html)“ analysiert, um die Raumbelegung strukturiert als Dictionary zu speichern.

Daraufhin werden die extrahierten Daten mit „save_json_data(schedule, JSON_OUTPUT_FILE)“ lokal gespeichert. Die aufbereiteten Raumplandaten werden in eine JSON-Nachricht umgewandelt und mit „send_mqtt_data()“ an das ESP32-Gerät gesendet, indem sie auf das dynamisch generierte Topic „esp32/schedule/{room_number}“ veröffentlicht werden. Falls während des Abrufs, des Parsings oder der MQTT-Übertragung ein Fehler auftritt, wird dieser protokolliert („logging.error()“), um eine spätere Analyse zu ermöglichen.



```python
# Main function
def main():
    broker_address = "192.168.178.61"
    mqtt_topic = "esp32/request_room_schedule"

    client = mqtt.Client()
    client.username_pw_set(username=MQTT_USERNAME, password=MQTT_PASSWORD)
    client.on_message = on_message
    client.connect(broker_address, MQTT_PORT, 60)
    client.subscribe(mqtt_topic)

    logging.info(f"Server gestartet. Warte auf MQTT-Anfragen unter Topic '{mqtt_topic}'...")
    try:
        client.loop_forever()
    except KeyboardInterrupt:
        logging.info("Server wird beendet...")
        client.disconnect()


if __name__ == "__main__":
    if not ROOM_DICT:
        logging.error("ROOM_DICT ist leer. Beende das Programm.")
    else:
        main()

```

Abbildung 40 schedule_server.py main ()

In Abb. 40 wird die Funktion „main()“ dargestellt, die für das Starten des Servers und die Verwaltung der MQTT-Kommunikation verantwortlich ist.
Diese Funktion sorgt dafür, dass der Server kontinuierlich auf Anfragen der ESP32-Geräte wartet, eingehende Nachrichten verarbeitet und die aktuellen Raumbelegungspläne bereitstellt. Die Variable „broker_address“ enthält die IP-Adresse des MQTT-Brokers, mit dem sich der Server verbinden soll. Die Variable „mqtt_topic“ definiert das MQTT-Topic („esp32/request_room_schedule“), auf dem der Server auf Anfragen von ESP32-Geräten wartet.

Ein neuer MQTT-Client („mqtt.Client()“) wird erstellt. Über „client.username_pw_set()“ werden die Authentifizierungsdaten („MQTT_USERNAME, MQTT_PASSWORD“) gesetzt, um eine sichere Verbindung mit dem Broker herzustellen. Danach verknüpft die Funktion „client.on_message = on_message“ den Empfang von MQTT-Nachrichten mit der„ on_message()“-Funktion. Dadurch wird jede empfangene Nachricht automatisch verarbeitet, sobald sie auf dem definierten Topic eintrifft. Der Client stellt mit client.connect(„broker_address, MQTT_PORT, 60“) eine Verbindung zum MQTT-Broker her. Anschließend wird mit „client.subscribe(mqtt_topic)“ das definierte MQTT-Topic („esp32/request_room_schedule“) abonniert, sodass der Server auf Anfragen von ESP32-Geräten reagieren kann. Mit „client.loop_forever()“ wird eine endlose Schleife gestartet, die den Server dauerhaft auf Nachrichten warten lässt. Dadurch bleibt der Server

kontinuierlich aktiv und verarbeitet eingehende MQTT-Anfragen in Echtzeit. Falls der Server manuell mit Strg + C („KeyboardInterrupt“) beendet wird, wird dies mit „logging.info("Server wird beendet...")“ dokumentiert. Anschließend wird mit „client.disconnect()“ die Verbindung zum MQTT-Broker sauber getrennt, um einen geordneten Shutdown zu ermöglichen. Im „if __name__ == "__main__"“-Block wird überprüft, ob die Raum-zu-ID-Zuordnung („ROOM_DICT“) erfolgreich geladen wurde. Falls „ROOM_DICT“ leer ist, wird dies mit „logging.error("ROOM_DICT ist leer. Beende das Programm.")“ dokumentiert und der Server nicht gestartet, um Fehler zu vermeiden. Falls die Raumliste erfolgreich geladen wurde, wird die „main()“-Funktion aufgerufen und der Server gestartet.

5.3.2 Roomlist_helper.py

Dieses Skript dient dazu, eine vollständige Liste aller verfügbaren Räume der Universitätsplattform zu extrahieren und als JSON-Datei („room_data.json“) zu speichern.
Die Daten werden später vom Hauptserver (schedule_server.py) genutzt, um MQTT-Anfragen der ESP32-Geräte mit den richtigen Raum-IDs zu verarbeiten.

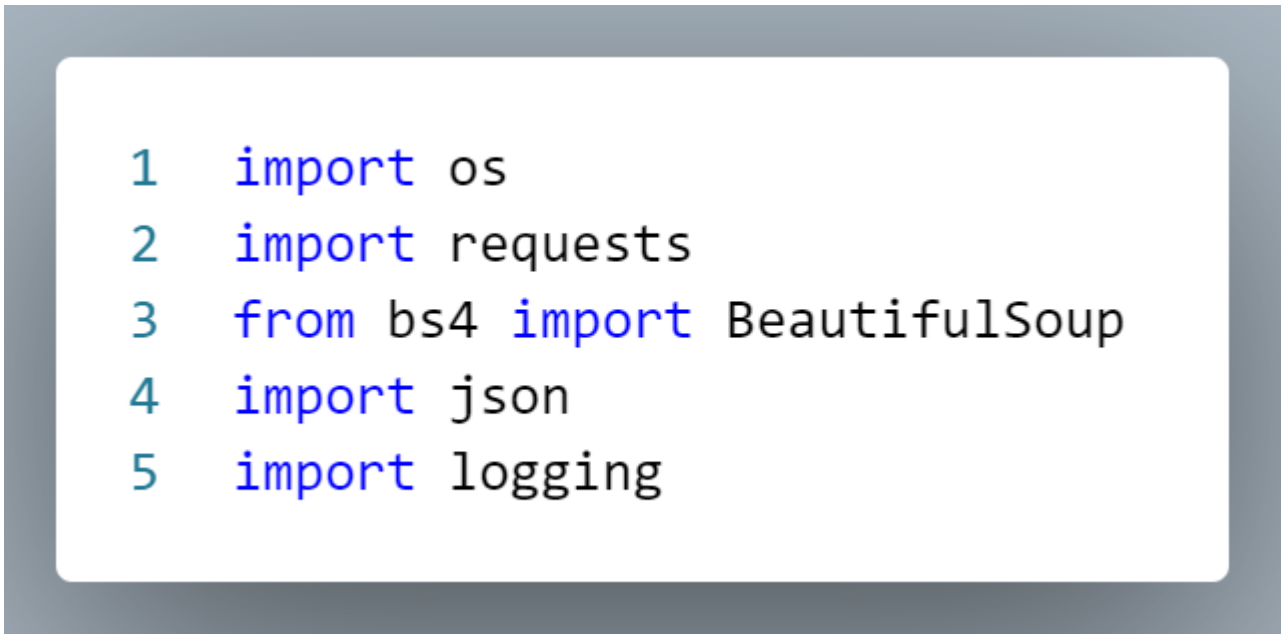


Abbildung 41 roomlist_helper.py Imports

In Abb. 41 sind die Importe für das Hilfsmodul roomlist_helper.py dargestellt. Diese Bibliotheken ermöglichen das Abrufen, Verarbeiten und Speichern der Raumdaten, die später vom Hauptserver für die MQTT-Kommunikation genutzt werden. „os“ wird wieder importiert, um mit Betriebssystem-Funktionen zu arbeiten und um Zugriff auf Datei- und Verzeichnispfade zu haben.

„requests“ wird für die HTTP-Kommunikation mit der Universitätsplattform verwendet. Das Skript sendet eine GET-Anfrage, um den HTML-Code der Raumübersicht abzurufen, der anschließend weiterverarbeitet wird. “BeautifulSoup“ wird für das Webscraping und HTML-Parsing genutzt. Erlaubt eine strukturierte Extraktion der Raumnummern und Raum-IDs aus der Webseite.
„json“ dient zur Speicherung der extrahierten Raumdaten in einer JSON-Datei („room_data.json“). „json“ ermöglicht eine einheitliche und leicht lesbare Speicherung, die für den Hauptserver und die ESP32-Geräte optimiert ist. „logging“ wird verwendet, um alle wichtigen Ereignisse und Fehler während der Laufzeit des Skripts zu dokumentieren. Falls keine Räume gefunden werden oder eine HTTP-Anfrage fehlschlägt, wird dies protokolliert, um eine spätere Fehleranalyse zu erleichtern. Die Protokolle werden in einer separaten Datei („room_data_log.txt“) gespeichert, sodass das Skript jederzeit überwacht werden kann.

```
7  script_dir = os.path.dirname(os.path.abspath(__file__))
8  log_file_path = os.path.join(script_dir, "room_data_log.txt")
9  logging.basicConfig(filename=log_file_path, level=logging.INFO, format='%(asctime)s - %(message)s')
10
11 BASE_URL = "https://lsf.htw-berlin.de/qisserver/rds?state=wplan&act=Raum&pool=Raum&show=plan&P.subc=plan"
12
13 room_data = []
```

Abbildung 42 roomlist_helper.py Globale Variabeln

In Abb. 42 werden die Logging-Konfiguration, die Basis-URL für das Webscraping sowie die Datenstruktur für die Raumliste („room_data“) dargestellt.
Diese Parameter sind essenziell für die Überwachung des Skriptverlaufs, den Abruf der Raumliste von der Universitätsplattform und die temporäre Speicherung der extrahierten Daten.
„script_dir = os.path.dirname(os.path.abspath(__file__))“ und“ log_file_path = os.path.join(script_dir, "room_data_log.txt")“ sorgen wieder dafür das die Dateien unabhängig vom aktuellen Arbeitsverzeichnis immer im selben Verzeichnis wie das Skript gespeichert oder geöffnet wird.
„logging.basicConfig(...)“ ist die Konfiguration des Logging-Systems.
„filename='room_data_log.txt'“ sorgt dafür das alle Log-Nachrichten in der Datei room_data_log.txt gespeichert, sodass Fehler und Ereignisse später analysiert werden können.
„level=logging.INFO“ gibt an das, dass Logging-Level auf INFO gesetzt ist, wodurch alle

wichtigen Informationen, Warnungen und Fehler protokolliert werden. „format='%(asctime)s - %(message)s'“ sorgt dafür das jede Log-Nachricht enthält einen Zeitstempel (asctime), um nachzuvollziehen, wann ein bestimmtes Ereignis eingetreten ist. Diese Konfiguration ermöglicht eine detaillierte Überwachung des Skripts, insbesondere bei Fehlermeldungen oder Änderungen in der Raumliste. Die Basis-URL („BASE_URL“) verweist auf die Universitätsplattform, von der die aktuelle Liste aller verfügbaren Räume abgerufen wird. Das Skript sendet eine HTTP-Anfrage an diese URL, um den HTML-Quellcode der Raumübersicht zu erhalten, der später durch BeautifulSoup analysiert wird. Falls sich die Struktur der Webseite ändert oder eine neue URL genutzt wird, kann die „BASE_URL“ einfach aktualisiert werden, ohne den gesamten Code anpassen zu müssen. „room_data = []“ initialisierung der Raumdatenliste. Diese leere Liste dient als Zwischenspeicher für die extrahierten Raumnummern und zugehörigen Raum-IDs. Während des Webscraping-Prozesses werden die gefundenen Raum-IDs und Raumnummern als Dictionary in diese Liste eingefügt. Nach Abschluss der Extraktion wird „room_data“ als JSON-Datei gespeichert, damit der Hauptserver darauf zugreifen kann.

```
12  def replace_umlauts(text):
13      umlaut_map = {"ä": "ae", "ö": "oe", "ü": "ue", "Ä": "Ae", "Ö": "Oe", "Ü": "Ue", "ß": "ss"}
14      for umlaut, replacement in umlaut_map.items():
15          text = text.replace(umlaut, replacement)
16      return text
```

Abbildung 43 roomlist_helper.py replace_umlauts(text)

In Abb. 43 wird die Funktion „replace_umlauts()“ dargestellt, die für die Standardisierung von Raumnamen durch Ersetzung deutscher Umlaute und Sonderzeichen verantwortlich ist.
Diese Funktion stellt sicher, dass die Raumnamen einheitlich gespeichert werden, um mögliche Darstellungs- oder Verarbeitungsprobleme zu vermeiden. Ein Dictionary „umlaut_map“ enthält die entsprechenden Ersetzungen für Umlaute und das "ß". Dadurch wird eine ASCII-kompatible Schreibweise der Raumnamen sichergestellt. Die Funktion iteriert über das „umlaut_map“-Dictionary und ersetzt jedes Vorkommen eines Umlauts im gegebenen Text („text“). Nach der vollständigen Ersetzung aller Umlaute gibt die Funktion den angepassten Text zurück.

```python
def extract_room_data(soup):
    fieldsets = soup.find_all("fieldset")
    if not fieldsets:
        logging.error("Keine <fieldset>-Elemente gefunden.")
        return []

    fieldset = fieldsets[1] if len(fieldsets) > 1 else None
    if not fieldset:
        logging.error("Kein relevantes <fieldset> mit Raumdaten gefunden.")
        return []

    rooms = []
    links = fieldset.find_all("a", class_="regular")
    if not links:
        logging.error("Keine Links mit der Klasse 'regular' gefunden.")
    else:
        logging.info(f"{len(links)} Links gefunden.")

    for link in links:
        href = link.get("href")
        room_text = link.get_text(strip=True)

        room_id = None
        if "raum.rgid=" in href:
            room_id = href.split("raum.rgid=")[1].split("&")[0]

        # Ersetze Umlaute und schneide bei "•" ab
        if "•" in room_text:
            room_text = room_text.split("•")[0].strip()
        room_number = replace_umlauts(room_text)

        if room_id and room_number:
            rooms.append({"room_id": room_id, "room_number": room_number})
    return rooms
```

Abbildung 44 : roomlist_helper.py extract_room_data (soup)

In Abb. 44 wird die Funktion „extract_room_data()“ dargestellt, die für das Extrahieren der verfügbaren Räume aus dem HTML-Code der Universitätsplattform verantwortlich ist.
Diese Funktion sorgt dafür, dass die Raum-IDs und Raumnummern aus der Webseite korrekt erfasst und für die weitere Nutzung im System gespeichert werden. Die Webseite enthält die Raumlisten innerhalb von „<fieldset>“-Blöcken. Die Funktion sucht alle „<fieldset>“-Elemente mit „soup.find_all("fieldset")“. Falls keine solchen Elemente gefunden werden, wird dies als Fehler protokolliert („logging.error()“), und die Funktion gibt eine leere Liste zurück. Falls mehrere „<fieldset>“-Elemente vorhanden sind, wird das zweite Element („fieldsets[1]“) als relevanter Block angenommen. Falls dieser Block nicht existiert, wird erneut ein Fehler protokolliert und die Funktion abgebrochen. Danach sucht die Funktion alle Links („<a>-Tags mit class="regular"“), die

die Raumnummern und zugehörigen IDs enthalten. Falls keine Links gefunden werden, wird dies im Log-System vermerkt („logging.error()“). Andernfalls wird die Anzahl der gefundenen Links im Log dokumentiert („logging.info()“). Für jeden gefundenen Link („<a>“) wird die „room_id“ aus der URL („href“) extrahiert, falls sie das Muster "raum.rgid=" enthält. Die Raumnummer („room_number“) aus dem Text des Links erfasst und von Umlauten bereinigt „(replace_umlauts()“). Falls der Raumname ein „•“-Symbol enthält, wird nur der Teil vor dem Symbol gespeichert, um redundante Informationen zu vermeiden. Anschließend wird der bereinigte Raumname („replace_umlauts(room_text)“) für eine einheitliche Speicherung formatiert. Falls sowohl die Raum-ID als auch die Raumnummer vorhanden sind, wird ein Dictionary mit „{"room_id": room_id, "room_number": room_number}“ erstellt und zur Liste der Räume „rooms“ hinzugefügt. Am Ende der Funktion wird die Liste „rooms“ mit allen extrahierten Raumdaten zurückgegeben.

```
try:
    print("Starte Abfrage der Raumdaten...")
    response = requests.get(BASE_URL, timeout=30, verify=False)
    response.raise_for_status()
    soup = BeautifulSoup(response.text, "html.parser")

    debug_file_path = os.path.join(script_dir, "debug.html")
    with open(debug_file_path, "w", encoding="utf-8") as debug_file:
        debug_file.write(soup.prettify())

    room_data = extract_room_data(soup)
    if room_data:
        logging.info(f"{len(room_data)} Räume erfolgreich extrahiert.")
    else:
        logging.warning("Keine Raumdaten extrahiert.")
except requests.exceptions.RequestException as e:
    logging.error(f"Fehler bei der Anfrage: {e}")
    print(f"Fehler bei der Anfrage: {e}")

if room_data:
    json_file_path = os.path.join(script_dir, "room_data.json")
    with open(json_file_path, "w", encoding="utf-8") as json_file:
        json.dump(room_data, json_file, indent=4, ensure_ascii=False)
    print("Die Daten wurden in 'room_data.json' gespeichert.")
    logging.info("Die Daten wurden erfolgreich gespeichert.")
else:
    print("Keine Daten zum Speichern vorhanden.")
    logging.info("Keine Daten zum Speichern vorhanden.")
```

Abbildung 45 roomlist_helper.py Scriptablauf

In Abb. 45 wird der Hauptablauf des Skripts roomlist_helper.py zur Extraktion der Raumdaten von der Universitätsplattform dargestellt.

Dieser Codeblock steuert das Abrufen, Verarbeiten, Speichern und die Fehlerbehandlung der Raumliste. Mit „print("Starte Abfrage der Raumdaten...")“ wird angezeigt, dass der Webscraping-Prozess beginnt. Anschließend sendet „requests.get(BASE_URL, timeout=30, verify=False)“ eine HTTP-Anfrage an die Universitätsplattform. „timeout=30“ begrenzt die Wartezeit auf 30 Sekunden, falls die Webseite nicht reagiert. „verify=False“ deaktiviert die SSL-Zertifikatsprüfung, falls die Webseite ein selbstsigniertes Zertifikat nutzt. Mit „response.raise_for_status()“ wird überprüft, ob die Anfrage erfolgreich war. Falls der Server einen Fehlerstatus zurückgibt (z. B. 404 oder 500), wird eine Exception ausgelöst. Der HTML-Code wird mit „BeautifulSoup(response.text, "html.parser")“ in ein parse bares Format umgewandelt. Dadurch können die relevanten

Rauminformationen leichter extrahiert werden. Der vollständige HTML-Code wird mit „soup.prettify()“ formatiert und in der Datei debug.html gespeichert. Dies ermöglicht eine manuelle Überprüfung, falls sich die Struktur der Webseite geändert hat. Die Funktion „extract_room_data(soup)“ wird aufgerufen, um die Raumnummern und IDs aus der HTML-Struktur zu extrahieren. Falls Räume erfolgreich gefunden wurden, wird ihre Anzahl im Log („logging.info()“) gespeichert. Falls keine Räume gefunden wurden, wird eine Warnung („logging.warning()“) ausgegeben. Falls die HTTP-Anfrage fehlschlägt (z. B. durch einen Netzwerkfehler oder einen nicht erreichbaren Server), wird der Fehler gefangen („except requests.exceptions.RequestException“). Der Fehler wird im Log protokolliert und gleichzeitig als Fehlermeldung im Terminal ausgegeben. Falls die Liste „room_data“ erfolgreich mit Raumdaten gefüllt wurde, wird sie in der Datei „room_data.json“ gespeichert. Die JSON-Daten werden mit „indent=4“ formatiert, um eine bessere Lesbarkeit zu ermöglichen. Eine Erfolgsmeldung („print()“) zeigt an, dass die Daten gespeichert wurden. Falls keine Raumdaten gefunden wurden, wird dies im Log protokolliert und als Warnung ausgegeben, sodass das Problem später analysiert werden kann.

5.4 Aufgetretene Probleme

In diesem Kapitel werden aufgetretene Probleme während der Entwicklung des digitalen Raumplans näher betrachtet.

5.4.1 Begrenzter Arbeitsspeicher des ESP32

Während der Implementierung des Systems stellte sich der begrenzte Arbeitsspeicher des ESP32 als eine wesentliche Herausforderung dar. Der ESP32 verfügt zwar über bis zu 520 KB SRAM, wovon jedoch nur ein Teil für den MicroPython-Interpreter und benutzerdefinierten Code zur Verfügung steht. Dies führte zu Speicherengpässen, insbesondere bei der gleichzeitigen Darstellung der Raumbelegungsdaten auf dem E-Paper-Display und der Verarbeitung von Over-the-Air (OTA)-Updates.

Die ursprüngliche Architektur sah vor, dass das Anzeigen der Raumpläne auf dem E-Paper-Display und die MQTT-basierte OTA-Update-Funktionalität gleichzeitig im Hintergrund mit „uasyncio“ laufen. Dies hätte den Vorteil gehabt, dass der ESP32 stets auf neue Updates reagieren kann, während er gleichzeitig die Anzeige der Belegungspläne aktualisiert.

Allerdings traten dabei folgende Probleme auf. Speicherüberläufe, die Kombination aus JSON-Datenverarbeitung, Framebuffer-Rendering und MQTT-Kommunikation beanspruchte mehr RAM als verfügbar war, was zu Systeminstabilitäten und Abstürzen führte. Blockierende Display-Ansteuerung, die Aktualisierung des E-Paper-Displays benötigt eine erhebliche Menge an RAM, da der Framebuffer für die Darstellung des gesamten Bildschirms gespeichert werden muss. Dies führte dazu, dass der ESP32 während der parallellaufenden WLAN-Verbindung und auf MQTT-Nachrichten warten nicht genug Arbeitsspeicher für die Erstellung des Framebuffers zur Verfügung hatte.

Aufgrund dieser Einschränkungen wurde das ursprüngliche Konzept der gleichzeitigen Verarbeitung von Display-Rendering und OTA verworfen und stattdessen ein zweistufiger Betriebsmodus eingeführt:

1. Darstellungsmodus (Display-Update)
    - Der ESP32 rendert die aktuellen Raumbelegungsdaten und zeigt sie auf dem E-Paper-Display an.

- Während dieser Phase ist die OTA-Update-Funktion deaktiviert, um Speicher zu sparen und Fehlverhalten zu vermeiden.

2. OTA-Modus (Update-Bereitschaft)

    - Falls keine gültigen Raumbelegungsdaten vorhanden sind, startet der ESP32 den Update-Modus.
    - In diesem Zustand hört der ESP32 auf MQTT-Nachrichten, kann neue Firmware-Dateien empfangen und verarbeiten und wartet auf mögliche weitere Befehle, bevor er in den Darstellungsmodus zurückkehrt.

Diese Trennung stellt sicher, dass sowohl das Display zuverlässig aktualisiert werden kann als auch OTA-Updates stabil ausgeführt werden, ohne dass Speicherüberläufe oder unerwartete Abstürze auftreten.

Durch die Einführung der Betriebsmodi wurde eine stabile Systemarchitektur geschaffen, die den ESP32 nicht überlastet. Der Wechsel zwischen den Modi ist effizient und ermöglicht es dem Gerät, flexibel zwischen Anzeige und Update-Funktionalität umzuschalten. Die Speicherbeschränkungen des ESP32 verdeutlichen die Notwendigkeit, speicherintensive Prozesse klar zu trennen und gezielt zu steuern.

Diese Lösung zeigt, dass ein direktes paralleles Ausführen speicherintensiver Aufgaben auf dem ESP32 nicht praktikabel ist, sodass alternative Architekturansätze erforderlich sind, um eine zuverlässige Funktionalität trotz begrenzter Ressourcen sicherzustellen.

Ebenfalls ist der E-Paper-Display aktuell um 180° gedreht im Aufbau angebracht, Softwareseitiges drehen des Bildes liegt leider auch außerhalb der Arbeitsspeicherkapazitäten, da ein 2. Framebuffer erstellt werden müsste um die Inhalte des 1. Framebuffers gedreht auf den 2. zu übertragen, wofür jedoch dann der noch freie Arbeitsspeicher nicht ausreicht. Deshalb wurde entschieden den E-Paper-Display einfach physisch zu drehen.

5.4.2 Begrenzte Akkukapazität und Energieeffizienz

Ein weiteres zentrales Problem bei der Entwicklung des Systems war die begrenzte Akkukapazität des ESP32-gestützten E-Paper-Displays. Das System sollte möglichst lange ohne manuelles Nachladen der Batterie betrieben werden, während es gleichzeitig eine zuverlässige und aktuelle Anzeige der Raumbelegungsdaten gewährleistet.

Der ESP32 wird über eine Lithium-Ionen-Batterie mit einer Kapazität von 4000 mAh betrieben. Die Herausforderung bestand darin, eine ausreichend lange Laufzeit zu ermöglichen, ohne die Funktionalität des Systems einzuschränken. Besonders energieintensive Prozesse, wie die WiFi-Kommunikation, das Aktualisieren des E-Paper-Displays und Over-the-Air (OTA)-Updates, haben einen erheblichen Einfluss auf den Stromverbrauch.

Während der Testphase stellte sich heraus, dass die WLAN-Verbindung und MQTT-Kommunikation eine erhebliche Menge an Energie verbrauchen, insbesondere wenn häufige Anfragen oder OTA-Updates durchgeführt werden. Die Aktualisierung des E-Paper-Displays ebenfalls einen hohen Stromverbrauch verursacht, da für das Neuzeichnen des Bildschirms ein kompletter FrameBuffer geladen und verarbeitet werden muss. Der ESP32 im Deep-Sleep-Modus den Stromverbrauch drastisch reduzieren kann, sodass nur minimale Energie für den Betrieb benötigt wird.

Um den Energieverbrauch zu optimieren und die Akkulaufzeit zu maximieren, wurde entschieden:

1. Die Raumbelegungsdaten nur einmal täglich zu aktualisieren
    - Statt einer häufigen Synchronisierung wird der Raumplan nur einmal pro Tag aktualisiert.
    - Die Daten bleiben auf dem E-Paper-Display erhalten, da dieses den zuletzt angezeigten Inhalt ohne zusätzliche Energie dauerhaft speichern kann.
2. Einsatz des Deep-Sleep-Modus zur Reduzierung des Stromverbrauchs
    - Nach der täglichen Aktualisierung geht der ESP32 in den Deep-Sleep-Modus und das E-Paper-Display wird in den Sleep versetzt, wodurch der Energieverbrauch auf etwa 0,5 Milliampere (mA) reduziert wird.
    - Der Mikrocontroller wacht nur zur festgelegten Update-Zeit, um den neuen Plan abzurufen oder OTA-Updates zu empfangen.
3. Entfernen der Power LED
    - Um unnötigen Energieverbrauch zu vermeiden, wurde die Power LED entfernt, die kontinuierlich leuchtete beim eingeschalteten Zustand.

Durch die Begrenzung der Aktualisierung auf einmal täglich konnte die Akkulaufzeit erheblich verlängert werden. Die Nutzung des Deep-Sleep-Modus reduziert den Energieverbrauch drastisch und ermöglicht einen effizienten Batteriebetrieb. Die Architektur zeigt, dass eine intelligente

Steuerung der Energieverbraucher (WiFi, Display-Updates, OTA) notwendig ist, um eine nachhaltige Betriebsdauer zu gewährleisten.

5.4.3 Ungenauigkeit des WLAN-Fingerprinting Projekts

Ein wesentliches Ziel des Projekts war die automatische Standortbestimmung des ESP32 über WLAN-Fingerprinting, um eine manuelle Konfiguration der Raumzuordnung zu vermeiden. Dabei sollte der ESP32 eine Liste verfügbarer WLAN-Netzwerke erfassen, deren Signalstärken auswerten und diese Daten an einen Server senden, der anhand eines zuvor trainierten Modells den aktuellen Raum des Geräts bestimmt.

Während der Tests stellte sich jedoch heraus, dass die aktuelle Implementierung des Fingerprinting-Ansatzes eine Genauigkeit von lediglich 51,2 % erreicht. Diese geringe Präzision ist für den praktischen Einsatz nicht ausreichend, da der ESP32 in fast der Hälfte der Fälle einem falschen Raum zugeordnet würde (vgl. Völkers, 2024).

Aufgrund dieser Ungenauigkeiten wurde entschieden, dass das WLAN-Fingerprinting zwar als Schnittstelle im System erhalten bleibt, die tatsächliche Raumzuordnung jedoch aktuell noch durch einen statischen (gemockten) Wert ersetzt wird. Dies bedeutet, dass der ESP32 weiterhin die Fingerprinting-Daten erfasst und an den Server sendet, die Raumzuweisung aber vorerst manuell festgelegt wird, bis das Modell verbessert und die Genauigkeit gesteigert werden kann.

5.5 Eingesetzte Werkzeuge

Für die Entwicklung des Systems wurden verschiedene Software- und Entwicklungswerkzeuge eingesetzt, um den Code zu schreiben, zu testen, zu versionieren und zu debuggen. Diese Werkzeuge erleichterten die Programmierung des ESP32, die Verwaltung des Codes und die Fehleranalyse, insbesondere bei der Entwicklung der Firmware in MicroPython und der Serverkomponenten in Python.

Thonny (IDE für MicroPython) - Entwicklungsumgebung zur Programmierung und Debugging der ESP32-Firmware in MicroPython. Unterstützt serielle Verbindungen zum ESP32 und ermöglicht das direkte Hochladen und Testen von Skripten. Integrierte Debugging-Funktion für eine schrittweise Fehleranalyse (vgl. Annamaa, 2025).

MicroPython REPL (Read-Eval-Print Loop) über Thonny - Erlaubt das direkte Testen von MicroPython-Befehlen auf dem ESP32 über eine serielle Verbindung. Besonders nützlich für

Debugging von Netzwerkverbindungen, Dateioperationen und GPIO-Steuerung. Echtzeit-Interaktion mit dem ESP32, um Speichernutzung oder Fehlermeldungen zu analysieren (vgl. MicroPython, 2024).

GitLab (Versionskontrolle & Code-Management) - Zentrale Plattform zur Versionskontrolle, um Code-Änderungen effizient nachzuverfolgen und zu verwalten. Branching- und Merge-Funktionalität, um verschiedene Entwicklungszweige parallel zu bearbeiten. Issue-Tracking und Code-Reviews zur Qualitätssicherung und Dokumentation des Entwicklungsprozesses (vgl. GitLab Inc., 2024).

Node-RED - Eingesetzt für OTA-Updates, um MQTT-basierte Firmware-Updates an den ESP32 zu senden. Visuelle Programmieroberfläche, um MQTT-Workflows einfach zu konfigurieren. Ermöglichte eine zentralisierte Verwaltung von Firmware-Updates ohne manuelle Eingriffe (vgl. OpenJS Foundation, 2024).

Visual Studio Code (VS-Code) Gelegentlich genutzt für Code-Review und Bearbeitung von ESP32 Micropython Code. Unterstützt Syntax-Highlighting, Autovervollständigung und Git-Integration (vgl. Microsoft, 2024).

PyCharm (IDE für die Server-Programmierung) Verwendet zur Entwicklung des Python-Servers, der die Raumpläne verarbeitet und MQTT-Nachrichten verwaltet. Erweiterte Debugging-Funktionalität, um Fehler in der Webscraping-Logik (BeautifulSoup) und MQTT-Kommunikation zu identifizieren. Unterstützt virtuelle Umgebungen, um Python-Abhängigkeiten gezielt zu verwalten (JetBrains, 2024).

6. Validierung

Die Validierung des Systems erfolgte durch eine Kombination aus funktionalen Tests, Simulationen und Energieverbrauchsmessungen, um die Korrektheit, Stabilität und Effizienz des entwickelten Systems zu überprüfen. Dabei wurde sowohl die Kommunikation zwischen ESP32 und Server als auch der Stromverbrauch in verschiedenen Betriebszuständen getestet.

6.1 Funktionstests der Server-Kommunikation

Um die korrekte Kommunikation zwischen dem ESP32 und dem Server sicherzustellen, wurde ein Testprogramm entwickelt, das die Abfrage des ESP32 an den Server simuliert. Dieses Testprogramm ermöglichte eine kontrollierte Überprüfung der MQTT-Kommunikation, ohne dass

ein physisches ESP32-Gerät benötigt wurde. Im Rahmen der Tests wurden mehrere Szenarien durchgespielt. Zunächst wurde eine Raumanfrage über MQTT gesendet, um zu prüfen, ob der Server die Belegungspläne korrekt abrufen und zurücksenden kann. Anschließend wurde getestet, ob die empfangenen JSON-Daten vom ESP32 oder vom Testprogramm ordnungsgemäß verarbeitet und gespeichert werden können.

Zusätzlich wurden gezielt ungültige oder unvollständige Anfragen an den Server geschickt, um die Fehlerbehandlung zu testen. Hierbei zeigte sich, dass der Server auf fehlerhafte Anfragen nicht abstürzte, sondern diese korrekt verarbeitete und entsprechende Fehlermeldungen protokollierte. Die Ergebnisse der Tests bestätigten, dass die Kommunikation zwischen Server und ESP32 stabil funktioniert, alle Daten korrekt übertragen werden und Fehlerfälle zuverlässig behandelt werden.

6.2 Validierung auf realer Hardware (ESP32-Testlauf)

Zusätzlich zur Simulation wurde das vollständige System mit einem realen ESP32 getestet, um sicherzustellen, dass alle Komponenten wie erwartet funktionieren. Dabei wurde überprüft, ob der Mikrocontroller sich erfolgreich mit dem WLAN und dem MQTT-Server verbindet, eine Anfrage zur Raumbelegung sendet und die empfangenen Daten korrekt auf dem E-Paper-Display darstellt. Auch das Verhalten des ESP32 bei Verbindungsabbrüchen oder fehlenden Daten wurde untersucht.

Besonderes Augenmerk lag auf der Überprüfung der OTA-Update-Funktion. Dabei wurde getestet, ob der ESP32 neue Firmware-Versionen über MQTT empfangen und installieren kann. Außerdem wurde überprüft, ob das System zuverlässig zwischen den beiden Betriebsmodi – Darstellung der Raumbelegungsdaten und Empfang von OTA-Updates – umschalten kann. Die Tests bestätigten, dass der ESP32 korrekt zwischen diesen Modi wechselt und die Firmware-Updates fehlerfrei verarbeitet.

Insgesamt zeigten die Tests, dass die vollständige Systemkette – vom Server über die Datenverarbeitung bis zur Anzeige auf dem Display – stabil läuft. Die empfangenen Daten wurden fehlerfrei verarbeitet, und das System konnte sowohl auf Netzwerkereignisse als auch auf Aktualisierungen zuverlässig reagieren.

6.3 Energieverbrauchsmessungen

Da das System batteriebetrieben ist, war es essenziell, den Energieverbrauch des ESP32 in verschiedenen Betriebszuständen zu messen. Dabei stellte sich heraus, dass die WLAN-Kommunikation den höchsten Energieverbrauch verursacht, insbesondere während des Sendens von MQTT-Anfragen oder OTA-Updates. In diesen Phasen lag der Stromverbrauch bei etwa 200–250 mA.

Im Gegensatz dazu zeigte sich, dass die Anzeige des Raumplans auf dem E-Paper-Display nur während der Aktualisierung Energie benötigt. Danach bleibt das Bild ohne weiteren Stromverbrauch sichtbar, wodurch die Leistungsaufnahme auf etwa 32 mA begrenzt werden konnte. Der größte Effekt auf die Energieeinsparung wurde jedoch durch den Einsatz des Deep-Sleep-Modus erreicht. Sobald der ESP32 in diesen Zustand versetzt wurde, sank der Energieverbrauch auf etwa 1.6 mA. Durch das Ablöten der Power LED sowie dem Sleep-Modus des E-Paper-Displays konnte dies auf etwa 0.53 mA reduziert werden.

Abbildung 46 Messung des Strombedarfs des digitalen Raumplans im minimalen Verbrauchszustand

In der Abb. 46 ist der digitale Raumplan zu sehen, während der ESP32 im Deep Sleep ist, der E-Paper-Display im Sleep, und die Power LED abgelötet ist.

Die Messungen machten deutlich, dass eine häufige Aktualisierung der Raumpläne die Akkulaufzeit erheblich verkürzen würde. Um den Stromverbrauch zu optimieren, wurde daher entschieden, dass die Raumpläne nur einmal täglich aktualisiert werden. Diese Optimierung ermöglicht es, dass der ESP32 über einen längeren Zeitraum ohne manuelles Nachladen betrieben werden kann. Zusätzlich wurden OTA-Updates nur in definierten Zeitfenstern zugelassen, um unnötigen Energieverbrauch zu vermeiden. Bei der aktuellen Implementation sollten rund 22,57mAh in der 1h in der OTA

Updates für 5min empfangen werden können und der Bildschirm sich aktualisiert verbraucht werden und die restlichen 23h am Tag sollten nur 0,53 mA pro Stunde verbraucht werden. Was zu einen Verbrauch von ca. 34,59mA am Tag führt, wodurch der 4000mA Akku bei voller Ladung ca. 115 Tage halten sollte. In dieser Rechnung sind keine längeren oder häufigen OTA-Updates oder Verschleiß der Komponenten mit einberechnet. Es wurde von 5min OTA-Update Bereitschaft und 2min Zeit zum Updaten des E-Paper-Displays pro Tag ausgegangen. Hier die Rechnung dazu:

Update Stunde: 5/60 * 250mA + 2/60 * 32mA + 53/60 * 0,53mA ≈ 22,4mA

Täglicher Verbrauch: 22,4mA + 23*0,53 = 34,59mA

6.4 Zusammenfassung der Validierungsergebnisse

Die durchgeführten Tests haben gezeigt, dass das System stabil und zuverlässig arbeitet. Die Kommunikation zwischen ESP32 und Server wurde erfolgreich getestet, sowohl durch eine Simulation als auch auf der realen Hardware. Der ESP32 kann alle Daten fehlerfrei empfangen und verarbeiten, während der Server korrekt auf Anfragen reagiert.

Die Energieverbrauchsmessungen haben verdeutlicht, dass die WLAN-Nutzung der größte Energieverbraucher ist, weshalb der Deep-Sleep-Modus für eine lange Akkulaufzeit essenziell ist. Durch die Begrenzung der Aktualisierung auf einmal täglich konnte der Stromverbrauch erheblich reduziert werden.

Insgesamt bestätigen die Validierungsergebnisse, dass das System für den geplanten Einsatz geeignet ist. Die Kombination aus stabiler Kommunikation, optimiertem Energieverbrauch und robuster Fehlerbehandlung ermöglicht eine zuverlässige und effiziente Nutzung des digitalen Raumbelegungssystems.

7.Fazit

Die vorliegende Arbeit beschäftigte sich mit der Entwicklung eines energieeffizienten digitalen Raumbelegungssystems auf Basis eines ESP32-Mikrocontrollers und eines E-Paper-Displays. Ziel war es, eine automatisierte und wartungsarme Lösung zur Anzeige von Raumbelegungsplänen zu entwickeln, die sowohl kabellos aktualisiert als auch über Over-the-Air (OTA)-Updates wartbar ist. Neben der Entwicklung der Firmware wurde ein zentraler Server zur Bereitstellung der Raumpläne

realisiert, der Daten über Webscraping aus der Universitätsplattform extrahiert und über MQTT an die ESP32-Geräte verteilt.

7.1 Ergebnisse

Die Umsetzung des Projekts zeigte, dass ein automatisiertes Raumbelegungssystem mit minimalem Energieverbrauch realisierbar ist. Die Kombination aus E-Paper-Technologie, Deep-Sleep-Modus des ESP32 und einer optimierten Betriebslogik ermöglichte eine lange Akkulaufzeit bei gleichzeitig stabiler Datenübertragung.

Wichtige Ergebnisse der Implementierung sind:

- Stabile Kommunikation zwischen ESP32 und Server: Durch den Einsatz von MQTT konnte eine zuverlässige Datenübertragung gewährleistet werden.
- Energieeffiziente Architektur: Durch die Begrenzung der WLAN-Nutzung und die Nutzung des Deep-Sleep-Modus konnte der Stromverbrauch erheblich reduziert werden, sodass das System ohne externe Stromversorgung über lange Zeiträume betrieben werden kann.
- OTA-Update-Funktionalität: Die Firmware des ESP32 kann remote aktualisiert werden, was eine wartungsarme Skalierung und Weiterentwicklung des Systems ermöglicht.
- Datenvisualisierung auf dem E-Paper-Display: Die empfangenen Belegungsdaten werden in einer übersichtlichen tabellarischen Darstellung angezeigt und bleiben sichtbar, ohne zusätzliche Energie zu verbrauchen.
- Testprogramme zur Validierung der MQTT-Kommunikation und Datenverarbeitung: Neben dem physischen Testlauf mit dem ESP32 wurde ein Testprogramm entwickelt, das den Anfrageprozess des Mikrocontrollers simuliert.

Die durchgeführten Funktionstests bestätigten, dass das System die Raumpläne zuverlässig abruft und darstellt, wobei der gesamte Prozess vollständig automatisiert abläuft.

7.2 Limitationen

Trotz der erfolgreichen Implementierung des Systems gibt es einige Einschränkungen, die zukünftige Arbeiten adressieren können:

- Geringer Arbeitsspeicher des ESP32: Die begrenzte Speicherkapazität führte dazu, dass die gleichzeitige Ausführung von OTA-Updates und der Anzeige der Raumpläne nicht möglich war. Daher wurden zwei getrennte Betriebsmodi eingeführt.
- Eingeschränkte WLAN-Fingerprinting-Genauigkeit: Das WLAN-basierte Indoor-Lokalisierungssystem konnte bislang nur eine Genauigkeit von 51,2 % erreichen. Aufgrund dieser Ungenauigkeit wird die Raumzuordnung derzeit statisch (gemockt) gesetzt, bis eine zuverlässigere Lösung implementiert werden kann.
- Begrenzte Aktualisierungshäufigkeit aufgrund des Energieverbrauchs: Um eine lange Batterielaufzeit zu gewährleisten, wurde entschieden, dass der Raumplan nur einmal täglich aktualisiert wird. Häufigere Updates würden den Energieverbrauch deutlich erhöhen.
- Fehlende Interaktionsmöglichkeit: Der ESP32 zeigt den Raumplan an, bietet aber aktuell keine Möglichkeit für manuelle Nutzerinteraktion, etwa zur Anforderung von Aktualisierungen außerhalb des festgelegten Zeitplans.

7.3 Ausblick

Für zukünftige Weiterentwicklungen des Systems gibt es mehrere Ansatzpunkte:

- Verbesserung der WLAN-Fingerprinting-Genauigkeit: Die Raumzuordnung soll durch ein erweitertes Fingerprinting-Modell mit mehr Trainingsdaten und optimierten Algorithmen zuverlässiger werden.
- Optimierung der Speicherverwaltung: Durch effizientere Nutzung des ESP32-Speichers oder den Einsatz eines leistungsfähigeren Mikrocontrollers könnte es möglich sein, OTA-Updates und die Anzeige der Raumpläne parallel auszuführen.
- Erweiterung der Energieoptimierung: Neben dem Deep-Sleep-Modus könnten weitere Low-Power-Techniken oder alternative Energiequellen wie Solarzellen erforscht werden, um den Betrieb noch nachhaltiger zu gestalten.
- Erhöhte Update-Frequenz durch differenzierte Energiemodi: Ein dynamisches Energiemanagement könnte es ermöglichen, dass der ESP32 bei wichtigen Änderungen der Raumpläne häufiger aufwacht, während er bei unveränderten Daten weiterhin im energiesparenden Modus bleibt.

- Benutzerinteraktion und erweiterte Funktionalitäten: Zukünftig könnten zusätzliche Interaktionsmöglichkeiten zur manuellen Anfrage von Belegungsplänen integriert werden.

Insgesamt zeigt die Arbeit, dass energieeffiziente, vernetzte Raumbelegungssysteme mit Mikrocontrollern erfolgreich realisierbar sind. Die Implementierung stellt eine skalierbare und wartungsarme Lösung dar, die durch zukünftige Verbesserungen noch leistungsfähiger und flexibler gestaltet werden kann.

8. Quellen

Annamaa, A. (2025). Thonny - Python IDE für Anfänger. [online] https://github.com/thonny/thonny/?tab=readme-ov-file (abgerufen am 04.02.2025).

Eclipse Foundation (2024). *Paho MQTT Python Client Library Documentation.* Eclipse. *[online]* https://eclipse.dev/paho/files/paho.mqtt.python/html/client.html (abgerufen am 04.02.2025).

E Ink Corporation (2024). *E Ink Applications.* E Ink. *[online]* https://www.eink.com/application (abgerufen am 04.02.2025).

Espressif Systems (2023a). ESP32-WROOM-32 Datasheet. Espressif Systems. *[online]* https://www.espressif.com/sites/default/files/documentation/esp32-wroom-32_datasheet_en.pdf (abgerufen am 04.02.2025).

Espressif Systems (2023b). *OTA Updates with ESP32.* Espressif Systems. *[online]* https://docs.espressif.com/projects/esp-idf/en/stable/esp32/api-reference/system/ota.html (abgerufen am 04.02.2025).

Fasolo, E., Pastore, G., Mantovani, F., Tondella, G. und Ferretto, A. (2022). A secure and reliable OTA update system for IoT devices. *HardwareX*, 11, e00214. [online] https://www.sciencedirect.com/science/article/abs/pii/S2542660522000142 (abgerufen am 04.02.2025).

Flemming R., Zubenin L. (2024). *ESP32 Bildschirm*. GitLab. *[online]* https://gitlab.rz.htw-berlin.de/s0584322/esp32bildschirm (abgerufen am 04.02.2025).

George, D. (2023). *MicroPython – Python for microcontrollers.* MicroPython. *[online]* https://micropython.org/ (abgerufen am 04.02.2025).

Giesa, T. (2024). Vortrag. GitLab. [online] https://gitlab.rz.htw-berlin.de/huhn/b_iot_ss24/-/tree/giesa_s0584373 (abgerufen am 04.02.2025).

GitLab Inc. (2024). GitLab - The DevSecOps Platform. [online] https://about.gitlab.com/ (abgerufen am 04.02.2025).

Hinch, P. (2016–2018). Writer.py: Implements the Writer class [Computer-Software]. GitHub. [online] https://github.com/peterhinch/micropython-font-to-py (abgerufen am 4.2.2024)

JetBrains. (2024). PyCharm - Die professionelle IDE für Python-Entwicklung. [online] https://www.jetbrains.com/pycharm/ (abgerufen am 04.02.2025).

Koubek, E. (2007). *E-Paper.* Universität Bayreuth. *[online]* https://medienwissenschaft.uni-bayreuth.de/wp-content/uploads/assets/Koubek/forschung/KoubekE-Paper.pdf (abgerufen am 04.02.2025).

Lutz, M. (2009): *"Learning Python"*, 4.Auflage, O'Reilly Media

Microsoft. (2024). Visual Studio Code - Code Editing Redefined. [online] https://code.visualstudio.com/ (abgerufen am 04.02.2025).

MicroPython. (2024). MicroPython REPL - Interaktive Python-Shell für Mikrocontroller. [online] https://docs.micropython.org/en/latest/reference/repl.html (abgerufen am 04.02.2025).

OpenJS Foundation. (2024). Node-RED - Low-code Entwicklung für IoT-Anwendungen. [online] https://nodered.org/ (abgerufen am 04.02.2025).

Richardson, L. (2024). *Beautiful Soup Documentation.* Crummy. *[online]* https://www.crummy.com/software/BeautifulSoup/bs4/doc/ (abgerufen am 04.02.2025).

tanahy (2020). *Commit b5d20249d858a0a84629b9b3004a9779b746e76f - MicroPython Waveshare ePaper.* GitHub. *[online]* https://github.com/mcauser/micropython-waveshare-epaper/pull/12/commits/b5d20249d858a0a84629b9b3004a9779b746e76f (abgerufen am 04.02.2025).

Usmani, M. F. (2021). *MQTT Protocol for the IoT - Review Paper.* ResearchGate. *[online]* https://www.researchgate.net/publication/373640610_MQTT_Protocol_for_the_IoT_-_Review_Paper (abgerufen am 04.02.2025).

Völkers, F. (2024). Bachelorarbeit. GitLab. [online] https://gitlab.rz.htw-berlin.de/s0585012/wifi-fingerprint-based-indoor-localization/-/tree/main?ref_type=heads (abgerufen am 04.02.2025).

Waveshare (2024). *7.5inch e-Paper V2 Specification.* Waveshare. *[online]* https://files.waveshare.com/upload/6/60/7.5inch_e-Paper_V2_Specification.pdf (abgerufen am 04.02.2025).

9. Abbildungsverzeichnis

A Abkürzungsverzeichnis

Abkürzung	Bedeutung
API	Application Programming Interface
DB	Datenbank
ESP32	Mikrocontroller von Espressif Systems für IoT-Anwendungen
GPIO	General Purpose Input/Output
HTML	Hypertext Markup Language
I²C	Inter-Integrated Circuit (serielles Kommunikationsprotokoll für elektronische Bauteile)
IoT	Internet of Things
JSON	JavaScript Object Notation
LCD	Liquid Crystal Display
MAC	Media Access Control
MHz	Megahertz
MQTT	Message Queuing Telemetry Transport
OTA	Over-the-Air (Update-Verfahren)
RAM	Random Access Memory
SPI	Serial Peripheral Interface
SSID	Service Set Identifier (WLAN-Name)
WLAN	Wireless Local Area Network

B Fotos des Aufbaus

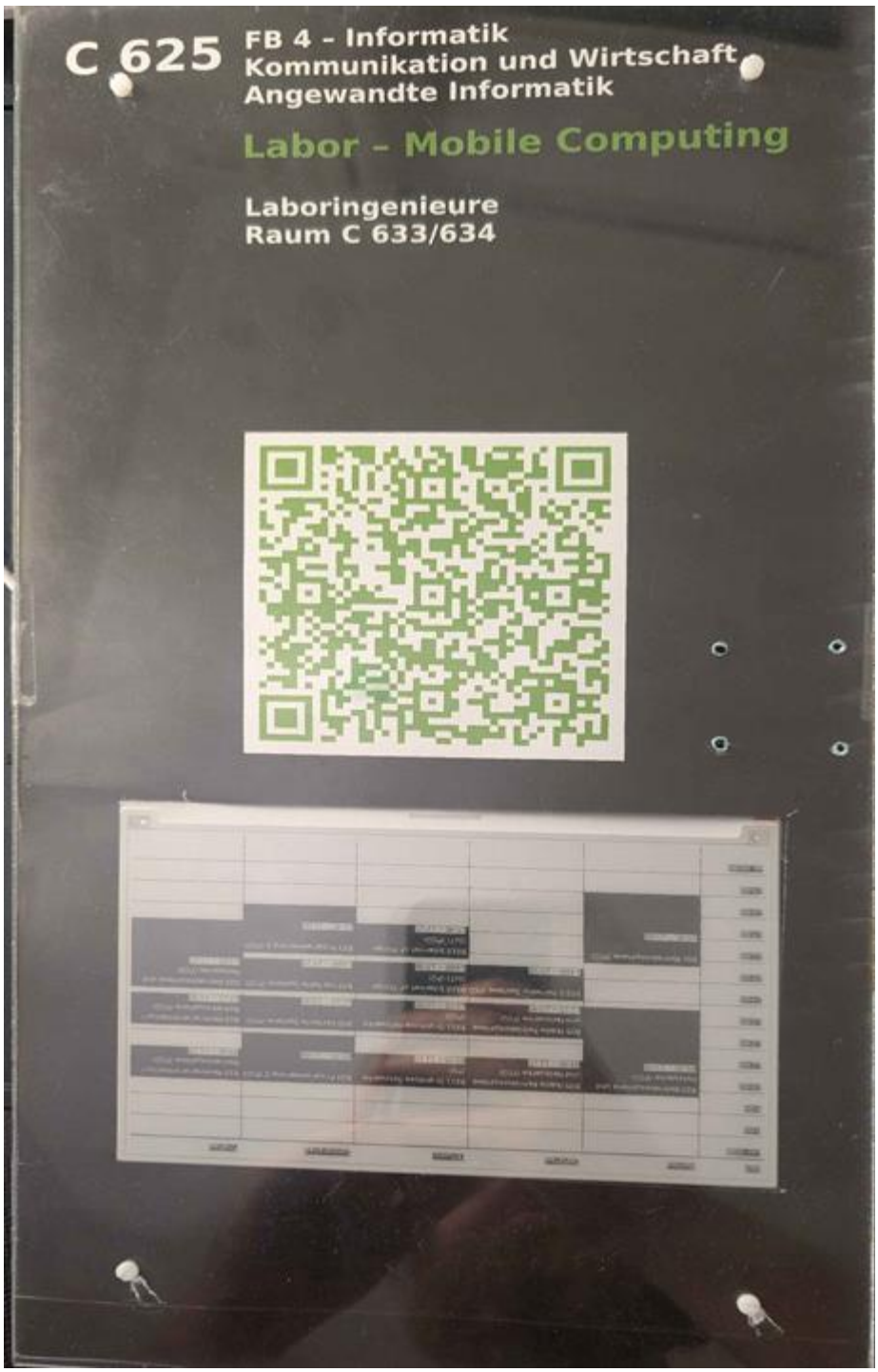


Abbildung 47 Prototyp Vorderseite

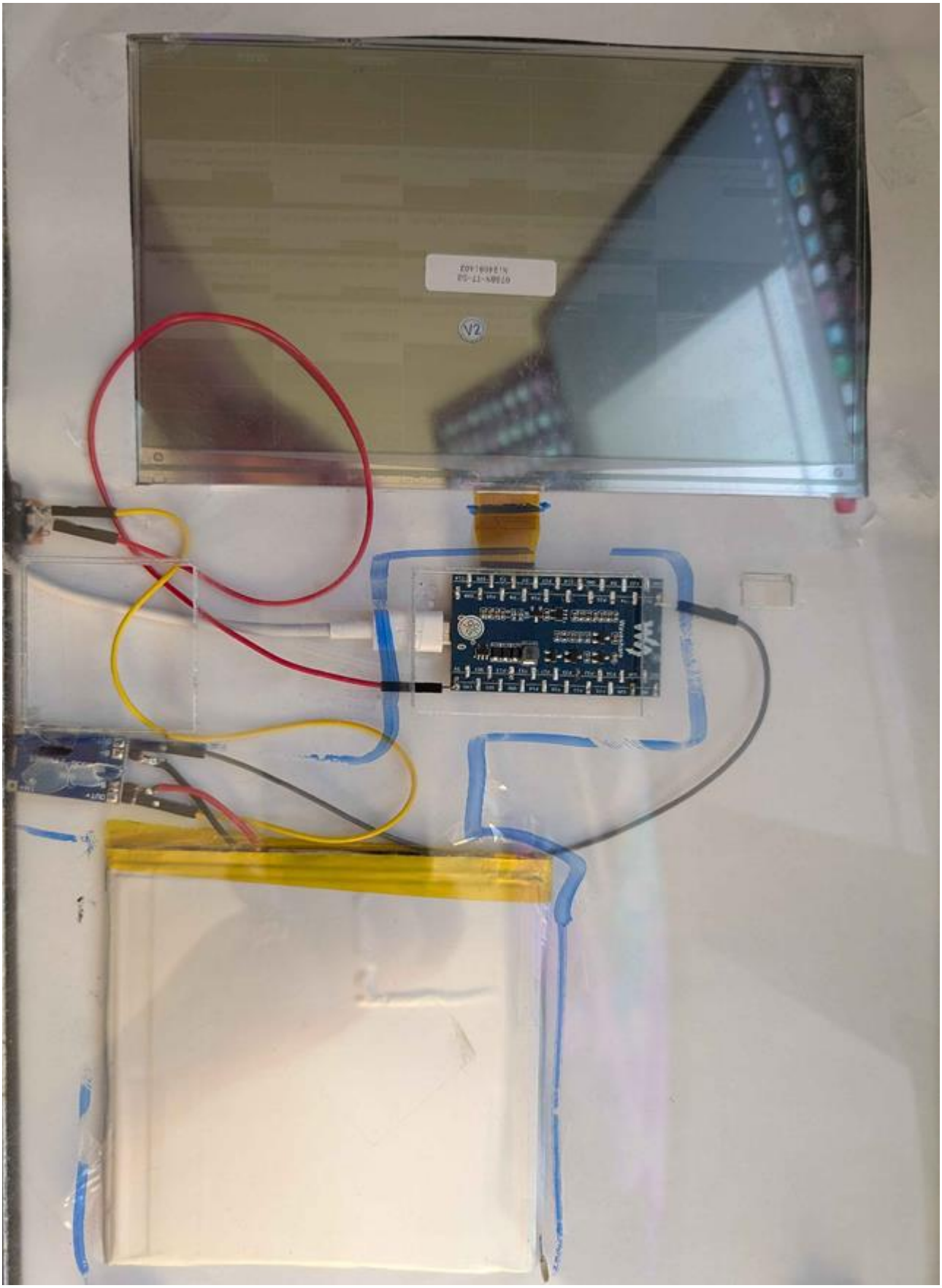


Abbildung 48 Prototyp Rückseite

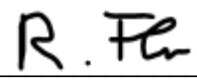
C Gitlab Link des Projektes

Hierbei handelt es sich um den Gitlab Link des Projektes mit der Firmware für den ESP32, sowie der „server_schedule.py“ mit ihrer „requirements.txt“, der „roomlist_helper.py“ mit ihrer „requirements.txt“ und dem OTA-Flow für NodeRed von Till Giesas Projekt.

https://gitlab.rz.htw-berlin.de/mobile-computing/projekte/raumplan/-/tree/main?ref_type=heads

Eidesstattliche Versicherung

Hiermit versichere ich an Eides statt durch meine Unterschrift, dass ich die vorliegende Arbeit selbstständig und ohne fremde Hilfe angefertigt und alle Stellen, die ich wörtlich oder annähernd wörtlich aus Veröffentlichungen entnommen habe, als solche kenntlich gemacht und mich auch keiner anderen als der angegebenen Literatur oder sonstiger Hilfsmittel bedient habe. Die Arbeit hat in dieser oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen.

10.02.2025, Berlin  Datum, Ort, Unterschrift