eNose

An electronic nose for solid stools detection
Software perspective

Maxim Chin-On
Mark Fijneman
Gabriel Yousef

Technische Universiteit Delft

eNose

An electronic nose for solid stools detection
Software perspective

by

Maxim Chin-On
Mark Fijneman
Gabriel Yousef

A thesis presented for the degree of BSc Electrical Engineering

Project duration:	April 12, 2021 – June 30, 2021	
Thesis committee:	Prof. dr. ir. R.E. Kooij,	TU Delft
	Prof. dr. P.J. French,	TU Delft, supervisor
	Dr. M. Mastrangeli,	TU Delft, supervisor
	M. Gravemaker,	Momo Medical

TU Delft

Abstract

This report details the software part of the development process of the eNose technology. The technology is posed by Momo Medical. Momo Medical is a start-up company located in Delft, it provides and develops non-intrusive monitoring systems in the nursing sector. The project is the next step in an already existing product: BedSense. BedSense enables nurses to check for among others decubritus, whether the patient is out of bed, and even if the patient has passed away. The finished project will be able to detect solid stool and hence, when integrated into the system of BedSense, will greatly assist the nurses.

The eNose technology is able to detect solid bowel movement using its gas sensing abilities. It consists of gas sensors that detect the relevant gasses that are related to feces. These sensor values are then fed into an algorithm that is able to interpret them and detect defecation. Besides it includes a communication system that handles the internal and external communication. Finally, the technology supports Over The Air (OTA) updates which allows to update the firmware of the devices remotely.

The final prototype functions accurately in certain restrooms and can be regarded as a proof of concept. However more work and data is needed in order to make the eNose work in various environments, with possible integration of machine learning analysis, as it has showed great potential.

The project is executed in two groups, hardware and software. This report contains only the software part of the process. It includes the development of the detection algorithm. And the development of a communication and OTA programming system.

Preface

This thesis is written for the Bachelor Graduation Project of the study Electrical Engineering. The subject of the project has been proposed by Momo Medical. The company was closely involved in the execution of the project. As the results of the project is a fully functional prototype, the eNose will be tested inside care facilities soon after the deadline of the thesis. The conclusions and results of the resting will be presented during the defence of the thesis.

We would like to sincerely thank Menno Gravemaker, CEO of Momo Medical and Thomas Bakker, CTO of Momo Medical, for giving us the opportunity to work on this hands-on project, challenging us and supporting us.

Not to mention are we grateful for the excellent guidance and support provided by our supervisors, prof. dr. Paddy French and dr. Massimo Mastrangeli. The quick feedback and critical question were of immense value.

Last but not least, we want to acknowledge the other subgroup's members: Niels van Damme, Stein Fakkel and Anne Stijns. We enjoyed and valued the close and effective collaboration during the project.

Maxim Chin-on
Mark Fijneman
Gabriel Yousef
Delft, 17 June 2021

Contents

1 Introduction

As part of the bachelor end project of the study electrical engineering of the TU Delft, a team of future electrical engineers chose to work on a technical challenge posed by Momo Medical[1]. Momo Medical is a start-up company located in Yes!Delft that provides and develops non-intrusive monitoring systems in the nursing sector. The project is the next step in an already existing product: BedSense. BedSense is a system that exists out of an under-the-mattress sensor and a server application. It enables nurses to check for decubitus, whether the patient is out of bed, and even if the patient has possibly passed away.

A common problem of incontinent elderly people is incontinence-associated dermatitis, diaper rash in layman terms, caused by prolonged exposure to the moisture of stools and urine combined with changes in pH [1]. Currently, nurses routinely inspect the diaper of patients throughout the day and night for defecates. This is done either by smelling or checking for humidity with a hand placed inside the diaper. This method is time-consuming for the healthcare workers and disruptive for the patient. Hence, a method to detect feces would represent a great addition for the BedSense.

The goal of the project is to develop a non-intrusive device that can be eventually integrated into the BedSense system and can detect solid stool. The device should be easy for nurses to use and not burden them with extra work. To develop and implement this product, the project is split up into two parts. The first part relates to the sensors and sensor read-out. The second part relates to the detecting algorithm and communication. An overview of the project is given in Fig. 1.1. This report will specifically be about the second part

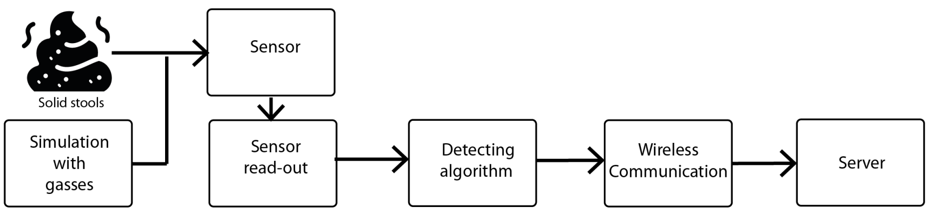


Figure 1.1: Flowchart of project components.

The detection algorithm depends on the data supplied by sensors and will be transmitted wirelessly to a server. Hence, the research question raised is: "What are effective methods related to extracting, interpreting, and communicating the stool sensor data?". To answer this question, this report will first discuss the specifications which are desired. The second chapter will discuss the algorithm part. The next chapter is about the communication between the eNose and BedSense. Inside the fourth chapter Over The Air (OTA) programming is explored for updating the device remotely. This thesis will finish with a conclusion and recommendations for future work.

[1] https://www.momomedical.com/

2
Specification

This chapter will introduce the requirements for the algorithm, communication part and Over The Air (OTA) programming. It will first reiterate the general requirements of the project and continues with specific requirements.

General requirements

The main requirement of the eNose is to be able to correctly distinguish solid stools from other odors and smells. On top of that, both the eNose and the BedSense, are mainly used by healthcare workers for patients that require nightly care. Both devices aim to reduce the workload. Health care workers experience a high workload and are generally not technically schooled. Hence, the eNose should be easy to use for the nurses and not burden them with extra technical work. As the eNose is monitoring all the time, the device should be as non-intrusive for the patient as possible. The price of a unit should be aimed around 20 euros, but it has an absolute maximum price of 300 euros per unit.

Detecting

One of the most important requirements for this project is that the device needs to be able to detect solid stools. An algorithm needs to do this based on different sensor values and should ignore all other types of smells and odors. To prove the working of the concept, the eNose will first be placed in different restrooms owned by members of the team, and if time allows for it, the device will also be tested in nursing homes.

Communication

For the development of the algorithm and for monitoring the devices, a large quantity of data is required. As the eNoses will be distributed among many rooms at different locations, the data should be collected at a central location automatically. An important factor to consider is the scale-ability of the server/database system as many devices will be connected eventually. Furthermore, a monitoring/-graphing tool should be used to make the data easily visible and understandable. For the actual usages by the health care workers, an SMS containing the location needs to be sent when solid stool has been detected.

Over The Air

As the end product will be used in many rooms over several care facilities, it can be difficult to perform updates and tweaks by physically accessing the devices. Hence, the devices should be update-able remotely. This includes installing a complete new firmware on the device as well as changing the parameters of the Wi-Fi.

3
Hardware

Literature shows that solid stools detection can be done using multiple techniques such as gas detection, moisture detection or ultrasound-based detection. [2] However based on the requirements, gas detection was chosen as it can solely detects solid stools and is not sensitive to urine, unlike moisture sensors. And it can happen remotely without touching the patients skin, which is the case with ultrasonic detection.[2]

The Sensors

Gas detection can happen through several chemical properties of gas. Hence there are various commercial gas sensors available, each with a different sensing mechanism. [3] There are two important metrics regarding gas sensors. The selectivity of a sensor, which specifies to which degree a sensor can detect one gas from another and the sensitivity of a sensor, which is indicator of the magnitude of output as a response to a certain gas. After conducting an extensive literature study about gas sensors, MQ sensors were chosen. MQ sensors are a well known and a cheap series of gas sensors. Since they were relatively cheap and available for delivery within a week, as quick delivery was an important requirement as well.

The MQ sensors are Metal Oxide Semiconductor (MOS) gas sensors. MOS sensors have a relatively simple sensing mechanism. Once these sensors are surrounded by the target gas, their resistance changes as a function of the partial pressure of the target gas. This gas interaction with the surface stimulates an electronic change in this oxide surface. This surface electronic change, translates into an electrical resistance. [4]

The MQ sensors are also provided with a heater, that regulates the internal temperature of the sensor. Following the chemical principle that different gasses show different optimal inter-action with the surface of the semiconducting material based on this surface temperature, this improves the sensitivity and the selectivity of the sensor. [5] This chemical property also explains the need for a warm-up before putting the sensors to work. This assures that the semiconductor has a sufficient temperature to ensure ideal sensing. The sensing degree also depends on the surrounding humidity, for this reason humidity should also be considered.

The main target gas for defecation detection is methane (CH_4). To detect methane, a methane sensitive MQ sensor is used. However, MQ sensors have a low selectivity and the sensor might react to other gasses that are exhaled by cleaning and sanitizers. For this reason, a non methane sensitive MQ sensor is used to filter out false detection caused by these gasses.

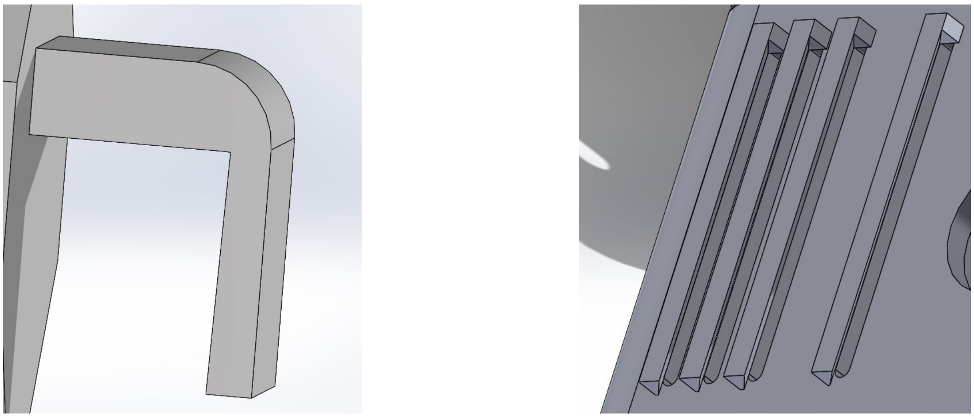
Other sensors included on the eNose are a temperature and humidity sensor, due to the dependence of the gas sensors on these properties as discussed prior. The sensor used for temperature and humidity is DHT22.

Microcontroller

The main core of the system consists of the microcontroller. It should have enough computational power and storage to support the software of the system and to make the necessary communication with the web-based part of it.
The BedSense device by Momo Technology uses the ESP32 as the internal microcontroller. The ESP32 supports Wi-Fi communication, and has built-in Analog to Digital Converters (ADCs), which are extremely useful in our case to transform the analog sensor readout to digital, without the need of an external ADC, The internal ADC pins of the ESP32 read the analog voltage signal and transform that into a 12 bits digital number, with a maximum value of 1.1 V.
The ESP32 is also characterized by having a low power consumption.
Based on these features the ESP32 was selected as the internal microcontroller of the eNose.

Hardware Integration

The final step was integrating the hardware. A PCB was designed to connect the hardware components. The focus while creating the final PCB design was the compactness of the total device. Finally, a case was designed to embrace all the electrical components. The case fulfilled two important design requirements: being water-resistant and having ventilation slots to allow passive cooling to stabilise the internal temperature. This was achieved by creating roof-like ventilation slots that allowed passive cooling and water resistance (as shown in Fig. 3.1b). The case also includes a bedclip to mount the eNose device on the bed of the patient (as shown in Fig. 3.1a). The gas sensors protrude out of the case to ensure optimal gas sensing. The final case design is illustrated in Fig. 3.2.



(a) The bed clip design. [2]

(b) The roof-like structure around ventilation holes.[2]

Figure 3.1: Features of the case design.

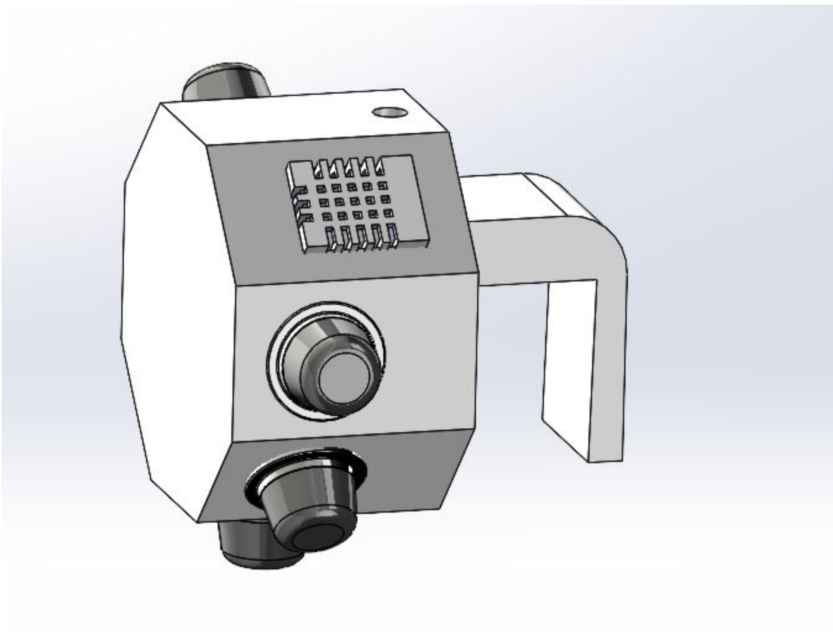


Figure 3.2: The final design of the case. [2]

4
Detection algorithm

The main purpose of the algorithm is to check whether the sensor values suggest solid stools rather than cleaning solutions, farts or other stimulants. These other stimulants also release gasses into the room affecting the sensor values. The device should be able to distinguish these other stimulants from feces, this is why the algorithm is needed. Whilst detecting feces, unwanted influences and signals can be present. To determine if this can cause problems, the significance of these influences will be examined. Next, relevant theories and approaches for the detection algorithm will be explained. The chapter will be concluded with the description of the implementation of the algorithm.

4.1. External influences

The sensors used are MQ gas sensors, these type of sensors can be affected by both temperature and humidity [6]. These influences could significantly change the sensor measurements and should be further examined. On top of that the sensors heat up by themselves, this results in a certain warm-up time until a stable output is reached, this could make detecting solid stools more difficult and should be looked into.

4.1.1. Temperature and Humidity

As discussed previously, the MQ sensors sensing ability is depending on temperature and humidity. The device is meant to be used in nursing homes. Climate control in the rooms only allows for small deviations in temperature and humidity. It is estimated that the sensor output voltages will not change by more than 5-10% due to these small deviations. This decision will be reevaluated if the temperature and humidity end up playing a bigger role, but for now this seems to suffice. This response is similar for all MQ sensors.

4.1.2. Warm-up time

The sensors have a heating element that makes sure that the sensor stays at a stable temperature because the sensors are temperature dependent [6]. This means that the sensors need a warm-up time. It was decided to plot the warm-up time. The warm-up time is defined as the time it takes for the sensor to reach a stable value. This is when the deviation between the last 10 sensor measurements is less than a percent. This definition was chosen because the stable value allows the algorithm to detect coming spikes correctly. In the first minute of the warm-up, the sensors output voltage spike after which the sensor voltage exponentially decrease to their stable output voltage (Fig. 4.1).

To better understand this process, the sensor warm-up time was measured for different cool-down periods in between measurements. The maximum temperature deviation for these data points was only three degrees Celsius and the relative humidity stayed quite stable. Because of this, these environment conditions were neglected. This resulted in Fig. 4.2. The warm-up time of the sensors seems to depend on the time it was offline. If the device had been offline for a longer period of time, the warm-up time would also increase in comparison to a shorter offline period.

As can be seen from the data, the warm-up time does not go over 10 minutes and is not expected to go over 10 minutes for longer cool-down periods. This value should give sufficient time for the sensor to heat up enough so it should not influence the detecting algorithm.

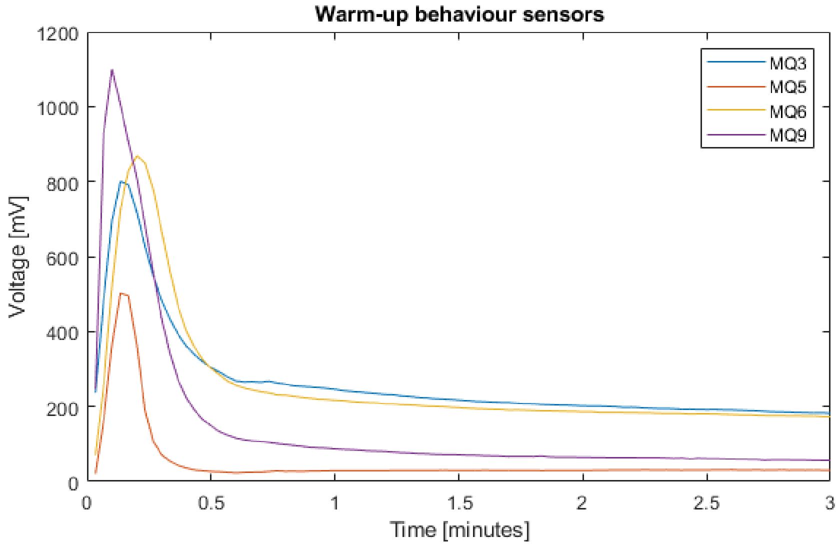


Figure 4.1: Plot showing the general warm-up behaviour of different sensors.

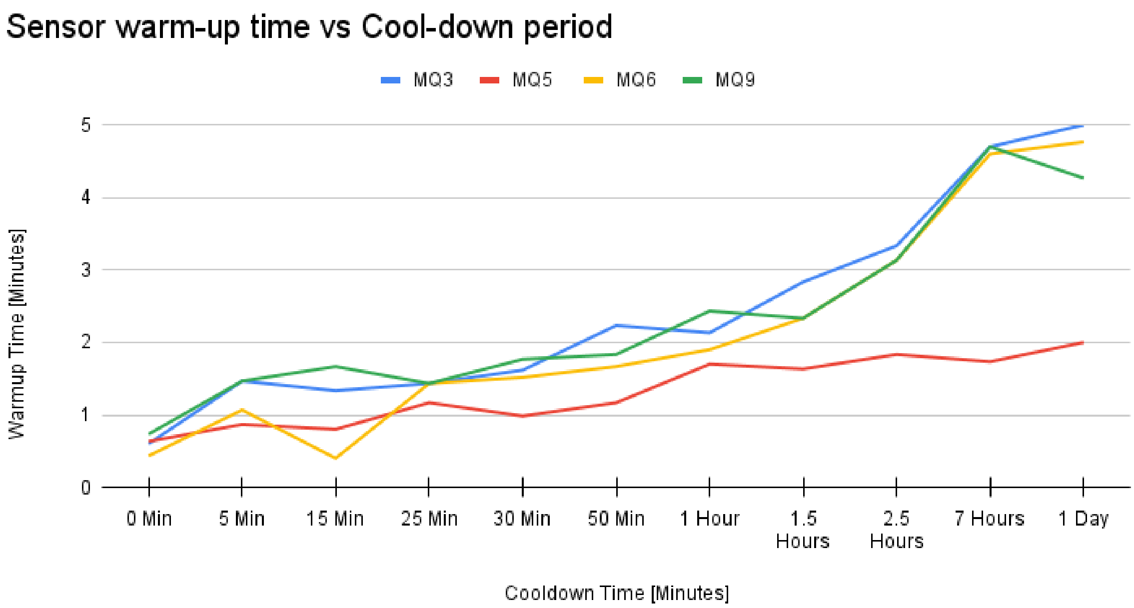


Figure 4.2: Warm-up time versus cool-down time.

4.2. Detection

Because the lack of literature on discriminating gasses released by feces from other stimulants, decisions needed to be made based on our own data. Data collection was started early in the project to improve the chances of creating a sufficiently accurate algorithm. This data was then analysed to find ways to differentiate between instances of stimulants.

All instances of feces and flatulence should consist of a similar concentration of gasses [7]. This means that the sensors measurements should react similarly every time they detect feces in a comparable environment. The MQ sensors should react differently compared to one another when exposed to gasses. By looking at these different changes of the sensor measurements when detecting feces and comparing these to each other, certain ratio's can be found. Because both small and large quantities of feces need to trigger the algorithm, the data needs to be normalized. The sensor to be used for the normalization is the MQ-3 sensor, this was done because when exposed to any gas its spike has the biggest increase of value. This makes it easier to neglect noise. This was first done with data collected from the first prototype board, this particular device was using the MQ-3 for normalizing MQ-5, MQ-9 and MQ-135. The resulting plot can be seen in Fig. 4.3. As can be seen in the figures, the feces boxplots

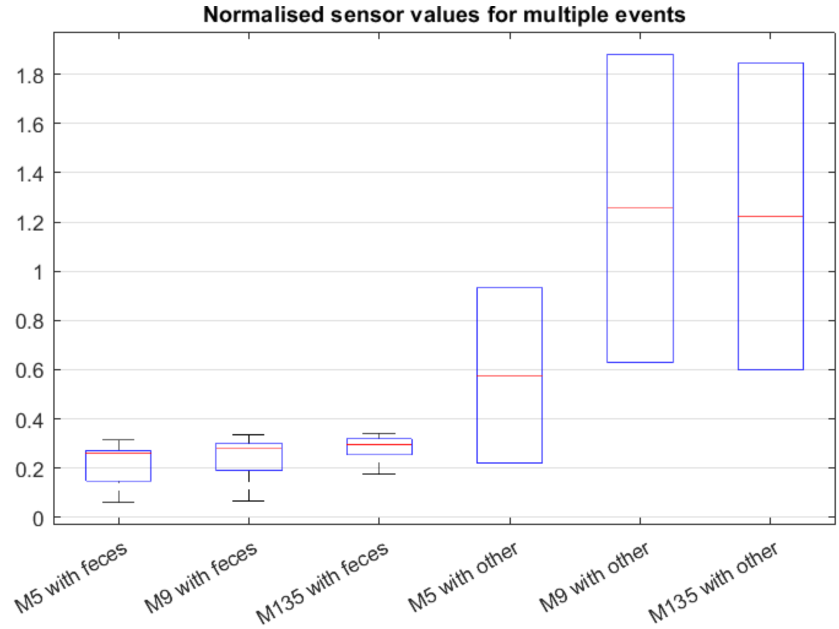
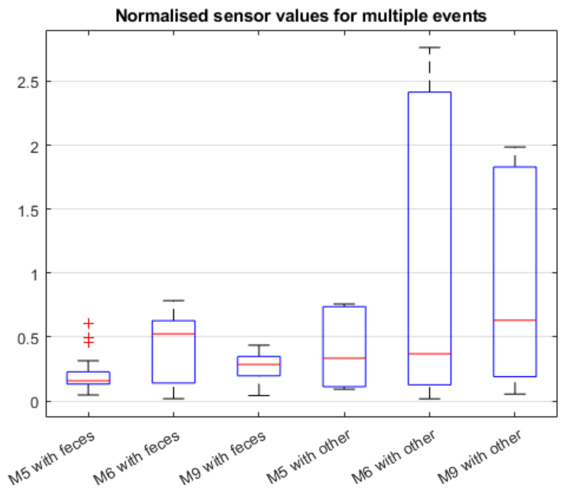
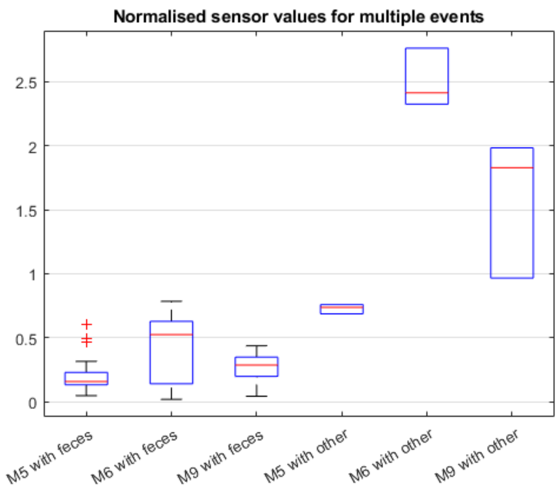


Figure 4.3: Normalized sensor behaviour plotted out in box plots.

do not overlap with the other boxplots, which is preferred. Both MQ-9 and MQ-135 show potential for detecting defecate data. The same plot was recreated for the second board, which uses the MQ-5, MQ-6 and MQ-9, and again uses MQ-3 for normalizing. Fig. 4.4a shows the box plots for the sensor measurements based on feces and other inputs. The boxplots here overlap, these sensor anomalies were results of defecation with an open restroom door. When the open door datapoints are taken out of the plot, Fig. 4.4b results.



(a) Normalized sensor behaviour plotted out in box plots shows the behaviour with open door data.



(b) Normalized sensor behaviour plotted out in box plots shows the behaviour without open door data.

Figure 4.4: Boxplots showing the normalized behaviour of different sensors

This data shows again that the MQ-9 sensor shows great potential for the detecting algorithm. This is because it is the only sensor here where the blue and red lines can be split using a value between the lowest blue line and the highest red line. However, even though the cases with the open door can be neglected for now, since the detecting algorithm is only applied in restrooms, it does show that bigger rooms could prove big challenges for this type of device.

4.2.1. Sensor Imperfection

Analysing the obtained data yielded further findings of the sensors. It was observed that the output voltages of the same type of sensors were not always equal. However, this was expected since the sensors are not calibrated before use. This phenomenon is illustrated in Fig. 4.5. In the figure, the readout of 4 MQ-135 sensors is plotted. The four sensors were placed next to each other and connected to a single ESP32 and placed above a toilet.

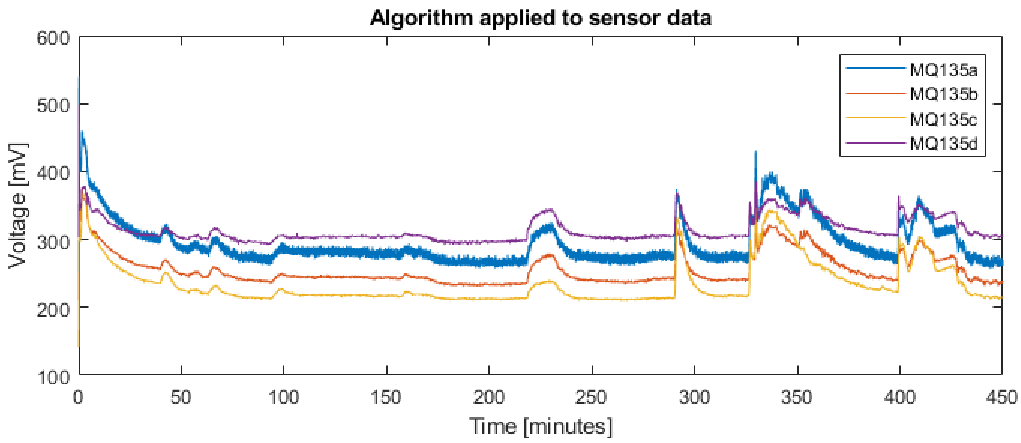


Figure 4.5: This plot shows four MQ-135 sensor, each with a slightly different output.

It can be seen that the mean of the sensors differs, but also the shape of some of the peaks is different. The difference between the means is 33% which could have large consequences. While creating the algorithm, the possibility of sensor calibration was still being investigated. This could mean that to ensure that all devices have the same detection precision, the algorithm has to be specifically designed for each device based on its sensors. The plots to choose the algorithm parameters in Fig. 4.4b and Fig. 4.3 have been made with three devices so far, so these devices are expected to accurately detect the poop. Other devices, however, do need to be tested before putting them into use to ensure selectivity.

4.2.2. Algorithm

The algorithm is written in C++ and runs on the ESP32. This makes it easier to scale as opposed to running the algorithm on a central system. This is because this way each device can make decisions on itself instead of the central system having to make decisions for each device. The algorithm works from the buffer class. This class stores the most recent sensor measurements and deploys the algorithm and a multitude of functions on the data. The complete C++ code fo the algorithm can be found in Appendix A.1. The algorithm is tested using collected data and plotting it in Matlab using the code in Appendix A.2.

The algorithm is initialized with `setBuffer`. This sets the variables including its maximum capacity, the warm-up time needed, sample rate (Fs), the period that should be considered and the detecting threshold. If these variables need to be changed during operation, they can be changed using `setValues`. Every time new data is collected, it is stored in `sensor1, sensor 2` and `sensor3` using `addData`. This function keeps track of the amount of data it has collected, and if the number surpasses its `capacity` it deletes the oldest datapoints. Every time data gets added to the buffer, the algorithm is applied to that data using `algo`. This function calls `getMean` to get the mean of the data. Using the mean, the stored data and the remaining variables, the algorithm makes a decision to either say defecate has been detected or not. The algorithm will be further explained. If the stored data is needed it can be requested using `getData` or printed using `printData`. Currently the buffer only stores data for three sensors. This is because the algorithm does not yet have the need for more. The class is easily extendable if more sensors are necessary to accurately detect feces.

Version 1

The first version of the algorithm worked according to a very simple principle. If a sensor value rises by enough in a short time, it would say it detected something. This algorithm proved to be able to detect defecation, and by keeping track of the duration of the value change, flatulence was neglected. However, this method also responds to any other change of gas concentration. Deodorant and a shower next doors proved to also be detected by the algorithm. To test the algorithm, it was applied on an already collected data set which included multiple instances of feces but also a spike caused by deodorant. The results of the algorithm can be seen in Fig. 4.6.

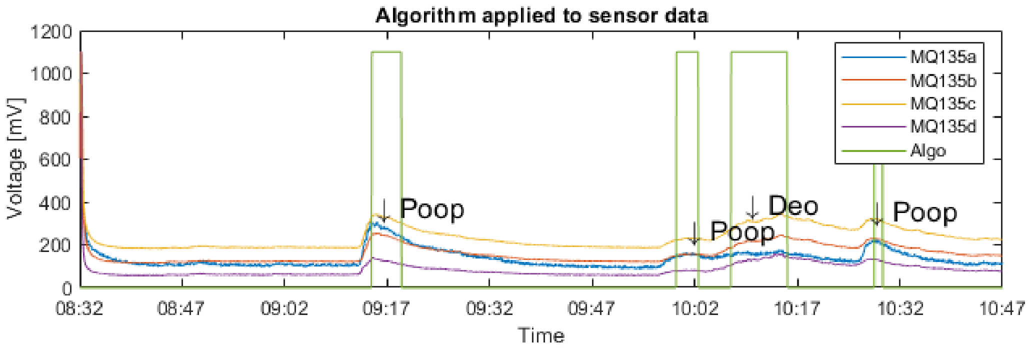


Figure 4.6: First version algorithm applied to dataset.

Although this version does indeed detect feces, its selectivity is way too low to be used in actual nursing homes. The next algorithm should focus on differentiating defecate from other stimulants.

Version 2

This version of the algorithm is an enhancement of the first version that uses the inputs from multiple sensors to help differentiate between different stimulants. It is based on the ratios shown by Fig. 4.3. A finite state model is used to implement the algorithm. The model can be seen in Fig. 4.7. This method features six different states. The starting state is the warm-up state, this state is only entered when booting up the device to ensure the device is warmed up before the algorithm is deployed. After the 10 minute mark has been reached it will continue to the neutral state. The neutral state is pretty straight forward, it waits until a spike is detected and if not it stays in the neutral state. If a spike is detected it goes to the 'Waiting for peak' state. This state waits for a peak to be reached, and when that happens it checks if the ratios are within the bounds explained above. The ratios that are considered in the bounds now only consists of the normalized value of MQ-9 being below 0.5. This number was chosen based on the plots shown in Fig. 4.3 and Fig. 4.4b. If the ratios are off or the peak takes too long to reach, the system waits in the cooldown state until the sensor values are normal again and then goes back to the neutral state. A normal sensor value is described as a sensor value that has or almost has returned to its mean. If the ratios are within the bounds it will go to the counter state. The counter state is to make sure small spikes such as flatulence are neglected. When the counter reaches a chosen number and the sensor value is still high, it will go to the detected state, otherwise it will return to neutral. In the detected state an output will be send, telling the system it has detected poop. The algorithm remains in this state until the signal returns to normal again. The ratios that are considered in the bounds now only consists of the normalized value of MQ-9 being below 0.5. This number was chosen based on the normalized sensor ratios shown in Fig. 4.3 and Fig. 4.4b

Using this algorithm on the same dataset as used to test the first version, Fig. 4.8 shows the results. The results look promising, the feces are correctly distinguished from the deodorant. However, as mentioned in the analysis section, when defecating with an open door the algorithm does not detect anything. This suggests that it will be more difficult to detect feces in a bigger room. To improve this algorithm more data needs to be collected. This data needs to be based on examining the effect of room size and ventilation.

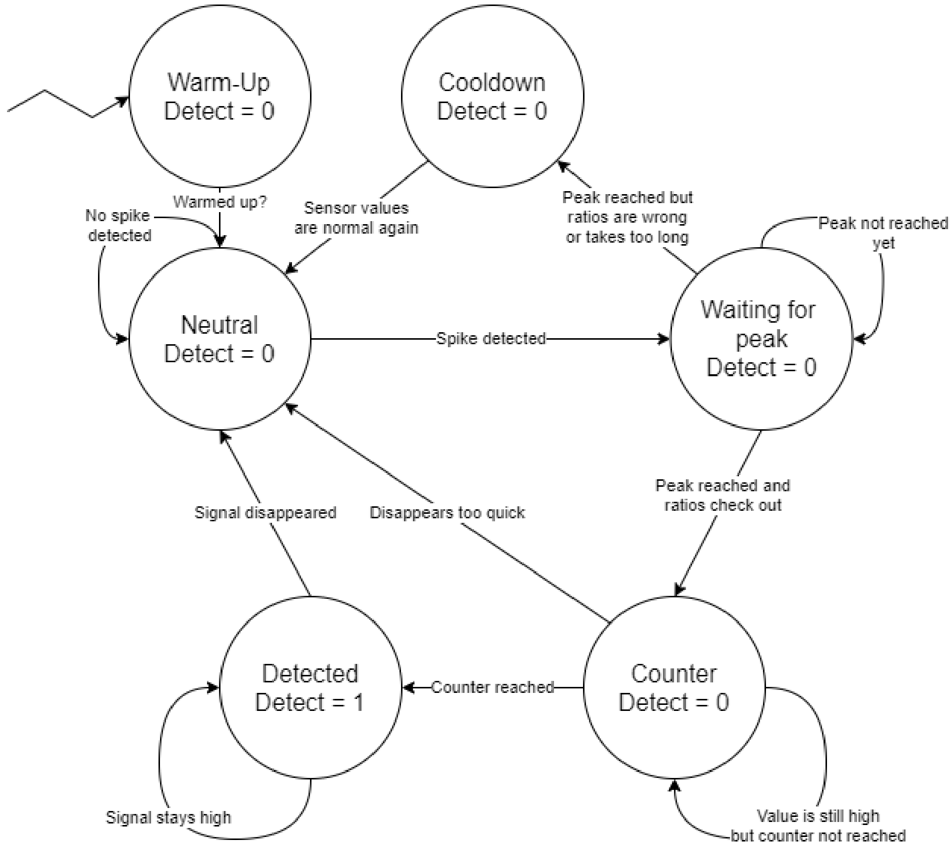


Figure 4.7: State diagram describing working of second version algorithm.

4.2.3. Machine Learning

On top of the standard algorithm, a decision was made to also explore the possibilities of machine learning (ML). This is because the ML algorithm is capable of identifying more patterns in the data provided. The algorithm was first coded in Python, but with existing libraries it can be put on the ESP32 in C++ code. This has not been done yet due to time constraints.

To train the ML algorithm, the already collected data was used. Out of the 7 devices with enough data points, 3 are chosen to train the algorithm. The remaining 4 are used to test the data. This is because the algorithm needs to work on new devices without having collected data for them. The ML is done with some extracted parameters. The datasets are divided in parts of 16 samples. The mean, standard deviation and difference between the first and last value are extracted for each of the parts and the dataset as a whole. Because the amount of non-poop datasets greatly outnumbers the amount of poop sets, they are equalized. This means that the poop datapoints are copied so there is an equal amount of poop as non-poop datasets.

Using the extracted parameters, the ML algorithm is trained. The results of this training process can be seen in Fig. 4.9a in a confusion matrix. The y-axis shows the true value of the dataset, and the x-axis the predicted value. The figures show that about 18% of the data containing no feces is detected as feces and 20% of feces data as non feces. The validation data results in an accuracy of 29% and 5% respectively. Although the majority of the datapoints is predicted correctly, these still are not great results. To improve the ML algorithm, an assumption was made. If feces are present, the gasses are not suddenly going to disappear. This means that if one datapoint detects feces, the next one should as well. Otherwise it is probably a wrong prediction. Using this assumption the algorithm was tested again. The results are shown in Fig. 4.10. Using this method, the accuracy improved to 0% and 5%

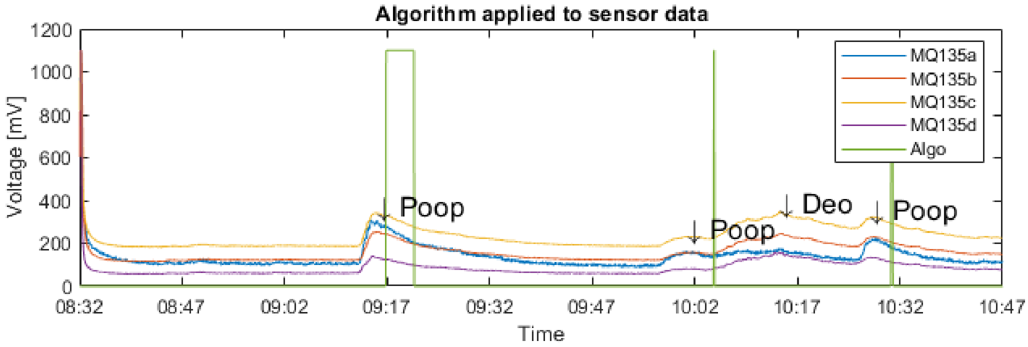
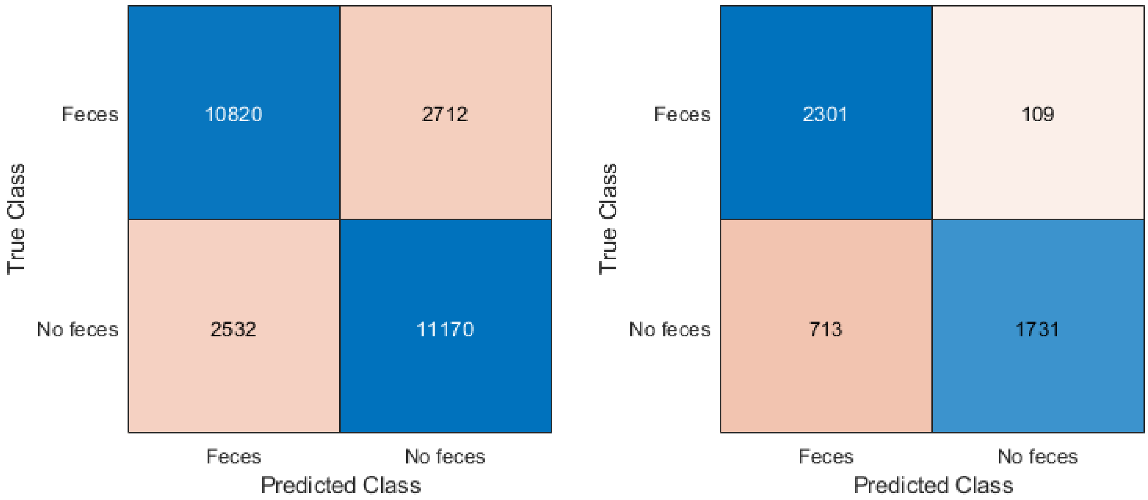


Figure 4.8: Second version algorithm applied to dataset.



(a) The confusion matrix for the training set

(b) The confusion matrix for the validation set

Figure 4.9: Confusion matrices of the ML Algorithm

respectively. These results are way closer to the desired accuracy. However, it should be mentioned that this test, as well as the others, are done for relatively small data sets compared to other ML algorithms. If more data is added the performance could change. Just as for the previous algorithm versions, once more data has been collected, the performance of the ML algorithm should improve as well.

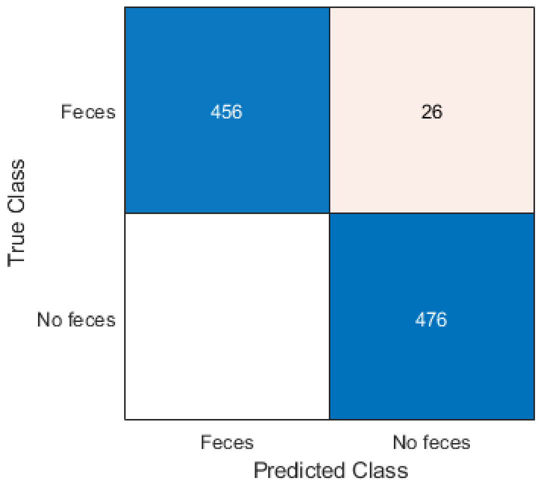


Figure 4.10: The confusion matrix for the improved validation set

4.3. Testing the algorithm

Testing the eNose algorithm was executed in 2 phases.

Phase 1

After understanding the behaviour of the sensors the first version of the algorithm was constructed and six units were distributed on six toilets to test the current algorithm, the outputs of these devices were closely monitored in the dashboard. The problem about this setup however was that the incoming data was unlabeled, which means that it was not obvious whether the current algorithm was indeed detecting correctly, since it is not known what exactly is happening in the toilets. To solve this problem two methods were implemented. First a button was added to the device, and the participants were requested to press the button each time they defecated. However, some devices were placed out of reach, for instance above the toilet, which made it inconvenient to press the button, so a second method was implemented. The second method was a short questionnaire that participants accessed via a QR-code placed next to the toilet. The second method yielded high quality data since the participants could also mention other activities carried out in the toilet such as showering, and they could describe their defecation. The obtained data was then analyzed and used to improve the algorithm and create the upgraded versions of the algorithm that were described prior. This process is illustrated in Appendix A.3.

The performance of the first phase was evaluated based on the selectivity of the algorithm. Because it is hard to know if an instance of defecate happened without it being detected this was left out of consideration. This is because roommates might not notify the team of defecate and thus the team has no way of knowing if it should have been detected. This is why the selectivity is defined as the percentage of correctly predictions of feces as opposed to all predictions saying feces have been detected. The results are based on the devices of Maxim, Menno and Mark. This is because these devices have collected enough data using the newest prototype board with the correct sensors and the newest software. Most of the data was collected at Maxim's location. This is because of roommates that were contributing to the amount of data. It is expected that this location will have the best performance since most of the datapoints were from this location resulting in the algorithm being fine tuned for this site.

Using the questionnaire and the algorithm predictions, the selectivity was computed. The devices of Maxim, Mark and Menno had selectivities of 95%, 59% and 57% respectively. The overall performance was calculated to be 73%. Even though most of the instances are correctly predicted, this still is not the performance sought after. As stated before, it is necessary to collect more data on more different locations to improve the algorithm significantly.

Phase 2

After upgrading the algorithm based on fase 2, the complete device will be tested in the intended environment, namely, the nursing home.
In this phase a number of devices will be distributed among several rooms of the nursing homes, and the devices will be put to work. To verify whether the eNose is detecting correctly, an SMS will be sent to the nurses each time the device detects solid tools. The nurses are then requested to respond to the SMS with a "Yes" if the detection is correct and "No" if it is not.

The obtained data based on this testing phase will then be used to upgrade the algorithm again. These results will however not be discussed in this report due to time restrictions beyond our control.

5
Communication

The eNose needs to communicate with the server when solid stools are detected. As the eNose operates in close proximity to the Bedsense, which is already connected to the server of MomoMedical, there are two possibilities for communication: direct or via the BedSense module. BedSense contains the microcontroller unit (MCU) ESP32-WROOM-32D, which has an onboard radio chip capable of various communication techniques in the 2.4GHz band. This gives great freedom to choose the wireless communication method between BedSense and eNose. On the other hand, direct communication with the server limits the method to the internet only. The benefits of using direct communication rather than indirect communicating are that it enables easy OTA updates and integration with monitoring systems. Hence, direct communication is used.

In order to send data to the server, the ESP32 must connect to the internet and send data through a protocol to the server which in turn must interpret the received data. As nursing homes do not have a spare Ethernet connection in every patient room and there is already an existing Wi-Fi network present, the connection mode to the internet is Wi-Fi. Regarding the protocol, there are various options that can be transmitted over the internet and require further analysis. How the server-side processes the received process depends mostly on the chosen protocol. Hence, this chapter will first discuss how the sensors are read out by and stored on the ESP32. Secondly, Wi-Fi and the relevant key concepts of Wi-Fi are explored. During the same section, the implementation of the software to connect to the Wi-Fi for the ESP32 is addressed. The third section discusses the HTTP protocol and the implementation of the communication between ESP32 and the server. Lastly, the server-side of the communication is discussed.

5.1. Sensor read-out

The read-out functions, as well as the sensor data, are grouped together inside the class sensor (See Fig. 5.2 and Appendix B.1 for the code). The function `read sensors`, as the name might suggest, reads the values of all the sensors. The gas sensors output the concentration in an analogue way to the GPIO pins of the ESP32. The ESP32 has two types of 12-byte ADCs that can convert the analog signal into a number. ADC1 has 8 channels while ADC2 has 10 channels but cannot be used in combination with an active Wi-Fi component. [8, p. 663] Hence, only ADC1's channels are used. To use the ESP-IDF functions `adc1_get_raw(channel)`, the output length and attenuation needs to be configured. As the sensors do not exceed 1 V [2], the best resolution is obtained by setting the attenuation to zero dB. The ADC output can be noisy and not consistent. However, this can be reduced by using multisampling which averages multiple rapid successive measurements. [8, p. 676] In Fig. 5.1 the benefit of multisampling is displayed. On the left is the signal of the MQ-3 sensor varies with 20 mV. While after an OTA update with multisampling, the signal of the MQ-3 sensors varies only with 2 mV.

The DHT sensor is a digital sensor that consists out of a thermistor and a humidity sensor. The data of the sensors are encoded and sent with a proprietary protocol to the ESP32. Rather than designing the system that can read the sensor, an existing C++ class is forked from `https://github.com/gosouth/DHT22` due to time constraints. Another class included inside the sensor class is the buffer. The buffer is the class that contains the algorithm functions and variables, as discussed in Chapter 4.

When the button, used to signal solid stools, is pressed, it should be registered instantaneously rather than waiting for the function `update_sensor` to read the button. This is realised by setting the GPIO interrupt configuration to a Positive Edge, create and set up a task, and register an Interrupt Service Register (ISR) service as well as a handler. When the button is pressed the ISR sends a message into the Queue that the button has been pressed. The task continuously runs and checks whether a message

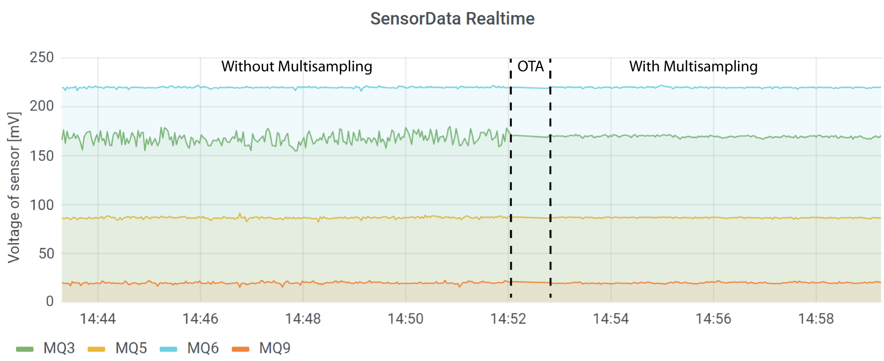


Figure 5.1: Effects of multisampling on noise.

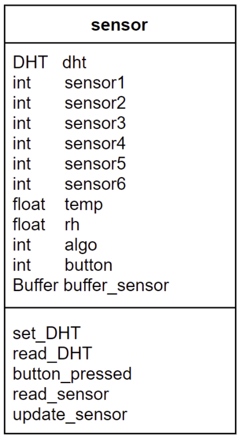


Figure 5.2: Class sensor.

from the Queue has been received and starts the function `button_pressed`. This function sets the button value to 20, which will be subtracted by one in the `update_sensor` function each time. This will ensure that the button value is sent multiple times to the server.

To obtain the values stored inside the class, the function `update_sensor` can be used. The input of this function is an integer in the range 1 to 6 which represents the MQ-3, MQ-4, MQ-5, MQ-6, MQ-9 and MQ-135 respectively. The numbers 7 and 8 are used for temperature and humidity. The algorithm value uses the number 9 while the button value uses number 10. With a simple switch case instruction, the value is read from the class and returned.

5.2. Wi-Fi

Most Wi-Fi networks use a security protocol, such as WEP, WPA, WPA2 or WPA3, to prevent unauthorised users access to the network. The earlier protocols WEP and WPA are now mostly obsolete due to security weaknesses that can be easily exploited. Currently, the most widely used security protocol is WPA2, which will eventually be replaced by WPA3 as it provides additional security. Fortunately, WPA3 is backwards compatible with WPA2 [9]. Hence, the focus will lie on WPA2 as most nursing homes still use this protocol. Within WPA2 there are two modes: PreShared Key (PSK) which uses a password and Enterprise. With a WPA2-enterprise network, a user communicates to the router which will verify with a Remote Authentication Dial-In User Service (RADIUS) if the connecting user is authorised to connect. Within WPA2-enterprise there are additional sub-modes such as EAP-MSCHAPv2 or EAP-TLS. The former only requires an anonymous identity, username and a password from the client while the latter also requires that the client has a certificate.

All the functions related to connecting to Wi-Fi are combined into the class `wifi_connect` (See Fig. 5.3 and Appendix B.2 for the code). Inside this class, the necessary Wi-Fi parameters are stored.

The function `wifi_init_sta` connects to an AP based on the stored parameters. The function can be divided into seven steps. The first step is to initialise `netif`, which is an abstraction on top of the TCP/IP stack developed by ESP-IDF. Secondly, instances of event handles are registered. Thirdly, through the struct `wifi_config` the Wi-Fi settings and mode are copied to the underlying Wi-Fi handler. During this step, it is possible to configure the WPA2-enterprise settings. The fourth step creates a station control block and starts the Wi-Fi station. The next step blocks the main program and performs the task which tries x times or until it is connected. In the sixth step, the bits set by the task are read to check whether the task was successful or failed. The last step is deregistering the event handles created in the second step.

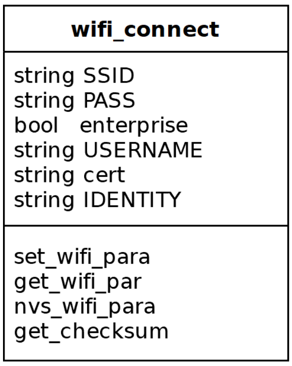


Figure 5.3: Class responsible for connecting to Wi-Fi.

5.3. Protocol

In Internet of Things (IoT) applications, the four main application protocols are MQTT, CoAP, AMQP and HTTP. [10] These protocols are all handled further by a TCP/IP-stack. Important aspects to select the application protocol on are: overhead size, reliability, security possibilities, as well as ease of use.

The bandwidth usage of the ESP32 should be minimised as a nursing home will have many eNose devices present. Those many devices all use a small amount of bandwidth which, when summed up, can be significant. The data will be transported with a header included. This header provides extra overhead and needs to be minimalised to increase the efficiency of the used bandwidth. The default size of the Header of MQTT is 2 Byte, CoAP is 4 Byte, AMQP is 8 Byte, and the HTTP header is of an undefined size. A basic HTTP header used for this project is around 100 bytes due to the mandatory header fields such as the hostname, HTTP version, user agent and content-type. Hence, HTTP has a significantly larger overhead size compared to MQTT, CoAP, and AMQP.

Reliability is an important factor as all the sensor data should be visible in the monitoring dashboard. Moreover, it is bandwidth expensive to compensate for possible transmission losses with the extra transmission of the same data. Unlike CoAP which uses UDP as transport protocol, the other application protocols use TCP. The TCP protocol can guarantee delivery and use extensive error checking while the UDP protocol cannot. However, UDP is faster than TCP since it for example does not require handshakes. Hence, CoAP is not a suitable application protocol for the ESP32.

As the communication contains medical sensitive information, solid stool frequency, the communication between device and server should be protected to prevent eavesdropping. This security, however, comes with a cost of higher data usage and introduces communication latency. [11] All the protocols support TLS, which is an encryption protocol between server and client. The server will be authorised by the client using a stored certificate and the communication will be encrypted. In the case of HTTP and a TLS connection is used, the protocol is called HTTPS.

The last aspect to consider for the application protocol is the ease of use. The bodies of MQTT, CoAP and AMQP are all in binary to reduce bandwidth. The body of the HTTP request can be in plain text as well as in binary data. A common format of the body is JavaScript Object Notation (JSON) which is a standardised data format that is easy for humans to read. Another format for the HTTP request' body is Protocol Buffers (PROTOBUF). The format uses binary data to represent a message that can be deserialised by the server to obtain the values sent. This reduces the used bandwidth significantly but it is unreadable for humans. To speed up development, the HTTP protocol with a JSON format is chosen. The format can eventually be replaced by PROTOBUF to reduce the bandwidth.

The actual implementation of the HTTP protocol as communication can be split up into three parts: generating the request, setting up the connection, and processing the response. The code regarding to the implementation of the HTTP protocol can be found in Appendix B.4. All the functions and variables are grouped inside the class `http_request` (Fig. 5.5). An overview of the communication of the measurement data between the server and ESP32 is displayed in Fig. 5.4

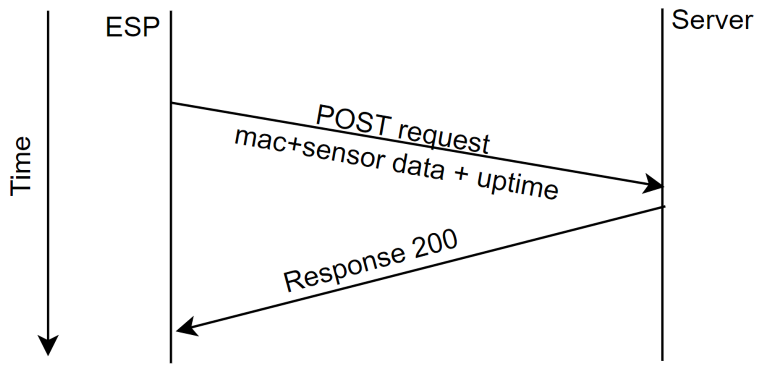


Figure 5.4: Communication measurements between server and ESP32 during measurements.

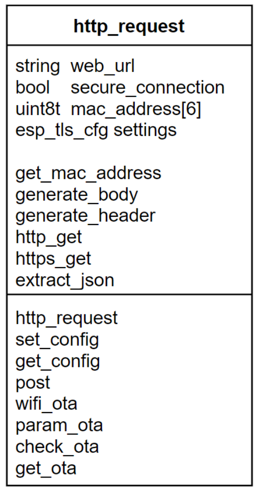


Figure 5.5: Class http_request.

5.3.1. Generating the request

The HTTP request is HTTP method dependent and consists out of a body and a header. HTTP methods, often called verbs, describe the action that the HTTP request performs. The most common verbs are GET, used to retrieves data, and POST, used to create a resource on the server. Only the GET request has no body. To determine which verb to use, it needs to be considered whether the operation needs to safe or idempotent. Verbs such as GET are called safe as they can be called many times without changing anything on the server. Idempotent verbs such as PUT and GET can be called multiple times and produce the same result. Verbs as POST and PATCH are neither idempotent nor safe. To publish the measurements data into the database a non-idempotent non-safe action is used, hence a POST method is used

The headers of the request with a body should at least contain the verb, host URL, host path, content-type and content length (See Listing 1 for an example). GET methods do not require a content-type and content-length header as it does not contain a body. Both types of headers can be generated in the function `generate_header` which takes as input the content length and outputs a string with the header. The actual content of the message depends on what the server expects. The variables that are sent to the server are all the sensor data, algorithm values and button values. Additionally, to estimate the reliability and check for fatal errors of the ESP32s' it is useful to also send the uptime of the devices. A low uptime can indicate that the device periodically reboots on itself due to a bug. Another variable transmitted is the MAC address of the device. This MAC address is unique for every device and programmed by the manufacturer. Hence, it makes an excellent device identifier. The function `generate_body` takes a pointer to the sensor class as input and generates the body in JSON format.

5.3.2. Setting up the connection

To communicate with HTTPS the server needs to be verified by the client and the communication must be encrypted. Verifying the server is done using certificates. Certificates are issued in a chain of trust by certificate authorities, which means that certificates of one authority are signed with a higher trusted certificate. The highest level of certification is called a root certificate. A copy of these are stored on devices to verify the root certificates.

The first step of the connection is the client obtaining the certificate of the server. To verify if the certificate is neither expired nor invalid, the client verifies if it is signed by the authorising certificate authority. This does not require another connection to the authority due to asymmetric encryption.

```
POST /API/endpoint.php HTTP/1.1
Host: URL
User-Agent: ESP32 v9
Content-Type: application/json
Content-Length: 202

{
    "tag": "4C:11:AE:A5:F1:CC",     "T": 10,          "H": 100,             "S1": 10,
    "S2": 123,                      "S3": 523,        "S4": 213,            "S5": 213,
    "S6": 213,                      "UP": 912,        "A": 0,               "B": 1
}
```

Listing 1: Example of HTTP post request to publish the measurements.

The verifying of the intermediate signing certificate authority continues until the device recognises a trusted authority, i.e. root certificate. The trusted authority is recognised as its certificate is stored on the device.

The ESP-IDF `esp_tls` is very useful and contains all the functions required to set up an HTTPS connection through TLS. The root certificate for the TLS connection is configured by the ESP-IDF function `esp_tls_set_global_ca_store`. After initialising the function `esp_tls_conn_http_new` is used to start the TLS connection with the server. When connected successfully, the HTTP request is written to the server through the function `esp_tls_conn_write`. The following step is to read the response of the request using the function `esp_tls_conn_read`.

5.3.3. Processing the response

When the measurements are successfully processed by the server, it will respond with an HTTP 200 code and a message 'measurement processed' and the ESP32 continues its program. There are two kinds of failures possible where the measurement is not placed in the database: server errors and network errors. Server errors rarely occur and are caused primarily by wrongful input. In this case, the server will pass an error code to the ESP, which the ESP32 will discard and continue with the program like no error occurred. Re-transmission is undesirable as the server error is likely to repeat itself. The other kind of errors, network errors, can occur due to a poor Wi-Fi connection or an internet outage. In this case, no message is received by the ESP32 and the device will time-out waiting for a response and reboot. The reboot is done to reset the device in the case that the device is stuck. This solution will lose the last measurement as well as the measurements that would have been performed during the rebooting.

A more elegant solution is to generate a sort of queue where the HTTP requests are stored in case of a time-out. This could not be implemented due to time constraints of the project. To implement this several changes to the structure/code are advised. Firstly, all the requests need to be stored inside a First In - First Out (FIFO) queue. This queue should be limited to a certain amount of requests as otherwise there would be no memory left during a long internet outage. When the transmission is successful the request needs to be removed from the queue. Such queue can be generated using the ESP-IDF ring buffer API which can create a queue of variable-sized items. Another method is to create a linked list using a struct with a string of the request and the pointer to the next struct. The second change is to place the functions that perform the HTTPS request inside a task that runs continuously. A task can run in parallel from the main software. [8]. Inside the task, it is checked whether a new HTTP request is present inside a queue. Thirdly, in case of time-out the device should not reboot but rather continue with the program and try again later. Fourthly, the server checks the timestamp of the previous entry and the current time against the uptime of the previous entry and current, to calculate what time the measurement has been taken. Otherwise, the dashboard could show only bursts of data rather than continuous data.

5.4. Server side

The server should exist out of three sub servers: a web server that is reachable via the HTTP protocol, a database server to store any data send by the ESP32 and an application server for the dashboard. To ensure that all data is captured and stored, reliability and uptime are very important. Commercially available web/database servers are relatively cheap nowadays with high uptime, unlike self-hosted servers where it can be tricky to have high uptime. For this reason, the web and database servers are outsourced to a commercial service. Unlike the database and webserver, the dashboard does not need to be on all the time as it only used for debugging or monitoring. The application can be run on a commercial Linux server but to save prototyping cost, the dashboard is run on a local Linux machine.

5.4.1. Database

The database server is the opensource MariaDB, based on an MYSQL database, run on an InnoDB engine. To control the database, opensource PHPMyAdmin is used which can control and manipulate the database. Important aspects for designing a good database is data integrity, reduce the duplication of data, and reduce the size of the database [12]. Rather than storing all the measurement data in one table, multiple relational tables are generated. There are three tables: Locations, Devices, Measurement (see Appendix B.3 for a SQL structure export). This would reduce the need to store all the location information, such as phone number, for each measurement. The table devices in turn reduces the need to store all device information as well as the location information, as the device is part of the location. In Fig. 5.6 an overview is given of the relations of/and tables.

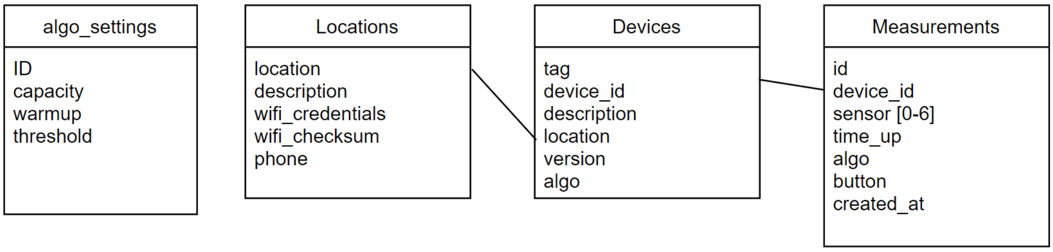


Figure 5.6: Tables on the Database.

Measurements

Gas sensor data are 12 bit number, which can be stored inside a small int that requires just 2 bytes. The data of temperature and humidity lies between -40.0-80.0 and 0-100.0% respectively, can be represented by a float or a decimal. But it is more data-efficient to multiply with ten and store it inside a small int. To sort the data on time, there is also a DateTime column that stores information about the time and date. As there are multiple devices that send data to this table, an identifier needs to be given to each device. One way to accomplish this is to store the send mac-address with each entry. However, a string of 17 chars requires a lot of storage and searching is more resource-intensive. A better method is to store the 17 char mac address in the separate look-up table `devices` and store the id, which is of the type int, in the `measurements` table. With 1000 devices and a sample rate of .5 Hz this method saves 576 MB of redundant data a day.

The `device id` of the measurement is labelled as a foreign key of the `device` table column `device id`. This means that it is restricted to delete or update any `device id` from the `device` table. This ensures data integrity. The device id is indexed to speed up the loading times for the dashboard.

Devices

In the table `devices` there are six columns: `tag` (text), `device` id (int), `description` (int), `location` (int), `version` (int), `algo` (tinyint). The `tag` contains the MAC address of the device, while the `description` column describes in what room the device is. In this table, the column `location` is a foreign key of the table `location` column `location`, which restricts updating and deleting. The `version` column is used to keep track of the device version for OTA. The last algorithm value is also stored in this table. This is redundant data but as the API, discussed in the Subsection 5.4.2, uses the previously algorithm value it is faster to store it inside a smaller table rather than searching the large measurements table.

Location
The `location` table contains the `location` (int), `description` (text), `phone` (tinytext) and Wi-Fi parameters. The `description` contains the location name and the `phone number` the number to which an SMS needs to be sent. The table stores Wi-Fi parameters as this will be used for Wi-Fi OTA discussed in Subsection 6.2.1.

5.4.2. Webserver

The web server is an Apache server that allows an HTTP client to access files present on the server through the HTTPS protocol. On this server, an Application programming interface (API) needs to be created to interact with the database. An API is used to link two different software applications and in this case the software of the ESP32 and the database/webserver. There are various kinds of APIs styles such as Representational state transfer (REST) and GraphQL. Both can work through HTTP request and hence can be used as API on the webserver. GraphQL is a quickly upcoming and very flexible API language designed by Facebook focused on data retrieval queries. [13] REST, on the other hand, is not query-based and more focused on URLs and predefined structures, which makes it less flexible for data retrieval. The provided flexibility of GraphQL is not necessary as the API needs to only store a standard measurement and perform OTA updates. Hence, a RESTful API will be developed.

REST is a stateless client-server communication, which means that there is no information stored between requests. Richardson maturity model is used to describe how well an API adheres to the REST principles, which provides 4 maturity levels with each their own specific requirements. [14] Level 0 is the Swamp of POX where HTTP is merely used as a transfer protocol to execute a specific piece of code based on the arguments. E.g. there is a single endpoint that provides all the operations. Level 2 requires that a device can make a request to multiple endpoints where functions are grouped by functionality. E.g. the storing of the measurement has a different endpoint than OTA related operations. Level 0 and 1 primarily uses just HTTP post request to communicate with the API. Level 2, on the other hand, uses the HTTP verbs such as GET, PUT, PATCH and POST requests which reduces the functionality of each operation. Level 3 APIs can return hypermedia such as a URL which allows having some state-based experience. The state-based experience is not useful in the basic API required by the eNose and hence the focus lies on a level 2 RESTful API and requires a focus on verbs and resource distribution.

The posting of the measurements can be performed inside a single endpoint as a POST request the request is supposed to be not safe nor idempotent. The implementation of this endpoint `measurements.php` and other relevant files are given in Appendix B.4. When the ESP32' HTTP request is received at the server, the script executes five steps. In the first step, the script connects to the database server and creates a connection handle. Secondly, the scripts retrieve and decode the HTTP request and check whether it is a POST request and all the parameters (MAC address, temperature, relative humidity, all the sensors, uptime, algorithm value, and the button value) are set. When it is a valid request, the device id is searched based on the received mac address in table devices. If the result is empty, the device is not registered and an error message is passed. Otherwise, all the parameter data as well as a timestamp in the format of Y-m-d H:i:s. The fifth and last step is to update the solid stools detected parameter in the table `devices`. When the previous value stored in the table is the same as the incoming packet, nothing needs to be updated or called. When the incoming and previously stored value differs, the latter needs to be updated. If the value is updated to true, an SMS message needs to be sent to the number registered to the location using the `smsWebHook` function. The function requires a number and a message text, which sends an HTTP post request to an SMS service server. It is useful to put in the text message the location description as well as the device description of the detecting eNose. To obtain all this data from the tables devices and locations, aliasing is used in the SQL query (See Listing 2).

To make the documentation of the API easier as well as providing useful HTTP request simulators, the API is also described in the OpenAPI 3.0 specification (See Appendix B.3). This specification is a

```sql
SELECT devices.tag, devices.location, devices.description as place, location.phone,
    location.description as user
FROM devices
INNER JOIN location ON devices.location=location.location
WHERE devices.tag = mac address
```

Listing 2: SQL query with aliasing to obtain phone number, device and location description.

way to describe REST APIs in a standardised way. As it is standardised there are many tools that can either generate documentation with examples, spin up mock servers which return dummy data when called or even create server implementations in various kind of programming languages.

5.4.3. Dashboard

The dashboard should visualise all the data relevant for the visualisation. There are many open-source visualisation platforms such as Apache Superset, Apache Druid, Graphite, or Grafana. All these platforms can plot graphs based on SQL queries and are capable of meeting the requirements. For future integration purposes, the platform Grafana will be used as the client currently uses it as well.

The operating software of the application server is centos 8, a server distro based on Red Hat Linux, which runs multiple other servers. Hence, the open-source containerization platform Docker is used. A container, unlike a virtual machine, runs on the OS of the main system which reduces the footprint of docker significantly while still offering isolation. The container includes any dependencies or libraries that are necessary to execute the application. Docker container can easily port exposed ports, run multiple instances of the same software, set up underlying networks, and can provide extra security by increased isolation (in contrast to running the service barebone on the device). To create the docker container a docker-compose file is needed, which is given in Listing 3. The script configures a network port from and to the container as well as persistence storage, which can be reached outside the container. To configure Grafana, inside the configuration file `grafana.ini` the eventual domain and root_url are changed to ensure the correct functioning of user invites etc. After running the terminal command `docker-compose up` from the relevant folder, Grafana starts on the default port 3000 and can be configured further through a web browser.

```yaml
version: '3.3'
services:
    grafana:
        ports:
            - '3000:3000'
        container_name: grafana
        volumes:
                - ./volumes/grafana/data:/var/lib/grafana
                - ./volumes/grafana/log:/var/log/grafana
                - ./volumes/grafana/config/custom.ini:/etc/grafana/grafana.ini
        image: 'grafana/grafana:latest'
```

Listing 3: Docker Compose file for Grafana.

The server is up to this point only reachable through port 3000 inside the same Wi-Fi network. To fix this, on the router the default HTTP port 80 and HTTPS port 443 on the outside are port forwarded internally to the server. To translate the incoming ports 80 and 443 to ports used by Grafana on the server, a simple reverse proxy is used. Such proxy is exposed to the default HTTPS ports and scans incoming network connections' origin. Based on the origin, the traffic is redirected to another server, in this case Grafana. As the reverse proxy and Grafana run on the same server, the traffic in between is not encrypted. HTTPS requires certificates to operate securely and without a warning. With the free service letsencrypt certificates can be generated for a specific domain which can be linked to the configuration files of the reverse proxy server. To promote a secure connection, SSL is forced for every connection. As an HTTP connection is requested, a redirect is given to HTTPS.

Within Grafana it is possible to create variables to be used in the queries of the plots. It is useful to select the setup_type, setupID, and the size limit due to speed performance. An example code for the plot for the sensor data is given in Listing 4.

```
location = select location as __value,  description as  __text from location;
select device_id as __value,  description as  __text from devices WHERE location IN (location);

--------
SELECT created_at, sensor_1, sensor_2, sensor_3, sensor_4, sensor_5, sensor_6
FROM measurements
WHERE tag IN (setupID)
ORDER BY created_at desc
```

Listing 4: Code of graphana plot for the sensors.

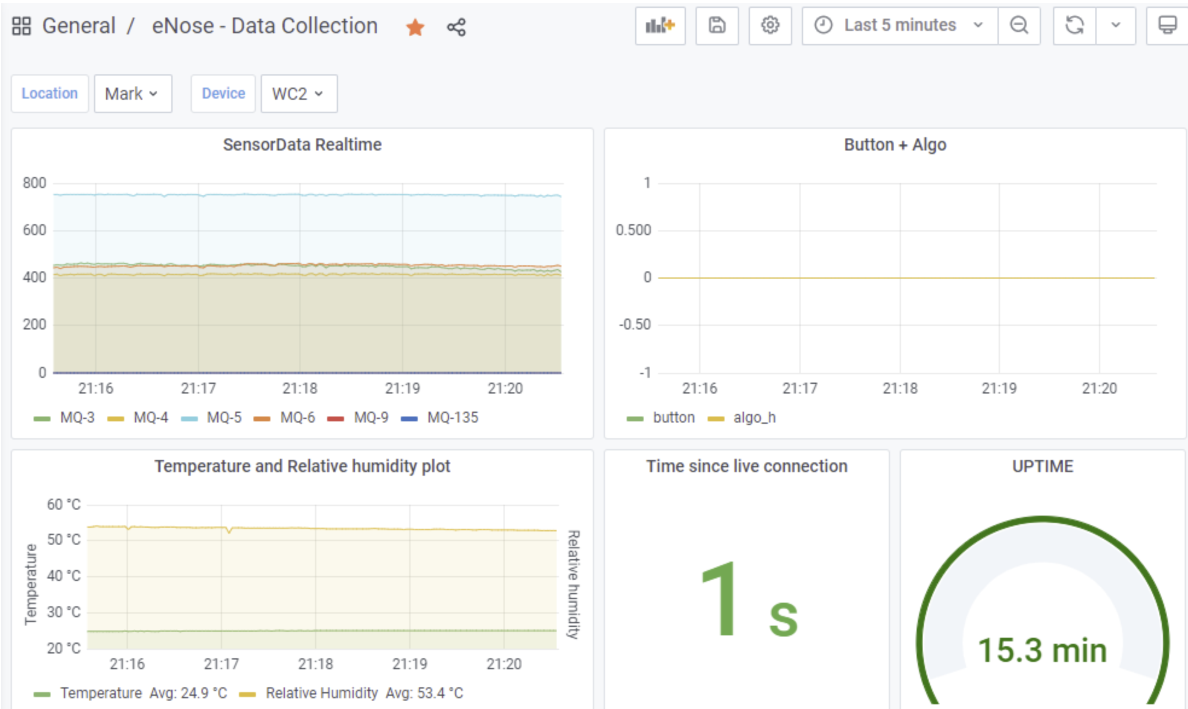


Figure 5.7: Dashboard of Grafana.

6
Over The Air Programming

OTA programming is useful when the product is difficult to reach and physical access. A device can receive a complete firmware update through Firmware Over The Air (FOTA) or some small adjustments through Over The Air Parameter adjustment (POTA). It is possible to use OTA for complete firmware updates as well as parameter adjustments. The eNose uses both kinds to update the software as well as the settings of the algorithms and Wi-Fi.

There are various differences between FOTA and POTA. An important difference between firmware updates and parameter updates are the size of the data. Firmware of the ESP32 is stored in .bin files and is around 1MB, 25% of the flash memory, in the case of the eNose. Meanwhile, parameter adjustments are presented in JSON format and are less than 1kb in size. The relatively large size of the firmware update makes it much more prone to error than the parameters. Moreover, Firmware updates require that the firmware is transported completely and without any error, as otherwise, a fatal bug can occur which could brick the MCU. Parameter adjustments are less sensitive to fatal bugs as the parameters should not affect the actual functions of the device. Hence, special attention needs to be given to file integrity when using FOTA. Another difference between FOTA and POTA are the distribution capabilities. Firmware updates need to be compiled which makes it undesirable to have many differences for each device. Parameters can be passed to any device without compiling which makes it an efficient method to customize settings for each device separately.

As discussed in Chapter 5, the communication method used for measurements of the eNose is Wi-Fi with the client-server HTTP protocol. It is convenient to use the same method for OTA and make the ESP32 the client and the remote server the server. This does not require changes to internet settings of the nursing location and it allows for easy one-to-many communication. The server has an API that reads and responds to the incoming HTTP requests of the client, the ESP32.

In this chapter the process of a firmware update is discussed firstly. The chapter continuous with parameter adjustments which can be separated into two parts: Wi-Fi OTA and Algorithm OTA.

6.1. Firmware Over The Air

The general process of the FOTA exists out of three steps. Firstly, the device needs to check the server whether there is a firmware update. The second step is to download and write the data to an empty space on the memory of the ESP32. Lastly, the device should change its boot records and start from the newly uploaded firmware. Within these steps, it is important to verify if the downloaded firmware file is intact and that the effects of corrupt firmware are mitigated. As discussed in the introduction of this chapter, the firmware is uniform which means it cannot be used to pass e.g. Wi-Fi parameters. How the Wi-Fi parameters are stored during an FOTA update is discussed in the Section 6.2.1. An overview of the communication is given in Fig. 6.1.

6.1.1. Partitions

Software of ESP32, FOTA updates and other kinds of data are all stored inside the memory of the ESP32. This memory is divided into partitions which are defined into a partition table (see Table 6.1). It contains any non-volatile storage, `otadata` which holds data about which partition to run, `phy init` which contains device-specific calibration data, as well as two applications partitions of 1.5MB each. The first application type partition is initially uploaded through USB to the device. When a firmware update is present, the factory partition will write to the second OTA partitions. After the downloading is finished, the device will boot from this partition. When again a new firmware update is presented, the update will be overwritten to the first OTA partition.

Table 6.1: Partition table of ESP32.

Name	Type	Offset	Size
nvs	data	0x9000	25.6 kB
otadata	data	0xd000	12.8 kB
Phy init	data	0x1000	6.4 kB
OTA 0	App	0x10000	1.5MB
OTA 1	App	,	1.5MB

6.1.2. Checking for an update

To check whether an update is available, the ESP32 will send a GET request to the API (Appendix C) which will respond with a checksum and version of the update. This checksum can be used to check file integrity. It is generated by a SHA256 algorithm which rarely has false positives of integrity, also called collisions, compared to a default MD5 hash algorithm. When the remote version number has the same as the version number on the device, nothing will happen. If the numbers vary, the ESP32 will start the function `preform_ota`. The function will first determine which partition is currently run from and which partition to write through. Using the OTA library of ESP-IDF, the function `esp_ota_begin` is called which erases the partition and provides an update handle.

6.1.3. Downloading

The next step is to download the update from the server. Although the HTTP protocol supports a 32 bit - or 64 bit depending on the server - content length and thus have a maximum body size of 2TB in theory. Hence, the protocol does not pose a limitation for the direct transfer of the file. However, given the nature of the HTTP interface discussed in Chapter 5 and the required processing of the data, it is not possible to directly write the file to the OTA partition. It is also not possible to store the whole 1MB firmware update in ram as the ram has only a few hundred KB of storage. A solution of this problem lies in sending the information partially such that it can be processed in parts. The HTTP protocol offers such possibility via chunking, where the connection is opened and the server sends multiple responses with each a part of the data. However, it is not clear how the ESP-IDF functions can support chunking. Hence, the ESP32 will send a request to the server with an offset and length variables. The server will respond with that part of the firmware update. The ESP32 write that specific part to the OTA partition using the function `esp_ota_write` with the help of the update handle generated by `esp_ota_begin`. This process is repeated until no bytes in the body are sent by the server to the ESP32.

The body of the HTTP response of the server can directly send binary data. However, the binary data contains NULL characters which are difficult to handle in C as it is used to indicate an end of a char array. A solution to this problem is to encode the raw binary data of the server into BASE64. This introduces an overhead of 33% as only 64 chars. The received data on the ESP32 is decoded using the mbedtls library, which is already used for HTTPS connection, before being written to the OTA partition.

After the download is finished, the new OTA partition is validated using the default ESP32 `esp_ota_end` function. If it is validated, the boot partition is switched to the new OTA partition and the ESP32 reboots.

6.1.4. Reliability

With FOTA updates it is important to keep the device always connected to the internet and capable to perform another update as physically accessing all the devices is time-consuming and difficult. The threats to an online and FOTA-capable device are that either the firmware update is corrupted during the transmission or that there is a mistake/bug in the firmware that prevents a future FOTA update.

To combat corruption caused by transmission, the server sends a generated SHA-256 hash of the software update. Due to time constraints, this checksum is currently not used. However in further updates, this checksum should be stored and compared to the ESP32 generated checksum of the new OTA partition during the validation after downloading. The checksum can easily be generated using the default ESP32 function `esp_partition_get_sha256`. When the checksums differ, the whole OTA update should be

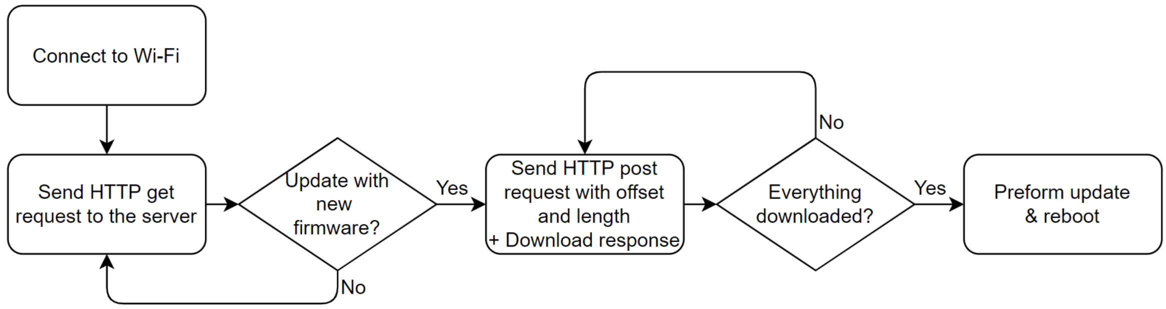


Figure 6.1: Communication process of the Firmware Over The Air.

discarded.

Another reliability improving method is a firmware rollback system. After the downloaded firmware update is validated and the ESP32 reboots to that partition, the device does not immediately delete the old OTA partition. Rather the bootloader of the ESP32 checks during the reboot whether there is already an `IMG_PENDING_VERIFY` flag on the new OTA partition. If there is, the bootloader discards the new update and rollsback to the old partition. When there is no flag set, the bootloader continues and sets the `IMG_PENDING_VERIFY` and boots the partition. During the partition, an active confirmation needs to be given to verify the correct working of the ESP32 and cancel the rollback. The correct working of the partition occurs in the class-less `partition_check` function. This function currently verifies the working of the firmware by pulling up and down certain unused GPIO pin. After 5 seconds these pins are read and when the result is as expected, the flag `IMG_PENDING_VERIFY` is removed. Ideally, this partition check would check whether the device is able to connect to the internet and decode the OTA message. This would ensure that the device is online and capable of FOTA

6.2. Parameter adjustments

POTA updates are very useful as it allows for quick and small updates as well as the personalising of devices. Wi-Fi credentials vary between location and can change over time, hence the Wi-Fi credentials of the ESP32 should be able to update per location. The algorithm values are not location depended but as many small improvements can be made through new parameters, it can be useful to pass these to the eNose. The benefit over a FOTA in this case is that the system-critical software is not changed. This reduces the time needed to testing non-algorithm part of the software and reduces accidental errors in those parts.

6.2.1. Wi-Fi Over The Air

Wi-Fi credentials should always be present on the eNose in order to connect to the internet and receive updates. This implies that the credentials should be stored inside the non-volatile memory to ensure that reboot or FOTA does not erase the credentials. With Wi-Fi Over The Air (WOTA) those credentials can be updated.

The function `nvs_wifi_para` is used to store and retrieve the Wi-Fi parameters from the non-volatile memory. When the firmware is flashed through USB for the first time, Wi-Fi credentials are given in the firmware and the global flag `UPDATE` is set to false. Inside the function, the non-volatile memory is opened. Within this storage, the Wi-Fi credentials are searched. When there are not any credentials in the storage, the `UPDATE` flag or the `WOTA` flag is set, the Wi-Fi credentials are written to the NVS. The `UPDATE` flag is given to each FOTA update and the WOTA flag is set when Wi-Fi parameters are updated over the air. If the flags are not set, the credentials are obtained from the NVS and stored inside the Wi-Fi class. Currently, the storage is not encrypted which means that anybody with physical access to the device can obtain Wi-Fi credentials and even the software can be reverse engineered which allows an attacker to obtain Wi-Fi credentials of all the locations.

Even though the communication is secured through the use of HTTPS, sending all the credentials periodically is undesirable due to security and due to unnecessary bandwidth. The solution to this problem is to generate a SHA256 checksum of all the Wi-Fi credentials on the ESP32. This checksum is sent to the API (Appendix C) with an API key and the device tag. Such key varies per location and is used for increased security. The API will retrieve the location of the device and the location's Wi-Fi checksum and credentials. When the checksums differ but the API keys are the same, the Wi-Fi credentials are sent to the eNose. The eNose will decode the JSON response and use the `nvs_wifi_para` to store the credentials.

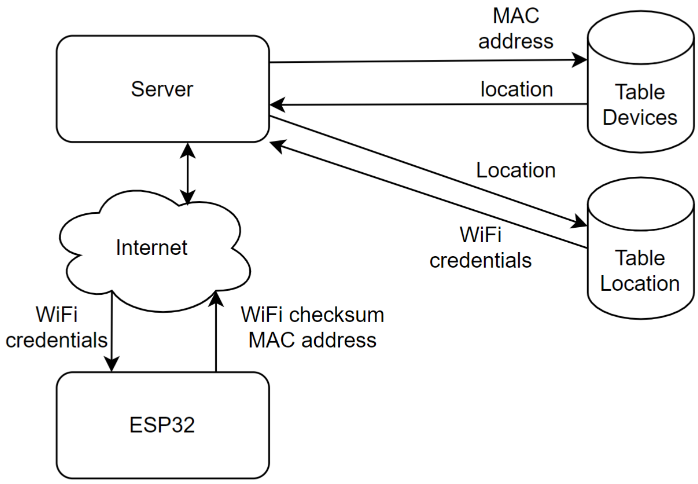


Figure 6.2: Process of updating Wi-Fi parameters OTA.

6.2.2. Algorithm Over The Air

Compared to WOTA and FOTA, the algorithm has no specific requirements. Using a GET request to the API (Appendix C), the capacity, warm up, and threshold are obtained from the table settings. After decoding the response on the eNose, the values are passed to the function `setValues` of the class buffer.

7 Conclusion

The work described inside this thesis tried to answer the following research question: "What are effective methods related to extracting, interpreting, and communicating the stool sensor data?". To answer the question an algorithm has been developed as well as a communication and Over The Air (OTA) method.

The algorithm shows that the concept works accurately for certain restrooms. The machine learning version shows great potential but needs to be expanded further. For both approaches more data would help improving the algorithm to be able to be deployed in bigger rooms such as those in nursing homes.

The eNose directly connects to Wi-Fi to an server through an Application programming interface (API). The protocol used between the eNose and the server is Hypertext Transfer Protocol (HTTP) with a JavaScript Object Notation (JSON) body. Although the JSON body uses more bandwidth than a binary body such as Protocol Buffers (PROTOBUF), it allows for faster development and easy debugging. The data is stored inside a simple relational database with multiple tables in order to reduce redundant data and increase speed. When feces are detected, an SMS alerts called by the API used which contains the location of the alerting device. There is also an dashboard which allows for easier development and monitoring of all the devices. The dashboard is an open source tool called Grafana.

As the devices are difficult to reach for updating, OTA programming is used. The firmware of the devices is stored centrally on the server and is the same of each device. Using HTTP request the update is obtained and executed through the OTA libary of ESP-IDF. As not all the devices must run the exact same software with the same settings, Over The Air Parameter adjustment (POTA) is used. This allows individually device configuration for parameters suchs as Wi-Fi credentials and Algorithm settings.

Future work

There are several recommendations and further developing steps needed to make the eNose market ready. As repeated many times, the amount of data directly influences the effectiveness of the algorithm. Before it is able to work robustly for all locations, more data is needed based on different factors such as room size, ventilation and other behaviours. On top of that, it would be beneficial if sensors were chosen that are less prone to cross-sensitivity (e.g. infrared gas sensors). These sensors, however, come at a certain price tag, which is why they were not considered for this project.

The communication should be based around FreeRTOS tasks to allow for parallel operation of independent parts. This allows for a queue for messages, to increase the reliability of the system. Another communication related part is the connection to WPA2 enterprise networks, which is not completely tested due to time constraints.

A recommendation of the OTA part is to encrypt the flash storage. The OTA currently stores Wi-Fi parameters in the flash, which can pose a security threat. Also API-keys can be reversed. This is solved when the flash memory of the ESP32 is encrypted.

Bibliography

[1] H Beele, s Smet, N van Damme, and D Beeckman. Incontinence-associated dermatitis: Pathogenesis, contributing factors, prevention and management options. Drugs & Aging, 35(1):1–10, December 2017. doi: 10.1007/s40266-017-0507-1.

[2] N. van Damme, S. Fakkel, and A. Stijns. E-nose for detecting solid stools: Sensor, 2021.

[3] Z. Yunusa, M.N. Hamidon, A. Kaiser, and Z. Awang. Gas sensors: A review. Sensors and Transducers, 168:61–75, 04 2014.

[4] R Jaaniso and O. Kiang Tan. Semiconductor gas sensors. Woodhead Publishing, Cambridge, 2013. ISBN 9780857098665.

[5] T. Schütze, A. Baur, W. Conrad T. Leidinger, M. Reimringer, and T. Sauerwald. Highly sensitive and selective voc sensor systems based on semiconductor gas sensors: How to? 03 2017. doi: 10.3390/environments4010020Âă.

[6] Ajiboye A. T., Opadiji J. F., and Yusuf A. O.and Popoola J. O. Analytical determination of load resistance value for mq-series gas sensors: Mq-6 as case study. TELKOMNIKA JOURNAL, 2020. doi: 10.12928/telkomnika.v19i2.17427.

[7] A.B. Karki. Biogas as renewable energy from organic waste. In BIOTECHNOLOGY, Volume X, 2009.

[8] ESP32 ESP-IDF Programming Guide. Espressif Systems, 2021. URL `https://www.espressif.com/sites/default/files/documentation/esp32-wroom-32d_esp32-wroom-32u_datasheet_en.pdf`.

[9] B. Indira Reddy and V. Srikanth. Review on wireless security protocols (WEP, WPA, WPA2 & WPA3). International Journal of Scientific Research in Computer Science, Engineering and Information Technology, pages 28–35, July 2019. doi: 10.32628/cseit1953127.

[10] N. Naik. Choice of effective messaging protocols for IoT systems: MQTT, CoAP, AMQP and HTTP. In 2017 IEEE International Systems Engineering Symposium (ISSE). IEEE, October 2017. doi: 10.1109/syseng.2017.8088251.

[11] David Naylor, Alessandro Finamore, Ilias Leontiadis, Yan Grunenberger, Marco Mellia, Maurizio Munafò, Konstantina Papagiannaki, and Peter Steenkiste. The cost of the "s" in HTTPS. In Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies. ACM, December 2014. doi: 10.1145/2674005.2674991.

[12] Dedi Iskandar Inan and Ratna Juita. Analysis and design complex and large data base using MySQL workbench. International Journal of Computer Science and Information Technology, 3(5): 173–183, October 2011. doi: 10.5121/ijcsit.2011.3515.

[13] O. Hartig and J. Pérez. An initial analysis of facebook's graphql language. In CEUR Workshop Proceedings, 05 2019. URL `http://repositorio.uchile.cl/handle/2250/169110`.

[14] Arne Koschel, Maximilian Blankschyn, Kevin Schulze, Dominik Schoner, Irina Astrova, and Igor Astrov. RESTfulness of APIs in the wild. In 2019 IEEE World Congress on Services (SERVICES). IEEE, July 2019. doi: 10.1109/services.2019.00114.

A

Algorithm

A.1. C++ code

buffer.h

```
/*
 *  Project:        eNose software
 *  File:           buffer.cpp
 *  Author:         Maxim Chin-On
 *  Created on:     4 May 2021
 *  Description:    header file for buffer.cpp
 */
#ifndef BUFFER_H_INCLUDED
#define BUFFER_H_INCLUDED
#include <iostream>
#include <vector>
using namespace std;

class Buffer
{
public:
    void setBuffer(int g, int h, int i, int j);
    int addData(int a, int b, int c);
    void printData();
    int  getMean(std::vector<int> data);
    int  algo(std::vector<int> mq3, std::vector<int> mq5, std::vector<int> mq9);
    void setValues(int thres, int g, int h, int i, int j);
    std::vector<int> getData(int sensor);
    ~Buffer();
private:
    //Constants
    int capacity;                                   //Maximum number of minutes saved in buffer
    int Fs;                                                   //Sample rate in "per minute"
    int warmup;                                         //Warmup period that should be neglected in minutes
    int period;                                         //Period that should be looked at for rolling mean
    int threshold;
    //Variables used for algorithm
    int benchmark3,benchmark5,benchmark9;
    int counter;
    int max3,max5,max9;
    int state;
    int maxcounter;
    int slopecount;
    int avg;
    //Sensors
    std::vector<int> sensor1;
    std::vector<int> sensor2;
    std::vector<int> sensor3;


};
```

```

#endif // BUFFER_H_INCLUDED
```

buffer.cpp

```
/*
 *  Project:        eNose software
 *  File:           buffer.cpp
 *  Author:         Maxim Chin-On
 *  Created on:     4 May 2021
 *  Description:    contains the functions used for the buffer class
 */
#include <iostream>
#include <algorithm>
using namespace std;
#include <vector>
#include "buffer.h"
//Initializes values
void Buffer::setBuffer(int g,int h, int i, int j)
{
    capacity           = g;
    Fs              = h;
    warmup          = i;
    period          = j;
    benchmark3         = 0;
    benchmark5        = 0;
    benchmark9        = 0;
    counter         = 0;
    threshold         = 60;
    state                  = 1;
}

//Adds data to the vector and gives new rolling mean
int Buffer::addData(int a, int b, int c)
{
        int detect = 0;
        int mean3;
        int size = sensor1.size();
        //Adding new data in the vectors
    sensor1.push_back(a);
    sensor2.push_back(b);
    sensor3.push_back(c);
    //If the vector is too large, delete the oldest entry
    if(sensor1.size() > unsigned(capacity*Fs))
    {
        sensor1.erase(sensor1.begin());
        sensor2.erase(sensor2.begin());
        sensor3.erase(sensor3.begin());
    }
    //Call algorithm to check for feces
    detect = algo(sensor1, sensor2, sensor3);
    return detect;
}

int Buffer::algo(std::vector<int> mq3, std::vector<int> mq5, std::vector<int> mq9)
{
        int size         = mq3.size();
        int diff3, diff5,diff9;
```

```
    int algod         =0;
    switch(state){
        case 0: //warm-up
            if(size > (warmup + period) * Fs)
            {
                state = 1;
            }
            break;
        case 1: //neutral
            int cvalue         = mq3[size-1];
            int means         = getMean(mq3);
            int limit         = threshold + means;
          algod = 0;
        counter = 0;
        max3 = 0;
        maxcounter =0;
        slopecount        = 0;
        //if the current value is above the limit, meaning there is a spike, the data is saved
        if(cvalue > limit)
            {
            benchmark3 = mq3[size-10];
            benchmark5 = mq5[size-10];
                benchmark9 = mq9[size-10];
          state = 2;
            }
        break;
    case 2: //waiting for max
        //if new value is significantly higher than the saved maximum it is used as the new maximum
        if(cvalue > max3*1.01)
        {
            max3 = mq3[size-1];
            max5 = mq5[size-1];
            max9 = mq9[size-1];
            maxcounter = 0;
            slopecount        = slopecount + 1;
        }
        else
        {
            slopecount        = 0;
            maxcounter = maxcounter + 1;
        }
        //if the data keeps rising without reaching a maximum for too long, the data does not represent feces
        if(slopecount > Fs * 5)
        {
            state = 5;
        }
        else
        {
                if(maxcounter > 10) //the saved maximum is sufficient to use for the algorithm
                {
                    avg   = (max3 + benchmark3) /2;
                    diff3 =  max3 - benchmark3;
                    diff5 =  max5 - benchmark5;
                    diff9 =  max9 - benchmark9;
                    if(diff9/diff3 < 0.45 ) //if the ratios check out we go to counter state
                    {
                        state = 3;
                    }
                    else
```

```
                                        {
                                                state = 5;
                                        }
                                }
                }
            break;
            case 3: //counter

                    if(cvalue > avg)
                    {
                            if(counter > 1 * Fs) //if counter reaches the chosen time it means we detected feces
                            {
                                    state = 4;
                            }
                            else //if signal disappears below avg it means it is not feces but probably a fart
                            {
                                    counter = counter + 1;
                            }
                    }
                    else
                    {
                            state = 1;
                    }
            break;
            case 4: //detect
                    algod = 1;
                if(cvalue < limit)
                {
                        state = 1;
                }
            break;
            case 5: //wait till normal again
                    if(cvalue < limit)
                    {
                            state = 1;
                    }
            break;
            default:
                    state = 1;
            break;
        }

        return algod;
}

//Prints the entire vector array
void Buffer::printData()
{
    int vecsize = sensor1.size(); //vectors have same size so this will suffice
    for(int i = 0; i < vecsize;i++)
    {
    // cout<< sensor1[i]<< " " << sensor2[i]<< " " << sensor3[i] << endl;
    }
}

//Based on input, gives certain sensor vector
std::vector<int> Buffer::getData(int sensor)
{
        //returning sensor array
```

```
    std::vector<int> result;
    switch(sensor){
    case 1:
        result = sensor1;
        break;
    case 2:
        result = sensor2;
        break;
    case 3:
        result = sensor3;
        break;
    default:
        //cout<< "wrong input!" << endl;
            break;
    }
    return result;

}

//Calculates rolling mean
int Buffer::getMean(std::vector<int> data)
{
    int mean = 0;
        int size = data.size();
        int start;
        int sum = 0;
        int diff;
        start = size - period*Fs;
        for(int i = start; i<size; i++)
        {
                sum = sum + data[i];
        }
        mean = sum/(period*Fs);
        return mean;
}

void Buffer::setValues(int thres, int g, int h, int i, int j)
{
        threshold         = thres;
        capacity           = g;
        Fs               = h;
        warmup           = i;
        period           = j;
}
Buffer::~Buffer()
{
}
```

A.2. Matlab Code

```
%%
% setting data out of the data2table function
% Authors: Maxim
% Date: 19/05/21
% Description: Takes the data read out and puts it in variables
    % List of devices for chosing the right device number
    devices = ["4C:11:AE:A5:F1:CC","84:CC:A8:60:6:C4","84:CC:A8
        :60:7:8","84:CC:A8:60:8:3C","84:CC:A8:60:9:4","84:CC:A8:60:B1
```

```
        :14","84:CC:A8:60:B1:70","84:CC:A8:60:B1:8C","84:CC:A8:60:B2
        :8","84:CC:A8:60:B2:A8","84:CC:A8:60:B6:DC","84:CC:A8:60:B7:18"];
    %Loading in csv file
    data = data2tables("localhost.csv");
    datapoint = 30; %data set
    device = 4; %device number
    row = [3,4,5,6,7,8]; %the three sensors
    % Converting the adc value to voltage
    amptovolt = 1100/4095;
    % Loading in the sensor data
    devicedata = data{1,device};
    devicenr = devices(device);
    dataset = devicedata{1,datapoint};
    timestamp = dataset(1,11);
    xl = length(dataset);
    %removing last value
    x = str2double(dataset(1:xl-1,:));
    %trimming if necessary for analysis
    % x = x(61000:end,:);
    sensor1 = x(:,row(1));
    sensor2 = x(:,row(2));
    sensor3 = x(:,row(3));
    sensor4 = x(:,row(4));
    sensor5 = x(:,row(5));
    sensor6 = x(:,row(6));

    y = 1:length(sensor1);
    hum     = x(:,1);
    temp    = x(:,2);
%%
% plotting data
% Authors: Maxim
% Date: 19/05/21
% Description: plots data
% tiledlayout(2,1)
period = 10;
Fs = 30;
%Version 1
% algor = algorithm(sensor3, 60, 30, 5, 30);
%Version 2
[algor,states] = algorithm2(sensor1,sensor2,sensor4,60,period,30);
warm1 = warmuptime(sensor1, 0.05, Fs);
warm2 = warmuptime(sensor2, 0.05, Fs);
warm3 = warmuptime(sensor3, 0.05, Fs);
warm4 = warmuptime(sensor4, 0.05, Fs);
% Normalizing
% mult31 = (max(sensor3(500:end))-min(sensor3(500:end)))/(max(sensor1(500:
    end))-min(sensor1(500:end)));
% mult32 = (max(sensor3(500:end))-min(sensor3(500:end)))/(max(sensor2(500:
    end))-min(sensor2(500:end)));
% mult34 = (max(sensor3(500:end))-min(sensor3(500:end)))/(max(sensor4(500:
    end))-min(sensor4(500:end)));
% sensor1 = sensor1*mult31;
% sensor2 = sensor2*mult32;
% sensor4 = sensor4*mult34;
% add31 = mean(sensor3(500:end))-mean(sensor1(500:end));
```

```
% add32 = mean(sensor3(500:end))-mean(sensor2(500:end));
% add34 = mean(sensor3(500:end))-mean(sensor4(500:end));
% Plotting
plot(y/30,sensor1*amptovolt);
hold on;
plot(y/30,sensor2*amptovolt);
plot(y/30,sensor3*amptovolt);
plot(y/30,sensor4*amptovolt);
plot(y, algor*amptovolt);
% xline(warm1*30);
% xline(warm2*30);
% xline(warm3*30);
hold off;
legend("MQ3","MQ5","MQ6","MQ9","Algo");
%title("Sensor data for starttime " + timestamp + "and device " + devicenr );
title("Algorithm applied to sensor data");
xlabel("Time [minutes]");
ylabel("Voltage [mV]");
%Humidity and Temperature plots
% plot(y,temp/10);
% ylabel("Humidity [%]");
% ylim([0 100]);
% hold on;
% legend("RH","Temperature");
%%
% Warmup time check
% Authors: Maxim
% Date: 24/05/21
% Description: Determines warmup time in minutes
function time = warmuptime(sensor, thres, Fs)
    min = 10000;
    counter = 0;
    for i = 10:1000
      mean = getmean(sensor, i, 11, 1);
      %if sensor value is 0 it cant get lower
      if sensor(i) == 0
          time = i/30;
          break;
      end
      %If sensor value is withing the threshold relative to the sensor
      %value 10 samples back, the sensor is considered warmed up
      if sensor(i)/sensor(i-10) < 1-thres
        min = mean;
        time = i;
      end
    end
end
%%
% Function: Applying algo on the data
% Authors: Maxim
% Data: 20/05/21
% Applying the algorithm on the data to verify it

function algo = algorithm(sensor, thres, countlim, period, Fs)
    scale = max(sensor);
```

```
    lengths = length(sensor);
    algo(1:lengths) = 0;
    maxv = 0;
    benchmark = 0;
    counter = 0;
    detect = 0;
    for i = 10:lengths
        cvalue = sensor(i);
        lvalue = sensor(i-1);
        means = getmean(sensor, i, period, Fs);
        limit = thres + means;
        %keeping track of maximum value
        if cvalue > maxv
            maxv = cvalue;
        end
        avg = (maxv + benchmark)/2;
        %if current value surpasses limit, start counting
        if cvalue >= limit

                if detect == 0
                    benchmark   = sensor(i-6);
                    detect              = 1;
                    counter     = 1;
                    maxv                = 0;
                else
                    %if value drop, return to no detection
                    if cvalue <avg
                        detect = 0;
                        poop = 0;
                        counter = 0;
                    else
                        counter = counter + 1;
                    end
                end
        else
            %if value drop, return to no detection
            if detect == 1
                if cvalue < limit
                    detect = 0;
                    poop = 0;
                    counter = 0;
                end
            end
        end
        %if counter reaches countlimit, it detects feces
        if counter >= countlim
            algo(i) = scale;
        end
    end
end
%%
% Function: Applying algo on the data
% Authors: Maxim
% Data: 28/05/21
% Applying the algorithm on the data to verify it

```

```
function [algo2, states] = algorithm2(mq3, mq5, mq9, thres, period, Fs)
    scale = max(mq3);
    lengths = length(mq3);
    algo2(1:lengths) = 0;
    states(1:lengths) = 0;
    state = 1;
    diff3 = 0;
    diff5 = 0;
    diff9 = 0;


    for i = 20:lengths
        cvalue = mq3(i);
        means = getmean(mq3, i, period, Fs);
        limit = thres + means;
        switch state
            case 1 %neutral
                %variables
                detect = 0;
                max3 = 0;
                counter = 0;
                maxcounter =0;
                slopecount     = 0;
                if cvalue > limit
                    benchmark3 = mq3(i-10);
                    benchmark5 = mq5(i-10);
                    benchmark9 = mq9(i-10);
                    state = 2;
                end
            case 2 %waiting for max
                %looking for maximum
                if cvalue > max3
                    max3 = mq3(i-1);
                    max5 = mq5(i-1);
                    max9 = mq9(i-1);
                    maxcounter = 0;
                    slopecount  = slopecount + 1;
                else
                    maxcounter = maxcounter + 1;
                end
                %if it takes too long to find maximum, it is not feces
                if(slopecount > Fs * 5)
                    state = 5;
                else
                    %if maximum is reached, check for ratios
                    if maxcounter > 10

                        avg = (max3 + benchmark3) /2;

                        diff3 = max3-benchmark3;
                        diff5 = max5-benchmark5;
                        diff9 = max9-benchmark9;

                        if diff9/diff3 < 0.5
                            %if ratios check out, go to count state
                            state = 3;
```

```
                            else
                                %otherwise go to wait for normal state
                                state = 5;
                            end
                        end
                    end
                case 3 %counter
                    %variables
                    if cvalue > avg
                        %if counter reaches threshold, go to detect state
                        if counter > 1 * Fs
                            state = 4;
                        else
                            counter = counter + 1;
                        end
                    else
                        %if value drops to fast go to neutral state
                        state = 1;

                    end
                case 4 %detect
                    %variables
                    detect = 1;
                    %once value drops go to wait till normal state
                    if cvalue < limit
                        state = 5;

                    end
                case 5 %wait till normal again

                    if cvalue < limit
                        state = 1;

                    end
                otherwise
                    state = 1;
            end
            %save results
            states(i) = state;
            if detect == 1
                algo2(i) = scale;
            end
        end
end
%%
% Function: getMean
% Authors: Maxim
% Date: 13/05/21
% Description: Function that gets the mean of sensor values
function means = getmean(sensor, i, period, Fs)
    means = 0;
    tsum = 0;
    length = period*Fs;
    %if there isnt enough data for normal mean use this
    if i <= length
        start = 6;
```

```
            length = i-6;
        else
            start = i - length;
        end
        %otherwise just compute mean normally
        for k = start:i
            tsum = tsum + sensor(k);
        end
        if tsum == 0 %avoiding dividing by zero
            means = 0;
        else
            means = tsum/(length+1);
        end
end
%%

% Function: data2table()
% Authors: Maxim, Gabriel and Mark
% Date: 13/05/21
% Description: Function that takes csv and seperates it per device



function  data = data2tables(datafile)
table = readtable(datafile);
% Find unique values
tags = [0 2 3 4 5 6 7 8 9 10 11 12];
% Seperate data per tag
for i=1:length(tags)
    deviceDATA = [];
    %Retrieve locations of unique tags
    INDEX = find(double(table.device_id) == tags(i));
    % Retrieve corresponding data
    sensor_1=table.sensor_1(INDEX);
    sensor_2=table.sensor_2(INDEX);
    sensor_3=table.sensor_3(INDEX);
    sensor_4=table.sensor_4(INDEX);
    sensor_5=table.sensor_5(INDEX);
    sensor_6=table.sensor_6(INDEX);
    time_uo=table.time_up(INDEX);
    algo=table.algo_h(INDEX);
    temprature=table.temp(INDEX);
    RH=table.rh(INDEX);
    TS=string(table.created_at(INDEX));

    % Find local minima
    i_local_minima=islocalmin(time_uo );
    local_min=find(i_local_minima==1);
    local_min=[1; local_min; length(time_uo)];

    for j=1: (length(local_min==1)-1)
        % create INDEX per run
        index=linspace(local_min(j), local_min(j+1), local_min(j+1)-
            local_min(j)+1);
        %put data in cell
        deviceDATA{j}=[temprature(index), RH(index), sensor_1(index),
```

```
                sensor_2(index), sensor_3(index), sensor_4(index), sensor_5(
                index), sensor_6(index), time_uo(index), algo(index),TS(index)
                ];
    end

    %Make cell aray per device
    data{i}= [deviceDATA, tags(i)];

end
end
```

A.3. Testing

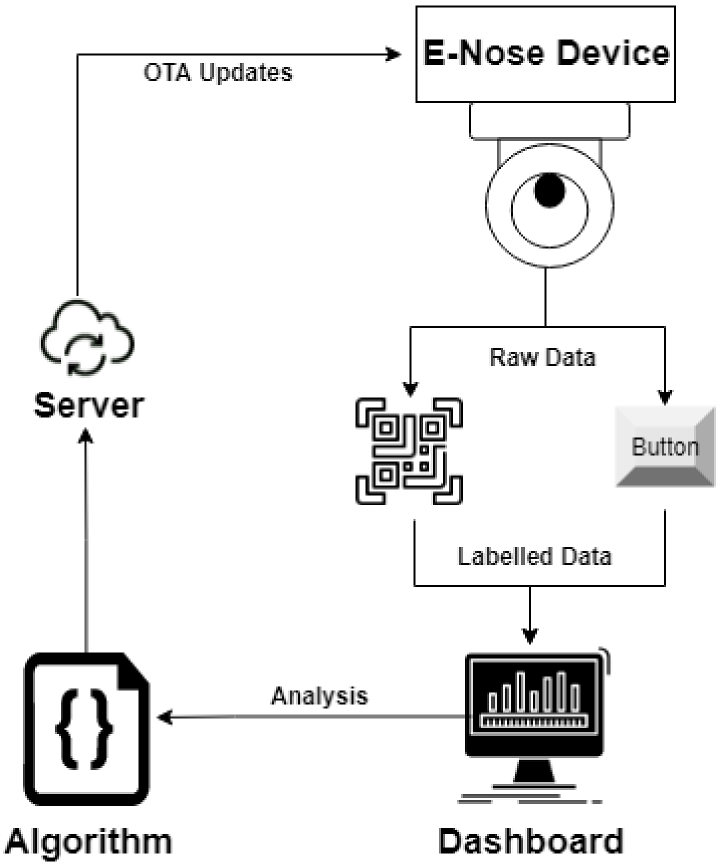


Figure A.1: The testing setup of phase 1.

B
Communication

B.1. Sensor read-out

sensor.h

```
/*
 *  Project:                eNose software
 *  File:                       sensor.h
 *  Author:                 Mark Fijneman
 *  Description:        Header function of sensor.cpp
 *  History:                 05/05/21 - created class
 *
*/

#ifndef MAIN_SENSOR_H_
#define MAIN_SENSOR_H_

#include "DHT.h"
#include "buffer.h"
#include <driver/adc.h>

// Define classes
class sensor {
  private:
        DHT dht;                                // DHT sensor class
        int sensor1;                       // MQ3 sensor
        int sensor2;                       // MQ4 sensor
        int sensor3;                       // MQ5 sensor
        int sensor4;                       // MQ6 sensor
        int sensor5;                       // MQ9 sensor
        int sensor6;                       // MQ135 sensor

        float temperature;               // Temp in celcius
        float rh;                               // Relative humidity in percentage
        int algo;                                // Algorithm (1=detected)
        int button;                               // Button (1=pressed)

        // Helper function which multisamples a ADC channel
        int multi_sampling(adc1_channel_t channel);

  public:
        Buffer buffer_sensor;        // Buffer and algorihm class

        sensor(void);                         // Constructor functions

        // Functions related to the DHT sensor
        void set_DHT(gpio_num_t pin);
        void read_DHT(void);

        // Function that is called by ISR when button is pressed
        void button_pressed(void);

```

```
        // Functions to read and update the sensor
        int read_sensor(int number);
        void update_sensor(void);
};

#endif /* MAIN_SENSOR_H_ */
```

sensor.cpp

```
/*
 *  Project:                eNose software
 *  File:                        sensor.cpp
 *  Author:                 Mark Fijneman
 *  Description:        Contains the functions of the class sensor
 *  History:                 05/05/21 - created class
 *                                   12/05/21 - added sensor adc readout
 *
 */

// Libaries
#include <driver/adc.h>
#include "driver/gpio.h"
#include "freertos/FreeRTOS.h"
#include "freertos/task.h"
#include "freertos/queue.h"
#include <algorithm>

// Local files
#include "esp_log.h"
#include "sensor.h"

//////////////////////////////////////////////////////////////////////////////////////////////////////
// Name: set_DHT
// Description: sets the DHT pin in the DHT class
// IO: Input the pin connect to the DHT sensor
void sensor::set_DHT(gpio_num_t pin) {
  dht.setDHTgpio(pin);
}

//////////////////////////////////////////////////////////////////////////////////////////////////////
// Name: read_dht
// Description: Retrieves the DHT values from the DHT class and store it inside the sensor class
// IO: N/A
void sensor::read_DHT(void) {
  int ret = dht.readDHT();
  dht.errorHandler(ret);
  temperature = dht.getTemperature();
  rh = dht.getHumidity();

  // The interval of whole process must be beyond 2 seconds !!
}

//////////////////////////////////////////////////////////////////////////////////////////////////////
// Name: read_sensor
// Description: getter function for the variables stored in the sensor class
// IO: input number, output value of the sensor
int sensor::read_sensor(int number) {
  int result;
```

```
  switch (number) {
  case 1:
    result = sensor1;
    break;
  case 2:
    result = sensor2;
    break;
  case 3:
    result = sensor3;
    break;
  case 4:
    result = sensor4;
    break;
  case 5:
    result = sensor4;
    break;
  case 6:
    result = sensor4;
    break;
  case 7:
    result = temperature * 10;
    break;
  case 8:
    result = rh * 10;
    break;
  case 9:
    result = algo;
    gpio_set_level(GPIO_NUM_13, algo); // update led
    break;
  case 10:
    // To normalise the button data, an one or zero is passed
    result = std::min(button, 1);
    break;
  default:
    // Pin number is not valid
    result = -1;
    ESP_LOGE("SENSOR_READ", "Invalid readout pin selected");
  }

  return result;
}

////////////////////////////////////////////////////////////////////////////////////////////////////
// Name: button_pressed
// Description: Function that is called by ISR when button is pressed
// IO: N/A
void sensor::button_pressed(void) {
  button = 20; // Button will be repeated 20 times
}

////////////////////////////////////////////////////////////////////////////////////////////////////
// Name: multi sampling
// Description: Obtains and averages the ADC value to reduce noise
// IO: input channel, output the averaged ADC value
int sensor::multi_sampling(adc1_channel_t channel) {
  int temp = 0;
  adc1_config_channel_atten(channel, ADC_ATTEN_DB_0);

  // Average the readings
```

```
  for (int i = 0; i < 10; i++) {
    temp += adc1_get_raw(channel);
    vTaskDelay(1 / portTICK_PERIOD_MS);
  }
  return temp / 10;
}

////////////////////////////////////////////////////////////////////////////////////////////////////////
// Name: update_sensor
// Description: Update the sensors, button and algo
// IO: N/A
void sensor::update_sensor(void) {
  adc1_channel_t channel;
  adc1_config_width(ADC_WIDTH_BIT_12); //12bit output

  // Read every sensor
  sensor1 = multi_sampling(ADC1_CHANNEL_0); //Sensor 1 ~ pin 36
  sensor2 = multi_sampling(ADC1_CHANNEL_3); //Sensor 2 ~ pin 39
  sensor3 = multi_sampling(ADC1_CHANNEL_4); //Sensor 3 ~ pin 32
  sensor4 = multi_sampling(ADC1_CHANNEL_5); //Sensor 4 ~ pin 33
  sensor5 = multi_sampling(ADC1_CHANNEL_4); //Sensor 3 ~ pin 32
  sensor6 = multi_sampling(ADC1_CHANNEL_5); //Sensor 4 ~ pin 33

  // Update Algoritm
  algo = buffer_sensor.addData(sensor1, sensor2, sensor4);

  // Update button
  if (button > 0) {
    button = button - 1;
  } else {
    button = 0;
  }
}

////////////////////////////////////////////////////////////////////////////////////////////////////////
// Name: sensor
// Description: constructor function
// IO: N/A
sensor::sensor(void) {

}
```

B.2. Wi-Fi connection

wifi_connect.h

```
/*
 *  Project:                Main software eNose
 *  File:                        wifi_connect.h
 *  Author:                 Mark Fijneman
 *  Description:        header of wifi_connect.cpp that initalise the wifi conneciton
 *  History:                 04/05/21 - Init
 */

#ifndef MAIN_WIFI_CONNECT_H_
#define MAIN_WIFI_CONNECT_H_

#include "connection.h"
```

```

// Define wifi_para struct
struct struct_wifi_para {
  bool connection_open;
  std::string SSID;
  std::string PASS;
  bool enterprise;
  std::string USERNAME;
  std::string cert;
  std::string IDENTITY;
};

// Define connecting classes
class wifi_connect {
  bool connection_open;
  std::string SSID;
  std::string PASS;
  bool enterprise;
  std::string USERNAME;
  std::string cert;
  std::string IDENTITY;
  std::string checksum;
  public:

  // Function that starts WiFi
  void wifi_init_sta(void);

  // Wifi parameters functions
  void set_wifi_para(std::string i_ssid, std::string i_pass);
  void set_wifi_para(std::string i_ssid, std::string i_identity, std::string i_username, std::string i_pass);
  void set_wifi_para(struct_wifi_para para);
  void set_wifi_para(bool connection_opened);
  struct struct_wifi_para get_wifi_para(void);

  // OTA related functions
  void nvs_wifi_para(bool ota);
  std::string get_checksum(void);

};

#endif /* MAIN_WIFI_CONNECT_H_ */
```

wifi_connect.cpp

```
/*
 *  Project:                Main software eNose
 *  File:                       wifi_connect.cpp
 *  Author:                 Mark Fijneman
 *  Description:        Contains the function that initalise the wifi conneciton
 *  History:                 04/05/21 - Project forked from https://github.com/espressif/esp-idf/
 *                                                                  Translated to c++
 *                                          12/05/21 - Added support for connection class
*/

#include "freertos/FreeRTOS.h"
#include "freertos/task.h"
#include "freertos/event_groups.h"
#include "esp_system.h"
```

```
#include "esp_wifi.h"
#include "esp_event.h"
#include "esp_log.h"
#include "nvs_flash.h"
#include "nvs.h"

#include "lwip/err.h"
#include "lwip/sys.h"
#include "esp_wpa2.h"
#include "esp_netif.h"
#include "esp_tls.h"
#include "mbedtls/base64.h"

#include "esp_wifi.h"
#include "esp_wpa2.h"


// CPP libaries
#include <algorithm>
#include <string.h>
#include <stdlib.h>

// Local files
#include "wifi_connect.h"
#include "connection.h"
#include "config.h"



#define ESP_MAXIMUM_RETRY  50

/* FreeRTOS event group to signal when we are connected*/
static EventGroupHandle_t s_wifi_event_group;
/* esp netif object representing the WIFI station */
static esp_netif_t *sta_netif = NULL;

/* The event group allows multiple bits for each event, but we only care about two events:
 * - we are connected to the AP with an IP
 * - we failed to connect after the maximum amount of retries */
#define WIFI_CONNECTED_BIT BIT0
#define WIFI_FAIL_BIT      BIT1

static const char *TAG = "WIFI_connect";

static int s_retry_num = 0;

static void event_handler(void* arg, esp_event_base_t event_base,
                                int32_t event_id, void* event_data)
{
    if (event_base == WIFI_EVENT && event_id == WIFI_EVENT_STA_START) {
        esp_wifi_connect();
    } else if (event_base == WIFI_EVENT && event_id == WIFI_EVENT_STA_DISCONNECTED) {
        if (s_retry_num < ESP_MAXIMUM_RETRY) {
            esp_wifi_connect();
            s_retry_num++;
            ESP_LOGI(TAG, "Retry to connect to the AP");
        } else {
            xEventGroupSetBits(s_wifi_event_group, WIFI_FAIL_BIT);
        }
```

```
        ESP_LOGI(TAG,"Connect to the AP fail");
    } else if (event_base == IP_EVENT && event_id == IP_EVENT_STA_GOT_IP) {
        ip_event_got_ip_t* event = (ip_event_got_ip_t*) event_data;
        ESP_LOGI(TAG, "Got ip:" IPSTR, IP2STR(&event->ip_info.ip));
        s_retry_num = 0;
        xEventGroupSetBits(s_wifi_event_group, WIFI_CONNECTED_BIT);
    }
}

void wifi_connect::wifi_init_sta(void)
{
    s_wifi_event_group = xEventGroupCreate();
    ESP_LOGI(TAG, "Wifi_init_start");
    ESP_ERROR_CHECK(esp_netif_init());

    ESP_ERROR_CHECK(esp_event_loop_create_default());
    sta_netif = esp_netif_create_default_wifi_sta();
    assert(sta_netif);

    wifi_init_config_t cfg = WIFI_INIT_CONFIG_DEFAULT();
    ESP_ERROR_CHECK(esp_wifi_init(&cfg));

    esp_event_handler_instance_t instance_any_id;
    esp_event_handler_instance_t instance_got_ip;
    ESP_ERROR_CHECK(esp_event_handler_instance_register(WIFI_EVENT,
                                                        ESP_EVENT_ANY_ID,
                                                        &event_handler,
                                                        NULL,
                                                        &instance_any_id));
    ESP_ERROR_CHECK(esp_event_handler_instance_register(IP_EVENT,
                                                        IP_EVENT_STA_GOT_IP,
                                                        &event_handler,
                                                        NULL,
                                                        &instance_got_ip));


    // Changed to be used in c++ rather than c
    //Allocate storage for the struct
    wifi_config_t wifi_config = {};

    //Assign ssid & password strings by copying in uint8_t arrays
    strcpy((char*)wifi_config.sta.ssid, SSID.c_str());
    strcpy((char*)wifi_config.sta.password, PASS.c_str());
//  wifi_config.sta.threshold.authmode = WIFI_AUTH_WPA_WPA2_PSK;  // Not set right yet
    wifi_config.sta.pmf_cfg.capable=true;
    wifi_config.sta.pmf_cfg.required=false;
    ESP_LOGI(TAG, "Setting WiFi configuration SSID %s...", wifi_config.sta.ssid);

    ESP_ERROR_CHECK(esp_wifi_set_mode(WIFI_MODE_STA) );
    ESP_ERROR_CHECK(esp_wifi_set_config(ESP_IF_WIFI_STA, &wifi_config) );

    if(enterprise==true){
            ESP_ERROR_CHECK( esp_wifi_sta_wpa2_ent_set_identity((uint8_t *)CONST_IDENTITY, strlen(CONST_IDENTITY)) )
            ESP_ERROR_CHECK( esp_wifi_sta_wpa2_ent_set_username((uint8_t *)CONST_USERNAME, strlen(CONST_USERNAME)) )
            ESP_ERROR_CHECK( esp_wifi_sta_wpa2_ent_set_password((uint8_t *)CONST_PASS, strlen(CONST_PASS)) );
            ESP_ERROR_CHECK( esp_wifi_sta_wpa2_ent_enable() );


    }
```

```

    ESP_ERROR_CHECK(esp_wifi_start() );
    /* Waiting until either the connection is established (WIFI_CONNECTED_BIT) or connection failed for the maximum
     * number of re-tries (WIFI_FAIL_BIT). The bits are set by event_handler() (see above) */
    EventBits_t bits = xEventGroupWaitBits(s_wifi_event_group,
            WIFI_CONNECTED_BIT | WIFI_FAIL_BIT,
            pdFALSE,
            pdFALSE,
            portMAX_DELAY);

    /* xEventGroupWaitBits() returns the bits before the call returned, hence we can test which event actually
     * happened. */
    if (bits & WIFI_CONNECTED_BIT) {
        ESP_LOGI(TAG, "Connected to ap SSID:%s password:%s",
                        SSID.c_str(), PASS.c_str());
    } else if (bits & WIFI_FAIL_BIT) {
        ESP_LOGI(TAG, "Failed to connect to SSID:%s, password:%s",
                        SSID.c_str(), PASS.c_str());
    } else {
        ESP_LOGE(TAG, "UNEXPECTED EVENT");
    }

    /* The event will not be processed after unregister */
    ESP_ERROR_CHECK(esp_event_handler_instance_unregister(IP_EVENT, IP_EVENT_STA_GOT_IP, instance_got_ip));
    ESP_ERROR_CHECK(esp_event_handler_instance_unregister(WIFI_EVENT, ESP_EVENT_ANY_ID, instance_any_id));
    vEventGroupDelete(s_wifi_event_group);
}



///////////////////////////////////////////////////////////////////////////////////////
// Wifi parameters functions
void wifi_connect::set_wifi_para(bool connection_opened)
{
        connection_open = connection_open;
}

void wifi_connect::set_wifi_para(std::string i_ssid, std::string i_pass)
{
        SSID = i_ssid;
        PASS = i_pass;
        enterprise = false;
}

void wifi_connect::set_wifi_para(std::string i_ssid, std::string i_identity, std::string i_username, std::string i_pass)
{
        enterprise = true;
        SSID = i_ssid;
        IDENTITY=i_identity;
        USERNAME=i_username;
        PASS = i_pass;
}

void wifi_connect::set_wifi_para(struct_wifi_para para){
        connection_open=para.connection_open;
        SSID=                        para.SSID;
        PASS=                        para.PASS;
        enterprise=                para.enterprise;
        USERNAME=                para.USERNAME;
```

```
        cert=                        para.cert;
        IDENTITY=                para.IDENTITY;
}

struct struct_wifi_para wifi_connect::get_wifi_para(void)
{
        struct_wifi_para para;
        return para;
}


//////////////////////////////////////////////////////////////////////////
// Non voiliate wifi parameter retrieval
void wifi_connect::nvs_wifi_para(bool ota)
{
        esp_err_t ret;
        nvs_handle handle;
        ret = nvs_open("wifi_para", NVS_READWRITE, &handle);
    if (ret != ESP_OK) {
            ESP_LOGE("NVS_WiFi", "Error opening NVS handle (%s)", esp_err_to_name(ret));
    }
    else{
            ESP_LOGI("NVS_WiFi", "NVS WiFi space opened");

            char SSID_buf[30];
            char PASS_buf[30];
            size_t SSID_size;
            size_t PASS_size;

            // Obtain or write SSID
        nvs_get_str(handle, "SSID", NULL, &SSID_size );
        ret = nvs_get_str(handle, "SSID", (char *)&SSID_buf, &SSID_size);
        if( (ret==ESP_ERR_NVS_NOT_FOUND) || !UPDATE || ota){

            // SSID is not present yet
                ESP_LOGI("NVS_WiFi", "SSID written to NVS");
            strcpy (SSID_buf, CONST_SSID);
            ret = nvs_set_str(handle, "SSID", (const char*)SSID_buf);
            if(ret!=ESP_OK){
                       ESP_LOGE("NVS_WiFi", "Error writing SSID (%s)", esp_err_to_name(ret));
            }
            ret = nvs_commit(handle);
            if(ret!=ESP_OK){
                    ESP_LOGE("NVS_WiFi", "Error commiting SSID (%s)", esp_err_to_name(ret));
            }
        }
        else if(ret== ESP_OK){
                // Read succesfull
                ESP_LOGI("NVS_WiFi", "Read of SSID succesful");
        }
        else{
                ESP_LOGE("NVS_WiFi", "Error reading SSID (%s)", esp_err_to_name(ret));
        }

            // Obtain or write PASS
        nvs_get_str(handle, "PASS", NULL, &PASS_size );
        ret = nvs_get_str(handle, "PASS", (char *)&PASS_buf, &PASS_size);
        if(ret==ESP_ERR_NVS_NOT_FOUND|| !UPDATE || ota){
                // SSID is not present yet
```

```
                ESP_LOGI("NVS_WiFi", "PASS written to NVS");
                strcpy(PASS_buf, CONST_PASS);
            ret = nvs_set_str(handle, "PASS", (const char*)PASS_buf);
            if(ret!=ESP_OK){
                    ESP_LOGE("NVS_WiFi", "Error writing PASS (%s)", esp_err_to_name(ret));
            }
            ret = nvs_commit(handle);
            if(ret!=ESP_OK){
                    ESP_LOGE("NVS_WiFi", "Error commiting PASS (%s)", esp_err_to_name(ret));
            }
        }
        else if(ret== ESP_OK){
                // Read succesfull
                ESP_LOGI("NVS_WiFi", "Read of PASS succesful");
        }
        else{
                ESP_LOGE("NVS_WiFi", "Error reading PASS (%s)", esp_err_to_name(ret));
        }

        set_wifi_para(SSID_buf, PASS_buf);
        nvs_close(handle);
    }
}

std::string wifi_connect::get_checksum(void){
        if(checksum.empty()){
                char *payload = "Hello SHA 256 from test";
                char shaResult[32];
                unsigned char output[64];
                size_t outlen;

                // Generate hash
                mbedtls_md_context_t ctx;
                mbedtls_md_type_t md_type = MBEDTLS_MD_SHA256;

                const size_t payloadLength = strlen(payload);

                mbedtls_md_init(&ctx);
                mbedtls_md_setup(&ctx, mbedtls_md_info_from_type(md_type), 0);
                mbedtls_md_starts(&ctx);
                mbedtls_md_update(&ctx, (const unsigned char *) payload, payloadLength);
                mbedtls_md_finish(&ctx, (unsigned char*)shaResult);
                mbedtls_md_free(&ctx);

                // Base64 encode
                mbedtls_base64_encode((unsigned char*)output, 64, &outlen, (unsigned char*)shaResult, 32);
                ESP_LOGI("wifi checksum", "%s", output);
                std::string output_str (output, output + sizeof output / sizeof output[0]);
                checksum=output_str;
        }

        return checksum;

}
```

B.3. Webserver

OPENAPI 3.0 documentation

```yaml
openapi: 3.0.0
info:
  version: 1.0.0
  title: sensorDATA
servers:
  - url: 'https://url.com'
paths:
  /measurements.php:
    post:
      summary: 'Endpoint: takes the sensor data of esp32 and stores it in the db'
      requestBody:
        required: true
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/sensor_data'
      responses:
        '200':
          description: 'Measurements succeeded: return algorihm parameters'
        default:
          description: Unexpected error
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/Error'
  /OTA/parameters.php:
    get:
      summary: 'Endpoint: retrieves the latest algorithm parameters from the data base and parses it to the client'
      responses:
        '200':
          description: 'Measurements succeeded: return algorihm parameters'
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/algo_para'
        default:
          description: Unexpected error
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/Error'
  /OTA/firmware.php:
    get:
      summary: 'Endpoint: retrieves the latest version number and a checksum of the OTA firmware'
      responses:
        '200':
          description: 'OTA firmware found on the server'
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/firmware_checks'
        default:
          description: Unexpected error
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/Error'
```

```yaml
    post:
      summary: 'Retrieve part of the OTA firmware'
      requestBody:
        required: true
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/firmware_request'

      responses:
        '200':
          description: 'Piece found of the OTA firmware'
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/firmware_response'
        default:
          description: Unexpected error
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/Error'
  /OTA/wifi.php:
    post:
      summary: 'Retrieve part of the OTA firmware'
      requestBody:
        required: true
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/wifi_request'

      responses:
        '200':
          description: 'Wifi is not up to date and new credentials are returned'
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/wifi_response'
        default:
          description: Unexpected error
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/Error'

components:
  schemas:
    Error:
      type: object
      required:
        - message
      properties:
        message:
          type: string
    sensor_data:
      type: object
      required:
        - tag
```

```
        - T
        - C
        - S1
        - S2
        - S3
        - S4
        - S5
        - S6
        - UP
        - A
        - B
      properties:
        tag:
          description: MAC address of the ESP32 seperate by an semicolon
          type: string
          pattern: '^([0-9A-FA-F]{2}[:]){5}([0-9A-FA-F]{2})$'
        T:
          description: Temperature measurement of ESP32 multiplied by ten
          type: integer
          format: uint16
          minimum: 0
          maximum: 600
        H:
          description: Humidity measurement of ESP32 multiplied by ten
          type: integer
          format: uint16
          minimum: 0
          maximum: 1000
        S1:
          description: Sensor 1 measurement of ESP32
          type: integer
          format: uint16
          minimum: 0
          maximum: 4096
        S2:
          description: Sensor 2 measurement of ESP32
          type: integer
          format: uint16
          minimum: 0
          maximum: 4096
        S3:
          description: Sensor 3 measurement of ESP32
          type: integer
          format: uint16
          minimum: 0
          maximum: 4096
        S4:
          description: Sensor 4 measurement of ESP32
          type: integer
          format: uint16
          minimum: 0
          maximum: 4096
        S5:
          description: Sensor 5 measurement of ESP32
          type: integer
          format: uint16
          minimum: 0
          maximum: 4096
        S6:
```

```yaml
          description: Sensor 6 measurement of ESP32
          type: integer
          format: uint16
          minimum: 0
          maximum: 4096
        UP:
          description: Uptime of the esp32 in seconds
          type: integer
          format: uint32
          minimum: 0
        A:
          description: Algorithm value of esp32
          type: integer
          format: uint16
          minimum: 0
          maximum: 4096
        B:
          description: Algorithm value of esp32
          type: integer
          format: uint16
          minimum: 0
          maximum: 4096
    algo_para:
      description: Offset of the to-be-received datafile
      type: object
      required:
        - capacity
        - fs
        - warmup
        - period
        - threshold
      properties:
        capacity:
          type: integer
          format: uint64
          minimum: 0
        fs:
          type: integer
          format: uint64
          minimum: 0
        warmup:
          type: integer
          format: uint64
          minimum: 0
        period:
          type: integer
          format: uint64
          minimum: 0
        threshold:
          type: integer
          format: uint64
          minimum: 0
    firmware_request:
      description: Parameters to receive a part of the firmware update
      type: object
      required:
        - offset
        - length
      properties:
```

```
        offset:
          description: Offset of the to-be-received datafile
          type: integer
          format: uint64
          minimum: 0
        length:
          description: Length of the to-be-received datafile
          type: integer
          format: uint64
          minimum: 0
    firmware_response:
      description: The response which contains the actual data of the firmware OTA update
      type: object
      required:
        - data
      properties:
        offset:
          description: String in base64 that contains the actual data of the firmware with an offset and length
          type: string
          format: byte
    firmware_checks:
      description: Retrieves the current OTA firmware version along with an checksum
      type: object
      required:
        - version
        - checksum
      properties:
        version:
          description: Current version of the OTA firmware
          type: integer
          format: uint64
          minimum: 0
        checksum:
          description: SHA256 checksum of the total firmware used to check the integrity of the update
          type: string
          format: byte
    wifi_request:
      description: Check if wifi settings are valid and up-to-date
      type: object
      required:
        - tag
        - checksum
      properties:
        tag:
          description: MAC address of the ESP32 seperate by an semicolon
          type: string
          pattern: '^([0-9A-FA-F]{2}[:]){5}([0-9A-FA-F]{2})$'
        checksum:
          description: SHA256 checksum of wifi settings
          type: string
          format: byte
    wifi_response:
      description: Check if wifi settings are valid and up-to-date
      type: object
      required:
        - checksum
        - ssid
        - pass
      properties:
```

```yaml
        location:
          description: Location number
          type: integer
          format: uint64
          minimum: 0
        checksum:
          description: SHA256 checksum of wifi settings
          type: string
          format: byte
        ssid:
          description: SSID of wifi network
          type: string
        pass:
          description: PASS of wifi network
          type: string
        cert:
          description: PASS of wifi network
          type: string
        identity:
          description: PASS of wifi network
          type: string
```


SQL database structure export

```sql
-- phpMyAdmin SQL Dump
-- version 5.0.3
-- https://www.phpmyadmin.net/
--
-- Host: localhost:3306
-- Gegenereerd op: 16 jun 2021 om 10:44
-- Serverversie: 10.2.36-MariaDB
-- PHP-versie: 7.3.26

SET SQL_MODE = "NO_AUTO_VALUE_ON_ZERO";
START TRANSACTION;
SET time_zone = "+00:00";


/*!40101 SET @OLD_CHARACTER_SET_CLIENT=@@CHARACTER_SET_CLIENT */;
/*!40101 SET @OLD_CHARACTER_SET_RESULTS=@@CHARACTER_SET_RESULTS */;
/*!40101 SET @OLD_COLLATION_CONNECTION=@@COLLATION_CONNECTION */;
/*!40101 SET NAMES utf8mb4 */;

--
-- Database: `poop_db`
--

-- --------------------------------------------------------

--
-- Tabelstructuur voor tabel `algo_settings`
--

CREATE TABLE `algo_settings` (
  `ID` int(10) UNSIGNED NOT NULL,
  `capacity` int(11) NOT NULL,
  `fs` int(11) NOT NULL,
  `warmup` int(11) NOT NULL,
```

```sql
  `period` int(11) NOT NULL,
  `threshold` int(11) NOT NULL
) ENGINE=InnoDB DEFAULT CHARSET=utf8;

-- --------------------------------------------------------

--
-- Tabelstructuur voor tabel `devices`
--

CREATE TABLE `devices` (
  `tag` text NOT NULL,
  `device_id` int(10) UNSIGNED NOT NULL,
  `description` tinytext NOT NULL,
  `location` int(10) UNSIGNED NOT NULL,
  `version` int(11) NOT NULL,
  `algo` tinyint(4) NOT NULL
) ENGINE=InnoDB DEFAULT CHARSET=utf8;

-- --------------------------------------------------------

--
-- Tabelstructuur voor tabel `location`
--

CREATE TABLE `location` (
  `location` int(10) UNSIGNED NOT NULL,
  `description` text NOT NULL,
  `wifi_type` enum('WPA2-PSK','WPA2-enterprise') NOT NULL,
  `ssid` tinytext NOT NULL,
  `pass` tinytext NOT NULL,
  `identity` tinytext NOT NULL,
  `username` tinytext NOT NULL,
  `wifi_checksum` text NOT NULL,
  `phone` tinytext NOT NULL
) ENGINE=InnoDB DEFAULT CHARSET=utf8;

-- --------------------------------------------------------

--
-- Tabelstructuur voor tabel `measurements`
--

CREATE TABLE `measurements` (
  `id` int(10) UNSIGNED NOT NULL,
  `device_id` int(10) UNSIGNED NOT NULL,
  `temp` smallint(3) NOT NULL,
  `rh` smallint(3) NOT NULL,
  `sensor_1` smallint(4) NOT NULL,
  `sensor_2` smallint(4) NOT NULL,
  `sensor_3` smallint(4) NOT NULL,
  `sensor_4` smallint(4) NOT NULL,
  `sensor_5` smallint(4) NOT NULL,
  `sensor_6` smallint(4) NOT NULL,
  `time_up` int(11) NOT NULL,
  `algo_h` tinyint(2) NOT NULL,
  `button` tinyint(1) NOT NULL,
  `created_at` datetime NOT NULL
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

```
--
-- Indexen voor geëxporteerde tabellen
--

--
-- Indexen voor tabel `algo_settings`
--
ALTER TABLE `algo_settings`
  ADD PRIMARY KEY (`ID`);

--
-- Indexen voor tabel `devices`
--
ALTER TABLE `devices`
  ADD PRIMARY KEY (`tag`(18)) USING BTREE,
  ADD UNIQUE KEY `device_id` (`device_id`),
  ADD KEY `location_foreign` (`location`);

--
-- Indexen voor tabel `location`
--
ALTER TABLE `location`
  ADD PRIMARY KEY (`location`);

--
-- Indexen voor tabel `measurements`
--
ALTER TABLE `measurements`
  ADD PRIMARY KEY (`id`),
  ADD KEY `device_id_foreign` (`device_id`),
  ADD KEY `created_at` (`created_at`);

--
-- AUTO_INCREMENT voor geëxporteerde tabellen
--

--
-- AUTO_INCREMENT voor een tabel `algo_settings`
--
ALTER TABLE `algo_settings`
  MODIFY `ID` int(10) UNSIGNED NOT NULL AUTO_INCREMENT;

--
-- AUTO_INCREMENT voor een tabel `devices`
--
ALTER TABLE `devices`
  MODIFY `device_id` int(10) UNSIGNED NOT NULL AUTO_INCREMENT;

--
-- AUTO_INCREMENT voor een tabel `location`
--
ALTER TABLE `location`
  MODIFY `location` int(10) UNSIGNED NOT NULL AUTO_INCREMENT;

--
-- AUTO_INCREMENT voor een tabel `measurements`
--
ALTER TABLE `measurements`
```

```
  MODIFY `id` int(10) UNSIGNED NOT NULL AUTO_INCREMENT;

--
-- Beperkingen voor geëxporteerde tabellen
--

--
-- Beperkingen voor tabel `devices`
--
ALTER TABLE `devices`
  ADD CONSTRAINT `location_foreign` FOREIGN KEY (`location`) REFERENCES `location` (`location`);

--
-- Beperkingen voor tabel `measurements`
--
ALTER TABLE `measurements`
  ADD CONSTRAINT `device_id_foreign` FOREIGN KEY (`device_id`) REFERENCES `devices` (`device_id`);
COMMIT;

/*!40101 SET CHARACTER_SET_CLIENT=@OLD_CHARACTER_SET_CLIENT */;
/*!40101 SET CHARACTER_SET_RESULTS=@OLD_CHARACTER_SET_RESULTS */;
/*!40101 SET COLLATION_CONNECTION=@OLD_COLLATION_CONNECTION */;
```

B.4. Protocol

http_post.h

```
/*
 *  Project:                Main software eNose
 *  File:                        http_post.h
 *  Author:                 Mark Fijneman
 *  Description:        Contain functions to construct the JSON content and preform the http post through POSIX-socket
 *  History:                 10/05/21:         Init
*/


#ifndef MAIN_HTTP_POST_CPP_
#define MAIN_HTTP_POST_CPP_
#include "connection.h"
#include "sensor.h"
#include <string.h>
#include "esp_tls.h"


struct https_data{
        std::string  web_full_url;
        std::string REQUEST;
        esp_tls_cfg_t* settings;
        std::string received_message;
};

class http_request{
        // Private variables
        std::string  web_port;
        std::string  web_path;
        std::string  web_url;
        std::string  web_full_url;
        bool secure_connection;
```

```
        uint8_t mac_address[6]= { 0 };
        esp_tls_cfg_t settings;

        //Helper for http requesr functions
        uint8_t* get_mac_address(void);
        std::string generate_body(sensor* sensor_main);
        std::string generate_body(int offset, int length);
        std::string generate_header(int content_length);
        std::string http_get(void);

        //Helpers to decode http response
        std::string extract_json(std::string message);
        std::string extract_json(std::string message, std::string component);
public:

        std::string https_task_start (void);
        esp_tls_cfg_t https_settings(void);
        http_request(void);
        void set_config(std::string url, std::string port, std::string path);
        struct struct_http_config get_config(void);

        // Sensor data + algo update
        void post(sensor* sensor_main);

        // wifi ota
        void wifi_ota(std::string checksum, struct_wifi_para *wifi_settings, bool *current);

        // Firmware update related
        std::string check_ota(int version, int* remote_version);
        std::string get_ota(int offset, int length);
};




#endif /* MAIN_HTTP_POST_CPP_ */
```

http_post.cpp

```
/*
 *  Project:                Main software eNose
 *  File:                           http_post
 *  Author:                 Mark Fijneman
 *  Description:        Contain functions to construct the JSON content and preform the http post through POSIX-socket
 *  History:                 10/05/21:         Forked from ESP http_request example
 *                                   11/05/21:         Added JSON constructor + Variable message
 *                                   01/06/21:         Added https support
*/

// Deault libaries
#include <string.h>
#include "freertos/FreeRTOS.h"
#include "freertos/task.h"
#include "esp_system.h"
#include "esp_wifi.h"
#include "esp_event.h"
#include "esp_log.h"
#include "nvs_flash.h"
```

```
#include <iostream>
#include <string>

// HTTP post specific libaries
#include "protocol_examples_common.h"
#include "lwip/err.h"
#include "lwip/sockets.h"
#include "lwip/sys.h"
#include "lwip/netdb.h"
#include "lwip/dns.h"
#include "esp_timer.h"
#include "esp_tls.h"
#include "esp_crt_bundle.h"

// Local libaries
#include "wifi_connect.h"
#include "sensor.h"
#include "http_post.h"

#include "config.h"

static const char *TAG = "HTTP_POST";

////////////////////////////////////////////////////////////////////////////////////////////////////////
// Name: https_get and http_get
// Description: Perform the actual http(s) communication
// IO: Outputs string containing http request response
extern const uint8_t server_root_cert_pem_start[] asm("_binary_server_root_cert_pem_start");
extern const uint8_t server_root_cert_pem_end[]   asm("_binary_server_root_cert_pem_end");
static char REQUEST[600]; // Place holder for actual request for later development

void https_get(void *Struct)
{
        https_data * data = (https_data *) Struct;
        std::cout<<data->web_full_url;
        std::cout<<data->REQUEST;
                char buf[512];
                int ret, len;
                std::string received_message;
                esp_tls_cfg_t *cfg = data->settings;
                ESP_LOGD(TAG, "start new connection");


                struct esp_tls *tls = esp_tls_conn_http_new(data->web_full_url.c_str(), cfg);

                if (tls != NULL) {
                        ESP_LOGD(TAG, "Connection established...");
                } else {
                        ESP_LOGE(TAG, "Connection failed...");
                        esp_tls_conn_destroy(tls);
                        //received_message=https_get();
                        //return
                }

                size_t written_bytes = 0;
                do {
                        ret = esp_tls_conn_write(tls,
                                        data->REQUEST.c_str() + written_bytes,
                                         sizeof(data->REQUEST.c_str()) - written_bytes);
```

```
				if (ret >= 0) {
					ESP_LOGD(TAG, "%d bytes written", ret);
					written_bytes += ret;
				} else if (ret != ESP_TLS_ERR_SSL_WANT_READ  && ret != ESP_TLS_ERR_SSL_WANT_WRITE) {
					ESP_LOGE(TAG, "esp_tls_conn_write  returned: [0x%02X](%s)", ret, esp_err_to_name(ret));
				}
			} while (written_bytes < sizeof(REQUEST));

			ESP_LOGD(TAG, "Reading HTTP response...");

			do {
				len = sizeof(buf) - 1;
				bzero(buf, sizeof(buf));
				ret = esp_tls_conn_read(tls, (char *)buf, len);
				received_message.append(buf);

				if (ret == ESP_TLS_ERR_SSL_WANT_WRITE  || ret == ESP_TLS_ERR_SSL_WANT_READ) {
					continue;
				}

				if (ret < 0) {
					ESP_LOGE(TAG, "esp_tls_conn_read  returned [-0x%02X](%s)", -ret, esp_err_to_name(ret));
					break;
				}

				if (ret == 0) {
					ESP_LOGD(TAG, "connection closed");
					break;
				}

				len = ret;
				ESP_LOGD(TAG, "%d bytes read", len);


			} while (1);

			esp_tls_conn_destroy(tls);

	//return received_message;
}

std::string http_request::https_task_start(void){
	https_data https;
	https.REQUEST=REQUEST;
	https.web_full_url=web_full_url;
	https.settings=&settings;

	xTaskCreate(&https_get, "https_get", 8192, (void*)&https, 5, NULL);

	return "placeholder";
}

std::string http_request::http_get(void)
{
	const struct addrinfo hints = {
		.ai_family = AF_INET,
		.ai_socktype = SOCK_STREAM,
	};
	struct addrinfo *res;
```

```
        struct in_addr *addr;
        int s, r;
        char recv_buf[64];
        std::string received_message;

                // Obtain socket and adress information
                int err = getaddrinfo(web_url.c_str(), web_port.c_str(), &hints, &res);

                if(err != 0 || res == NULL) {
                        ESP_LOGE(TAG, "DNS lookup failed err=%d res=%p", err, res);
                        vTaskDelay(1000 / portTICK_PERIOD_MS);
                        //continue;
                }

                /* Code to print the resolved IP.
                   Note: inet_ntoa is non-reentrant, look at ipaddr_ntoa_r for "real" code */
                addr = &((struct sockaddr_in *)res->ai_addr)->sin_addr;
                ESP_LOGD(TAG, "DNS lookup succeeded. IP=%s", inet_ntoa(*addr));

                s = socket(res->ai_family, res->ai_socktype, 0);
                if(s < 0) {
                        ESP_LOGE(TAG, "... Failed to allocate socket.");
                        freeaddrinfo(res);
                        vTaskDelay(1000 / portTICK_PERIOD_MS);
           //         continue;
                }
                ESP_LOGD(TAG, "... allocated socket");

                if(connect(s, res->ai_addr, res->ai_addrlen) != 0) {
                        ESP_LOGE(TAG, "... socket connect failed errno=%d", errno);
                        close(s);
                        freeaddrinfo(res);
                        vTaskDelay(4000 / portTICK_PERIOD_MS);
   //                 continue;
                }

                ESP_LOGD(TAG, "... connected");
                freeaddrinfo(res);


                if ( write(s, REQUEST, strlen(REQUEST))< 0) {
                        ESP_LOGE(TAG, "... socket send failed");
                        close(s);
                        vTaskDelay(4000 / portTICK_PERIOD_MS);
        //                continue;
                }
                ESP_LOGD(TAG, "... socket send success");

                struct timeval receiving_timeout;
                receiving_timeout.tv_sec = 5;
                receiving_timeout.tv_usec = 0;
                if (setsockopt(s, SOL_SOCKET, SO_RCVTIMEO, &receiving_timeout,
                                sizeof(receiving_timeout)) < 0) {
                        ESP_LOGE(TAG, "... failed to set socket receiving timeout");
                        close(s);
                        vTaskDelay(4000 / portTICK_PERIOD_MS);
  //                  continue;
                }
                ESP_LOGD(TAG, "... set socket receiving timeout success");
```

```
                /* Read HTTP response */

                do {
                        bzero(recv_buf, sizeof(recv_buf));
                        r = read(s, recv_buf, sizeof(recv_buf)-1);
                        received_message.append(recv_buf);
                } while(r > 0);

                close(s);
                return received_message;
}

//////////////////////////////////////////////////////////////////////////////////////////////////////////
// Name: extract_json
// Description: Extracts information from the http response
// IO: Input string containing http request response, Output either body of https response or specific parts
std::string extract_json(std::string message, std::string component)
{
        std::string component_json_form= "\"" + component + "\":";

        int pos=message.find(component_json_form)+component_json_form.size();        // remove compent string
        int end=pos+message.substr(pos).find(",")-1;         // Remove ,

        std::string result=message.substr (pos+1,end-pos-1); // Remove quotation marks
        return result;
}

std::string http_request::extract_json(std::string message, std::string component)
{
        std::string component_json_form= "\"" + component + "\":";

        int pos=message.find(component_json_form)+component_json_form.size();        // remove compent string
        int end=pos+message.substr(pos).find(",")-1;         // Remove ,

        std::string result=message.substr (pos+1,end-pos-1); // Remove quotation marks
        return result;
}

std::string http_request::extract_json(std::string message)
{
        int start_json=message.find("{");
        int end_json=message.find("}");
        std::string result=message.substr (start_json+1,end_json-start_json-1);
        return result;
}

//////////////////////////////////////////////////////////////////////////////////////////////////////////
// Name: generate_body and generate_header
// Description: Generates body and header for the http request
// IO: Input sensor or offset + length, Outputs string containing header/body.

std::string http_request::generate_body(sensor* sensor_main)
{
        char BODY[350];
        int body_size = 0; // Contains the body size nessecary for the header
        uint8_t* mac = get_mac_address();

        body_size=snprintf(BODY, 350, "{ \"tag\": \"%X:%X:%X:%X:%X:%X\",",
```

```
                                        mac[0], mac[1], mac[2], mac[3], mac[4], mac[5]);
        body_size=snprintf(BODY, 350-body_size, "%s \"T\": %d,",BODY, sensor_main->read_sensor(5));
        body_size=snprintf(BODY, 350-body_size, "%s \"H\": %d,",BODY, sensor_main->read_sensor(6));
        body_size=snprintf(BODY, 350-body_size, "%s \"S1\": %d,",BODY, sensor_main->read_sensor(1));
        body_size=snprintf(BODY, 350-body_size, "%s \"S2\": %d,",BODY, sensor_main->read_sensor(2));
        body_size=snprintf(BODY, 350-body_size, "%s \"S3\": %d,",BODY, sensor_main->read_sensor(3));
        body_size=snprintf(BODY, 350-body_size, "%s \"S4\": %d,",BODY, sensor_main->read_sensor(4));
        body_size=snprintf(BODY, 350-body_size, "%s \"S5\": %d,",BODY, 0);
        body_size=snprintf(BODY, 350-body_size, "%s \"S6\": %d,",BODY, 0);
        body_size=snprintf(BODY, 350-body_size, "%s \"UP\": %lld,",BODY, esp_timer_get_time()/1000000);
        body_size=snprintf(BODY, 350-body_size, "%s \"A\": %d,",BODY, sensor_main->read_sensor(7));
        body_size=snprintf(BODY, 350-body_size, "%s \"B\": %d } \r\n",BODY, sensor_main->read_sensor(8));

        return BODY;
}
std::string http_request::generate_body(int offset, int length)
{
        char BODY[350];
        int body_size = 0; // Contains the body size nessecary for the header

        body_size=snprintf(BODY, 350, "{\"offset\": %d, \"length\": %d}", offset, length);

        return BODY;
}

std::string http_request::generate_header(int content_length)
{
        char HEADER[350];
        int header_size =0;

        if(content_length != 0){
                header_size=snprintf(HEADER, 350, "POST %s HTTP/1.0\r\n", web_path.c_str());
                header_size=snprintf(HEADER, 350-header_size, "%sHost: %s \r\n", HEADER, web_url.c_str());
                header_size=snprintf(HEADER, 350-header_size, "%sUser-Agent: ESP %d \r\n", HEADER, VERSION);
                header_size=snprintf(HEADER, 350-header_size, "%sContent-Type: application/json\r\n", HEADER);
                header_size=snprintf(HEADER, 350-header_size, "%sContent-Length: %d \r\n\r\n",

        }
        else{
                header_size=snprintf(HEADER, 350, "GET %s HTTP/1.0\r\n", web_path.c_str());
                header_size=snprintf(HEADER, 350-header_size, "%sHost: %s \r\n", HEADER, web_url.c_str());
                header_size=snprintf(HEADER, 350-header_size, "%sUser-Agent: ESP %d \r\n\r\n", HEADER, VERSION);
        }
        return HEADER;
}

////////////////////////////////////////////////////////////////////////////////////////////////////////
// Name: post
// Description: General script that controls the posting of sensor data
// IO: Input sensor
void http_request::post(sensor* sensor_main)
{
        std::string received_message;
        std::string BODY = generate_body(sensor_main);
        std::string HEADER = generate_header(BODY.size());

        // Write to request buffer
        snprintf(REQUEST, 600, "%s%s", HEADER.c_str() , BODY.c_str());
        // Post and read http response
```

```
        if(secure_connection==false)
        {
                received_message = http_get();
        }
        else{
                received_message=https_task_start();
        }
        // Decode JSON by first remove header, dividing into substrings and converting to ints
        // Need to add checker for npos and boundary conditions

        // Remove header completely

        std::string received_json=extract_json(received_message);

        if(received_message.find("ID")!=string::npos){
                int capacity         =stoi(extract_json(received_message, "capacity"));
                int fs                         =stoi(extract_json(received_message, "fs"));
                int warmup                 =stoi(extract_json(received_message, "warmup"));
                int period                 =stoi(extract_json(received_message, "period"));
                int threshold         =stoi(extract_json(received_message, "threshold"));

                uint8_t* mac = get_mac_address();

                ESP_LOGI(TAG, "Data posted and received at %X:%X:%X:%X:%X:%X",
                                                        mac[0], mac[1], mac[2], mac[3], mac[4], mac[5]);

                sensor_main->buffer_sensor.setValues(threshold, capacity, fs, warmup, period);
        }
        else
        {
                ESP_LOGE(TAG, "JSON decode failed");
        }

}

////////////////////////////////////////////////////////////////////////////////////////
// Name: hexval and hex2ascii
// Description: Script to convert hex data to ascii data
// IO: Input hex string, output ascii string

unsigned char hexval(unsigned char c)
{
        if ('0' <= c && c <= '9')
                return c - '0';
        else if ('a' <= c && c <= 'f')
                return c - 'a' + 10;
        else if ('A' <= c && c <= 'F')
                return c - 'A' + 10;
        else abort();
}
void hex2ascii(const std::string& in, std::string& out)
{
        out.clear();
        out.reserve(in.length() / 2);
        for (std::string::const_iterator p = in.begin(); p != in.end(); p++)
        {
           unsigned char c = hexval(*p);
           p++;
           if (p == in.end()) break; // incomplete last digit - should report error
```

```
            c = (c << 4) + hexval(*p); // + takes precedence over <<
            out.push_back(c);
        }
}

//////////////////////////////////////////////////////////////////////////////////////////////////////
// Name: get_ota
// Description: Function that retrieves part of the ota program
// IO: Input offset and length, output string which contains ota data

std::string http_request::get_ota( int offset, int length)
{
        std::string received_message;
        char BODY[350];

        // Generate body of POST request
        int body_size = 0; // Contains the body size nessecary for the header

        body_size=snprintf(BODY, 350, "{\"offset\": %d, \"length\": %d}", offset, length);
        //body_size=snprintf(BODY, 350-body_size, "%s \n next line of the body",BODY);

        // Write to request buffer
        snprintf(REQUEST, 600, "%s%s", generate_header(body_size).c_str(), BODY);

        //////////////////////////////////////////////////////////////////////////////////
        // Post and read http response
        std::string received_data;
        std::string received_encoded;
        received_message = http_get();

        // Remove header completely
        received_encoded=extract_json(received_message);

        // Convert hex to bin string
        hex2ascii(received_encoded, received_data);

 return received_data;
}

//////////////////////////////////////////////////////////////////////////////////////////////////////
// Name: check_ota
// Description: Function that checks the remote version against current function
// IO: Input version and pointer to remote version, output checksum of remote version

std::string http_request::check_ota(int version, int* remote_version)
{


        // Generate GET REQUEST and write to buffer
        snprintf(REQUEST, 250, "%s", generate_header(0).c_str());

        //////////////////////////////////////////////////////////////////////////////////
        // Post and read http response
        std::string received_json;
        std::string checksum;
        int new_version;

        // Remove header completely
        received_json=extract_json(http_get());
```

```
        new_version                =stoi(extract_json(received_json, "version"));
        checksum=                extract_json(received_json, "checksum");
        *remote_version=new_version;

 return checksum;
}

////////////////////////////////////////////////////////////////////////////////////////////////////////
// Name: set_config and get_config
// Description: Sets and retrieves certain parameters of the class

void http_request::set_config(std::string url, std::string port, std::string path)
{
        web_url = url;
        web_port = port;
        web_path = path;
}

struct struct_http_config http_request::get_config(void)
{
        struct_http_config http_config;
        return http_config;
}

////////////////////////////////////////////////////////////////////////////////////////////////////////
// Name: get_mac_address
// Description: Function that retrieves part of the ota program
// IO: Input offset and length, output string which contains ota data

uint8_t* http_request::get_mac_address(void)
{
        // Only need to read the mac adress from the fuses once
        if(mac_address[0] == 0 && mac_address[1] == 0 && mac_address[2] == 0)
        {
                esp_read_mac(mac_address, ESP_MAC_WIFI_STA); //obtain mac of station
        }

        return &mac_address[0];

}

////////////////////////////////////////////////////////////////////////////////////////////////////////
// Name: https_settings
// Description: Sets the https certifactes
// IO: Output config file

esp_tls_cfg_t http_request::https_settings(void)
{


        esp_err_t esp_ret = ESP_FAIL;
                ESP_LOGI(TAG, "https_request using global ca_store");
                esp_ret = esp_tls_set_global_ca_store(server_root_cert_pem_start, server_root_cert_pem_end -

                if (esp_ret != ESP_OK) {
                        ESP_LOGE(TAG, "Error in setting the global ca store: [%02X] (%s),
                                                                could not complete the https_request using global_ca_store",
                                                                esp_ret, esp_err_to_name(esp_ret));
                        abort();
```

```
		}
		esp_tls_cfg_t cfg = {
			.use_global_ca_store = true,
		};

		settings=cfg;
		return cfg;
}

//////////////////////////////////////////////////////////////////////////////////////////////////////
// Name: wifi_ota
// Description: checks and retrieves new wifi
// IO: Output config file
void http_request::wifi_ota(std::string checksum, struct_wifi_para *wifi_settings, bool *current){
	std::string received_message;
	char BODY[350];
	uint8_t* mac = get_mac_address();

	// Generate body of POST request
	int body_size = 0; // Contains the body size nessecary for the header
	body_size=snprintf(BODY, 350, "{\"tag\": \"%X:%X:%X:%X:%X:%X\", \"WIFI_checksum\": \"%s\", "
			"\"key\":\"74621bb8-7e7c-4ba4-90a0-1c3a78935dd9\"}",
			mac[0], mac[1], mac[2], mac[3], mac[4], mac[5], checksum.c_str());

	// Write to request buffer
	snprintf(REQUEST, 600, "%s%s", generate_header(body_size).c_str(), BODY);

	// Post and read http response
	std::string received_data;
	received_message = http_get();
	// Remove header completely
	received_data=extract_json(received_message);
	std::cout<<received_message;
	if(received_message.find("null")!=string::npos){ // device not found in db

	}
	else if(received_data.find("WiFi OK")!=string::npos){ // wifi up to date
		*current = true;
		ESP_LOGI("WiFi OTA", "WiFi up-to-date");
	}
	else if(received_data.find("user")==string::npos){ // Not enterprise
		*current = false;
		ESP_LOGI("WiFi OTA", "New WiFi PSK found");
		wifi_settings->enterprise=false;
		wifi_settings->SSID=extract_json(received_message, "ssid");
		wifi_settings->PASS=extract_json(received_message, "pass");
		std::cout<<wifi_settings->SSID;
		std::cout<<wifi_settings->PASS;
	}
	else{
		ESP_LOGI("WiFi OTA", "New WiFi ENT found");
		// enterprise
	}
}

////////////////////////////////////////////////////////////////////////////////
// Constructor function
http_request::http_request(void)
{
```

```
        web_url = "poop.markfijneman.nl";
        web_port = "80";

#if TESTMODE==true
        web_path = "/tesp.php";
#else
        web_path = "/esp.php";
#endif
        web_full_url = "https://";
        web_full_url.append(web_url);
        web_full_url.append(web_path);

        secure_connection=false;
}
```

measurements.php

```
<?php

include "connection.php";
include "discordhook.php";
include "smshook";
/*
 * Obtain HTTP request
 */
$_POST = json_decode(file_get_contents('php://input'), true);


/*
 * creating new measurement
 */
if ($_SERVER['REQUEST_METHOD'] === 'POST') {

        if(isset($_POST['tag'])
        && isset($_POST['T'])
        && isset($_POST['H'])
        && isset($_POST['S1'])
        && isset($_POST['S2'])
        && isset($_POST['S3'])
                && isset($_POST['S4'])
                && isset($_POST['S5'])
                && isset($_POST['S6'])
        && isset($_POST['UP'])
                && isset($_POST['A'])
        && isset($_POST['B']))
    {
                // Obtain device ID
                $device=saveMysql($_POST['tag']);
                $sql = "SELECT tag, device_id FROM devices WHERE tag=\"$device\"";
                $result=$conn->query($sql)->fetch_assoc();
                $device_id=$result['device_id'];

                if(empty($device_id))       // No device ID found
                {
                        die(json_encode(['message' => 'Device not registered']));
                }

                // Post sensor data in DB
```

```
        $sql = "INSERT INTO measurements (device_id, temp, rh, sensor_1, sensor_2, sensor_3, sensor_4,
                                                                                         sensor_5, sensor_6, time_up, alg
        ('".saveMysql($device_id)."',
        '".saveMysql($_POST['T'])."',
        '".saveMysql($_POST['H'])."',
        '".saveMysql($_POST['S1'])."',
        '".saveMysql($_POST['S2'])."',
        '".saveMysql($_POST['S3'])."',
                '".saveMysql($_POST['S4'])."',
                '".saveMysql($_POST['S5'])."',
                '".saveMysql($_POST['S6'])."',
        '".saveMysql($_POST['UP'])."',
                '".saveMysql($_POST['A'])."',
        '".saveMysql($_POST['B'])."', '".date("Y-m-d H:i:s")."')";
        $conn->query($sql);

                // Update poop detected param of device
                $sql = "SELECT algo, tag FROM devices WHERE tag=\"$device\"";
        $result=$conn->query($sql)->fetch_assoc();
                if(($_POST['A']=='0') && $result["algo"]=='1' ) // Poop is not detected anymore
                {
                        // Set poop detected to negative
                        $sql = "UPDATE devices SET algo='0' WHERE tag=\"$device\"";
                        $conn->query($sql);
                }
                if($_POST['A']=='1'&& $result["algo"]=='0')
                {
                        // Poop detected and need to activate webhook
                        $sql = "UPDATE devices SET algo='1' WHERE tag=\"$device\"";
                        $conn->query($sql);

                        // Retrieve descriptions of user
                        $sql = "SELECT devices.tag, devices.location, devices.description as place, location.phone,
                                        location.description as user FROM devices INNER JOIN location
                                        ON devices.location=location.location WHERE devices.tag=\"$device\"";
                        $result=$conn->query($sql)->fetch_assoc();

                        // Start sms and discordhook
                        sendWebHook($result["user"], $result["place"]);
                        smsWebHook($result["user"], $result["place"], $result["phone"]);
                }


                // returing algo para
                $sql = "SELECT * FROM algo_settings ORDER BY ID desc limit 1";
                $result = $conn->query($sql);

        die(json_encode($result->fetch_assoc()));
    }
    else
        {
        die(json_encode(['message' => 'Not all required values are filled.']));
    }

}

/*
 * no valid endpoint
 */
```

```
die(json_encode(['message' => 'Endpoint not valid.']));


?>
```

```
<?php
// Init default timezone
date_default_timezone_set('Europe/Amsterdam');

/*
 * connect with db
 */

$host = "localhost";
$user = "USERNAME";
$pass = "PASSWORD";
$db = "DATABASE NAME";

$conn = mysqli_connect($host, $user, $pass, $db);


/*
 * Function that prevents Cross-Site Scripting Attacks
 */
function saveMysql($input)
{
    return str_replace(array("\"", "'", "\\", "<", ">"), "", htmlentities($input, ENT_QUOTES));
}

?>
```

```
<?php
/*
 * Discord webhook
 */
function sendWebHook($location, $device){
$curl = curl_init();
$user = "test";
curl_setopt_array($curl, array(
  CURLOPT_URL => 'discord hook URL',
  CURLOPT_RETURNTRANSFER => true,
  CURLOPT_ENCODING => '',
  CURLOPT_MAXREDIRS => 10,
  CURLOPT_TIMEOUT => 0,
  CURLOPT_FOLLOWLOCATION => true,
  CURLOPT_HTTP_VERSION => CURL_HTTP_VERSION_1_1,
  CURLOPT_CUSTOMREQUEST => 'POST',
  CURLOPT_POSTFIELDS =>'{
    "content": "Poop detected at user '.$location .' on device '.$device .'"
}',
  CURLOPT_HTTPHEADER => array(
    'Content-Type: application/json',
    'Cookie: __dcfduid=4e6464ce248649518d79002a510ed3a8'
  ),
));

$response = curl_exec($curl);
```

```
curl_close($curl);
}

?>
```

```
<?php
/*
 * SMS webhook
 */
function smsWebHook($location, $device, $cellphone){
$curl = curl_init();

curl_setopt_array($curl, array(
  CURLOPT_URL => 'sms hook url server',
  CURLOPT_RETURNTRANSFER => true,
  CURLOPT_ENCODING => '',
  CURLOPT_MAXREDIRS => 10,
  CURLOPT_TIMEOUT => 0,
  CURLOPT_FOLLOWLOCATION => true,
  CURLOPT_HTTP_VERSION => CURL_HTTP_VERSION_1_1,
  CURLOPT_CUSTOMREQUEST => 'POST',
  CURLOPT_POSTFIELDS => 'msg=Poop detected at user '.$location .' on device '.$device .'&no=%2B' . $cellphone,
  CURLOPT_HTTPHEADER => array(
    'Content-Type: application/x-www-form-urlencoded'
  ),
));

$response = curl_exec($curl);

curl_close($curl);

}
?>
```

C

Over the Air Programming

webserver

firmware.php

```php
<?php

include "connection.php";
include "discordhook.php"
include "smshook.php"
/*
 * Obtain HTTP request
 */
$_POST = json_decode(file_get_contents('php://input'), true);


/*
 * Requesting new piece of OTA firmware data
 */
if ($_SERVER['REQUEST_METHOD'] === 'POST') {
    if(isset($_POST['offset']) && isset($_POST['length'])){
                if (file_exists($file)) {
                        $totalsize = filesize($file);

                        $handle = fopen($file, 'r');
                        //$data = base64_encode (fread($handle, filesize($file)));
                        $data = bin2hex (fread($handle, filesize($file)));
                        fclose ( $handle );


                    header('Content-Description: File Transfer');
                    header('Content-Type: application/octet-stream');
                  //  header('Content-Disposition: attachment; filename="'.basename($file).'"');

                        echo '{'. substr($data, $_POST['offset'], $_POST['length']) . '}';

                        exit;
                }
                else{
                        die(json_encode(['message' => 'Update is not found.']));
                }

        }
    else {
        die(json_encode(['message' => 'Not all required values are filled.']));
    }
}


/*
 * Obtain version numbers
 */
```

```
if ($_SERVER['REQUEST_METHOD'] === 'GET') {
        $arr = array('version' => '8',
                        'checksum' => 'a67a6a30a6bc5e8f86e797d7a095d96588c811de082462de1acb5d9d6bdb8043',
                        ' created_at' => 3);
    echo json_encode($arr);
        exit;
}

/*
 * no valid endpoint
 */
die(json_encode(['message' => 'Endpoint not valid.']));





?>
```

parameters.php

```
<?php
include "connection.php";

/*
 * Process OTA updates
 */
if ($_SERVER['REQUEST_METHOD'] === 'GET') {
        // return algo para
        $sql = "SELECT * FROM algo_settings ORDER BY ID desc limit 1";
        $result = $conn->query($sql);

        die(json_encode($result->fetch_assoc()));
}


/*
 * no valid endpoint
 */
die(json_encode(['message' => 'Endpoint not valid.']));


?>
```

wifi.php

```
<?php

include "connection.php";

/*
 * Obtain HTTP request
 */
$_POST = json_decode(file_get_contents('php://input'), true);

/*
 * Check and obtain new wifi parameters
```

```
 */
if ($_SERVER['REQUEST_METHOD'] === 'POST') {

    if(isset($_POST['tag'])
        && isset($_POST['WIFI_checksum'])
        && ($_POST['key']=='74621bb8-7e7c-4ba4-90a0-1c3a78935dd9'))
    {
                // Obtain version of esp from user agent field and update device table
                $local_version = preg_replace('/[^0-9]/', '', $_SERVER['HTTP_USER_AGENT']);
                $device=saveMysql($_POST['tag']);
                $sql = "UPDATE devices SET version=$local_version WHERE tag=\"$device\"";
        $conn->query($sql);

                // Obtain location of the device and check if wificheckum is valid
                $sql = "SELECT devices.tag, location.wifi_checksum, location.location,
                                location.wifi_type FROM devices INNER JOIN location
                                ON devices.location=location.location WHERE devices.tag=\"$device\"";
        $result=$conn->query($sql)->fetch_assoc();
                $location=$result["location"];
                if($_POST['WIFI_checksum']==$result["wifi_checksum"]){
                        die(json_encode(['message' => 'WiFi OK']));
                }
                else { // WiFi not up to date
                        // Need to check wheter it is psk or enterprise
                        $sql = "SELECT location, ssid, pass, wifi_checksum FROM location WHERE location=\"$location\"";
                        $result=$conn->query($sql)->fetch_assoc();
                        die(json_encode($result));
                }


    }
    else {
        die(json_encode(['message' => 'Not all required values are filled.']));
    }

}


/*
 * no valid endpoint
 */
die(json_encode(['message' => 'Endpoint not valid.']));


?>
```