

# Dokumentation - Luftqualität in Innenräumen - Gruppe 1

Friedrich Just 1326699  
Stipe Knez 1269206  
Lucas Merkert 1326709  
Achim Glaesmann 1309221  
Max-Rene Konieczka 1211092  
Can Cihan Nazlier 1179244

14. Februar 2022

# Inhaltsverzeichnis

<b>1</b>	<b>Projektthema</b>	<b>2</b>
<b>2</b>	<b>Projektorganisation</b>	<b>3</b>
2.1	Kommunikation . . . . .	3
2.2	Tools und Frameworks . . . . .	3
2.3	Prozessverlauf . . . . .	4
<b>3</b>	<b>Aufgabenaufteilung</b>	<b>5</b>
3.1	Friedrich Just . . . . .	5
3.2	Stipe Knez . . . . .	5
3.3	Lucas Merkert . . . . .	5
3.4	Achim Glaesmann . . . . .	5
3.5	Max-Rene Konieczka . . . . .	5
3.6	Can Cihan Nazlier . . . . .	6
<b>4</b>	<b>Luftqualität</b>	<b>7</b>
4.1	Luftfeuchte und Temperatur . . . . .	7
4.2	CO2 . . . . .	7
4.3	VOC . . . . .	8
<b>5</b>	<b>Lösungsansatz</b>	<b>8</b>
5.1	SHT21 . . . . .	8
5.2	Probleme bei der Programmierung des SHT21 . . . . .	11
5.3	Funktionsweise des SHT21 . . . . .	11
5.4	SCD41 . . . . .	12
5.5	Probleme bei der Programmierung des SCD41 . . . . .	15
5.6	Funktionsweise des SCD41 . . . . .	15
5.7	CCS811 . . . . .	17
5.7.1	Probleme bei der Programmierung des CCS811 . . . . .	18
5.7.2	Funktionsweise des CCS811 . . . . .	20
5.8	Messung der Eingangsspannung . . . . .	21
5.9	Probleme mit der Batterie . . . . .	21
5.10	Datenübertragung . . . . .	21
5.11	Netzwerkaufbau . . . . .	22
5.11.1	Aufbau . . . . .	22
5.11.2	Probleme beim Zusammenfügen der Sensoren und des Netzwerkes . . . . .	23
5.12	Serverimplementierung . . . . .	26
5.13	Zeichentool . . . . .	26
5.13.1	Aufbau und Funktionsweise . . . . .	26
5.13.2	Probleme beim Zeichentool . . . . .	28
5.14	Dashboard . . . . .	29
5.14.1	Funktionsweise . . . . .	29
5.14.2	Aufgetretene Probleme . . . . .	29

5.15	COM-Port & Mockdaten . . . . .	30
5.15.1	Mockdaten . . . . .	30
5.15.2	Kommunikation über den COM-Port . . . . .	31
5.15.3	Aufgetretene Probleme . . . . .	32
5.16	Installationsdatei . . . . .	34
<b>6</b>	<b>Auswertung des Testlaufs</b>	<b>34</b>
<b>7</b>	<b>Ausblick und Erweiterungsmöglichkeiten</b>	<b>37</b>
7.1	Optimierung für Batterien . . . . .	37
7.2	Genauere Bestimmung der Werte des CCS811 . . . . .	37
7.3	Bugfixes in der Applikation . . . . .	37
7.4	Dynamische Gestaltung . . . . .	37
7.5	Migration auf andere Geräte oder Plattformen . . . . .	37
7.6	Genauere Berechnung der Corona-Ampel . . . . .	38
7.7	Dark Mode . . . . .	38

# 1 Projektthema

Die durch Covid-19 verursachte Pandemie prägte die letzten 2 Jahre der gesamten Welt. Insgesamt forderte die Krankheit etwa 5,18 Millionen Menschenleben. Dennoch ist das Thema aktueller denn je. Europaweit steigen die Infektionszahlen auf nie dagewesene Werte, während die im Sommer verabreichten Impfungen langsam an Effektivität verlieren. Ein Hauptrisiko zur Infektion besteht dabei in Innenräumen. Unser Auftrag besteht nun darin, ein System zu entwickeln welches in der Lage ist, das Infektionsrisiko einzelner Räume eines Gebäudes einzuschätzen und so einen Richtwert für den Anwender darstellt, wie er sein Verhalten diesem Wert anpassen kann. Realisiert werden soll dieses System über die Verwendung verschiedener Sensoren zur Erfassung mit dem Infektionsrisiko direkt verknüpfter physikalischer Größen. Die zur Ansteckung, vermutlich, wichtigen Aerosole, können dabei nur bedingt durch Masken zurückgehalten werden, es macht also Sinn abzuschätzen inwieweit die Luft eines Raumes durch Aerosole belastet ist. Da die direkte Messung von Aerosolen zeitaufwendig und schwer umsetzbar ist, konzentrieren wir uns hierbei auf Werte, die einen direkten Rückschluss auf die Ausatemmenge der Personen im Raum ermöglichen, mit der Annahme, dass die Aerosolkonzentration dabei direkt abhängig zur Ausatemmenge ist.

Um dies zu erreichen, bedienen wir uns folgender Sensoren: Dem CCS811 von Adafruit, dem SCD41 und dem SHT21 von Sensirion. Der CCS811 ist ein energieeffizienter digitaler Gassensor, welcher über die Verwendung eines Metalloxid-sensors ein breites Spektrum an flüchtigen organischen Verbindungen messen kann. Diese gibt der Sensor als CO<sub>2</sub>-Äquivalente an. Der CO<sub>2</sub>-Äquivalenzbereich geht hierbei von 400ppm bis 8192ppm. Eine Kompensation für Abweichungen durch Temperatur und Luftfeuchtigkeit ist dabei über einen externen Feuchtigkeits- und Temperatursensor möglich. Dies ermöglicht in Verbindung mit einem Mikrokontroller die Überwachung der Luftqualität. Der Sensor enthält dabei eine I<sup>2</sup>C-Schnittstelle sowie einen Analog-Digital-Converter (ADC). Der SHT21 ermöglicht die Messung der relativen Luftfeuchtigkeit so wie der Temperatur mit einer Genauigkeit von  $\pm 2\%$  relativer Luftfeuchtigkeit so wie  $\pm 0.3^\circ$  Celsius bezüglich der Temperatur.

Beim SCD41 Sensor handelt es sich um einen miniaturisierten CO<sub>2</sub>-Sensor basierend auf dem photoakustischem Sensorprinzip. Dabei wird über einen integrierten SHT41 Feuchtigkeits- und Temperatursensor eine chipinterne Signalkompensierung ermöglicht. Die Genauigkeit des SCD41 befindet dabei im Bereich von 400 ppm – 5'000 ppm  $\pm$  (40 ppm + 5% des abgelesenen Wertes). Beide Sensoren von Sensirion können über die I<sup>2</sup>C-Schnittstelle angesprochen werden. Genauere Informationen zu den Sensoren sind im Unterkapitel Sensoren vermerkt.

Die so ermittelten Daten sollen anschließend in eine graphische Darstellung in Form einer Ampelindikation überführt werden, um dem Anwender eine direkte Bewertung der Gefahrenlage zu ermöglichen. Die Kommunikation zwischen Mikrokontroller und Sensoren soll dabei via I<sup>2</sup>C-Schnittstelle erfolgen, die Kommunikation zwischen den einzelnen Mikrokontrollern auf Basis des ZigBee-

Protokolls. Um dies zu ermöglichen wurde entschieden, einen Mikrokontroller auf Basis des Atmega256rfr2-Chips zu verwenden, welcher die Verwendung beider Übertragungsprotokolle ermöglicht, so wie eine serielle Kommunikation über eine UART-Bridge. Die ermittelten Daten werden wie erwähnt seriell an eine von uns entwickelte Applikation auf einem leistungsfähigeren Endgerät übermittelt, welches die Daten auswertet und in ein graphisches Modell des zu beschreibenden Zimmers einbettet. Das Frontend sowie das Backend der Applikation soll dabei mittels TypeScript, JavaScript und dem Framework Vue realisiert werden. Die Applikation soll es dem Anwender grundsätzlich ermöglichen eigene Grundrisse zu zeichnen, Räume zu definieren und Sensoren in diesen zu platzieren. Die ermittelten Werte der Sensoren sollen dabei einmal innerhalb von Graphen zeitlich dargestellt werden sowie die über einen eigens zu entwickelnden Algorithmus errechnete Gefahrenlage mittels eines Ampelsystems einmal innerhalb der Anwendung als auch über eine am jeweiligen Funkmodul angebrachte LED dargestellt werden.

## **2 Projektorganisation**

### **2.1 Kommunikation**

Um unsere nächsten Schritte abzustimmen und Meetings zu planen, wurden zu Beginn des Projekts eine WhatsApp Gruppe sowie ein Discord Server eingerichtet, damit jeder Gruppenteilnehmer auf dem neuesten Stand sein konnte, was den Fortschritt zu den Sensoren oder der Applikation angeht. Zur Überprüfung, wer welche Aufgabe übernommen hat, ist entschieden worden den Aufgaben-Verwaltungs-Onlinedienst Trello zu verwenden. Hierdurch können innerhalb den Teams “Karten” zugewiesen werden, die dann Schritt für Schritt bearbeitet werden. Wenn eine Aufgabe erledigt wurde, musste dies von einem anderen Gruppenmitglied überprüft und bestätigt werden.

Zudem gab es innerhalb der Gruppe mindestens einmal pro Woche ein Meeting, um die weitere Entwicklung oder aufgetretene Probleme zu besprechen.

### **2.2 Tools und Frameworks**

Als IDE wurde schlussendlich nur Webstorm benutzt, da die gesamte Applikation in Javascript und Typescript implementiert wurde. Für die Versionsverwaltung wurde ein GitHub-Repository erstellt. Es wurde ursprünglich geplant influxDB zu verwenden um die Sensordaten abspeichern zu können, doch dies hätte bei der Erstellung einer Installationsdatei für den Anwender zu Komplikationen führen können, daher wurde entschieden auf SQLite umzusteigen. Um das Senden von Mockdaten zu simulieren, wurde für kurze Zeit das Programm Virtual Serial Port verwendet um virtuelle COM-Ports zu erstellen um die Mockdaten übertragen zu können.

Die Applikation verwendet auf der Seite des Backends das Express.js Framework von Node.js, welches viele Funktionen zur Erstellung von Webanwendun-

gen anbietet. Auf der Frontend-Ebene wird das Vue.js Framework benutzt, mit welchem sich einzelne Komponenten, wie z.B. Leinwände, Buttons oder Module, erstellen lassen.

## 2.3 Prozessverlauf

Derzeitig läuft der Prozess zwischen Sensoren und Applikationen folgendermaßen ab:

Zunächst messen die Sensoren jeweilige Werte in der Umgebung des Raumes und übertragen diese weiter an die ZigBee-Module. Das ZigBee-Module schicken diese an ein Koordinator Modul und leitet die Werte über eine UART-Bridge weiter an eine COM-Port Schnittstelle. Ab hier übernimmt die Applikation. Durch die SerialPort.js Library wird dafür gesorgt, dass die seriell übertragenen Daten an das Backend weitergeleitet werden. Das Express-Backend kommuniziert mit der SQLite-Datenbank über Queries, welche mit Knex.js geschrieben sind. Die nötigen Daten werden per Websockets an das Frontend geschickt. Zum einen werden die Daten anhand von Graphen im Dashboard visuell dargestellt, zum anderen bekommt auch das Zeichentool die Sensorwerte übermittelt, da hier die Ampelindikation implementiert ist.

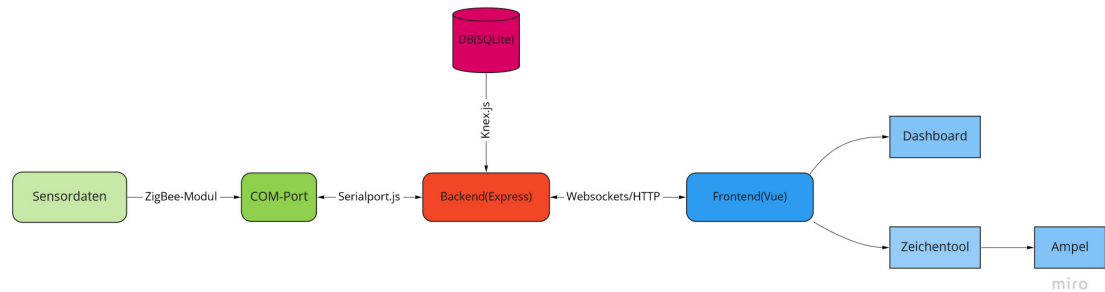


Abbildung 1: Architekturdiagramm des Vorgangs zwischen Sensoren und Applikation

### **3 Aufgabenaufteilung**

Zu Beginn des Projektes wurde entschieden die Aufgaben in zwei Teile aufzuteilen, einerseits in einen Sensorteil und andererseits in einen Applikationsteil. Es wurde festgelegt, dass Herr Merkert, Herr Just und Herr Glaesmann für die Einrichtung und Programmierung der Sensoren zuständig sind, während Herr Knez, Herr Nazlier und Herr Konieczka für die Einrichtung eines Servers und die Entwicklung der Applikation zuständig sind.

#### **3.1 Friedrich Just**

Herr Just war für die Implementierung des Funknetzwerks, Erstellung des zugehörigen Zustandsdiagramm, das Messen der Batteriespannung und für den Testlauf verantwortlich. Zudem hat er sich um verschiedene Fehler bei der Sensordaten Ausgabe gekümmert.

#### **3.2 Stipe Knez**

Herr Knez hat sich zu Beginn und im Verlauf des Projektes um die korrekte Erstellung und Verwaltung der Mockdaten gekümmert. Darüber hinaus war er bei der Einrichtung des Backends samt der Datenbank beteiligt. Des Weiteren hat er sich um die Verbindung zwischen ZigBee-Modul und Backend über die COM-Port Schnittstelle und somit den Datenempfang gekümmert. Zuletzt hat er bei der Entwicklung des Dashboards mitgewirkt.

#### **3.3 Lucas Merkert**

Aufgabe von Herr Merkert war die Programmierung des CCS811 Sensors, das erstellen des zugehörigen Zustandsautomaten, verschiedene administrative Aufgaben, wie das organisieren von Meetings und Abgabe der Dateien und Recherche zum Thema Luftqualität in Innenräumen.

#### **3.4 Achim Glaesmann**

Herr Glaesmann war damit beauftragt, auf dem Mikrokontroller die Kommunikation mit dem SHT21 so wie den SCD41 Sensor von Sensirion so wie deren Messfunktionen zu implementieren, so wie zusammen mit dem Mikrokontroller-Team den Zustandsautomaten zu entwerfen, das Format der Kommunikation mit dem Backend, die Kalibrierung der Sensoren und andere allgemein anfallende Aufgaben zu erledigen.

#### **3.5 Max-Rene Konieczka**

Herr Konieczka war anfangs mit der Einarbeitung in Websockets beschäftigt um diese korrekt für die Übertragung der Daten zu verwenden. Zudem hat er Beihilfe bei der Integrierung von SerialPort.js geleistet, anschließend hat auch

er sich wie Herr Knez und Herr Nazlier um die richtige Implementierung des Dashboards gekümmert, darunter auch Teile der Ampelindikation.

### **3.6 Can Cihan Nazlier**

Herr Nazlier hat die Entwicklung in der Applikationsgruppe koordiniert und Herr Knez und Herr Konieczka dementsprechend Aufgaben zugewiesen. Er hat sich zum einen um die Konfiguration der Programmierumgebung gekümmert. Dazu gehören die Erstellung eines GitHub-Repository und Trello-Boards und die Integrierung von Vue, einem JavaScript-Framework. Darüber hinaus hat er sich um die Integrierung einer Datenbank, die Implementierung des Zeichentools, anfangs in Form eines Prototypen, sowie einige Teile des Dashboards und der Ampelindikation gekümmert.



## 4 Luftqualität

Die Qualität der Luft setzt sich aus verschiedenen Faktoren zusammen. Für diese Projekt sind insbesondere die Faktoren Temperatur, Luftfeuchte, CO<sub>2</sub> Gehalt und VOC (volatile organische Komponenten) wichtig, da wir mit den gegebenen Sensoren nur diese messen können.

### 4.1 Luftfeuchte und Temperatur

Luftfeuchte im allgemeinen bezeichnet die Menge an Wasserdampf die von der Umgebungsluft aufgenommen werden kann, bevor dieser kondensiert. Dabei unterscheidet man zwischen der absoluten, der maximalen und der relativen Luftfeuchte. Maximale Luftfeuchte bezeichnet hierbei den maximalen Wassergehalt in der Luft bevor das Wasser anfängt zu kondensieren. Die absolute Luftfeuchte gibt an wie viel Wasser sich derzeit in der Luft befindet und die relative Luftfeuchte gibt die Relation der beiden Größen in Prozent an. Die Luftfeuchte ist zudem stark von der Temperatur abhängig, dabei gilt: je höher die Temperatur der Luft desto größer die maximale Luftfeuchte und umgekehrt. Laut Inverter[9] ist für Wohn- und Arbeitsräume bei einer Temperatur von 20°C eine Luftqualität von 40 - 60% optimal. Bei zu geringer Luftfeuchte (< 40%) kann es u.A. zu Augen-, Haut-, und Schleimhautreizungen kommen. Bei Werten unter 23% setzt Feuchtigkeitsverlust ein, der nicht über das Atmen kompensiert werden kann und es kann zu Stromschlägen bei Berührung mit Metall kommen. Zudem kann es zu Schäden an Bausubstanzen wie Holz u.ä. kommen. Bei zu hoher Luftfeuchte (> 60%) bildet sich schnell Schimmel welcher nicht nur die Bausubstanz schädigen kann sondern auch gesundheitliche Schäden nach sich ziehen kann. Überschreitet die Feuchtigkeit 80% kann es zudem zur Vermehrung von Pilzen, Milben und anderen Parasiten kommen. Um dem Ganzen vorzubeugen, hilft es regelmäßig zu lüften.

### 4.2 CO<sub>2</sub>

Laut verschiedenen Quellen, u.A. dem Umweltbundesamt [12][14], gilt CO<sub>2</sub> als Leitparameter um zu bestimmen wie viele Menschen sich in einem Raum aufhalten, vorausgesetzt es befinden sich keine weiteren CO<sub>2</sub>-Quellen in diesem Raum und kann somit auch als Indikator für die Ansteckungsgefahr durch Corona (SARS-COVID-19) verwendet werden. CO<sub>2</sub> ist ein geruch- und farbloses Gas und ist mit einer Konzentration von 400ppm natürlicher Bestandteil der Umgebungsluft und wird vom Menschen hauptsächlich beim Ausatmen ausgestoßen, wie auch die Corona Viren. Das Umweltbundesamt [11] hat hierfür eine Corona-Ampel eingeführt, welche hauptsächlich für die Anwendung in Klassenräumen vorgesehen ist. Dabei gilt eine Konzentration bis 1000ppm als gute Raumluft (grün), bei 1000 - 2000ppm schaltet die Ampel auf orange und bei über 2000ppm auf rot. Diese Ampel soll als Indikator dafür genutzt werden, das Lüftverhalten in einem Raum zu bestimmen. Zudem ist es wichtig die Sensoren möglichst mittig und auf Kopfhöhe, der sich im Raum befindenden Personen, zu

platzieren. Allerdings kann man über die CO<sub>2</sub> Konzentration nicht eine definitive Aussage über die Ansteckungsgefahr treffen, da die CO<sub>2</sub> Konzentration auch durch verschiedene andere Faktoren beeinflusst wird im Gegensatz zum Ausstoß der Corona Viren. Hierbei spielen das Verhalten der Personen sowie das Tragen einer Maske eine große Rolle. Betätigen sich die Personen in einem Raum schwerer körperlicher Arbeit (Fitnessstudio, Sport) so ist der CO<sub>2</sub> Ausstoß sehr viel größer als wenn diese Personen nur Sitzen würden. Gleichzeitig verändert sich der Ausstoß der Viren nur wenig. Das gleiche gilt auch für das Sprechen und Singen. Beim Tragen einer Maske wird der Ausstoß von Viren deutlich gemindert, der CO<sub>2</sub> Ausstoß bleibt aber gleich. Man kann also zusammenfassend sagen, dass die CO<sub>2</sub> Konzentration ein guter Leitparameter ist, jedoch die räumlichen Gegebenheiten auch eine große Rolle spielen.

### 4.3 VOC

Die flüchtigen organischen Komponenten (VOC) setzen sich aus verschiedenen Stoffen unterschiedlicher Emission zusammen. Meist betrachtet man hier organische Emissionsquellen wie Lebewesen, Baumaterialien wie Holz oder andere chemische Verbindungen. Diese Stoffe sind jederzeit in der Luft enthalten und in größeren Mengen schädlich für den Menschen und andere Lebewesen. Da VOC viele Schadstoffe zusammenfasst ist es schwer daraus eine konkrete Aussage über die gesundheitlichen Folgen zu machen. Schäden können laut Inverte [9] von Geruchsbelästigung, Augen- und Atemwegsreizung über akute Vergiftungen bis hin zur Schädigung des Nervensystems, der Verstärkung von Allergien und dem Auslösen von Krebs reichen. Das Umweltbundesamt [6] unterscheidet hierbei sehr flüchtige und schwer flüchtige organische Verbindungen. Die Summe der Konzentrationen aller VOC ergibt den TVOC-Wert (total volatile organic compounds). Als Richtwerte existieren laut einem Bericht des Umweltbundesamts [1] der sog. Richtwert I und der Richtwert II. Diese Richtwerte gelten jedoch nicht für TVOC als Gesamtes sondern sind für die meisten enthaltenen Stoffe festgelegt. Dabei bestimmt der Richtwert I eine Grenze, die gilt, dass bei der Unterschreitung des Wertes auch bei lebenslanger Aussetzung keine gesundheitlichen Schäden zu erwarten sind. Im Gegensatz dazu stellt der Richtwert II eine Grenze dar, bei deren Überschreitung eine akute Gesundheitsgefahr besteht und ein unverzüglicher Handlungsbedarf besteht. Zudem wurde generell festgelegt, dass die Konzentration der TVOC bei langfristiger Aussetzung nicht über 1 - 3 mg/m<sup>3</sup> (ca 150 - 500ppb) überschritten werden sollte.

## 5 Lösungsansatz

### 5.1 SHT21

Beim SHT21 handelt es sich um einen Sensor des Unternehmens Sensirion, zur Bestimmung der Temperatur und Luftfeuchte.[4] Die Werte können hierbei mit einer Auflösung von bis zu 12 bit im Bereich der Luftfeuchte und bis zu 14

bit im Temperaturbereich bestimmt werden. Dies entspricht im abgedeckten Temperatur und Luftfeuchtebereich von -40 °C bis 125 °C bzw. 0-100 % relativer Luftfeuchtigkeit einer Auflösung von 0.04 % RH so wie 0.01° Celsius. Die vom Hersteller angegebene Genauigkeit beträgt dabei +- 2 % Luftfeuchte so wie +- 0.3 °C. Die tatsächliche Abweichung vom echten Wert ist jedoch als höher zu betrachten. Eine erneute Kalibrierung der Hardware ist vom Hersteller jedoch nicht vorgesehen. Die Messung kann dabei zu jederzeit über die I2C Schnittstelle gestartet werden. Die Maximal nötige Zeit vom Starten der Messung bis zum bereitstehen der Messung des entsprechenden Wertes über die I2C Schnittstelle beträgt hierbei 85ms bei Temperaturmessungen so wie 29ms bei Messungen der Luftfeuchte. Der Sensor hat 6 Anschlüsse (4 genutzte) für Kommunikation und Stromversorgung die wie folgt aufgebaut sind:

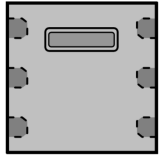
Pin	Name	Comment	
1	SDA	Serial Data, bidirectional	
2	VSS	Ground	
5	VDD	Supply Voltage	
6	SCL	Serial Clock, bidirectional	
3,4	NC	Not Connected	

Abbildung 2: Tabelle mit den Beschreibungen der Pins des SHT21[4]

Die Integration der Kommunikation und Steuerung des Sensors mit dem Mikrokontroller wurde mit folgendem Zustandsautomaten realisiert: Entsprechend der Abbildung ist ersichtlich, dass sich der SHT21 logisch hinter den dem SCD zugeordneten Zuständen befindet. Die Kommunikation der Messergebnisse ist hierbei mit 4 Zuständen ermöglicht. Der SHT21 benötigt laut Datenblatt keine manuelle Kalibrierung und keine Initialisierung und bietet im Datenblatt auch keine entsprechende Funktion an. Sobald der SHT21 mit Strom versorgt wird, kann man über den I2C-Write Befehl 0xf3 an diesen die Messung der Temperatur auf Seiten des Sensors veranlassen. Innerhalb des Zustandsdiagramm passiert dies im APP\_CALL\_FOR\_READ\_TEMP\_SHT\_STATE. Wichtig ist hier, dass diese Messung stattfindet ohne die Kommunikationsleitung während der Messung zu blockieren (no hold master), da der Prozessor des Mikrokontrollers keine Berechnung vornehmen sollte, die mehrere ms ohne Unterbrechung dauert. Sonst kann dies die Zigbee Kommunikation des Mikrokontrollers stören. Nach Übertragung des entsprechenden Befehls ist es nötig einige Zeit zu warten, der Sensor misst innerhalb von max 87ms die Temperatur und schreibt das entsprechende Ergebnis in das Register, welches über einen I2C-Read Befehl übertragen wird. Nach einer Wartezeit von 100ms wird ein Zustandsübergang in den APP\_READ\_TEMP\_SHT\_STATE veranlasst. Der dort etablierte entsprechende Read Befehl per I2C führt zur Übertragung von 2 Byte zwischen SHT21 und Mikrokontroller welche die entsprechende Temperatur enthalten. Der Übertragene Wert wird in einer Variablen gesichert und kann später über eine einfache Formel in eine menschenlesbare Temperatur überführt werden.

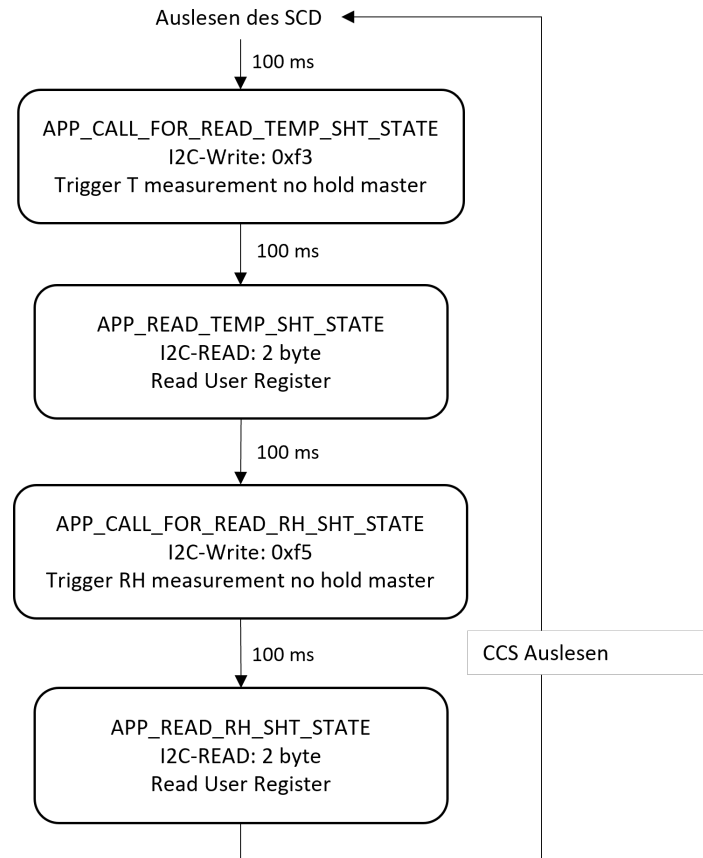


Abbildung 3: SHT21 spezifischer Teil des Zustandsautomaten[4]

Nach der Übertragung der Temperatur ist es möglich mit 2 beinahe identischen Zuständen auch die Luftfeuchtigkeit auszulesen. Hier wird zunächst in den Zustand `APP_CALL_FOR_READ_RH_SHT_STATE` gewechselt. In diesem wird über einen I2C-Write Befehl die Bytes 0xf5 an den Sensor übertragen, was eine Messung der Luftfeuchtigkeit veranlasst. Die entsprechende Messung und Übertragung in das auszulesende Register kosten den Sensor dabei bis zu 29ms. Nach 100ms wird der Zustand in den `APP_READ_RH_SHT_STATE` gewechselt und über einen I2C-Read Befehl das vorher bereitgestellte Register ausgelesen und in einer Byte-Array Variablen gesichert. Auch dieser Wert kann zu einem späteren Zeitpunkt über eine einfache Gleichung in menschenlesbare Werte übertragen werden. Nach dem Auslesen der entsprechenden Sensordaten ist die Kommunikation mit dem SHT21 beendet bis das Auslesen der Sensoren über den entsprechenden Timer erneut veranlasst wird.

## 5.2 Probleme bei der Programmierung des SHT21

Der SHT21 war der erste Sensor in diesem Projekt mit welchem die Kommunikation über I2C etabliert wurde. Entsprechend ergaben sich eine Reihe von Problemen die in erster Linie auf die Entwicklung innerhalb des durch den Atmel Mikrokontroller vorgegeben Bitcloudstack zurückzuführen sind. Die meisten Probleme traten hierbei durch die Verwendung der Funktion `HAL_OpenI2cPacket(&i2cdescriptor)` auf. Dabei war es nur schwer möglich Fehlerursachen zu identifizieren und zu replizieren. Das größte Problem bestand darin, dass beim Ausführen des Programms unvorhersehbare Sprünge zwischen den Anweisungen stattfanden. Diese waren nicht replizierbar und traten vorwiegend beim Verwenden der Uart-Kommunikation auf. Daher wurde auch zunächst nach Fehlern in der Verwendung der Uart-Schnittstelle gesucht. Es wurden entsprechende Anweisungen ausgegliedert und untersucht ob der Code weiterhin Fehler verursacht. Jedoch traten nach einiger Zeit ähnliche Fehler an anderen Stellen der Anweisungen auf. Es wurden nacheinander verschiedene Funktionen aus dem Quellcode isoliert um zu untersuchen ob der Fehler auch ohne diese noch auftritt. So konnte festgestellt werden, dass der Fehler ausschließlich auftrat wenn Anweisungen im Rahmen der bereitgestellten Funktionen des Bitcloudstacks bezüglich der I2C-Kommunikation verwendet wurden. Nach ausgiebiger Recherche des Quellcodes konnte keine Dokumentation gefunden werden, die eine gut verständliche Erklärung zu der Verwendung der entsprechenden Funktionsarchive bereitstellte. Daher wurde angedacht, die I2C-Kommunikation selbst zu programmieren. Nach einiger Recherche war jedoch klar, dass dies ein langwierigeres Unterfangen wäre als zunächst angenommen. Bei Tests wurde festgestellt, dass die entsprechenden Register zwar angesprochen werden konnten, aber bei der Programmierung waren einige Bedingungsabfragen nötig, welche den Prozessor für eine nicht explizit definierbare Zeit blockieren. Dies ist wie erwähnt nicht möglich, da die Hintergrundaktivität der Zigbeefunktionalität keine durchgehende Blockade des Prozessors erlaubt. Vermutlich ist es auch in dieser Umgebung möglich entsprechendes zu verwirklichen, aber es wurde entschieden, dass der Aufwand nicht im Verhältnis zum potenziellen Funktionsgewinn steht. Daher wurde die Entscheidung getroffen weiterhin die Funktionsweise der durch Atmel bereitgestellten Funktionen zu erfassen. Nach weiterem Testen konnte beobachtet werden dass der Fehler auftritt wenn die Funktion `HAL_CloseI2cPacket(&i2cdescriptor)` außerhalb der für die Kommunikation zu entwerfenden Callbackmethode aufgerufen wurde. Dieses Problem steht exemplarisch für einige vergleichbare aber weniger gravierende Probleme mit dem bereitgestellten Funktionsumfang von Atmel. Nachdem dieses Problem behoben wurde konnte die Implementierung des Sensors in den Zustandsautomaten jedoch effizient verwirklicht werden.

## 5.3 Funktionsweise des SHT21

Da der SHT21 zwei verschiedene physikalische Größen messen muss, sind in diesem auch 2 verschiedene Sensorikschaltkreise integriert. Zur Messung der Temperatur verwendet der SHT21 einen Bandlücken Temperatursensor. Dieser

baut auf dem Prinzip auf, dass die Spannung, welche durch eine Silikondiode übertragen wird, direkt abhängig von der Temperatur der Diode ist. Vergleicht man die Spannungsdifferenz zweier Silikondioden gleicher Temperatur bei jeweils unterschiedlicher (aber im in einem festen bekannten Verhältnis zueinander) Stromstärke so ist eine Beziehung bekannt, welche den Rückschluss auf die Temperatur der Dioden ermöglicht.

$$\Delta V_{BE} = \frac{kT}{q} \cdot \ln\left(\frac{I_{C1}}{I_{C2}}\right)$$

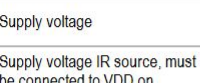
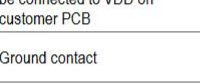
Abbildung 4: Zusammenhang zwischen Temperatur und Spannungsdifferenz zweier Silikondioden gleicher Temperatur und unterschiedlicher Stromstärke[8]

Der Luftfeuchtigkeitssensor basiert auf dem kapazitiven Prinzip. Herzstück bildet hier ein Dielektrikum, deren dielektrischen Eigenschaften möglichst Stark mit der eingeschlossenen Wassermenge und möglichst wenig mit der Temperatur korrelieren soll. Dieses Dielektrikum steht im ständigen Gleichgewicht mit der Umgebung, ist die Wasserkonzentration der Umgebungsluft höher als die des Dielektrikums, so gibt die Umgebungsluft Wasser an dieses ab und der Gehalt steigt bis sich ein Gleichgewicht eingestellt hat. Dies verändert die kapazitiven Eigenschaften des Dielektrikum welche man einfach innerhalb eines Schaltkreises bestimmen kann. Beim verwendeten Dielektrikum handelt es sich dabei meist um eine Keramik oder ein Polymer, welche gut in der Lage sind Wasser aus der Umgebungsluft aufzunehmen. [2]

## 5.4 SCD41

Beim SCD41 handelt es sich um einen photoakustischen Sensor des Unternehmens Sensirion, zur Bestimmung des CO<sub>2</sub>-Gehaltes der Umgebungsluft.[5] Weiterhin ist sowohl ein Temperatur als auch ein Luftfeuchtesensor integriert. Die Werte werden hierbei für jede gemessene physikalische Größe innerhalb von 16 bit übertragen. Die Werte für CO<sub>2</sub> werden hierbei direkt Übertragen, während bei Temperatur und Luftfeuchte ein einfacher linearer Zusammenhang zwischen übertragenem Wert und dem Wert in Grad Celsius beziehungsweise relativer Luftfeuchtigkeit besteht. Eine erneute Kalibrierung der Hardware ist vom Hersteller jedoch vorgesehen. Die Kalibrierung geschieht hierbei im Werkszustand automatisch. Dabei geht der Sensor davon aus, dass die geringste Messbare CO<sub>2</sub>-Konzentration die momentane Atmosphärenkonzentration ist und setzt den innerhalb der letzten Woche geringsten gemessenen Wert automatisch als diesen Fest. Möchte man geringere Konzentrationen messen ist es möglich die automatische Kalibrierung zu deaktivieren. Es ist ebenfalls jederzeit möglich eine Kalibrierung zu veranlassen, dabei ist jedoch sicherzustellen dass der CO<sub>2</sub>-Gehalt der untersuchten Umgebungsluft exakt 490 ppm misst. Dies ist innerhalb unserer Entwicklungsumgebung nur schwer sicherzustellen. Die Messung kann dabei zu jederzeit über die I2C Schnittstelle ausgelesen werden. Dabei wird stets

Name	Comments
VDD	Supply voltage
VDDH	Supply voltage IR source, must be connected to VDD on customer PCB
GND	Ground contact
SDA	I <sup>2</sup> C Serial data, bidirectional
SCL	I <sup>2</sup> C Serial clock
DNC	Do not connect, pads must be soldered to a floating pad on the customer PCB

Die Integration der Kommunikation und Steuerung des Sensors mit dem Mikrokontroller wurde mit folgendem Zustandsautomaten realisiert: Entsprechend der Abbildung ist ersichtlich, dass sich der SCD logisch an erster Stelle des Zustandsautomaten des gesamten Programmcodes befindet. Dabei wird zu Beginn eines jedes Zyklus zunächst die Informationen des SCD41 ermittelt. Beim starten des Mikrokontrollers wird zunächst die Initialisierung des CCS811 vorgenommen. Im Anschluss wird die Initialisierung des SCD41 durchlaufen. Hierbei wechselt die Mikrokontrollerunit in den APP\_RESET\_SCD\_STATE. In diesem Zustand überträgt die Mikrokontrollerunit den I2C-Write Befehl "0x3F86" welcher dem Sensor die Anweisung überträgt die periodische Messung zu unterbrechen, sollte eine bestehen. Dieser Befehl wird vom Sensor innerhalb von maximal 500ms Sekunden verarbeitet, in welcher Zeit keine Kommunikation mit dem SCD nicht möglich ist. Dieser Reset ist auch nach einem Neustart nötig, da sonst nicht definierte Zustände auf Seiten des Sensors auftreten können. Nach den besagten 500ms wechselt die Mikrokontrollerunit in den APP\_INIT\_SENSOR\_STATE in diesem wird wiederum die periodische Messung des Sensors veranlasst. Dies geschieht über den I2C-Write Befehl "0x21b1". Nach diesem Befehl beginnt der Sensor die periodische Messung. Um Fehlern in der Kommunikation vorzubeugen wurde sichergestellt, dass die ersten 5 Sekunden nach Beginn der periodischen Messung keine Kommunikation mit dem Sensor über die I2C-Schnittstelle stattfindet. Am Ende der Anweisungen dieses Zustands wird ein periodischer 60 Sekunden Timer des Mikrokontrollerunits gestartet an dessen Ende stets in den Zustand APP\_CALL\_FOR\_READ\_SCD\_STATE gewechselt wird. Hier wird die

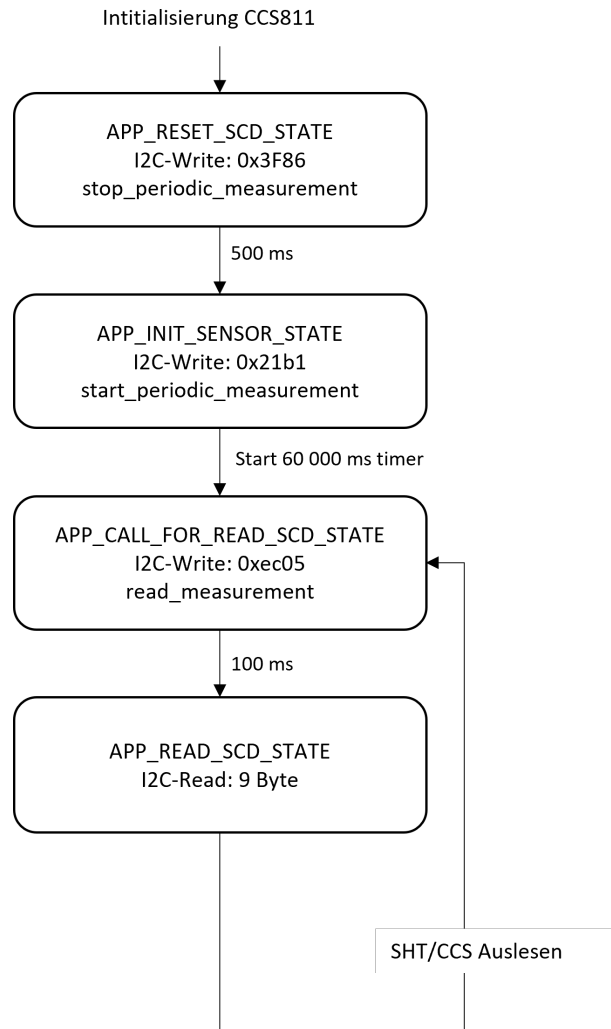


Abbildung 6: SCD41 spezifischer Teil des Zustandsautomaten[5]

I2C-Write Kommunikation mit den Bytes "0xec05" durchgeführt um dem Sensor zum bereitstellen der Daten für den nächsten I2C-Read Befehl zu veranlassen. Nach 100ms können dann die Messergebnisse über einen I2C-Read Befehl empfangen werden. Hierzu wird in den Zustand APP\_READ\_SCD\_STATE gewechselt. Die eigentliche Kommunikation der Messergebnisse ist so mit 2 Zuständen ermöglicht. Der SCD41 benötigt laut Datenblatt ebenfalls keine manuelle Kalibrierung und keine Initialisierung, bietet im Datenblatt jedoch entsprechende Funktionen an. Die standardmäßig aktivierte automatische Kalibrierung wurde dabei behalten. Es ist weiterhin möglich verschiedene Werte einzustellen,



wie die Höhe über Normalnull, Atmosphärischer Druck so wie den Temperatur Offset. Über die uns zur Verfügung stehenden Sensoren ist hierbei jedoch ausschließlich der Temperaturoffset zu bestimmen. Laut Datenblatt hat dieser jedoch keinen Einfluss auf die CO<sub>2</sub> Messung, weswegen darauf verzichtet wurde. Nach dem Auslesen der entsprechenden Sensordaten und speichern derselben in entsprechenden Variablen ist die Kommunikation mit dem SCD41 beendet bis das Auslesen der Sensoren über den entsprechenden Timer erneut veranlasst wird.

## 5.5 Probleme bei der Programmierung des SCD41

Der SCD41 war der zweite Sensor bei welchem die Kommunikation mit der Mikrokontrollereinheit etabliert wurde. Grundlegende Erkenntnisse aus Problemen der Verbindung des SHT21 konnten so erfolgreich berücksichtigt werden. Zunächst war es jedoch nicht möglich die Kommunikation mit dem Sensor zu etablieren. Selbst bei exaktem Vorgang nach dem Datenblatt lehnte der Sensor teilweise nach Neustart die Kommunikation mit der Mikrokontrollereinheit ab. Das Problem trat hierbei vereinzelt auf und war schwer replizierbar. Innerhalb des Datenblatts konnten dabei keine Hinweise auf diese Fehlfunktion entdeckt werden. Um dem Problem auf den Grund zu gehen wurde versucht die Funktionsweise des Sensors zunächst auf dem beiliegenden Arduino Nano zu etablieren. Hierbei trat das gleiche Problem auf. Um mögliche Fehlerquellen zu identifizieren wurde die vom Hersteller zur Verfügung gestellte Code-Bibliothek mit Beispielcode verwendet um herauszufinden ob hier ähnliche Probleme zu erkennen sind. Bei Verwendung dieser konnten keine Probleme festgestellt werden. Es wurden daraufhin die entsprechenden Codebausteine analysiert um herauszufinden in welcher Form der Hersteller die Kommunikation mit dem Sensor etabliert. Hierbei wurde herausgefunden, dass der Hersteller zur Initialisierung des Sensors stets den `stop_periodic_measurement` Befehl sendet und danach erneut die `start_periodic_measurement` Anweisung ausführt. Nach Einfügen dieses Vorgangs in die Initialisierung des Sensors, konnte das beobachtete Verhalten des Sensors in der I<sup>2</sup>C-Kommunikation korrigiert werden.

## 5.6 Funktionsweise des SCD41

Wie erwähnt handelt es sich bei dem Sensor um einen photoakustischen Sensor. Bei diesem wird über kurze Lichtblitze dem Gas in einer Gaskammer Energie zugeführt. Diese Energie führt zu einer Steigerung der thermischen Bewegung der Moleküle und damit zu einer Ausdehnung des eingeschlossenen Gases. Zwischen den Lichtblitzen, wird diese zugeführte Energie teilweise wieder an die Umgebung abgegeben, das Gas zieht sich wieder zusammen. Dies verursacht eine Vibration innerhalb der Gaskammer, welche mit einem Mikrofon gemessen werden kann. Nun ist es die Eigenschaft verschiedener Moleküle, dass die verschiedenen Freiheitsgrade der Molekülbewegung wesentlich besser von Lichtenergie einer Wellenlänge angeregt wird die in einem spezifischen Verhältnis zum Atomabstand innerhalb des Moleküls steht. So ist es wie in der Abbil-

dung ersichtlich, möglich mit Licht einer spezifischen Wellenlänge beinahe ausschließlich die CO<sub>2</sub>-Moleküle anzuregen. Die Stärke der gemessenen Vibrationen (Lautstärke) steht dann in direkter Korrelation mit dem CO<sub>2</sub>-Gehalt der Luft, was einen Rückschluss auf diesen ermöglicht.

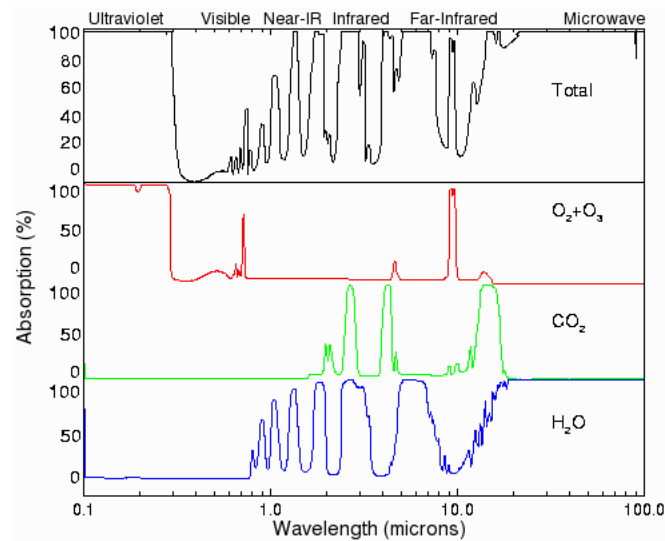


Abbildung 7: Absorptionsspektren einiger Atmosphärenbestandteile[10]

## 5.7 CCS811

Der CCS811 [3] ist ein Sensor von ams, der den äquivalenten CO<sub>2</sub> (eCO<sub>2</sub>) Gehalt und den Gehalt der totalen flüchtigen organischen Verbindungen (TVOC) in der Luft messen kann. Dabei kann eCO<sub>2</sub> von 400ppm bis 8192ppm, TVOC von 0ppb bis 1187ppb gemessen werden und ein Messintervall von 250ms, 1s, 10s, 60s festgelegt werden. Die Pin-Beschreibungen kann man in Abbildung 8 sehen. Wichtig sind hiervon die Pins:

- VDD: Anschluss an 3,3V als Versorgungsspannung (min 1,8V)
- nWAKE: Anschluss an GND (active low), damit der Sensor durchgehend aktiv ist
- SDA: Data Anschluss für I2C, wird mit einem SDA Pin des Zigbee-Boards verbunden
- SCL: Clock Anschluss für I2C, wird mit einem SCL Pin des Zigbee-Boards verbunden
- GND: Ist nicht in der Tabelle angegeben, muss aber mit GND auf dem Zigbee-Board verbunden werden

In Abbildung 9 ist das Zustandsdiagramm des CCS811 Sensors dargestellt. Zustände die nicht mit dem Sensor interagieren sind hier nicht dargestellt. Zu Beginn wird 2000 ms gewartet da der SCD41 Sensor entsprechende Zeit zum starten braucht. Die eigentliche Startzeit des CCS811 beträgt maximal nur 20ms. Der Zugriff auf den Sensor über den I2C-Bus spricht die Adresse 0x5A an, da der ADDR Pin nicht gesetzt ist. Im APP\_RESET\_CSS\_SW\_STATE wird mit 0xFF das SOFTWARE\_RESET\_REG angesprochen und der Befehl 0x11 0xE5 0x72 0x8A geschrieben. Dieser Befehl dient dazu einen versehentlichen Software Reset zu verhindern. Ist diese Sequenz geschrieben worden, befindet sich der Sensor im Boot mode. Zwischen den einzelnen I2C-Befehlen wird immer 1ms gewartet um sicherzugehen, dass der Befehl richtig ausgeführt werden kann. Dies führt zwar zu einem erhöhten Zeitaufwand, ist jedoch unbedenklich, da der Messintervall 60 s beträgt und somit genügend Zeit zur Verfügung steht. Im nächsten Schritt wird das HW\_ID\_REG angesprochen und im darauf im folgenden Zustand auch ausgelesen. Hierbei ist wichtig, dass die HW\_ID 0x81 ist. Im APP\_CCs\_CHANGE\_TO\_APPSTATE\_STATE wird mit dem Befehl 0xF4 zurück in den Appstate gewechselt. Nun wird das MEAS\_MODE\_REG 0x01 angesprochen und der Befehl 0x30 geschrieben. Dieser Befehl setzt den Messintervall auf 60s, welcher den geringsten Stromverbrauch hat. Zum Testen der Applikation wurde hier ein kürzerer Intervall gewählt. Zudem werden keine Interrupts gesetzt die feuern würden wenn Daten vorhanden wären. Um zu überprüfen ob die Messung funktioniert, wird nun 60s gewartet und dann das STATUS\_REG 0x00 überprüft, ob Daten vorhanden sind. Ist dies der Fall, ist das 3. Bit gesetzt. Falls ein Fehler auf dem I2C-Bus oder dem Sensor aufgetreten ist, könnte man dies über das 0. Bit feststellen und im Register 0xE0 auslesen. Sind Daten

Pin No.	Pin Name	Description
1	ADDR	Single address select bit to allow alternate address to be selected <ul style="list-style-type: none"> <li>• When ADDR is low the 7 bit I<sup>2</sup>C address is decimal 90 / hex 0x5A</li> <li>• When ADDR is high the 7 bit I<sup>2</sup>C address is decimal 91 / hex 0x5B.</li> </ul>
2	nRESET	nRESET is an active low input and is pulled up to V <sub>DD</sub> by default. nRESET is optional but external 4.7KΩ pull-up and/or decoupling of the nRESET pin may be necessary to avoid erroneous noise-induced resets.
3	nINT	nINT is an active low optional output. It is pulled low by the CCS811 to indicate end of measurement or a set threshold value has been triggered.
4	PWM	Heater driver PWM output. Pins 4 and 5 must be connected together.
5	Sense	Heater current sense. Pins 4 and 5 must be connected together.
6	V <sub>DD</sub>	Supply voltage
7	nWAKE	nWAKE is an active low input and should be asserted by the host prior to an I <sup>2</sup> C transaction and held low throughout.
8	AUX	Optional AUX pin which can be used for ambient temperature sensing with an external NTC resistor. If not used leave unconnected.
9	SDA	SDA pin is used for I <sup>2</sup> C data. Should be pulled up to V <sub>DD</sub> with a resistor
10	SCL	SCL pin is used for I <sup>2</sup> C clock. Should be pulled up to V <sub>DD</sub> with a resistor
EP	Exposed Pad	Connect to ground

Abbildung 8: Tabelle mit den Beschreibungen der Pins des CCS811

zum abrufen bereit, werden die Sensoren SHT21 und SCD41 initialisiert. Dabei wird auch der Messtimer von 60s gestartet. Sind diese 60s abgelaufen und die beiden Sensoren ausgelesen, so wird das ALG\_RESULT\_DATA\_REG 0x02 angesprochen in dem die bereits berechneten Daten der Messung bereit liegen. Diese werden als 4 Byte ausgelesen, wobei die Bytes 0 und 1 den eCO<sub>2</sub> Wert beinhalten und Byte 3 und 4 den TVOC Wert. Eine Umrechnung der Daten ist nicht nötig, da diese bereits der Sensor selbst umrechnet und somit im richtigen Format ausgelesen werden können. Nachdem die Daten ausgelesen wurden werden sie an den Koordinator geschickt.

### 5.7.1 Probleme bei der Programmierung des CCS811

Da sich der CCS811 in seiner Funktionsweise von den anderen beiden Sensoren unterscheidet, war es zu Anfang schwierig einen richtigen Ansatz zu bekommen. Vor allem das Wechseln in den Bootmodus war erst nach der Erläuterung in der

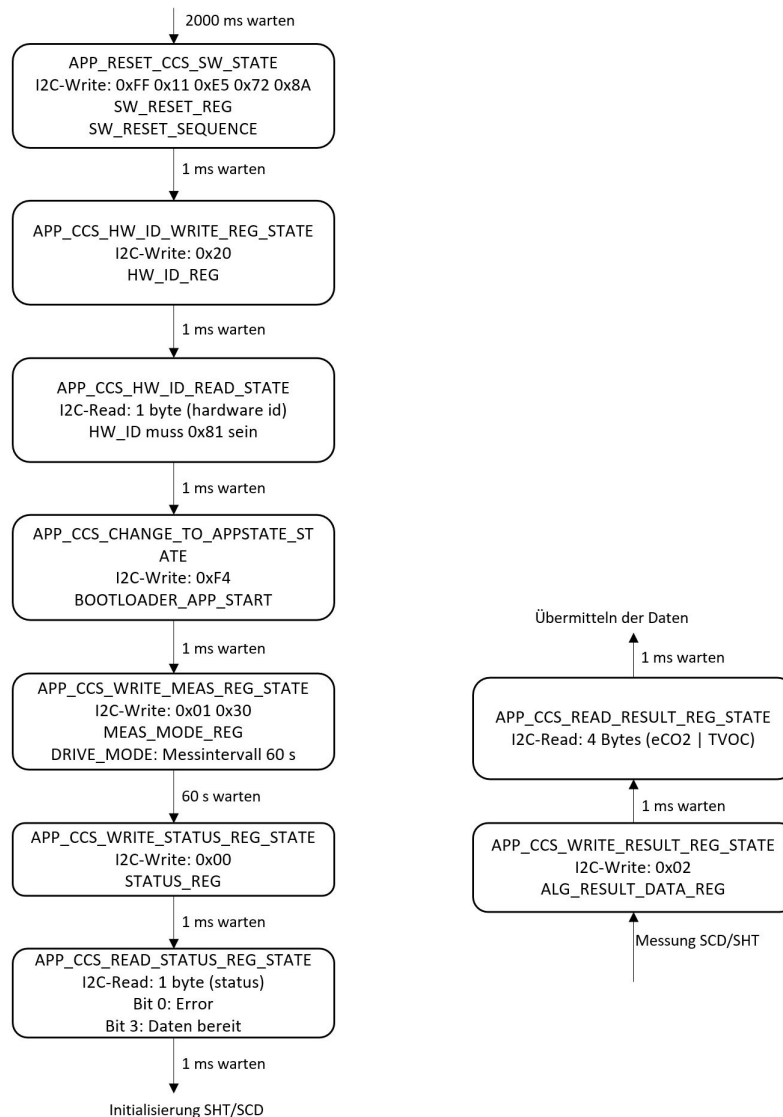


Abbildung 9: Zustandsdiagramm des CCS811 Sensors, rechts: Initialisierung, links: Auslesen der Werte

Vorlesung klar und damit auch nachvollziehbarer. Zudem haben teilweise die Funkmodule nicht richtig funktioniert, da entweder überhaupt nichts über die UART ausgegeben werden konnte oder die bei einem I2C-Befehl der Callback mit Error zurück gekommen ist. Nachdem wir den Code mehrmals auf Richtigkeit der Befehle und Ablauf überprüft haben, haben wir das ZigBee-Board

ausgewechselt und das Messen hat ohne Probleme funktioniert. Was dabei der genaue Fehler war ist uns nicht bekannt. Zudem war die Funktion der Pins nWAKE und ADDR zu Anfang nicht klar. Der nWAKE Pin ist dafür da damit der Sensor aktiv ist und muss dafür auf low gezogen werden. Also haben wir diesen and GND angeschlossen und somit ist der Sensor durchgehend aktiv. Der ADDR Pin gibt an ob die I2C-Adresse des Sensors 0x5A oder 0x5B ist. Da der ADDR Pin nicht angeschlossen ist, ist die I2C-Adresse 0x5A.

### 5.7.2 Funktionsweise des CCS811

Der CCS811 Sensor ist ein Metalloxid Sensor, der mittels eines anorganischen Metalloxid-Halbleiters, welcher mit der Zielsubstanz (Hier CO<sub>2</sub> und VOC) reagiert und dadurch seinen Widerstand ändert. Aus der Änderung des Widerstands kann dann die Konzentration der Zielsubstanzen ermittelt werden [7]. Laut dem Datenblatt des Sensors wurde hierfür die von ams entwickelte micro-hotplate technologie"verwendet. Probleme bei diesem Verfahren ist das Aussondern der Zielsubstanz und das Verhindern der Beeinflussung durch andere Substanzen (andere Gase) oder Einflüsse (Luftfeuchte). Die Beeinflussung durch andere Substanzen kann durch die Wahl des Metalloxids eingeschränkt werden, andere Einflüsse können wenn überhaupt nur durch ständige Anpassung der Berechnung ausgeglichen werden. Um eine Substanz, hier z.B. CO<sub>2</sub> zu messen, reagiert das Metalloxid mit dem Gas. CO<sub>2</sub> ist dabei ein oxidierendes Gas, gibt also Sauerstoff an das Metalloxid ab und verringert dadurch die Leitfähigkeit des Halbleiters. In Abbildung 10 kann man dazu einige Beispiele sehen.

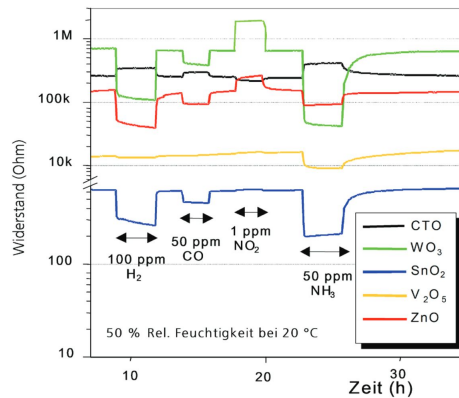


Abbildung 10: Verschiedene Metalloxide und die Veränderung des Widerstands durch die Reaktion mit verschiedenen Gasen[13]

## 5.8 Messung der Eingangsspannung

Für die Messung der Eingangsspannung wurde der Analog-to-Digital Converter genutzt, auch ADC genannt, welcher sich auf dem ATmega256RFR2 befindet. Dieser wandelt analoge Signale in digitale um. Für die Eingangsspannung muss der ADC0-Pin ausgelesen werden. Hierfür nutzt man die Register, die im Datasheet S.465 beschrieben sind. Die Bits von MUX5:0 werden auf 0 gesetzt (Referenz)Z.4-5. Des Weiteren wird die interne Referenzspannung von 1.5V im ADMUX Register ausgewählt. Dafür setzt man das REFS1 Bit auf eins und REFS0 auf Null (Referenz)(Z.4). Jetzt wird noch der ADC eingeschaltet, indem man im ADCSRA Register das ADEN Bit auf Eins stellt. Danach startet man die Messung, indem man im ADCSRA Register das ADSC Bit auf Eins stellt (Referenz)(Z.8). Wenn die Messung erfolgt ist, wird das ADSC Bit auf Null gestellt und das Ergebnis im ADCW Register gespeichert. Diesen Wert wird mit sechs multipliziert, um den Messwert in Millivolt zu erhalten. In Listing (ATMEGADATASHEETT REFERENCE S. 465) Es werden mehrere Werte gemessen, um so einen genaueren Durchschnitt zu berechnen.

```
1  static uint16_t vcc_read(void){
2      uint32_t adcval = 0;
3
4      ADMUX |= (1<<REFS1)|(0<<REFS0)|(0<<MUX4)|(0<<MUX3)|(0<<MUX2)|(0<<MUX1)|(0<<MUX0);
5      ADCSRB |= (0<<MUX5);
6      ADCSRA |= (1<<ADEN); // ADC einschalten
7      for(uint16_t i=0; i<NUM_MESS_ADC; i++){
8          ADCSRA |= (1<<ADSC); // start messung
9          while (ADCSRA&(1<<ADSC)){
10             }
11             adcval += ADCW;
12         }
13         ADCSRA &= ~(1<<ADEN); // ADC ausschalten
14         return (((adcval)/NUM_MESS_ADC) * 6);
15     }
```

Listing 1: Delay Timer

## 5.9 Probleme mit der Batterie

Für die korrekte Funktion eines Endgerätes mit den drei Sensoren ist die Batteriespannung wichtig. Da sonst die Sensoren falsche Werte senden. Bei den verwendeten Batterien ist nach einer Laufzeit von ca 10 Stunden die Mindestspannung von 2.4V für den SCD41 unterschritten. Während des Betriebs wurden zwischen 45-55 mA Stromaufnahme für das Endgerät gemessen. Für eine dauerhafte Nutzung des Endgerätes im Batteriebetrieb muss der Stromverbrauch deutlich gesenkt werden.

## 5.10 Datenübertragung

Bei der Datenübertragung wurde entschieden, alle Sensordaten in einem uint8\_t Array zu versenden. Dieses Array ist somit immer 44 Byte lang. Die verschiedenen Werte sind durch Semikolons getrennt. Es folgt die Liste mit allen Werten aus dem Array mit passender Umrechnung und Maßeinheit:

1. ID es Mikrocontrollers
2. SHT21 Temperatur (Wert/100 in °C)
3. SHT21 Relative Luftfeuchte (Wert/100 in %)
4. SCD41 Temperatur (Wert/100 in °C)
5. SCD41 Relative Luftfeuchte (Wert/100 in %)
6. SCD41 CO2 (Wert in ppm)
7. CCS811 eCO2 (Wert in ppm)
8. CCS811 TVOC (Wert in ppb)
9. Spannung (Wert in Millivolt)

1.	2.	3.	4.	5.	6.	7.	8.	9.
<pre>1;+2164;4224;+2124;3914;0675;0700;0045;2622;</pre>								

Abbildung 11: Beispiel Messdaten Array

Wenn alle Werte im Array sind, müssen diese in die Payload übertragen werden. Mit diesem Befehl `APS_DataReq(&datareq);` werden die Daten an den Koordinator gesendet.

## 5.11 Netzwerkaufbau

### 5.11.1 Aufbau

In dem Netzwerk gibt es drei verschiedene Rollen.

1. Der Router leitet die ankommenden Daten an den Koordinator oder an andere Router weiter.
2. Der Koordinator ist dafür verantwortlich, dass das ZigBee-Netzwerk startet. Er fungiert auch als Router.
3. Das Endgerät kommuniziert nur mit dem Router und Koordinator.

In der Header Datei `configurations.h` müssen verschiedene Voreinstellungen vorgenommen werden.

1. Die Rolle des Gerätes in der Variablen `CS_DEVICE_TYPE`
2. Der Unique identifier auch genannt `CS_UID`. Dieser ist 64-Bit lang. Der Koordinator hat die UID `0x0700000A01LL`. Die Endgeräte beginnen bei der UID `0x0700000A03LL` und jedes weitere wird die letzte Stelle um eins erhöht.



3. Die eindeutige Netzwerk PAN-ID. Diese ist 64-Bit lang und wird in der Variablen CS\_EXT\_PANID mit dem Wert 0x1AAAAAAAAAAAAACA07LL definiert.

In Abbildung Nr.12 wird der Verbindungsaufbau dargestellt.

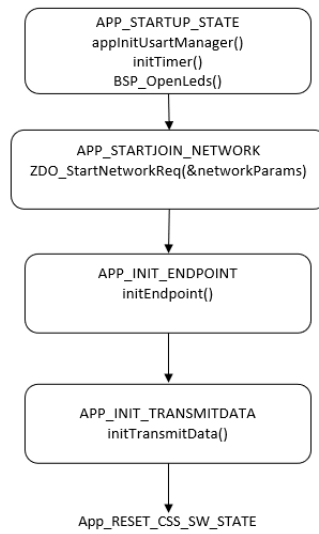


Abbildung 12: Beispiel Messdaten Array

### 5.11.2 Probleme beim Zusammenfügen der Sensoren und des Netzwerkes

Als der Code für die Sensoren zusammengefügt wurde, hat der SCD41 und der CSS811 plausible Werte ausgegeben. Der SHT21 hingegen hat zu hohe Werte geliefert. Wenn man eine Batterie an das Endgerät angeschlossen hat, wurden falsche Messdaten empfangen. Bemerkenswert war, dass nur der SCD41 falsche Werte liefert. Später bei dem Messen der Spannung hat sich herausgestellt, dass die minimale benötigte Spannung von 2,4 Volt unterschritten worden ist. Deshalb konnte der Sensor nicht korrekt ausgelesen werden. Darum wurden die aktuellen Werte nicht mehr aktualisiert und immer der gleiche Wert versendet. Wenn das Gerät neu gestartet worden ist, konnte der Sensor nicht initialisiert werden und es wurde immer -45.00 °C und Nullen bei der Luftfeuchte und CO2 gesendet.

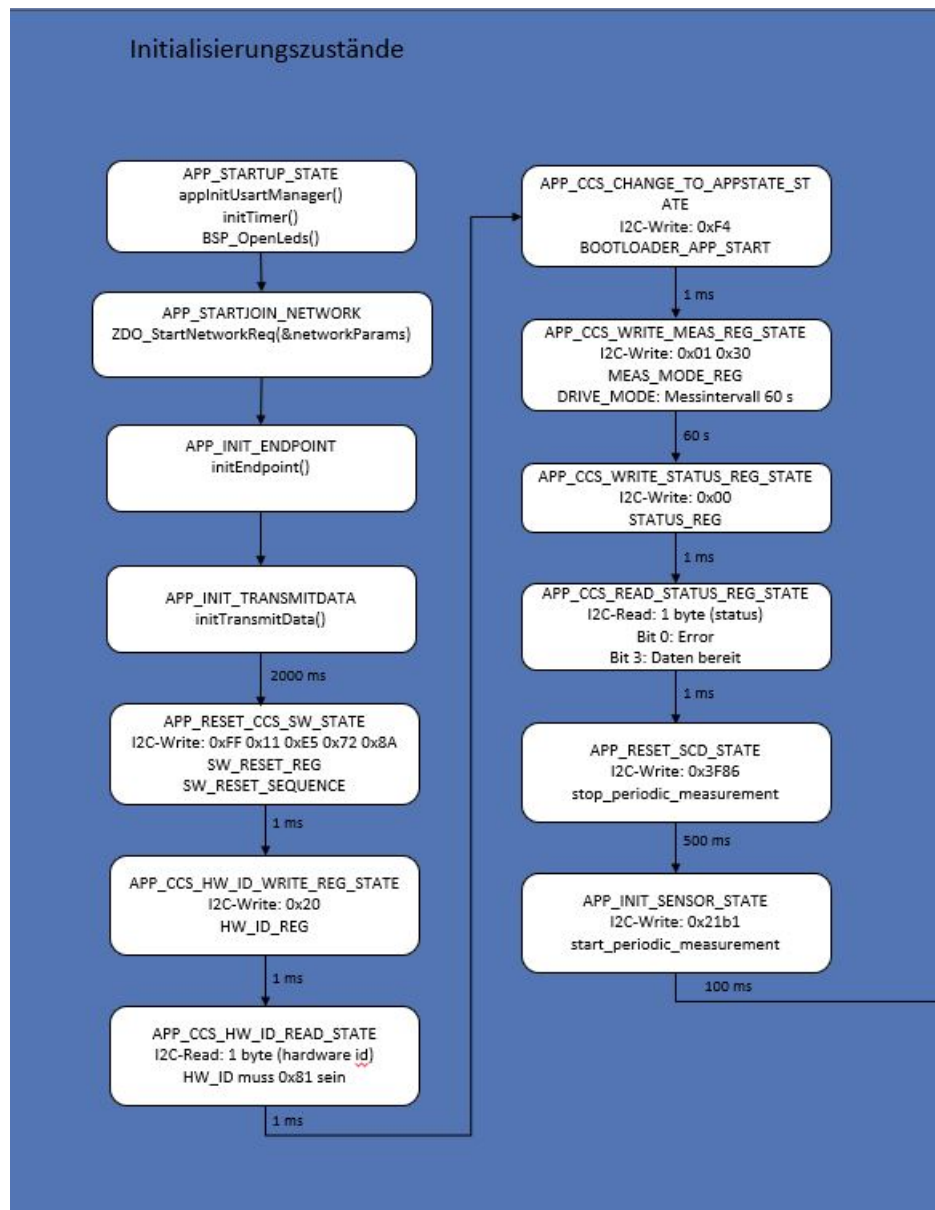


Abbildung 13: Übersicht über die Initialisierungszustände

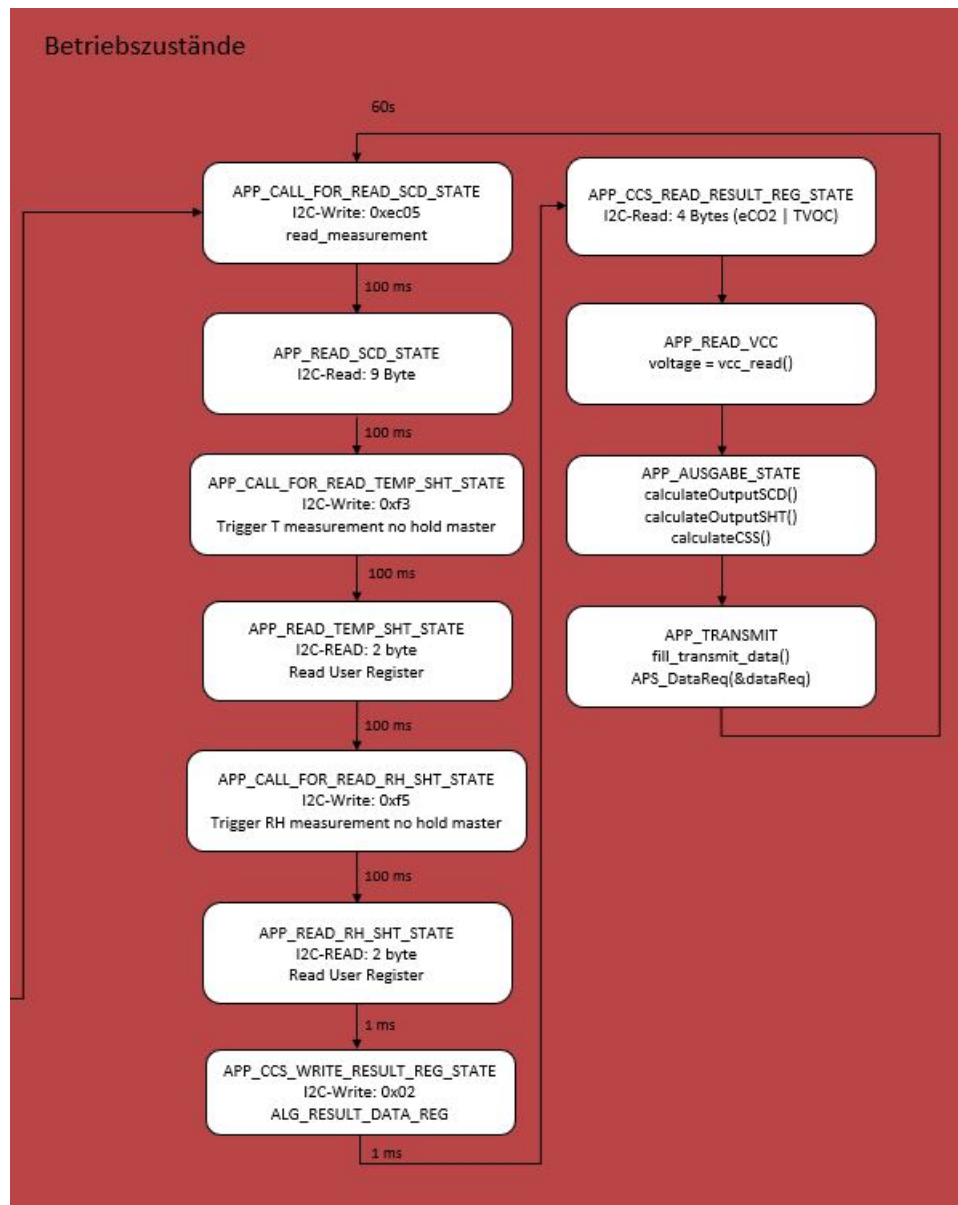


Abbildung 14: Übersicht über die Betriebszustände

## 5.12 Serverimplementierung

Für die Serverimplementierung wurde zum einen die SocketIO-Library integriert. Hiermit lässt sich ein HTTP-Server erstellen, welcher verschiedene Anfragen des Clients bearbeiten kann. In der Applikation gibt es sieben verschiedene Endpunkte, welche behandelt werden, darunter das Anzeigen, Updaten oder Löschen der Räume und Sensoren. Allerdings werden über HTTP keine Sensordaten versendet. Der Grund dafür ist, dass selbst wenn der Server Daten von der Datenbank erhält, solange nichts passiert, bis der Client eine Anfrage bezüglich der Daten senden würde. Dies würde die Effizienz und Performance stark einschränken.

Doch für dieses Hindernis werden für die Datenübertragung Websockets verwendet. Die Websockets sind im Zeichentool sowie im Dashboard integriert. Diese ermöglichen grundlegend eine Echtzeit-Kommunikation zwischen Client und Server und übermitteln Daten ohne auf die Anfrage eines Clients zu warten. Die Datensätze werden in einem SensorData-Objekt erstellt und in einer Variable abgespeichert. Diese Objekte werden dann wiederum in einem Array abgespeichert, welches z.B. verwendet wird um Sensordaten in den Diagrammen anzeigen zu lassen.

## 5.13 Zeichentool

### 5.13.1 Aufbau und Funktionsweise

Die einzelnen Komponenten des Zeichentools wurden mit Vue erstellt. Diese bestehen aus HTML und CSS Elementen sowie einem Skript-Teil, in welchem jegliche Funktionalitäten in Java- oder Typescript programmiert werden können. Die Buttons sind beispielsweise im HTML-Abschnitt eingebettet und Funktionen zur Realisierung des Hinzufügens von Sockets für die Live-Daten Übertragung oder eine Anknüpfung an die Datenbank um gezeichnete Räume abspeichern zu können, sind im Skript-Abschnitt implementiert. Für das Zeichentool wurde zunächst ein Prototyp geschrieben, um zu testen inwiefern man Räume zeichnen können soll. Nachdem dieser erstellt worden war, wurde auf diesem Prototyp aufgebaut und das eigentliche Zeichentool implementiert. Mit einem Doppelklick lassen sich Punkte auf die Leinwand platzieren, sobald ein zweiter Endpunkt hinzugefügt wurde, entsteht eine Linie zwischen diesen Punkten. Erst nachdem ein geschlossener Raum gezeichnet und ihm ein Name gegeben wurde, kann man diesen permanent speichern und anzeigen lassen. Wenn man den Button **Show Rooms** anklickt, wird in einem kleinen Fenster jeder Raum angezeigt, welcher gespeichert wurde. In dieser Übersicht gibt es drei weitere Funktionen, es können Sensoren in den verschiedenen Räumen platziert werden, überflüssige Räume können wieder entfernt und es kann über den Info-Button eine Tabelle angezeigt werden lassen, welche die Zahlenwerte der Sensoren anzeigt.

Auch die Ampelindikation ist im Zeichentool verwirklicht worden. Wenn man einen Sensor in einen Raum hinzufügt, kann dieser frei innerhalb des Raumes bewegt werden. In diesem Fall ist der Sensor selbst die Ampelindikation. Abhängig von der CO<sub>2</sub>-Konzentration, ändert die Indikation die Farbe von grün auf gelb

und von gelb auf rot.

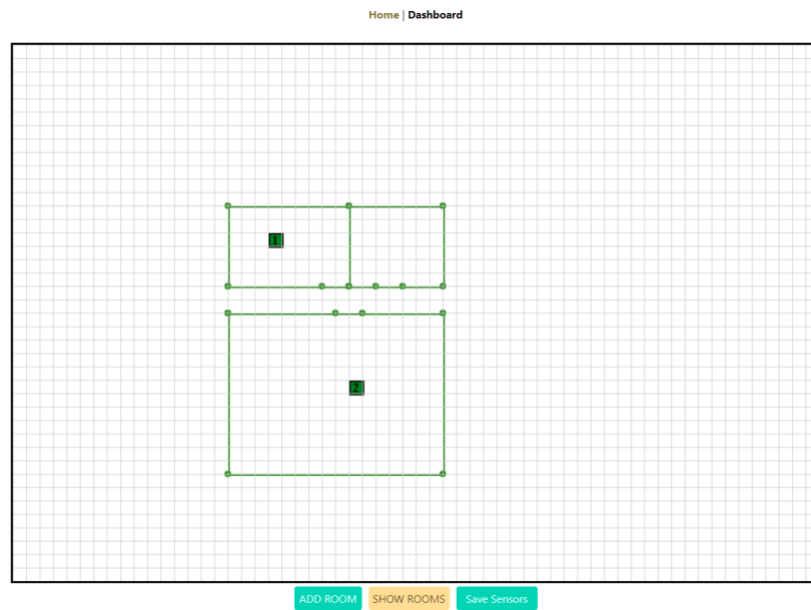


Abbildung 15: Aufbau des Zeichentools

### 5.13.2 Probleme beim Zeichentool

Wenn ein Raum gezeichnet und nicht direkt abgespeichert wird, kann man die Eckpunkte des Raumes auswählen und dadurch weiterhin durch die Leinwand bewegen. Somit können Räume möglicherweise noch nachjustiert werden. Wenn man allerdings unbeabsichtigt die Linie auswählt, wird diese von den Punkten getrennt. Dies hat den Grund, dass die Linie dynamisch implementiert wurde um dem User eine Nachjustierung zu ermöglichen. Wäre die Linie statisch eingebunden worden, könnte man sie nach der Entstehung zwischen zwei Punkten in keinerlei Weise beeinflussen. Der Anwender kann diesen Fehler allerdings ganz einfach wieder beheben, indem einer der Punkte ausgewählt und bewegt wird. Ein weiterer Punkt wäre eine Verzögerung bei der Informationstabelle, wenn der Info-Button betätigt wird. Die Tabelle zeigt zunächst keine Werte an, diese werden erst nach 1 Minute geladen.

## 5.14 Dashboard

### 5.14.1 Funktionsweise

Das Dashboard ist dazu da, die ankommenden Sensordaten visuell darzustellen. Es besteht aus 4 Liniendiagrammen, die die Werte von Temperatur, relativer Feuchtigkeit, CO<sub>2</sub> und TVOC anhand eines Liniengraphen anzeigen. Der Anwender hat die Möglichkeit verschiedene Zeiten einzustellen, somit kann er die Daten der vergangenen letzten 6, 24 und 168 Stunden aufrufen. Wenn einer der Zeit-Buttons benutzt wird, kann man für die nächsten drei Sekunden keine weitere Eingabe tätigen und werden erst nach diesem Zeitraum wieder freigeschaltet. Es braucht nämlich 3 Sekunden, bis die Daten vom Backend im Frontend geladen werden.

Die Y-Achse passt sich entsprechend den Werten der übertragenen Daten an. Die X-Achse hingegen besteht aus genau 120 Ticks, abhängig davon ob 6, 24 oder 168 Stunden angegeben werden, passen sich die Sensorwerte an die 120 Ticks an. Das bedeutet, wird die 6 Stunden Option ausgewählt, werden Daten in 3 Minuten Takten angezeigt, bei 24 Stunden sind es 12 Minuten Takte und bei 168 Stunden bzw. 1 Woche sind es 84 Minuten Takte. Der Grund für die statische Anzahl an Ticks ist, dass es bei einer höheren Zahl zu Performance-Einbrüchen kommen würde und die Anwendung dadurch langsamer arbeitet.



Abbildung 16: Aufbau des Dashboards

### 5.14.2 Aufgetretene Probleme

Zunächst war festgelegt, dass die JavaScript-Library D3.js verwendet wird um die graphische Darstellung der Sensordaten zu realisieren. Diese enthält Funk-

tionen um verschiedenste Diagramme zu erstellen. Die Funktionen manipulieren HTML-Elemente, dadurch können z.B. einerseits deren Größe und Breite eingestellt oder andererseits können komplett neue HTML-Elemente hinzugefügt werden. Das Erstellen von vier verschiedenen Liniendiagrammen und einer Ampel war zunächst schnell abgeschlossen. Nach dieser Erstellung wurde ein Socket implementiert um zu überprüfen, ob die Sensordaten im Dashboard ankommen, was im Anschluss auch der Fall war. Allerdings ist hier ein Problem hervorgekommen.

Die Sensorwerte sollten als Y-Koordinate und die dazugehörige Zeit als X-Koordinate angezeigt werden. Zwar gibt es in D3.js eine eigens dafür integrierte `scaleTime`-Funktion um eine Zeitachse zu erstellen, aber durch die Zeiteinheit konnte nun kein Graph angezeigt werden. Dies hatte den Grund, dass zu diesem Zeitpunkt kein Zeitwert von der Datenbank übertragen wurde. Für dieses Problem konnte keine direkte Lösung hinsichtlich der Implementierung gefunden werden, daher wurde kurzfristig entschieden auf die `Chart.js`-Library umzusteigen. Diese bietet beispielsweise fertige Diagramme an, in welchen man lediglich personalisierte Werte hinzufügen kann.

Während der Implementierung ist hervorgekommen, dass wenn der Anwender zu schnell zwischen dem Zeichentool- und Dashboardfenster hin und her wechselt, kann es passieren das keinerlei Graphen bzw. Daten im Dashboard angezeigt werden, da diese nicht fertig geladen wurden. Dies kann allerdings mit einem erneuten Neuladen der Dashboardseite wieder behoben werden. Ein ähnliches Problem gibt es bei den Zeit-Buttons unter den Diagrammen. Wenn zu schnell zwischen den Zeiträumen gewechselt wird, kann es passieren, dass die Graphen nicht geladen werden.

## 5.15 COM-Port & Mockdaten

### 5.15.1 Mockdaten

Zu Beginn des Projekts wurde zur Simulation von Sensordaten ein Python-Programm genutzt. Der Sinn dahinter war, auch Gruppenmitgliedern, die sensorisch nicht voll ausgestattet sein konnten, Daten zur Verfügung zu stellen, um die Anwendung im Back- und Frontend testen zu können. Auch sollte durch die Verwendung von Mockdaten das Back- und Frontend-Team nicht auf die Vollendung der Programmierung der ZigBee Module durch das ZigBee-Team warten müssen, um die Applikation zu testen.

Um die Idee in die Tat umzusetzen, wurde die `“pySerial”` API genutzt. Das Programm erstellte Datensätze und schickte diese an eine COM-Schnittstelle des Computers. Da zu diesem Zeitpunkt noch die Idee im Raum war, dass die Daten als JSON-string von den ZigBee Modulen an den Computer geschickt werden sollen, wurden die Datensätze im Python-Programm als JSON-Strings formatiert und abgeschickt.

Im weiteren Verlauf des Projekts wurde sich, was das Format der übertragenen Daten angeht, umorientiert. Es wurde klar, dass das Verhältnis zwischen Aufwand und Nutzen bei Nutzung eines JSON-Formats nicht gut genug war. Des-



halb wurde sich dazu entschieden, die Daten in Form eines einfachen Strings, in dem die Daten jeweils durch ein Semikolon getrennt werden, an den Computer zu übertragen (Abb. 17). Des Weiteren wurde, um eine realitätsnahe Datenübertragung simulieren zu können, die Verwendung des Python-Programms verworfen und auf ein ZigBee C-Programm zur Datensimulation gewechselt. Hierzu wurde ein ZigBee Modul darauf programmiert, einen String im zuvor erwähnten Format an die COM-Schnittstelle in einem vordefinierten Intervall zu schicken. Die Intervalllänge wurde dabei aus Zeiteffizienzgründen auf 10 Sekunden gesetzt.



```
"sensorId;tempSHT21;humSHT21;tempSCD41;humSCD41;co2SCD41;eco2CCS811;tvocCCS811;battery;"
```

Abbildung 17: Format in dem die Datensätze übertragen werden

### 5.15.2 Kommunikation über den COM-Port

Die Kommunikation zwischen Computer (Backend) und dem ZigBee Modul und somit auch den Datenempfang und die Datenverarbeitung haben wir mit Hilfe des Node.js Packages “SerialPort” realisiert.

Im Backend wird zur Kommunikation mit dem ZigBee Modul ein SerialPort-Objekt genutzt. Dieses wird erstellt, sobald der Anschluss eines ZigBee Moduls an den Computer von unserer Anwendung registriert wird.

Das erstellte SerialPort-Objekt benötigt bei der Erstellung einige Informationen über das angeschlossene Gerät, um einen reibungslosen Datenempfang und eine daran anschließende Verarbeitung zu ermöglichen. Dabei handelt es sich vor Allem um den Port, über den das Gerät kommuniziert und die verwendete Baudrate. Da der verwendete COM-Port von Nutzer zu Nutzer variiert, musste sich überlegt werden, wie eine automatische Erkennung des richtigen COM-Ports implementiert werden kann. Eine manuelle Eingabe des Namens des richtigen COM-Ports durch den Nutzer war nicht gewollt, da dies auf Kosten des Nutzerkomforts geschehen würde.

Da das SerialPort Package keine Funktion zur Erkennung des richtigen COM-Ports zur Verfügung stellt, musste sich überlegt werden, wie dies mit den vorhandenen Mitteln umgesetzt werden könnte. Die Umsetzung ist durch die Entwicklung und Implementierung einer Funktion gelungen, die den richtigen Portnamen ermittelt. Das Vorgehen ist dabei wie folgt:

Die Funktion erstellt eine Liste aller verfügbaren COM-Ports des Computers und iteriert über alle (Abb. 18, Z.4-9). Bei jedem Port wird dabei überprüft, welchen manufacturer und welche vendorId das angeschlossene Gerät besitzt. Wenn dabei das angeschlossene Gerät den manufacturer “FTDI” und die vendorId “0403” hat, dann handelt es sich bei dem angeschlossenen Gerät um ein ZigBee Modul und es wird der entsprechende Name des Ports, über den das Modul kommuniziert, in die Variable “portName” gespeichert.

Durch einen in den Server eingebauten Timer, der immer wieder `getPortName()` in einem Intervall von einer Sekunde aufruft, bis ein ZigBee Modul an den Computer angeschlossen wird, wird `getPortName()` erneut aufgerufen und führt dabei diesmal den `else`-Teil der Funktion aus. Anschließend wird das Aufrufintervall zurückgesetzt und somit vorerst ein erneuter Aufruf ausgeschlossen. Daraufhin wird dann “portName” bei der Erstellung des `SerialPort`-Objektes mitgegeben, womit dann die Grundlage für den Datenempfang gebildet wird (Abb. 18, Z.13). Für den Fall, dass dies erfolgreich ist, erkennt ein Event-Listener dies (Abb. 18, Z.14-16) und bestätigt den Erfolg durch eine Konsolenausgabe. Für den Fall, dass es fehlschlägt, erkennt dies ein weiterer Event-Listener (Abb. 18, Z.17-22) und gibt eine entsprechende Meldung in der Konsole aus und wartet auf einen erneuten Verbindungsversuch. Im Erfolgsfall wird dann des Weiteren die Funktion `listen()` aufgerufen (Abb. 18, Z.23), welche dann auf eingehende Daten wartet und diese dann dementsprechend verarbeitet.

Wie zuvor erwähnt, wird `getPortName()` immer wieder aufgerufen, bis ein ZigBee Modul erkannt und somit die Variable “portName” gefüllt wird. Dies kann unmittelbar nach Serverstart oder auch in Folge eines Aussteckens des Moduls der Fall sein. Für den Fall, dass eine bestehende Verbindung zwischen dem Modul und dem Computer getrennt wird, wurde in den `Else`-Teil von `getPortName()` noch ein weiterer Event-Listener integriert (Abb. 18, Z.24-31). Nach dem Einrichten des Datenempfangs durch `listen()`, sorgt die asynchrone Funktion dafür, dass während einer bestehenden Verbindung von Modul und Computer direkt auf den Fall eines Verbindungsabbruchs reagiert werden kann. Wenn vom Port ein “close” Event emittiert wird, das durch einen Verbindungsabbruch ausgelöst wurde, wird die Variable “portName” automatisch wieder geleert und das einsekündige Intervall tritt wieder in Kraft, wodurch wieder auf ein erneutes Verbinden des Moduls mit dem Computer gewartet wird.

### 5.15.3 Aufgetretene Probleme

Ein anfängliches Problem stellte der Absturz des Servers direkt nach seinem Start dar, wenn bei Serverstart kein Modul verbunden war. Der Grund dafür war, dass zu diesem Zeitpunkt der Name des COM-Ports hardcoded war und die Anwendung dabei versucht hat, einen Port anzusprechen, an dem kein Gerät verbunden ist. Dies führte dann zwangsläufig zum Absturz des Servers. Außerdem kam es bei einem Verbindungsabbruch zwischen Modul und Computer dazu, dass der Server zwar weiterlief, aber ein erneutes Verbinden des Moduls nicht zur erneuten Erkennung der Verbindung führte und auch kein Datenempfang im Backend stattfand.

Diese beiden Probleme konnten durch die Implementierung der automatischen COM-Port Erkennung in Form von `getPortName()` in Verbindung mit dem Timer gelöst werden.

Ein weiteres Problem, das nach der Lösung der beiden vorherigen auftrat, war, dass sich ein Serverabsturz provozieren ließ, wenn man das Modul vom Computer getrennt hat, während die Verbindung zum Modul initialisiert wurde und noch nicht abgeschlossen war. In der Praxis bedeutet das, dass man das Modul

```

1 function getPortName(){
2   if(portName == ""){
3     SerialPort.list().then(async(ports?: any, err?: any) =>{
4       for (const port of ports) {
5         if(port.manufacturer.includes('FTDI') && port.vendorId.includes('0403')){
6           portName = port.path
7           console.log("Port found:", portName)
8         }
9       }
10    })
11  }else{
12    clearInterval(waitForZigBee)
13    const serial = new SerialPort(connection, portName)
14    serial.port.on('open', async() =>{ // port opened successfully
15      console.log("Port opened!");
16    })
17    serial.port.on('error', async(err?: any) =>{ // port not opened successfully
18      console.log("An error has occurred --> " + err);
19      console.log("Please reconnect the module to your computer and don't remove it while the connection
      is being established.");
20      portName = ""
21      waitForZigBee = setInterval(getPortName, checkInterval)
22    })
23    serial.listen(io)
24    serial.port.on('close', async (err?: any) => { // device disconnected
25      console.log("Port closed.");
26      if (err.disconnected) {
27        console.log("Disconnected!");
28        portName = ""
29        waitForZigBee = setInterval(getPortName, checkInterval)
30      }
31    })
32  }
33 }

```

Abbildung 18: Funktion “getPortName()”

etwas mehr als eine Sekunde nachdem man es angesteckt hat, wieder absteckt. Der technische Hintergrund dabei ist, dass unsere Anwendung sobald ein ZigBee-Modul angesteckt wird, den richtigen Port erkennt und daraufhin dazu übergeht ein SerialPort-Objekt zu erstellen. Bei der Erstellung des SerialPort-Objektes wird dann versucht, den angegebenen Port zu öffnen. Dies ist aber nur möglich, wenn es ein angeschlossenes Gerät gibt, das auf diesen Port zugreift. Wenn wir nun, noch bevor das SerialPort-Objekt erstellt wird, die Verbindung zwischen Modul und Computer unterbrechen, kann der Port nicht geöffnet werden. Dieser Fehler löste den Serverabsturz aus.

Dieses Problem ließ sich durch den Einbau eines bestimmten Event-listeners (Abb. 18, Z.17-22), der bei der Erstellung eines SerialPort-Objektes darauf wartet, dass ein “error” Event emittiert wird, beheben. Bei der erfolgreichen Erstellung eines solchen Objektes wird nämlich ein “open” Event emittiert und bei einem Fehlschlag ein “error” Event. Wenn nun ein “error” Event emittiert wird, wird der Fehler in der Konsole geloggt und auf ein erneutes Anschließen des Moduls gewartet. Somit wird ein Absturz des Servers durch das abfangen des Fehlers verhindert und ein Warten auf eine erneute Verbindung durch wiederholtes Ausführen der getPortName() Funktion ermöglicht.

## 5.16 Installationsdatei

Alles bisherige wurde in Express.js gebündelt und in zwei PowerShell-Skripte unterteilt. Einmal wird das kompilierte Frontend mit Electron, und das kompilierte Backend in Express.js gestartet. Es konnte keine EXE.-Datei erstellt werden, da die Nodeversion vom Serialport.js, was verwendet wurde um die Kommunikation mit den COM-Port Schnittstellen herzustellen, und die serverseitige Nodeversion verschieden und daher nicht kompatibel waren. Auch Electron war von der Versionsinkompatibilität betroffen.

## 6 Auswertung des Testlaufs

Diese Messdaten wurden im Innen- und Außenbereich getestet. Die Temperatur, Luftfeuchtigkeit und CO2 Messwerte wurden mit einem Sensor der Firma NETATMO verglichen. In den folgenden Abbildungen werden drei Zeitpunkte markiert

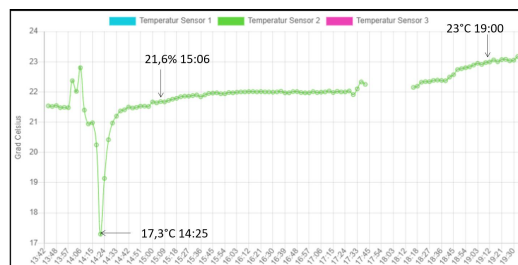


Abbildung 19: Funktion "getPortName()"

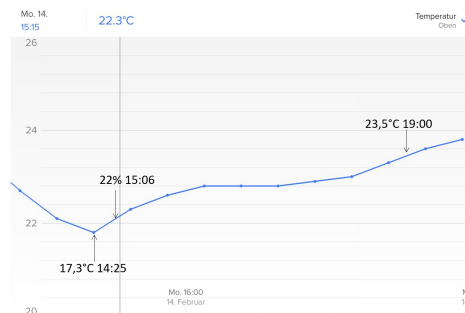


Abbildung 20: Funktion "getPortName()"

Die Verläufe der Graphen sind sehr ähnlich. Da der Netatmo von einem Gehäuse umschlossen ist werden vermutlich Messwertsprünge (z.b. beim Öffnen eines Fensters) verzögert gemessen und fallen deshalb nicht so stark aus wie bei den Sensoren am Endgerät.

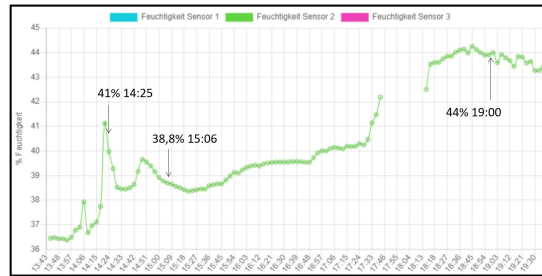


Abbildung 21: Funktion “getPortName()”

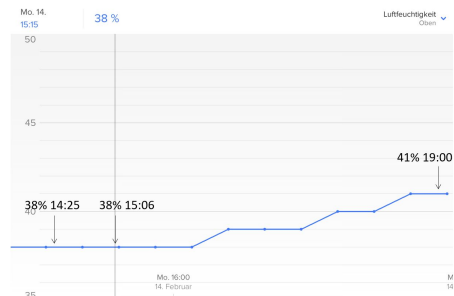


Abbildung 22: Funktion “getPortName()”

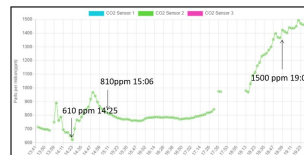


Abbildung 23: Funktion “getPortName()”



Abbildung 24: Funktion “getPortName()”

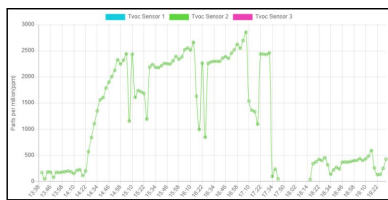


Abbildung 25: Funktion “getPortName()”

## **7 Ausblick und Erweiterungsmöglichkeiten**

### **7.1 Optimierung für Batterien**

Wie bereits im Abschnitt der Batterie erwähnt wurde, muss der Stromverbrauch des Endgerätes deutlich gesenkt werden. Dies kann geschehen durch das Herunterfahren der Sensoren nach einer Messung und einem größerem Messintervall. Außerdem könnte die Verwendung anderer Sensoren mit geringerem Verbrauch und niedrigeren Mindestbetriebsspannung die Laufzeit verlängern. Auch der Mikrocontroller verbraucht mit 15 mA (gemessen am Koordinator) für eine dauerhafte Verwendung an zwei AA Batterien zu viel.

### **7.2 Genauere Bestimmung der Werte des CCS811**

Wie bereits angesprochen ist der Sensor durch Einflüsse wie die Luftfeuchte beeinflussbar. Dies könnte man einschränken, indem man die gemessene Luftfeuchte der anderen Sensoren an den CCS811 überträgt und damit die Werte berichtigt. Selbiges gilt für die Temperatur.

### **7.3 Bugfixes in der Applikation**

In den Abschnitten zum Zeichentool und Dashboard wurden kleinere Bugs aufgeführt, welcher der Anwender berücksichtigen muss, damit die Anwendung ordentlich funktioniert. Aufgeführt wurden der zu schnelle Wechsel zwischen der Zeichentool und Dashboard sowie das Verhalten bei einem nicht beabsichtigten Anklicken der Linie. In Zukunft müsste vorerst darauf hingearbeitet werden, diese Bugs zu beheben.

### **7.4 Dynamische Gestaltung**

Zu diesem Zeitpunkt ist die Applikation nur auf die drei zur Verfügung gestellten Sensoren ausgelegt, dies bedeutet das keine weiteren hinzugefügt werden können. Daher könnte die Anwendung dynamisch ausgeweitet werden, um weitere Sensoren zuzulassen. Somit wäre es dem User selbst bestimmt, wie viele Messungen er tätigen könnte. Für jeden weiteren Sensor könne individuell ein Diagramm erstellt werden und diese könnten zusätzliche Optionen anbieten.

### **7.5 Migration auf andere Geräte oder Plattformen**

Zurzeit kann man die Anwendung lediglich auf dem Desktop verwenden. Eine Idee wäre es eine mobile Version für ein Handy zu entwickeln, somit kann man die Veränderung der Luftqualität überall in einer Wohnung oder einem Haus mitverfolgen.

## **7.6 Genauere Berechnung der Corona-Ampel**

Wie im bei dem Thema Luftqualität-CO<sub>2</sub> schon erwähnt, ist die CO<sub>2</sub> Konzentration nicht der alleinige Indikator für die Ansteckungsgefahr durch Corona. Für eine genauere Betrachtung wäre eine deutlich komplexere Berechnung der Ansteckungsgefahr nötig die auch andere äußerliche Faktoren miteinbezieht. Hierfür könnte man die Berechnung von Peng und Jimenez verwenden [14].

## **7.7 Dark Mode**

Im Pflichtenheft wurde außerdem noch erwähnt, dass bei genügend Zeit ein Dark Mode implementiert oder auch weitere Designs angeboten werden könnten.



## Literatur

- [1] *BMU Bericht 2005 - Verbesserung der Luftqualität in Innenräumen*. 11. Feb. 2022. URL: <https://www.umweltbundesamt.de/dokument/bmu-bericht-2005-verbesserung-der-luftqualitaet-in>.
- [2] Prof. Dr.-Ing Jörg Böttcher. *Sensoren für Temperatur, Feuchte und Gaskonzentrationen*. <https://messtechnik-und-sensorik.org/10-sensoren-fuer-temperatur-feuchte-und-gaskonzentrationen/>. [accessed 1.02.2022].
- [3] *Datasheet CCS811*. 23. Dez. 2016. URL: [https://cdn.sparkfun.com/assets/learn\\_tutorials/1/4/3/CCS811\\_Datasheet-DS000459.pdf](https://cdn.sparkfun.com/assets/learn_tutorials/1/4/3/CCS811_Datasheet-DS000459.pdf).
- [4] *Datasheet SHT21*. 1. Apr. 2014. URL: [https://sensirion.com/media/documents/120BBE4C/61642236/Sensirion\\_Humidity\\_Sensors\\_SHT21\\_Datasheet.pdf](https://sensirion.com/media/documents/120BBE4C/61642236/Sensirion_Humidity_Sensors_SHT21_Datasheet.pdf).
- [5] *Datasheet SCD41*. 1. Apr. 2021. URL: [https://sensirion.com/media/documents/C4B87CE6/61652F80/Sensirion\\_CO2\\_Sensors\\_SCD4x\\_Datasheet.pdf](https://sensirion.com/media/documents/C4B87CE6/61652F80/Sensirion_CO2_Sensors_SCD4x_Datasheet.pdf).
- [6] *Flüchtige organische Verbindungen*. 11. Feb. 2022. URL: <https://www.umweltbundesamt.de/themen/gesundheit/umwelteinfluesse-auf-den-menschen/chemische-stoffe/fluechtige-organische-verbindungen#welche-gesundheitlichen-wirkungen-konnen-voc-haben>.
- [7] *Gassensor*. 10. Feb. 2022. URL: <https://de.wikipedia.org/wiki/Gassensor>.
- [8] James Bryant Analog Devices Inc. *IC Temperature Sensors*. [https://archive.ph/20130827154740/http://www.analog.com/static/imported-files/rarely\\_asked\\_questions/moreInfo\\_raq\\_gapTempSensors.html](https://archive.ph/20130827154740/http://www.analog.com/static/imported-files/rarely_asked_questions/moreInfo_raq_gapTempSensors.html). [accessed 1.02.2022].
- [9] *Luftqualität und Gesundheit*. 11. Feb. 2022. URL: <https://www.inventer.de/wissen/luftqualitaet-gesundheit/>.
- [10] J.P. Peixoto und A.H. Oort. *Physics of Climate*. Springer, 1992.
- [11] *Richtig lüften in Schulen*. 22. Dez. 2021. URL: <https://www.umweltbundesamt.de/richtig-lueften-in-schulen#warum-ist-ein-regelmassiger-luftaustausch-in-klassenzimmern-grundsatzlich-wichtig-und-in-der-pandemie-umso-mehr>.
- [12] Umweltbundesamt. *Gesundheitliche Bewertung von Kohlenstoffdioxid in der Innenraumluft*. 2008.
- [13] Prof. Dr. Jürgen Wöllenstein. “Halbleiter-Gassensoren in Dünn- und Dick-schichttechnik”. In: (10. Feb. 2022).
- [14] Jose L. Jimenez Zhe Peng. *Exhaled CO2 as a COVID-19 Infection Risk Proxy for Different Indoor Environments and Activities*. 2021.