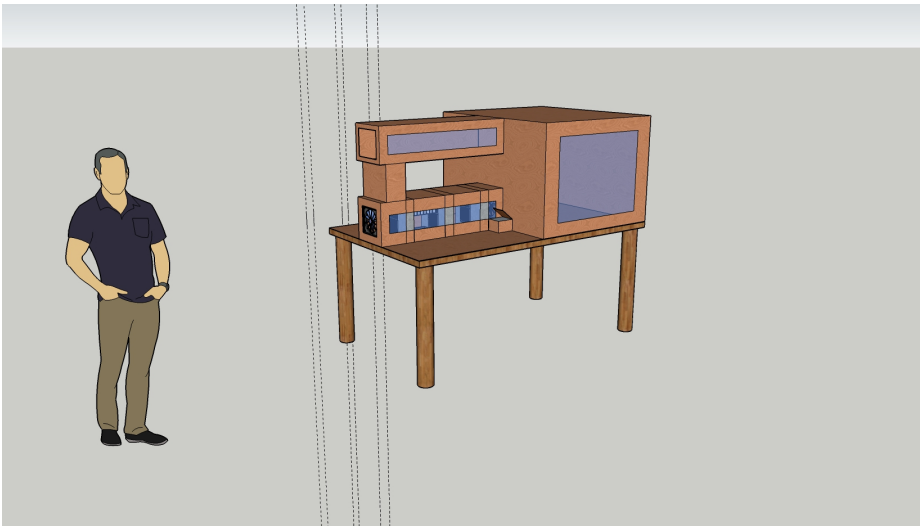
Air Handling Unit

Saliem Amirkhan – 14045338 Thijs Bril – 14044951
Richard Kokx – 14139227 Tobi van Westerop – 13019236

Final report – 21 June 2017
Second revised version – November 5, 2017

The Hague University of Applied Sciences
Technische Informatica
TI-H-PR – Bedrijfsproject



Summary

Air handling units are complex machines. To help students understand them better, they need hands-on experience with a unit. However, real units are too large and too expensive for most schools, and simulations are not accurate enough. Therefore, we were tasked with designing, creating and programming a miniature prototype of an air handling unit. This unit had several requirements concerning temperature, humidity, and CO_2 sensors, and actuators for air speed, heating, cooling, and humidity.

We built a wooden case, we decided on a Raspberry Pi (Raspberry Pi Foundation, 2017b) (hereafter often referred to as simply "Raspberry" or "Pi") as the controller, and wrote an API in C++to communicate with the sensors and actuators. We are controlling the heating and cooling elements, the fans, and the servos for the vents using PWM. The temperature and humidity sensors are controlled using I2C, and communication with the CO_2 sensors occurs using UART.

To operate the air handling unit, and to create some example code, we built a control application with a web interface. This application can manually control the actuators and reads out all of the sensors. In the end, most of the requirements were successfully fulfilled.

Introduction

Air handling units are complex machines with a lot of variables, all of which can have an influence on the way the system works. This makes it difficult for students to fully comprehend systems like these so that they are able to design and implement air handling units themselves. The problem with normal sized air handling units is that they are just too big. How can students experiment with it to get an understanding of how an air handling unit works? Currently, the only possible way is with simulators. But as with every simulator, there is the question if they are realistic enough. Are they applicable for students who want to learn how an air handling unit works? The answer is no. While they perform the basic functions, there are ways to make a room even hotter than the sun. This is of course not physically possible. That's why we were given the task to create a scaled down model of a normal sized air handling unit. Designing, building and coding the system to be working at the demonstration all need to be done within this assignment.

Designing the build was the first thing addressed during the project. During this stage research had to be done on what components would be used, which type of material to use and all the measurements had to be calculated. The decision of what parts or components to use was also based on the given requirements. The research on the components can be found in the "Hardware" sections of this report, and the requirements are available in Chapter 2.

Building the unit itself was done after research was finished. This task contained getting the resources, cutting them to the calculated measurements and assembling everything. More in-depth information on this subject can be found in Section 3.2 – Case.

At last the code had to be written for all the components. In the beginning all the components were tested separately of each other. This meant coding went per component. After coding all the components, everything was put together to make an complete API which we decided to complement with a control software to demonstrate the API in action. Information about this subject can be found in Chapter 4 - Software.

Contents

Chapter 1

Approach

In the beginning of the project, we decided on using Scrum (Scrum Alliance, 2017). Scrum is a project methodology that would allow us to quickly get a working prototype, on which we could then apply improvements in fast iterations. It is also a methodology we have used before, which means we could get started quickly without having to learn new procedures.

Our plan was to take the first three weeks as "Sprint 0", to formulate a plan of approach and get all requirements and plans figured out. This plan of approach can be found in Appendix D, a short outline or our planning is given here.

In the next sprint (sprint 1), we were going to start our research, and we were going to design our prototype. We were also going to start building the prototype. In sprint 2, the plan was to have finished our research. We also wanted to our prototype hardware complete. In this second sprint, we were also planning on starting on software development for the prototype. The plan then was to have a working prototype in sprint 3, at the assessment of the first half of the project.

In sprints 4 through 7 (weeks 4.1 - 4.8), our plan was to elaborate the control program in two-week iterations. In the meantime, we were going to evaluate the hardware and improve it if necessary.

Chapter 2

Requirements

In order to design the air handling unit and to make sure the choices made are in line with the desires of the client, clear requirements need to be defined. These requirements can be split into two types: functional requirements, defining the functions that the air handling unit must or should be able to carry out, and non-functional requirements, defining certain preconditions that the design should be compliant with.

2.1 Functional requirements

In essence, the function of an air handling unit is fairly simple, which is why there are not a lot of functional requirements. The air handling unit must be able to measure CO_2 levels, humidity levels, and temperature. It must also be able to control those levels, using a variety of actuators. It should also be possible for users of the unit to readout the measured levels, and determine how the air handling unit responds to those measurements in order to control the actuators.

In the first meeting with the client, he indicated he wanted a chamber size of approximately $50 \times 50 \times 50\,\text{cm}$. He also expressed the wish that the specifications of the mock-up unit should be similar to those of real air handling units. The desired chamber size leads to a volume of 125 L. During our excursion to Systemair [1], we discovered that an air handling unit is supposed to refresh all of the air in the building three times per hour. Plugging in those numbers leads to an airflow of 375 L/h, or $104.2\,\text{cm}^3/\text{s}$. To simulate an air handling unit in Dutch weather, we looked at data from the KNMI (Royal Dutch Weather Institute), which showed the lowest average minimum temperature in February is $-1.0\,^\circ\text{C}$ (KNMI, 2011b), and the highest average maximum temperature in August is $24.0\,^\circ\text{C}$ (KNMI, 2011a). To take some margins into account, we assumed an intake air temperature between $-10\,^\circ\text{C}$ and $30\,^\circ\text{C}$ four our calculations. We also assumed a desired room temperature between $18\,^\circ\text{C}$ and $25\,^\circ\text{C}$. The relative humidity should ideally be between 50% and 60%, but anywhere between 30% and 70% is acceptable. The relative humidity of the intake air (assuming the Dutch climate) can be anywhere between 20% and 100% (KNMI, 2011c). The

[1] Systemair Group is a worldwide manufacturer of heating, ventilation and airconditioning products. We visited their production facilities in Waalwijk.

temperature and relative humidity are within these ranges in many parts of the world, so designing the air handling unit according to these criteria should mean it can be used by students worldwide.

2.2 Non-functional requirements

Because of the specific purpose of this mock-up air handling unit, there are many more non-functional requirements, which are in some cases more important than the functional requirements in making certain decisions. First, the air handling unit must be built out of materials that are easily available to any technical educational institution anywhere in the world, so that anyone needing to instruct students on the workings of an air handling unit can build a mock-up similar to ours. This also means the cost of the materials needs to as low as possible. Building the air handling unit should also be very simple.

Chapter 3

Components

3.1 Controller

3.1.1 Hardware

For the controller, we considered five options: a Raspberry Pi (Raspberry Pi Foundation, 2017b), a single-board PC from a different manufacturer (Sinovoip, 2017), an Arduino (Arduino, 2017), a microcontroller chip with our own circuits around them (Atmel Corporation, 2015), or a fully fletched desktop. Each of these options had their own advantages and disadvantages. A Raspberry Pi has the advantages of being widely available and easily programmable. It also easily supports a screen, and has built-in support for the protocols of the available sensors. It also has WiFi and Bluetooth on board. The disadvantages are that the input/output ports function at 3.3 V, and have a very limited power delivery.

The single-board PC from a different manufacturer is very similar to the Raspberry Pi, except that it can deliver more power. However, it is twice as expensive as the Raspberry Pi, and it is less well known, which means that it isn't as widely available and has less of a support community.

The Arduino has the advantages of being a highly adaptable, low-power board, which functions completely at 5 V logic levels. It also has a lot of input/output pins, and is widely available worldwide. The disadvantages are that it requires a computer to program it, and that loading programs onto the Arduino can be a little complicated and prone to errors. It is also not possible to connect anything but a very simple LED display to an Arduino.

A microcontroller in a home-made circuit is in essence very similar to an Arduino, except that it is a lot cheaper. This price difference comes with a tradeoff, namely that it requires a lot of time and effort to develop the circuit. The home-made circuit also makes the system harder to reproduce than when using any other controller.

The last option we looked into was a fully fletched desktop computer. The price of such a system can vary a lot, but it is more expensive than any of the other choices. The advantages are that a desktop computer is powerful, multifunctional and easily programmable. The disadvantages are that a desktop computer does not have any real input/output ports, which makes it a lot harder to interact with sensors and actuators, and that it is too expensive.

After weighing all the advantages and disadvantages, we decided that us-

ing a Raspberry will be the best option for this project, because of the easy reproducibility, the relatively low costs, and the easily accessible input/output ports. As a added bonus we can display a monitoring UI on the supported hdmi interface which shows a real time representation of the unit

The controller unit is installed to the right of the intake tube of the air handling unit. Next to the two breadboards which is where all the cables from the sensors and actuator are combined and eventually are connected to the Raspberry Pi. The setup for the demo is using a consumer wireless solution to create a local network for the Raspberry Pi and developers PCs. The router is setup with a static DHCP lease for the Raspberry Pi MAC address for easy connection even when we switch to a different setup. Power is delivered through a standard 2 Amp Micro-USB wall adapter. This could however be changed too feed directly from the power supply but for the demo setup is was preferred to separate the two so that in the case of an overload or other incident we could turn off the power supply without shutting down the Raspberry Pi.

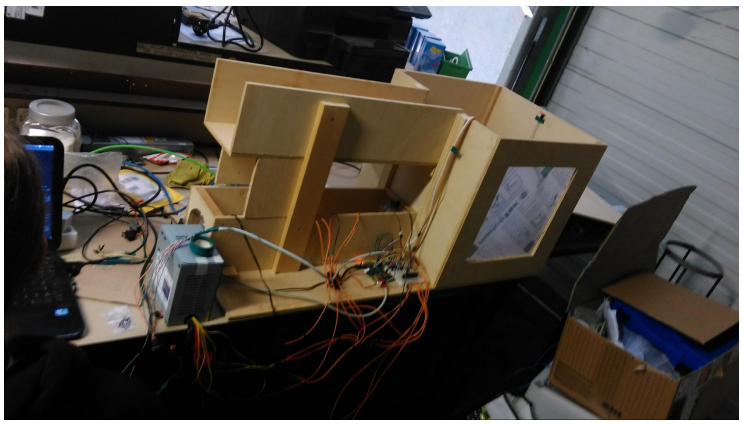


Figure 1: A picture of the case in-progress, with the Raspberry Pi installed.

3.1.2 Software

The Raspberry Pi is running Raspbian Jessie Lite[1] which due to being a Linux distribution is easy to administer over an SSH connection from one of the development PCs. Code is imported to the Pi via Git[2]. If one of the developers pushes a new version it is pulled to the Pi and compiled through a custom makefile which first compiles the API (if necessary) to object files. Next, it compiles the main controller program, and finally it links all object files together into a single executable.

[1]Raspbian (Raspberry Pi Foundation, 2017a) is the official supported operating system for the Raspberry Pi, based on the Debian Linux distribution (Software in the Public Interest, Inc., 2017). Jessie is the previous version of Raspbian, the most recent version ("Stretch") was released in the summer of 2017. The "Lite" version does not include a desktop environment.

[2]Git is a distributed version control system, originally created for the development of the Linux kernel (Software Freedom Conservancy, Inc., 2017)

	Metal	**Wood**	**Plastic**
Advantages	Very sturdy	Not affected by temperature	Lightweight
	Strong while thin	Easy to shape	Cheap
	Medium availability	Cheap	Relatively easy to shape
		Easy to mount together	
		Easy to replace a small piece	
		High availability	
Disadvantages	Affected by temperature	Affected by water	Hard to mount together
	Hard to shape	Not as sturdy as metal	Affected by temperature
	Chance of electrical short-circuits		Low availability
	As TI hard to weld together		

Table 1: Comparison of materials

3.1.3 Problems

While setting up the Raspberry Pi we encountered a couple problems. During our first setup of Raspbian we used a cheap poor quality SD card which resulted in agonisingly slow install and update times. This was however resolved when we received the 16GB Class 10 SD card. Class 10 means the card has a minimal sequential writing speed of 10MB/s, as well as having a higher reading speed than the cheap SD card we used earlier. The other problem we ran into was with the ip address since we did not install a monitor and we did not have any way to check what ip address got assigned by the router. We fixed this by using our own router with a static DHCP binding.

3.1.4 Recommendations

We would recommend using a high quality SD card class 10 or similar. We also recommend implementing a way to power the Raspberry Pi from the power supply, while keeping it separate from the rest of the components, to be able to shut down the components without powering off the Raspberry Pi.

3.2 Case

3.2.1 Hardware

For the building materials the choice came down to three choices: metal, wood and plastic.

With the considerations, given in Table 1, we went for wood. Wood is the easiest to adjust and has a high availability, so later on it will be easy for other students to replicate it, as stated in the non-functional requirements. Of course it's possible to create the mock-up with metal or plastic, but for our timespan, skills and the assignment requirements wood was the best choice.

While buying the wood plate and while holding it we concluded that the 4 mm plate would not be enough for the case. So we decided to go with a double thickness and bought a 9 mm thick plywood plate. We first started cutting the one 244×122 cm hardwood plywood plate in sections as outlined in our design. After that we cut out a piece of 35×35 cm perspex for the window.

Then we started to assembly of the case this took considerable amount of time. During this time we decided to go with modular setup so that we could easily change the wiring and install additional compartment walls. After completing the main chamber and fixing some off the measuring flaws, we needed to cut of some edges which were forgotten to be taken into account, we started on the first tube. This intake tube will be housing for all of the actuators and most of the sensors. Due to this fact we made the top of this part slidable so we could reach in and place/replace components. The same is true for tunnel connecting it to the upper tube. Too support this tube and since there is no complete interconnect we installed some support beams to support the upper tube. We then finished the case by installing the upper tube.

3.2.2 Problems

We ran into a amount of problems the first being was easily fixed but worth noting. While buying the stock wood plate we had the plate cut at the store but due to a mistake from the store clerk we ended up with wrong size pieces. After some complaining with the store we got replacement parts so we ended up with extra pieces. This ended up helping later with our next problem while we were building the case we ended up missing a couple mm on either side of the tube walls this was due to the original design being meant for 4 mm thick wood but we went with 9 mm because we found that 4 mm would not be structurally sound enough. Another problem we ran into while constructing the case was because a 9 mm sheet is still rather thin the nails would split the plywood to fix this we used small wood strips this also helped with the modular approach we went with because now we could support the top piece of the tubes on the strips without nailing them too the rest of the case.

3.2.3 Recommendations

We would recommend that future prototypes use even thicker material or use better support strips this would also allow for almost complete use of screws instead of nails this would make taking the case apart for changes easier as well as making the case more rigid and stronger overall. If you would use materials that are more than 2 cm thick you could even forgo the use of support material what would give the case a more aesthetically pleasing look and more place for cables and components. Also to complete the case when all the components are installed we would recommend using silicon sealant to make the case airtight and to make the whole case airtight we recommend using a rubber ring and a pressure locking mechanisms to seal the top of the main chamber. Another recommendation would be to implement an efficient cable management route instead of bent over nails.

3.3 Heating elements

3.3.1 Hardware

To calculate the power requirements of the heating elements, we will assume the worst-case scenario, whereby the intake temperature is $-10\,^{\circ}\mathrm{C}$, and the desired temperature of the air in the chamber is $25\,^{\circ}\mathrm{C}$. Using a Mollier diagram (see

Figure 2), we can figure out how much energy this needs. The vertical axis is the temperature of the air in °C, the horizontal axis is the absolute humidity in kg water per kg air. The curved lines are lines of equal relative humidity, while the diagonal lines are the amount of energy in a kg of air, relative to the chosen origin of 0 °C without any humidity.

Heating up air means moving straight up in the diagram, because the temperature changes without affecting the absolute humidity. This means if we heat the air up from −10 °C to 25 °C at zero humidity, we need to go from −10 J/kg to 25 J/kg, adding 35 J/kg. If we start at 100% humidity, we start at −6 J/kg and go to 29 J/kg, so we are also adding 35 J/kg. In Chapter 2, we calculated that the air needs to be flowing at 104.2 cm^3/s. The density of air is 1.225 kg/m^3, so we will need to heat up about 1.28×10^{-4} kg/s, which (plugging in the 35 J/kg) requires approximately 0.004 47 J/s, or about 4.47 mW. However, if we use the airflow as described in Section 3.7 of 140.3 m^3/h, we get 38.972 cm^3/s, or 0.0477 kg/s, which means we need about 1.67 W. However, this power requirement does not contain any power needed to compensate for the cooling element. As visible in Section 3.4, the cooling element will cool the air no more than 25 °C, which means the calculated 1.67 W is at least half of the total power required. This means that (factoring in some margins) a total heating power of at least 5 W should be enough for our purposes. Because our design uses two heating elements, the required power rating of each heating element is 2.5 W.

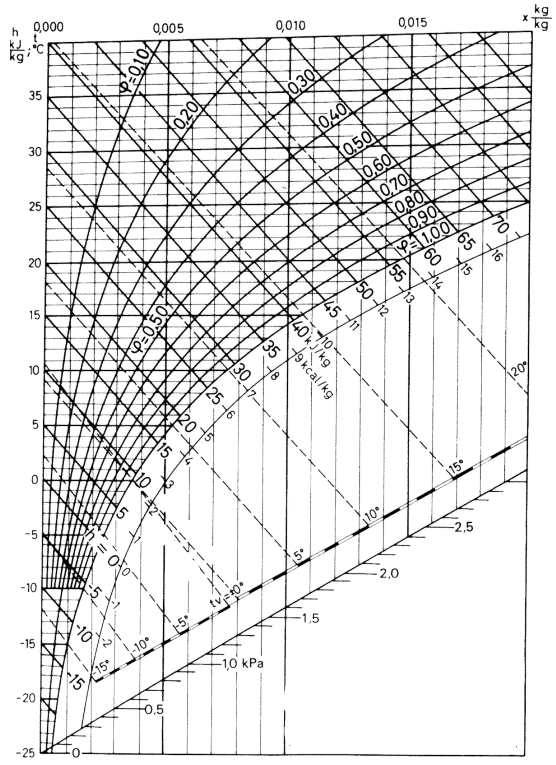


Figure 2: Mollier diagram. (The Engineering Toolbox, 2017)

For the heating elements, we had a choice out of several possibilities. In regular-sized air handling units, air is heated using hot water, which usually comes from a gas-burning heater. However, due to the complexity of a design using hot water, we had been given the task of finding a purely electrical heater. Electrical heating elements are practically all based on a resistance wire, which heats up if a current flows through it. The differences between the elements come from the length and width of the wire (which determine voltage and power requirements), and from the type of the element the wire is embedded in. For the type of the element, we found three widely available options: a tube-shaped element with slats, a ceramic element, or a resistance wire without an element around it. Each of these options has its own advantages and disadvantages.

The tube-shaped element as visible in Figure 3 is specially designed to heat air, and it's a standard element that's easy to connect. The disadvantages are that tube-shaped elements are all rated at 230V, which means we will need a powerful relais switch. There also isn't a wide range of power available. A last and critical disadvantage is that the shortest tube-shaped elements have a minimum length of 200 mm, while for our design the maximum length of the heating element is 150 mm. This means there is no way to fit a tube-shaped element into the air handling unit.

Figure 3: Tube-shaped heating elements. (Electricfor, S.A., 2017)

The ceramic elements have several advantages: they are available in a wide variety of sizes and shapes, have a large range of possible power ratings, and are available both in 230 V and 12 V versions. The main disadvantage is that the only ceramic element available at Conrad, pictured in Figure 4, is smaller than we would like it to be. This element is $90 \times 27 \times 17$ mm, which means we will have to figure out a way to prevent the air from flowing around it instead of through the element.

Lastly, there is the option to purchase a spool of resistance wire, as pictured

Figure 4: Ceramic heating element. (Conrad Electronic International GmbH & Co. KG, 2017a)

in Figure 5. The advantage is that we can choose the length of the resistance wire, which means any combination of power, voltage and amperage is possible. However, this advantage does not weigh up against the disadvantages: if we use a resistance wire, we have to build a system to transfer the energy efficiently to the air, while at the same time preventing short circuits or electric shocks. There is also a chance that the resistance wire gets very hot in certain places, which has a non-negligible risk of fire.

Figure 5: Resistance wire. (Conrad Electronic International GmbH & Co. KG, 2017d)

In conclusion, we decided that a resistance wire would be too difficult to implement safely, because of the risks of electric shocks or fire. It also would not fit with the idea of an air handling unit mock-up that is easily built by others. The choice was now between the tube-shaped element or the ceramic element. Because of the tube-shaped elements are only available starting at 200 mm, we decided on using a ceramic element, because it fits better in our design. The only ceramic heating element that would not have to be shipped overseas (thus requiring too long of a delivery time) was the small 150 W element

mentioned earlier. This element can deliver more power than we need, but this mainly means it will not have to be switched on continuously.

This element when it was received tested and we observed that it heats up incredibly fast. However we were using simple underpowered transistors, which heated up and failed very fast. During this time we decided to wire up the heating elements and place them in the case and order mosfets that would be able to withstand the immense current required for the heating elements. However, these have not been delivered yet and could not be installed for this reason.

3.3.2 Software

To accurately control the heating elements, we are using a pulse-width modulation (PWM) signal to rapidly turn a transistor on and off. To do this, we created the PWM class, which we will explain first. This class implements the iCommunication interface, so it has the standard `Initialize()`, `Read()` and `Write()` functions (lines 6, 8, 9). The `Read()` function is empty on purpose, because it is not possible to read from a PWM pin. Due to the specifics of the softPwm API of the WiringPi library, we need a more sophisticated Initialize function which receives a list of pins that will be used for PWM signals. Therefore, the standard `Initialize()` function will always return false.

```
1  class PWM : public iCommunication
2  {
3  public:
4    PWM();
5    virtual ~PWM();
6    virtual bool Initialize();
7    virtual bool Initialize(uint8_t* pins, size_t size);
8    virtual bool Write(uint8_t data, uint8_t address);
9    virtual bool Read(uint8_t* buffer, uint8_t length);
        //empty
10 };
```

The `Initialize()` function receives an array of GPIO pins that will be used for PWM signals. All of these are initialized with the `softPwmCreate()` function. This function also requires an initial value and a maximum value. In our case, we always set the initial value to 0 to make sure the pin is turned off. The maximum value is always set to 100, which means we can set the pin to anywhere between 0 and 100.

```
1  bool PWM::Initialize(uint8_t* pins, size_t size){
2    for(uint8_t i = 0; i<size; i++){
3      if(softPwmCreate(pins[i], 0, 100)){
4        return false;
5      }
6    }
7    return true;
8  }
```

The `Write()` function only requires a data value and an address. All it does

is call the `softPwmWrite` function. Since this function has no error checking, we can not detect whether or not the call has succeeded.

```
1 bool PWM::Write(uint8_t data, uint8_t address){
2   softPwmWrite(address, data);
3   return true;
4 }
```

The heating element class, HeatingElement, is relatively simple, and inherits from the iActuator class. Its most important function is `SetValue()`. This function first limits the value to within an acceptable range, and then calls the `SetValue()` function of the parent class. The acceptable range for the heating elements is between 0 and 30, because that ensures the transistors do not get too hot.

```
1 void HeatingElement::SetValue(uint8_t value)
2 {
3   if (value < 0)
4     value = 0;
5   else if (value > 30)
6     value = 30;
7   iActuator::SetValue(value);
8 }
```

The `SetValue()` function of the iActuator class is very simple. All it does is call the `Write()` function of the Communication member.

```
1 void iActuator::SetValue(uint8_t value)
2 {
3   if (!Communication->Write(value, Address))
4     std::cerr << "ERROR: writing went wrong" << std::
          endl;
5 }
```

3.3.3 Problems

As alluded to in the hardware sections of the heating element the problem we ran into with the heating element was not the heating element but rather the hardware required to control the power of the element. We used TIP120 transistors but these are only rated for 5 A (Semiconductor Components Industries, LLC, 2014) Since our heating elements each pull more than 5 A (Conrad Electronic International GmbH & Co. KG, 2003) the transistor heated up and melted. After this we did not use the transistors to control the heating element anymore and tested the api software using a voltmeter.

3.3.4 Recommendations

We would recommend using a mosfet like FQP30N06L. These are rated for up to 32 A (Fairchild Semiconductor Corporation, 2013), this will be more than enough to handle the heating elements. However this only covers a very simple use case and to safely and efficiently implement this we would recommend someone with the required knowledge of electronics to design an analog circuit which

can handle the high frequency pwm without potentially breaking the power supply and maybe even separate the elements completely from the Raspberry Pi.

3.4 Cooling element

3.4.1 Hardware

The cooling element in a regular air handling unit is based on vapor-compression, but due to the size of this solution it is not applicable on this small-scale model. As a replacement of the vapor-compression based cooling element we choose to go with a Peltier-element, which cools based on thermoelectric cooling. The pros for us are the size, prize and the fact there are no moving parts. The cooling element is not only used to reduce the temperature of the room it is aswell used to reduce the humidity in the air and that is where the cooling potential of the element comes in.If we take the worst case scenario of 100% humidity to 20% humidity as stated in the requirements we need to cool the air down from 25 °C to 5 °C. So we need atleast a temperature difference of maximum outside temperature as in the requirements specified minus 5 °C for the humidity to get to the minimum humidity. This results in a total 25 °C temperature difference.

Category	Hightech Peltier element	Hightech Peltier element
Type	QC-127-1.4-8.5MD	QC-241-1.0-3.9M
Rated Voltage	15.5 V	29.5 V
Max. current	8.5 A	3.9 A
Wattage (max.)	72 W	64 W
Temperature difference	71 K	71 K
Sizes	$40 \times 40 \times 3.4$ mm	$40 \times 40 \times 3.6$ mm
Price	€43,54	€61,01

Table 2: Comparison of Peltier elements

Table 2 shows a variety of Peltier elements. We chose to go with the lower voltage one to be compliant with the power requirements of other components. Both of the elements are in the range of the temperature difference specified in the above part.

Our chamber is pretty small so we don't need a huge cooling surface, but we wanted to place a cooling element on the cold side of the element to help with the disperse of the cold. This heatsink however has not been installed yet both due to not having been bought. Also during the testing of the element we found out that because the device transfers heat from one of its sides too the other there will have to be another heatsink with fan to dissipate the heat. These however also haven't been bought yet and so have not been installed yet. Wiring has been done but due to the fact the heat can't be dissipated right now we can't run the element for long periods of time.

3.4.2 Software

The code for the cooling element is completely identical to that of the heating elements. It also uses the PWM class, and its `SetValue()` function also limits the value to the 0-30 range before calling the parent class `SetValue()` function.

3.4.3 Problems

We ran into two problems with our cooling element, the first being the same as with the heating element. The cooling element also draws too much power which overloads the transistor. The other problem was that in order to function the element needs to dissipate the heat it extracts from the cold side on the hot side. But since we did not know this during our design phase this wasn't included in the design. To compensate for this we needed to cut a hole in the top of the lit of the intake tube. This would give space to install a heatsink and maybe a fan. But again due to this item not being able to be bought in time this hasn't been installed yet.

3.4.4 Recommendations

The same recommendation as for the heating element stands, we recommend using a mosfet that supports the amperage required for the cooling element. Additionally we recommend looking into other methods of cooling because while this approach will work but the heat will be dissipated in close vicinity of the intake which may interfere with the process.

3.5 Temperature and humidity sensors

3.5.1 Hardware

When researching the temperature and humidity sensors, our original plan was to have separate temperature sensors and humidity sensors. However, it turned out that almost all humidity sensors can also report the temperature, so in the end we decided to use combined temperature/humidity sensors. For the sake of completeness, our original research into temperature-only sensors is also described here.

Temperature-only sensors

Beginning the research process, all components were researched separate of each other. That's why the temperature sensors were also researched as temperature-only sensors. While there are hundreds of sensors that read the temperature, within this research there were a couple of options. Each option doing the same thing but with different maximum temperature readings, connection types or just costs. One however stood out because of the way it communicated with a controller. It was the DS18B20. The way this sensor communicates with the controller is by a 1-wire interface and an unique identifier. This meant that there could be multiple sensors connected to a single interface on the controller where every sensor is reached by its identifier. As the ports on the controller were limited for the project's needs, this was the sensor of choice. This sensor also met

all the other requirements. For example, it can read temperatures from −55 °C to 125 °C. This was more than enough looking at the functional requirements in Chapter 2, where it is stated that we will be measuring temperatures from −10 °C to 30 °C. However since there are sensors where the temperature and humidity sensors are combined, it was decided that buying one sensor with two operations is the way to go. Another reason for going with this decision was because of the physical layout, the temperature sensor would sit next to a humidity sensor at all times.

Combined temperature and humidity sensors

Early on in the design process, we decided on using I2C to communicate with the humidity sensors, because it allows many devices to be placed on a single bus. However, when researching the humidity sensors, it turned out there are many humidity sensors that only have a single fixed I2C address. There were a few with the possibility to select one of two I2C addresses, and we found one sensor that allows a choice between four different I2C addresses.

As visible in Table 3, this was the Texas Instruments HDC1010 (Texas Instruments Inc., 2016a). Using the HDC1010, we'd have to use two I2C busses. While there is a second I2C bus on the Raspberry Pi, enabling it is not straightforward and requires both changing kernel configuration files and soldering a header onto the Raspberry Pi. Another disadvantage of the HDC1010 is that it's only sold as a surface-mount device, which means it needs to be soldered onto a PCB by baking it. While there exist breakout boards for the HDC1010, they were all out of stock or discontinued.

Another option was to use an I2C multiplexer, and a simple humidity sensor with one I2C address. We found several of these, their details are listed in Table 3. It becomes apparent that there is a fairly direct relationship between price and capabilities. The more you pay, the higher range and the better accuracy you get. Because all three sensors are good enough for our purposes, we chose to use the Silicon Labs Si7021 (Silicon Laboratories Inc, 2016), the cheapest of the three.

For the I2C multiplexer, the one that was recommended to us online was the Texas Instruments TCA9548A (Texas Instruments Inc., 2016b), which is an easily controllable multiplexer. It is readily available on a breakout board from several Dutch online stores, and does not cost much.

Name	HDC1010	HTU21D	Si7021	SHT31-D
No. of I2C addresses	4	1	1	1
Min. temp	5 °C	−40 °C	−10 °C	−40 °C
Max. temp	60 °C	125 °C	85 °C	125 °C
Temp. uncertainty	±0.2 °C	±0.3 °C	±0.4 °C	±0.3 °C
Min. humidity	0 %	5 %	0 %	0 %
Max. humidity	100 %	95 %	80 %	100 %
Humidity uncertainty	±2 %	±2 %	±3 %	±2 %
Price	€2,52	$14.95	$6.95	$13.95

Table 3: Comparison of temperature and humidity sensors (Texas Instruments Inc., 2016a) (Measurement Specialties, Inc, 2013) (Silicon Laboratories Inc, 2016) (SENSIRION AG, 2015)

When we chose to go with the combined sensor which includes both the humidity and the temperature sensors, we chose to go with Si7021. These I2C-enabled sensor have hardcoded I2C addresses, in this case as shown in the datasheet 0x40 (Silicon Laboratories Inc, 2016). To overcome this problem we installed an I2C multiplexer, a TCA9548A. This multiplexer is where all of the wiring harnesses are connected. After wiring the wiring harnesses which include power, ground and sda/scl for each individual sensor, we routed them through the case. The wires are capped off at each end with headers so we can easily swap out components if they break. The power for the sensors is supplied by the power supply, and is routed through the breadboard to the sensors. This was needed because the Raspberry Pi was not capable of delivering the required power.

3.5.2 Software

Sensors

The the temperature and humidity sensors are very similar in code, because they both read data from the Si7021 chips. They both implement the iSensor abstract class, and the only differences are in the `GetValue()` function.

The `Humidity::GetValue()` function first calls the `Communication->Write()` function (line 4), with the data byte 0xF5. This data byte is the command for "Measure relative humidity" (Silicon Laboratories Inc, 2016). After that, the relative humidity can be fetched with a call to `Communication->Read()` (line 8). Next, we convert the two bytes received into an integer, and use that to calculate the relative humidity in percent using the formula in §5.1.1 of the Si7021 Datasheet (lines 10-11).

```
1  double Humidity::GetValue()
2  {
3    //Measure RH, No-hold master mode
4    Communication->Write(0xF5, Address);
5
6    //Get result
7    uint8_t buffer[2] = {0};
8    Communication->Read(buffer, 2);
9
10   int result = (buffer[0] <<8) + buffer[1];
11   double humidity = (125.0*result)/65536 - 6;
12   return humidity;
13 }
```

The temperature is measured in a very similar manner. There are two differences: we write byte 0xE0 ("Read Temperature Value from Previous RH Measurement") instead of 0xF5, and the formula for calculating the temperature is `double temperature = (175.72*result)/ 65536 - 46.85` (Si7021 Datasheet, §5.1.2)

Multiplexer

Because all the Si7021 chips use the same I2C address, they can not be connected to the same I2C bus. This is why we decided to use an I2C multiplexer. All of the code to handle the multiplexer is in the I2C class.

First, the I2C class implements the iCommunication interface. This means it has a `Write()` function, which writes a byte to an address, and a `Read()` function, which reads data into a buffer. It also has some helper functions, namely `_setSlave()`, `_write()`, and `_writeByte()`.

```
1  class I2C : public iCommunication
2  {
3  public:
4    I2C();
5    virtual ~I2C();
6    bool Write(uint8_t data,uint8_t address); //return
        true on succes
7    bool Read(uint8_t* buffer, uint8_t length);
8    bool Initialize();
9  private:
10   bool _setSlave(uint8_t address); //return true on
        succes
11   bool _write(uint8_t address, uint8_t data); //return
         true on succes
12   bool _writeByte(uint8_t data);
13   int filedescriptor;
14   uint8_t currentTarget;
15 };
```

To fully understand the I2C class, let's walk through a call to `Write()`. The `Write()` function takes two parameters: the byte to send (`data`), and the address, which is a number from 1 to 8, depending on which pins of the multiplexer we want to use. First, we check if the multiplexer is already set to the address we need to talk to (line 2). If it is, we immediately go to the `_write()` call at the end of the function (line 8). If it is not, we first call `_setSlave()` (line 3).

```
1  bool I2C::Write(uint8_t data, uint8_t address){
2    if(address != currentTarget){
3      if(!_setSlave(address)){
4        std::cerr << "Did not try to write " << data <<
            " to " << address << std::endl;
5        return false;
6      }
7    }
8    return _write(SENSOR_ADDRESS, data);
9  }
```

`_setSlave()` first creates a bitmask for the multiplexer to work with (line 2) (source: TCA9548A datasheet). Next, it calls `_write()` to write the bitmask to the multiplexer (line 3). If the `_write()` call succeeds, we also save the `currentTarget` (line 4).

```
1  bool I2C::_setSlave(uint8_t target){
2    int data = 1 << target; // create bitmask from
       target number
3    if(_write(MULTIPLEXER_ADDRESS, data)){
4      currentTarget = target;
5      return true;
6    }
7    std::cerr << "Failed to set slave to " << target <<
       std::endl;
8    return false;
9  }
```

The `_write()` function uses the syscall `ioctl()` (line 2), which tells Linux which slave address to use for the next I2C read and write calls (source: Kernel i2c documentation). (N.B.: the filedescriptor is an integer which points to the `/dev/i2c-1` special file. This file is opened in the constructor.) Next, it calls `_writeByte()` to actually send the data to the I2C bus (line 8).

```
1  bool I2C::_write(uint8_t address, uint8_t data){
2    if((ioctl(filedescriptor, I2C_SLAVE, address)) < 0)
3    {
4      std::cerr << "Failed to acquire bus access and/or
         talk to slave.\n";
5      std::cerr << strerror(errno) << std::endl;
6      return false;
7    }
8    return _writeByte(data);
9  }
```

Last, there is the `_writeByte()` function. This function first creates a two-character buffer, containing the data byte and a null character (line 2). Next. we call the syscall `write()`, which requires a null-terminated character array (line 4). If the syscall succeeds, we have succesfully written the data byte to the I2C bus.

```
1  bool I2C::_writeByte(uint8_t data){
2    uint8_t buffer[2] = {data, 0};
3    int length = 1;
4    if(write(filedescriptor, buffer, length) != length)
5    {
6      std::cerr << "Failed to write to the i2c bus.\n";
7      std::cerr << strerror(errno) << std::endl;
8      return false;
9    }
10   return true;
11 }
```

Completely separate from the write functions, there is the `I2C::Read()` function. First, we create a temporary buffer to store data in (line 2). Next, we keep trying to read data into this buffer from the I2C special file (line 5). Once we succeed, we copy all but the last byte (which is the end-of-file marker) from the temporary buffer to the buffer that was provided by the caller (line 14).

```
1  bool I2C::Read(uint8_t* buffer, uint8_t length){
2    uint8_t tmpbuffer[length + 1] = {0};
3    bool success = false;
4    while(!success){
5      if(read(filedescriptor, tmpbuffer, length+1) !=
           length+1)
6      {
7        usleep(100);
8      }
9      else
10     {
11       success = true;
12     }
13   }
14   memcpy(buffer, tmpbuffer, length);
15   return true;
16 }
```

3.5.3 Problems

One of the problems we could have run into was that of addressing the Si7021 sensors, because their I2C address is not modifiable. Thankfully, we foresaw this issue, so we dealt with it by purchasing an I2C multiplexer. Another issue is that in the current design of the code, after a measurement command, the program blocks while the measurement is being taken. This means the system could noticeably slow down if this part of the API is used in larger systems with more sensors. It currently is not a problem because the system does not have to be fast. Finally the Raspberry Pi wasn't capable of delivering the required amount of power; this was resolved by powering them from the power supply directly.

3.5.4 Recommendations

We would recommend using sensors with programmable addresses because this would greatly simplify the cable harness and eliminate the need for a multiplexer.

3.6 CO_2 sensors

3.6.1 Hardware

The CO_2 sensors are needed to measure how much fresh air the unit needs to take in to control the CO_2 level in the room. There are several options to measure the CO_2, one of which is the NDIR-type sensor. NDIR stands for non-dispersive infrared, which is a principle to detect the existence of CO_2 in the air. The benefit of an NDIR sensor is that an NDIR sensor already has a built-in heating element, which is necessary for measuring the CO_2 levels. If we go for a non-NDIR sensor, we also have to provide an external heating source to measure the CO_2 level. By choosing an NDIR sensor we don't need an extra

heating element, therefore the controls of the CO_2 sensor are simpler than with a non-NDIR sensor where you have to control the sensor and the external heating element. This makes it easier for the students who are going to work with the air handling unit to get a general overview of the unit, without having to delve into the details of the CO_2 sensor. It also makes it easier for them to change the sensor for another model if they have little programming skills. That's why we chose an NDIR-type CO_2 sensor.

For our purposes, mentioned earlier, we decided to use an MH series CO_2 sensor. The MH series has small sensor who could fit easily in the unit and they are relatively cheap for an NDIR CO_2 sensor. Therefore the choice came down to two sensors, the MH-Z14a and the MH-Z19, which are compared in Table 4.

	MH-Z14a	MH-Z19
Working voltage	4.5–5.5 V	4.5–5.5 V
Average current	<60 mA	<85 mA
Interface Level	3.3 V (5 V compatible)	3.3 V
Measuring range	0–10 000 ppm	0–5000 ppm
Output signal	PWM	PWM
	UART	UART
	Analog voltage signal	Analog voltage signal
Preheat time	3 min	3 min
Working temperature	0–50 °C	0–50 °C
Working humidity	0–90 %RH	0–90 %RH
Dimension (LxWxH)	57.5 × 34.7 × 14.6 mm	57.5 × 34.7 × 16 mm

Table 4: Comparison of CO_2 sensors (Zhengzhou Winsen Electronics Technology Co., Ltd., 2017a) (Zhengzhou Winsen Electronics Technology Co., Ltd., 2017b)

The working temperatures of both sensors don't go as far as the −10 °C described in the function requirements, but this applies to all the sensors in our price range. If we go for a CO_2 sensor that is capable of working in −10 °C and has a good availability, the price would be about ten times higher as the price is now. For that reason we went for a sensor with a working temperature that only goes to 0 °C, but is affordable and fits inside the budget.

The MH-Z14a has a measuring range from 0 to 10000 ppm (parts per million). Legally the maximum allowed CO_2 value is 1000 ppm in the Netherlands (Blok, 2015). Knowing this, the measuring range of the MH-Z19, which is 0 to 5000 ppm, will be more than enough. Furthermore, it is going to be difficult to reach a CO_2 value higher than 5000 ppm in our mock-up.

Therefore the choice went to the MH-Z19 NDIR sensor, as depicted in Figure 6. The functionality is as good as the MH-Z14a, while the MH-Z19 is a bit cheaper, fits better in our design and has a better availability.

We decided to use the UART communication capabilities of the MH-Z19. This immediately presented a problem as the Raspberry Pi only has one UART interface, and since the sensor address is hardwired in the sensor, we needed to find a solution. By wiring the transmit pin to both sensors and the receive wires to different pins on the Raspberry Pi we could distinguish between the two sensors. These wires were also incorporated into the cable harness but due

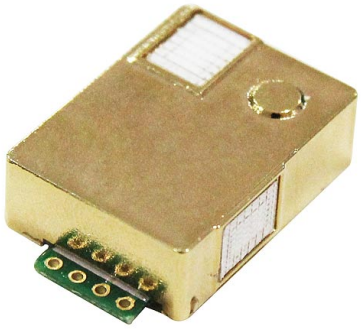


Figure 6: MH-Z19 CO_2 sensor (Zhengzhou Winsen Electronics Technology Co., Ltd., 2017b)

to them working on 5 V instead of the 3.3 V on which the I2C works they needed a different power line.

3.6.2 Software

The chosen hardware, MH-Z19, has 2 manners of sending data to the Raspberry Pi. The first being a PWM signal and the second being UART. After researching the possibilities we decided on working with UART. This way of communicating is in this case only used for the CO_2 sensors.

Serial

To get serial communication working on a Raspberry Pi which is running Linux, a few values need to be changed within the OS its files. These values keep the serial port occupied for the OS itself, which in our case is not good as we need that port to communicate. These values have to be changed by calling the `raspi-config` and disabling serial communication for the OS. Another change has to be done in the file `/boot/cmdline.txt` where the specified port name is set to the OS it's communication, which in our case again needs to be released.

After doing this the port is open to use and will work with any code. The initialization for serial is seen below:

```
1  bool Serial::Initialize()
2  {
3    uart0_filestream = open("/dev/ttyS0", O_RDWR |
       O_NOCTTY | O_NDELAY);
4    if (uart0_filestream == -1){
5      std::cout << "Error - Unable to open UART." << std
         ::endl;
6      return false;
7    }
8
9    struct termios options;
10   tcgetattr(uart0_filestream, &options);
11   options.c_cflag = B9600 | CS8 | CLOCAL | CREAD;
```

```
12    options.c_iflag = IGNPAR;
13    options.c_oflag = 0;
14    options.c_lflag = 0;
15    tcflush(uart0_filestream, TCIFLUSH);
16    tcsetattr(uart0_filestream, TCSANOW, &options);
17    return true;
18  }
```

Line 3 opens the serial port with line 4 to 7 being its error handling. Line 9 to 16 contain the settings that are used for the serial communication, for example `baudrate = 9600` and no parity.

Besides this the Serial class has two other functions, The `Read()` and `Write()` functions. While the only main function for using serial for the CO_2 sensor is reading the value, you have to write to the sensor to give it a signal to start a measurement.

The write function can be seen below:

```
1   bool Serial::Write(uint8_t data, uint8_t address)
2   {
3     unsigned char txBuffer[20];
4     unsigned char *p_tx_buffer;
5
6     p_tx_buffer = &txBuffer[0];
7     *p_tx_buffer++ = 0xFF;
8     *p_tx_buffer++ = address;
9     *p_tx_buffer++ = 0x86;
10    *p_tx_buffer++ = 0x00;
11    *p_tx_buffer++ = 0x00;
12    *p_tx_buffer++ = 0x00;
13    *p_tx_buffer++ = 0x00;
14    *p_tx_buffer++ = 0x00;
15    *p_tx_buffer++ = 0x79;
16
17    if (uart0_filestream != -1)
18    {
19      int count = write(uart0_filestream, &txBuffer,
          sizeof(txBuffer) - 1);
20      if (count < 0)
21      {
22        std::cerr << "UART TX ERROR" << std::endl;
23        std::cerr << "data " << data << " ,adress " <<
            address << std::endl;
24        return false;
25      }
26    }
27    return true;
28  }
```

Line 6 to 15 define the array that is sent to the CO_2 signal to trigger it to start the measurement, this array is predefined by the makers of the MH-Z19 and can be traced back to its datasheet (Zhengzhou Winsen Electronics Technology

Co., Ltd., 2017b). This is hardcoded within the write function because this was just not taken into account. At the time of coding the code was written with the array hardcoded for testing reasons and speed up the process of getting everything working. Line 19 is where the data is sent to the sensor with line 20 to 25 being its error handling.

The read code:

```
1  bool Serial::Read(uint8_t* data, uint8_t length)
2  {
3    bool doneReading = false;
4
5
6    while (!doneReading)
7    {
8      if (uart0_filestream != -1)
9      {
10       unsigned char rx_buffer[length];
11       int rx_lenght = read(uart0_filestream, (void*)
             rx_buffer, length);
12       if (rx_lenght > 0)
13       {
14         uint8_t high = (uint8_t)rx_buffer[2];
15         uint8_t low = (uint8_t)rx_buffer[3];
16         data[0] = high;
17         data[1] = low;
18         doneReading = true;
19       }
20     }
21   }
22   return true;
23 }
```

The main code of this function is put in a while because the port needs to be open for data reading. Whenever the data is read once, the while loop stops and the function carries on. The main read of data is done on line 10, after which the CO_2 specific decoding is done and the global variables are set.

Sensors

The CO_2 sensors have a specific class as well. This class is for the component management and is used to handle the received data. As a class itself this isn't the most complicated class because a part of the handling is done by the serial class.

The main function of the class, `getValue()` can be seen below:

```
1  double CO2::GetValue()
2  {
3    Communication->Write(NULL, Address);
4
5
6    uint8_t* value = { 0 };
7    Communication->Read(value, 9);
```

```
8
9
10    double result = (256 * value[0]) + value[1];
11    return result;
12  }
```

The function simply calls the `Write()` function of the communication class followed by a `Read()` call. The value that comes out of this is then recalculated to the exact format, ppm in this case and then returned to be stored by the Controller class. The way the ppm has to be calculated is also specified in the datasheet (Zhengzhou Winsen Electronics Technology Co., Ltd., 2017b).

3.6.3 Problems

If using multiple CO2 sensors there is a problem. The ID's of the sensors can't be changed so the identification of multiple sensors is difficult to do when they are all connected to the same serial port. A solution for this problem is using a multiplexer or using a different GPIO pin for each sensor, in which case the sensors can be kept separate of each other. So the program writes to all sensors at one time and reads on different GPIO pins

3.6.4 Recommendations

We would recommend using a multiplexer if it's desired to have even more sensors, in other cases using different GPIO pins for each sensor will suffice.

3.7 Fans

3.7.1 Hardware

The fans are used to blow the air through the handling tunnel into the room. There are two fans in our air handling unit, one to take air into the handling tunnel and one at the end of the handling tunnel just before the air enters the room. We placed the second fan so that the intake air doesn't go into the tunnel for return air, but goes straight into the room.

The kind of fan they use in a real air handling unit is way too powerful and too big for our mock-up. In fact, almost all normal fans are too big for our purpose. To solve this problem we came up with a computer fan, which are small but still relatively powerful.

The only important specification for the fan are the sizes and that the rpm must be controllable, so that the intake of air can be controlled.

After some research, we came to the Aerocool Shark Fan, which is depicted in Figure 7. With its 12 cm diameter it will fit into the 15×15 cm handling tunnel. Because of its three pin connection it is possible to control the rpm. As stated in the functional requirements, the air handling unit needs to be able to refresh the room at least three times per hour. That came down to the fan needing to be capable of moving 104.2 cm^3 air per hour. As visible in Table 5, the Aerocool Shark Fan has an maximum airflow of 140.3 m^3/h, which is more than enough to refresh the room three times per hour. The fan of choice actually is a bit overpowered but the air we need to move to refresh the room three times

per hour is very low, which makes almost every computer fan suitable for out mock-up, as long as the rpm is controllable.

A last argument is that the fan is not that expensive and has a good availability at several webshops.

Power connection	3-pin
Voltage range	7–12 V
Maximum rpm	1500 rpm
Volume	12.6–26.5 dB
Maximum airflow	140.3 m^3/h
Power	3.6 W
sizes(WxHxD)	120 × 120 × 25 mm

Table 5: Aerocool Shark Fan specifications (Alternate Nederland, 2017)

Figure 7: Aerocool Shark Fan (Alternate Nederland, 2017)

However, due to ordering problems, we decided to use some cheap 120 mm pc fans which we had laying around. The fans have been installed on either side of the intake tube. These are connected to the power supply via two TIP120 transistors. While these were not powerful enough for the heating and cooling elements, they are powerful enough for the fans.

3.7.2 Software

The code for the fans is very similar to that of the heating and cooling elements. It also uses the PWM class to send a PWM signal, and it has a `SetValue()` function which limits the value to within an acceptable range. However, for the fans, we found that the acceptable range was between 0 and 100.

```
1 void Fan::SetValue(uint8_t value)
```

```
2 {
3   if (value < 0)
4     value = 0;
5   else if (value > 100)
6     value = 100;
7   iActuator::SetValue(value);
8 }
```

3.7.3 Problems

We didn't run into any problems with the fans. We however had some problems with some wiring (we shorted a wire) but this was quickly discovered and fixed.

3.8 Vents

3.8.1 Hardware

To control where the air goes to, we have decided to use vents. By using these vents we could potentially return exhaust air into the intake tube to be mixed with fresh air or maybe run the system on a loop where there is no fresh air taken from outside. While researching the possible options for these vents, the conclusion could be made that there are no off the shelf electrically controlled air vents. So the task to research this component instantly became bigger. The task was no longer for only a controllable vent but for a manual controllable vent and a way to automate this so its electronically controlled. Within this task there were multiple manual vent options as seen in Table 6.

While looking at this table the decision was made to use the "Sliding Vent" option and use a servo in place of the manual knob to automate it. When this was discussed with the client, he expressed the fact that it would compromise airflow when fully open and he requested to come with another option.

The next best option was the "Self-made car-style vent". While this vent is more labour intensive and harder to automate than the "Sliding Vent", this vent will have near to none airflow compromise when fully open. For the building of this vent, there will be some slats which can rotate to open and close that given passageway. The technical side is that the slats will be rotated with a servo motor, more about which servo in the next paragraph of this chapter. The servo motor will be connected to all the slats via a rod or stick which can rotate all slats at once. By doing this we will be able to set the servo at a given angle by which will rotate and either open or close depending the set angle.

Looking at the servos needed with all the options for the vents, the decision was made to use the simplest servo that is adequate for our requirements, because we do not need anything more expensive or complicated. The servos also need to be capable of rotating the desired amount of degrees. After setting these requirements, we found out that there are two main types of motor: the servo type and the stepper motor type. As the stepper motor needs a driver and its only main advantage is a more precise control, we went with the servo motor. Whe chose the Modelcraft MC1811 (Conrad Electronic International GmbH & Co. KG, 2017c). It met the requirements set before, was in stock and within budget.

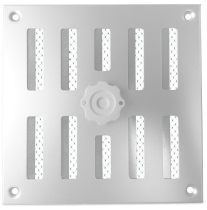
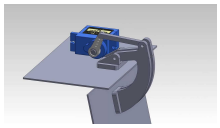
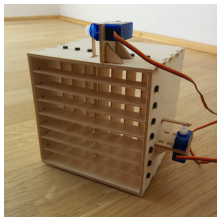
	Sliding Vent	Butterfly Vent	Self-made latch mechanism	Self-made car-style vent
Graphical example	 (Ventilatieshop Ede, 2017)	 (Conrad Electronic International GmbH & Co. KG, 2017b)	 (Goodliffe, 2013)	 (Hanno, 2016)
Positives	Fairly easy to add servo in the middle to automate Desired dimensions easily found Fairly cheap	Air tight sealed when closed Airflow highly controllable	Fairly cheap Not too difficult to build	Fairly Cheap Not too difficult to build Much more airflow control if compared to the "Self-made latch mechanism"
Negatives	Amount of airflow compromised even when fully open	Expensive Very difficult to add servo for automation	Making it close air tight could be challenging	Making it close air tight could be challenging Required precision work with small pieces (doing something wrong while building it is very easy)
Price	€11,99	€49,99	N/A	N/A

Table 6: Comparison of vent options

We installed and wired the servos to the Raspberry Pi while the power was delivered by the power supply. After this we started to think about how to make the vent mechanism. However at this point we were so late in the building phase, and since we could not come up with a good way of manufacturing the pieces, we decided together with the client that the vents would not be implemented in this project, and that we should focus on the api and other software components.

3.8.2 Software

The software for the vents needs to control the servos. For this, we can use the PWM class. However, because of the strict timing requirements of the servos, the PWM class needs to be modified. Due to a lack of time, this has not yet been done. Therefore, at the moment of writing, the servos can not yet be controlled.

3.8.3 Problems

We ran into quite a lot of problems with the vents, first our initial design restricted to much airflow according to the client. Then we could not find a replacement design that would and that could be delivered in time. After this we decided to build our own but this ended up being too much of a job to tackle with available knowledge and building skills. So we did not end up with a working prototype for the vents.

3.8.4 Recommendations

We would recommend finding someone with the required skills to design and build a vent which would work with the design. Preferably while using the servos that are included with the current prototype.

3.9 Power supply

3.9.1 Hardware

To power the entire system, we needed a strong and reliable power supply, which would be easy to connect and control. To do this, we chose to use an ATX power supply, because they are widely available (being used in practically all desktop computers). They are also generally well-documented, both in terms of connections and interfaces and concerning the power draw capabilities. After adding up the power requirements of the components, we concluded we would be drawing a maximum of just over 500 W. This power is mainly for the heating elements (two elements at 180 W each) and the cooling element (at 130 W). The controller can draw about 13 W, the fans draw less than 4 W each, and the sensors all draw less than 2 W, with the humidity sensors drawing a negligible 650 µW.

With the total coming just over 500 W, we decided to look for a 650 W power supply, because ATX power supplies are at their best performance when their power draw is between 70 % and 80 % of their maximum power draw. Because we have to work with a limited budget, we started looking at power supplies in order of price. However, the first power supply unit (PSU) we came across already fit all of our requirements, while being significantly cheaper than the next power supply in the list. This was the Sharkoon SHP650 v2, which we decided on buying.

While ordering the components we found out that ordering PC components would be a bigger problem than expected. Due too this we were forced to use a power supply a project member had laying around so we could continue with the building process. Near the end of the building process, we managed to purchase a power supply, this time a LC-Power LC420H-12 PSU. This power supply is capable of delivering 420 W instead of the calculated 650 W. We decided this would not be a problem, because we had not noticed any issues using the old temporary power supply, which was rated at only 230 W. Swapping out the temporary power supply with the new one was practically effortless.

3.9.2 Problems

The only problem we encountered during the installation of the power supply we shorted a wire which caused the power supply to stop working for a while, but fortunately the power supply has a built-in protection and kept working after we cleared it.

3.9.3 Recommendations

We would highly recommend using a PC ATX power supply due to its tough nature, it won't break the power supply and/or breaker box when shorted and wiring is easy since it delivers the desired voltages natively.

Chapter 4

Software

4.1 API

We decided to create an API structure for the software. All the control and communication of the components will be taken care of in the API implementation. By doing it this way, the students can implement their mathematical model for controlling the air in the room, but they do not need to worry about the communication between the components and the Raspberry Pi. Another reason for this is that we can make sure that all the components will work and communicate correctly, so when we deliver the air handling unit with the API, you will get a working unit. See Appendix 10.A. for the software design.

We divided the unit into three categories: actuators, sensors and communication. All the components that are controllable belong under the actuator category, such as the heating elements and the fans. The components that give data belong under the sensor category, these are all the sensors. As last we have the communication category. All the components have a different way of communicating, like I2C or UART for example. These communication methods are put within the communication category.

To keep the code expandable and clear, the categories are translated to an interface. Behind the interfaces are all the different kinds of sensors, actuators or communication methods defined as a class. The actuator and sensor classes both have a communication member, so that it is easy to see which sensor or actuator has which kind of communication method. During the implementation the discovery was made that by using the iActuator class as an interface, there would be duplicated code in the inherited classes. Therefore the iActuator class transformed from an ABC class to a base class. In this way, the code that's the same for all the classes will be placed in the iActuator class to prevent duplicate code.

Because we went for an API design, we wanted a master class. This master class, named Controller, is the only class people can talk to. The Controller will take care of the rest. This means everybody only needs to talk to the Controller, and they do not need to worry about the rest. For example, when somebody wants to set heating element 1 to its maximum capacity, it can say that to the Controller and the Controller will pass it on to the heating element. The Initialize function from the Controller will create all the components that are in

the air handling unit. The Controller saves all the components into two maps, a SensorMap for all the sensors and an ActuatorMap for all the actuators. So when somebody asks the controller to set a particular actuator, it will be able to find it in the map. The code for the `SetActuatorValue` is as follows:

```
1  void Controller::SetActuatorValue(std::string name,
       uint8_t value){
2    std::map<std::string, iActuator*>::iterator f(
         ActuatorMap.find(name));
3    if (f != ActuatorMap.end())
4    {
5      f->second->SetValue(value);
6    }
7    else
8      std::cout << "ERROR: actuator " << name << " doesn
         't exist" << std::endl;
9  }
```

The function searches through the map for the actuator name. When it finds the component, the function calls the `SetValue` function from the found actuator. The `GetSensorValue` does something similar, with the differences that it runs through a list of all the sensors, and that it returns a map with all the sensor names and corresponding values in it.

```
1  std::map<std::string, double> Controller::
       GetSensorValue()
2  {
3    std::map<std::string, double> returnMap;
4    for (std::map<std::string, iSensor*>::const_iterator
         i = SensorMap.begin(); i != SensorMap.end(); i
        ++)
5    {
6      double value = i->second->GetValue();
7      returnMap[i->first] = value;
8    }
9    return returnMap;
10 }
```

To make sure the outside classes who use the API all keep talking to the same Controller (and thus to the same components behind it), a Controller-Builder class is created. The ControllerBuilder class is the one who creates the Controller. A random class can ask the Builder to give the Controller and set its own Controller member with this information. The ControllerBuilder has a static Controller member named Instance. The first time a class calls the Get-Controller function, the ControllerBuilder will create the Controller and set its own Instance member it. The next time a class calls the function, the Instance is already initialized so the ControllerBuilder can directly return the Instance. The code looks as follow:

```
1  Controller* ControllerBuilder::GetController()
2  {
3    if (Instance == NULL)
```

```
4       Instance = new Controller();
5     return Instance;
6   }
```

Looking back at it now, it would have been a better choice to remove the ControllerBuilder and convert the Controller to a singleton. This is a pattern which we learned halfway during the project. At this time the design had already been made and pieces were implemented. The singleton does in principle the same as the ControllerBuilder, only with one class fewer.

4.2 Control software

With the API alone, it is not possible to operate the air handling unit. You need something that talks width the API to control the actuators inside the unit. Therefore we decided to make a control application in combination with a user interface, in this case a web based interface. By making this application we create an example for the other students that are going to use this system, the mock-up with the API software. They can see in this application how the control software communicates with the API and how the flow is. The mathematics in the code that are necessary for a good automatic control system are going to be poor from our side, because we don't have the correct expertise for that. The students who are going to use it do have the correct expertise, so they can correct and expand that part. The important thing is that they can see how it works in combination with the API and show them a way to do it.

The application is relatively simple, just because it is a way to show that the API works and because the fact that it is an example application. In appendix 10.B is the class diagram shown of the control application. There are two ways of controlling the air handling unit, manual and automatic. Therefore there are two classes which inherit from the Control interface. The SettingReader is a static class which is being used by the Automatic and Manual class. The functionality of the SettingReader class consist of reading and writing to several files. The control application communicates through these files with the web interface. The files contain the values of all the sensors and actuators.

The update function from Manual control is shown in appendix 10.C as a sequence diagram. In the show cycle is there has been a change in a fan setting. The `settingchanged` file is being checked first. Instead of checking every file every cycle, it only checks the settingchanged file. The contents of this file include which actuator value has been changed. The file is empty when there are no changes. In this scenario the file will contain `fan`. Because there has been change, the WriteActuatorValue function is called with fan as a parameter. Now the system knows it needs to check the fan file for the value of the changed fan. This is done by the SettingReader class. This function, ReadSettings, will return a map with the fan name and the corresponding value. The name is necessary for the Controller to find the correct fan. So the name and value are passed on to the Controller who takes care of the rest. The Controller finds the correct fan through its name. The fan class will then do some checks on the given value to make sure it is a correct value and passes it through to its base class, iActuator, who will eventually send the value via its communication method to the actual fan.

The next thing is updating the sensor values for the web interface. The WriteSensorValues function of the SettingReader is called with as parameter the return map from the GetSensorValue function of the Controller.

As last there is a check if it is time to log all the sensor values. For educational reasons the application logs all the values of the sensors every ten seconds, so students can look back at how the values changed after a certain operation on the air.

The code in the update function is as follow:

```
1  void Manual::Update(){
2    std::map<std::string, bool> changed = SettingReader
         ::CheckChangedSettings();
3    if (changed["fan"])
4    {
5      WriteActuatorValue("fan");
6    }
7    if (changed["heating"])
8    {
9      WriteActuatorValue("heating");
10   }
11   if (changed["cooling"])
12   {
13     WriteActuatorValue("cooling");
14   }
15   if (changed["vents"])
16   {
17     WriteActuatorValue("vents");
18   }
19   SettingReader::WriteSensorValues(mController->
         GetSensorValue());
20   LogSensorValues();
21 }
```

The SettingReader::CheckChangedSettings() function returns a map with the component name and a true if it's existing in the settingchanged file.

```
1  void Manual::WriteActuatorValue(std::string actuator){
2    std::map<std::string, int> componentsValue;
3
4    componentsValue = SettingReader::ReadSetings(
         actuator);
5    for (std::map<std::string, int>::const_iterator i =
         componentsValue.begin(); i != componentsValue.end
         (); i++)
6      mController->SetActuatorValue(i->first, i->second)
           ;
7  }
```

```
1  void Manual::LogSensorValues(){
2    static std::time_t startTime = std::time(nullptr);
3    std::time_t timePassed = std::time(nullptr) -
         startTime;
```

```
4     if (timePassed > 10.0)
5     {
6       SettingReader::LogSensorValues(mController->
            GetSensorValue());
7       startTime = std::time(nullptr);
8       std::cout << "logging because duration is " <<
            timePassed << std::endl;
9     }
10  }
```

Unfortunately the Automatic class is not implemented. This has to do with the limited time we had when all the components were installed into the unit. There was not enough time to create and check the control system for an automatic control. We did the writing of the code and the building of the unit simultaneously, but you need a working unit to test the automatic control, while for the manual mode the testing can be done with the loose components. We saw this coming and discussed it with the product owner. He told us it is unfortunate, but the most important thing for him was that the software which was controlling the unit, so the API, was working and that we laid a solid base for other students to work with. By finishing the manual control we could show that the API was working correctly.

4.3 Webdesign

To interface with the control software, we chose to go the route of using a web page. By using a web page access is granted for all users connected to the network and there is no need for software on the host pc except for a standard web browser. Using this webpage, the user can perform basic tasks to adjust or read data from the air handling unit. For example the operation mode can be chosen within the webpage, the website gets the user input, saves it in a specified file and changes a file that specifies which variable changed within the system. The usage of this extra file that specifies the change is explained in Section 4.2 - Control software. Like this all the other working values (speed, working percentage) of each component can be changed.

The webpage also has a section that displays all the values of the different sensors that are present within the unit. While this shows that the API works, this can also be very helpful when students are implementing their own code and want to see the actual values of the sensors. The only disadvantage on this example is that the web page would only work with the manner of information that the by us created control software delivers. This could be copied to a new project or the control software could be used to display information only (for when students want to implement their own control software and don't want to use the control side of our control software).

As example for the web code, the code for changing the operating mode is seen below:

```
1  <!DOCTYPE HTML>
2  <html>
3  <head>
4  </head>
```

```
5  <body>
6  <h2>Please select the desired mode</h2>
7
8  <?php
9  #/home/pi/git/AirHandlingUnit/_Controls/config/mode.
       txt
10 $file_name = "mode";
11 $path = "/home/pi/git/AirHandlingUnit/_Controls/config
       /" . $file_name . ".txt";
12 $file = fopen($path, "r+") or die("Unable to open file
       ");
13 $data = fread($file,(filesize("mode.txt") + 1));
14 $operation_mode = NULL;
15 $value = NULL;
16
17 if (!empty($_POST))
18 {
19   switch($_POST['Mode']) {
20        case "Manual":
21       $operation_mode = "manual";
22       break;
23     case "Automatic":
24       $operation_mode = "automatic";
25       break;
26   }
27
28   $file = fopen($path,"w");
29   fwrite($file, $operation_mode);
30
31   $setting = fopen("settingChanged.txt", "w");
32   fwrite($setting, $file_name);
33   fclose($setting);
34 }
35
36 if($value !== NULL) {
37   $data = $operation_mode;
38 } else {
39   if(empty($data)){
40     $data = NULL;
41   }
42 }
43 ?>
44 <form action=<?php echo htmlspecialchars($_SERVER["
       PHP_SELF"]); ?> method="post">
45   <input type="radio" name="Mode" value="Manual">
       Manual
46   <input type="radio" name="Mode" value="Automatic"
       disabled=disabled>Automatic<br><br>
47   <input type="submit" value="Save!">
48 </form>
```

```
49 <?php if($data !== NULL){ ?>
50 <h4> Current value: <?php echo $data ?> </h4>
51 <?php }
52 fclose($file);
53 ?>
54 </body>
55 </html>
```

Line 9 to 13 are used to read the file that specifies the operation mode. The webpage reads this before any changes to display the current operation mode. This would therefore be displayed at any time. The implementation of this can be found on lines 49 to 51.

Then lines 14 and 15 declare 2 variables that are used within the program. At line 17 the webpage checks if the user has done an input. This is necessary because the page is also used as display for the current operation mode as stated before. If the user has changed the operation mode, the page checks to which operation mode is switched. After this the chosen operation mode is written to the specified file and the general `settingChanged.txt` file is also changed to signal the control software that a setting changed.

In line 36 to 41 the page checks if there is a new operation mode (stored in variable `value`) and sets the display variable to this value. This way the new operation mode is directly displayed. If there is no new operation mode chosen and the file was empty as well (in case of starting the system), the display variable is set to `NULL`. If this variable is `NULL`, the web page does not display the current value.

In Figure 8 a snapshot of the webpage is seen. It is a snapshot of the page where the user can select the operation mode. At the left side all the components and subjects can be selected. The main part is where the user can select the operation mode and see the current value.

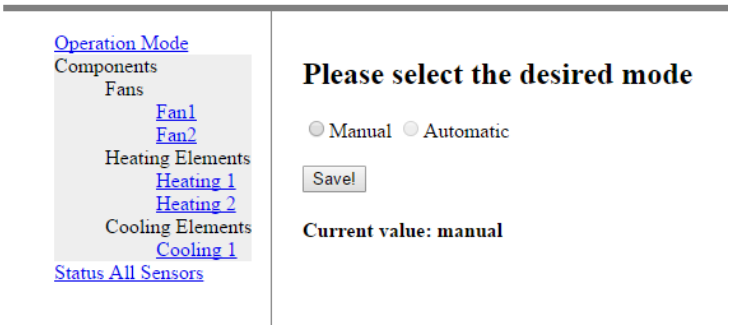


Figure 8: A part of the webpage that controls the AHU

Chapter 5

Conclusion

This report details the selection and implementation of the sensors and actuators needed for completing the requirements. These requirements as detailed in Chapter 2 were mostly finished. We are able to read CO_2, humidity, and temperature levels, and they are shown in the web interface. We are also able to manipulate these variables in the room, however not to the extent we wanted. Physically it's not all possible, but the code infrastructure is capable of controlling all the actuators. The fans and heating elements can be controlled. The cooling element and the vents are not working, just because they are physically not connected or working. During the project a lot of effort was put into ensuring the case was small, cheap and easy to replicate, this has been accomplished. The price was under the allotted amount, the case can be carried by two people and the components are all sourced locally. One other must was the capability to be programmable by any other technical school in the world, this was accomplished by publishing an API with basic C++code and a tutorial. Of the could haves none were fully accomplished, we did build the physical return system but the vents that direct the air are not implemented. As for the non-functional requirements there is a document online on how to build the AHU mock up, a document explains how every sensor and the API are implemented and there is a project with example control software in combination with the web page.

Concluding this only one requirements that is a must didn't quite make it. The reason for that is that the case was at a very late stage of the project, so there was not enough time to test the code with the hardware. The cooling elements are connected electrically, but since there is no way at this time to get rid of the heat generated by the cooling element we can't operate it with its intended purpose. The could haves are not implemented just because there wasn't enough time for it. We did look into the air flow meters for example, but realizing that it had a lower priority it was left aside and we continued with the subjects that were of high priority. However we did design the case to be modified in the future to accompany these requirements, like the tunnels for return airflow.

Chapter 6

Evaluation and reflection

In the beginning of the project, we decided to use Scrum as our methodology. After a few weeks, however, we noticed that Scrum might not have been the best methodology for the first parts of the project. What we actually needed was more of a waterfall-like method, with a research phase, a design phase, and a build phase. This would have enabled us to get a functional prototype up and running within the first six to eight weeks of the project. After that, we could have switched to Scrum to iterate on the project and improve it in small steps.

However, by the time we figured this out, we were already close to done with building the prototype, so we decided not to switch away from Scrum, because we would have needed to switch back only a few weeks later. This means that the first three or four sprints were not actually sprints, because they did not have a deliverable product at the end of the sprint.

Another problem we had was with the speed of the project. We regularly had delays, mainly in the process of ordering components. These delays were partly caused by ourselves, and partly by the bureaucracy of the ordering process. Some of the delays were caused by the web shops where we ordered. All of these delays added up, which means that in the end, some of the components were only delivered a few days before our demonstration. Due to the fact that we had already implemented almost all of the API, we were able to get most of the prototype working for the demonstration, but we feel that we cut it a little too close.

References

Alternate Nederland. (2017). *Aerocool Shark Fan 12cm Evil Black Edition, Case fan.* Retrieved from https://www.alternate.nl/Aerocool/Shark-Fan-12cm-Evil-Black-Edition-Case-fan-/html/product/255010?lk=9301

Arduino. (2017). *Arduino Mega.* Retrieved from https://www.arduino.cc/en/Main/arduinoBoardMega

Atmel Corporation. (2015, 11). 8-bit microcontroller (Computer software manual No. ATmega48A/-PA/-88A/-PA/-168A/-PA/-328/-P). Retrieved from http://www.atmel.com/images/Atmel-8271-8-bit-AVR-Microcontroller-ATmega48A-48PA-88A-88PA-168A-168PA-328-328P_datasheet_Complete.pdf (Rev. 8271J)

Blok, S. A. (2015, 6). *Regeling van de minister voor wonen en rijksdienst van 18 juni 2015, nr. 2015-0000335240, houdende wijziging van de regeling bouwbesluit 2012 met betrekking tot de eisen aan kooldioxidemeters en het aanwijzen van normen.* Retrieved from https://zoek.officielebekendmakingen.nl/stcrt-2015-17338.html

Conrad Electronic International GmbH & Co. KG. (2003, 11 20). *PTC Heat Element (Beehive type). DC 12V Ceramic element.* Retrieved from http://www.produktinfo.conrad.com/datenblaetter/525000-549999/532894-da-01-en-Keramik_Heizelement_12V_150W.pdf

Conrad Electronic International GmbH & Co. KG. (2017a). *Heating element 12 Vdc 150 W.* Retrieved from http://www.conrad.com/ce/en/product/532894/Heating-element----12-Vdc----150-W----L-x-W-x-H-90-x-27-x-17-mm

Conrad Electronic International GmbH & Co. KG. (2017b). *Luchtvergrendelingsklep met afdichtrubber Wallair.* Retrieved from https://www.conrad.nl/nl/luchtvergrendelingsklep-met-afdichtrubber-wallair-n35987-geschikt-voor-buisdiameter-15-cm-verzinkt-561553.html

Conrad Electronic International GmbH & Co. KG. (2017c). *Modelcraft Miniservo MC1811 Analoge servo.* Retrieved from https://www.conrad.nl/nl/modelcraft-mini-servo-mc1811-analoge-servo-materiaal-aandrijving-kunststof-stekkersysteem-jr-275460.html

Conrad Electronic International GmbH & Co. KG. (2017d). *Resistance wire* 5.65 Ω m. Retrieved from http://www.conrad.com/ce/en/product/421201/Resistance-wire----------565-m

Electricfor, S.A. (2017). *Air heaters - Electricfor - Electric heating elements.* Retrieved from http://www.electricfor.com/en/333325/Air-heaters.htm

Fairchild Semiconductor Corporation. (2013, 11 4). N-Channel QFET MOSFET (Computer software manual No. FQP30N06L). Retrieved from http://www.farnell.com/datasheets/2299856.pdf (Rev. C1)

Goodliffe, M. (2013, 8). *R2D2 - Servo Driven Hinge Design.* Retrieved from https://www.youtube.com/watch?v=e58yMo2MXdY

Hanno. (2016, 6). *AutoFan - Automated Control of Air Flow.* Retrieved from https://hackaday.io/project/12384-autofan-automated-control-of-air-flow/log/40672-the-mechanical-prototype

KNMI. (2011a). *Langjarig gemiddelde 1981-2010, gemiddelde maximumtemperatuur, augustus.* Retrieved from http://www.klimaatatlas.nl/kaart/temperatuur/tx_aug_8110.png

KNMI. (2011b). *Langjarig gemiddelde 1981-2010, gemiddelde minimumtemperatuur, februari.* Retrieved from http://www.klimaatatlas.nl/kaart/temperatuur/tn_feb_8110.png

KNMI. (2011c). *Langjarig gemiddelde 1981-2010, gemiddelde relatieve vochtigheid om 12 UT per seizoen, zomer.* Retrieved from http://www.klimaatatlas.nl/kaart/vocht/rv12_8110_zom.png

Measurement Specialties, Inc. (2013, 10). Digital Relative Humidity sensor with Temperature output (Computer software manual No. HTU21D(F)). Retrieved from https://cdn-shop.adafruit.com/datasheets/1899_HTU21D.pdf (Rev. 3)

Raspberry Pi Foundation. (2017a). *Download Raspbian for Raspberry Pi.* Retrieved from https://www.raspberrypi.org/downloads/raspbian/

Raspberry Pi Foundation. (2017b). *Raspberry pi - teach, learn, and make with raspberry pi.* Retrieved from https://www.raspberrypi.org/

Scrum Alliance. (2017). *What is scrum? an agile framework for completing complex projects.* Retrieved from https://www.scrumalliance.org/why-scrum

Semiconductor Components Industries, LLC. (2014, 11). Plastic medium-power complementary silicon transistors (Computer software manual Nos. TIP120, TIP121, TIP122 (NPN); TIP125, TIP126, TIP127 (PNP)). Retrieved from https://www.onsemi.com/pub/Collateral/TIP120-D.PDF (Rev. 9)

SENSIRION AG. (2015, 5). Humidity and Temperature Sensor (Computer software manual No. SHT3x-DIS). Retrieved from https://cdn-shop.adafruit.com/product-files/2857/Sensirion_Humidity_SHT3x_Datasheet_digital-767294.pdf (Version 0.93)

Silicon Laboratories Inc. (2016, 8). I2C Humidity and Temperature Sensor (Computer software manual No. Si7021-A20). Retrieved from https://www.silabs.com/documents/public/data-sheets/Si7021-A20.pdf (Rev. 1.2)

Sinovoip. (2017). *Banana Pi - BPI Single Board Computers Official Website.* Retrieved from http://www.banana-pi.org/

Software Freedom Conservancy, Inc. (2017). *Git.* Retrieved from https://git-scm.com/

Software in the Public Interest, Inc. (2017). *Debian – The Universal Operating System.* Retrieved from https://www.debian.org/

Texas Instruments Inc. (2016a, 8). Low Power, High Accuracy Digital Humidity Sensor with Temperature Sensor (Computer software manual No. HDC1010). Retrieved from http://www.ti.com/lit/ds/snas685a/snas685a.pdf (Rev. A)

Texas Instruments Inc. (2016b, 11). Low-Voltage 8-Channel I2C Switch with Reset (Computer software manual No. TCA9548A). Retrieved from http://www.ti.com/lit/ds/symlink/tca9548a.pdf (Rev. F)

The Engineering Toolbox. (2017). *Mollier diagram.* Retrieved from https://www.engineeringtoolbox.com/psychrometric-chart-mollier-d_27.html

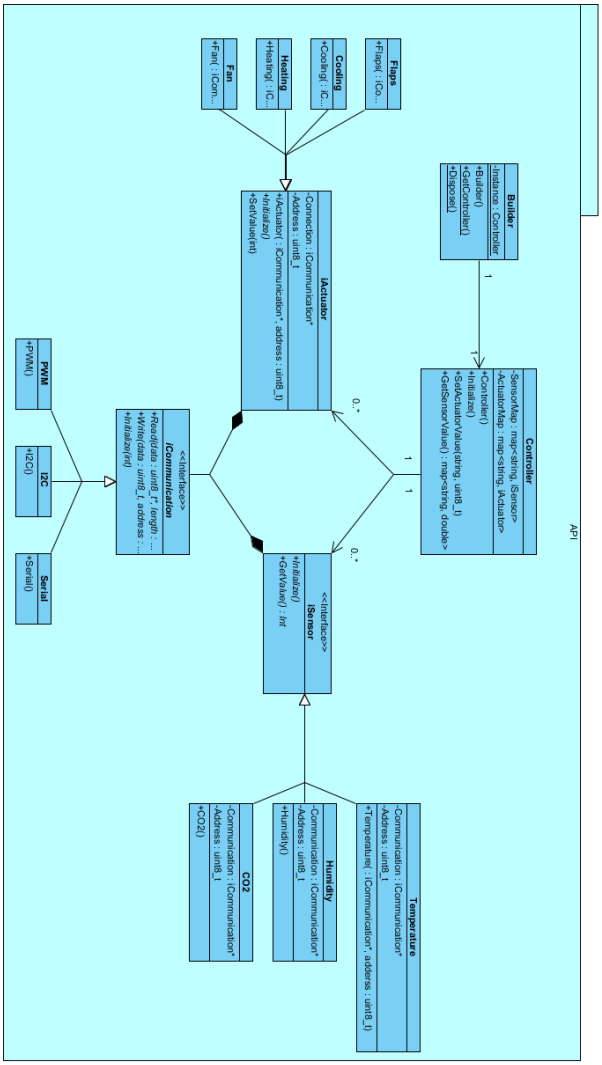
Ventilatieshop Ede. (2017). *Aluminium afsluitbaar schuifrooster opbouw.* Retrieved from https://www.ventilatieshop.com/aluminium-afsluitbaar-schuifrooster-opbouw-155-x-155mm-alu-3-1616aa/

Zhengzhou Winsen Electronics Technology Co., Ltd. (2017a). *MH-Z14A NDIR CO2 Sensor for Carbon Dioxide Detection.* Retrieved from http://www.winsen-sensor.com/products/ndir-co2-sensor/mh-z14a.html

Zhengzhou Winsen Electronics Technology Co., Ltd. (2017b). *MH-Z19B NDIR CO2 Sensor.* Retrieved from http://www.winsen-sensor.com/products/ndir-co2-sensor/mh-z19.html

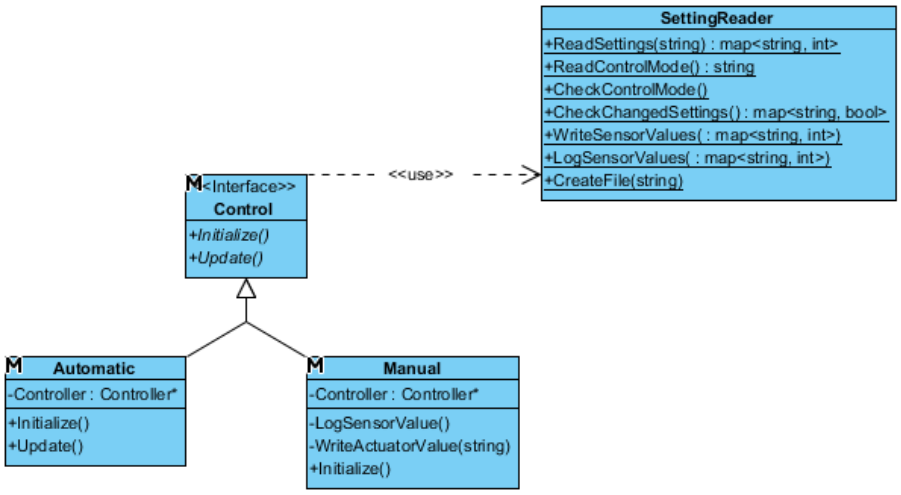
Appendix A

API Design



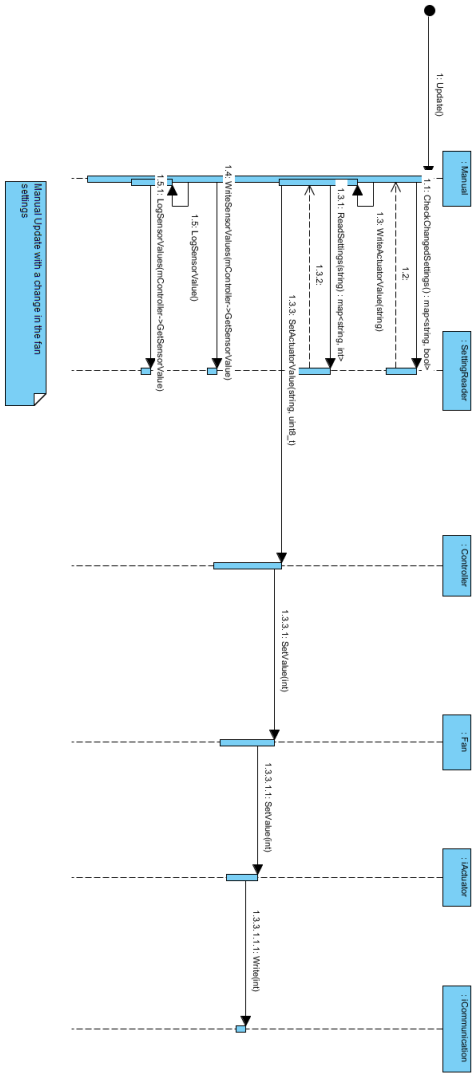
Appendix B

Control Software



Appendix C

Manual Control



Appendix D

Plan of Approach

Please see the file “pva.pdf”

Appendix E

API Code tutorial

Please see the file “AHU API manual.pdf”

Appendix F

Case tutorial

Please see the URL `http://www.instructables.com/id/Project-AHU/`

Appendix G

API Code documentation

All of the API code has been documented using Doxygen. For the resulting document, please see the file "refman.pdf", or the URL `http://ahu.rchard2scout.nl/` (also available on Archive.org: `https://web.archive.org/web/20171105001731/http://ahu.rchard2scout.nl/`).