

Integrated Sensors

WS2022/23

Messsystem Spektral-Sensor

Yani Vutov

Benjamin Kammer

Dennis Spohrer

[GitHub](#)

[Zur Dokumentation](#)

Integrated Sensors

Einleitung - Vision - Motivation

Unsere Motivation ist es ein portables Messsystem zu bauen, welches überall genutzt werden kann, um Spektren von Objekten zu messen. Ableitend dazu ist das Ziel unseres Projekts, ein transportables, handliches (Stempel-) Messsystem mit dem Spektral-Sensor AS7265x zu erstellen, welches die gemessenen Daten in einer Datenbank speichert und grafisch visualisiert.

Mit diesem Messsystem soll es möglich sein unterwegs, beispielsweise im Baumarkt oder Supermarkt, verschiedene Produkte zu scannen. Aus den beim Scan-Vorgang erhaltenen Daten, sollen anschließend Aussagen über die Eigenschaften des Produktes gemacht werden.

Damit Eigenschaften zu den Produkten ermittelt werden können, muss zunächst eine Datenbank mit Messdaten von ausgewählten Produkten aufgebaut werden. Durch vergleichen der Datensätze sollen darauffolgend, die Aussagen über die Produkteigenschaften erzeugt werden.

In dieser Dokumentation werden zunächst allgemeine Informationen über die verwendeten Technologien/Hardware aufgezeigt. Dazu werden außerdem einige Basisinformationen wiedergegeben.

Im Anschluss werden die zwei Projektaufbauten näher beleuchtet. Der zweite Projektaufbau ist in 3 Stages unterteilt, welche jeweils einem Zwischenergebnis entspricht.

AS7265x

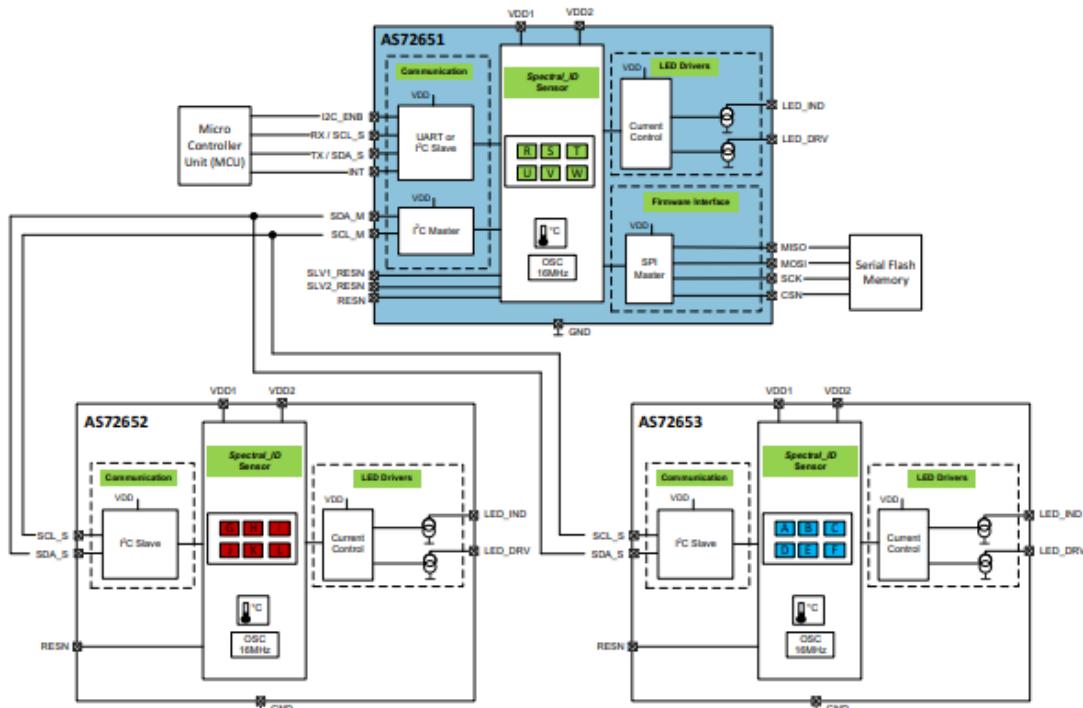
Aufbau des Sensors

Der Sensor besitzt drei separate Spektral-Sensorchips (AS72651, AS72652, AS72653). Jeder Sensorchip besitzt eine ansteuerbare LED, welche sich in den Zwischenräumen der einzelnen Chips befinden. Im Weiteren besitzt jeder Chip einen eingebauten Temperatursensor. Für eine akkurate Messung werden die LEDs benötigt, da diese das benötigte Lichtspektrum erzeugen. Bei diesem Vorgang wird das erzeugte Licht auf das Objekt gestrahlt und die reflektierte Strahlung gemessen. Dieses Prinzip wird "Reflektometrie" genannt. Jeder der drei Spektral-Sensorchips besitzt 6 Kanäle, welche einen Teil des Lichtspektrums abdecken. Zusammen decken sie das Spektrum des sichtbaren Lichtes ab, vom nahem infrarot Licht bis hin zu ultraviolettem Licht. Die Sensor-Chips kommunizieren untereinander über das I²C Protokoll. Der erste der drei Sensor-Chips (AS72651) ist dabei der Master und die anderen beiden (AS72652, AS72653) sind die Slaves des I²C Busses.

Alle drei Sensor-Chips besitzen virtuelle Adressen ihrer Kanäle, die mit I²C angesteuert werden können. Im Kapitel [Stage1](#) wird mit Hilfe von Code näher darauf eingegangen. Die externe Kommunikation läuft dabei ausschließlich über den Master Chip AS72651, welcher Informationen der Slaves nach Bedarf bereitstellt. Es ist nicht bekannt, wie die Bereitstellung der Daten intern über I²C funktioniert.

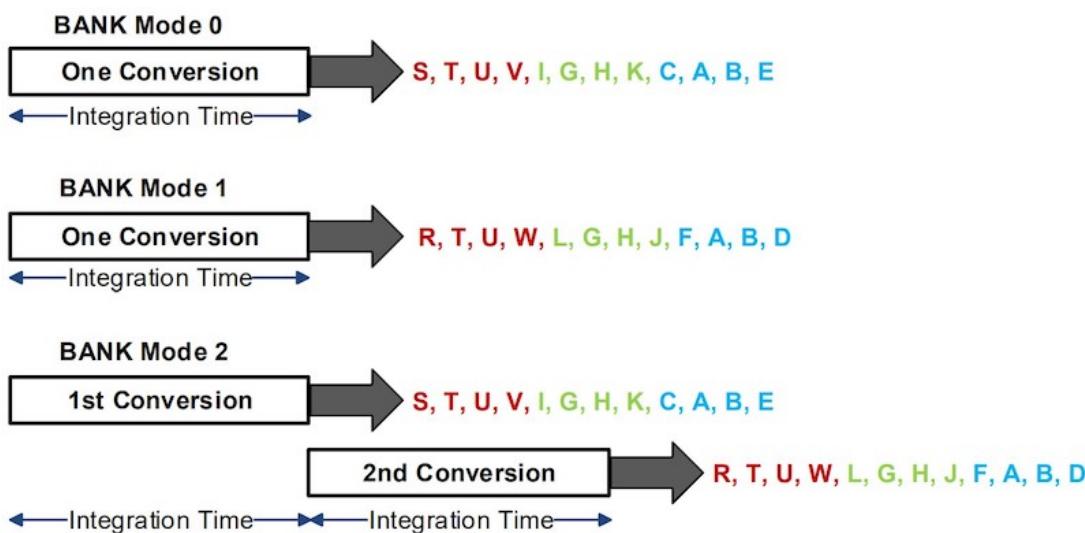
Betrieben wird der Sensor mit 3,3 Volt.

- Datenblatt: [AS7265x](#)
- Hersteller Website: [SparkFun Triad Spectroscopy Sensor - AS7265x \(Qwiic\)](#)



Bank Modes

Jeder Chip besitzt zwei Photodioden-Banken, welche die gemessenen Werte beinhalten. Nachdem eine Messung durch das Konfigurationsregister ausgelöst wird, werden die dazugehörigen Werte in den Photodioden-Banken bereit gestellt. Dafür werden die Werte aus den Photodioden-Banken mit A/D-Wandler umgewandelt und in die korrespondierenden Kanäle/Register gespeichert. Dabei können vier Kanäle auf einmal integriert werden, weshalb bei einer vollen Messung von sechs Kanälen zwei Integrationszyklen notwendig sind.



Welcher Wert für den Integrationszyklus benutzt wird und wie die Berechnung dafür aussieht, wird im Rahmen des Projektes nicht weiter verfolgt. Im Rahmen dieses Projektes wird der vordefinierte Wert aus der Bibliothek verwendet. Der genaue Wert kann aus dem Datenblatt entnommen werden.

Je nachdem welcher Modus im Konfigurationsregister gesetzt ist, stehen unterschiedliche Sensorwerte zur Verfügung. Dabei sind zwischen drei Modi zu unterscheiden, welche jeweils ein anderes Bit-Muster initiieren.

3:2	BANK	10	R/W	Measurement mode: b00=Mode 0: 4 channels b01=Mode 1: 4 channels b10=Mode 2: All 6 channels b11=Mode 3: One-Shot operation of mode 2
-----	------	----	-----	---

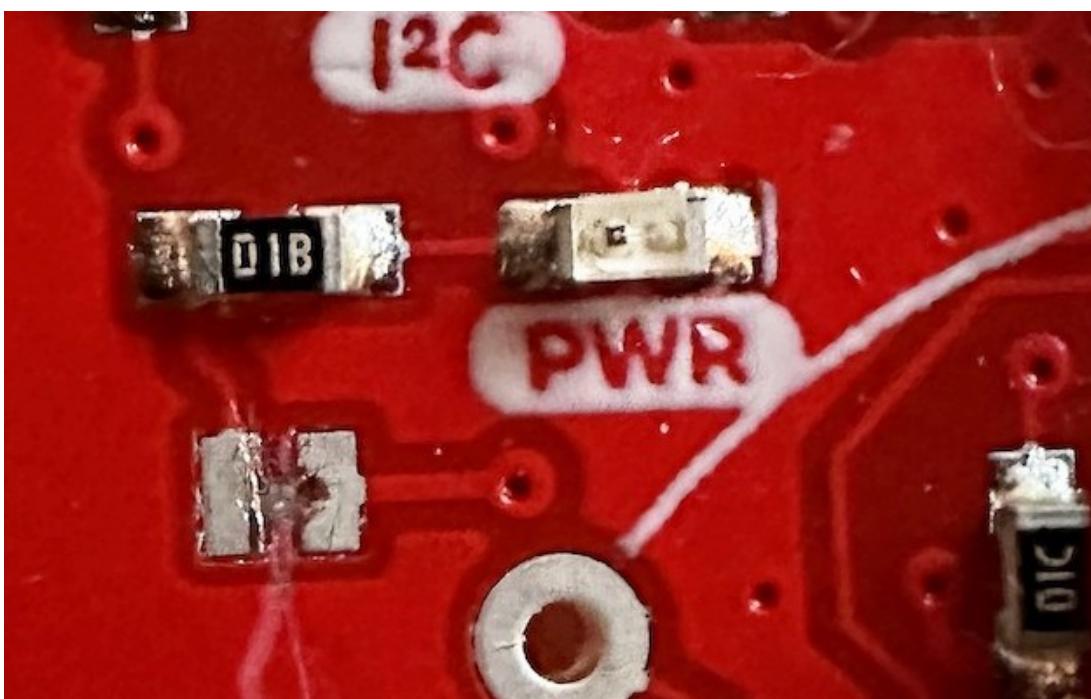
Für die Bit-Muster **0b00** und **0b01** wird der selbe Modus initiiert. Für die komplette Messung aller 18 Kanäle muss der letzte Modus gewählt werden.

Wie genau die Messung erfolgt und wie die Werte in die jeweiligen Photodioden-

Banken gespeichert werden, konnte aus dem Datenblatt nicht entnommen werden.

LED Control

Jeder Chip besitzt zwei integrierte LED Treiber, mit programmierbarer Stromstärke. Für akkurate Messungen haben wir die beiden LED's, Status und PWR (Power), ausgeschaltet. Diese würden ansonsten zu weiteren Reflektionen führen, welche die gemessenen Werte verfälschen würden. Die Status LED konnte im Code, durch das aufrufen der Methode `disableIndicator()` ausgeschaltet werden. Für die PWR LED wurde der Jumper, welcher sich links neben der LED befindet, durchtrennt.



Im Projekt wird nicht näher auf die Kontrolle der LED's eingegangen, sondern lediglich die Default-Optionen der Bibliothek verwendet, da diese nicht relevant waren.

Was ist InfluxDB?

InfluxDB ist ein Open-Source-Datenbankmanagementsystem, das von der Firma InfluxData entwickelt wurde und speziell für die Speicherung von Zeitreihen gedacht ist. Dazu gehören APIs zum Speichern und Abfragen von Daten, deren Verarbeitung im Hintergrund für Überwachungs- und Warnzwecke, Benutzer-Dashboards sowie die Visualisierung und Untersuchung der Daten. Um mit dem Datenbankmanagementsystem zu interagieren besitzt es zwei Abfragesprachen. Eine

funktionale Skriptsprache Flux und eine SQL ähnliche Sprache InfluxQL, welche, wie die Weboberfläche, auf Port 8086 angesprochen wird.

Was ist Grafana?

Grafana ist eine Open-Source-Anwendung für Analysen und interaktive Visualisierung. Es bietet Diagramme, Grafiken und Warnungen, wenn es mit unterstützten Datenquellen verbunden ist. Mithilfe von Plugins können die Anzeigemöglichkeiten auf Dashboards und Datenquellen erweitert werden. Dashboards können mit interaktiven Abfragegeneratoren erstellt werden. "Grafana wird häufig für Überwachungsanwendungen verwendet und unterstützt die drei Säulen der Observability: Metriken, Logging und Tracing, kann aber auch zur Darstellung von statischen Daten in relationalen Datenbanken genutzt werden." Quelle [Wikipedia](#) (13.1.23)

NEXT >

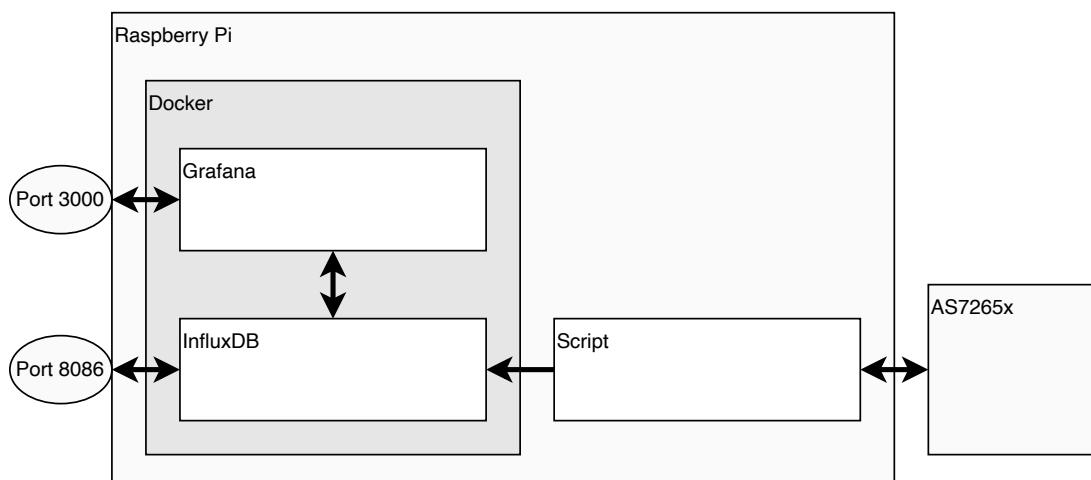
Überblick

Projektaufbau 1

Umsetzungsidee

Die Idee der Umsetzung des Messsystems besteht zu Beginn aus dem Spektral-Sensor AS7265x und einem Raspberry Pi 2. Der Raspberry Pi 2 kommt aus dem Grund zum Einsatz, da dieser vorhanden ist und nicht bestellt werden muss. Der Sensor wird dabei direkt an die I²C Pins der GPIO Schnittstelle des Raspberry Pis (RPi) angeschlossen.

Auf dem Raspberry Pi wird zuerst Docker installiert, mit dessen Hilfe dann ein InfluxDB Container zur Datenspeicherung und ein Grafana Container zur Datenvisualisierung gestartet wird. Mit einem Python Skript, dass direkt (ohne Docker) auf dem RPi laufen soll, soll die Messung auf dem Spektral-Sensor gestartet und die gemessenen Daten in die InfluxDB gespeichert werden.



In den folgenden Abschnitten wird nur auf die wichtigsten Alleinstellungsmerkmale des ersten Prototyps eingegangen, um Redundanz mit dem zweiten Prototyp zu vermeiden. Ausführlichere Beschreibungen zu folgenden Punkten finden sich beim 2. Prototyp wieder.

Verwendete Komponenten

Hardware Materialien:

- [SparkFun Triad Spectroscopy Sensor - AS7265x \(Qwiic\)](#)
- [Raspberry Pi 2 Model B](#)

Verwendete Software:

- Raspberry Pi
 - [Wiring Pi](#)
 - [Docker](#)
 - [InfluxDB](#)
 - [Grafana](#)

Weitere Komponenten:

- 3D-Drucker (Filament)
- [Fusion360](#)

Prototyp 1 (Raspberry Pi)

Anschließen des Spektral-Sensors an den Raspberry Pi 2

Durch die GPIO Pins des Raspberry Pis, die auf 3,3 Volt arbeiten, ist es möglich den Spektral-Sensor direkt ohne externe Spannungsquelle und Logic-Converter an den RPi anzuschließen.

Um überprüfen zu können, ob dieser auch korrekt angeschlossen ist werden die I2C-Tools mit dem Befehl `apt-get install i2c-tools` installiert. I2C-Tools beinhalten vier Befehle von dem **i2cdetect** ein Programm ist. Dieses kann I2C-Busse nach Geräten scannen. Mit dem Befehl `i2cdetect -y 1` wurde die Busadresse des Spektral-Sensors erfolgreich gefunden.

```
[user@raspberrypi:~ $ i2cdetect -y 1
      0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
00:          -- -- -- -- -- -- -- -- -- -- -- -- -- --
```

```
10: ---  
20: ---  
30: ---  
40: --- 49 ---  
50: ---  
60: ---  
70: ---  
[user@raspberrypi:~ $
```

Damit das Python-Skript mit dem Spektral-Sensor kommunizieren kann, importiert das Skript die Bibliothek Wiring Pi. Zur Zeitersparnis wird getestet, ob das Skript der [Bachelor Thesis](#) auf diese Anwendung übertragbar ist. Bei dem RPi 4 ist Wiring Pi, nach Aussage der Bachelor Thesis ([SpectralSensor](#)), bereits vorinstalliert, bei dem verwendeten RPi 2 ist das jedoch nicht der Fall. Die Bibliothek Wiring Pi ist schon etwas älter, sodass erst eine passende Version der Bibliothek gefunden werden muss. Installieren lässt dich die Bibliothek durch folgenden Befehl:

```
wget https://project-downloads.drogon.net/wiringpi-latest.deb  
sudo dpkg -i wiringpi-latest.deb
```

Es gibt inzwischen bessere Bibliotheken, um die GPIOs des RPis zu steuern, sodass in Erwgung gezogen werden sollte die Bibliothek Wiring Pi auszutauschen. Das verwenden einer anderen Bibliothek wird sogar von Wiring Pi (siehe git.drogon.net) empfohlen.

Das Skript der Bachelor Thesis konnte ausgeführt werden, jedoch kann die vorhergesehene Arbeitsweise des Programmes nicht überprüft werden. Die Messung scheint ausgeführt zu werden, aber wohin die Daten gespeichert werden ist nicht bekannt. Die Vermutung belief sich darauf, dass keine Datenbank für die Datenspeicherung angegeben wurde, sodass der weitere Schritt sich auf das Aufsetzen der Datenbank ausrichtet.

Datenspeicher- und Visualisierungssystem aufsetzen

Im zweiten Schritt wird geprüft, ob die Datenbank InfluxDB und Grafana mit Docker auf dem RPi 2 betrieben werden kann.

Die Installationsschritte sind im folgenden Shell Skript (`install_docker.sh`) zusammengefasst:

```
sh

#!/bin/sh

# Uninstall old versions
sudo apt-get remove docker docker-engine docker.io containerd runc

# Update the apt package index and install packages to allow apt to
# easily pull from https://
sudo apt-get update
sudo apt-get install ca-certificates curl gnupg lsb-release

# Add Docker's official GPG key:
curl -fsSL https://download.docker.com/linux/debian/gpg | sudo gpg --dearmor -o /usr/share/keyrings/docker-archive-keyring.gpg

# Set up the stable repository.
echo "deb [arch=$(dpkg --print-architecture) signed-by=/usr/share/keyrings/docker-archive-keyring.gpg] https://download.docker.com/linux/debian $(lsb_release -cs) stable" | sudo tee /etc/apt/sources.list.d/docker.list >> /dev/null

# Update the apt package index, and install the latest version of Docker CE
sudo apt-get update
sudo apt-get install docker-ce docker-ce-cli containerd.io docker-buildx-plugin docker-compose-plugin
```

Github: [install_docker.sh](#)

Damit der Besitzer des Skriptes dieses ausführen kann, muss die entsprechende Ausführ-Berechtigung (`chmod u+x install_docker.sh`) gesetzt werden. Anschließend kann mit dem Shell-Befehl `./install_docker.sh` das Skript ausgeführt werden.

Nach der Installation von Docker wird eine Docker Compose Datei erstellt, um InfluxDB und Grafana aufzusetzen. Dabei ist es nicht möglich die neuste Version von InfluxDB zu nehmen, da diese nicht auf dem RPi 2 startet. Denn die neuste Version von InfluxDB läuft nur auf einem 64-Bit System. Jedoch handelt es sich bei dem Prozessor des Raspberry Pi 2 um einen 32-Bit Architektur.

Um dies zu umgehen, setzen wir auf die aktuellste Version von InfluxDB, die noch 32-

Bit unterstützt, welche Version 1.8 ist. Die hierfür verwendete `docker-compose.yml` sieht damit wie folgt aus:

```
yaml

services:
  influxdb:
    image: influxdb:1.8 # 1.8 is the latest version which is run
    ports:
      - '8086:8086'
    volumes:
      - influxdb-storage:/var/lib/influxdb
    environment:
      - INFLUXDB_DB=${INFLUXDB_DATABASE}
      - INFLUXDB_ADMIN_USER=${INFLUXDB_USERNAME}
      - INFLUXDB_ADMIN_PASSWORD=${INFLUXDB_PASSWORD}
  grafana:
    image: grafana/grafana
    ports:
      - '3000:3000'
    volumes:
      - ./grafana-storage:/var/lib/grafana # To keep grafana con
      - ./grafana-provisioning/:/etc/grafana/provisioning
    depends_on:
      - influxdb
    environment:
      - GF_SECURITY_ADMIN_USER=${GRAFANA_USERNAME}
      - GF_SECURITY_ADMIN_PASSWORD=${GRAFANA_PASSWORD}

volumes:
  influxdb-storage:
```

Konzeptänderung

Beim Einrichten der InfluxDB als Datenquelle in Grafana, wurde festgestellt, dass diese Datenbankversion (InfluxDB 1.8) nicht mehr in Grafana unterstützt wird. Ein möglicher Lösungsansatz könnte dabei sein, auf eine ältere Version von Grafana zurückzugreifen. Dabei stellt sich die Frage wie sinnvoll dieser Lösungsansatz ist, weil die ältere

Hardware und Software nicht mehr maintained bzw. aktualisiert wird. Eine weitere Lösung verfolgt einen anderen Ansatz. Diese zieht eine Änderung in der Hardware auf sich. Anstelle des RPi 2 könnte ein RPi 4 zum Einsatz kommen dessen Prozessor auf einer 64-Bit Architektur setzt. Damit kann die neuste Version von InfluxDB und Grafana verwendet werden.

Hardwaretechnisch kann auch ein anderer Lösungsansatz in Betracht gezogen werden. Der Spektral-Sensor könnte auch mit einem Mikrocontroller, wie den ESP32, betrieben werden. Dieser besitzt bereits, im Gegensatz zu dem RPi 2, eine integrierte WiFi-Schnittstelle. Diese kann genutzt werden, um die Daten des Sensors an die Datenbank zu senden. Weitere Vorteile des ESP32 beziehen sich auf die Arduino Entwicklungsumgebung, mit der sich der ESP32 programmieren lässt. Dieser Vorteil kommt dadurch zum Tragen, da es für die Arduino Plattform bereits eine Bibliothek für den Spektral-Sensor sowie InfluxDB gibt. Ein weiterer Vorteil ist, dass der Aufbau mit dem Sensor und einem ESP32 wesentlich handlicher und portabler als mit einem RPi ist.

Jedoch lässt sich InfluxDB und Grafana nicht auf einem ESP32 betreiben, sodass hier eine Alternative gefunden werden muss. Hier kann wieder ein RPi zum Einsatz kommen. Bei dieser Lösung besteht das Problem, dass dieser sich immer im gleichen Netzwerk wie der ESP befinden muss, um für den ESP erreichbar zu sein. Dadurch müssten beide Systeme unterwegs mitgenommen werden, was nicht praktikabel ist.

Lösung hier ist es auf eine Cloud-Instanz zu setzen, die von "überall" erreichbar ist. Damit können auch mehrere der Messsysteme auf die gleiche Cloud-Datenbank zurückgreifen, welches die Herausforderung des Datenaustausches unter den Systemen vereinfacht. Damit kann hier auf eine Datenquelle zurückgegriffen werden, um zukünftig Rückschlüsse auf Eigenschaften von gemessenen Produkten aufzubauen. Ebenfalls kann auch über jedes (mobiles) Endgerät auf die visualisierten Daten in Grafana zurückgegriffen werden.

◀ PREVIOUS

Home

NEXT ▶

Überblick

Projektaufbau 2

Umsetzungsidee

Der neue Projektaufbau besteht aus dem Spektral-Sensor, der an den ESP32 angeschlossen ist. Dieser führt auf Knopfdruck eine Messung aus und sendet die Daten an eine in der Cloud gehosteten Datenbank. Grafana läuft ebenfalls auf dieser Cloud Instanz, um von jedem Endgerät erreichbar zu sein. Zudem besitzt das Messsystem für die Mobilität ein Batteriepack und einen Buzzer für eine akustische Rückmeldung bei Messungen.

Verwendete Komponenten

Die hierfür verwendeten Hardware Materialien belaufen sich auf:

- [SparkFun Triad Spectroscopy Sensor - AS7265x \(Qwiic\)](#)
- [ESP32 NODEMCU Module](#)
- [Lademodul / Entlademodul - HW-107](#)
- [KY-012 Buzzer Modul aktiv](#)
- [SX1308 Step-Up Converter Regulator](#)
- [Li-Ion Akku Lithium Ionen Batterie](#)
- [Single Battery holder](#)

Verwendete Software

- Auf dem ESP32
 - [AS7265x Bibliothek](#)
 - [InfluxDB Bibliothek](#)
- In der Cloud
 - [Docker](#)
 - [InfluxDB](#)
 - [Grafana](#)

Im Verlauf des Projektes werden noch Zugriff auf andere Komponenten gebraucht:

- 3D-Drucker (Filament)
- Server zum hosten der Datenbank und Grafana.
- [Fritzing](#)
- [Fusion360](#)

Für Studenten besteht die Möglichkeit in der BwCloud eine virtuelle Instanz zu erstellen.

◀ PREVIOUS

Überblick

NEXT ▶

Stage 1

Stage 1 - Testen der Konzeptidee

Im ersten Schritt wird überprüft, ob mit dem ESP und den Bibliotheken eine Messung auf dem Spektral-Sensor ausgeführt werden kann und diese gemessenen Daten in der Cloud InfluxDB gespeichert werden können.

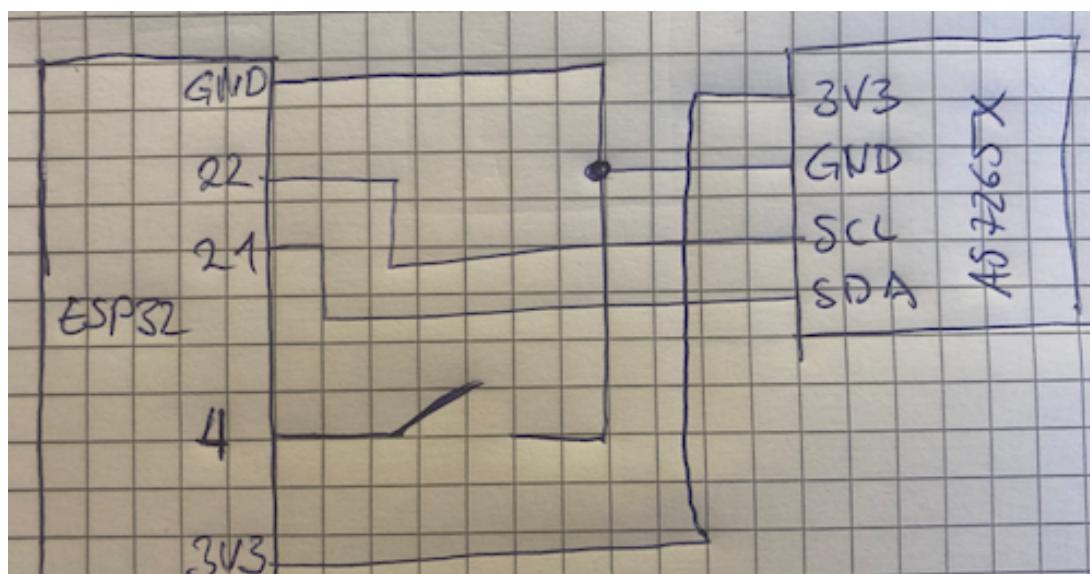
ESP32

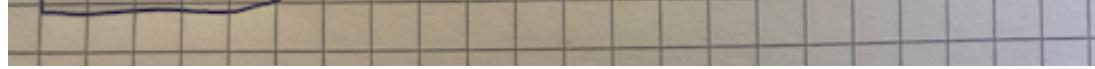
Generelle Informationen

Gründe für die Verwendung eines [ESP32](#) zur Ansteuerung des Spektral-Sensors sind:

- Integrierter WiFi-Chip
- Erfahrung in der WiFi-Kommunikation
- Handliche Form für die Umsetzung der Stempelidee (Vergleich: RPi sehr unhandlich)
- Schon vorhandene und getestete Bibliotheken für die InfluxDB-Kommunikation und den Spektral-Sensor

Anschließen des Sensors





Code

Im Folgenden wird der Code für die verschiedenen Stages dargestellt. Für Stage 1 und Stage 2 werden Durchführungen und Ergebnisse kurz erwähnt. In Stage 3 wird der finale Code näher beleuchtet.

Zunächst wird die verwendete Entwicklungsumgebung näher erläutert, sowie die verwendeten Bibliotheken und deren Funktionsweisen. Im Projekt wird Plattform IO als IDE (Integrated Development Environment) verwendet.

Dies ermöglicht es, einfach cpp Bibliotheken zu installieren und gleichzeitig im Arduino Framework zu bleiben. Dadurch können schnell erste Versuche mit dem Sensor durchgeführt werden.

Die Code Konzeption für den ersten Stage ist übersichtlich. Zunächst werden folgende grundlegende Vorgehen geprüft:

- Messung durch Knopf-Druck durchführen.
- Daten im ESP32 sammeln, durch Auslesen der Sensordaten
- Daten in die Cloud Influxdb senden.
- Daten in der Influxdb zur Überprüfung darstellen.

Wie schon erwähnt wurden hierfür zwei Bibliotheken verwendet.

Bibliothek zum Auslesen des AS7265X

Die von uns für den ESP32 zum Auslesen der Sensordaten verwendete Bibliothek ist die [SparkFun_AS7265x_Arduino_Library](#).

Zunächst werden Komponenten zum Auslesen des Sensors dargestellt. Die einzelnen Funktionen werden aufgezählt und deren Funktionsweise erklärt. Die genaue Umsetzung im Code wird in Stage 3 erklärt.

Objektinitialisierung und begin() Funktion

Damit Werte vom Lichtspektrum ausgelesen werden können, muss zunächst eine Objektinstanz der Klasse deklariert und definiert werden.

cpp

```
//Instanziierung und Aufrufen der Funktion begin()
AS7265x specSensor;
specSensor.begin();
```

Beim Ausführen der Funktion **begin()**, wird die TwoWire Bibliothek initialisiert. Diese Methode hat in der Definition den Pointer für das TwoWire Objekt. Dieses wird hier explizit weggelassen, da es in der Header-File aus der Wire.cpp als Parameter bei der Deklaration mitgegeben wird.

cpp

```
//Deklaration der Funktion begin() in der SparkFun_AS7265X.h
boolean begin(TwoWire &wirePort = Wire);
```

Zu Beginn der Methode, wird zunächst das I²C Protokoll initiiert. Im Anschluss wird geprüft, ob der Sensor erfolgreich mit dem ESP32 verbunden ist (**isConnected()**). Danach wird überprüft, ob die beiden Slaves bereit sind, um mit ihnen zu kommunizieren. Dazu wird die **virtualReadRegister()** Methode benutzt und das Register **0x4F** ausgelesen, welches diese Information besitzt. Dabei steht das fünfte Bit jeweils für den ersten und zweiten Slave, wobei dieses nicht gesetzt sein darf.

Addr: 0x4F		DEV SEL		
Bit	Bit Name	Default	Access	Bit Description
5	SECOND SLAVE	0	R	Second slave Available
5	FIRST SLAVE	0	R	First slave available

Im nächsten Schritt, werden die einzelnen LED Stromstärken gesetzt. Dazu wird in die jeweiligen Register für die weiße, infrarot und ultraviolet LED eine 1 gesetzt. Das Limit wird hier jeweils auf 12,5mA gesetzt.

cpp

```
setBulbCurrent(AS7265X_LED_CURRENT_LIMIT_12_5MA, AS7265x_LED_WHITE)
setBulbCurrent(AS7265X_LED_CURRENT_LIMIT_12_5MA, AS7265x_LED_IR)
setBulbCurrent(AS7265X_LED_CURRENT_LIMIT_12_5MA, AS7265x_LED_UV)
```

Danach werden die jeweiligen LEDs explizit ausgeschaltet, um den Sensor vor Überhitzung zu schützen.

cpp

```
disableBulb(AS7265x_LED_WHITE); //Turn off bulb to avoid heating
disableBulb(AS7265x_LED_IR);
disableBulb(AS7265x_LED_UV);
```

Im Anschluss wird der Strom für die Status-LED auf 8mA gesetzt und eingeschaltet.

cpp

```
setIndicatorCurrent(AS7265X_INDICATOR_CURRENT_LIMIT_8MA); //Set
enableIndicator();
```

Die Status LED wurde im Code explizit ausgeschaltet. Zum Ausschalten der PWR (Power) LED muss ein Jumper durchtrennt werden.

Daraufhin wird der Measurement-Modus gesetzt. Damit können alle Kanäle auf einmal gelesen werden, wenn eine Messung initiiert wird.

Zum Schluss werden noch die Interrupts aktiviert.

Für die letzten Schritte wird ein spezifisches Konfigurationsregister verwendet (siehe Bild unten). Die einzelnen Bits korrespondieren jeweils mit den unterschiedlichen Aufgaben, die aus den Namen des Bildes unten entnommen werden können.

Addr: 0x04		Configuration		
Bit	Bit Name	Default	Access	Bit Description
7:0	SRST	0	W	[W] software reset [R] gain error
6	INT	0	R/W	Enable interrupt pin
5:4	GAIN	01	R/W	Gain configuration: b00=1x; b01=3.7x; b10=16x; b11=64x
3:2	BANK	10	R/W	Measurement mode: b00=Mode 0: 4 channels b01=Mode 1: 4 channels b10=Mode 2: All 6 channels b11=Mode 3: One-Shot operation of mode 2
1	DATA_RDY	0	R	Data ready to read
0	FRST	0	W	Factory reset

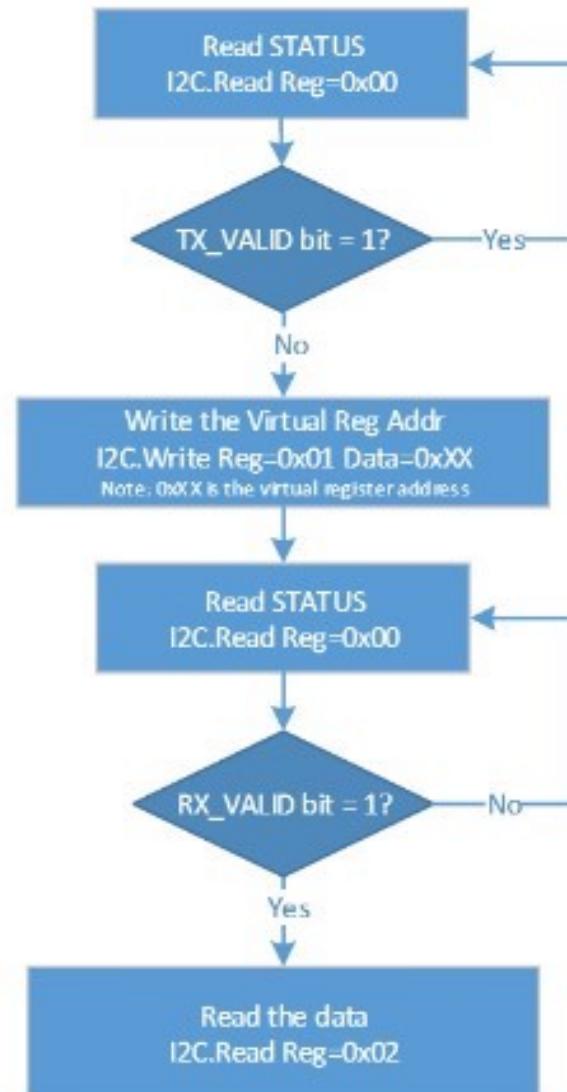
Funktionen zum auslesen der 18 Kanäle

Zum Auslesen der Sensordaten, besitzt die Bibliothek verschiedene Funktionen.

Zunächst werden Funktionen erklärt, die über I²C die jeweiligen virtuellen Register auslesen.

Funktion: virtualReadRegister

Flow Chart for Virtual Register Read



Zunächst wird das Statusregister, unter der Adresse **0x00**, ausgelesen. Mit einer Bitmaske wird anschließend geprüft, ob in dem Leseregister noch Daten zu lesen sind. Falls Daten zu lesen sind, wird das Leseregister ausgelesen. Diese gelesenen Daten werden jedoch nicht weiter betrachtet und somit verworfen.

cpp

```
//Read a virtual register from the AS7265x
uint8_t AS7265X::virtualReadRegister(uint8_t virtualAddr)
{
    uint8_t status;

    //Do a prelim check of the read register
    status = readRegister(AS7265X_STATUS_REG);
    if ((status & AS7265X_RX_VALID) != 0) //There is data to be re
    {
        readRegister(AS7265X_READ_REG); //Read the byte but do nothi
    }
}
```

Im Anschluss wird das Statusregister wieder ausgelesen um zu überprüfen, ob in das Schreibregister geschrieben werden kann. Kann nicht ins Schreibregister geschrieben werden, wird ein kurzes Delay ausgeführt und der Vorgang solange wiederholt, bis in dieses geschrieben werden kann.

cpp

```
//Wait for WRITE flag to clear
while (1)
{
    status = readRegister(AS7265X_STATUS_REG);
    if ((status & AS7265X_TX_VALID) == 0)
        break; // If TX bit is clear, it is ok to write
    delay(AS7265X_POLLING_DELAY);
}
```

Nun kann die Adresse der entsprechenden Daten in das Schreibregister geschrieben werden. Das 7. Bit in diesem "Adress-Byte" ist hier 0 um dem System zu sagen, dass es sich hier um einen Lesevorgang handelt.

cpp

```
// Send the virtual register address (bit 7 should be 0 to indic
writeRegister(AS7265X_WRITE_REG, virtualAddr);
```

Im Anschluss wird das Statusregister solange ausgelesen, bis in diesem das Leseregister Bit gesetzt ist. Damit wird signalisiert, dass die angefragten Daten nun im Leseregister vorhanden sind.

cpp

```
//Wait for READ flag to be set
while (1)
{
    status = readRegister(AS7265X_STATUS_REG);
    if ((status & AS7265X_RX_VALID) != 0)
        break; // Read data is ready.
    delay(AS7265X_POLLING_DELAY);
}
```

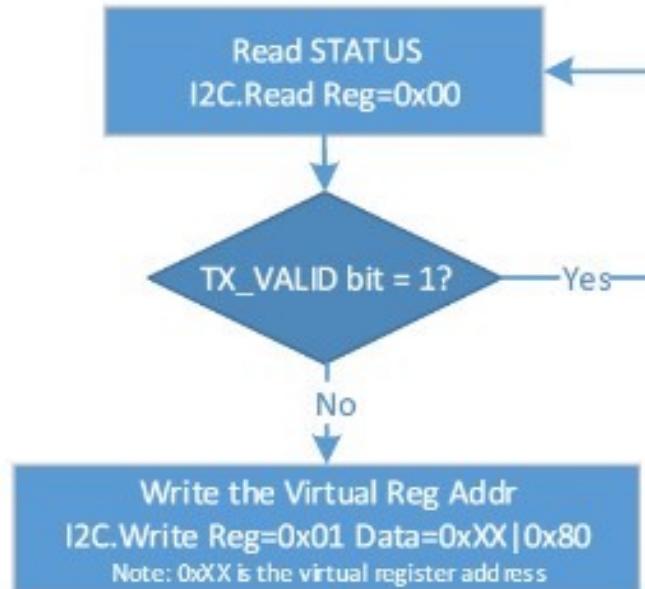
Wenn das Bit gesetzt ist, lese aus dem Leseregister die Daten heraus.

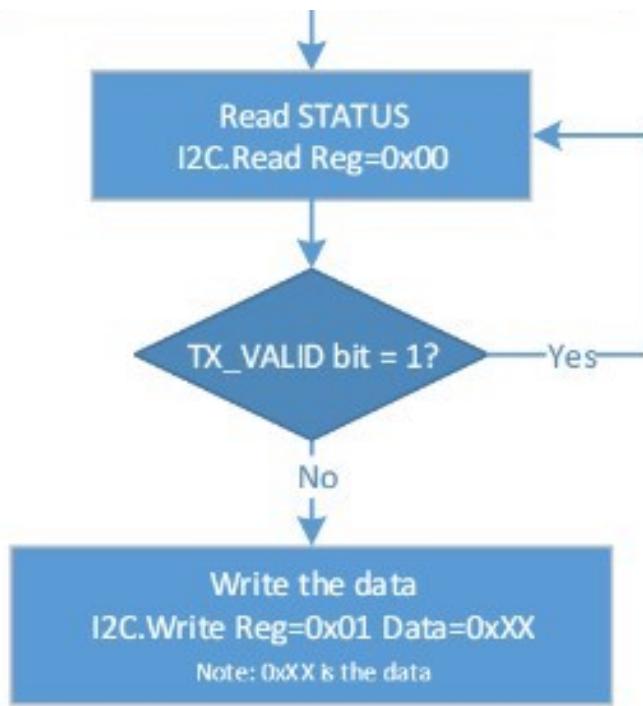
cpp

```
uint8_t incoming = readRegister(AS7265X_READ_REG);
return (incoming);
```

Funktion: virtualWriteRegister

Flow Chart for Virtual Register Write





Um in ein Register zu schreiben wird zunächst geprüft, ob im Statusregister das Bit für das Schreibregister gesetzt ist. Falls nicht, wird gewartet bis dieses nicht mehr gesetzt ist.

Anschließend wird in das Schreibregister die virtuelle Adresse geschrieben. Zusätzlich wird an der Stelle des 7. Bit eine 1 geschrieben, um dem Sensor zu sagen, dass es sich hier um einen Schreibvorgang handelt.

cpp

```
// Send the virtual register address (setting bit 7 to indicate
writeRegister(AS7265X_WRITE_REG, (virtualAddr | 1 << 7));
```

Anschließend wird wieder gewartet bis das Schreibregister keinen gesetzten Bit mehr hat. Wenn das Bit nicht mehr gesetzt ist, dann kann in das Schreibregister die Register-Daten geschrieben werden.

Funktion: readRegister

`readRegister()` ist eine Funktion, die die Register der Sensor-Chips über I²C ausliest.

Die TwoWire Bibliothek bedient sich dabei der Abfolge der Schritte des I²C Protokolls. Zunächst wird die Übertragung mit dem AS7265X initiiert, indem die Funktion

beginTransmission() aufgerufen wird. Im Anschluss wird das zu lesende Register gesendet. Dann wird gewartet, bis die zu lesenden Daten ausgelesen werden können. Zum Schluss werden die Daten ausgelesen.

cpp

```
//Reads from a give location from the AS726x
uint8_t AS7265X::readRegister(uint8_t addr)
{
    _i2cPort->beginTransmission(AS7265X_ADDR);
    _i2cPort->write(addr);
    if (_i2cPort->endTransmission() != 0)
    {
        //Serial.println("No ack!");
        return (0); //Device failed to ack
    }

    _i2cPort->requestFrom((uint8_t)AS7265X_ADDR, (uint8_t)1);
    if (_i2cPort->available())
    {
        return (_i2cPort->read());
    }

    //Serial.println("No ack!");
    return (0); //Device failed to respond
}
```

Funktion: writeRegister

Die **writeRegister()**-Funktion funktioniert so: Zunächst wird wieder die Funktion **beginTransmission()** aufgerufen, um mit dem AS7265X zu kommunizieren. Im Anschluss wird die Adresse gesendet, wohin die Daten schlussendlich gesendet werden sollen. Danach werden die Daten gesendet. Die Verbindung wird mit dem AS7265x geschlossen.

cpp

```
//Write a value to a spot in the AS726x
boolean AS7265X::writeRegister(uint8_t addr, uint8_t val)
```

```

{
    _i2cPort->beginTransmission(AS7265X_ADDR);
    _i2cPort->write(addr);
    _i2cPort->write(val);
    if (_i2cPort->endTransmission() != 0)
    {
        //Serial.println("No ack!");
        return (false); //Device failed to ack
    }

    return (true);
}

```

Funktion: takeMeasurementsWithBulb

Diese Funktion wird benutzt, um erst alle 3 LEDs einzuschalten und danach die 18 Kanäle auszulesen. Zum Schluss werden die einzelnen LEDs wieder ausgeschaltet.

cpp

```

//Turns on all bulbs, takes measurements of all channels, turns off all bulbs
void AS7265X::takeMeasurementsWithBulb()
{
    enableBulb(AS7265x_LED_WHITE);
    enableBulb(AS7265x_LED_IR);
    enableBulb(AS7265x_LED_UV);

    takeMeasurements();

    disableBulb(AS7265x_LED_WHITE); //Turn off bulb to avoid heating
    disableBulb(AS7265x_LED_IR);
    disableBulb(AS7265x_LED_UV);
}

```

Um die LEDs jeweils ein und wieder aus zuzuschalten, wird sich dem LED_CONFIG Register bedient.

Addr: 0x07	LED Configuration
------------	-------------------

Bit	Bit Name	Default	Access	Bit Description
7	READ_ERR	0	R	Error while reading status
5:4	LED_DRV	00	R/W	LED_DRV current limit: b00=12.5mA; b01=25mA; b10=50mA; b11=100mA Device depends on register DEV_SEL
3	ENABLE LED_DRV	0	R/W	Enable LED DRV Device depends on register DEV_SEL
2:1	LED_INT	01	R/W	Current limit: b00=1mA; b01=2mA; b10=4mA; b11=8mA Device depends on register DEV_SEL
0	ENABLE LED_INT	0	R/W	Enable LED IND Device depends on register DEV_SEL

Hier wird zunächst mit einem virtualWriteRegister die richtige LED ausgewählt (entweder weiß, infrarot oder UV). Danach wird in das Register an die Stelle des 3. Bits eine 1 geschrieben, um die LED einzuschalten.

cpp

```
//Enable the LED or bulb on a given device
void AS7265X::enableBulb(uint8_t device)
{
    selectDevice(device);

    //Read, mask/set, write
    uint8_t value = virtualReadRegister(AS7265X_LED_CONFIG);
    value |= (1 << 3); //Set the bit
    virtualWriteRegister(AS7265X_LED_CONFIG, value);
}
```

Für die Funktion `disableBulb()`, wird das Bit an der selben Stelle gelöscht.

cpp

```
//Disable the LED or bulb on a given device
void AS7265X::disableBulb(uint8_t device)
{
    selectDevice(device);

    //Read, mask/set, write
    uint8_t value = virtualReadRegister(AS7265X_LED_CONFIG);
    value &= ~(1 << 3); //Clear the bit
```

```

    virtualWriteRegister(AS7265X_LED_CONFIG, value);
}

```

Für die eigentliche Messung ist die Funktion **takeMeasurements()** verantwortlich, welche die Messung aller 18 Kanäle steuert. Ein volle Messung der 18 Kanäle wird dadurch ausgeführt, indem das Konfigurationsregister zuerst gelesen und dann das der Modus auf "Modus 3" gesetzt wird. Danach wird das Register wieder auf die entsprechende Adresse geschrieben.

cpp

```

//Mode 0: 4 channels out of 6 (see datasheet)
//Mode 1: Different 4 channels out of 6 (see datasheet)
//Mode 2: All 6 channels continuously
//Mode 3: One-shot reading of all channels
void AS7265X::setMeasurementMode(uint8_t mode)
{
    if (mode > 0b11)
        mode = 0b11; //Error check

    //Read, mask/set, write
    uint8_t value = virtualReadRegister(AS7265X_CONFIG); //Read
    value &= 0b11110011; //Clear BAN
    value |= (mode << 2); //Set BAN
    virtualWriteRegister(AS7265X_CONFIG, value); //Write
}

```

Zuletzt wird die Funktion **dataAvailable()** in einer While-Schleife aufgerufen. Dort wird mit einem `virtualReadRegister` das Konfigurationsregister solange ausgelesen, bis an Bit Position 1 (nicht verwechseln mit Bit-Position 0!) eine 1 steht. Dadurch wird signalisiert, dass die Daten bereit liegen.

cpp

```

//Checks to see if DRDY flag is set in the control setup register
boolean AS7265X::dataAvailable()
{
    uint8_t value = virtualReadRegister(AS7265X_CONFIG);
    return (value & (1 << 1)); //Bit 1 is DATA_RDY
}

```

}

Lesen der einzelnen Kanäle

Die einzelnen Kanäle werden mit den **getCalibrated()** Funktionen ausgelesen. Diese Methode ist wiederum nur eine Wrapper-Funktion für die Funktion **getCalibratedValue()**. Als Parameter erhält diese die jeweils entsprechende Adressen der auszulesenden Kanäle und die Geräte-Adresse. Die Geräte-Adresse entspricht jeweils der Adresse für entweder den AS72651, AS72652 oder AS72653.

Funktion: **getCalibratedValue()**

Zunächst wird die Funktion **selectDevice()** aufgerufen, um das jeweilige Gerät auszuwählen, von dem der Kanal ausgelesen werden soll.

Dazu wird in das Device-Control Register unter der Adresse **0x4F** an der jeweiligen Stelle ein Bit gesetzt, je nachdem, von welchem Gerät gelesen werden soll.

cpp

```
//As we read various registers we have to point at the master or
void AS7265X::selectDevice(uint8_t device)
{
    //Set the bits 0:1. Just overwrite whatever is there because m
    virtualWriteRegister(AS7265X_DEV_SELECT_CONTROL, device);

    //This fails
    //uint8_t value = virtualReadRegister(AS7265X_DEV_SELECT_CONTR
    //value &= 0b11111100; //Clear lower two bits
    //if(device < 3) value |= device; //Set the bits
    //virtualWriteRegister(AS7265X_DEV_SELECT_CONTROL, value);
}
```

Im Anschluss wird jeweils ein Byte aus dem zu lesenden Kanal gelesen. Aus dem Datenblatt haben wir uns hergeleitet, dass immer nur eine Gruppe an Kanälen gleichzeitig gelesen werden kann. (siehe Bild unten) Deshalb werden hier direkt 4 Bytes ausgelesen. Eine genaue Erklärung haben wir hierfür jedoch nicht gefunden.

Calibrated Value Channel R,G,A Register

Addr: 0x17,0x016,0x15,0x014		Calibrated Value Channel R,G,A			
Bit	Bit Name	Default	Access	Bit Description	
31:24	CAL CHAN0_3	FF	R	Channel R or J or D depends on register DEV_SEL	
23:16	CAL CHAN0_2	FF	R		
15:8	CAL CHAN0_1	FF	R		
7:0	CAL CHAN0_0	FF	R		

Die gelesenen Bytes werden nun der Ordnung nach in einen 32-Bit Integer zusammengesetzt und danach in einen Float konvertiert.

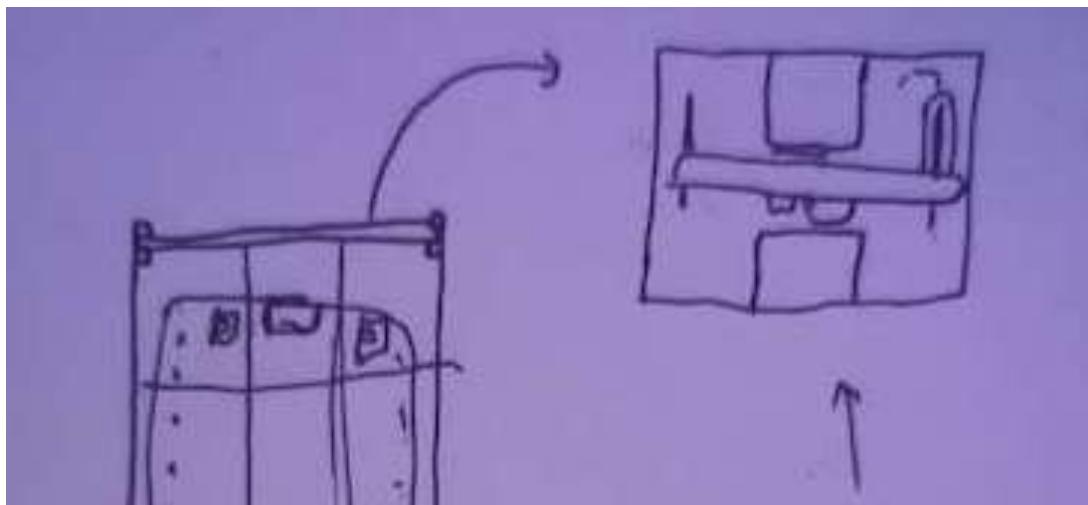
cpp

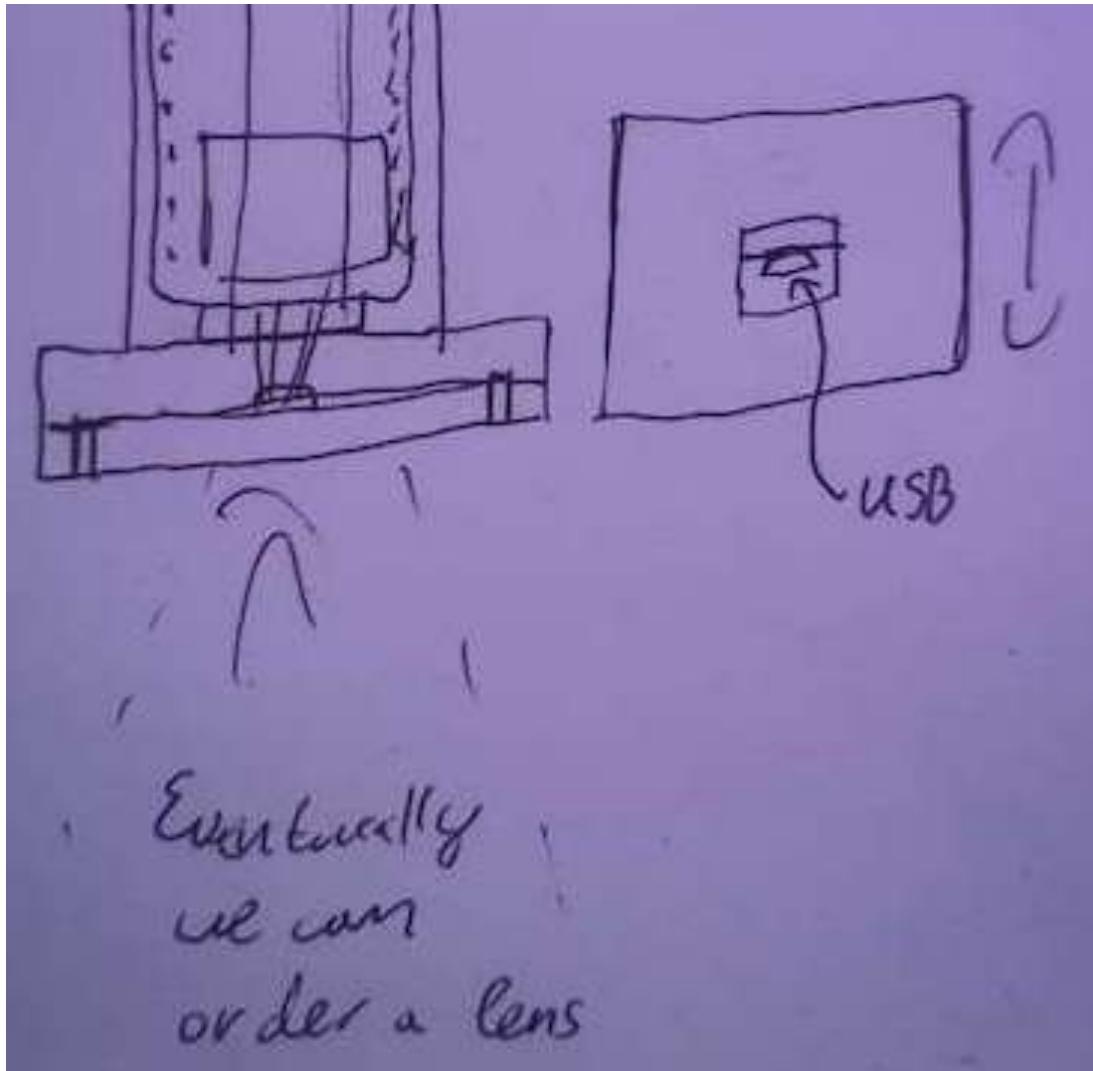
```
//Channel calibrated values are stored big-endian
uint32_t calBytes = 0;
calBytes |= ((uint32_t)b0 << (8 * 3));
calBytes |= ((uint32_t)b1 << (8 * 2));
calBytes |= ((uint32_t)b2 << (8 * 1));
calBytes |= ((uint32_t)b3 << (8 * 0));

return (convertBytesToFloat(calBytes));
```

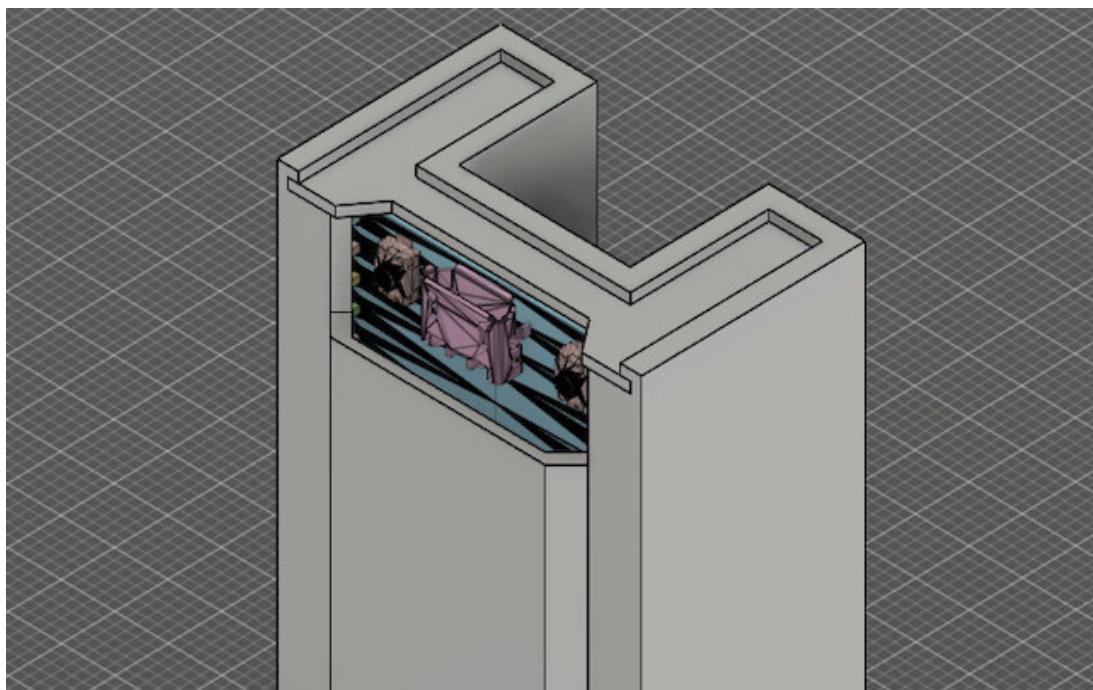
Gehäuse

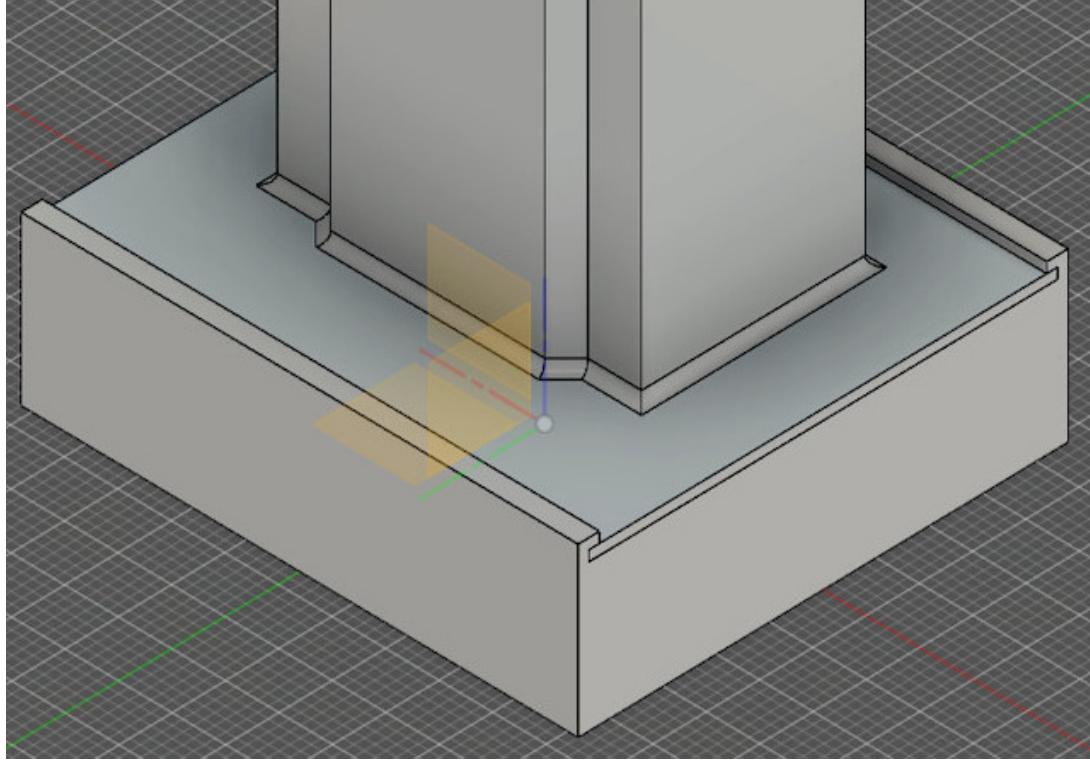
Erstes Design



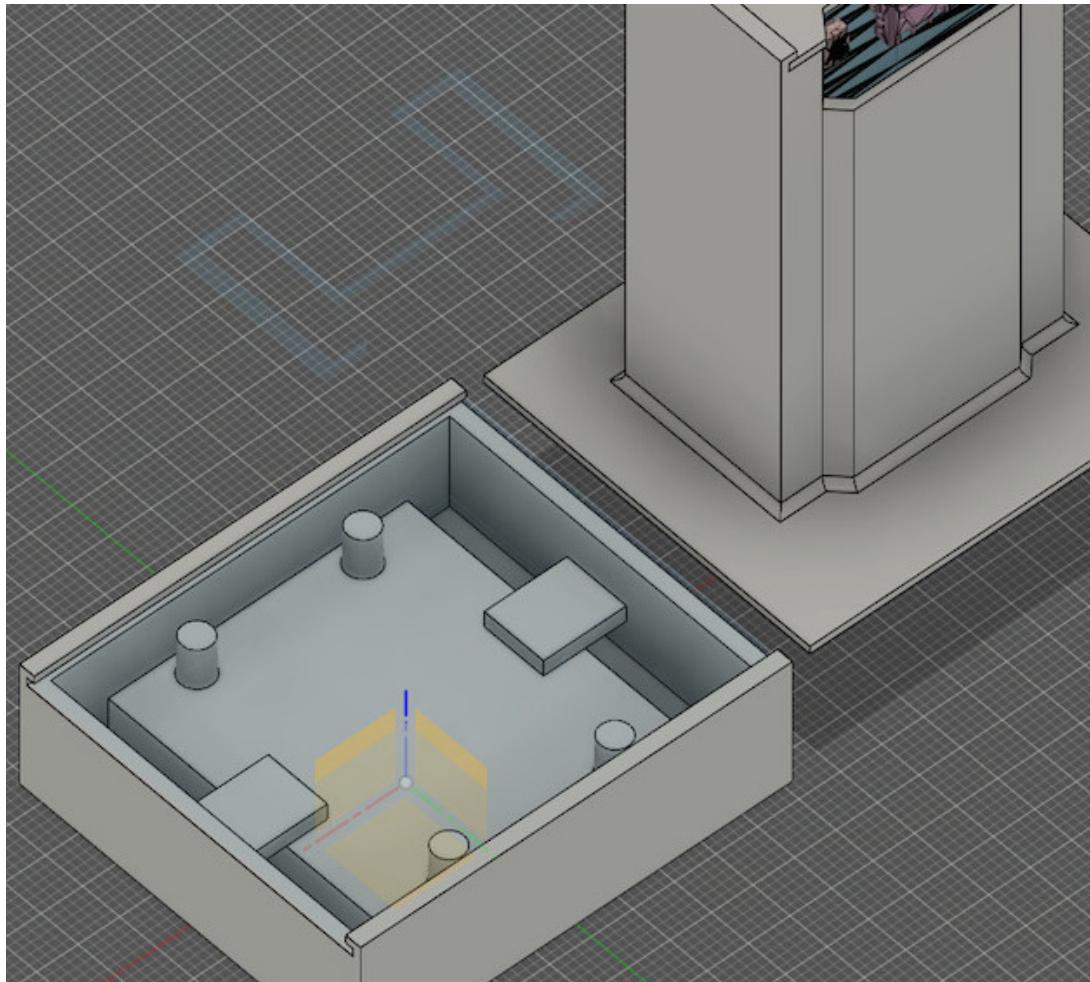


Umsetzung in Fusion 360



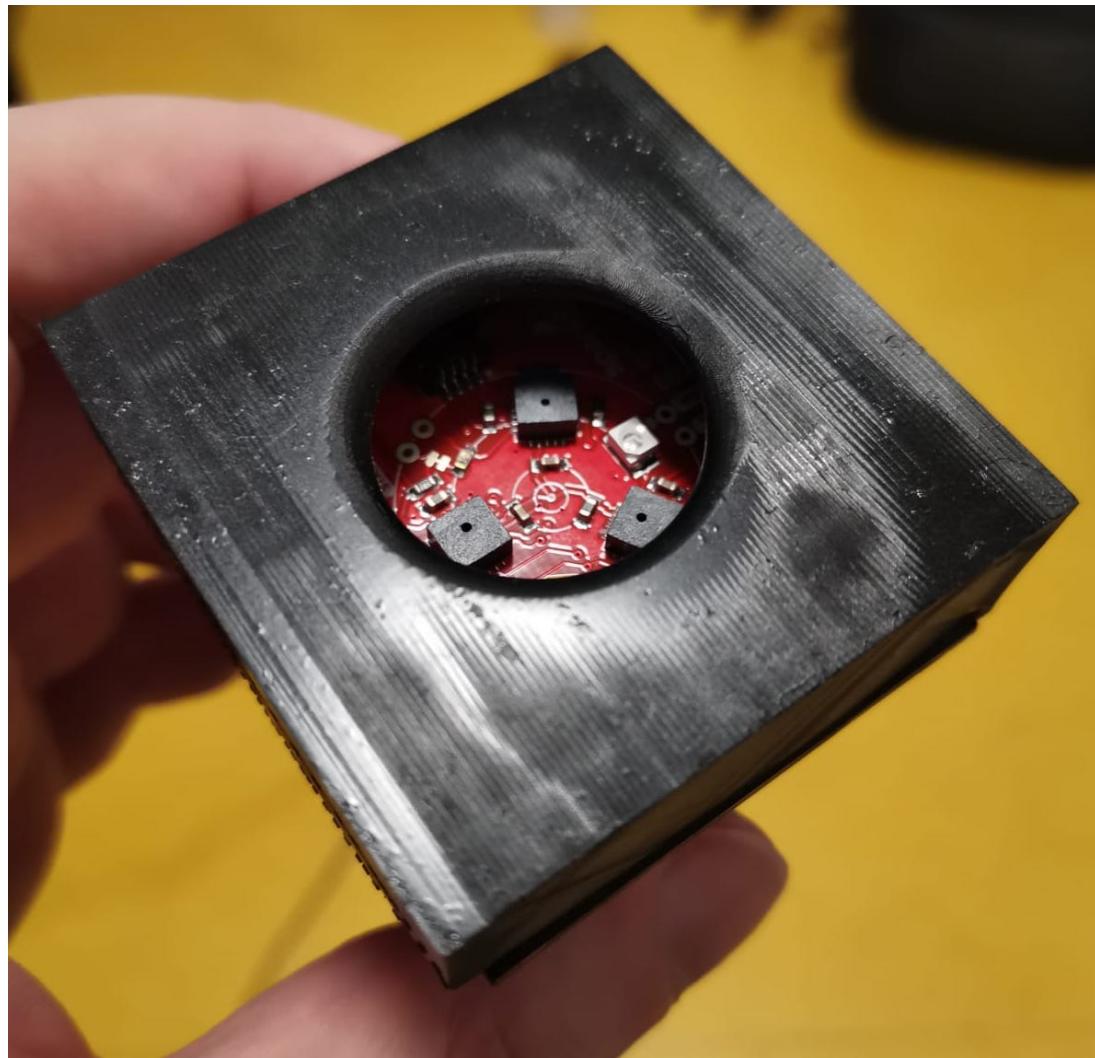


Das Design des Geräts hatte einige Probleme. Zunächst fehlten Abstände zwischen den gleitenden Teilen, wodurch viel Bearbeitung erforderlich war.

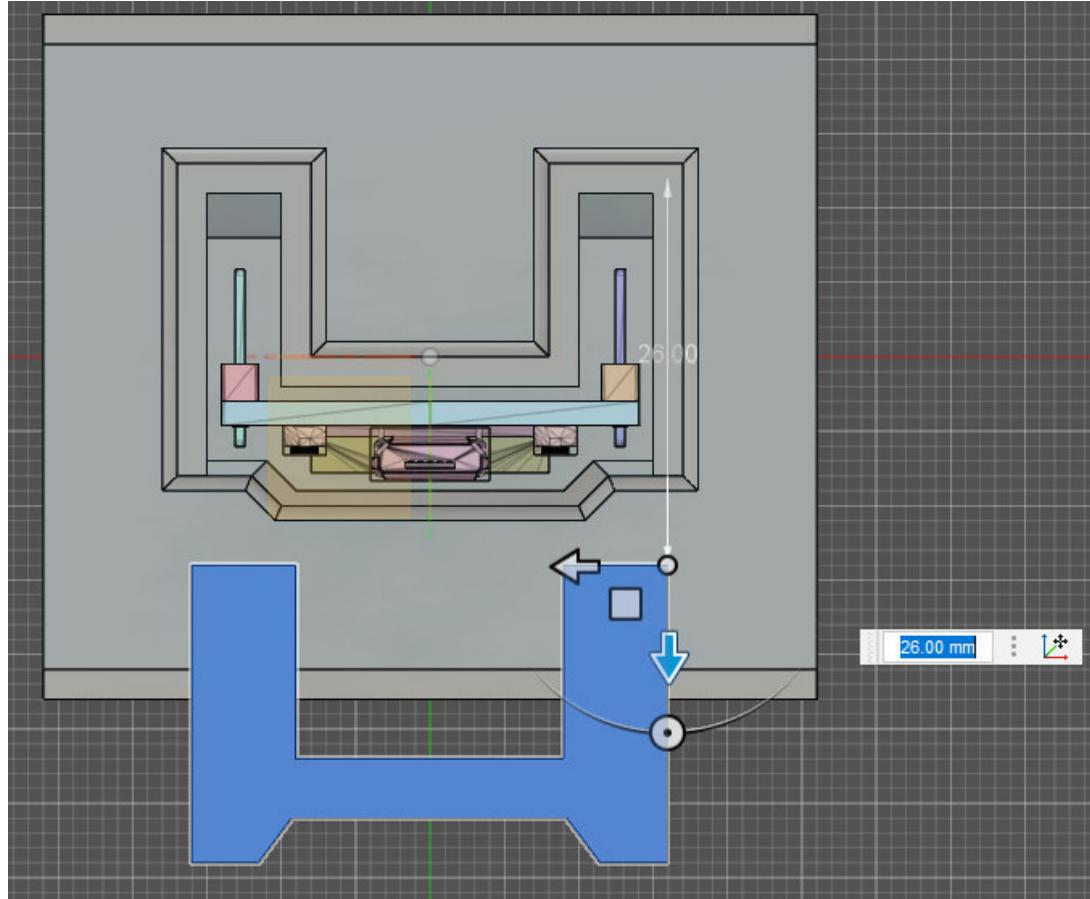




Im unteren Teil des Gehäuses, in dem sich der AS7265X-Sensor befand, waren die Positionierungsstifte sowohl in der Position als auch in der Breite um 1 mm falsch gemessen. Dies führte zu viel Schleifarbeiten in einem schwer zugänglichen Bereich. Diese Bearbeitung hätte vor dem Aushärten des Gehäuses erfolgen sollen, um das Gehäuse leichter bearbeiten zu können.



Das Gehäuse war zudem viel zu hoch, wodurch der Sensor zu weit von der Öffnung entfernt saß. In der nächsten Revision wurde das Gehäuse um 4 mm abgesenkt, um sicherzustellen, dass der Sensor möglichst nah an der Öffnung ist. Ein zu großer Abstand zwischen dem Sensor und der Öffnung kann dazu führen, dass das Gehäuse selbst gemessen wird und somit den Messwert des Sensors verfälscht. Es ist daher wichtig, dass der Sensor möglichst nah an der Öffnung angebracht ist, um eine genaue Messung zu gewährleisten.



Im oberen Teil des Gehäuses befanden sich die Öffnungen für die Kabel des ESP. Diese waren etwas eng und es war daher schwierig, sie einzuführen, ohne die Pins zu verbiegen. Die beiden Stiftbänke führten auch zu unterschiedlichen Seiten des AS7265-Sensors, was eine umständliche Umverlegung erforderte. Die Kappe für das ESP-Gehäuse bot zudem nicht genug Platz für ein USB-Kabel und war zu klein, was dazu führte, dass sie schwer zu positionieren war und eine fragwürdige Haltbarkeit aufwies.

Um das Gehäuse zu optimieren, gibt es einige mögliche Verbesserungen, die über die bereits genannten Probleme hinausgehen. Zum Beispiel könnte man das AS7265X-Gehäuse asymmetrisch gestalten und nur einen der Qwicc-Anschlüsse zugänglich machen. Dies würde das Design kleiner machen oder mehr Platz für die Kabelführung bieten und den Abstand zwischen der Kabelführungsöffnung und dem Sensor vergrößern, wodurch die Störung durch externes Licht verringert würde. Eine seitliche Montage des ESP würde das Handling erleichtern, da das Gerät in der Regel an einen Laptop angeschlossen wird und sich in diesem Fall in der Höhe oder unterhalb des Laptops befindet. Abgerundete Kanten würden das Gerät besser in der Hand liegen lassen und eine reduzierte Wandstärke von maximal 2 mm würde Defekte durch Gewicht und Materialverschwendungen verringern.

Setup Cloud DB

BwCloud

Mit folgenden Schritten kann man sich auf der BwCloud Weboberfläche anmelden:

- Zur BwCloud Weboberfläche www.bw-cloud.org navigieren.
- Menüpunkt **Dashboard** auswählen.
- **Anmelden**.
- BwCloud SCOPE: Hochschule Heilbronn auswählen.
- Mit dem Hochschulaccount anmelden.

[BwCloud Dokumentation](#) zu den vorherigen Schritten.

Erstelle anschließend eine virtuelle Instanz (vServer). BwCloud Dokumentation: [Eine Instanz starten](#)

Um sich mit der virtuellen Instanz zu verbinden ist standardmäßig Port 22 für den SSH Zugang freigeschaltet. Damit aber auch auf die InfluxDB und Grafana zugegriffen werden kann müssen noch Port 3000 (Grafana) und 8086 (InfluxDB) freigegeben werden. BwCloud Dokumentation: [Einen Port für Zugriff \(von außen\) öffnen](#)

Wir haben uns für die BwCloud entschieden, da sie für Studenten einen kostenlosen Server bereitstellt. Außerdem kann auch jeder andere Serverprovider verwendet werden. Wenn die Daten nur lokal zur Verfügung stehen müssen, kann auch der eigene Rechner verwendet werden.

Docker Installation

Eine Anleitung wie Docker auf Ubuntu zu installieren ist, kann in der [Docker Dokumentation](#) gefunden werden. Damit bei einer erneuten Installation nicht wieder alle Befehle kopiert und ausgeführt werden müssen, haben wir diese wieder in einem Shell Skript zusammengefasst.

bash

```
#!/bin/bash
```

```
# Uninstall old versions
sudo apt-get remove docker docker-engine docker.io containerd runc

# Update the apt package index and install packages to allow apt to
# easily pull from https://
sudo apt-get update
sudo apt-get install ca-certificates curl gnupg lsb-release -y

# Add Docker's official GPG key:
sudo mkdir -p /etc/apt/keyrings
curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo gpg --dearmor -o /etc/apt/keyrings/docker.gpg

# Use the following command to set up the repository:
echo "deb [arch=$(dpkg --print-architecture) signed-by=/etc/apt/keyrings/docker.gpg] https://download.docker.com/linux/ubuntu $(lsb_release -cs) stable" | sudo tee /etc/apt/sources.list.d/docker.list >> /dev/null

# Install Docker Engine
sudo apt-get update
sudo apt-get install docker-ce docker-ce-cli containerd.io dockerd

# Start Docker Engine
sudo systemctl start docker
```

Dieses Docker installations Skript unterscheidet sich von dem des Raspberry Pis in wenigen Punkten!

InfluxDB

Im ersten Schritt haben wir dieses Mal nur die InfluxDB aufgesetzt um zu testen, ob Daten von dem ESP32 aus an die Datenbank geschickt werden können.

Erstelle dafür die Datei `docker-compose.yml` mit folgendem Inhalt:

```
version: '3'
services:
  influxdb:
    image: influxdb:latest
    container_name: influxdb
    restart: always
    ports:
      - "8086:8086"
```

```
volumes:  
  - ./influxdb/data:/var/lib/influxdb2/  
  - ./influxdb/config:/etc/influxdb2/  
  
environment:  
  - DOCKER_INFLUXDB_INIT_MODE=setup  
  - DOCKER_INFLUXDB_INIT_USERNAME=${INFLUXDB_ADMIN_USER}  
  - DOCKER_INFLUXDB_INIT_PASSWORD=${INFLUXDB_ADMIN_PASSWORD}  
  - DOCKER_INFLUXDB_INIT_ORG=${INFLUXDB_ORG}  
  - DOCKER_INFLUXDB_INIT_BUCKET=${INFLUXDB_BUCKET}  
  - DOCKER_INFLUXDB_INIT_ADMIN_TOKEN=${INFLUXDB_ADMIN_TOKEN}
```

Starte die Docker Container mit `docker compose up -d`.

API- und UI-Zugriff erlangen

Das User-Interface (UI) und die API nutzen Port 8086. Um diesen Container Port auch von außen erreichbar zu machen wird in der docker-compose.yml unter dem Punkt `ports:` der Host (BwCloud-Server) Port 8086 auf den Container Port 8086 zugeordnet.

Daten persistieren

Das InfluxDB-Image stellt ein freigegebenes Volume unter `/var/lib/influxdb2` bereit. Um die Containerdaten der Datenbank zu persistieren, haben wir hier ein Host-Verzeichnis an diesem Punkt bereitgestellt.

Beachte, dass dieser Pfad sich von den InfluxDB Versionen 1.x unterscheidet!

Automatisierte Datenbankeinrichtung

Das InfluxDB-Image enthält einige zusätzliche Funktionen, um das System automatisch zu booten. Diese Funktionalität wird aktiviert, indem die Umgebungsvariable `DOCKER_INFLUXDB_INIT_MODE` auf den Wert `setup` gesetzt wird, wenn der Container ausgeführt wird. Zusätzliche Umgebungsvariablen werden verwendet, um die Setup-Logik zu konfigurieren:

- **DOCKER_INFLUXDB_INIT_USERNAME** : Der Benutzername, der für den anfänglichen Superuser des Systems festgelegt werden soll (erforderlich).
- **DOCKER_INFLUXDB_INIT_PASSWORD** : Das Kennwort, das für den ersten Superuser des Systems festgelegt werden soll (erforderlich).
- **DOCKER_INFLUXDB_INIT_ORG** : Der Name, der für die anfängliche Organisation des Systems festgelegt werden soll (erforderlich).
- **DOCKER_INFLUXDB_INIT_BUCKET** : Der Name, der für den anfänglichen Bucket des Systems festgelegt werden soll (erforderlich).
- **DOCKER_INFLUXDB_INIT_ADMIN_TOKEN** : Das Authentifizierungstoken, das dem anfänglichen Superuser des Systems zugeordnet werden soll. Wenn nicht festgelegt, wird vom System automatisch ein Token generiert.

Das automatisierte Setup generiert Metadaten-Dateien und CLI-Konfigurationen. Es wird empfohlen, Volumes auf beiden Pfaden (`/var/lib/influxdb2/` , `/etc/influxdb2/`) bereitzustellen, um Datenverluste zu vermeiden.

InfluxDB Weboberfläche

Den erfolgreichen Start des Containers kann überprüft werden, indem man sich auf der Weboberfläche <http://YOUR-SERVER-ADDRESS:8086/> anmeldet.

Ob der ESP die Messdaten über die API Schnittstelle auf der Datenbank speichern kann, testet man indem in der Weboberfläche nachgeschaut, ob die Daten in der Datenbank vorliegen.



Die Repräsentation der Daten in der Oberfläche entspricht noch keinem Format, das zum Vergleichen verwendet werden kann. Aber es zeigt das die Daten erfolgreich in der Datenbank liegen und diese Daten auch wieder ausgelesen werden können.

[◀ PREVIOUS](#)

Überblick

[NEXT ▶](#)

Stage 2

Stage 2 - Erweitern des Konzeptes

In der zweiten Stage wird das Konzept weiter ausgebaut, indem die Datenvisualisierung auf einem Grafana Dashboard aufgebaut wird. Damit diese Daten gruppiert und verglichen werden können, müssen sie beschriftet werden. Der Schritt der Datenbeschriftung muss vor dem Senden in die Datenbank auf dem EPS umgesetzt werden. Zum Beschriften der Messdaten müssen dem ESP vor dem Messvorgang die Eigenschaften übermittelt werden.

Messdaten beschriften

Dazu wird prototypisch die serielle Schnittstelle des ESPs genutzt. Das hat den Vorteil, dass keine weitere Hardware benötigt wird bis auf den Rechner, der schon für das Programmieren des ESPs benötigt wird.

Measurement und Tags dynamisch setzen

Es wurde entschieden, die Speicherung und Beschriftung der Daten dynamisch zu realisieren. Dazu wurden im Code die Funktionen `setNewTag()` und `setNewMeasurement()` verwendet. Die Funktion `setNewTag()` zur Beschriftung der Daten. Die Funktion `setNewMeasurement()`, um festzulegen in welches Measurement die Daten gespeichert werden sollen. Unter den gleichnamigen Unterkapiteln, wird in Stage 3 auf die Funktionsweise eingegangen.

Grafana

Cloud Änderungen

Aufgrund der aktuellen Lage des Hacker Angriffs auf die Hochschule Heilbronn, sind deren Systeme nicht mehr mit dem Internet verbunden. Somit ist es nicht möglich

sich in der BwCloud mit dem Hochschul-Account anzumelden, da der Authentifizierungsserver nicht erreichbar ist.

Beim Aufsetzen von Grafana stehen wir somit vor der Herausforderung, dass wir keinen Zugriff auf die Firewall der BwCloud haben und dadurch keinen weiteren Port für Grafana freigeben können.

Durch die Skripte und Docker ist es einfach möglich das bestehende System auf einem alten vorhandenen vServer neu aufzusetzen. Zu diesem Zeitpunkt haben wir die BwCloud durch einen vorhandenen vServer ausgetauscht. Codetechnisch muss lediglich kleine Konfigurationsänderungen wie zum Beispiel die IP-Adresse vorgenommen werden.

Daten die bereits in der Datenbank liegen, könnten durch Kopieren des influxdb/ Ordners auf den "neuen" vServer übertragen werden.

docker-compose.yml anpassen

Um auch Grafana mit Docker-Compose starten zu können, wird der neue Service "Grafana" der `docker-compose.yml` hinzugefügt. Hier werden ebenfalls, für die Persistenz der Dashboards und weiteren Konfigurationsdaten, Volumes angelegt.

yml

```
volumes:  
  - ./grafana/data:/var/lib/grafana/ # working directory  
  - ./grafana/provisioning:/etc/grafana/provisioning/ # provisi
```

Die Volume `Provisioning` wird zu diesem Zeitpunkt noch nicht gebraucht. Diese kann jedoch zukünftig für automatisiertes Einrichten von Datenquellen und Dashboards verwendet werden.

Grafana läuft auf dem Port 3000, der dem Host Port 3000 zugeordnet wird.

Für das automatisierte Einrichten von Grafana, werden folgende Environment-Variablen für den Container gesetzt:

- GF_SECURITY_ADMIN_USER=\${GF_SECURITY_ADMIN_USER}
- GF_SECURITY_ADMIN_PASSWORD=\${GF_SECURITY_ADMIN_PASSWORD}
- GF_AUTH_ANONYMOUS_ORG_ROLE=Admin
- GF_AUTH_ANONYMOUS_ENABLED=true
- GF_ENABLE_GZIP=true

Die Änderungen an der `docker-compose.yml` sehen wie folgt aus:

```
yaml

version: '3'
services:
  influxdb:
    image: influxdb:latest
    container_name: influxdb
    restart: always
    ports:
      - "8086:8086"
    volumes:
      - ./influxdb/data:/var/lib/influxdb2/
      - ./influxdb/config:/etc/influxdb2/
    environment:
      - DOCKER_INFLUXDB_INIT_MODE=setup
      - DOCKER_INFLUXDB_INIT_USERNAME=${INFLUXDB_ADMIN_USER}
      - DOCKER_INFLUXDB_INIT_PASSWORD=${INFLUXDB_ADMIN_PASSWORD}
      - DOCKER_INFLUXDB_INIT_ORG=${INFLUXDB_ORG}
      - DOCKER_INFLUXDB_INIT_BUCKET=${INFLUXDB_BUCKET}
      - DOCKER_INFLUXDB_INIT_ADMIN_TOKEN=${INFLUXDB_ADMIN_TOKEN}

  grafana:
    image: grafana/grafana:latest
    container_name: grafana
    restart: unless-stopped
    ports:
      - "3000:3000"
    volumes:
      - ./grafana/data:/var/lib/grafana/
      - ./grafana/provisioning/:/etc/grafana/provisioning/
    environment:
      - GF_SECURITY_ADMIN_USER=${GF_SECURITY_ADMIN_USER}
```

```
- GF_SECURITY_ADMIN_PASSWORD=${GF_SECURITY_ADMIN_PASSWORD}
- GF_AUTH_ANONYMOUS_ORG_ROLE=Admin
- GF_AUTH_ANONYMOUS_ENABLED=true
- GF_ENABLE_GZIP=true

depends_on:
- influxdb

user: "1000" # USER ID des Docker Users anpassen
```

Der Ordner `grafana/` auf dem Host muss dem User 1000 gehören, sonst kann Grafana im Docker-Container keine weiteren Ordner anlegen!

Wie schon in Stage 1 kann die `docker-compose.yml` mit dem Befehl `docker-compose up -d` gestartet werden. Ob der Grafana-Container ebenfalls startet, kann mit dem Aufrufen der Weboberfläche `http://YOUR-SERVER-ADDRESS:3000/` überprüft werden.

- Weitere Dokumentation: [Run Grafana Docker image](#)

Datenquelle konfigurieren

Wenn man in der Weboberfläche angemeldet ist sollte eine Datenquelle hinzugefügt werden, bevor ein Dashboard zur Visualisierung erstellt wird.

Bei der Datenquelle handelt es sich nicht immer um eine Datenbank.

Eine Datenbank kann über das `Zahnradsymbol` im Seitenmenü > `Data sources` > `Add data source` hinzugefügt werden. Wähle hier die Datenquelle (in diesem Fall: InfluxDB) aus, die hinzugefügt werden soll und konfiguriere die Datenquelle gemäß den für diese Datenquelle spezifischen Anweisungen. Bei der InfluxDB handelt es sich hier um die `URL`, die `Organisation`, den `Token` und den `Bucket`, als die relevantesten Daten. Anstelle des Tokens kann auch `Basic auth` mit Benutzername und Passwort verwendet werden. Eine ausführlichere Beschreibung kann in der [Grafana Dokumentation](#) gefunden werden.

Dashboard einrichten

Ein Dashboard soll die Daten der letzten Messung visualisieren. Es soll ein Panel enthalten, dass das Spektrum des Produktes zeigt, damit hierfür ein Eindruck gewonnen werden kann. Somit kann auch Ansatzweise die korrekte Messung bestätigt werden, aus dem Erwartungswert und dem tatsächlich gewonnenen Spektrum. Weitere Dashboards sollen jeweils eine Kategorie von Produkten beinhalten. Diese beinhalten zwei Panels. Das Erste beinhaltet das Spektrum das alle Messwerte in dieser Kategorie als Durchschnitt repräsentiert. Ein weiteres Panel zeigt die letzte Messung der Kategorie, um diesen parallel mit dem anderen Panel vergleichen zu können, wie dieser Messwert vom Durchschnitt abweicht.

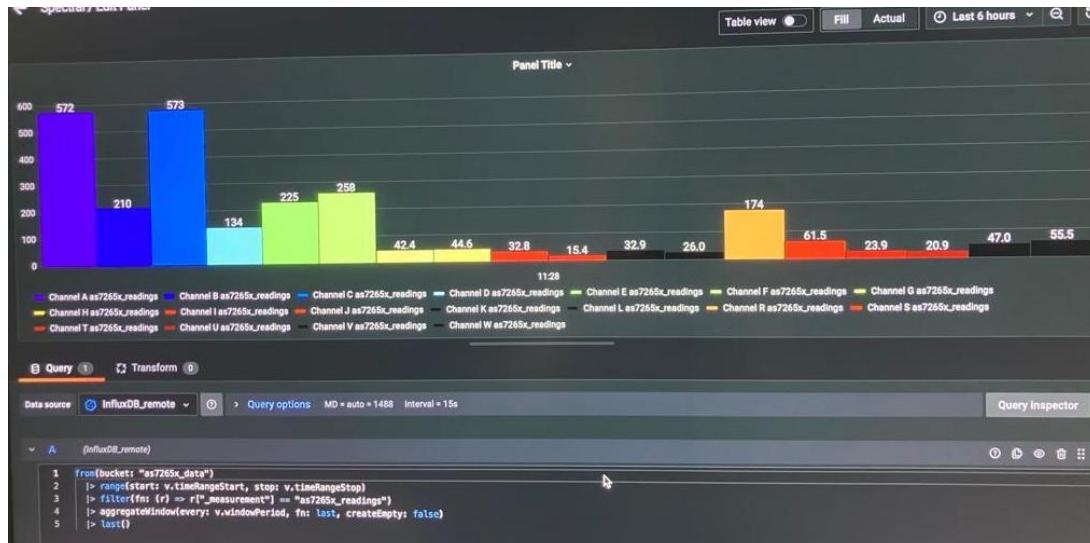
- Grafana Dokumentation: [Build your first dashboard](#)

Damit diese Dashboards diese Darstellungen visualisieren können, muss zunächst ein Flux Query erstellt werden, der die Datensätze in der Datenbank in die Datenform der Darstellung bringt. Für die Erstellung dieser Queries ist der Query Builder in der InfluxDB UI (User Interface) sehr hilfreich. Denn hier kann der Query Stück für Stück in die richtige Form gebracht werden. Danach kann dieser Query in das Grafana Panel übertragen werden.

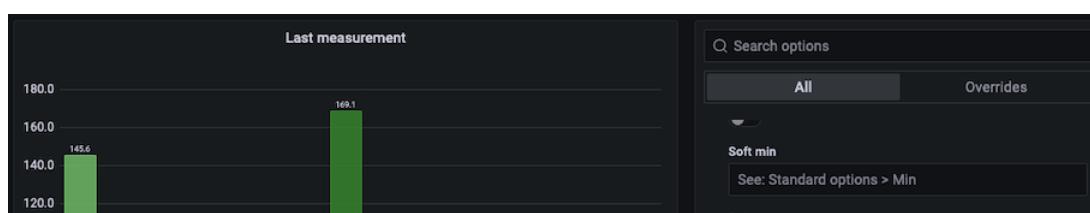
In Grafana kann dann die gewünschte Visualisierung konfiguriert werden. Herausfordernd wurde das Konfigurieren der gleichen Darstellung in den Barcharts. Denn ein Flux Query kann auf verschiedene Weise aufgebaut werden um an die selben Daten zu kommen. Jedoch sind diese Daten unterschiedlich sortiert und gruppiert, was eine andere Visualisierung in Grafana hervorruft. So mussten manche Flux Queries angepasst werden. Ein weiterer Punkt, der bei den Queries berücksichtigt werden musste, ist der vorhandene Zeitstempel. Ohne diesen kann Grafana die Daten nicht in einem Barchart darstellen. Sortiert wurden die Balken des Barcharts nach dem Namen der Channels die in der Datenbank hinterlegt sind. Diese Reihenfolge entspricht jedoch nicht der Reihenfolge des Wellenspektrums.

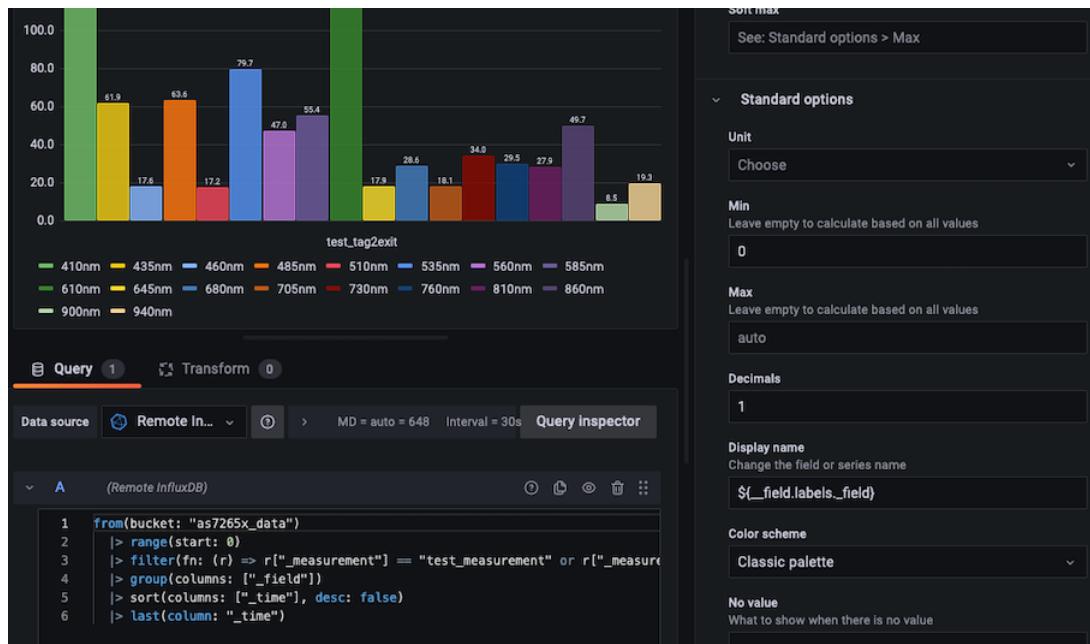
Channel	Spektrum
A	410nm
B	435nm
C	460nm
D	485nm
E	510nm

Channel	Spektrum
F	535nm
G	560nm
H	585nm
I	645nm
J	705nm
K	900nm
L	940nm
R	610nm
S	680nm
T	730nm
U	760nm
V	810nm
W	860nm



Um das Problem zu umgehen wird anstatt des Channel-Namens die entsprechende Wellenlänge hinterlegt. Ohnehin ist mit der Wellenlänge mehr anzufangen als mit dem Channel-Namen des Spektral-Sensors.





Für eine schönere Visualisierung wurden Overrides erstellt, die die Farbe der Balken, je nach Wellenlänge, entsprechend einfärbt. Weiterhin enthält der "Display name" noch weitere Attribute, sodass diese Attribute mit `__field.labels.__values` auf lediglich die Wellenlänge herunter gekürzt wird.



Das Konfigurieren der Datenquelle und besonders des Dashboards ist sehr zeitaufwendig, sodass hier eine Automatisierung sehr hilfreich wäre damit wiederholtes Einrichten nicht notwendig ist.

[◀ PREVIOUS](#)

Stage 1

[NEXT ▶](#)

Stage 3

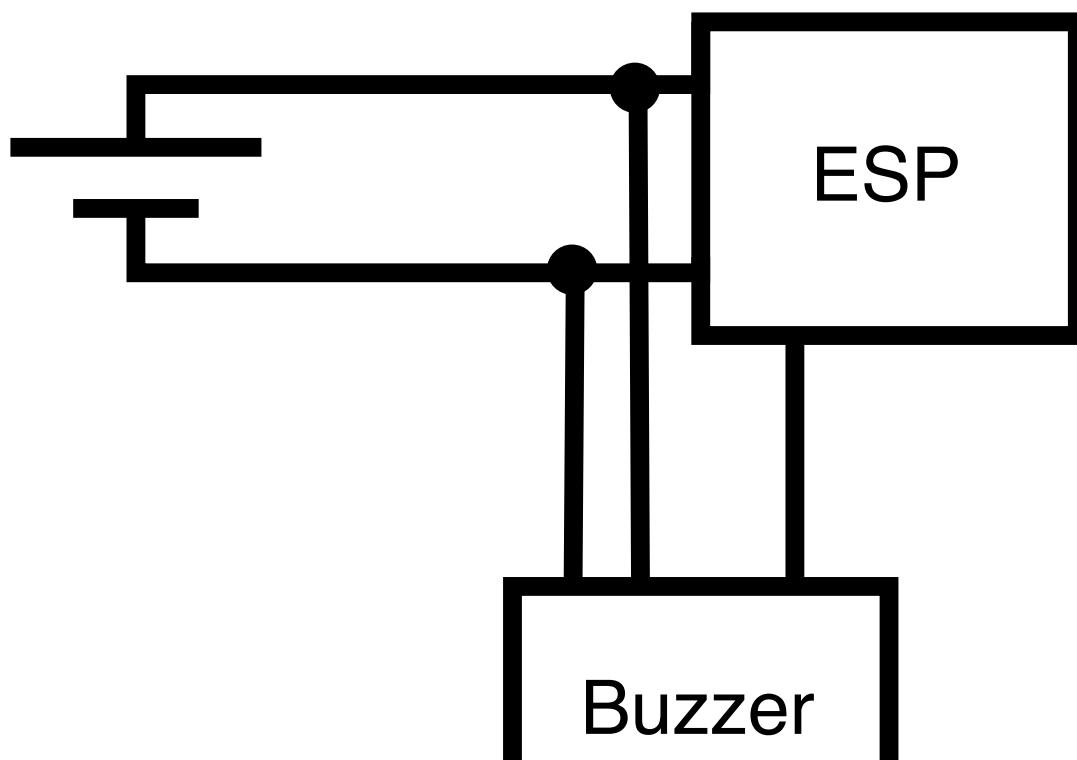
Stage 3 - Automatisierung / Erweiterungen

Um die Schritte bei erneutem Aufsetzen nicht erneut manuell ausführen zu müssen, werden diese Schritte in Stage 3 automatisiert. Hinzu kommen ebenfalls noch eine akustische Rückmeldung und ein Batteriepack um mit dem Messsystem mobil zu werden.

Messsystem Erweiterungen

Buzzer

Für die Feststellung, ob eine Messung durchgeführt wurde soll das Messsystem während der Messung eine akustische Rückmeldung geben. Dafür wird ein aktiver Buzzer verwendet, welcher zwei Pins für die Spannungsversorgung und ein weiterer Pin für das Signal besitzt.





Zum Ansteuern des Buzzers wird vor dem Messvorgang des Spektral Sensors, der Signal Pin (13) des Buzzers auf **HIGH** gezogen. Ist das Auslesen der Daten vom Spektral-Sensor beendet wird dieser Pin wieder auf **LOW** gezogen. Somit ist bekannt wie lange der Messvorgang aktiv ist und das Messsystem nicht bewegen werden sollte.

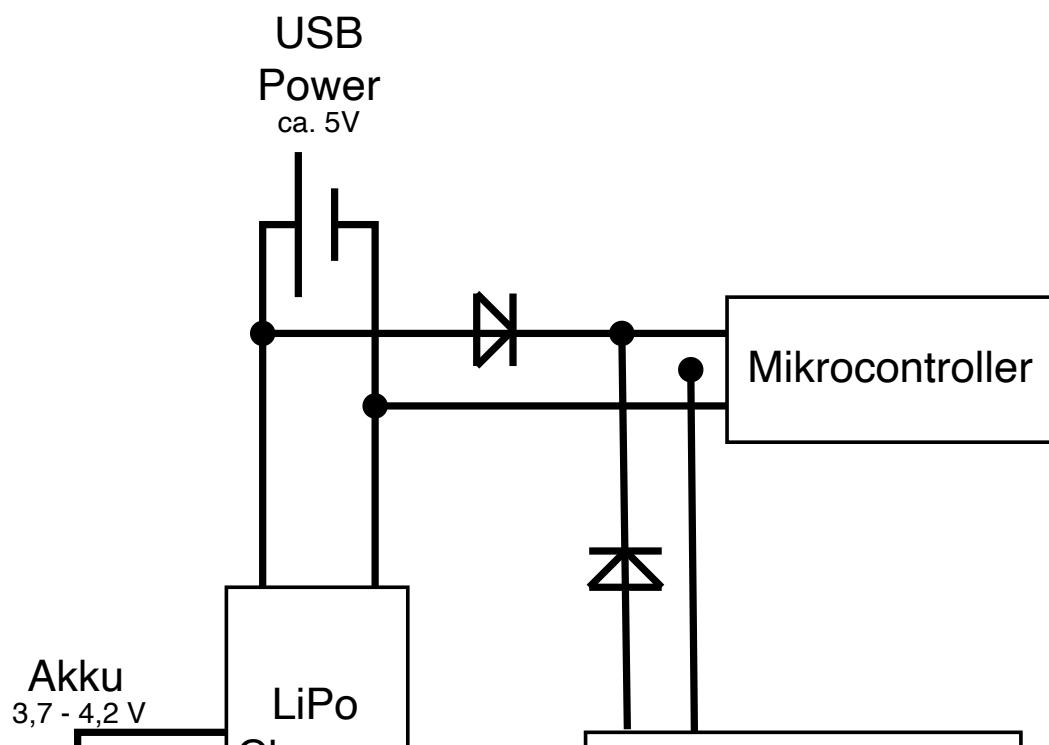
Hier der entsprechende Ausschnitt aus der **takeSensorData()** Funktion:

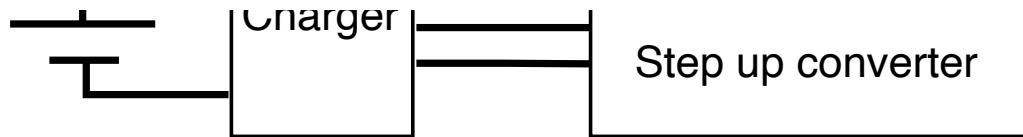
cpp

```
digitalWrite(13, HIGH);
Serial.println("Begin measurement....");
specSensor.takeMeasurementsWithBulb(); // This is a hard wait wh
digitalWrite(13, LOW);
```

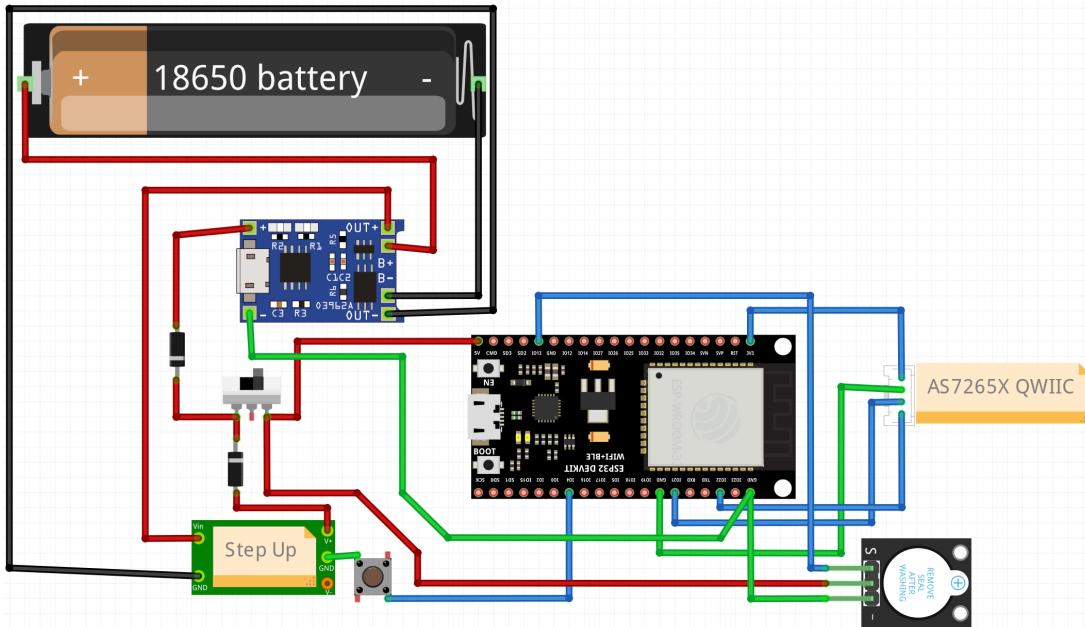
Batteriepack

Ein Batteriepack wird für das Messsystem hinzugenommen, um nicht auf ein Netzteil und eine Steckdose in Reichweite angewiesen zu sein. Hierfür wird ein [Lade-/Entlademodul \(HW-107\)](#), ein [Step-Up Converter \(SX1308\)](#), ein [Akku](#) und ein [Battery holder](#) benötigt.





Bei dieser Schaltung wird das ESP-Modul mit dem Spektral-Sensor bei angeschlossenem Netzteil am Lade-/Entlademodul über das Netzteil versorgt und der Akku geladen. Wird das Netzteil entfernt so speist der Akku über den Step-Up Konverter den ESP mit Sensor.

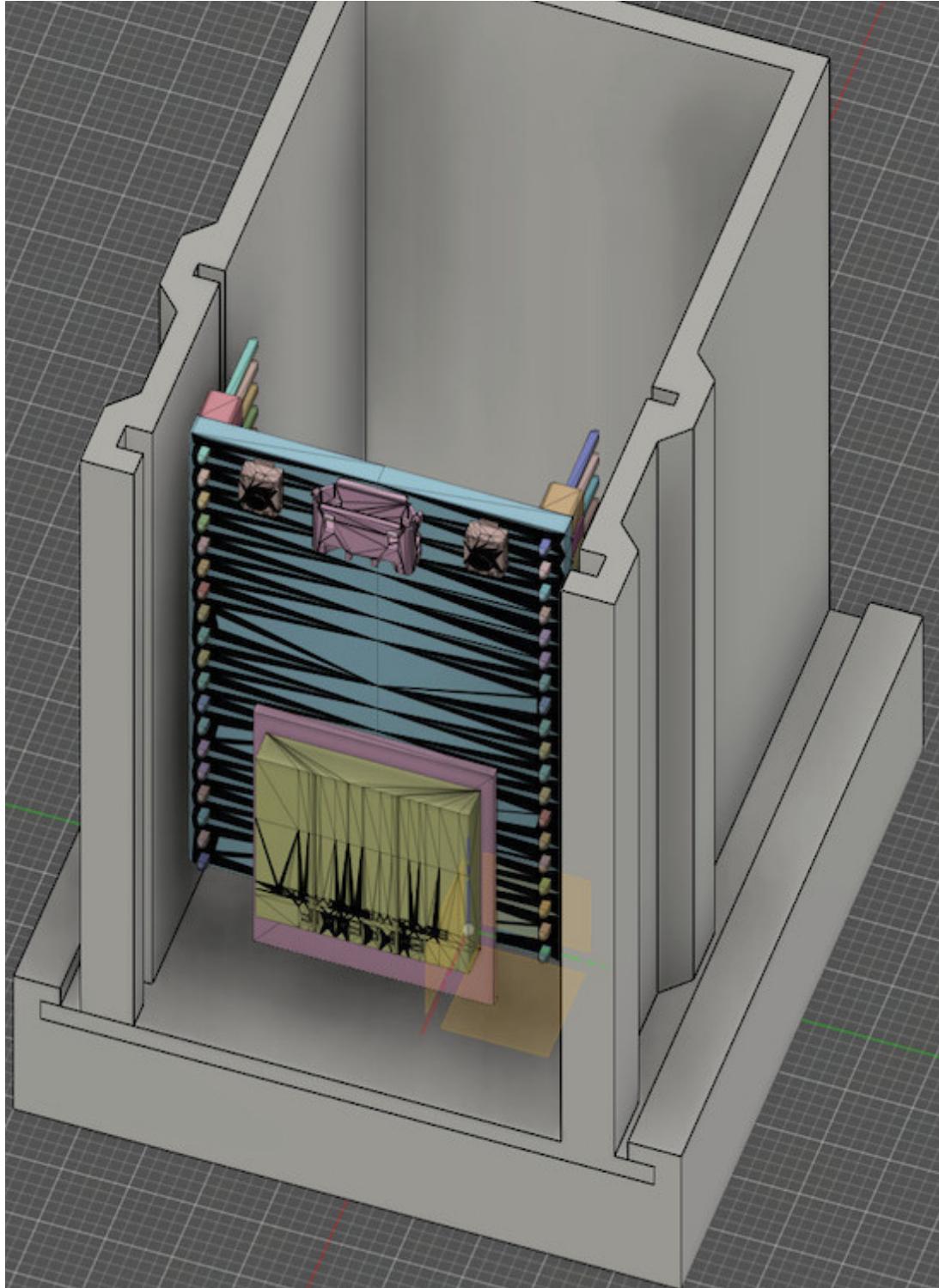


Durch hinzufügen von Batteriepack und Buzzer musste nun Änderungen am Gehäuse vorgenommen werden.

Gehäuse Anpassungen

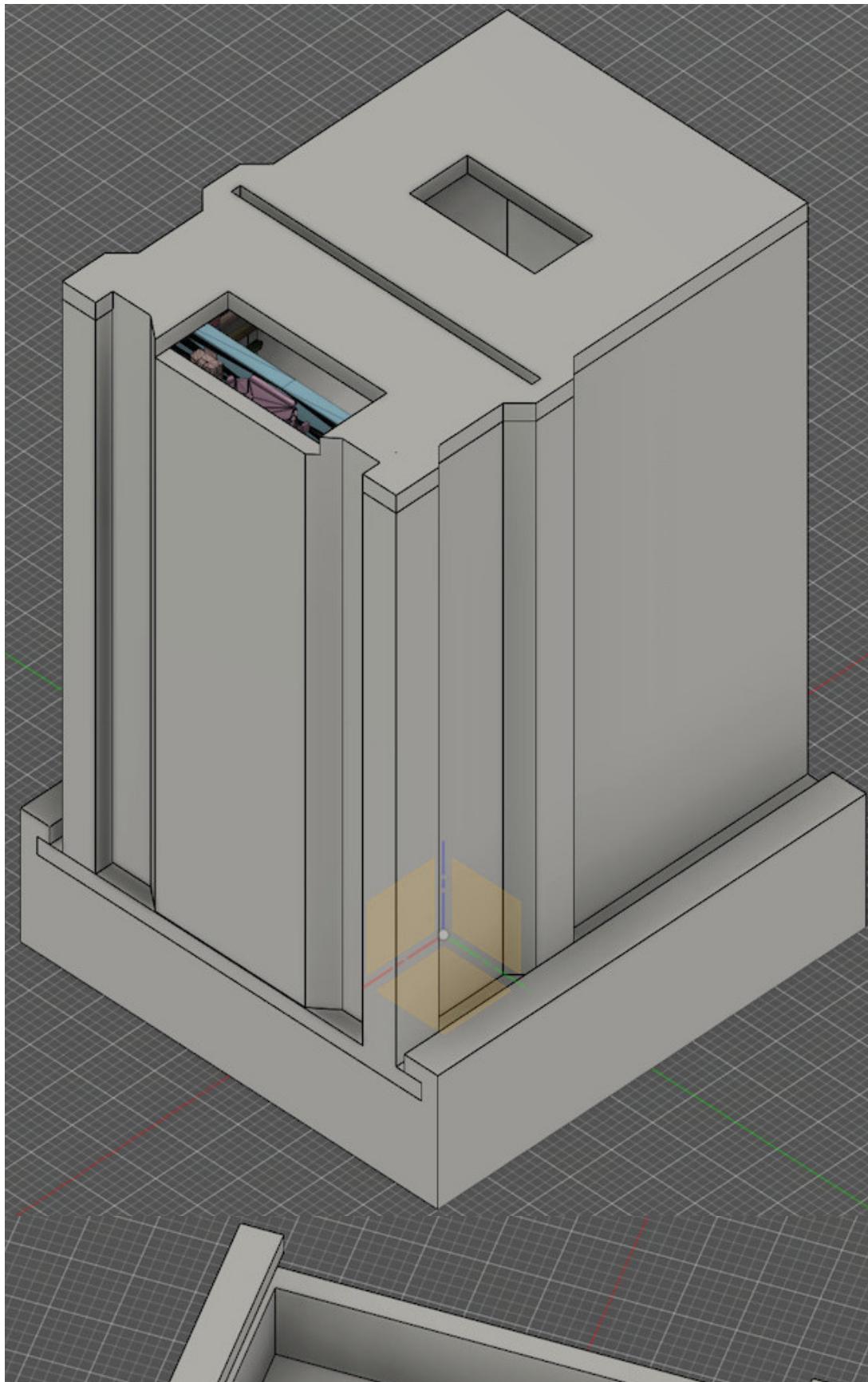
Die Hinzufügung der Batterie in Verbindung mit der mangelnden Zugänglichkeit des Vorgängers führte dazu, dass das ESP-Gehäuse völlig neu gestaltet werden musste. Das neue Design baut auf einem Breadboard auf. Auf der einen Seite des Breadboards wird der ESP und auf der anderen Seite die weiteren Komponenten wie der LiPo-Charger montiert. Durch eine Fuge wird das Breadboard auf beiden Seiten reibschlüssig in Position gehalten. Der Abstand zwischen Außenwand und Breadboard wird auf der Seite des ESPs durch diesen bestimmt. Auf der anderen Seite besitzt der Buzzer die höchste Einbautiefe, sodass dieser den Abstand vorgibt.

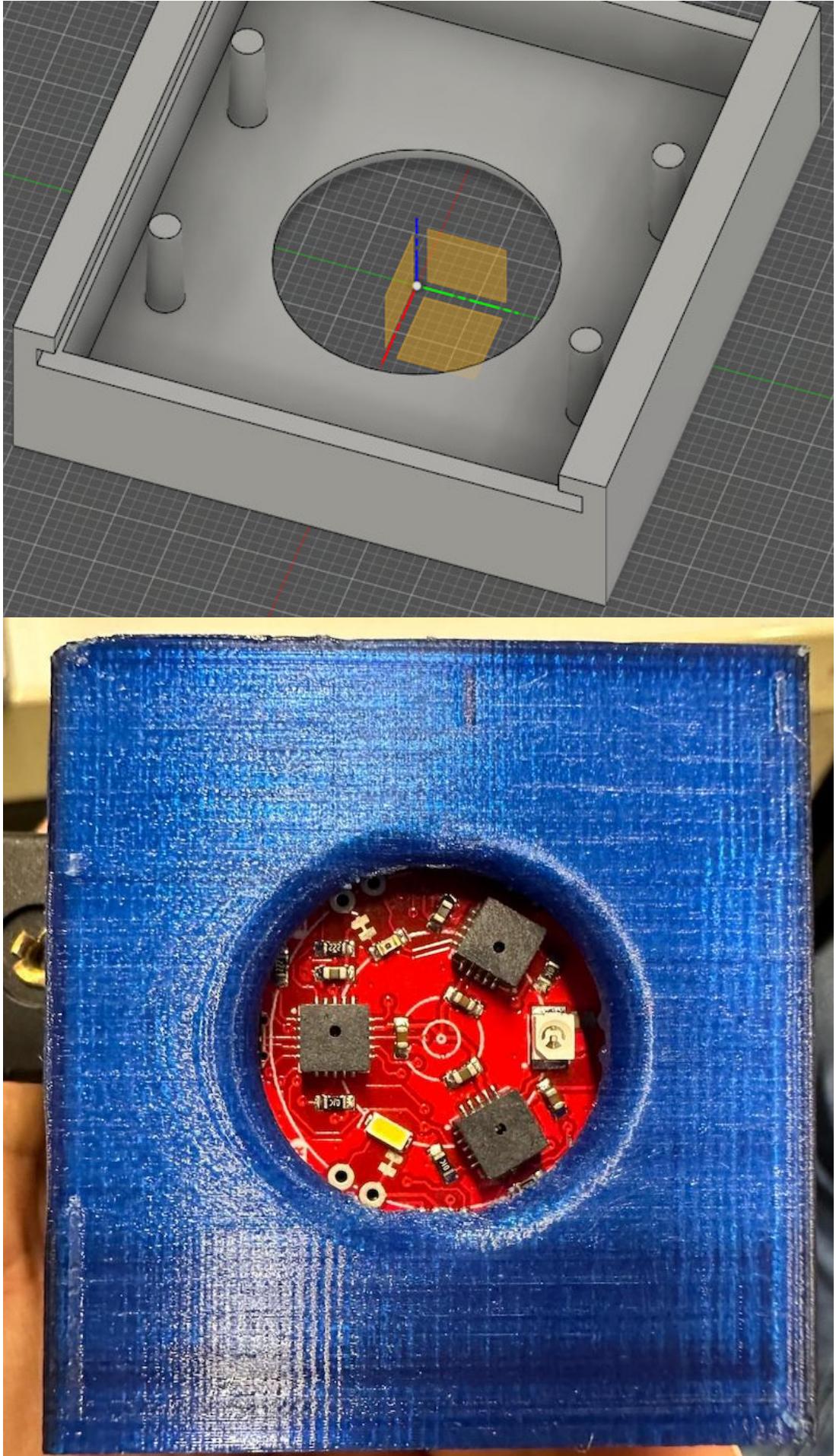




Das Ganze wird mit einer um 90 Grad abgewinkelten Kappe verschlossen, welche etwas komplizierter als eine flache Abdeckkappe ist. Die abgewinkelte Kappe bietet aber einen guten Zugang zum Breadboard, insbesondere zur ESP-Seite. Des weiteren besitzt sie zwei Löcher um die USB-Anschlüsse des ESPs sowie des LiPo-Chargers von außen zu erreichen und einen Schlitz, um sie am Ende des Breadboards zu befestigen. Jedoch war der Schlitz zu dünn, um wirklich als Befestigung zu funktionieren und stellte eher eine Schwachstelle als Bruchstelle dar. Die Abdeckkappe wurde nicht

U-förmig gestaltet, da sie sich durch Druck verformen könnte und kein ständiger leichter Zugang zur nicht-ESP-Seite erforderlich ist. Im weiteren würde die Breite des Gehäuses durch eine weitere Fuge zunehmen.



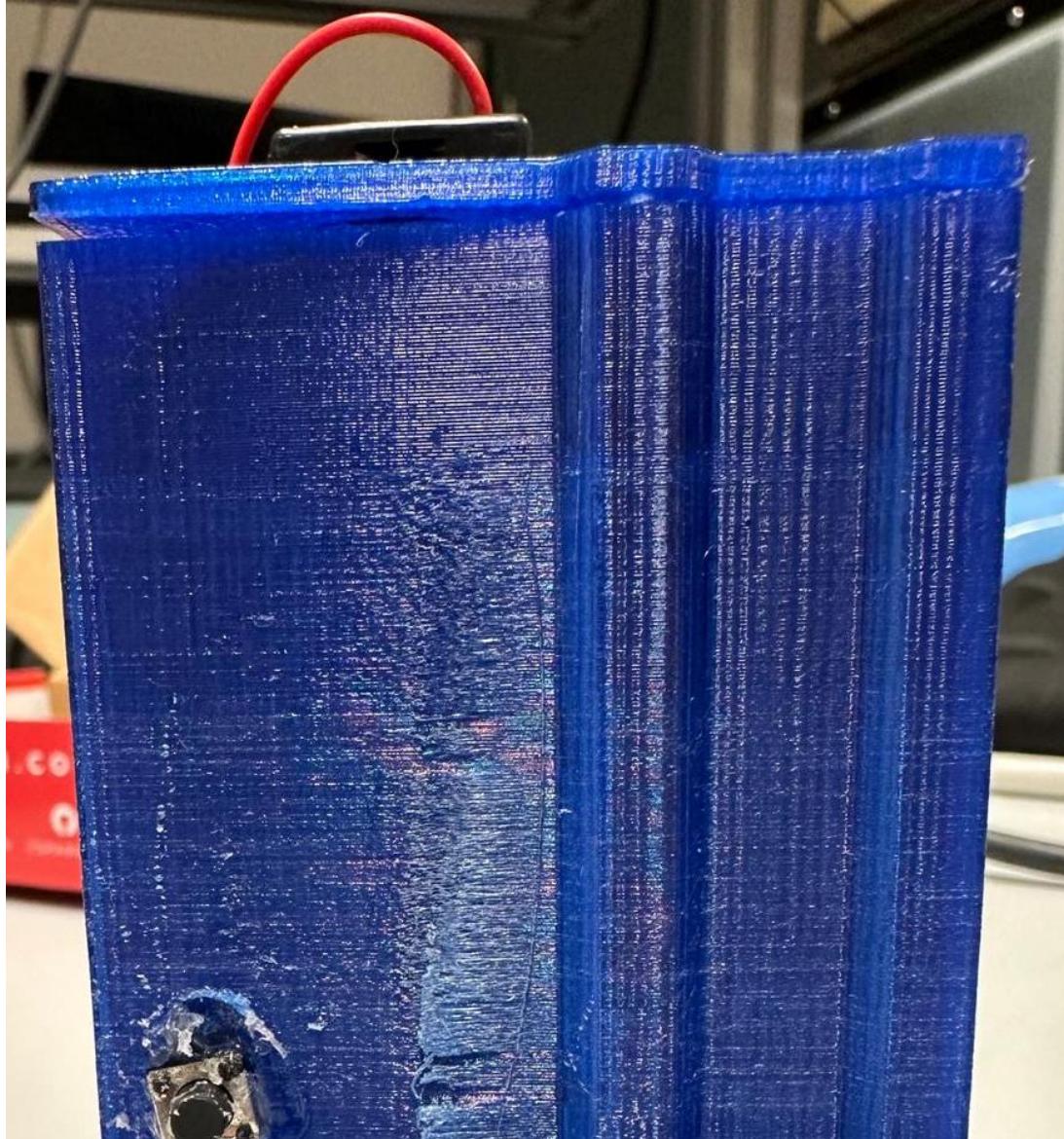


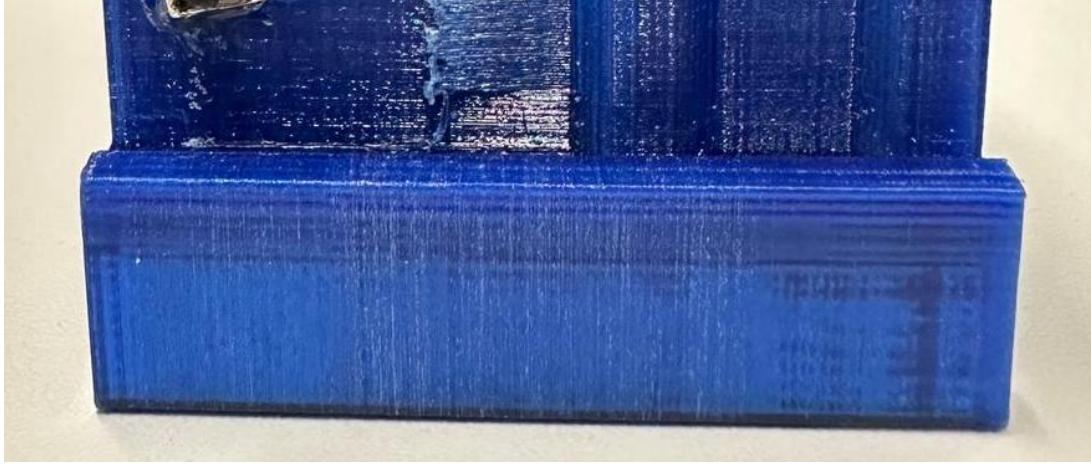
Indem der AS7265X weiter seitlich zur Kabelöffnung positioniert wird, kann der Lichteinfall von oben reduziert werden. Dies könnte aufgrund der asymmetrischen Front jedoch zu Lichteinfall von unten führen. Da das Problem des Lichteinfalls von oben leicht mit einem Stück undurchsichtigem Klebeband über der Kabelöffnung behoben werden kann, wurde der Sensor mittig im Gehäuse gelassen.

Bei der Montage wurde festgestellt, dass die zum AS7265X führenden Kabel durch die Kabelöffnung gezogen werden müssen, nachdem sie die Seitenwand dessen Gehäuses durchquert haben. Dies lässt sich am besten durch den Wechsel zu einer Schraubbefestigung beheben.

Die Batterie wurde an der Seite angebracht, um die gedruckten Teile gering zu halten, und mit einem Schalter zum Ein- und Ausschalten des Geräts versehen. Auf der anderen Seite befindet sich ein Taster, mit dem ein Scavorgang ausgelöst wird.







Kurz gefasst würden wir bei der nächsten Version des Gehäuses auf ein Schraubsystem umsteigen, da es etwas Platz spart und das Ratespiel bei der Entscheidung über die Abstände zwischen den verschiebbaren Teilen eliminiert. Auch das Problem der Kabelführung wäre damit gelöst. Als kleine Änderung kann das untere Gehäuse um weitere 3-4mm abgesenkt werden, wenn die Stifte im Durchmesser um 0,5mm dünner gemacht werden, so dass der Sensor direkt an der Öffnung anliegt. Außerdem können die Löcher für den Batterieschalter, die Batterieverkabelung und die Taste für die Abtastung vorgefertigt werden, um eine bessere Passform zu erreichen.

Grafana automatisieren

Provisioning (Automatisierte Konfiguration)

Das wiederholt neue Einrichten wollen wir uns durch das Provisioning (dt. bereitstellen) ersparen. Zunächst ist es aufwendiger, aber folgend kann schnell ein weiteres System (z.B. Lokal zum Testen) aufgesetzt werden.

- Dokumentation: [Provision dashboards and data sources](#)

Datasources bereitstellen

Um eine Datenbank bereitzustellen, wird unter dem Container Pfad `/etc/grafana/provisioning/` einen Ordner `datasources` angelegt.

Beachte, dass dies durch Docker nicht dem Pfad auf dem Host-System

entspricht! Hier können die Konfigurationsdateien der Datenquellen im yaml Format abgelegt werden. Jede Konfigurationsdatei kann eine Liste von Datenquellen enthalten, die während des Starts von Grafana hinzugefügt oder aktualisiert werden.

Beispiel Konfigurationsdatei:

```
yaml
apiVersion: 1

datasources:
  - name: InfluxDB_v2_Flux_Spectral
    type: influxdb
    access: proxy
    url: http://influxdb:8086
    secureJsonData:
      token: <INFLUXDB_TOKEN>
    jsonData:
      version: Flux
      organization: <INFLUXDB_ORG>
      defaultBucket: <INFLUXDB_BUCKET>
      tlsSkipVerify: true
```

Weitere Informationen zum Datasource Provisioning und speziell zur InfluxDB kann in der Dokumentation unter den folgenden Links gefunden werden:

- Dokumentation: [Provisioning data sources](#)
- Dokumentation: [Provisioning InfluxDB](#)

Dashboards bereitstellen

Ähnlich wie zu den Datasources können auch die Dashboards im Vorhinein bereitgestellt werden, indem unter dem Pfad `/etc/grafana/provisioning/` einen Ordner `dashboards` angelegt wird. Erstelle in diesem Ordner ein Dashboard Provider Konfigurationsdatei (`dashboard.yaml`), um Dashboards beim Start von Grafana laden zu können.

yaml

```
apiVersion: 1

providers:
  - name: 'SpectralSensor'
    orgId: 1
    folder: ''
    type: file
    disableDeletion: true
    updateIntervalSeconds: 10
    allowUiUpdates: true
    options:
      path: /etc/grafana/provisioning/dashboards
      foldersFromFilesStructure: true
```

Unter `path` wird der Pfad angegeben, wo die Dashboard yaml Dateien abgelegt werden sollen. In unserem Fall sind diese im selben Ordner wie diese Dashboard Provider Konfigurationsdatei.

Um ein Dashboard durch Provisioning aufzusetzen, kann das zuvor auf der Grafana Weboberfläche erstellte Dashboard als JSON Datei exportiert und unter dem angegebenen `path` abgelegt werden. Beim erneuten Start von Grafana wird nun das Dashboard automatisch erstellt.

- Github Link: [Dashboard-Dateien](#).
- Dokumentation: [Provisioning Dashboards](#)

Finale Version des Codes

Im Folgenden soll der finale Code näher erläutert werden. Alle in Stage 1 erklärten Funktionen wurden hier benutzt. Der Code ist in zwei Blöcken aufgeteilt: einer `setup()`-Funktion und einer `loop()`-Funktion.

Zunächst werden alle verwendeten Bibliotheken initialisiert und die Anmeldedaten für die WiFi Verbindung gesetzt.

Hier ist zu beachten, dass die eigenen WiFi-Hotspot-Daten im Code gesetzt

werden müssen. Ansonsten kann keine Verbindung zum ESP32 hergestellt werden!

Anschließend werden die benötigten Credentials für die InfluxDB gesetzt. Ähnlich wie bei Grafana werden folgende Daten der Datenbank benötigt:

- Eine **URL** des Datenbank Servers
- Ein **Token**, zur Authentifizierung mit der InfluxDB
- Der **Bucket-Name**
- Unser **Organisationsname**

Vor dem Setup, werden zunächst die verwendeten Objekte zur Interaktion der InfluxDB, des Sensors und für die WiFi Verbindung initialisiert. Außerdem muss die Zeitzone, für die Synchronisation der Zeitzone, gesetzt werden.

cpp

```
// Initialize WiFi client
WiFiMulti wifiMulti;

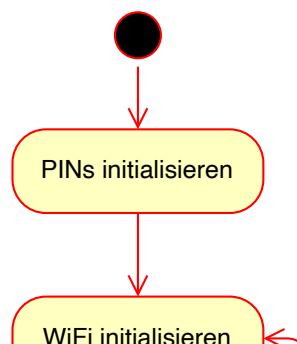
// Initialize specSensor
AS7265X specSensor;

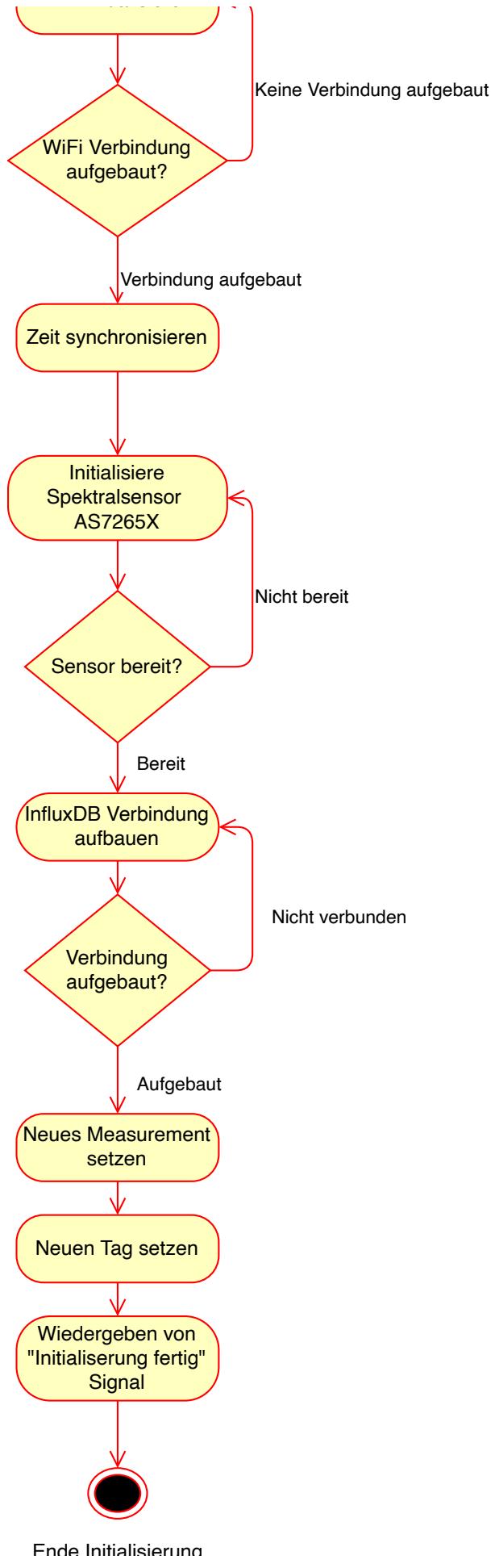
// Initialize InfluxDB client
InfluxDBClient client(INFLUXDB_URL, INFLUXDB_ORG, INFLUXDB_BUCKET);

// Declare the data point for the measurements
Point *sensorData = NULL;
```

Funktion: setup()

Start Initialisierung





Ende Initialisierung

Im Setup werden zunächst die PIN-Modi gesetzt. Für den Button wird hier auf PIN 4 ein INPUT gesetzt, mit internem Pullup-Widerstand. Zusätzlich wird diesem PIN ein Interrupt zugewiesen, welcher ausgeführt wird, wenn der Button gedrückt wird. Anschließend wird der Buzzer-PIN gesetzt.

cpp

```
// AS7265X
pinMode(4, INPUT_PULLUP);
attachInterrupt(4, setMeasurementFlag, FALLING);

// Buzzer
pinMode(13, OUTPUT);
```

Anschließend wird die WiFi Bibliothek und die I²C Bibliothek für den Spektral-Sensor initialisiert. Zusätzlich wird die Zeit, zur Erstellung von akkuraten Zeitstempeln für die InfluxDB, synchronisiert und die Status-LED ausgeschaltet.

cpp

```
// Setup wifi
WiFi.mode(WIFI_STA);
wifiMulti.addAP(WIFI_SSID, WIFI_PASSWORD);

Serial.print("Connecting to WiFi");
while (wifiMulti.run() != WL_CONNECTED)
{
    Serial.print(".");
    delay(100);
}
Serial.println();

// Sync time
timeSync(TZ_INFO, "pool.ntp.org", "time.nis.gov");

if (specSensor.begin() == false)
{
    Serial.println("specSensor does not appear to be connected.");
    while (1)
```

```
        ;  
    }  
    specSensor.disableIndicator();
```

Im letzten Schritt des Setups, wird die Verbindung mit der InfluxDB aufgebaut. Falls diese besteht, werden drei kurze Töne über den Buzzer wiedergegeben, mit einem Delay von 300ms zwischen den Tönen. Wir haben hier die Delay-Funktion verwendet, da hier ein abgesonderter Codeabschnitt besteht. Somit blockiert zwar die Delay-Funktion den Mikrocontroller, da die setup()-Funktion im Anschluss endet und danach die loop()-Funktion folgt, ist dies ohne negative Folgen bezüglich der Performanz zu benutzen.

Zuletzt werden hier die **setNewTag()** und **setNewMeasurement()**-Funktionen aufgerufen. Diese werden für die jeweilige Messung benötigt. Die Funktionsweise kann aus den Abschnitten [Funktion: setNewTag\(\)](#) und [Funktion: setNewMeasurement\(\)](#) entnommen werden.

Für das erfolgreiche Abschließen der Funktionen **setNewTag()** und **setNewMeasurement()**, muss der ESP mit einem seriellen Monitor verbunden werden. Hierfür kann das Tool HTerm benutzt werden. Beim Verwenden muss der richtige COM-Port ausgewählt werden (z.B. im Geräte-Manager unter Windows), sowohl als auch die Baud-Rate von **115200**.

cpp

```
// Check server connection  
if (client.validateConnection())  
{  
    Serial.print("Connected to InfluxDB: ");  
    Serial.println(client.getServerUrl());  
    Serial.println("Please initialize a measurement and tag name  
    setNewMeasurement());  
    setNewTag();  
    digitalWrite(13, HIGH);  
    delay(300);  
    digitalWrite(13, LOW);  
    delay(300);  
    digitalWrite(13, HIGH);  
    delay(300);
```

```
    digitalWrite(13, LOW);
}
else
{
    Serial.print("InfluxDB connection failed: ");
    Serial.println(client.getLastErrorMessage());
}
```

Funktion: setNewTag()

cpp

```
void setNewTag()
{
    Serial.println("Entered mode to set a new tag. Please enter th
    // Stay in mode, until the new tag is set
    while (1)
    {
        if (Serial.available())
        {
            String currentString = Serial.readString();
            if (currentString == "exit")
            {
                Serial.println("Exiting set new tag mode.");
                return;
            }
            else
            {
                currentTag = currentString;
            }
        }
    }
}
```

Zunächst wird eine Ausgabe im seriellen Monitor ausgegeben. Im Anschluss wird darauf gewartet, dass der User seinen neuen Tag-Namen eingibt. Dieser ist nötig, um in der jeweiligen Measurement das richtige Objekt identifizieren zu können, wenn

danach gequeried/gesucht werden soll in der InfluxDB.

Nachdem der gewünschte Name eingegeben wird, kann durch das eingeben des Wortes "exit" die Funktion verlassen werden.

Funktion: setNewMeasurement()

cpp

```
void setNewMeasurement()
{
    Serial.println("Entered mode to set a new measurement. Please enter the measurement name.");
    if (sensorData != NULL)
    {
        delete sensorData;
    }
    // Stay in mode, until the new measurement is set
    while (1)
    {
        if (Serial.available())
        {
            String currentString = Serial.readString();
            if (currentString == "exit")
            {
                Serial.println("Exiting set new measurement mode.");
                return;
            }
            else
            {
                sensorData = new Point(currentString);
            }
        }
    }
}
```

Die **setNewMeasurement()**-Funktion funktioniert ähnlich wie die **setNewTag()**-Funktion. Hier wird auch zunächst eine Ausgabe im seriellen Monitor ausgegeben. Im Anschluss wird der neue Measurement-Name erwartet/eingelesen. Dieser wird

benötigt, um in der InfluxDB ein neues Measurement anzulegen oder auszuwählen, welches eine neue Klasse an Objekten entspricht, um die Daten zu speichern.

Hier haben wir jedoch eine Schwierigkeit aufdecken können:

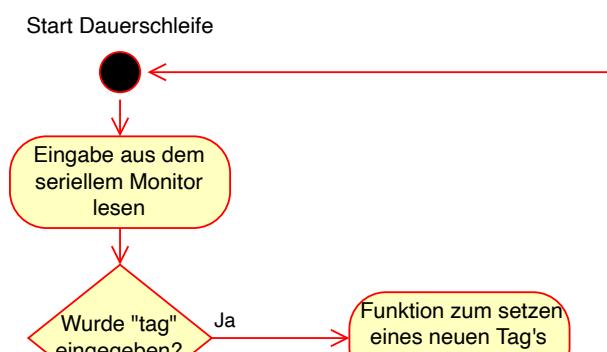
Um mit der InfluxDB zu interagieren, wird ein "Point-Objekt" benötigt. Dieses Point-Objekt wird mit einer Measurement initialisiert, welcher beim Aufrufen des Konstruktors als Parameter mitgegeben wird. Die Herausforderung war es hier eine Möglichkeit zu finden, die Point-Objekte dynamisch zu erstellen. Dadurch wurde ermöglicht, dynamisch die gewünschte Klasse an zu messenden Objekten nach belieben zu ändern. Die Bibliothek dafür sieht dieses Vorhanden jedoch nicht vor. Deshalb stehen zwei Optionen zur Verfügung:

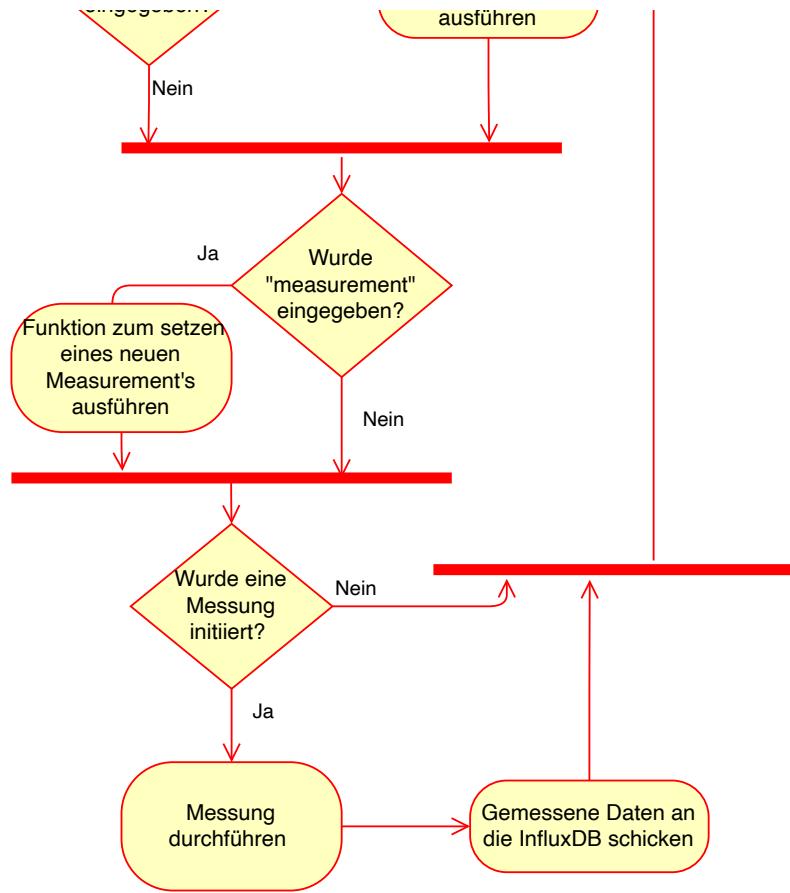
- Die Bibliothek mit einer passenden Funktion ergänzen oder
- einen anderen Workaround im vorhandenen Code finden.

Wir haben uns dann entschlossen, einen Pointer vom Typ Point zu erstellen, stattdessen das Point-Objekt initial zu erstellen. Dann haben wir einen initialen Aufruf der in der **setup()**-Funktion, vor dem eigentlichen Messen, implementiert. Dadurch wird sichergestellt, dass zu Beginn der Messung immer Tag und Measurement gesetzt sind. In der **setNewMeasurement()**-Funktion löschen wir dann das jeweilige Point-Objekt, falls es nicht **NULL** ist und erstellen auf dem Heap ein Neues, durch aufrufen mit "**new**". Dadurch erhalten wir einen Pointer, mit dem wir in der **takeMeasurement()** unsere Daten an die InfluxDB schicken können. Dieses Vorhaben war nötig, da wir die einzelnen Funktionalitäten gekapselt haben wollten, somit der Messvorgang und das Senden der Daten an die InfluxDB von allem anderen getrennt sind.

Wenn der neue Name eingegeben wurde, kann auch hier mit "exit" die Funktion verlassen werden.

Funktion: **loop()**





Die Loop-Funktion beinhaltet den Code welcher für die Messung benötigt wird. Getriggert wird eine Messung durch das betätigen des Buttons. Dadurch wird die Interrupt-Service-Routine (ISR) ausgelöst, welche eine Messung, durch setzen eines Flags, initiiert.

cpp

```

void setMeasurementFlag()
{
    takeMeasurement = 1;
}
  
```

Zunächst wird ein Signal über den Buzzer ausgegeben, um eine beginnende Messung zu signalisieren. Dieser Ton ertönt solange, wie die Messung andauert. Die Messung wird im Anschluss über die **takeMeasurementsWithBulb()**-Funktion initiiert. Im Anschluss werden die einzelnen Kanäle über die **getCalibratedX()**-Funktion ausgelesen, wobei das X für den jeweiligen Buchstaben des auszulesenden Kanals steht. Bevor die Daten in das Point-Objekt Daten gespeichert werden, werden die zuvor gemessenen Daten (Felder und tags) gelöscht. Die jeweiligen gemessenen Werte werden dann dem Point-Objekt übergeben. Im Anschluss wird geprüft, ob die WiFi-

Verbindung noch aktiv ist, falls nicht, wird die Nachricht "WiFi connection lost" im Terminal ausgegeben.

Mit der `writePoint()`-Funktion des Clients, werden die Daten schlussendlich an die InfluxDB gesendet.

cpp

```
void IRAM_ATTR takeSensorData()
{
    digitalWrite(13, HIGH);
    Serial.println("Begin measurement....");
    specSensor.takeMeasurementsWithBulb(); // This is a hard wait !
    digitalWrite(13, LOW);
    sensorData->clearFields();
    sensorData->clearTags();
    sensorData->addField("410nm", specSensor.getCalibratedA());
    sensorData->addField("435nm", specSensor.getCalibratedB());
    sensorData->addField("460nm", specSensor.getCalibratedC());
    sensorData->addField("485nm", specSensor.getCalibratedD());
    sensorData->addField("510nm", specSensor.getCalibratedE());
    sensorData->addField("535nm", specSensor.getCalibratedF());
    sensorData->addField("560nm", specSensor.getCalibratedG());
    sensorData->addField("585nm", specSensor.getCalibratedH());
    sensorData->addField("610nm", specSensor.getCalibratedR());
    sensorData->addField("645nm", specSensor.getCalibratedI());
    sensorData->addField("680nm", specSensor.getCalibratedS());
    sensorData->addField("705nm", specSensor.getCalibratedJ());
    sensorData->addField("730nm", specSensor.getCalibratedT());
    sensorData->addField("760nm", specSensor.getCalibratedU());
    sensorData->addField("810nm", specSensor.getCalibratedV());
    sensorData->addField("860nm", specSensor.getCalibratedW());
    sensorData->addField("900nm", specSensor.getCalibratedK());
    sensorData->addField("940nm", specSensor.getCalibratedL());
    sensorData->addTag("Object name", currentTag);

    // Check WiFi connection and reconnect if needed
    if (wifiMulti.run() != WL_CONNECTED)
    {
        Serial.println("WiFi connection lost");
    }
}
```

```
// Write point
if (!client.writePoint(*(sensorData)))
{
    Serial.print("InfluxDB write failed: ");
    Serial.println(client.getLastErrorMessage());
}
else
{
    Serial.print("Done sending data!");
}
```

◀ PREVIOUS

Stage 2

NEXT ▶

Offene Aufgaben

Möglichkeiten der Weiterentwicklung

Spektral-Sensor kalibrieren

Wir haben uns zu Beginn des Projektes überlegt, den Sensor zu mit definierten Farben zu kalibrieren. Diese wären uns von Professor zur Verfügung gestellt worden, um z.B. zu messen, was "reines Weiß" für ein Spektrum besitzt. Die gemessenen Daten sollten dann in ein eigenes Dashboard visualisiert werden, um damit z.B. Vergleiche mit anderen Messungen durchzuführen.

Weitere Messsysteme aufbauen

Wir haben uns auch überlegt weitere Messsysteme aufzubauen, welche nicht innerhalb eines Stempels realisiert werden. Für die gemessenen Daten müsste im Code eine neuer Tag erstellt werden, um die ESPs voneinander unterscheiden zu können. Damit könnten die gemessenen Daten zwischen unterschiedlichen Messsystemen verglichen werden

Eigenschaften ableiten

Eine weitere Möglichkeit wäre es aus den Datensätzen Eigenschaften zu erheben. Damit könnten dann anschließend Aussagen über neue Messungen getroffen werden.

◀ PREVIOUS

Stage 3

NEXT ▶

Nachwort

Nachwort

Zusammenfassend, wir sind überzeugt, dass das Projekt ein voller Erfolg geworden ist. Die selbst gesteckten Ziele konnten übertroffen werden. Wir haben ein portables Messsystem in handlicher Form hergestellt. Dieses System ermöglicht es uns, Messdaten von Objekten zu erfassen und diese durch einen Hotspot automatisiert an eine Datenbank in der Cloud zu schicken. Diese Messdaten, welche das Lichtspektrum der Objekte beinhalten, können für den Benutzer jetzt schon visualisiert dargestellt werden!

Eine Weiterentwicklung des Projektes ist jederzeit möglich. Es könnte eine automatisierte Analyse der Messdaten erstellt werden welche in ein Aktuatoriksystem zurückgespeist, beispielsweise unreife Bananen von reifen aussortiert.

Wir lernten im Team unsere Stärken zu nutzen und organisierten uns immer besser. Wir konnten sehr viel über die Funktionsweise des Sensors lernen, als auch wie die Reflektometrie funktioniert. Außerdem haben wir neue Tools kennengelernt (z.B. Fusion360 für das Gehäuse, Docsify für die Erstellung von strukturierter Dokumentationen). Das Arbeiten im Team wie auch das strukturierte, zielorientierte Arbeiten in diesem Projekt wird für unser Berufsleben als Software-Engineer sicherlich hilfreich sein.

◀ PREVIOUS

Offene Aufgaben

NEXT ▶

Quellen

Quellen

- [Dokumentation des AS7265X](#)

Bachelor Thesis

- [Github: SpectralSensor](#)
 - [PDF: Entwicklung und Realisierung einer Messeinrichtung mit den Sensoren AS7261 und AS72651 von ams](#)
-

◀ PREVIOUS

Nachwort