PROJEKTARBEIT

ELEKTRO- UND INFORMATIONSTECHNIK

Entwicklung eines IR-Empfängers für einen Stromzähler

Autoren

Katharina STEIB

Simon KOCHER

Julian RICO BIRNER

Betreuer

Bernd KLEHN

14. September 2021

steibka77404@th-nuernberg.de (3264785)
kochersi74936@th-nuernberg.de(3254887)
ricobirnerju74691@th-nuernberg.de(3273163)

Inhaltsverzeichnis

I. Hauptteil 4

1. Einleitung 5

2. Konzept 7
2.1. Soll-Ist-Analyse . . . 7
2.2. Lösungskonzept . . . 7

3. Implementierung 9
3.1. Stromzähler . . . 9
3.1.1. Infrarotschnittstellen . . . 9
3.1.2. SML - Smart Message Language . . . 9
3.2. IR Übertragungsstrecke . . . 10
3.3. Microcontroller . . . 12
3.4. Energieverbrauch . . . 13
3.4.1. Abschätzung der Akkulebensdauer . . . 14
3.4.2. Schaltungstechnische Realisierung des Standbyzustands . . . 15
3.4.3. Messtechnische Überprüfung des Energieverbrauchs . . . 17
3.5. Platine . . . 18
3.6. Software . . . 21
3.6.1. Programmablauf . . . 21
3.6.2. RTC . . . 22
3.6.3. WifiManager . . . 22
3.6.4. Einlesen der SML-Daten . . . 23
3.6.5. MQTT . . . 25
3.6.6. NodeRed und Datenbank . . . 26
3.7. Gehäuse . . . 27
3.8. Teststrecke . . . 28
3.8.1. Aufbau . . . 28

3.8.2. Funktionalität 29

4. Anleitung **31**
4.1. Raspberry Pi 31
4.1.1. Betriebssystem 31
4.1.2. Fernzugriff 32
4.1.3. Datenbank 33
4.1.3.1. Installation MariaDB 33
4.1.3.2. Passwort für Datenbanklogin zurücksetzen 34
4.1.3.3. Tabellen für Messwerte anlegen 34
4.1.4. MQTT 34
4.1.5. Node-Red 35
4.2. ESP8266 37
4.3. Teststrecke 38

5. Fazit **40**

II. Anhang **41**
5.1. Schaltplan 42
5.2. Software Teststand 43
5.3. Software Empfänger 46

Literaturverzeichnis **54**

III. Danksagung **57**

6. Danksagung **58**

Teil I.

Hauptteil

1. Einleitung

Durch die Coronapandemie ist der Stromverbrauch in Deutschland im Jahr 2020 um rund 4,4 Prozent im Vergleich zum Vorjahr gesunken, was einer Einsparung von 25 TWh entspricht. Dieses Phänomen lässt sich besonders auf den lockdownbedingten Konjunktureinbruch zurückführen. Trotz des Rückgangs ist aber noch viel Luft nach oben. Laut dem Stromspiegel 2019 verschwendeten die privaten Haushalte in Deutschland Strom im Wert von 9 Mrd. Euro.
Viele Deutsche wissen gar nicht wie hoch ihr Stromverbrauch eigentlich ist und ob dieser über oder unterdurchschnittlich ist. Zum Vergleich kann der durchschnittliche Stromverbrauch einer Familie von 4250 kWh pro Jahr heran gezogen werden. Den eigenen Stromverbrauch zu ermitteln und im besten Fall auch entsprechend zu regulieren ist schon schwieriger. Zwar findet man im Internet zahlreiche Seiten die anhand verschiedener Angaben den Verbrauch abschätzen, jedoch handelt es sich dabei lediglich um eine grobe Näherung. [Str]

Im Zuge der Digitalisierung verfügen immer mehr Haushalte über ein Smart Home System. Dabei handelt es sich eigentlich um einen Überbegriff der verschiedene Verfahren vereint, die mit automatisierbaren Abläufen das Leben in Bezug auf den Wohnraum vereinfachen sollen. Bekannte Beispiele sind das Regeln der Heizung auch von Unterwegs, das an und ausschalten verschiedener Geräte oder Lampen ohne zum Schalter gehen zu müssen oder auch das Öffnen und Schließen von Fenstern und Rollläden. Dabei können die Geräte sowohl über eine Programmierschnittstelle zu bestimmten Zeiten und/oder Bedingungen geschaltet werden als auch bequem per App vom Smartphone. Das Zuhause wird also sowohl intelligenter als auch bequemer für seine Bewohner.
Ein solches Smart Home System eignet sich auch zur genauen Stromerfassung und Regulierung besonders gut. Der Verbraucher kann nicht nur genaue Daten ablesen sondern auch gezielt daran arbeiten seinen Stromverbrauch mit speziell auf ihn angepassten Einstellungen zu minimieren. Leider ist die Installation eines solchen Systems oftmals mit einem hohen Aufwand und hohen Kosten verbunden, da meist viele bereits

vorhandene Teile ausgetauscht werden müssen. Auch die Kosten für die Installation und Einrichtung sind nicht unerheblich. [Wik21]

In diesem Projekt wurde an einer möglichst benutzerfreundlichen, kostengünstigen und einfachen Lösung gearbeitet mit der man einen Stromzähler zu Hause selbst visualisieren kann. Dabei ist hervorzuheben, dass das entwickelte Gerät an den bereits vorhandenen Stromzähler ergänzt wird und somit kein Austausch von Geräten nötig ist.
Der folgende Bericht wurde zur besseren Übersicht in zwei Teile unterteilt. Im ersten Teil werden die technischen Hintergründe und Funktionen erläutert. Der zweite Teil soll hierbei als eine Art Benutzerhandbuch zur Anleitung dienen.

2. Konzept

Im Folgenden wird zunächst auf die Aufgabenstellung eingegangen und anschließend ein Lösungsansatz schematisch erläutert.

2.1. Soll-Ist-Analyse

Gegeben ist ein Stromzähler des Typen „eHZ Generation K“ der Firma EMH metering GmbH & Co. KG [hau]. Der Zähler verfügt über mehrere Infrarotschnittstellen zur Datenübertragung [eHZ]. Es soll ein Empfänger für die Zählerstände, die über diese Schnittstellen übertragen werden entwickelt werden. Der Empfänger soll in sinnvollen Intervallen (z.B. zweimal pro Tag) den Zählerstand ermitteln und an einen bestehenden Smart-Home-Controller in Form eines Raspberry Pi übermitteln.
Der bestehende Controller zeigt bereits vorhandene Daten aus anderer Quelle in einem Dashboard auf Basis von Node-RED [nod] an. Die Zäherstände sollen ebenfalls in Node-RED angezeigt werden.
Um eine möglichst einfache Installation des Empfängers zu ermöglichen, soll der Empfänger sowohl im Hinblick auf die Datenübertragung als auch die Energieversorgung vollständig drahtlos funktionieren.
Weiterhin soll ein Teststand entwickelt werden, mit dem der Empfänger ohne den Stromzähler auf Funktion getestet werden kann.

2.2. Lösungskonzept

Um eine drahtlose Energieversorgung des Empfängers zu ermöglichen, wird der Empfängeraufbau mit einem Akku betrieben. Für den Akku müssen sowohl eine Ladevorrichtung als auch der Akkutechnologie entsprechende Schutzschaltungen vorgesehen werden. Der Datenempfang und die Weitergabe an den Smart-Home-Controller wird durch einen Microcontroller realisiert. Um auch für die Datenübermittlung an den

Controller kabellos zu bleiben, wird ein Microcontroller mit WiFi-Schnittstelle benötigt. Der Datentransfer zum Smart-Home-Controller kann dabei über das MQTT-Protokoll [MQT20] realisiert werden. Für den Empfang der Stromzählerstände muss ein Infrarotempfänger, der kompatibel zur IR-Schnittstelle des Zählers ist, entwickelt werden.

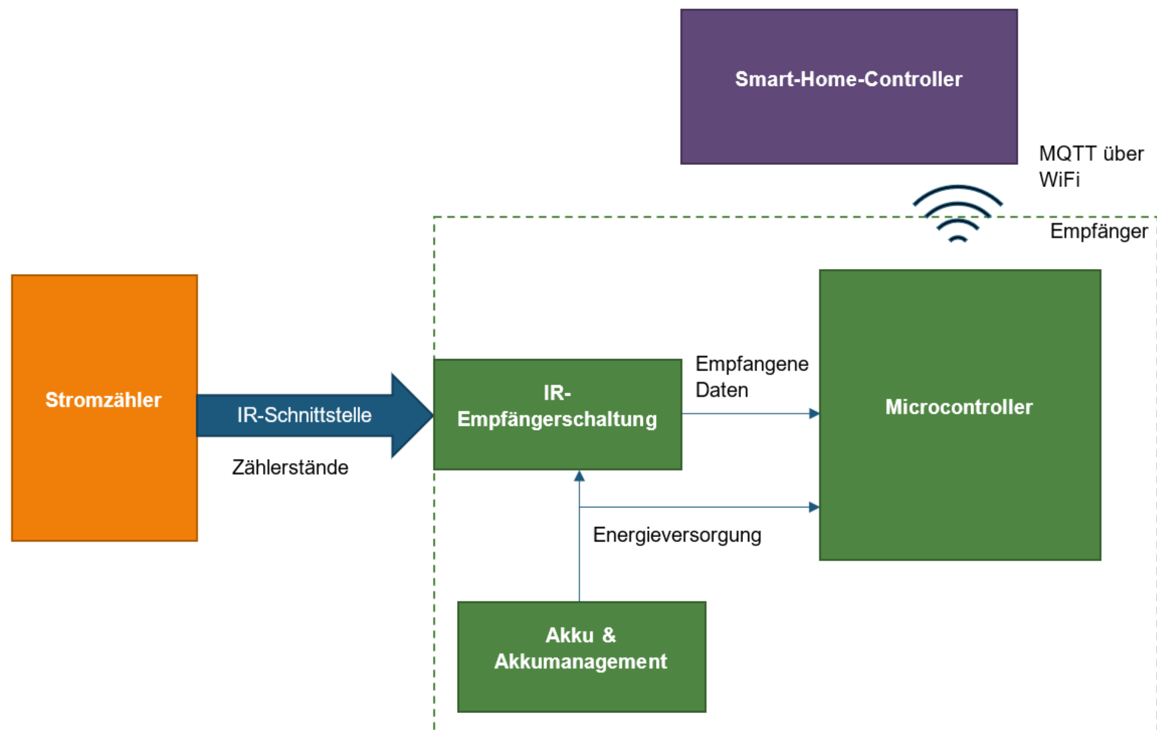


Abbildung 2.1.: Schematische Darstellung des Lösungskonzepts

Für den Teststand muss ein Infrarotsender entwickelt werden, der die IR-Schnittstelle des Stromzählers nachahmt und konfigurierbare Werte als Zählerstand versendet. Hier bietet sich ebenfalls eine Lösung mit einem Microcontroller an.

3. Implementierung

3.1. Stromzähler

3.1.1. Infrarotschnittstellen

Wie bereits erwähnt verfügt der Stromzähler über mehrere Infrarotschnittstellen. Dabei ist zwischen der Prüf-LED und den beiden optischen Datenschnittstellen zu unterscheiden. Über die Prüf-LED sendet der Stromzähler 10.000 Impulse pro kWh, vorausgesetzt es wird ein Strom oberhalb der Anlaufschwelle gemessen. Diese Schnittstelle dient jedoch, wie der Name bereits vermuten lässt, primär zur Funktionsprüfung des Zählers. Die beiden Datenschnittstellen hingegen können dazu verwendet werden, umfangreichere Daten auszulesen.
Auf der Rückseite des Zählers befindet sich eine bidirektionale Datenschnittstelle, worüber neben dem Auslesen von Zählstanden auch das Setzen und Lesen von Zählerparametern, Fernabfragung oder Prüfung des Zählers möglich ist. Diese Schnittstelle ist jedoch hauptsächlich für den Netzbetreiber gedacht und aus diesem Grund auch mit einer Betriebsplombe versehen. Für den Endkunden zugänglich ist die unidirektionale INFO-Schnittstelle auf der Vorderseite, worüber der gleiche Datensatz empfangen werden kann, aber keine aktive Kommunikation mit dem Messgerät möglich ist [EMH20].

3.1.2. SML - Smart Message Language

Über beide Datenschnittstellen wird alle paar Sekunden ein Datensatz verschickt, wobei hier zwischen einem reduzierten und einem vollständigen Datensatz gewechselt wird. Beide Datensätze verwenden SML (=Smart Message Language) [Bun13] als Kommunikationsprotokoll mit einer Baudrate von 9600 Baud, einer maximalen Übertragungszeit von 400ms und eine Auflösung des Gesamtverbrauchs von 100 mW h[EMH20]. Die dabei übertragenen Werte werden über das OBIS-Kennzahlen-System [BDE13] kodiert. Der für das Projekt relevante Gesamtverbrauch hat dabei die Kennzahl 1.8.0.

3.2. IR Übertragungsstrecke

Die Impulsschnittstelle des Stromzählers würde sich grundsätzlich zum Aufzeichnen der Zählerstände eignen, vorausgesetzt der Zählerstand zu Aufzeichnungsbeginn ist bekannt. Ein großer Vorteil dieser Schnittstelle ist ihre extrem simple Natur: per GPIO-Interrupt eines Microcontrollers oder sogar per diskretem Zähler-IC ist der Datenempfang sehr einfach realisierbar. Dagegen muss bei der IR-Datenschnittstelle sowohl das Übertragungsprotokoll als auch die Datencodierung, die der Zähler verwendet, unterstützt werden.
Gegen die Impulsschnittstelle sprechen allerdings einige Nachteile. Da der Zähler lediglich eine fixe Anzahl an Impulsen pro verbrauchter Kilowattstunde sendet, werden hier nur Verbrauchsdeltas und nie der absolute Wert übermittelt. Folglich wird zur Feststellung eines tatsächlichen Zählerstandes in Kilowattstunden der Zählerstand zu Aufzeichnungsbeginn benötigt. Außerdem führt ein temporärer Ausfall der Empfängerschaltung, bei dem Impulse nicht aufgezeichnet werden, zur Messung falscher Zählerstände.
Ein weiterer Nachteil der Impulsschnittstelle ist dass zumindest ein Teil der Empfängerschaltung dauerhaft in Betrieb sein muss um jeden Infrarotimpuls aufzunehmen. Gerade bei einem Microcontroller ist hier von einem relativ hohen Energieverbrauch auszugehen bei dem ein Großteil der Energie verschwendet wird.
Auf der Datenschnittstelle wird in regelmäßigen Intervallen der absolute Zählerstand übermittelt, damit entfallen beide Nachteile der Impulsschnittstelle. Daher wurde sich für dieses Projekt für den Empfang an der Datenschnittstelle entschieden.

Die Datenschnittstelle überträgt hier die Zählerstände mittels des sogenannten D0-Protokolls [Str09] nach DIN EN 62056-21. Die Beschreibung des Protokolls lässt bereits eine Ähnlichkeit mit RS232 vermuten, experimentell konnte bestätigt werden, dass die D0-Daten mittels der RS232-Peripherie eines ESP8266 Microcontroller empfangen werden können. Dabei muss lediglich der RX-Pin der RS232-Schnittstelle des Controllers an eine geeignete Empfangsschaltung angeschlossen werden.
Bei einem Test mit einem Stromzähler des Herstellers „Landis+Gyr“ konnte allerdings festgestellt werden, dass die Datenformatangabe aus [Str09], d.h. 1 Startbit, 7 Datenbits, 1 Paritätsbit und ein Stoppbit, von diesem Zähler nicht eingehalten wird, stattdessen wird von diesem Zähler 1 Startbit, 8 Datenbits und 1 Stoppbit verwendet. Laut Herstellerangabe [Lan18] handelt es sich bei dieser Schnittstelle ebenfalls um eine Da-

tenschnittstelle gem. DIN EN 62056-21. Bei der Einstellung der RS232-Peripherie kann es also zu zählerabhängigen Unterschieden kommen.

Um einen Empfang der Zählerstände zu ermöglichen, müssen die Infrarotsignale des Zählers zunächst in digitale elektrische Signale gewandelt werden. Für die Wandlung von Infrarotsignalen zu elektrischen Signalen wurde eine Photodiode vom Typ SFH 213 FA [Osr14] gewählt. Laut Datenblatt fließt bei einer Sperrspannung von $V_R = 20\,\mathrm{V}$ ein Dunkelstrom $I_R \leq 5\,\mathrm{nA}$. Bei $V_R = 5\,\mathrm{V}$ und bei einer Strahlungsleistung von $1\,\mathrm{mW\,cm^{-2}}$ ist ein Photostrom $I_P \geq 65\,\mu\mathrm{A}$ zu erwarten. Bei Anwendung der Photodiode in der Empfängerschaltung sind diese Rahmenparameter, sowohl die 20 V Sperrspannung als auch die genaue Strahlungsleistung nicht identisch mit den Vorgaben aus dem Datenblatt. Allerdings können der Dunkelstrom $I_R \approx 5\,\mathrm{nA}$ und der Photostrom $I_P \approx 65\,\mu\mathrm{A}$ als grobe Vorgaben zur Dimensionierung der Empfängerschaltung verwendet werden.

Um die Stromsignale der Photodiode zu verstärken, bietet sich ein einfacher NPN-Transistor an, gewählt wurde der BC548B. Mit der Beschaltung aus Abb. 3.1 und $200 \leq h_{FE} \leq 450$ [Con12], ergibt sich ohne IR-Bestrahlung am Ausgang bei $V_{CC} = 3{,}3\,\mathrm{V}$ $U_{out} \approx V_{CC} - R_1 \cdot 5\,\mathrm{nA} \cdot h_{FE} \approx 3{,}3\,\mathrm{V} \approx V_{CC}$. Mit IR-Bestrahlung bei $I_P \approx 65\,\mu\mathrm{A}$ ergäbe sich nach der obigen Formel eine negative Ausgangsspannung, in der Praxis geht der Transistor in Sättigung und $U_{out} \approx 0\,\mathrm{V}$.

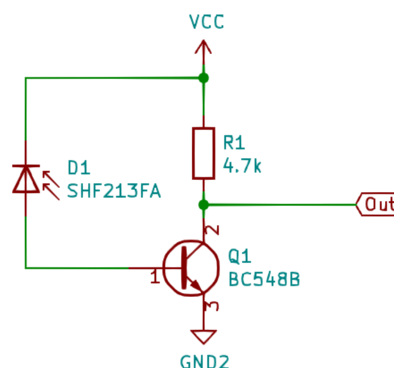


Abbildung 3.1.: Die Infrarotempfängerschaltung

Anhand der Berechnung lässt sich erkennen, dass die Verstärkerschaltung das Eingangssignal invertiert: bei Dunkelheit wird eine digitale 1 erzeugt, bei IR-Belichtung

eine digitale 0. Dieses Verhalten kann schaltungstechnisch mit einem einfach Inverter, wie in Abb. 3.2 zu sehen, behoben werden.

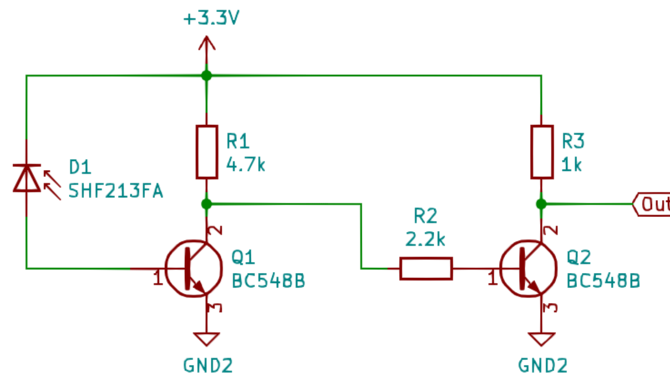


Abbildung 3.2.: Die Infrarotempfängerschaltung mit Inverter

Nach Entwicklung der Schaltung konnte experimentell festgestellt werden, dass der getestete Stromzähler der Firma Landis+Gyr bereits beim Senden das serielle Signal invertiert, sodass ein Inverter beim Empfänger entfällt. Daher wurde die Inverterstufe auf der fertigen Platine durch einen Jumper konfigurierbar ausgelegt (s. Abschnitt 5.1). So kann Anwendungsspezifisch die Schaltung aus Abb. 3.1 oder aus Abb. 3.2 verwendet werden.

Das Signal U_{out} der Empfängerschaltung kann direkt mit dem RX-Pin des Microcontrollers verbunden werden.

3.3. Microcontroller

Für das Empfangen und Verarbeiten der Daten wurde als Microcontroller ein D1 mini ausgewählt, welcher auf einem ESP8266 Prozessor basiert. Dieser verfügt über eine integrierte WLAN-Schnittstelle und ausreichend Peripherie, um den Anforderungen gerecht zu werden. Die serielle Schnittstelle des Microcontrollers kann außerdem direkt genutzt werden, um das serielle Signal vom Stromzähler mittels der bereits vorgestellten Empfängerschaltung relativ einfach einzulesen. Außerdem ist der Microcontroller mit der Arduino IDE programmierbar, was die Vorteile einer einfachen Programmierung und der gleichzeitig großen Vielfalt an Bibliotheken und Erweiterungen für diese Platt-

form kombiniert. Trotz seiner geringen Größe und dem vernachlässigbaren Gewicht verfügt das Board dennoch über ausreichend RAM und Flash-Speicher, eine Taktgeschwindigkeit von 80 bzw. 160 Mhz und einer I^2C-Schnittstelle zur Kommunikation mit der RTC, womit alle Anforderungen erfüllt sind und er sich perfekt zur Umsetztung des Projekts eignet [WEM21].

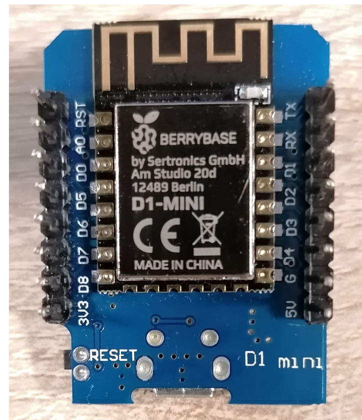


Abbildung 3.3.: ESP 8266 D1 Mini

3.4. Energieverbrauch

Da das Infrarotempfängerkonzept eine vollständig kabellose Anbindung des Empfängeraufbaus vorsieht, ist die Energieversorgung über einen Akku notwendig. Als Akku wurde ein Lithium-Ionen Akku im 18650 Formfaktor gewählt. Für das Akkumanagement, d.h. Laden des Akkus und Unterspannungsschutz, wird ein fertiges BMS-Board verwendet. Das gewählte BMS-Board ist unter der Bezeichnung „Wemos 18650 battery shield V3“ auf diversen Onlinemarktplätzen zu finden.
Das Board wird mit einer 18650 Lithium-Ionen-Zelle bestückt und liefert 3 V und 5 V Ausgangsspannung, die bei Unterspannung der Akkuzelle automatisch abgeschaltet werden. Weiterhin kann der Akku im BMS-Board bequem per Micro-USB aufgeladen werden.
Damit der Empfänger einen möglichst nützlichen Smart Home-Sensor darstellt, muss eine hohe Akkulebensdauer sichergestellt sein, da bei einem Empfänger, der sehr häufig aufgeladen werden muss auch das manuelle notieren der Stromzählerstände einen vergleichbaren Aufwand verursachen würde.

Bei einem handelsüblichen 18650 Lithium-Ionen-Akku kann von einer Kapazität in der Größenordnung von 2700 mAh ausgegangen werden [Pan]. Bei einer geschätzten Stromaufnahme der Empfängerschaltung im Dauerbetrieb von ca. 150 mA, ergäbe sich eine Akkulebensdauer von ca. 18 h, ein völlig inakzeptabler Wert. Aus dieser Überschlagsrechnung wird klar, dass die Empfängerschaltung in zwei Zuständen, einem Aktivzustand mit großem Energieverbrauch und einem möglichst sparsamen Passivzustand, realisiert werden muss.

3.4.1. Abschätzung der Akkulebensdauer

Um eine Realisierung dieser beiden Zustände zu beurteilen, ist ein Modell der Akkulebensdauer in Abhängigkeit der Parameter der Zustände (Verweildauer im Zustand und Stromaufnahme) sinnvoll.
Der Aktivzustand wird charakterisiert durch die Dauer t_{on} mit mittlerem Versorgungsstrom I_{on}, der in einem Zyklus der Periode T_{cycle} auftritt, der Standby-Zustand wir charakterisiert durch den Ruhestrom I_{off}.
Mit Ladung $C = \int I(\tau)d\tau$ ergibt sich näherungsweise eine Ladungsaufnahme der Schaltung von

$$C_{auf}(t) = I_{on} \cdot \frac{t_{on}}{T_{cycle}} \cdot t + I_{off} \cdot t. \tag{3.1}$$

Dabei wurden unter den Annahmen $t \gg T_{cycle}$ und $\frac{t_{on}}{T_{cycle}} \ll 1$ einige Vereinfachungen vorgenommen.

Modelliert man nun den Akku der Kapazität C_A (z.B. 2800 mAh) als simple Ladungsquelle, die die Schaltung versorgt, bis C_A „aufgebraucht“ ist, ergibt sich durch Gleichsetzen von $C_{auf}(t)$ und C_A eine geschätzte Lebensdauer

$$t_L = \frac{C_A}{I_{on} \cdot t_{on}/T_{cycle} + I_{off}}. \tag{3.2}$$

Für einen Ruhestrom von $I_{off} = 2\,\mu A$, eine Aktivzeit $t_{on} = 30\,s$, $T_{cycle} = 12\,h$, $C_A = 2800\,mAh$ und einen mittleren Versorgungsstrom $I_{on} = 150\,mA$ ergibt sich beispielsweise aus der Abschätzung $t_L = 1099\,d$. Wird bei den selben Parametern ein Ruhestrom von $I_{off} = 0{,}2\,mA$ angesetzt, drittelt sich die geschätzte Lebensdauer auf ca. 384 d.

Es ist fragwürdig, ob in der Praxis tatsächlich eine derartig hohe Lebensdauer erreichbar ist, mitunter, da kein realistisches Akkumodell verwendet wurde.
Allerdings ist die hohe Abschätzung der Lebensdauer eine gute Indikation dafür, dass die reale Lebensdauer nicht ausschließlich durch den Energieverbrauch der Schaltung sondern durch Eigenschaften des Akkus limitiert wird und somit eine weitere Optimierung des Energieverbrauchs nicht unbedingt zu einem signifikanten Anstieg der Akkulebensdauer führen würde. Um die Lebensdauer genauer zu bestimmen, ist allerdings eine Echtzeitmessung oder die Verwendung eines präzisen Akkumodells erforderlich.

3.4.2. Schaltungstechnische Realisierung des Standbyzustands

Der ESP8266 verfügt Hardwareseitig bereits über einen sogenannten „Deep-sleep“-Modus [Esp19]. In diesem Modus wird der Großteil des Mikrocontrollers deaktiviert und damit der Energieverbrauch deutlich gesenkt [Esp19]. Mittels der internen Uhr (RTC) des ESP wird der Microcontroller in bestimmten Intervallen wieder „geweckt“ [Esp19]. Durch diesen Mechanismus könnte der zuvor beschriebene Energiesparmodus umgesetzt werden. Da für dieses Projekt allerdings der ESP8266 nicht einzeln sondern in Form eines Entwicklungsboards mit zusätzlicher Peripherie (z.B. Spannungswandler und USB zu UART Wandler) verwendet wird und auch im Arbeitspunkt der IR-Empfängerschaltung ein geringer Strom fließt, würde der Deep-sleep-Modus zwar den Energieverbrauch des ESP reduzieren, alle weiteren Komponenten wären davon allerdings nicht betroffen. Wie die Lebensdauerabschätzung im vorherigen Abschnitt zeigt, ist ein Standbystrom im Microamperebereich wünschenswert, daher ist der Deep-sleep-Modus für diesen Zweck nicht ausreichend.
Eine alternative Lösung, die im Rahmen dieses Projekts gewählt wurde, ist im Standbyzustand die Energieversorgung der kompletten Schaltung zu unterbrechen.
Dafür wird mittels eines p-MOSFET ein high-side Schalter realisiert, der die Versorgungsspannung vom BMS-Board zum Empfänger schaltet. Der MOSFET wird angesteuert durch eine externe Echtzeituhr (RTC) vom Typ DS3231 [max15]. Die DS3231 liefert via I^2C-Schnittstelle nach einmaliger Konfiguration fortlaufend das aktuelle Datum und die aktuelle Uhrzeit. Die RTC wird dabei von einer Lithium-Knopfzelle versorgt und ist damit in ihrer Funktion unabhängig von einer externen Spannungsversorgung. Die für dieses Projekt wichtigste Funktion der DS3231 sind die zwei programmierbaren Alarme der RTC.

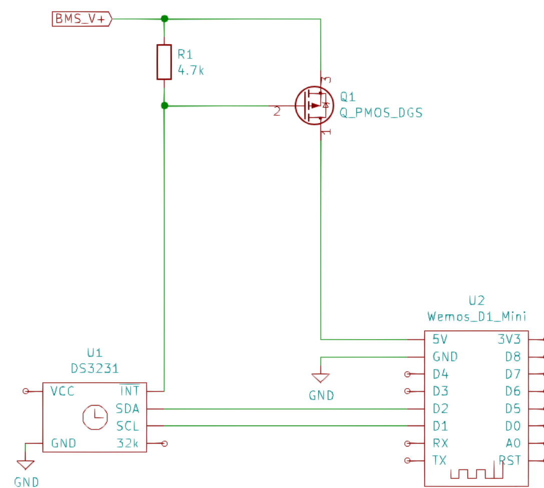


Abbildung 3.4.: Schematische Darstellung der geschalteten Energieversorgung

Mittels I^{2}C können in den Registern der RTC zwei Alarme dazu programmiert werden, in bestimmten Intervallen (minütlich, stündlich, täglich, ...) den $\overline{\text{INT}}$-Pin der RTC auf Masse zu ziehen [max15]. Dabei ist sehr hilfreich, dass die Alarmsignale selbsthaltend sind, d.h. wird ein Alarm ausgelöst bleibt $\overline{\text{INT}}$ auf Masse gezogen, bis via I^{2}C das jeweilige Alarmflag (A1E bzw. A2E) in den RTC-Registern zurückgesetzt wird [max15]. Das Signal eignet sich also direkt zum Schalten der Energieversorgung der Empfängerschaltung. Weiterhin handelt es sich bei dem $\overline{\text{INT}}$-Pin um einen open-drain Ausgang, d.h. der Pin muss mit einem externen Pull-Up-Widerstand beschaltet werden [max15], der die high-Spannung des Signals vorgibt. Damit bietet sich das Signal zum Steuern des p-MOSFET sehr an, da durch einen Pull-Up auf die Sourcespannung des MOSFET der p-MOSFET korrekt durch das Signal geschaltet wird.
In der praktischen Erprobung der Schaltung hat sich gezeigt, dass teils beim Ausschalten Glitches auftreten können, bei denen die Versorgung nicht vollständig ausgeschaltet wird, sondern die Spannung am Drain des MOSFET (die nahe 0 V liegen sollte) bei beispielsweise 1,6 V hängen bleibt und der Ausschaltvorgang somit fehlschlägt. Um ein sicheres Ausschalten zu gewährleisten, wurde zwischen das $\overline{\text{INT}}$-Signal der RTC mit Pull-Up und den MOSFET ein Spannungspuffer geschaltet (realisiert durch zwei nacheinander geschaltete NAND-Gatter aus einem CD4011B IC, Schaltung siehe Abschnitt 5.1).
Nach dieser Änderung konnten keine Glitches mehr festgestellt werden.

Als p-MOSFET wurde ein NX2301P gewählt [Nex10]. Das wichtigste Auswahlkriterium für den MOSFET war in dieser Funktion ein betragsmäßig ausreichend geringes V_{GS} um den MOSFET einzuschalten. Dem Datenblatt lässt sich bei $V_{GS} = -1{,}8\,\text{V}$ ein maximales R_{DSon} von 270 mΩ entnehmen. Dieser R_{DSon}-Wert ist für den Aktivzustand der Schaltung völlig ausreichend - bei einer Stromaufnahme von 150 mA fallen über dem MOSFET maximal 41 mV ab, der Leistungsverlust über dem MOSFET liegt im einstelligen Milliwattbereich. Die Gate-Source Spannung von $-1{,}8\,\text{V}$ ist ebenfalls komfortabel erreichbar, bei einer Akkuspannung von 3 V wird ein V_{GS} von knapp $-3\,\text{V}$ im Aktivzustand erreicht.

3.4.3. Messtechnische Überprüfung des Energieverbrauchs

Zur messtechnischen Überprüfung des Energieverbrauchs wurde zunächst das BMS-Board mit 3,7 V versorgt und die Stromaufnahme ohne angeschlossene Last bestimmt. Es konnte ein Wert von ca. 0,33 mA gemessen werden. Wie bereits bei der Abschätzung der Akkulebensdauer festgestellt wurde, handelt es sich hierbei um einen relativ hohen Standbyverbrauch. Es ist eine naheliegende Vermutung, dass dieser Stromverbrauch ohne Last durch den DC-DC Step-Up Konverter auf dem BMS-Board verursacht wird, der die 5 V Ausgangsspannung des Boards generiert.

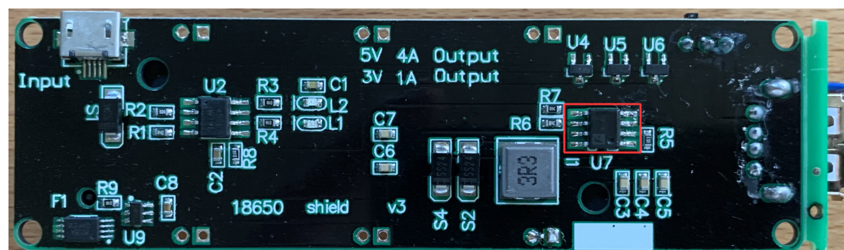


Abbildung 3.5.: Die Rückseite des BMS-Boards. Rot markiert der Step-Up Konverter

Durch Position der Induktivität und Nachschlagen der IC-Nummern konnte U7 (s. Abb. 3.5) als Step-Up Konverter des Typs FP6298 identifiziert werden. Durch Entfernen des Step-Up ICs von der Platine konnte die Stromaufnahme des BMS-Boards ohne angeschlossene Last auf unter 0,2 µA reduziert werden. Durch diese Modifikation geht selbstverständlich der 5 V Ausgang des BMS-Boards verloren, stattdessen steht maximal die Zellspannung zur Verfügung. Das stellt allerdings kein Problem dar, da experimen-

tell festgestellt wurde, dass eine Versorgungsspannung im Bereich von 3 V bis 3,7 V für den Betrieb der Empfängerschaltung ausreichend ist. Betrachtet man die Entladecharakteristik eines typischen 18650 Lithium-Ionen Akkus (s. Abb. 3.6), ist ersichtlich, dass in diesem Spannungsbereich der größte Teil der Akkuladung ausgenutzt werden kann.

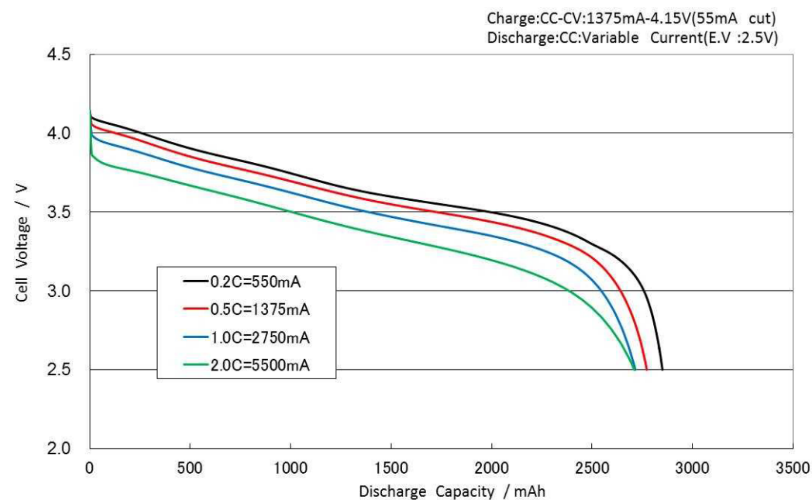


Abbildung 3.6.: Entladekurve eines 18650 Lithium-Ionen Akkus [Pan]

Anschließend wurde der Energieverbrauch der vollständigen Empfängerschaltung zusammen mit dem BMS-Board charakterisiert. Dafür wurden erneut 3,7 V an den Zellanschlüssen des BMS-Boards eingespeist und der Stromfluss am Zellanschluss gemessen.
Es konnte im Aktivzustand ein mittlerer Stromverbrauch von ca. 150 mA und im Standby-Zustand ein Stromverbrauch zwischen 0,3 µA und 1,5 µA gemessen werden. Mit der Abschätzung aus Abschnitt 3.4.1 ergibt sich mit diesen Werten bei $t_{on} = 30\,\text{s}$, $T_{cycle} = 12\,\text{h}$ und $C_A = 2800\,\text{mAh}$ eine geschätzte Lebensdauer von $t_L \approx 1104\,\text{d}$. Obwohl die reale Akkulebensdauer sicherlich deutlich geringer ausfallen wird, lässt dieser hohe Wert vermuten, dass Verbraucherseitig die Voraussetzungen für eine hohe Akkulebensdauer erfüllt sind.

3.5. Platine

Um alle benötigten Komponenten sicher miteinander zu verbinden wurde entschieden eine eigene Platine zu entwerfen. Zwar hätte ein Aufbau auf einer Lochrasterplatine

sicherlich auch funktioniert, jedoch wäre dieser bei der Anzahl der Komponenten und deren Größe, wie z.B. dem Microcontroller oder den NAND-Gates, sehr unübersichtlich geworden. Außerdem konnten die verschiedenen Konfigurationsmöglichkeiten so leichter implementiert und beschriftet werden.

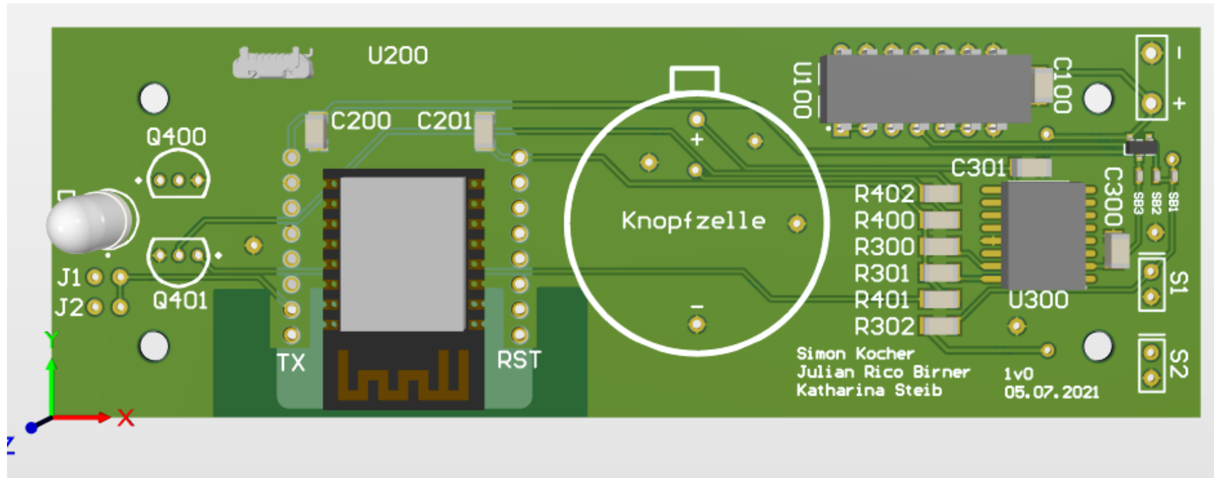


Abbildung 3.7.: 3D-Rendering der Platine in Altium

Wie bereits im vorherigen Kapitel beschrieben kam es zu Glitches beim Abschalten des pMOSFETs. Um sowohl die Möglichkeit des direkten Abschaltens, als auch die des gepufferten Abschaltens zu haben, wurden Lötbrücken platziert. Somit kann ohne besonders viel Aufwand die Konfiguration geändert werden.

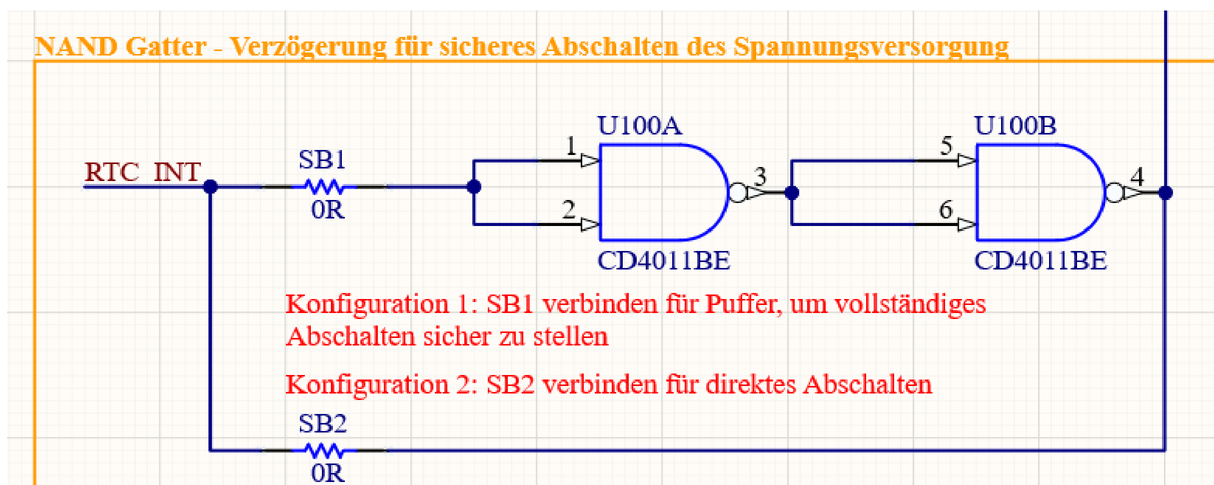


Abbildung 3.8.: Lötbrücke 1 und 2 für Abschaltung mit oder ohne Puffer

Eine weitere Lötbrücke wurde platziert um die RTC im aktiven Zustand aus dem Akku zu speisen. Damit wird vermieden, dass die Knopfzelle zu schnell entleert wird.

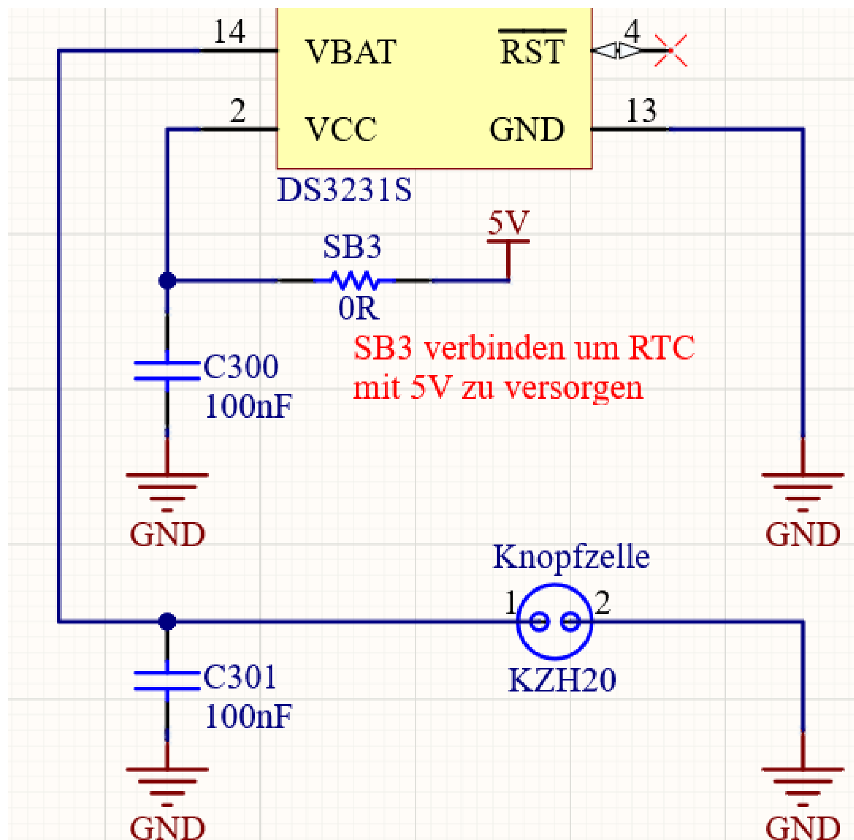


Abbildung 3.9.: Lötbrücke 3 für Versorgung der RTC per Akku

Außerdem wurde beim Testen mit einem anderen Stromzähler entdeckt, dass der Stromzähler des Herstellers „Landis+Gyr“ bereits ein invertiertes Signal sendet. Da der Stromzähler, an dem der Empfänger letzten Endes betrieben wird, nicht getestet werden konnte, wurden hierfür Jumper eingebaut. Somit kann die Inverterstufe im Empfangspfad leichter aktiviert oder deaktiviert werden.
Leider wurde nach Fertigstellung der Platine bemerkt, dass J2 vor R402 gesetzt werden müsste.

Im Auslieferungszustand wurden die Lötbrücken SB1 (Gepuffertes Abschalten des pMOSFET) und SB3 (Speisen der RTC aus dem Akku im aktiven Zustand) sowie der Jumper 1 (mit Invertierung im Empfangspad) gesetzt.

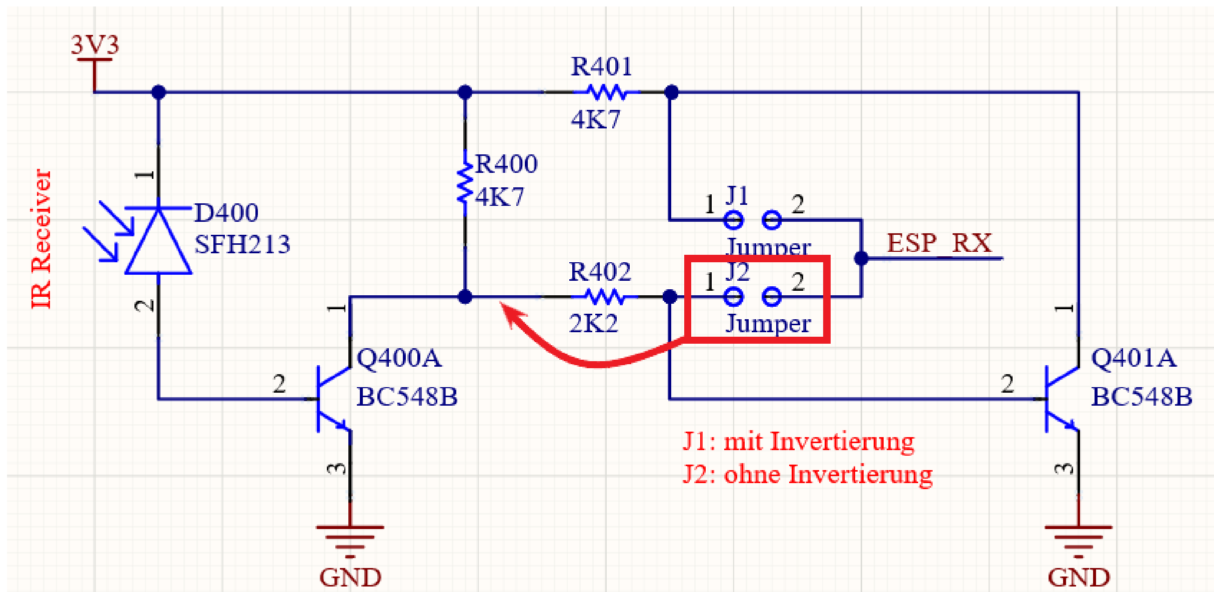


Abbildung 3.10.: Jumper zum Aktivieren und Deaktivieren der Inverterstufe mit nötiger Korrektur

3.6. Software

Das komplette Programm sowohl für den Empfänger, als auch für den Teststand, sind im Anhang 5.3 zu finden.

3.6.1. Programmablauf

Der ESP wurde mit der Arduino IDE programmiert. Diese ermöglicht den Zugang zu beliebiger Hardware, solange der korrekte Treiber installiert ist und es eine Anbindung an die Arduino IDE gibt. Als Schnittstelle gibt es eine setup()-Funktion, welche einmalig zu Beginn des Programms abgearbeitet wird und eine loop()-Funktion, welche nach dem Setup zyklisch als Endlosschleife abläuft.
Im Setup werden zunächst einzelne Pins initialisiert, danach wird die serielle Kommunikation gestartet, woraufhin die Verbindung zum WLAN hergestellt wird. Ist dies erfolgreich, versucht der ESP als nächstes sich mit dem MQTT-Broker zu verbinden. Sobald auch das funktioniert hat ist die Initialisierung beendet. Sollte jedoch keine Verbindung zum WLAN oder dem MQTT-Broker hergestellt werden können, wird nach einer bestimmten Zeit das Ausschalten initialisiert.

3.6.2. RTC

Die Spannungsversorgung des Microcontrollers wird von der RTC gesteuert. Diese arbeitet mit internen Alarmen und schaltet den INT-Pin (low-aktiv), sobald ein aktiver Alarm abgelaufen ist. Danach bleibt sie aktiv, bis über I^2C der Befehl zum Ausschalten gesendet wird. Dies geschieht, indem der Microcontroller nach Ablauf seines Programms im Control-Register der RTC die Flags für die Alarme zurücksetzt. Sobald das passiert ist, setzt die RTC den INT-Pin zurück und unterbricht damit die Stromversorgung für den ESP. Nach Ablaufen des Alarms beginnt dieser Zyklus von neuem.

3.6.3. WifiManager

Kernanforderung des Projekts war die drahtlose Kommunikation zwischen Empfänger und Smart-Home-Controller. Da uns aus ersichtlichen Gründen jedoch zur Projektlaufzeit die Zugangsdaten zum endgültigen WLAN nicht zur Verfügung standen war es nötig, dem Bediener die Möglichkeit zu geben, eine gültige WLAN Konfiguration an den Microcontroller zu übermitteln, ohne ihn dafür jedes Mal neu flashen zu müssen. Diese Funktionalität gibt es bereits in Form einer Bibliothek mit dem Namen WifiManager [Git21].

Mit Hilfe der Bibliothek startet der Microcontroller im Station-Mode und versucht, sich mit ggf. vorher gespeicherten Zugangsdaten anzumelden. Gelingt dies nicht oder ist noch keine Konfiguration hinterlegt, wechselt der ESP in Access-Point-Mode und startet einen eigenen Webserver. Nun kann sich der Bediener mit jedem beliebigen, WLAN-fähigen Gerät am WLAN des ESP anmelden und wird zu einer Anmeldeseite weitergeleitet, wo nun die WLAN Zugangsdaten eingegeben werden können. Sobald sich der ESP erfolgreich mit dem angegebenen Netzwerk verbunden hat wird der Programmablauf fortgesetzt.

Zwar bietet die Bibliothek auch Felder für MQTT-Broker IP und Port an, dies hat jedoch zum Zeitpunkt des Projekts nicht zuverlässig funktioniert und wurde deshalb aus dem Programm entfernt. Sollte die Bibliothek dahingehend verbessert werden, wäre es sinnvoll, dieses Feature nachzurüsten, um nicht jedes Mal den Microcontroller flashen zu müssen wenn ein anderer MQTT-Broker verwendet wird. Da dem Broker, welcher in diesem Fall ein Raspberry Pi ist, entweder in der internen Konfiguration des Raspberry Pi eine statische oder über den Router immer dieselbe IP zugewiesen werden

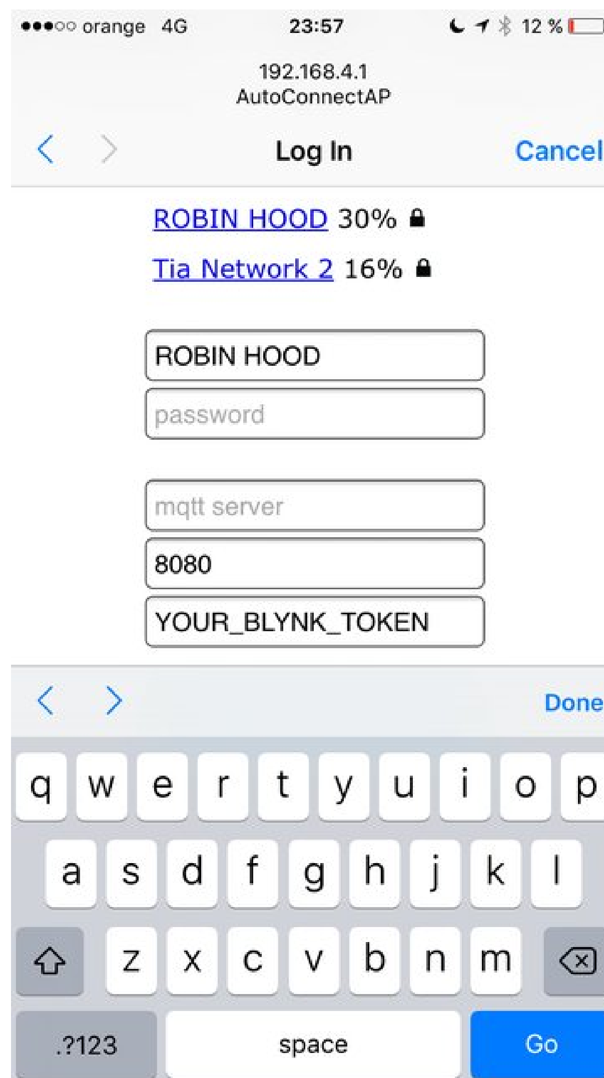


Abbildung 3.11.: Anmeldeseite des WifiManager aus Sicht des Bedieners

kann und das Gerät nicht für wechselnde MQTT-Broker gedacht ist, kann auf dieses Feature aber auch verzichtet werden. IP und Port des Brokers sind daher im Programm in Variablen hinterlegt.

3.6.4. Einlesen der SML-Daten

Hauptauftrag der Software ist das erfolgreiche Einlesen und Verarbeiten der eingehenden Daten. Da die serielle Schnittstelle des ESP genutzt wird kann dafür auch die Standardbibliothek von Arduino zum Einlesen serieller Daten genutzt werden. Sobald Daten verfügbar sind werden diese in einen internem Buffer, ein Int-Array der Länge

1000, gespeichert. Danach wird dieser Buffer auf bestimmte Zeichenfolgen untersucht. Da die Daten per SML kodiert sind muss auf eine bestimmte Zeichenfolge aus dem SML-Protokoll getriggert werden, hierbei wurde sich auf die OBIS-Kennzahl 1.8.0 (vgl. 3.1.2) konzentriert.

```
// Daten von Stromzähler: Gesamtverbrauch herausfiltern
if (                      /* OBIS Kennung: 1-0.1.8.0*255 = 01 00 */
                          /* 01 08 00 FF */
BUFFER[j]   == 0x77 &&    /* 77 - SML_Message.messageBody.SML_GetList */
                          /*  _Reponse.valList.valListEntry (Sequence) */
BUFFER[j+1] == 0x07 &&    /* 07 - objName (TL[1] + octet-string[6] */
BUFFER[j+2] == 0x01 &&    /* 01 - objName Teil A */
BUFFER[j+3] == 0x00 &&    /* 00 - objName Teil B */
BUFFER[j+4] == 0x01 &&    /* 01 - objName Teil C */
BUFFER[j+5] == 0x08 &&    /* 08 - objName Teil D */
BUFFER[j+6] == 0x00 &&    /* 00 - objName Teil E */
BUFFER[j+7] == 0xFF)      /* FF - objName Teil F */
                          /* xx - status         */
                          /* xx - valTime        */
                          /* xx - unit           */
                          /* xx - scaler         */
```

Nun kann es auch passieren dass der ESP genau dann hochfährt, wenn gerade ein SML Paket verschickt wurde, der relevante Teil aber bereits gesendet wurde und für den ESP verloren ist. In diesem Fall darf der ESP nach dem Einlesen der Daten nicht sofort heruntergefahren werden, sondern es muss auf das nächste Paket gewartet werden, welches den Gesamtverbrauch beinhaltet.

Der Gesamtverbrauch hat eine Auflösung von 0,1 W h und ist als Type-Length-Field kodiert. Dieses TL-Feld definiert die Bedeutung des aktuellen und der folgenden Bytes. Aus dem Datenblatt des Stromzählers lässt sich schließen, dass es sich dabei um eine 64-Bit große vorzeichenbehaftete Festpunktzahl (Integer) handelt, was sich zu „0x59h“ kodiert.

Bitindex	MSB, D7	6	5	4	3	2	1	LSB, D0
Verwendet für Merkmal ‚weiteres Byte zum TL-Field folgt‘	1	X	X	X	X	X	X	X
Verwendet für Merkmal ‚kein weiteres Byte zum TL-Field folgt‘	0	X	X	X	X	X	X	X
Verwendet für Merkmal ‚Datentyp Octet String‘ verwenden	X	0	0	0	L	L	L	L
Verwendet für Merkmal ‚Boolean‘ verwenden	0	1	0	0	L	L	L	L
Verwendet für Merkmal ‚Datentyp Integer‘ verwenden	X	1	0	1	L	L	L	L
Verwendet für Merkmal ‚Datentyp Unsigned‘ verwenden	X	1	1	0	L	L	L	L

Abbildung 3.12.: Type-Length-Field Definition aus [Bun13], siehe unter Abbildung

Als nächstes müssen die Bytes aus dem Buffer zur eigentlichen Zahl zusammengesetzt werden:

```
// mWh aus Buffer rekonstruieren
long long mWh = ((long long)BUFFER[j+1]) << 56 |
                ((long long)BUFFER[j+2]) << 48 |
                ((long long)BUFFER[j+3]) << 40 |
                ((long long)BUFFER[j+4]) << 32 |
                ((long long)BUFFER[j+5]) << 24 |
                ((long long)BUFFER[j+6]) << 16 |
                ((long long)BUFFER[j+7]) <<  8 |
                ((long long)BUFFER[j+8]);
```

Abschließend wird diese Zahl in kWh umgerechnet und dieser Messwert an den MQTT-Broker gesendet.

3.6.5. MQTT

MQTT (=Message Queuing Telemetry Transport [MQT20]) ist ein Nachrichtenprotokoll zur Vernetzung von IoT-Geräten und funktioniert nach einem Client-Server Modell. Die Clients können Daten versenden, indem sie eine Nachricht in einem Topic an den Broker senden. Auf der anderen Seite können Daten empfangen werden, indem Topics abonniert werden. Das besondere dabei ist, dass Clients Anfragen und Daten ausschließlich über den MQTT-Broker erhalten, welcher den Datenstrom verwaltet. Im

Fall dieser Projektarbeit ist der ESP ein MQTT-Client, welcher den Gesamtverbrauch an das Topic „smartmeter“ sendet. Der Raspberry Pi ist in diesem Fall der MQTT-Broker. Da MQTT bereits weit verbreitet ist und besonders für IoT-Geräte bereits ausreichend Implementierungen vorhanden sind wurde hier auf eine bereits bestehende Lösung zurückgegriffen. Dafür wurde die Bibliothek „PubSubClient“ verwendet, welche eine einfache Implementation bereitstellt. Für Details sei an dieser Stelle auf die GitHub-Seite verwiesen. [Git20]

3.6.6. NodeRed und Datenbank

Die Daten aus dem MQTT-Paket werden in einer MySQL-Datenbank auf dem Raspberry Pi gespeichert. Der genaue Aufbau dieser Datenbank wird weiter unten im Kapitel 4 beschrieben. Um die Daten vom ESP in der Datenbank abzuspeichern wird ein NodeRed-Flow verwendet. Um die Zuverlässigkeit der Daten zu erhöhen wird vor dem Einfügen in die Tabelle eine Plausibilitätsprüfung durchgeführt, da Tests ergeben haben dass unvollständige Datensätze vom Stromzähler zu unplausiblen Messwerten führen können. Deshalb wird ein neuer Messwert zuerst mit dem letzten Messwert verglichen. Der neue Wert wird nur dann in der Datenbank gespeichert wenn er größer und höchstens doppelt so groß ist wie der letzte Wert (in den Tests wurden in seltenen Fällen fälschlicherweise Nullwerte oder extrem hohe Werte bemerkt).

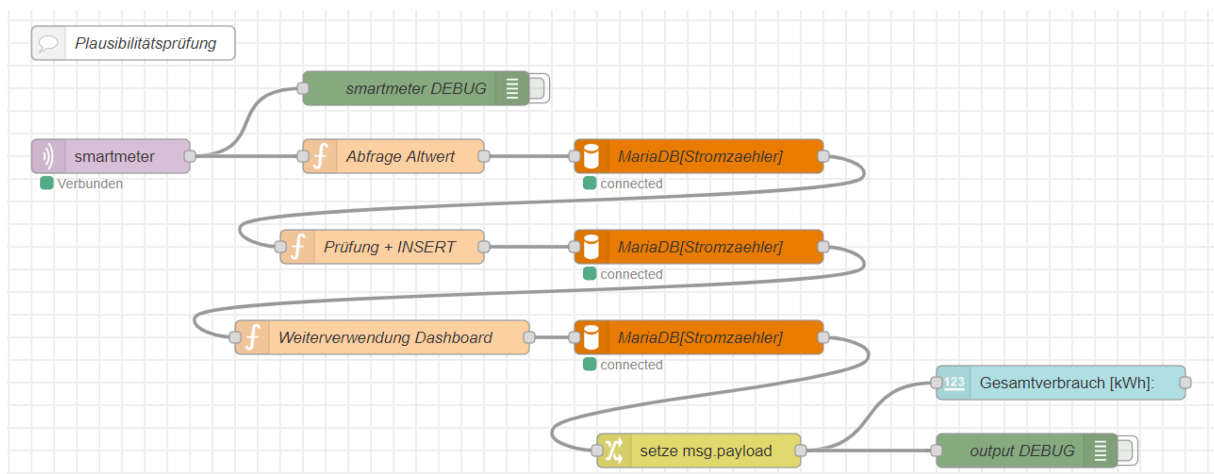


Abbildung 3.13.: NodeRed-Flow zum Speichern und Anzeigen des Messwertes

Abschließend wird der Wert auf einem Dashboard angezeigt. Dieses wurde bewusst simpel gehalten, da Herr Prof. Klehn hier eigene Ideen umsetzen möchte.

Abbildung 3.14.: NodeRed-Dasboard zum Anzeigen des letzten Messwerts

3.7. Gehäuse

Für den Empfängeraufbau aus BMS-Board, Platine, Photodiode, Ein- und Resetschalter wurde ein 3D-Druckgehäuse gezeichnet und gedruckt.

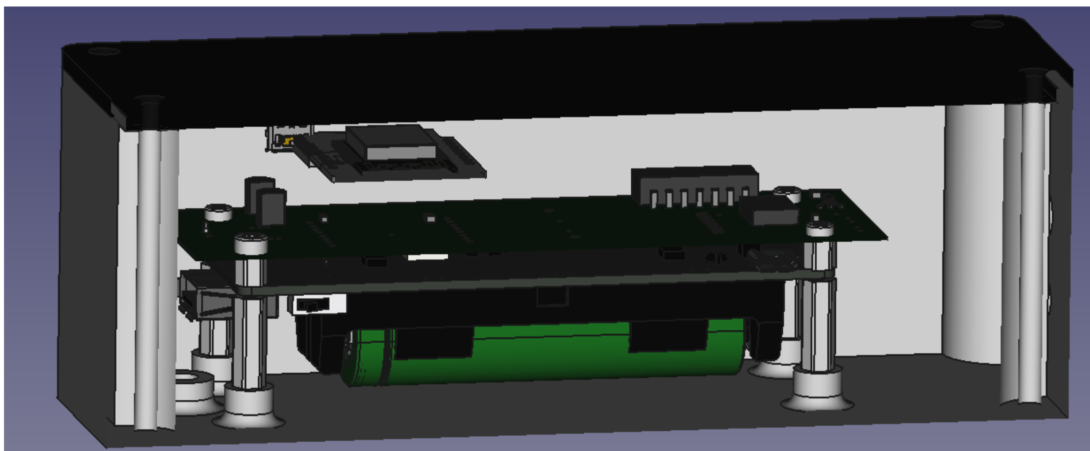


Abbildung 3.15.: CAD-Schnittbild des Gehäuses. Modellquelle BMS-Board: [Gra21]

Das Gehäuse kann mit einem Deckel verschlossen werden, der durch vier Schrauben befestigt wird. Die Platine und das BMS-Board werden durch metrische Distanzbolzen im Gehäuse und aneinander befestigt. Sämtliche Gewinde im gedruckten Material werden durch metallische Gewindeeinsätze realisiert.

Für die Photodiode befindet sich im Boden des Gehäuses eine Durchführung, in die die Diode eingepresst wird. An der Außenseite der Durchführung ist ein Ringmagnet montiert, der den Empfängeraufbau an die magnetische Fläche des Stromzählers fixiert.

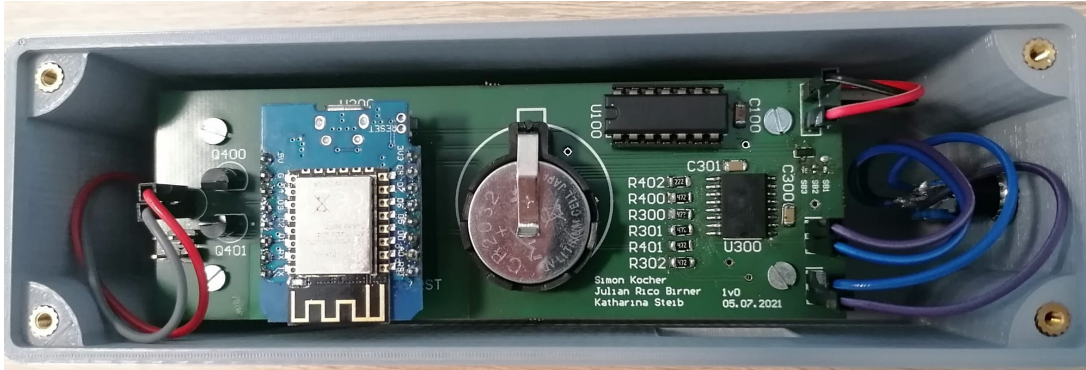


Abbildung 3.16.: Gehäuse mit allen Komponenten installiert

3.8. Teststrecke

Zum Testen der Funktion wurde eine kleine Teststrecke aufgebaut, mit der es möglich ist die Übertragung ohne den Stromzähler zu simulieren.

3.8.1. Aufbau

Zunächst wurde die Schaltung der Teststrecke auf einem Steckbrett aufgebaut, um zu überprüfen, ob die Grundidee des Projekts umsetzbar ist. Auch die bestellten Bauteile und Dimensionierungen konnten so überprüft werden. Da während der Arbeit am Projekt zunächst kein Stromzähler zur Verfügung stand wurde der Teststrecke zusätzlich zu dem bereits in Kapitel 3.3 erläuterten ESP ein Arduino UNO hinzugefügt. So konnte überprüft werden, ob auch die Kommunikation zwischen zwei komplett voneinander getrennten Systemen funktioniert. Der ESP dient also weiterhin als Empfänger, während der Arduino den Stromzähler als Sender simuliert.

Wie in Abb. 3.17 zu sehen ist sind die Schaltkreise komplett von einander getrennt. Um die besten Ergebnisse bei der Übertragung zu erzielen, werden die beiden LEDs gegenüber, Spitze an Spitze aufgebaut. Sendet man nun vom TX-Pin des Arduino eine

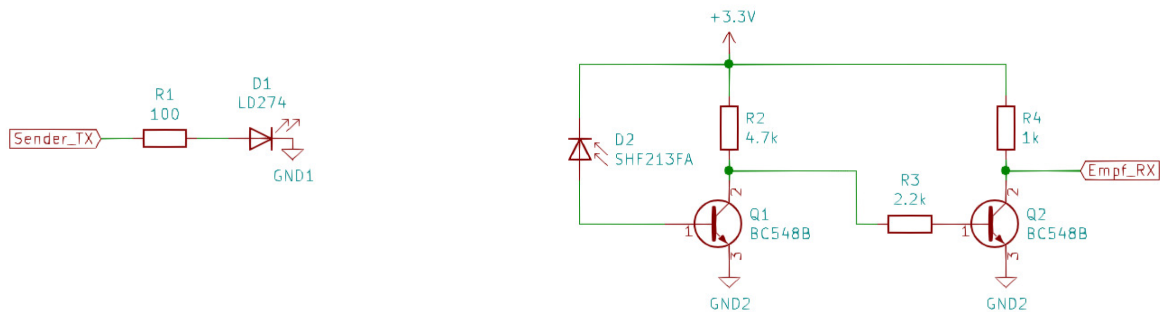


Abbildung 3.17.: Der Schalplan der Teststrecke

Zahl so kann diese am RX-Pin des ESP empfangen werden. Mit der richtigen Dimensionierung von R1 auf 100 Ω haben auch wechselnde Lichtverhältnisse keinen Einfluss auf die Funktion der Schaltung. Die Werte von R3 und R4 sind relativ frei wählbar, solange sie nicht zu klein sind. Nach ersten erfolgreichen Tests wurden für R3 2,2 kΩ und für R4 1 kΩ festgelegt.

Bei der Programmierung ist außerdem aufgefallen, dass das Hochladen des Programmes nur möglich ist, wenn RX- und TX-Pin des jeweiligen Mikrocontroller von der Schaltung getrennt sind. Nachdem die Funktion der Schaltung überprüft war, wurde der Aufbau auf eine Lochstreifenplatine gelötet. Dabei wurde mit Pin-Headern gearbeitet, damit das Ergebnis wie ein Art Arduino-Shield verwendet werden kann. Der Vorteil dabei liegt darin, dass beide Mikrocontroller zwar fest mit der Platine verbunden sind, aber bei Änderungen am Programm auch genauso leicht zum Hochladen getrennt werden können. Außerdem befinden sich die beiden Dioden an der Unterseite des Shields, also im Gebrauch zwischen Shield und Arduino, was die Schaltung noch zusätzlich von äußeren Lichteinflüssen schützt.

3.8.2. Funktionalität

Ein großer Vorteil an der Umsetzung der Teststrecke mit Arduino und ESP liegt darin, dass beide mit der Arduino IDE programmiert werden konnten (siehe Kapitel 3.6.1). Dies bedeutet für den Tester eine deutlich komfortablere Nutzung, da kein Wechsel zwischen verschieden Sprachen nötig ist.

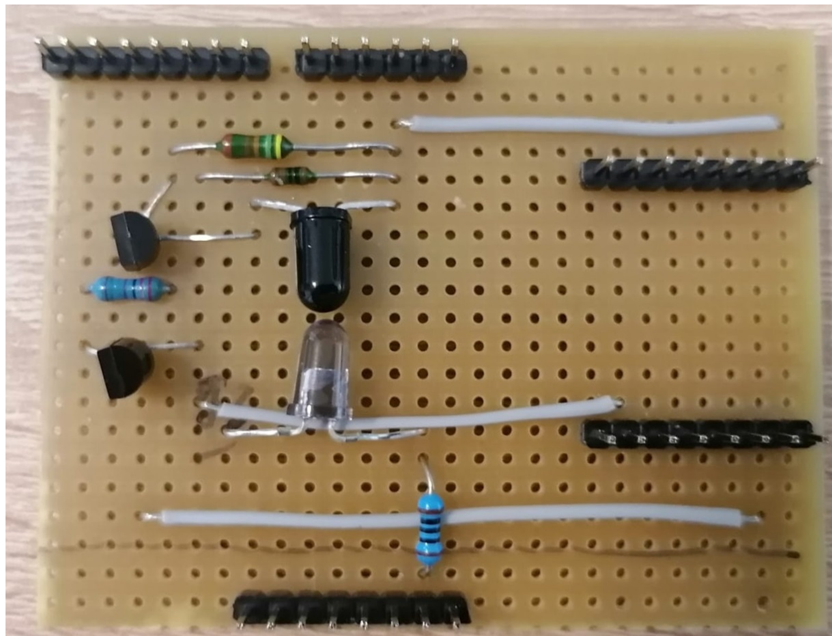


Abbildung 3.18.: Lochstreifenplatine mit Pin-Headern für Arduino (links) und ESP (rechts)

Um den Stromzähler zu simulieren, soll der Arduino einen Gesamtverbrauch in kWh im SML-Format über eine serielle IR-Schnittstelle senden. Den Gesamtverbrauch kann der Bediener dabei selbst bestimmen. Um den Stromzähler korrekt abzubilden wird der kWh-Wert in mWh umgerechnet, bevor er verschickt wird.

Die Verwendung der Teststrecke wird unter 4.3 erklärt, das dazugehörige Programm ist im Anhang 5.3 zu finden.

4. Anleitung

Damit diese Projektarbeit von Grund auf nachgestellt werden kann, wird im Folgenden Schritt für Schritt erklärt, wie man das entwickelte System nachbauen und notwendige Anpassungen vornehmen kann.

4.1. Raspberry Pi

Bevor der Stromzähler ausgelesen und der Empfänger in Betrieb genommen werden kann, muss die MQTT-Konfiguration und die Datenbank auf dem Raspberry Pi vorhanden sein. Alle dazu nötigen Schritte werden in den nächsten Unterkapiteln ausführlich erläutert.

4.1.1. Betriebssystem

Für Testzwecke wurde ein Raspberry Pi von Grund auf neu eingerichtet, um den bereits vorhanden Raspberry Pi möglichst genau abzubilden. Dies fängt beim Betriebssystem an, welches auf einer SD-Karte installiert wird. Diese Karte sollte genügend Speicherplatz für das Betriebssystem, die Programme und die Datenbank haben. Oft wird eine Speicherkarte mit 16GB verwendet, was für die meisten Anwendungsfälle ausreicht.

Als Betriebssystem wurde *Raspberry Pi OS* [RASa] verwendet, welches auf die SD Karte geflasht werden muss. Dafür Prozess wurde das Programm *Etcher* [Bal21] verwendet, Raspberry Pi bietet jedoch auch einen eigenen Imager an.
Sobald das Betriebssystem auf die SD-Karte kopiert ist kann diese in den Raspberry Pi gesteckt werden. Nach dem ersten Start erfolgt die Installation.
Nach der ersten Anmeldung wird empfohlen, die installierten Packages zu updaten. Dafür öffnet man auf dem Raspberry Pi ein Terminal geöffnet und folgende Befehle eingegeben werden:

```
sudo apt-get update
```

```
sudo apt-get upgrade
```

Danach empfiehlt es sich, den Raspberry Pi neu zu starten.

```
sudo reboot
```

4.1.2. Fernzugriff

Da ein Raspberry Pi oft remote verwendet wird müssen zuerst sowohl eine Möglichkeit für Fernzugriff und eine statische IP eingerichtet werden. Für den Fernzugriff im gleichen Netz bieten sich SSH oder Remotedesktop an. SSH (=Secure Shell) bietet die Möglichkeit, sich mit einem geeigneten Programm wie z.B. putty [PuT] per Konsole zu verbinden. Wer eine grafische Oberfläche möchte sollte Remotedesktop (meist mit RDP abgekürzt) oder ähnliche Lösungen wie z.B. VNC verwenden, wobei mit VNC sogar eine Verbindung möglich ist, wenn man sich nicht im gleichen Netz befindet (die Einrichtung dafür ist aber entsprechend umfangreicher). Hier wird nur die Einrichtung von SSH erklärt, es gibt jedoch im Netz weitreichende Erklärungen für weitere Zugriffsmöglichkeiten wie RDP oder VNC.
Da SSH aus Sicherheitsgründen nicht mehr standardmäßig aktiviert ist, muss dies nun nachgeholt werden. Dazu gibt man im Terminal ein:

```
sudo raspi-config
```

In dem sich nun öffnenden Menü muss nun SSH ausgewählt und aktiviert werden.

Dieser Schritt kann auch umgangen werden, indem die SD-Karte mit dem Betriebssystem vor dem ersten Start in einen Card-Reader gesteckt und im „boot“-Verzeichnis eine leere Datei mit dem Namen „ssh“ (ohne Dateiendung) hinterlegt wird. Dies aktiviert automatisch die Möglichkeit des Zugriffs per SSH. [Rasb]
Sobald SSH aktiviert ist kann man sich z.B. mit putty auf den Raspberry Pi verbinden.

Als nächstes sollte man eine statische IP vergeben, da standardmäßig DHCP aktiviert ist und sich somit die IP bei jedem Start ändern kann. Auch hierfür gibt es wieder mehrere Methoden, in diesem Fall wird die IPv4-Konfiguration in der Datei "/etc/network/interfaces“ angepasst. Dafür muss man die Datei zuerst öffnen

```
sudo nano /etc/network/interfaces
```

und dort nun die statische IP für das verwendete Interface (Ethernet oder WLAN) sowie die IP des Gateways und des DNS-Servers eingegeben. Eine Beispielkonfiguration für eine statische IP des Ethernet-Interfaces eth0 mit der IP-Adresse 192.168.178.100 und dem Router (welcher Gateway und DNS-Server ist) mit der IP 192.168.178.1 könnte wie folgt aussehen:

```
# Ethernet
auto eth0
allow-hotplug eth0
iface eth0 inet static
address 192.168.178.100
netmask 255.255.255.0
gateway 192.168.178.1
dns-nameservers 192.168.178.1
```

4.1.3. Datenbank

4.1.3.1. Installation MariaDB

Als Datenbank wird MariaDB verwendet, also wird der Datenbank-Server dafür zuerst installiert.

```
sudo apt install mariadb-server
```

Es wird empfohlen, zur Konfiguration der Sicherheitseinstellungen danach folgenden Befehl auszuführen:

```
sudo mysql_secure_installation
```

Die daraufhin erscheinende Abfrage kann mit "Enter" übersprungen werden. Alle folgenden Abfragen sollten mit ja bzw. "y" beantwortet werden. Damit erhöht sich die Sicherheit der MySQL Installation deutlich.

Der Login in die Datenbank erfolgt mit

```
mysql -u root -p
```

wobei nach "-u" der Benutzer angegeben wird, mit dem sich an der Datenbank angemeldet werden soll und nach der Abfrage noch das Passwort des jeweiligen Benutzers eingegeben werden muss.

4.1.3.2. Passwort für Datenbanklogin zurücksetzen

Gelegentlich kann es passieren, dass das Passwort für einen Datenbankbenutzer vergessen wird. In diesem Fall kann das entsprechende Passwort einfach zurückgesetzt werden, was im folgenden beispielhaft für den Datenbankbenutzer "root" geschehen soll, dem das neue Passwort "unforgettablePassword" zugewiesen wird:

```
sudo mysqladmin --user=root password "unforgettablePassword"
```

4.1.3.3. Tabellen für Messwerte anlegen

In Anlehnung an die vorangehende Projektarbeit einer andere Projektgruppe, welche bereits einen Gaszähler angebunden hat, wird die Tabelle für die Messwerte des Stromzählers ähnlich aufgebaut. Um die Datenbank "Stromzaehler" und die darin enthaltene Tabelle "Messwert" anzulegen müssen folgende drei Befehle ausgeführt werden:

```
create Database Stromzaehler;
use Stromzaehler;
create table Messwert
(MwNr int auto_increment primary key, wert float, zeit timestamp);
```

4.1.4. MQTT

um MQTT zu nutzen wird das Paket *mosquitto* installiert.

```
sudo apt install mosquitto
```

Dies installiert die MQTT-Broker Funktionalität. Zum Testen kann es aber sinnvoll sein, auch Topics zu abonnieren oder Nachrichten in Topics veröffentlichen zu können, was Teil des Client-Packages ist, welches mit folgendem Befehl installiert werden kann:

```
sudo apt install mosquitto-clients
```

Die damit installierten Befehle zum Testen lauten:

```
mosquitto_sub // Zum Abonnieren von Topics
mosquitto_pub // Zum Veröffentlichen von Nachrichten in Topics
```

Mosquitto wird automatisch beim Hochfahren gestartet, diesbezüglich ist hier nichts weiter zu tun. Gegebenenfalls kann noch der MQTT-Port geändert werden, dies ist aber in den meisten Fällen nicht notwendig und es wird der Standard-Port 1883 verwendet.

Zu MQTT sei schlussendlich noch angemerkt, dass das Programm mosquitto auch für Windows verfügbar ist es sich anbietet, zum Testen ein weiteres Gerät wie z.B. einen Laptop zu verwenden.

4.1.5. Node-Red

Sobald die Datenbank und MQTT eingerichtet sind kann als nächstes NodeRed installiert werden. Die Installation gestaltet sich sehr einfach, da die Entwickler auf ihrer Webseite speziell für den Raspberry Pi einen Befehl vorbereitet haben, womit alle notwendigen Packages, Abhängigkeiten und Basiskonfigurationen für den Dienst von NodeRed erledigt werden: [Opeb]

```
bash <(curl -sL https://raw.githubusercontent.com/node-red/linux-installers/
master/deb/update-nodejs-and-nodered)
```

Nach der Installation sollte man noch in der Systemkonfiguration des Raspberry Pi hinterlegen, dass der Dienst beim Hochfahren automatisch gestartet wird.

```
sudo systemctl enable nodered.service
```

Der Zugriff auf NodeRed erfolgt über ein Webportal, welches im Browser mit der IP des Hosts (Raspberry Pi) und dem Standard-Port 1890 aufgerufen werden kann. Das Testsystem hatte die IP *192.168.0.12*, entsprechend musste im Browser eingegeben werden:

```
192.168.0.12:1890
```

Der nächste Schritt wäre nun die Erstellung von neuen Flows. Es können allerdings auch bereits vorhandene Flows importiert werden, so wie es bei diesem Projekt der Fall sein wird. Einen existierenden Flow kann man ganz einfach als JSON exportieren und danach auf einem anderen Gerät wieder importieren [Opea].
Der erstellte Flow für diese Projektarbeit (vgl. 3.6.6) muss für die Inbetriebnahme auf dem bereits existierenden Raspberry Pi allerdings noch an einigen Stellen angepasst werden.

Um die IP des MQTT-Brokers zu ändern, muss auf einen MQTT-node geklickt werden (z.B. auf "smartmeter"), dann auf den Stift neben "Server" und dann die IP (und falls abweichend vom Standard, auch der Port) bearbeitet werden. Bei bereits existierenden MQTT-Brokern kann im Dropdown-Menü auch einer der bereits vorhandenen Einträge ausgewählt werden.

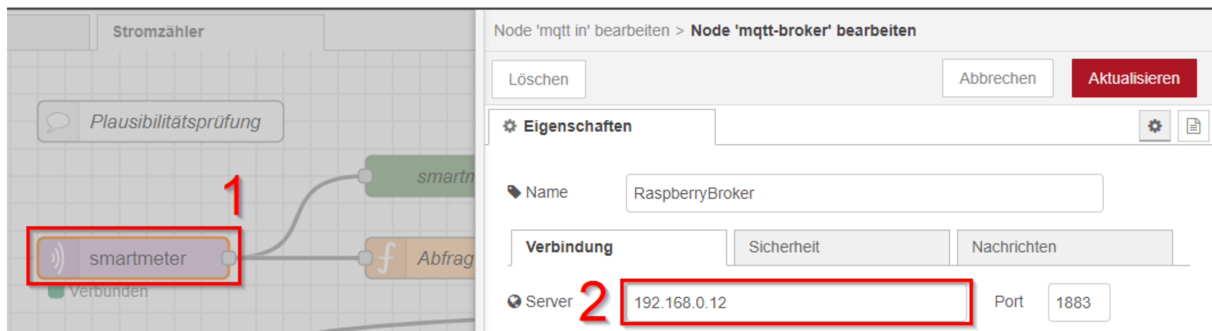


Abbildung 4.1.: Anpassung der IP des MQTT-Brokers in NodeRed

Außerdem muss die korrekte Datenbank angegeben werden. Dafür muss einer der MariaDB-nodes angeklickt werden, dann neben "Database" auf den Stift und dann in der Eingabemaske die korrekten Werte angeben. Wenn dieser Anleitung gefolgt wurde lautet der Datenbankname "Stromzaehler". Der Host kann auf "localhost" bleiben, sofern NodeRed und Datenbank auf dem gleichen Gerät laufen. Als Benutzer muss ein gültiger Datenbankbenutzer angegeben werden. Falls das Passwort nicht mehr bekannt ist kann dieses zurückgesetzt werden, wie unter 4.1.3.2 beschrieben.

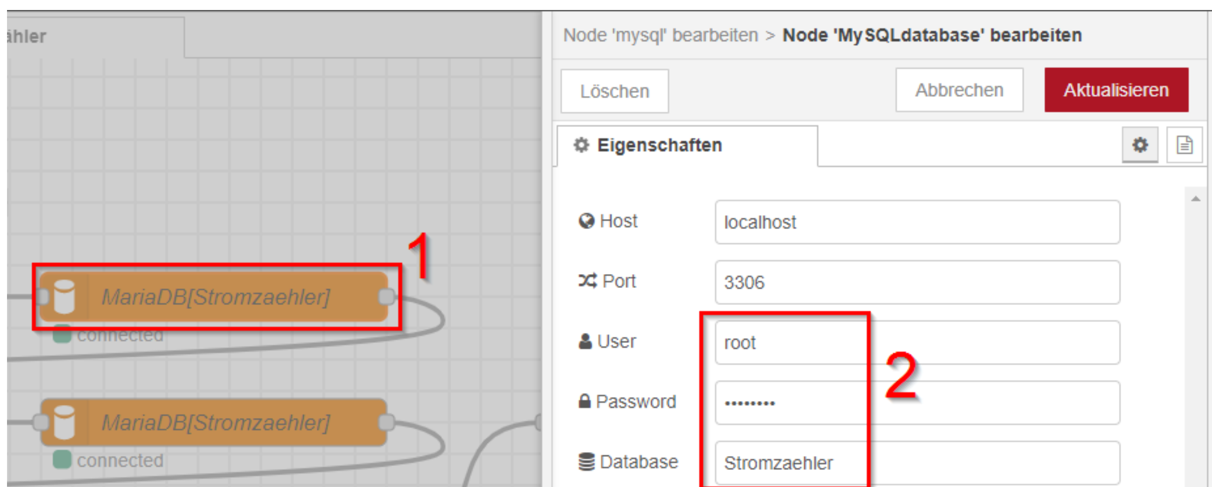


Abbildung 4.2.: Anpassung der Datenbank in NodeRed

4.2. ESP8266

Für die korrekte Verwendung des Empfängers muss lediglich die IP des MQTT-Brokers im Programm hinterlegt sein, alle weiteren Einstellungen können extern getätigt werden. Damit die Messwerte an die Datenbank gesendet werden können, benötigt der ESP eine gültige WLAN-Konfiguration. Um nicht auf den nächsten Weckruf durch die RTC warten zu müssen kann der Schalter auf der Unterseite des Gehäuses verwendet werden um die Spannungsversorgung des ESP einzuschalten und auf das Webportal der Wifi-Managers zugreifen zu können (vgl. 3.6.3).

Ursprünglich war es geplant einen dedizierten Button zum Zurücksetzen der WLAN-Konfiguration zu verwenden, welcher auch verbaut und am ESP angebunden ist. Ein neues WLAN-Netzwerk bedeutet jedoch automatisch dass sich auch die IP des Raspberry Pi geändert hat, wodurch diese neue IP im Programm hinterlegt und der ESP geflasht werden muss. Nach dem Flashen ist keine WLAN-Konfiguration hinterlegt und das neue WLAN-Netzwerk kann wie gewohnt über das Webportal eingestellt werden. Die Funktionalität des Buttons kann nachgerüstet oder der Button für andere Zwecke verwendet werden.

Abbildung 4.3.: Schalter und Taster auf der Unterseite des Gehäuses

4.3. Teststrecke

Zum Testen wird ein Computer mit installierter Arduino IDE oder einem Programm für serielle Kommunikation (z.B. Putty), ein Arduino Uno sowie der ESP und die Lochrasterplatine benötigt, auf welche die beiden Microcontroller nach erfolgter Programmierung aufgesteckt werden.

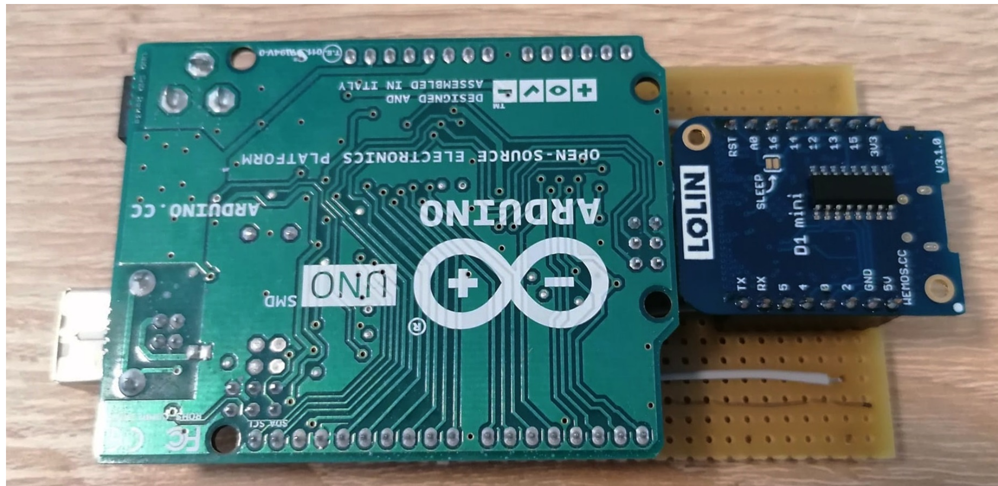


Abbildung 4.4.: Shield mit aufgestecktem Arduino und ESP

Als nächstes müssen die beiden Geräte mit dem Computer verbunden werden. Für die Eingabe am Arduino bietet es sich an, den seriellen Monitor der Arduino IDE zu verwenden, da dort auch mehrere Zeichen gesendet werden können, was beispielsweise bei Putty nicht funktioniert hat. Für die Anzeige der empfangenen Daten am ESP reicht Putty jedoch völlig aus.

Nun kann der zu sendende Wert eingegeben werden.

```
14:58:55.063 -> ¡Bitte geben Sie einen ganzzahligen Wert ein [kWh]!
15:02:55.911 -> w□□□□ʕ□□□□Yʕk = 1234
15:02:55.911 -> ¡Bei Bedarf nächsten ganzzahligen Wert eingeben [kWh]!
```

Abbildung 4.5.: Eingabefenster des Seriellen Monitors

Diesen Wert wird nun in SML-Kodierung an den ESP gesendet. Auf dem nächsten Bild sind das gesamte SML-Paket (rot)und der darin enthaltene Wert (blau) zu

erkennen (vgl. 3.6.4). Die eingegebenen 1234 kW h wurden vor dem Versenden mit 10 000 multipliziert, um die Auflösung des Stromzählers von 100 mW h korrekt abzubilden, was binär kodiert *00 00 00 00 BC 4B 20* entspricht.

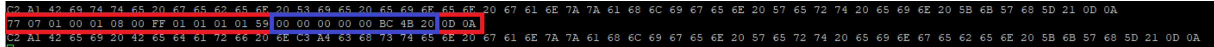


Abbildung 4.6.: Vom Teststand empfangene Daten aus Sicht des ESP

Die Daten vor bzw. nach dem SML-Teilpaket kommen von den Aufforderungen an den Bediener. Der in 4.6 zu sehende Text wird ebenfalls über die serielle Schnittstelle gesendet, da sowohl die Daten an den ESP als auch die Daten an den Computer über den gleichen TX-Pin des Arduino gesendet werden.

5. Fazit

Abschließend lässt sich festhalten, dass die Anforderungen an das Projekt erfüllt wurden. Es wurde ein voll funktionstüchtiger IR-Empfänger für einen Stromzähler entwickelt. Auch eine Anbindung an das bereits vorhandene Smarthome-System ist gelungen.

Ein großer Vorteil des entwickelten Systems ist, dass für die Umsetzung nur wenige, aber vor allem kostengünstige Bauteile nötig waren. Auch der Umgang damit ist sehr Benutzerfreundlich. Besonders sollte auch hervor gehoben werden, dass das Projekt ganz speziell an die gegebenen Umstände angepasst wurde und damit für den Benutzer personalisiert wurde.

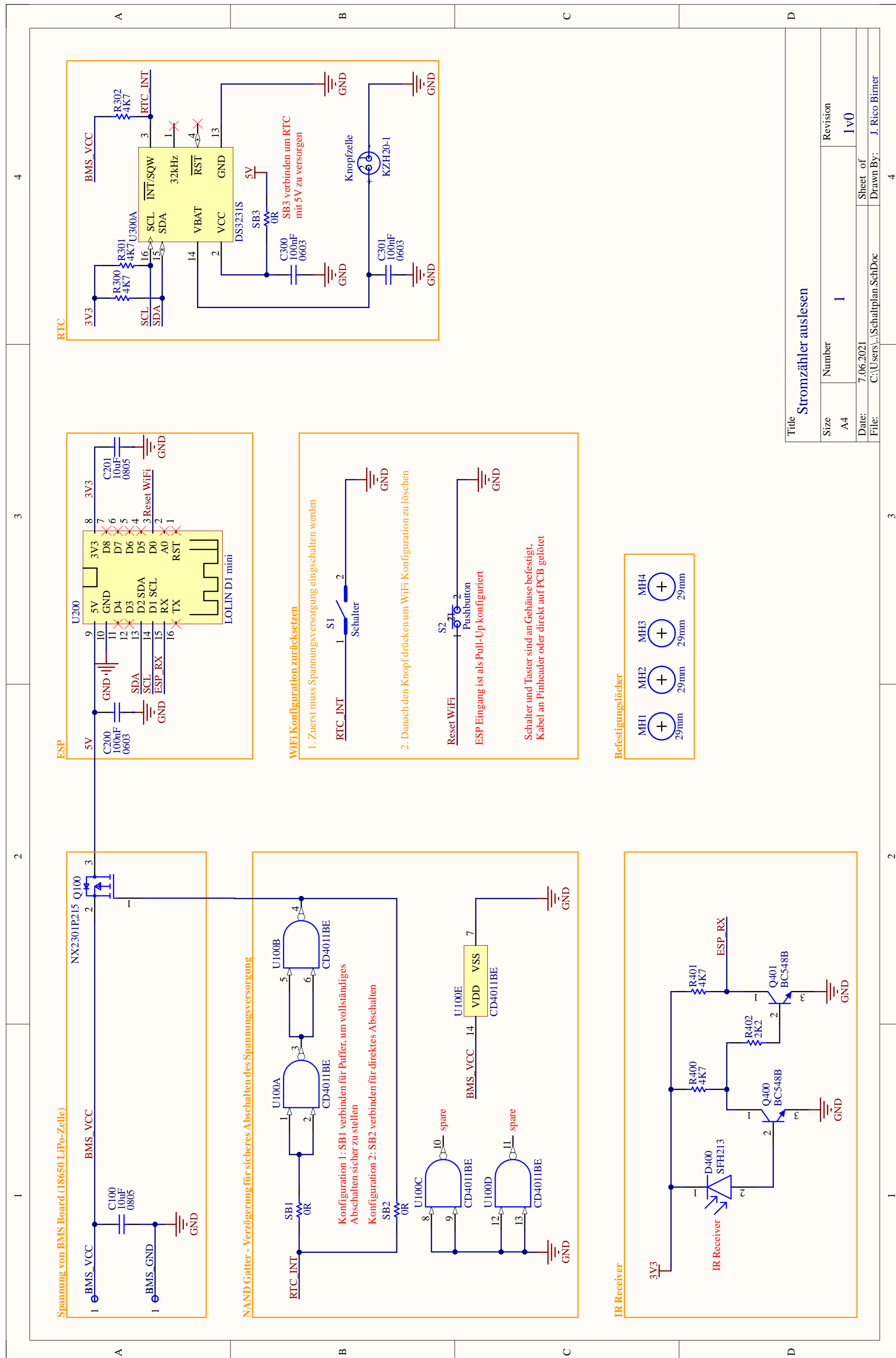
Ein kleiner Nachteil findet sich, wenn man es wieder in den Kontext des Stromsparens für einzelne Haushalte stellt. Es handelt sich nicht um ein Projekt, das für den Normalverbraucher geeignet ist. Das liegt zum einen an der Personalisierung, aber auch an den nicht unerheblichen technischen Kenntnissen, die zur Nutzung notwendig sind.

Nun ist mit diesem Projekt wohl nicht unbedingt dazu beigetragen worden, dass in Deutschland weniger Strom verbraucht wird, aber vielleicht konnte damit die Vorarbeit für ein allgemeineres Projekt geliefert werden, welches von vielen Verbrauchern nutzbar ist. Diese deutlich kostengünstigere Lösung könnte dann vielleicht wirklich einen Teil dazu beitragen, dass mehr Menschen ihren Stromverbrauch aufmerksam beobachten und deutlich senken können.

Teil II.

Anhang

5.1. Schaltplan



5.2. Software Teststand

```
/*
 *   Stand: 25.08.2021
 *   Author: Julian Rico
 *
 *   Teststand für Stromzähler-Empfänger.
 *
 *   Diese Software bietet ein Interface, um den Empfänger zu testen.
 *   Der Benutzer kann Zahlen eingeben, welche als Zählstand interpretiert werden.
 *   Der Zählstand wird dann in Form eines SML Pakets verschickt.
 */

/* Defines */
#define BAUDRATE    9600
#define BUFFERSIZE  100

/* Globale Variablen */
int BUFFER[BUFFERSIZE]; // Buffer für Einlesen serieller Daten
float kWh;              // Speichervariable für berechneten Input
int i, j;               // Laufvariablen

/* Funktionsprototypen */
void readInput();
void sendSML(unsigned long mWh);

/* Setup: einmalig */
void setup() {

  // Debug LED
  pinMode(LED_BUILTIN, OUTPUT);
  digitalWrite(LED_BUILTIN, LOW);

  // Serial Setup
  Serial.begin(BAUDRATE);
  Serial.println("¡Bitte geben Sie einen ganzzahligen Wert ein [kWh]!");
  Serial.flush();
```

```
}

/* Hauptprogramm */
void loop() {

  // Warten auf Eingabe von Bediener
  if (Serial.available() > 0)
    {
      digitalWrite(LED_BUILTIN, HIGH);
      readInput();
    }
    delay(50);
}

void readInput() {
  // Einlesen
  i = 0;
  do {
    if (i<BUFFERSIZE) {
      BUFFER[i] = Serial.read();
      i++;
    }
  } while (Serial.available());
  i--; // Letzter Eintrag in Buffer ist LF

  // Input umrechnen
  kWh = 0;
  for (j = i; j > 0; j--) {
    kWh = kWh + (BUFFER[j-1]-48) * pow(10, (i-j));
  }

  // SML senden
  // +0.5 für korrektes Auf- und Abrunden, cast rundet immer ab,
  // *10.000 für kWh->mWh
  sendSML(((unsigned long)(kWh+0.5)*10000));

  // Buffer löschen
```

```
  for (j = 0; j < i+2; j++)
    BUFFER[j] = 0;
}

void sendSML(unsigned long mWh) {
  // Hier wird ein Teil einer SML Nachricht verschickt
  // Konkret handelt es sich um die Sequence mit dem OBIS Kenncode
  // 1.8.0*255 (=Gesamtverbrauch)
  Serial.write(0x77);   /* 'M' ASCII = 77 - SML_Message.messageBody.
                        SML_GetList_Reponse.valList.valListEntry (Sequence) */
  Serial.write(0x07);   /* 07 - objName (TL[1] + octet-string[6] */
  Serial.write(0x01);   /* 01 - objName Teil A */
  Serial.write(0x00);   /* 00 - objName Teil B */
  Serial.write(0x01);   /* 01 - objName Teil C */
  Serial.write(0x08);   /* 08 - objName Teil D */
  Serial.write(0x00);   /* 00 - objName Teil E */
  Serial.write(0xFF);   /* FF - objName Teil F */
  Serial.write(0x01);   /* 01 - Status  (optional: 01 bedeutet "nicht versorgt") */
  Serial.write(0x01);   /* 01 - valTime (optional: 01 bedeutet "nicht versorgt") */
  Serial.write(0x01);   /* 01 - unit    (optional: 01 bedeutet "nicht versorgt") */
  Serial.write(0x01);   /* 01 - scaler  (optional: 01 bedeutet "nicht versorgt") */
  Serial.write(0x59);   /* 01 - value (TL[1] + Integer[8]) */
  Serial.write(0x00);   /* value[0] -- wir bekommen nur 4 Byte,
                            daher die ersten 4 nullen */
  Serial.write(0x00);   /* value[1] -- wir bekommen nur 4 Byte,
                            daher die ersten 4 nullen */
  Serial.write(0x00);   /* value[2] -- wir bekommen nur 4 Byte,
                            daher die ersten 4 nullen */
  Serial.write(0x00);   /* value[3] -- wir bekommen nur 4 Byte,
                            daher die ersten 4 nullen */

  // Letzten 4 Bytes: mWh (unsigned long) konvertieren in 4 Bytes
  unsigned long temp;
  temp = (mWh & 0xFF000000) >> 24;
  Serial.write(temp); /* value[4] */
  temp = (mWh & 0x00FF0000) >> 16;
  Serial.write(temp); /* value[5] */
```

```
  temp = (mWh & 0x0000FF00) >> 8;
  Serial.write(temp); /* value[6] */
  temp = (mWh & 0x000000FF);
  Serial.write(temp); /* value[7] */

  Serial.println();
  Serial.println("¡Bei Bedarf nächsten ganzzahligen Wert eingeben [kWh]!");
}
```

5.3. Software Empfänger

```
/*
 *   Stand: 25.08.2021
 *   Author: Julian Rico
 *
 *   Software für Stromzähler-Empfänger
 *
 *   Diese Software beinhaltet die Logik, um einen EMH Gen. K Stromzähler per
 *   Infrarot-Schnittstelle auszulesen und den Gesamtverbrauch des Zählers
 *   per MQTT an einen MQTT Broker zu schicken.
 */

/* Includes */
#include <Wire.h>
#include <WiFiManager.h>    /* https://github.com/tzapu/WiFiManager */
#include <PubSubClient.h>   /* https://github.com/knolleary/pubsubclient */

/* Defines */
#define BAUDRATE      9600
#define BUFFERSIZE    1000
#define RTC_I2C_ADDR  0x68

/* Globale Variablen */
int BUFFER[BUFFERSIZE];                       // Buffer für Einlesen serieller
                                              // Daten
int i, j, error;                              // Laufvariablen
int shutdown = 0;                             // Wird gesetzt nach erfolgreichem
```

```
                                              // Einlesen des Zählstands
WiFiClient espClient;                         // WiFi
PubSubClient client(espClient);               // MQTT Client
const char* MQTT_BROKER = "192.168.178.102";  // MQTT Broker IP -> TODO anpassen
const unsigned short MQTT_PORT = 1883;        // MQTT Broker Port -> TODO prüfen
char* MQTT_TOPIC_LIVE = "smartmeter";         // MQTT Topic für Stromzählerwerte
char* MQTT_TOPIC_TEST = "ESP";                // MQTT Topic für Eingaben von
                                              // Teststand
char* MQTT_MSG   = "ESP ONLINE.";             // Message von ESP wenn Setup
                                              // erfolgreich

/* Functionsprototypen */
// MQTT Callback: Gibt erhaltene Message aus
void callback(char* topic, byte* payload, unsigned int length);
// Liest serielle Daten ein, unterscheidet ob Daten von Teststand oder Stromzähler
// kommen
bool readTelegramm();
// Versendet Zählstand per MQTT (TOPIC_LIVE)
void read_smartmeter(int start, int komma, int ende);
// Versendet Testdaten per MQTT (TOPIC_TEST)
void read_teststation(int start);
// Alarm Register und Control Bits setzen
void RTC_Setup();
// Ausschalten
void RTC_Shutdown();

/* Setup: einmalig */
void setup() {

  // Debug LED
  pinMode(LED_BUILTIN, OUTPUT);
  digitalWrite(LED_BUILTIN, HIGH);

  // Serial Setup
  Serial.begin(BAUDRATE);
  Serial.flush();
  Serial.println("Serial init done.");
```

```
// Wifi Setup
WiFi.mode(WIFI_STA);
WiFiManager wifiManager;
bool res;
res = wifiManager.autoConnect("ESP32-STROMZÄHLER", "PASSWORD");
if(!res) {
  Serial.println("Failed to connect");
}
else {
  //if you get here you have connected to the WiFi
  Serial.println("connected...yeey :)");
}

// MQTT Setup
error = 0;
client.setServer(MQTT_BROKER, MQTT_PORT);
while (!client.connected()) {
  client.connect("ESP8266Client");
  if (client.connected()) {
    Serial.println("MQTT connected");
  }
  else {
    Serial.print("failed, rc=");
    Serial.print(client.state());
    Serial.println("try again in 5 seconds");
    delay(5000);
    error++;
    if (error == 12) { // Nach 60 Sekunden Shutdown
      RTC_Shutdown();
    }
  }
}

// Das kann dann später raus, ist nur zum testen der MQTT Verbindung per Serial
// Monitor
client.subscribe(MQTT_TOPIC_TEST);
```

```
  client.setCallback(callback);
  client.publish(MQTT_TOPIC_TEST, MQTT_MSG);

  // RTC
  RTC_Setup();
  Serial.println("RTC Setup done");
}

/* Hauptprogramm */
void loop() {
  // Benötigt für Empfangen von MQTT Messages -> kann später raus
  client.loop();

  // ESP kommst sonst seriell nicht hinterher
  delay(75);

  // Serielle Daten einlesen falls vorhanden
  if (Serial.available() > 0) {
    if (readTelegramm())
      shutdown = 1;
  }

  // Buffer für nächsten Datensatz leeren (benötigt, wenn Zählstand nicht dabei
  // war)
  if (i > 0) {
    for (int k = 0; k < i; k++)
      BUFFER[k] = 0;
    i = 0;
  }

  // Shutdown durch RTC vorbereiten
  if (shutdown)
    RTC_Shutdown();
}

/* MQTT Subscriber Callback */
void callback(char* topic, byte* payload, unsigned int length) {
```

```
  Serial.print("New message in topic: ");
  Serial.println(topic);

  Serial.print("Message: ");
  for (i = 0; i < length; i++) {
    Serial.print((char)payload[i]);
  }
  Serial.println();
  Serial.println("-----------------------");
  Serial.println();
}

bool readTelegramm() {
  // Read DO Telegram
  i = 0;
  do {
    if(i < BUFFERSIZE) {
      BUFFER[i] = Serial.read();

      /* Debug */
      if (BUFFER[i] < 0xF)
        Serial.print("0"); // Führende Null erzeugen
      Serial.print(BUFFER[i], HEX);
      Serial.print(" ");
      if (BUFFER[i] == 0x0A) { // 0A (hex) = LF (ASCII) => Neue Zeile
        Serial.println();
      }
      /* Debug Ende */
      i++;
    }
  } while (Serial.available());

  // Buffer nach Daten für Zählerstand absuchen
  for (j = 0; j < i; j++) { // Buffer nach Zeichen absuchen

    // Daten von Teststand: "¡Bitte geben Sie ... !");
    if (BUFFER[j] == 0xC2 && BUFFER[j+1]) { /* C2 A1 = '¡' */
```

```
    return false; // Shutdown für RTC nur schicken wenn Daten von Smartmeter
                  // kommen
  }

  // http://itrona.ch/stuff/F2-2_PJM_5_Beschreibung%20SML%20Datenprotokoll%20
  // V1.0_28.02.2011.pdf
  // Daten von Stromzähler: Gesamtverbrauch herausfiltern
  if (                        /* OBIS Kennung: 1-0.1.8.0*255 = 01 00 01 08 00
                                FF */
    BUFFER[j]   == 0x77 &&    /* 77 - SML_Message.messageBody.
                                SML_GetList_Reponse.valList.valListEntry
                                (Sequence) */
    BUFFER[j+1] == 0x07 &&    /* 07 - objName (TL[1] + octet-string[6] */
    BUFFER[j+2] == 0x01 &&    /* 01 - objName Teil A */
    BUFFER[j+3] == 0x00 &&    /* 00 - objName Teil B */
    BUFFER[j+4] == 0x01 &&    /* 01 - objName Teil C */
    BUFFER[j+5] == 0x08 &&    /* 08 - objName Teil D */
    BUFFER[j+6] == 0x00 &&    /* 00 - objName Teil E */
    BUFFER[j+7] == 0xFF)      /* FF - objName Teil F */
                              /* xx - status         */
                              /* xx - valTime        */
                              /* xx - unit           */
                              /* xx - scaler         */
  {
    j = j+8;

    // status, valTime, unit und scaler überspringen
    while (BUFFER[j] != 0x59) { j++; }  /* 59 - value (TL[1] + 64 Bit Integer */

    // Zahl aus SML in Variable überführen

    // 64 Bit: 2 x 32 Bit Variablen -> mWh
    long long mWh = ((long long)BUFFER[j+1]) << 56 |
                    ((long long)BUFFER[j+2]) << 48 |
                    ((long long)BUFFER[j+3]) << 40 |
                    ((long long)BUFFER[j+4]) << 32 |
                    ((long long)BUFFER[j+5]) << 24 |
```

```
                      ((long long)BUFFER[j+6]) << 16 |
                      ((long long)BUFFER[j+7]) <<  8 |
                      ((long long)BUFFER[j+8]);

      // Debug
      Serial.println();
      Serial.print("mWh:"); Serial.println(mWh);

      mWh = mWh / 10000;    // mWh -> kWh
      int kWh = (int) mWh;

      // Debug
      Serial.print("Gesamtverbrauch: ");Serial.println(kWh);

      // Zählstand an MQTT Broker schicken
      send_MQTT(kWh);

      // Wenn Gesamtverbrauch in SML gefunden: Signal für Shutdown geben
      return true;

    } // Ende if (OBIS Kennung)
    j++;
  } // Ende for-Schleife

  // Hier return falls keine gültige SML Nachricht erkannt wurde => ESP nicht
  // ausschalten, sondern auf nächste warten
  return false;
}

void send_MQTT(int kWh) {
  String temp = String(kWh);
  client.publish(MQTT_TOPIC_LIVE, temp.c_str());
}

void RTC_Setup() {
  /* Set Alarm 1 on seconds = 0, minutes = 0, hours = 0 */
  Wire.beginTransmission(RTC_I2C_ADDR);
```

```
  Wire.write(0x07);                        // Address of A1M1
  Wire.write(0x00);                        // A1M1 = 0, alarm value seconds = 0
  Wire.write(0x80);                        // A1M2 = 0, alarm value minutes = 0
  Wire.write(0x80);                        // A1M3 = 0, alarm value hours   = 0
  Wire.write(0x80);                        // A1M4 = 1, rest X
  Wire.endTransmission();

  /* Set alarm 2 on minutes = 0, hours = 12 */
  Wire.beginTransmission(RTC_I2C_ADDR);
  Wire.write(0x0B);                        // Address of A2M2
  Wire.write(0x00);                        // A2M2 = 0, alarm value minutes = 0
  Wire.write(0x12);                        // A2M3 = 0, alarm value hours = 12
  Wire.endTransmission();

  /* Set A1E & A2E control bits */
  Wire.beginTransmission(RTC_I2C_ADDR);
  Wire.write(0x0e);                        // Control byte
  Wire.write(0x1C | 3);                    // Default | A1E | A2E
  Wire.endTransmission();
}

void RTC_Shutdown() {
  Wire.beginTransmission(RTC_I2C_ADDR);
  Wire.write(0x0F);                        // Address of control/status register
  Wire.endTransmission();

  Wire.requestFrom(RTC_I2C_ADDR, 1);       // Read the current value of the register
  unsigned char reg_val = Wire.read();

  Wire.beginTransmission(RTC_I2C_ADDR);
  Wire.write(0x0F);                        // Address of control/status register
  Wire.write(reg_val & ~0x03);             // Write the old value with A1F&A2F flags
                                           // cleared
  Wire.endTransmission();                  // -> this resets the latching ~INT Pin
}
```

Literaturverzeichnis

[Bal21] BALENA: *Flash. Flawless.* https://www.balena.io/etcher/. Version: 2021

[BDE13] BDEW: *EDI@Energy OBIS-Kennzahlen-System*. https://www.bundesnetzagentur.de/DE/Beschlusskammern/BK06/BK6_81_GPKE_GeLi/Mitteilung_Nr_37/Anlagen/OBIS-Kennzahlensystem%202.2a.pdf?__blob=publicationFile&v=2. Version: 2013

[Bun13] BUNDESAMT FÜR SICHERHEIT IN DER INFORMATIONSTECHNIK: *Technische Richtlinie BSI TR-03109-1*. https://www.bsi.bund.de/SharedDocs/Downloads/DE/BSI/Publikationen/TechnischeRichtlinien/TR03109/TR-03109-1_Anlage_Feinspezifikation_Drahtgebundene_LMN-Schnittstelle_Teilb.pdf?__blob=publicationFile. Version: 2013

[Con12] CONTINENTAL DEVICE INDIA LIMITED: *NPN SILICON PLANAR EPITAXIAL TRANSISTORS*. https://cdn-reichelt.de/documents/datenblatt/A100/BC546_48-CDIL.pdf. Version: 2012

[eHZ] EMH METERING GMBH & CO. KG (Hrsg.): *eHZ Generation K*. https://emh-metering.com/wp-content/uploads/2021/08/eHZ-K-DAB-D-1.10.pdf

[EMH20] EMH METERING GMBH & CO. KG: *eHZ Generation K Elektronischer Haushaltszähler*. https://emh-metering.com/wp-content/uploads/2020/08/eHZ-K-BIA-D-1-20.pdf. Version: 2020

[Esp19] ESPRESSIF IOT TEAM: *ESP8266 Low-Power Solutions*. https://www.espressif.com/sites/default/files/documentation/9b-esp8266-low_power_solutions__en.pdf. Version: 2019

[Git20] GITHUB: *pubsubclient*. https://github.com/knolleary/pubsubclient. Version: 2020

[Git21] GITHUB: *WiFi Manager*. https://github.com/tzapu/WiFiManager. Version: 2021

[Gra21] GRABCAD: *Wemos 18650 battery shield V3*. https://cad.grabcad.com/library/wemos-18650-battery-shield-v3-1. Version: 2021

[hau] EMH METERING GMBH & CO. KG (Hrsg.): *haushaltszaehler*. https://emh-metering.com/produkte/haushaltszaehler-smart-meter/ehz-k/

[Lan18] LANDIS+GYR: *E220 Benutzerhandbuch*. https://www.landisgyr.de/webfoo/wp-content/uploads/2018/08/D000063497-E220-AMxD-Benutzerhandbuch-de-f.pdf. Version: 2018

[max15] MAXIM INTEGRATED: *Extremely Accurate I2C-Integrated RTC/TCXO/-Crystal*. https://datasheets.maximintegrated.com/en/ds/DS3231.pdf. Version: 2015

[MQT20] *MQTT*. https://mqtt.org/. Version: 2020

[Nex10] NEXPERIA B.V.: *NX2301P 20 V, 2 A P-channel Trench MOSFET*. https://assets.nexperia.com/documents/data-sheet/NX2301P.pdf. Version: 2010

[nod] OPENJS FOUNDATION (Hrsg.): *NodeRED*. https://nodered.org/

[Opea] OPENJS FOUNDATION: *Importing and Exporting Flows*. https://nodered.org/docs/user-guide/editor/workspace/import-export

[Opeb] OPENJS FOUNDATION: *Running on Raspberry Pi*. https://nodered.org/docs/getting-started/raspberrypi

[Osr14] OSRAM: *Silizium-PIN-Fotodiode*. https://cdn-reichelt.de/documents/datenblatt/A501/SFH213FA_ENG_TDS.pdf. Version: 2014

[Pan] PANASONIC: *Specifications for NCR18650PF*. https://b2b-api.panasonic.eu/file_stream/pids/fileversion/3447

[PuT] PUTTY: *Download PuTTY*. https://www.putty.org/

[RASa] RASPBERRY PI FOUNDATION: *Raspberry Pi OS*. https://www.raspberrypi.org/software/

[Rasb] RASPBERRY PI TUTORIALS: *Raspberry Pi SSH Zugriff einrichten via Putty (Windows).* https://tutorials-raspberrypi.de/raspberry-pi-ssh-windows-zugriff-putty/

[Str] HEIDJANN GMBH (Hrsg.): *StromAuskunft.* https://www.stromauskunft.de/stromverbrauch/stromverbrauch-haushalte/#backtotop

[Str09] STRECK ; EASYMETER (Hrsg.): *Protokoll Datentransfer zu Erweiterungsmodulen.* https://www.mikrocontroller.net/attachment/89888/Q3Dx_D0_Spezifikation_v11.pdf. Version: 2009

[WEM21] WEMOS: *LOLIN D1 mini.* https://www.wemos.cc/en/latest/d1/d1_mini.html. Version: 2021

[Wik21] WIKIPEDIA ; WIKIPEDIA (Hrsg.): *Smart Home.* https://de.wikipedia.org/wiki/Smart_Home. Version: 2021

Teil III.

Danksagung

6. Danksagung

Für die Betreuung dieser Projektarbeit und die herzliche Unterstützung bei der Durchführung möchten wir uns bei Prof. Dr. Klehn und Herrn Wölfling bedanken.