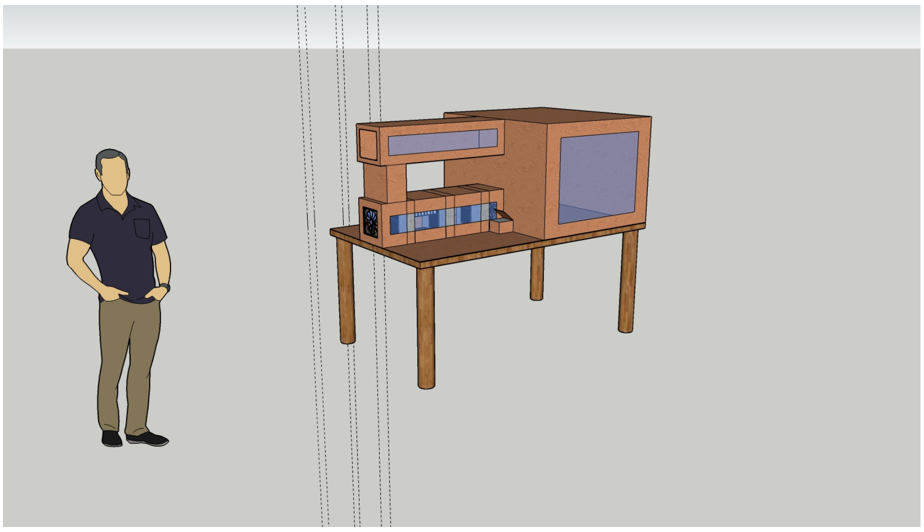
Air Handling Unit

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

Final report – 21 June 2017
Second revised version – November 4, 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 $50 \times 50 \times 50$ 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 , 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 cm^3/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 °C (KNMI, 2011b), and the highest average maximum temperature in August is 24.0 °C (KNMI, 2011a). To take some margins into account, we assumed an intake air temperature between -10 °C and 30 °C four our calculations. We also assumed a desired room temperature between 18 °C and 25 °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 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 using 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.

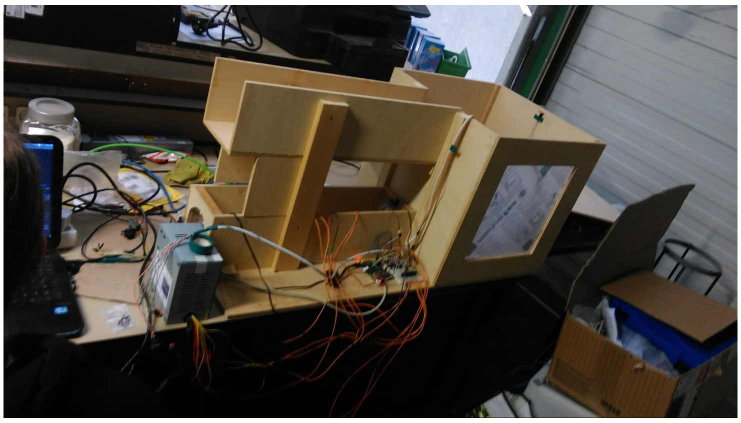


Figure 1: A picture of the case in-progress, with the Raspberry Pi installed.

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 $-10\,°\mathrm{C}$, and the desired temperature of the air in the chamber is $25\,°\mathrm{C}$. 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 $\mathrm{cm^3/s}$. The density of air is 1.225 $\mathrm{kg/m^3}$, 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 $\mathrm{m^3/h}$, we get 38.972 $\mathrm{cm^3/s}$, 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.

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.

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 in Figure 5. The advantage is that we can choose the length of the resistance

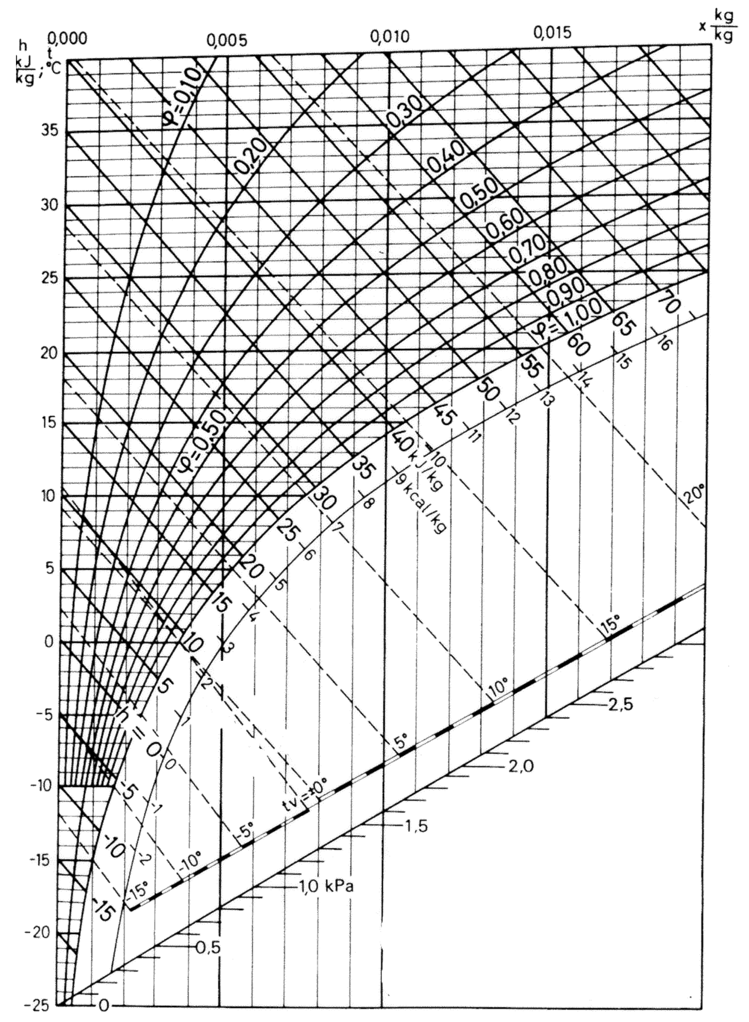


Figure 2: Mollier diagram. (The Engineering Toolbox, 2017)

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.

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

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

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 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 in Listing **??**). 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();
```

Figure 4: Tube-shaped heating elements. (Conrad Electronic International GmbH & Co. KG, 2017a)

```
  virtual bool Initialize(uint8_t* pins, size_t size);
  virtual bool Write(uint8_t data, uint8_t address);
  virtual bool Read(uint8_t* buffer, uint8_t length);
     //empty
};
```

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.

```
bool PWM::Initialize(uint8_t* pins, size_t size){
  for(uint8_t i = 0; i<size; i++){
    if(softPwmCreate(pins[i], 0, 100)){
      return false;
    }
  }
  return true;
}
```

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.

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

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

Figure 5: Tube-shaped heating elements. (Conrad Electronic International GmbH & Co. KG, 2017b)

elements is between 0 and 30, because that ensures the transistors do not get too hot.

```
void HeatingElement::SetValue(uint8_t value)
{
  if (value < 0)
    value = 0;
  else if (value > 30)
    value = 30;
  iActuator::SetValue(value);
}
```

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

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

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

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 × 40 × 3.4 mm	40 × 40 × 3.6 mm
Price	€43,54	€61,01

Table 2: Comparison of materials

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.

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 wasnt 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 $-55\,^{\circ}\mathrm{C}$ to $125\,^{\circ}\mathrm{C}$. This was more than enough looking at the functional requirements in Chapter 2, where it is stated that we will be measuring temperatures from $-10\,^{\circ}\mathrm{C}$ to $30\,^{\circ}\mathrm{C}$. 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.

When we chose to go with the combined sensor which includes both the

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

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;
```

```
  return humidity;
}
```

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()`.

```
class I2C : public iCommunication
{
public:
  I2C();
  virtual ~I2C();
  bool Write(uint8_t data,uint8_t address); //return true on succes
  bool Read(uint8_t* buffer, uint8_t length);
  bool Initialize();
private:
  bool _setSlave(uint8_t address); //return true on succes
  bool _write(uint8_t address, uint8_t data); //return true on succes
  bool _writeByte(uint8_t data);
  int filedescriptor;
  uint8_t currentTarget;
};
```

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;
```

```
  }
  return true;
}
```

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

3.5.4 Recommendations

3.6 CO2 sensors

3.6.1 Hardware

3.6.2 Software

3.6.3 Problems

3.6.4 Recommendations

3.7 Fans

3.7.1 Hardware

3.7.2 Software

3.7.3 Problems

3.7.4 Recommendations

3.8 Vents

3.8.1 Hardware

3.8.2 Software

3.8.3 Problems

3.8.4 Recommendations

3.9 Power supply

3.9.1 Hardware

3.9.2 Software

3.9.3 Problems

3.9.4 Recommendations

Chapter 4

Software

4.1 API

4.2 Control software

4.3 Webdesign

Chapter 5

Conclusion

Chapter 6

Evaluation and reflection

Chapter 7

Bibliography

References

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)

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). *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)

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

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)

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

Appendix A

API Design

Appendix B

Control Software

Appendix C

Manual Control

Appendix D

Plan of Approach

Appendix E

API Code tutorial

Appendix F

Case tutorial

Appendix G

API Code documentation