KARELIA UNIVERSITY OF APPLIED SCIENCES
Degree Programme in Electrical engineering

Jere Teittinen

Development of an Open Home Garden Automation System

Thesis
August 2013

Karelia UNIVERSITY OF APPLIED SCIENCES	**THESIS** **September 2013** **Degree Programme in Electrical engineering** Karjalankatu 3 FIN 80200 JOENSUU FINLAND

Author
Jere Teittinen

Title

Development of an Open Home Garden Automation System

Abstract

The main purpose of this thesis was to develop an open home garden automation system which lets the user control things like lights and water pumps, and log sensor data in their garden. The idea came from personal interest; the author grows chili plants at home and the idea for this project slowly started to form after building a hydroponics system. It was soon realised that amateur solutions that do garden automation do not really exist, so to automate the plant growing process required new tools to be written from the very beginning by myself.

The development process was long but fruitful. At the beginning of the project the author faced many aspects that were not familiar to him, but eventually the pieces started to fall into place. The end result is a rather stable and functional automation system that is capable of handling the basic needs of a home garden.

This document will also look in the future, because the current state of the project is by no means the end - rather it is only the beginning. Many improvements and new features have already been planned and implementing them all will easily take months, if not over a year; so the prospects for further development have a sound base.

Language English	Pages 52

home garden automation, open source, free, dynamic, Raspberry Pi, Arduino

Karelia UNIVERSITY OF APPLIED SCIENCES	**OPINNÄYTETYÖ** **Syyskuu 2013** **Sähkötekniikan koulutusohjelma** Karjalankatu 3 80200 JOENSUU

Tekijä
Jere Teittinen

Nimeke

Avoimen kodin kasvihuoneautomaatiojärjestelmän kehittäminen

Tiivistelmä

Tämän opinnäytetyön tarkoitus oli kehittää avoin kodin kasvihuoneautomaatiojärjestelmä joka mahdollistaa muun muassa kasvihuoneen valaistuksen ja vesipumppujen ohjauksen, ja sensoridatan loggaamisen. Alku projektille tuli henkilökohtaisesta mielenkiinnosta; opinnäytetyön tekijä kasvattaa chilejä kotonaan ja idea projektille alkoi pikku hiljaa muotoutumaan kun kasvihuoneen chilejä varten rakennettiin vesikastelujärjestelmä. Pian tajuttiin ettei sopivia olemassa olevia kasvihuoneen automaatioratkaisuja juuri ole olemassa, joten kasvatusprosessin automatisointiin tarvittavat työkalut piti kehittää alusta lähtien itse.

Kehitysprojekti oli pitkä, mutta tuloksellinen. Projektin alussa opinnäytetyön tekijä kohtasi paljon asioita jotka eivät olleet hänelle tuttuja, mutta lopulta projektin eri osat alkoivat loksahtamaan yhteen. Lopputulos on melko vakaa ja toimintakuntoinen automaatiojärjestelmä joka pystyy huolehtimaan kodin kasvihuoneen perustarpeista.

Tässä dokumentissa katsotaan myös tulevaisuuteen, sillä projektin nykytaso ei missään nimessä tarkoita tämän projektin loppua - se on enemmänkin vain alku. Monia parannuksia ja uusia ominaisuuksia on jo suunniteltu ja niiden toteuttaminen tulee viemään kuukausia, ellei jopa yli vuotta; tämän projektin tulevaisuuden näkymillä on siis vakaa pohja.

Kieli Englanti	Sivuja 52

Kotiautomaatio, kasvihuoneautomaatio, avoin lähdekoodi, dynaaminen, Raspberry Pi, Arduino

Table of Contents

ABBREVATIONS

API	Application programming interface, specifies how some software components should interact with each other
AVR	8-bit RISC microcontroller architecture by Atmel
CPU	Central processing unit, processor of a computer
CSS	Cascading Style Sheets, a style sheet language for describing the presentation semantics of a document written in a markup language
GPIO	General-purpose input/output, a generic, programmable pin on a chip
GPLv3	GNU General Public License version 3
GSM	Global System for Mobile
HDMI	High-Definition Multimedia Interface
HTTP	Hypertext Transfer Protocol Secure, an encrypted version of the HTTP
IP	Ingress Protection rating, describes the safety of a device against dust and water.
IP	Internet Protocol
I^{2}C	Inter-Integrated Circuit
JSON	JavaScript Object Notation
LAN	Local area network
PC	Personal computer
PWM	Pulse-width modulation
RISC	Reduced instruction set computing
SD	Secure Digital, a non-volatile memory card
TLS	Transport Layer Security, a cryptographic protocol used by HTTPS
USB	Universal Serial Bus

1 Preface

The premise for this thesis was to design and implement a home garden automation system that was robust, modifiable and easy to configure; and potentially very inexpensive. The initial idea for this project arose from a personal need, but the purpose was further expanded to provide other people an open software platform which they could easily adapt in their own systems. This thesis will use a software and hardware configuration of my choice to demonstrate the features and capabilities of this system, but by no means is it intended to be limited to work only on the kind of hardware this thesis project was produced with. Currently there are certain hardware limitations for the software, but with the internal architectural design it is possible to adapt the software for different kinds of hardware in the future.

This work will explain what kind of features were designed to be implemented in the software system and compare the design to existing amateur and professional solutions. It will also be explained what did I learned from those existing solutions and how can those experiences can help with this project. In the end there will be a look into how this home garden automation system could be commercialised.

The subject for this thesis was chosen partially because albeit there are some similar open source projects, they do not tend to be easily adaptable to different hardware configurations; they are designed with one purpose in mind. While it is not possible to write one library for different kinds of microcontrollers, there are still things that can be done to achieve a little better adaptability. Those methods will be discussed in this thesis.

The state of the project discussed here is by no means final, it is merely built to a point where it could potentially be used in real life circumstances.. This project is intended to be developed further long after this thesis paper has been reviewed.

2 Home garden automation

2.1 The concept

It is perhaps easier to explain what home garden automation is if it is compared to typical home automation which people are more familiar with, because as a concept they are similar. Both systems share similar traits like controlling devices like lights with relays, controlling thermostats and taking readings from sensors.

However there are some notable differences. Home automation usually comes with a security system and some energy saving functionality, like automatically adjusting a home's temperature when its inhabitants are away. Garden automation in turn concentrates on making plants' growing conditions ideal and in this way it resembles more process automation where a certain process is automated in a such manner that it yields the most optimal outcome.

It is also common for the systems to store data from the sensors which can be used to display graphs, showing things like humidity or temperature in the garden at certain times. This helps the human user to keep track of what is going on in the garden and also to find out about possible problems.

To sum things up: the concept of home garden automation relies on reading sensor data and controlling different devices like pumps and lights with the goal of producing optimal growth conditions to plants. It is also common to have some kind of a graphical interface that displays data about the growing conditions in the system.

2.2 Existing amateur solutions

When developing something new it is always good to study existing solutions because one can often learn from them about things that work well, or perhaps more importantly one can also learn about things that do not work well.

Many similar amateur projects were studied, and they all seemed to have a common problem. They were designed to work in just one user case and were not configurable without making changes to the source code. This kind of approach poses a number of problems which makes it difficult for them to be adapted widely. The systems are not flexible since the users can not change their parameters during runtime, and even adding new things requires programming knowledge which shrinks down the potential user base. Also the list of features is rarely extensive. One other thing to notice is that many projects that claim they perform automation actually do not. For example many projects found through Hack a Day, which is a website that writes about amateur hacks, say they perform for example lights automation, when in fact they just contain an interface to remotely control the lights. No automation logic is built in.

But there are some good things in some of the systems. Often one can see beautiful user interface designs with one purpose projects because the emphasis on these projects is not on the background logic. One project has a really helpful website which helps the user in setting up their system as the website guides them through every step. While this feature does not dictate what the system itself can do, it works as an important reminder of how important it also is to make the instructions clear for the users because without them using or even installing the automation system can be difficult; especially if the end user is not a technologically capable person.

Next, some use case studies made for this project are presented.

2.2.1 GardenBot

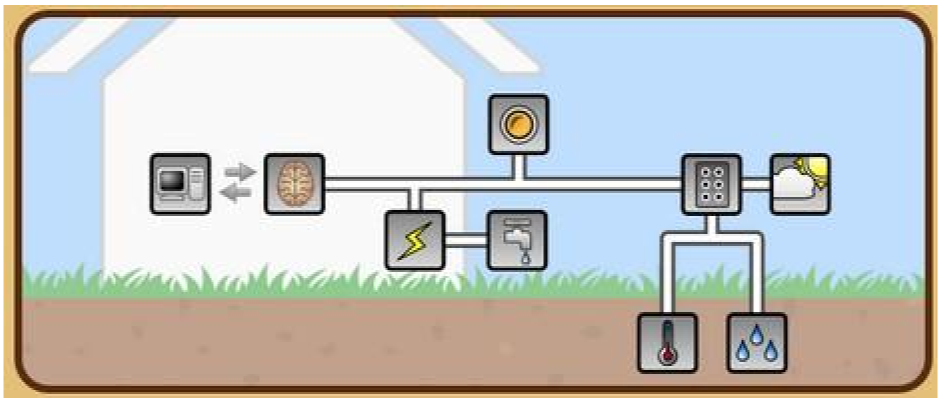


Figure 1 GardenBot structure (Copyright: Andrew Frueh)

"GardenBot is an open source garden monitoring system." This project is perhaps the closest equivalent to what this thesis project is about. The GardenBot site further states that "the ultimate goal of the GardenBot project is to be a complete garden monitoring and automation system". [1. What is GardenBot?] This project shares similar goals, however the completion of the GardenBot project is not that far yet that it could be called a home garden automation system because it does not really have built-in automation feature. It can handle watering, but not automatically. The user will still need to remotely push a button to start the watering routine.

It is also rather difficult for the user to add new sensors to the system. They need to tweak code in three separate places to get everything working which also means recompiling the project. This approach excludes most potential users who know nothing about programming. In an user friendly garden automation system the user should be allowed to dynamically add and remove sensors and devices during runtime from a graphical interface. This is one of the goals of this thesis project.

Even with its shortcomings, this project does have potential. Which is probably why Wired and SparkFun published an article about it. This project is also the most popular amateur garden monitoring and automation project that could be found.

The best part of the project is the website, which is clear and contains lots of resources about how the system works and how to install it and get it running. This is something this thesis project can really learn from. It is difficult for projects to succeed if they are difficult to use, so writing down good resources like GardenBot has done helps users to adapt the system in their use.

The system also allegedly runs stably which is of high importance with a system of this type. Having malfunctions when the system is controlling some real life devices is not acceptable.

2.2.2 Arduino Greenhouse mrk 2

This project was demonstrated in a YouTube video and is not intended to be a project other people can easily use. It is a private project by a YouTube user by the username “instrumenttek”. The reason why it is discussed is because it does some things well. The starting point is that the person has automated his greenhouse using an Arduino Mega to control appliances such as an analog louver control, heating and watering. It also features few temperature and humidity sensors. [2. Arduino Greenhouse mrk 2.]

The system is controlled by the aforementioned Arduino and some parameters such as the louver level can be controlled through a switch interface on the door of the box where the Arduino resides. The build quality is high as everything is installed rather professionally. The core where all the connections are made and the Arduino resides is an electrical center which is a good thing since the system is installed outdoors, so it keeps the system dry even in bad weather conditions. It is also possible to monitor the sensor readings from a remote computer which is also a goal of this thesis project.

What can be learned from this project? Encourage good hardware design. While the users of this thesis project will need to make their own hardware connections, it is good to encourage them to make them properly. It enhances safety and makes the system more durable.

2.2.3 Other projects

It was difficult to find similar projects that were worth writing about because most of them seemed to be narrow in their goals so the usage possibilities were not numerous. For example there was an article on Hack a Day about a lights control automation project which let the user control a Cloud Lamp by using a Raspberry Pi and a smartphone. [3. Web based automation courtesy of Raspberry Pi.] While it let drive the lamp with different settings, it did not really automate it much. It was not possible to schedule when to put the lights on and out, for example.

However the web based user interface was really well done and simple to use. This should be kept in mind when developing automation systems. Often the professional solutions feature a rather simple, functional interface. But in the days of modern Internet, a good interface design appeals to the end users.

Few grow box projects were also found through Hack a Day but they were designed for a certain use case which was helping the user to monitor and control their specific growing conditions.

This leads to the conclusion of this section. There are some amateur solutions that attempt to do some home garden monitoring and automation, but they do not attempt to be dynamic and naturally expandable, or general purpose. Natural here means they can not be configured during runtime. General purpose is important because the system needs to work in many use cases, not just in a very narrow selection. It felt strange to realise that there does not really exist a similar project as the one this thesis is about because the potential in this field is high.

2.3 Existing professional solutions

More features and a robust design can be expected from a professional solution, which often also means a steep cost, especially since most professional solutions are aimed at commercial gardens. There is a lot to learn from professional solutions because they tend to be well designed and packed with useful functionalities.

2.3.1 Telldus Technologies

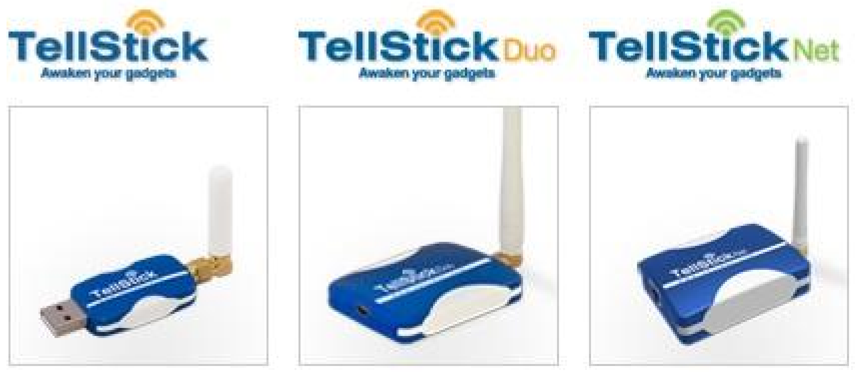


Figure 2 Various Telldus products (Copyright: Telldus)

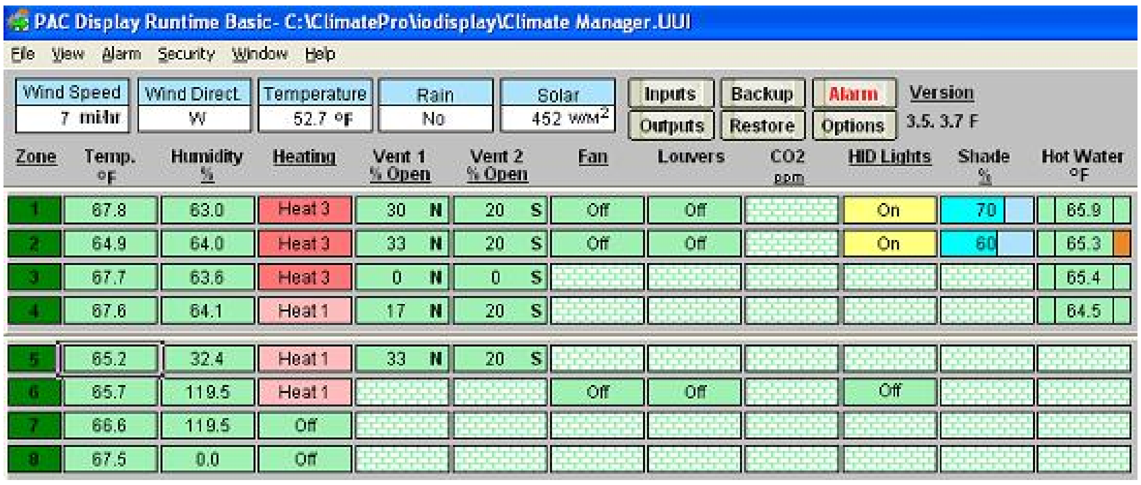
While not aimed at gardening, it is still well worth to analyse the Telldus platform which offers products for controlling home appliances wirelessly. The two main reasons why this company and its products were taken under scrutiny are their business model and the support of wireless devices. Both of these things are further discussed later on in this document in the main chapter **commercialisation**, but a brief glimpse is appropriate also in this chapter.

The business model Telldus practices could be suitable for a project like this. Their idea is to release almost all of their software under an open source license, but to complement that they sell hardware for which the software is designed. As the software for this project is open source as well, selling hardware seems like the most probable business model if any kind of commercialisation will ever be practiced.

Support of wireless devices in Telldus' case means supporting multiple 433 MHz using devices, such as Nexa's product line. Telldus itself does not produce these products that directly interact with hardware, they merely offer the connection layer between the devices and a software control interface. Since companies and Nexa already produce these wireless devices, it makes sense to develop support for their protocol. It is a prospect that this project should aim to fulfill in the future because it makes the usage of this system under discussion easier because no wiring needs to be done.

The biggest downside for their products is that they do not offer any kind of automation functionality. The user can remotely toggle devices and read sensor data, but there are no graphs or possibility to set timers so that the user can toggle devices on and off at certain times. This is the main reason why their products do not currently suit home gardens' needs. However on their forums one of their developers has stated that bringing some automation functionalities to their product line is planned for the future.

2.3.2 Climate Control Systems Inc.


PAC Display Runtime Basic- C:\ClimatePro\iodisplay\Climate Manager.UUI
File View Alarm Security Window Help
Inputs
Backup
Alarm
Outputs
Restore
Options

Wind Speed	Wind Direct.	Temperature	Rain	Solar	Version
7 mi/hr	W	52.7 °F	No	452 W/M^2	3.5. 3.7 F

Zone	Temp. °F	Humidity %	Heating	Vent 1 % Open	Vent 2 % Open	Fan	Louvers	CO2 ppm	HID Lights	Shade %	Hot Water °F
1	67.8	63.0	Heat 3	30 N	20 S	Off	Off		On	70	65.9
2	64.9	64.0	Heat 3	33 N	20 S	Off	Off		On	60	65.3
3	67.7	63.6	Heat 3	0 N	0 S						65.4
4	67.6	64.1	Heat 1	17 N	20 S						64.5
5	65.2	32.4	Heat 1	33 N	20 S						
6	65.7	119.5	Heat 1			Off	Off		Off		
7	66.6	119.5	Off								
8	67.5	0.0	Off								

Figure 3 Climate Manager interface (Copyright: Climate Control Systems Inc.)

This company has been manufacturing greenhouse automation systems since 1985. Their product line consist of solutions for garden climate and fertigation management. These products are holistic solutions consisting of both hardware and software. [4. Greenhouse Automation: Climate Control Systems Inc.]

The company's products are clearly aimed at professional greenhouses which can be seen in the quality and caliber of the hardware they sell. The different products are designed to be ready bundles which can not be altered - the consumer gets what they pay for and that is all they need to think about.

Business model for this company seems to be a working one. They sell quality products which are capable of handling most of the tasks in a greenhouse and also provide feedback about the growth conditions.

User interface for their product line looks like what could be expected from an industry grade solution. It is very simple, but functional; there are no developed graphics to appease the user's eyes. This is of course perfectly normal for an industrial application, but the user interface and the price point are out of reach for the average home plant grower.

The thing that this project can pick up from this company's products is a functional user interface design where the data is clearly displayed.

3 Design approach

3.1 The idea

This project's fundamental idea is to create the software for a potentially very inexpensive open source home garden automation system. I have elected to release the source code as open source because I believe it can benefit those numerous individuals who are growing plants at their homes. Proprietary solutions are often expensive and costs run easily high if home gardens are starting to get complicated, so an open source system like this can help people to keep their budgets down. I am also very fond of the open source philosophy and with the choices I've made, want to express the people that you can do great things by using open architectures. Also, I want to encourage people with similar ideas to take mine as their baseline and further develop it to serve their own needs. Perhaps in turn they can do more to benefit other people as well.

In order to make an open solution usable and adaptable, it first needs to be simple and easy to use. One of the goals was to design the software in a way that the end user would not need to know anything about programming. All they would need to do is to define things like password, LAN IP and port for the web interface and the rest could be handled from there.

The web interface would allow the user to add, remove and modify devices which can then be controlled and read through it. Sensors would automatically log data and the system would draw graphs about the readings without the user having to do anything. Additionally the user could define schedules for different tasks like running pumps at predefined times.

3.2 Requirements for a good garden automation system

First and foremost safety needs to be taken as the basis for every function that the system performs. It is not acceptable to develop a system that is prone to fail, and by doing so it might cause some permanent damage. Imagine, for example, if the pumps watering the plants in an ebb system got stuck on and they would overheat because most of the water is pumped out of the water reservoir. One thing that can be done to improve this kind of safety control is to use a timer inside the microcontroller which keeps track of the time that a certain pump has been running, automatically toggling it off after the threshold time has been achieved.

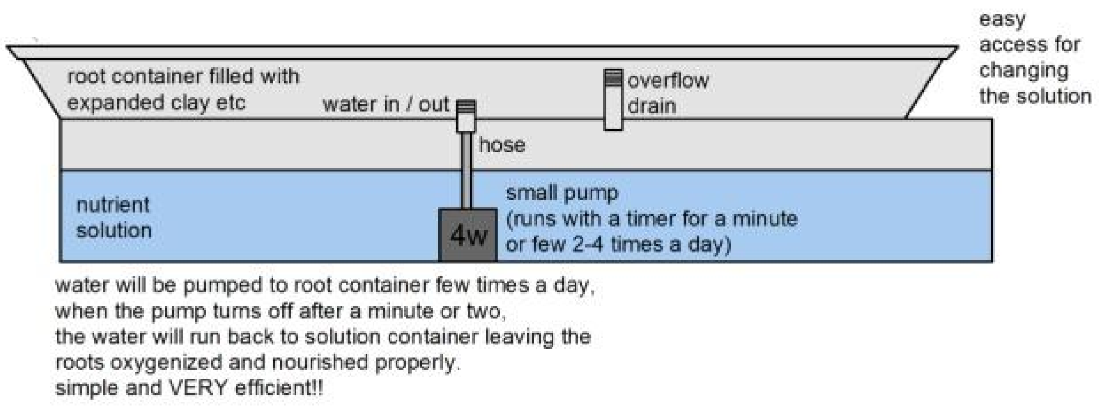

easy access for changing the solution
root container filled with expanded clay etc
water in / out
overflow drain
hose
nutrient solution
4w
small pump (runs with a timer for a minute or few 2-4 times a day)
water will be pumped to root container few times a day, when the pump turns off after a minute or two, the water will run back to solution container leaving the roots oxygenized and nourished properly. simple and VERY efficient!!

Figure 4 Principle how the ebb works. (Copyright: Jukka Kilpinen)

Usually a good system is also easy to use. By designing a system that the user understands makes it easier for them to actually perform operations using the system. It will allow them to use more of the potential the system offers. For example make it easy for the user to define that a certain sensor's certain reading means that a certain relay needs to be toggled on and soon they will not need to worry whether the greenhouse vent is open or closed - the system will take care of it and it will also keep the plant growth conditions more optimal.

The system also needs to offer an array of functionality that is relevant to the intended purpose. It would not make sense for this kind of system to store GPS data for example. Instead the focus should be concentrated on developing tools that allow controlling the growing conditions of the plants easily. These tools make it easy to toggle devices or tell the system when to perform those operations on its own, and they make the data gathered by sensors easily available for the user so the user can see immediately how the plants are doing - do they need special attention because the temperature is too high, or are they doing well.

3.3 How to make it inexpensive

The software itself is free but it can not function without hardware devices, which can cost a lot. To make it inexpensive to use this software, I've designed a reference open hardware configuration which people can use. The details will be explained later on in their respective chapters, but in this section the basic components are introduced to give the reader some kind of understanding of the principles that make the system inexpensive.

The microcontroller in this project is chosen to be Arduino which is well known and extremely easy to use even for a layman. However in future it should be possible to cut down cost even further by writing a version of the microcontroller library that does not rely on Arduino's own library to function. This way the users could build the system with only the parts they need.

Sensors chosen for this initial version are rather basic and commonly used, such as the DHT11 temperature and humidity sensor or the more accurate DS18B20 temperature sensor. They can be purchased as fully functional modules on eBay with mere 3-5 € a piece which can greatly reduce the cost of the build due the possible high number of used sensors.

The Arduino itself will not contain logic for scheduling tasks or logging sensor data, so the system needs some kind of computer server to control it. While the users can elect to run their system on any hardware that can run Python 3, can the cost just for the electricity can climb high. This problem can be nursed by using Raspberry Pi for the purpose. The board itself costs only around 33 € if it is bought straight from one of the official suppliers. It is a full blown computer which draws only few watts of energy, so running it as a garden automation server is very cheap in the long run.

4 The hardware

The hardware is designed to work in two layers, of which the lower level, microcontroller, layer can be expanded with additional microcontroller devices. The higher level layer is a PC running the server which handles the runtime logic, such as timing tasks and issuing commands to the lower layer. The lower microcontroller layer is not as powerful as a PC is, but contains peripherals that allow it to be connected to several low level hardware products, such as sensors or relays.

It does not matter which PC the higher layer device is as pretty much any computer capable of running a modern Linux distribution can also run Python as well. For example, the aforementioned Raspberry Pi is an excellent example of an inexpensive PC which can easily handle the task of running this system.

For the low level layer, Arduino Mega and its corresponding microcontrollers are currently supported. The two biggest limiting factors for microcontroller support are the amount of memory on the boards and the implementation of C++ that they support. For example an Arduino Uno does not contain enough memory to run the system and the microcontroller code is written for Arduino C/C++. However in the future it is possible to port the code to support other microcontrollers since the Arduino specific libraries are not used extensively.

4.1 Arduino

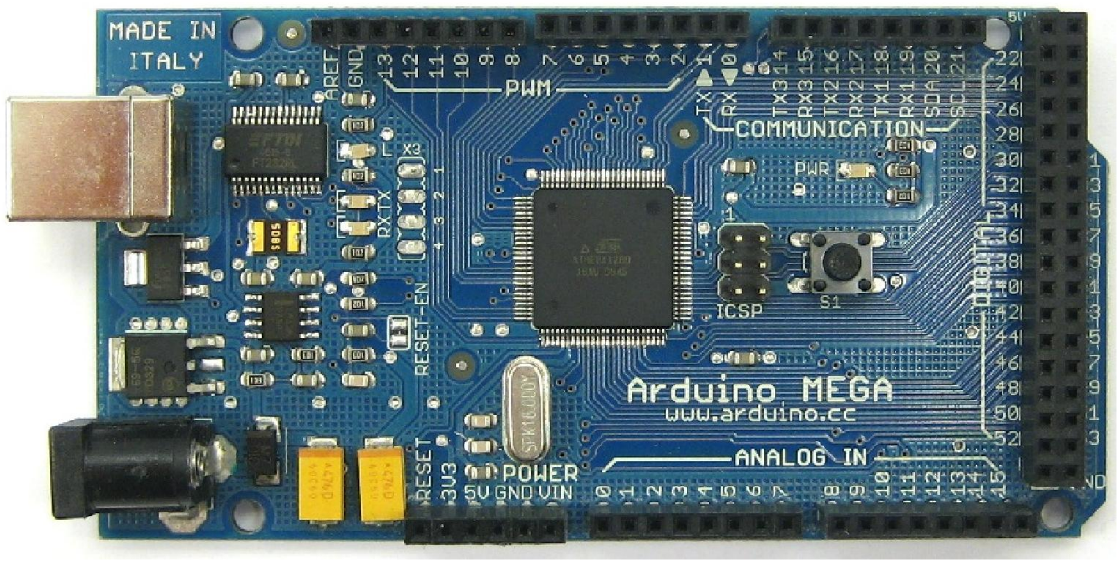


Figure 5 Arduino Mega (Figure: Wikimedia)

4.1.1 What is Arduino

Arduino is a popular, low-cost and open microcontroller. This means that anyone with the proper equipment may build their own Arduino without any licensing fees. [5. Arduino.]

The popularity of the board stems from its low cost and ease of use. Because the hardware is open and uses basic components, many third party manufacturers can create their own boards cheaply. The Arduino team has spent a lot of effort in making the software easy to use, so it is a fast and a simple way for do-it-yourself hobbyists to grab a board and express themselves with it. It is also proven to be durable so it can survive different difficult environments where one would not necessarily want to use expensive hardware. [6. Why Arduino Is a Hit With Hardware Hackers.]

Basic usage of the board usually contains interfacing with hardware pins. The board comes with numerous pins that serve different purposes. Some are digital inputs and outputs, some handle pulse-width-modulation and some read analog input. With these pins the user interfaces with the world. A pin can control the state of a relay, or read a value of a sensor attached to an analog input pin. The programming of the board happens by using Arduino's own language which is heavily based on Wiring's Processing language and implemented in AVR C/C++. [7. Arduino. Arduino/Processing Lanaguage Comparison.]

4.1.2 Modules

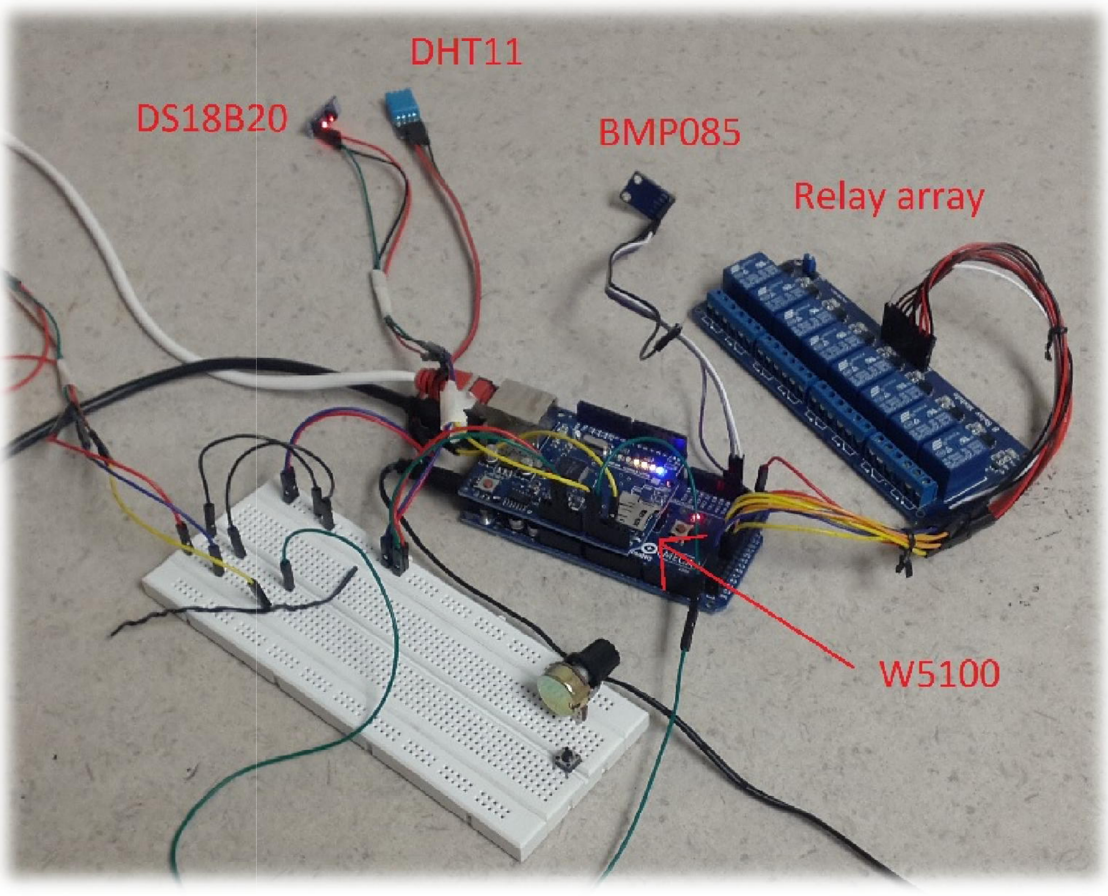


Figure 6 Prototype Arduino configuration with some sensors, a relay array and the Ethernet shield attached

One of the main factors why Arduino was chosen is the fact that support exists for almost any kind of hardware module, this including software libraries which let the user operate the additional hardware.

Hardware modules, which are often dubbed as shields, are parts that interface with Arduino physically, offering a specific functionality using the board's pins to relay communications. With hardware modules the functionality of Arduino can greatly be extended. Using a cheap Ethernet module one can turn the Arduino board into a simple web server with only few lines of code. Other possibilities exist, such as using the Arduino as a GSM device, or the user can attach the aforementioned DS18B20 temperature sensor to the Arduino and read temperatures using the board.

A large proportion of this all is cheap and open, so it is easily available for the user. With this kind of benefits Arduino makes it possible to develop a system that is accessible to almost anyone.

4.1.3 The board chosen for this project

Several versions of Arduino exist. The original developers of the board have since the launch of the original version developed different types of Arduino boards for different user cases. There exist Arduinos that are only the size of a small USB dongle, LilyPad versions are designed to be worn in clothing. In addition there's also the Mega family of boards for more heavy lifting. The differences in these boards along with their sizes are seen in different amounts of memory and ports where other devices can be connected to.

For this program the Arduino Mega version was selected as the board of use. The fundamental reason for the decision is simple - other boards simply do not have enough storage space and memory to hold the program binary that will handle Arduino logic. Furthermore the Mega board has the most ports so it allows the most flexibility considering features. The user has the possibility to attach more relays and more sensors to the board without the risk of running out of ports. Also, even with its relatively large size it can still be purchased for around 16 € which makes it very low-cost.

An Ethernet module was bundled together with the Arduino to allow the device to communicate with the outside world. Originally the cheaper option, ENC28J60 chip was chosen to provide this interface. However it featured an unprecedented disadvantage which made it impossible to use the chip. During testing it was quickly proven that the ENC28J60 module frequently crashed to the point it could not anymore function without a hard reset. These crashes happened often when it was not under much use, but almost a certain way to crash the module was to toggle a few relays connected to the Arduino's power supply and the module crashed. Eventually it got to the point that it could not even be initialized anymore. The conclusion was that the chip is extremely precise about the voltage supplied to it and even small fluctuations can harm it.

Instead money was invested to the more expensive W5100 module which is very popular amongst Arduino user and a library for running it comes with the official Arduino IDE. Still with the additional cost the module only costs around 7 € including shipping, so it is still highly inexpensive still. With the W5100 the problems with the Internet interface disappeared and it has been functioning reliably. The programming interface also is simpler than with the ENC28J60 so all in all it was the better option to use with this project.

4.1.4 Connectivity

The Arduino Mega and its relatives are often chosen because of their high pin number. In total there are over 50 pins which give the user a lot of ground to play with. Calculating a total number of sensors and devices that can be attached to one Arduino is not simple, however some estimates can be made. Nevertheless one board can easily house more sensors and devices than what an average user even needs.

In total there are 54 digital input/output pins and 16 analog inputs. 14 of the digital pins can be used as PWM (pulse-width modulation) pins to drive out 8-bit signal to imitate analog output. Four of the digital pins can be used for SPI (Serial Peripheral Interface) communications. For example the ENC28J60 Ethernet module uses these pins. 8 pins on the Mega board also support serial communications and 2 pins support the I^2C interface which addresses sensors and devices by their address. So if the sensors or devices the user uses support this interface, they can potentially connect almost countless devices in the same pin. [8. Arduino Mega.]

Given the number of different kinds of pins and all the possible hardware configurations the users might want to connect to the Arduino, a clear figure about how many sensors and devices can be connected can not be given, but the users can easily connect a few dozen sensors and another few dozen devices to the board with ease – if an external power supply is used since Arduino can only support about half an ampere of current to the peripherals.

4.2 Raspberry Pi

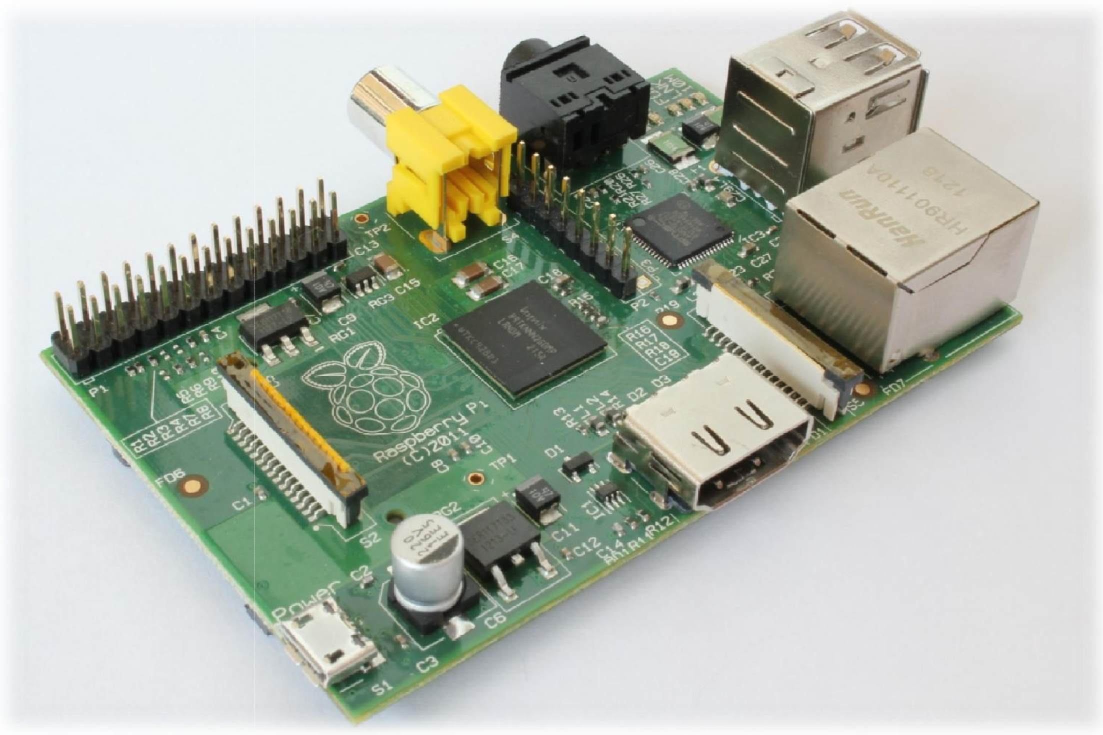


Figure 7 Raspberry Pi board (Figure: Wikimedia)

Similarly to Arduino, Raspberry Pi is an open credit-card-sized single-board computer. It comes with a 700 MHz ARM CPU, 512 MiB of memory and all the basic connectors needed by a computer to interface with the outer world, such as Ethernet, USB, HDMI and audio output; the developers have even packed in GPIO pins which makes it easy for the users to interface the board with microcontrollers or even sensors and relays. The Raspberry Pi offers the performance of a respectable computer and with the price point of only around 35 € it is also incredibly low-cost. It was originally developed in order to encourage young students to start learning computer sciences, but has quickly become a popular tool for hackers all around the world. [9. FAQs | Raspberry Pi.]

Although the board has to offer a rather respectable performance, it is not as speedy as the average desktop computer these days; but that is not Raspberry Pi's purpose. It has more than enough performance to run homebrew projects at a very low level of energy consumption, a point that is often an important factor for cheap projects.

The popularity, relative performance and low cost and power usage are one of the main reasons why the Raspberry Pi was chosen for this project alongside with the Arduino. With easily accessible hardware it is easier for the potential users of this system to actually adapt it in their own user cases.

5 The software

A good and robust software design is all in all important for a software project like this because it needs to be designed to be flexible and extensible. The reasoning behind this approach is that it makes future development easier. This means that in the development of different software components, a uniform interface design has been one of the areas that I have been heavily concentrating on.

The name for the software was chosen to be Naga Automation Suite, abbreviated as NAS. There's no real logic behind the name, it just sounds attractive. Naga comes from the extremely hot chili variety Naga Morich. The software is called an automation suite because with the name the author wants to emphasize that it is automation oriented.

5.1 Structure

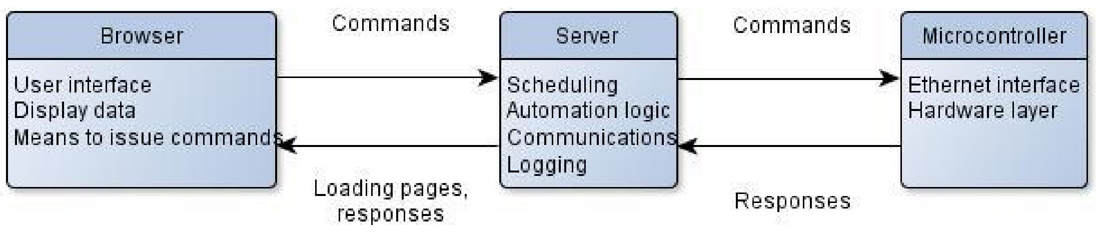


Figure 8 Multitier software structure

The software itself is divided in three layers following the multitier architecture principles: web user interface, server core and microcontroller client. In future a fourth layer for database is possible, but for now everything is stored in configuration files and stored in process memory during runtime.

"In software engineering, multi-tier architecture is a client-server architecture in which presentation, application processing and data management functions are logically separated". [10. Multitier Architecture.] This kind of software design defines certain ranges of logical responsibility between the different layers which makes the software simpler and easier to manage as certain kind of functionality, such as validating user input, is done in one place only. It makes sense to separate logic on different levels with this project since the software design is forced in any case to function on at least three hardware levels, the end terminal (browser), server (the main logical unit) and the microcontroller (interfacing with real life appliances).

5.2 Web user interface

Serving as the front end, the web user interface interacts graphically with the system's user. Its purpose is to offer a clear view to the system and let the user interface with lower levels of the application structure.

The reason behind why a web interface was chosen as the front end is because it is the most versatile and uniform front end package. Almost any device that could run any other type of front end for this project can also run a web browser. Having the interface in the web means that it will act the same way in any environment the user is using, like between Linux and Windows, or browsers like Chrome and Firefox. If the front end were to be made as a Windows application, it would then need to be rewritten for Linux which adds a large amount of work and makes the system more complicated.

5.2.1 Security

Normally security is an issue with web applications and an automation system meant for personal use is not an exception. Therefore one of the things that has been concentrated on is making the system secure.

One of the basic things that are pretty much required for this kind of system is some form of authentication. For this project normal password authentication was chosen. The user is prompted with a password, which needs to be validated in the server before the user can access the interface. It is important to handle the validation in the remote server, because web client based authentication is not secure. [11. Understanding Login Authentication.]

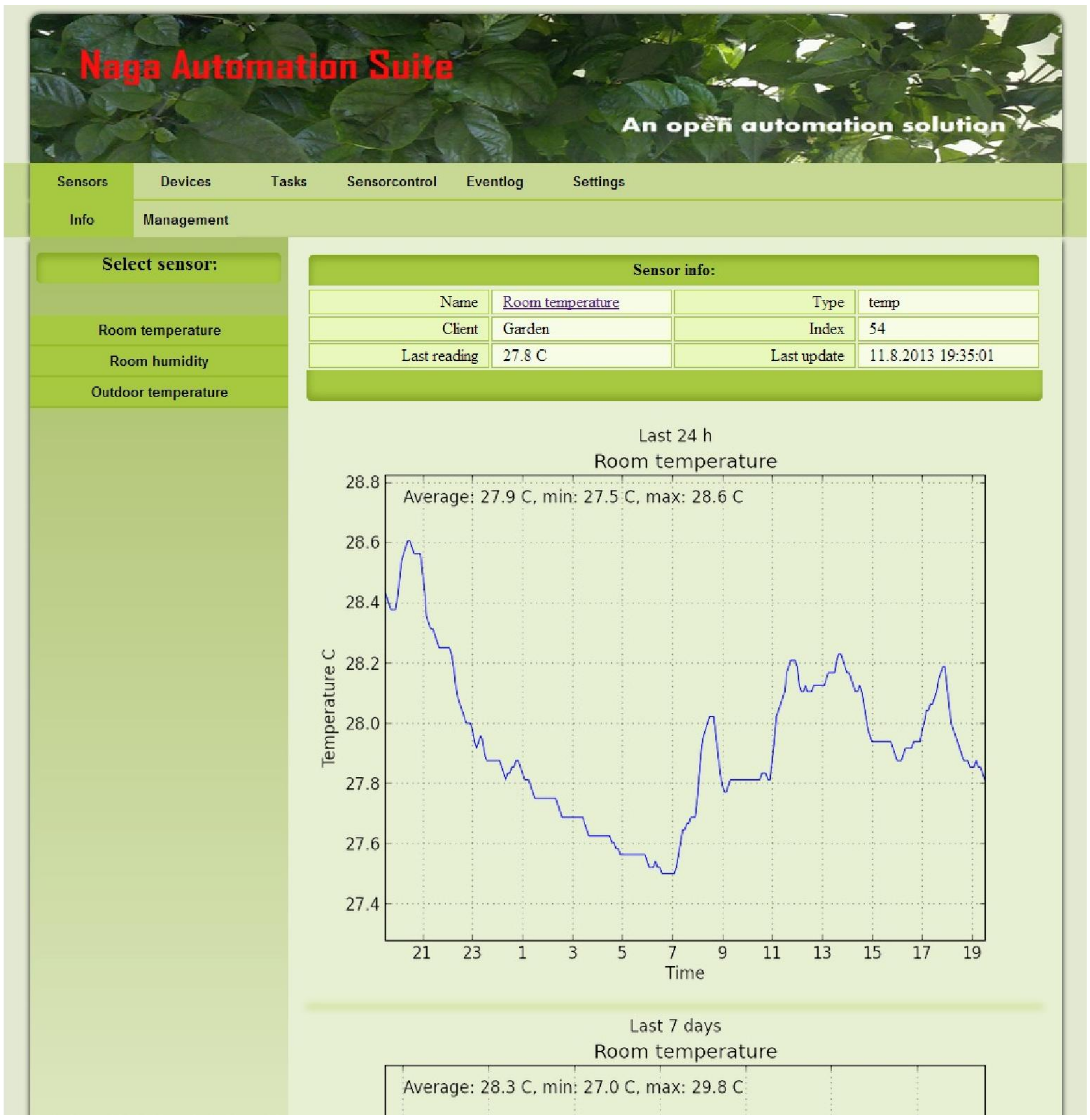
Furthermore it is important to encrypt all sensitive data. In plaintext form the possible attacker would gain straight access to all the data in case of a security breach in the server machine running the system. Lately this kind of hacking has been the most infamous way of gaining forbidden data. The situation is made even worse by the fact that today's computers are so powerful that they are capable of cracking a large proportion of passwords encrypted with algorithms such as MD5. [12. Anatomy of a hack: How crackers ransack passwords like “qeadzcwrsfxv1331”]

For this reason a sufficiently strong algorithm was chosen. One of such algorithm families is SHA-2 which has not been cracked to the date this document was written. More specifically the SHA-512 algorithm from the family was chosen because of its security in the present time. It will also likely remain secure in the near future.

The SHA-512 algorithm is used to encrypt the passwords in a file. The passwords are never stored in the memory of the system beyond the scope of their one-time usage. They are always loaded and discarded after use to make it more difficult for this sensitive data to leak for example in the form of accidentally sending it to the web interface amongst other data.

Encrypting passwords is not enough by itself. All the communication between the web interface and the server is still open for man-in-the-middle attacks. To make it harder to capture data being relayed, HTTPS with TLS is used to encrypt all the communication. This means that the possible attacker can only see incomprehensible bits of data being relayed between machines. Additionally this is important for security because browsers only base64 encode the login information before relaying it to the server. This data can be easily accessed by the possible attacker simply by decoding the base64 encoded authentication string relayed in the HTTP message header of each communication.

5.2.2 Design



Sensor info:

Name	Room temperature	Type	temp
Client	Garden	Index	54
Last reading	27.8 C	Last update	11.8.2013 19:35:01

Average: 27.9 C, min: 27.5 C, max: 28.6 C

Average: 28.3 C, min: 27.0 C, max: 29.8 C

Figure 9 Sensor view of Naga Automation Suite

The design is divided to tabs between different functionalities. For example sensors and devices have their separate pages because their functionality and properties are different. These tabs can currently have one sublevel of subtabs which bundle different actions together that correspond to the functionality described by the main tab level. For example subtabs of sensor tab may contain pages that display sensor info, or let the user add, modify and delete sensors. The different functionalities the system can perform are further discussed later on in this document.

The web interface itself consists of one single HTML file which is controlled by JavaScript routines and styling managed by CSS. Different tabs are loaded in div elements and hidden and displayed appropriately when the user wants to switch the view. This kind of approach was mainly chosen because the server core's webserver was written from scratch and kept simple. It uses only Python's standard HTTPServer libraries that only provide support for parsing and returning requests. It does not itself contain logic for sharing files.

Honing the outlook has not been one of the main goals for this project. Instead the attempt was to create an user interface that is at the same time usable and clear in design. However one of the limiting factors in the process of creating the user interface was however my inexperience with web development. This project was the first that was done by me that featured a functional interface; using JavaScript to modify web elements and communicate with the server was learned from scratch, although previous experience with Java helped to a great extent with the language syntax.

5.3 Server core

The server layer acts as the figurative brains of the system. Its purpose is to act as a middle point between the other layers, taking commands and also dispatching them. It also handles the responsibility of running the automation logic, such as scheduling tasks. It is the layer that defines the system's actual backend features and contains the most complexity.

5.3.1 Modularity

Modularity means that different functionalities are separated into independent modules that each contain the necessary tools to perform the one aspect that is defined for the module. This can mean for example that device handling is separated to its own module, which offers a solid interface for rest of the modules to use. Interfacing code separates the rest of the code from the internal logic of the module and greatly improves maintainability. Instead of all the logic being scattered around the code between many different functionalities, a modular approach means when you change one thing within the wanted module, it applies system-wide. [13. Modularity]

Attention has been paid during the development of this project to making the code modular. Not only does it make the code simpler to read and understand, but it also potentially makes it possible to only load certain modules at a time. This approach makes sense if one thinks about the user interface. Not all the users might need to use tasks and the current design makes it theoretically possible to simply not load the task functionality; the system would function normally without it. However to actually implement this idea, the project needs further restructuring of the code, but it is likely that we will see this kind of functionality in the future versions of this software.

Currently there are modules for logging, communications, plotting, scheduling, the HTTPS server and the logic serving the HTTPS request. There are also manager modules for configuration files, modules and devices. In future the modularity of serving the HTTPS request will be improved and the logic will be separated so that each page in the user interface has its own module which initializes the data on the page and handles processing the data the page sends to the server. This approach simplifies the user interface requests and makes it easier to choose which functionalities to load - which can be done dynamically.

The software already uses a dynamic module manager for serving form request. For example adding and removing sensors is performed by using it. The module manager was developed for a bit earlier project, but was ported for this project by changing just a couple lines that concerned loading the list of modules to be loaded. This module manager will be expanded in the future so it can handle the dynamic loading of almost any module in the system. The beauty of this kind of approach is that the user does not even need to reboot the program to do software updates. It is perfectly possible to update the source code during runtime and then use the user interface to reload the wanted modules. It will not even break the system if something goes wrong, because the module manager simply will not touch the existing module if the new one has some errors in it and can not be loaded.

5.3.2 Scheduler

Like many other parts of the software, the scheduler is also its own module; which in this case is independent of the rest of the software. This means that the scheduler is generic in nature. It could potentially be used in any project if the interface defined by the scheduler suits the purpose of that project. The reason why this scheduler is dealt with in such detail is because its definition and performance are vital for the functionality of this program.

The scheduler consists of a task manager and a thread which keeps track of time and polls the task manager to see if it is time to call the task manager's task handling functions. It would be possible to even switch the task manager implementation as long as the new task manager's interface supports five functions for resetting the task counter, getting the time when the next task will need to be executed, checking if all the tasks in the manager have been executed during the passing day, executing the next tasks and after that finding the next tasks to be executed. The resetting of the tasks and checking if all the tasks have been executed is because the scheduler is designed to handle only one day at a time - it is not a scheduler that understands days. While this approach is currently sufficient and was designed to be so, it is possible that in the future a new scheduler will be implemented which supports tasks spread across multiple days. This however would require redefinition of the software's tasks which is a different concept than the scheduler tasks.

What makes the scheduler generic on the low level is the scheduler's task implementation. Basically the task takes a function and the function’s parameter as input. After this the user sets scheduled events to the task and the scheduler handles calling the task's callback function and its parameter at the right time. This means that the scheduler can handle any function that takes one parameter. With Python's tuples the number of parameters can internally be extended because the tuple is a list-like data structure which is accessed like an array. In addition to this the user can for example specify if the task should automatically be removed after all its scheduled events have been executed once. The scheduler is also prioritized. Each task has its own priority and the scheduler puts the tasks in line according to their priority. This way it is possible to for example first force logging sensor data and only after this allow the scheduler to execute the tasks for drawing sensor data graphs.

5.3.3 Graphs

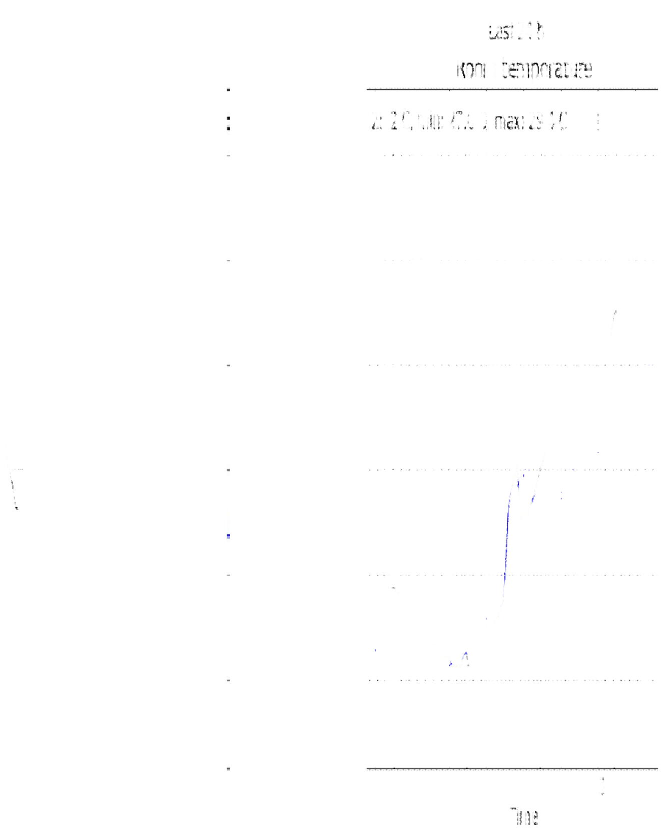


Figure 10 A graph generated by the server displaying temperature sensor data from the latest 24 hours.

Developing functional graph drawing proved to be a challenge. It is an important part of the software because it provides the user lots of informative feedback about the conditions in the area where the user has placed the sensor. Python's famous matplotlib was used to draw the graphs.

The biggest challenge was to develop a robust algorithm which fetches sensor data from the logging files and then processes it to the form that the matplotlib understands. The challenge was further aggravated by the initial bad logging decisions. At first the logging module stored the date and clock time in human-readable form to the logs. However this required quite a bit of additional parsing and taking dates into account which overcomplicated the graphing process. Later the approach was changed to use Unix time instead of date and time separately. Unix time is a single big number which marks the current time since the Unix epoch which was in 1.1.1970. With the Unix time approach it was simply a matter of handling seconds with the graphing module.

The graphing algorithm works by reading the time and reading values to two separate lists which present the x- and y-axis on the graph where x is the time and y is the reading.

The process is fairly straightforward, although there's a little smoothing done to the y-axis readings because in testing it was noticed that because the temperature sensors have a limited accuracy, the graphs tended to be blocky. The algorithm checks the reading's preceding and subsequent readings and calculates the average of these three values. The resulting value is then added to the y-axis list.

5.3.4 Devices and communications

Sensor and device management is handled with JSON objects which in Python are implemented with dictionaries which can contain different nested structures, such as other dictionaries, lists or primitive types. It is another part of the deliberate continuation of JSON object model in this project. The web user interface and the Arduino understand the structure of these objects which makes their processing easier.

The objects are sent to the web user interface just as they are, where they are parsed on the screen with JavaScript. For the Arduino communications only the needed parts of the devices are copied as a JSON object and sent to the microcontroller for further processing.

The communication protocol is responsible for parsing the objects to a transmittable string format which can then be easily reassembled in the microcontroller. The protocol messages are relayed through the Connection module which is agnostic to the messages it relays, it is merely a messenger when it comes to relaying messages. The module also calls the logging module internally so every message it receives back from the microcontrollers are automatically logged - this means the user does not need to worry about logging these messages which makes the code simpler.

5.3.5 Logging

The duty of the logging module is to turn the system's messages into a human-readable format and write it down in the logging files. This is achieved by dividing the logging module into several functions with each corresponding to a certain functionality, such a changing a device's state or adding a device or a sensor. In the functions the response is always a constant value which is checked and the appropriate message for the value is then written.

Currently all these message are hard-coded in the source code files, but in the future a filtering mechanism will be developed which dynamically reads the messages from the logging message files. This approach would allow modifying the logging messages while the system was running.

5.4 Microcontroller client

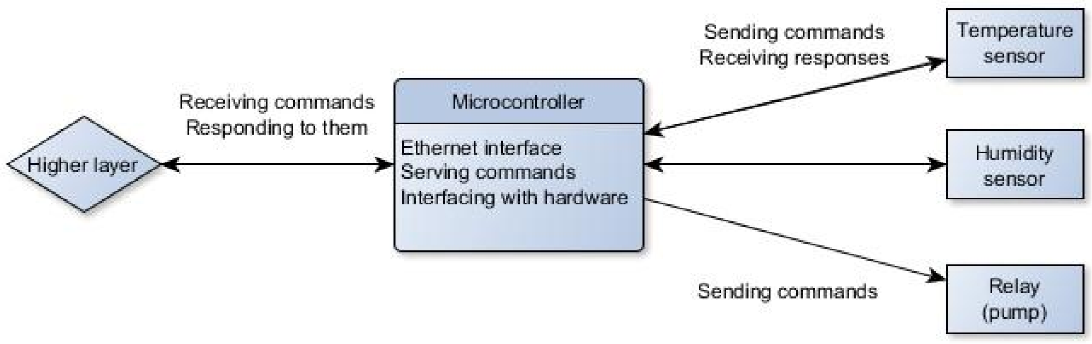


Figure 11 The microcontroller software model

The microcontroller layer was written as a client to server core which takes commands and executes them. These commands are usually used to control some real life devices, such as relays or reading sensor data.

5.4.1 Limitations

Many limitations exist when using microcontrollers, but they can be regarded as an unnecessary evil because of their convenience for controlling real life devices. One of the biggest limitations with microcontrollers are the hardware resources. They usually come with only a couple kilobytes of runtime memory and some dozen kilobytes of space for the actual program binary. With the Arduino Mega board these limitations did not affect the development progress, but they come also with another downside. Because of the limited resources the developers of the board had to cut down programming features so the Arduino programming environment is actually quite low-level. To name a few effects of this limitation, exception handling is not supported, the Arduino standard library is minimalistic and string operations are rather limited. Lack of good string management meant it was difficult to write code that interfaced with higher level routines, such as displaying data on a web page. Therefore a rather simple yet clear and expressive communication protocol was developed for communication between the microcontroller and the server.

There also is not onboard storage space which could be used for example to store web pages or sensor data. It is true though that this kind of functionality can be provided with a SD card module, but it was elected not to force the user to acquire the module and a spare SD card because the same functionality can be provided on a computer with much more ease.

5.4.2 Design

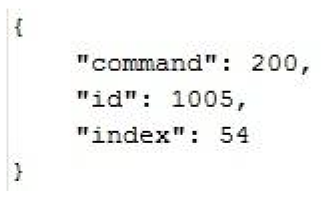
Taking into consideration the aforementioned limitations, it was decided that the microcontroller layer would provide mostly just an interface that lets the actual server access the hardware, however some logic is also implemented to act as safeguards against doing any harm with the hardware. The safeguards are important because one of the last things one wants to happen are physical damages caused by faulty software code. These safeguards concern driving pump devices. When they are on, depending on their type they pump out water constantly. In hydroponics and watering plants badly designed code can pose a danger of the water reservoirs overflowing and causing water damage. The built-in safeguards only let the pumps to be driven for a predefined time before they are forced to be shut down. Another safeguard for pumps uses hygrometer sensor to monitor water level in the container the water is pumped in. The user can define a sensor value and after this value has been exceeded the logic cuts off the power from the pumps.

Other than these features, the layer currently concentrates mostly on performing the hardware interfacing for the server layer. To make the interface easy to use and solid, a communication protocol was developed from a scratch. Originally it used commands that were divided into one byte length chunks where the byte position affected the meaning of the byte. For example first byte might tell the command's type, second byte would tell the device's type and the third the index of the device. This kind of approach proved difficult to be implemented on the kind of resources available. The code parsing the commands had to be hundreds of lines of code in length and it was difficult to create complex command structures, such as sending float values to the microcontroller.

Luckily a JSON implementation library for the microcontroller was found. JSON is a lightweight data-interchange format which maps values to keys and treats them as objects. Using this library it was possible to implement a communication protocol that had a clearer structure and made accessing values easier because of the simple interface the implementation features.

5.4.3 Communication protocol

The real core feature for the microcontroller is the communication protocol which defines a common interface that both the microcontroller and the server understand. How the protocol is defined is crucial for the function of the system because through it are defined the features which the system can perform in the low level are defined. What this means is that the protocol needs a clear interface that works the same way in every scenario. For this reason the protocol relies heavily on constants. Every action and command and response - with the exception of such as sensor reading - is defined as a constant. For example the key for the response is a constant, and the device operation responses are constants, like TOGGLED_DEVICE_ON which has the value 108 which in turn tells the server's constant map what the message means.



```
{
    "command": 200,
    "id": 1005,
    "index": 54
}
```

Figure 12 Example of a protocol compliant message. It takes the reading from sensor with ID 1005 at pin 54.

Why this kind of approach is important is that it reduces the chance of making mistakes during coding and also makes it easier to understand what the code does. It also tells exactly what features the system can perform and helps with error handling, because different scenarios have clearly distinguishable error codes.

The biggest challenge during the development was to implement the protocol with code. The first iteration of the protocol used byte coded command parameters. What this means is that commands consisted of certain length of bytes, with each byte defining a certain thing and also the byte position mattered. Implementing this kind of protocol proved to be difficult because of Arduino's lesser string handling capabilities. The messages had to be read byte by byte and for each byte there had to be an if-else table which handled finding what bit of code to enter next. In the end the implementation was mostly ready and functional, but consisted of multiple hundred lines of code and was hard to read how it worked.

A real fortunate turn of events was the discovery of a library for Arduino that implements support for JSON (JavaScript Object Notation). The reason this discovery was important is because by re-implementing the protocol with JSON-messages the code could be massively simplified and the previous, strict implementation got a lot more of freedom and flexibility. This is discussed in further detail later on.

But with the JSON protocol the messages are now defined with key-value maps which are syntactically supported by the Python language. With the Arduino code these values could be easily extracted from the keys and no string parsing was required.

This in turn made it easy to put the protocol implementation in its own class with clear internal structure. The way this implementation functions is that the software's Ethernet interface reads a message and passes it to the protocol handler. It then parses the message and calls the right function to handle the message. For example if the message's command is SEND_INSERT, the message is passed to Insert()-function which further processes the message and if all the rules have been fulfilled, inserts either a sensor or a device to the memory. If one of the rules has not been fulfilled, the functions are capable of composing an error message clearly stating at which point something went wrong. This way the software can tell the user easier what happened and the user does not have to guess.

5.4.4 The problem with the JSON library

It must be mentioned that it was a real struggle to get the JSON library working as intended. Several weeks were spent without almost no progress because the problem with the JSON could not be solved. At first I rewrote the protocol implementation and things worked well, but after a while the microcontroller always crashed and reset itself. Some debugging was done and the problem was revealed to be the microcontroller running out of memory. This finding was made possible by the MemoryFree library which lets the user read available memory in an Arduino during runtime.

The real difficult thing was to pinpoint the origin of the problem. The JSON object management requires the use of pointers and at first it was thought that the communication protocol's pointer management was faulty and therefore leaked code. However after extensive testing and trying to get it right the problem still could not be solved. Neither did reading the JSON implementation's GitHub page forum give any hints about the nature of the problem. Finally some debugging code was written into the JSON library's memory management code. This was the key in solving where the problem resided, yet it still did not make it easy to pinpoint the exact location of the problem because the nature of the problem was rather elusive.

The debugging code printed out Arduino's free memory in different phases of freeing memory when deleting JSON objects. The memory freeing routine works in a way that loops through the JSON object's tree structure and deletes one element at a time, then proceeding to the next one. The results varied, some objects could be freed completely, some got stuck at random elements. This was highly puzzling since the failures seemed to be random.

Eventually when I studied the structure of the messages that were to be freed and checked which elements could be freed, the solution started to unravel itself. The messages that could not be freed from memory contained JSON objects on two levels. This means that at least one JSON key held an another JSON object with a key-value map. While it is true that the JSON library that is used can handle JSON objects that contain this kind of multilevel tree structure and it can even free their memory, it does not take into account one perhaps rare case. Usually one would define a multilevel JSON structure so that the element which contains a nested lower level object is defined last in the structure. However the messages are generated using Python dictionaries which are not ordered, which means they can exist in any imaginable configuration. What happened was that the nested lower level object was not the last element in the protocol message. Whenever there was an element to be freed after this nested object, the JSON library lost track of where it was in the JSON object structure when it was erasing it and the system crashed. This is a clear bug, yet considering the low level of resources the Arduinos have, it might not be desirable to even fix it because this would mean that more resources would be used. There exists at least one other JSON implementation for Arduino that states it does not support nested structures for this reason. If the problem is at least known, then the users can circumvent it.

In the end the problem was fixed easily by removing the nested JSON structure from every protocol message. This does not affect the performance at all, the nested structure was mostly used because it followed object-oriented programming more strictly. In the nested messages the first level contained information about the command and its parameters and the nested structure contained the device or sensor sent to the Arduino.

5.4.5 Sensors and devices

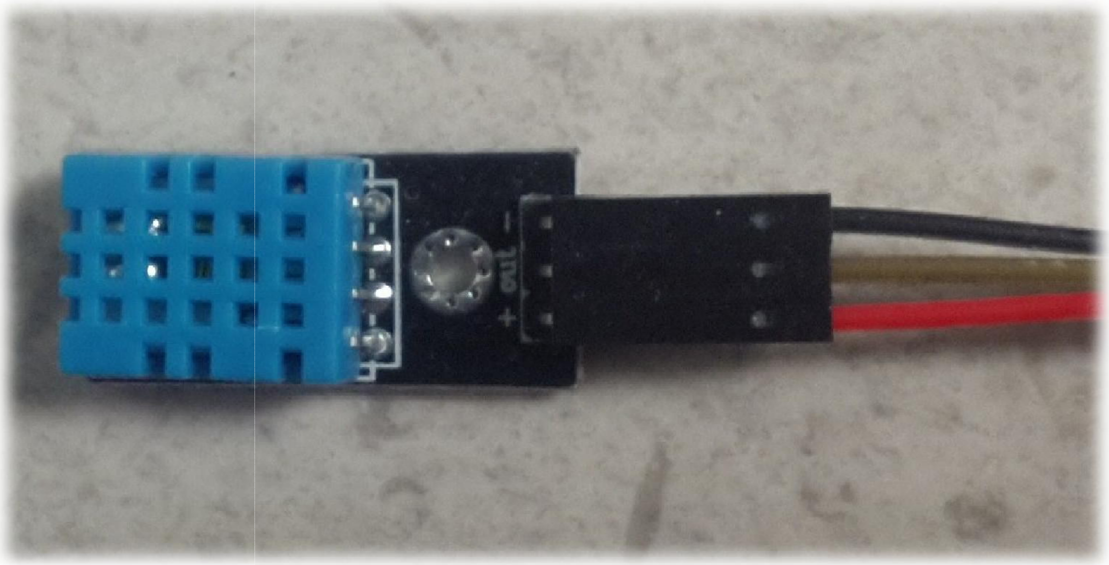


Figure 13 Example of a sensor, the DHT11 humidity & temperature sensor

In addition to what was defined abowe, the microcontroller layer has to deal with some kind of sensor and device management. It would have been somewhat possible to merely make the Arduino an interface that allows access straight to the hardware peripherals. However this kind of approach has its disadvantages and it was decided against it. The cons include less sophisticated error handling, lack of built-in safeguards for pumps and more difficult sensor data gathering because the readings would need to be converted to human readable values in the server layer. Furthermore some sensors and devices use Arduino's digital pins for communications and without a library implemented for Arduino, using this kind of sensors would not be possible because this kind of libraries require low level management and precise timing.

Making the system dynamic was one of the important goals for this project. On the microcontroller level being dynamic means being able to add, alter and remove sensors and devices on the fly instead of using precompiled objects. This is the kind of approach that is lacking in amateur solutions and the intention was to develop conclusively a good dynamic solution. It is achieved by storing objects inside arrays with manager objects. The manager objects are used to manage the sensor and device objects they contain. This way nothing is hard-coded; everything can be dynamically modified.

In the current model sensors are stored as generic objects that contain a field with information about the type of the sensor. When the server requests a sensor reading from the sensor, a generic getReading() function is called which checks internally which type the sensor is and then calls the appropriate function for the sensor's type. This kind of generic approach was taken after typecasting objects proved cumbersome. Originally the sensors were objects that inherited from the main sensor class and the subclasses contained the logic for handling everything needed. The logic was built behind a common interface which all the subclasses obeyed. The subclasses were cast to the base class and stored to an array. However before using the sensor object it had to be cast back to its subclass, but this approach did not work quite as intended because of different errors and ugly syntax. Therefore it was decided to rebuild the sensor layer and the generic model was adapted.

One good thing about this kind of generic model is that it makes it easy to add new sensor types later on. The user only needs to import the new device's library, write a function that fetches data from the sensor and add a new constant for the sensor's type, the rest works automatically.

The device management is not as organised as it needs to be. Currently there's a manager class for a relay board with eight adjacent relays and a manager class to manage pumps. What makes this a bad design is that the pump manager relies on the relay manager. The pump manager is kind of an additional layer upon the relay manager, but does not inherit from the relay manager. The current approach works, but is not the best kind of object oriented design. Also because of the limitations of the relay manager, the layer will be redesigned at some point. The aim is to make each device its own object that inherits from the relay class. This way each hardware pin could be assigned to be a relay pin instead of allocating eight adjacent pins like in the current model. This of course allows more flexibility and the system will be more dynamic because there are no limitations from the relay layer.

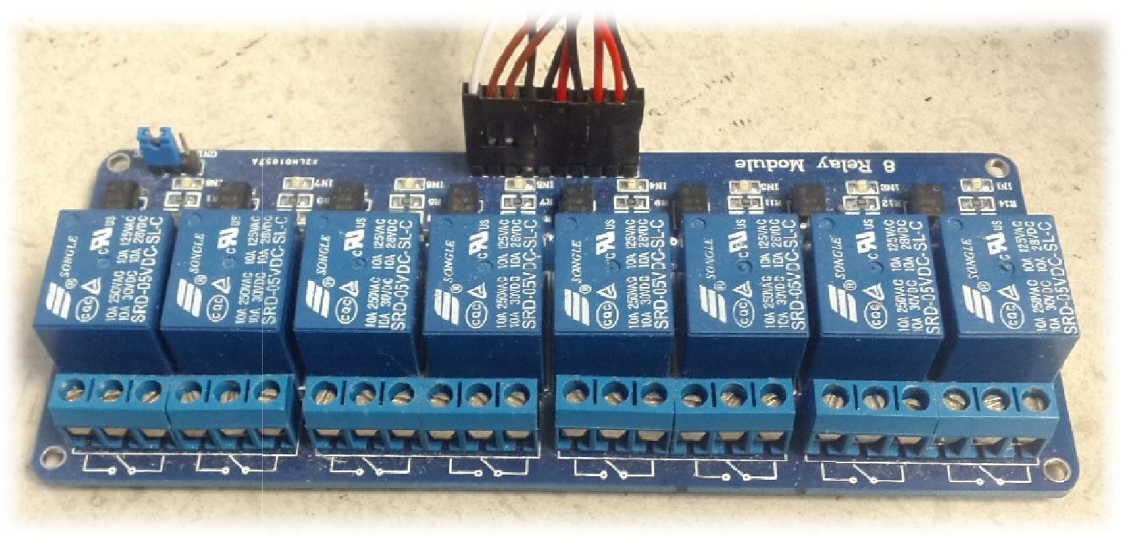


Figure 14 The relay array used to drive pumps and lights

It was explained earlier that the pump objects contain few safeguards to make using them safer. These safeguards are implemented in the pump by keeping track of how long the pump has been running and only allowing a certain maximum time limit for them to operate; and by measuring the water levels with a hygrometer.

The timed safeguard is simple. When a pump is started, the start time is logged in the pump object. With each cycle the Arduino software runs through all the pumps and checks if there have been changes in their states. If the pump is marked to be running, the software updates the current time and compares it to the start time of the pump. If the delta is larger than the maximum allowed, the pump is shut down and the server is notified of the action.

The hygrometer approach uses a hygrometer sensor. If this safeguard is enabled the pump object's usesHygrometer flag is set to true in the software and hygrometer sensor's pin index is stored as a field to the pump object. This means that the hygrometer sensor needs to exist in the supplied pin or the safeguard will not work. Internally the safeguard is handled generally in the same way as the timed safeguard. During each software cycle Arduino checks if certain time is passed since the last sensor measurement and makes a new measurement if it has. If the new value exceeds the threshold value set to the sensor the pump will be shut down and the server is notified of the action.

During testing it was noticed that most hygrometer sensors do not like to be submerged in water for extended periods so they are not really a very durable approach especially for hydroponics. Therefore using this safeguard is not openly recommended. However it will remain in the system for the time being. It is possible that a good hygrometer solution will be found some time in the future. It should be noticed that both of the safeguards can be enabled at once; they are not exclusive.

5.5 Python

The language of choice for implementing the server is Python. It is a "widely used general-purpose, high-level programming language". [14. Python (programming language)] Python's features include, but are not limited to dynamic type system, automatic memory management, object-oriented programming paradigm and being highly expressive (more features in less lines of code). It is most commonly used as an interpreted language, with the standard and open CPython implementation being the most used implementation. However there exist other implementations too, such as Jython with which the source code is compiled to Java byte code and run atop of the Java Virtual Machine.

There are many defining factors as to why the language was chosen. The language allows rapid development because of its clear and well-designed syntax. Python code is sometimes described to be self-documenting. It is also widely used by the coder community and comes preinstalled on most Linux and Mac environments.

The language being interpreted in nature often raises questions over its performance, but on modern machines this is not a problem. Python is by no means slow. In most cases it might lose to compiled languages like C, but it still can achieve a satisfactory level of execution speed even without any optimizations. Especially if the program is not performance limited, then Python should not pose any problems for the user. [15. PythonSpeed] In the case of this project, the heavy processing is done in the sensor graph plotting module. The graphs do not need to be generated all the time and the execution time on a Raspberry Pi for a single graph is 7 seconds, which is acceptable if there are not dozens of sensors in the system. On a modern PC system it was measured that the generation of one plot took a couple hundred milliseconds which is not a bad figure. In the future the plotting module can be multithreaded to provide even faster plot generation on multicore systems.

5.6 JSON

JSON (JavaScript Object Notation) was chosen as the data storage model because of its suitability with Python and JavaScript and the flexibility this gives for the code. JSON objects are handled naturally by the two languages' syntax so no additional coding is required to process the data. On both languages the objects are handled as key-value map pairs which lets the user access the map values with the syntax *object[key] = value*. The significance of this syntax is that no getter or setter methods are needed. On a language like Java which does not syntactically support JSON doing simple operations can easily take three lines of code. On Python and JavaScript the code can be made more compact and easy to understand.

While it is true that the JSON model is not natural to use on the Arduino's microcontroller C/C++, it does not make things inconvenient either because in the end not much JSON handling is required on this level. Besides the alternative of parsing strings manually was an even worse option.

JSON's structure is a tree which can contain elements - or key/value pairs - of primitive types like integers, booleans or doubles. The elements can also be strings or objects, maps or lists which makes it possible to create nested tree structures. [16. Introducing JSON.]

6 Functionalities

This section discusses functions that the system can perform. The aim was to divide the user interface into views so that each view offers a certain distinct functionality that it can perform. For example there are views for sensor data and task management. This in part helps with the aim to make the system configurable. If each view performs a certain functionality, then the view is easier to exclude from the interface if it is not needed. In future the aim is to provide more functionality through new pages.

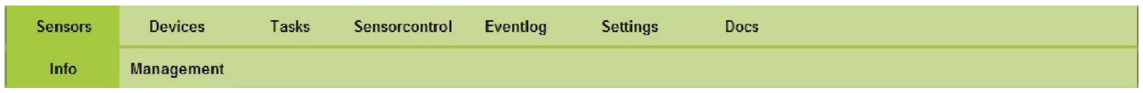


Figure 15 The current menu view of Naga Automation Suite

On the main interface menu level the view describes loosely the idea of what the view is responsible for, for example sensors. There's a second menu level which divides the view to smaller functionalities that concern the idea. For example there's a page for the sensor info, with data like the sensor's last reading and the sensor graphs. Then there's a page for sensor management which offers functionality for adding and removing sensors. In future a possibility to modify sensors and other functionalities will be added.

Like explained above, in the **sensors view** the user can view information about the sensors added to the system, including which port they use and graphs that display the sensor's readings over the last 24 hours and 7 days. In the management page the user can add new sensors or if they click the sensor's name on the info page, they get to the remove the sensor view on the management page. In fact every view which governs a functionality that interfaces with real world objects contains their own management page. Currently these views include sensors, devices, tasks and sensorcontrol.

Devices view contains a list of devices added to the system. On the info page the user can see which devices are on and which are off. It is also possible to control the devices' states from the info page which can mean for example toggling on lights.

Tasks perform timed jobs on devices. The user can add tasks which currently either toggle one device on or off at certain times each day. There can be multiple times a task can perform the action set to it. If the user wants to for example automatically toggle lights on and later toggle them off, they will need to add two tasks. In future it is possible that it will be made possible to drive devices for a certain time with one task. For example a task might be "drive lights A for 3 hours starting from 20:00".

Sensorcontrol as a concept means controlling a device's state with a sensor's reading. For each sensorcontrol a low and a high threshold value is set. For example, if sensor A reading is over 25, toggle on device B. If sensor A value dips under 20, toggle off device B.

In the **eventlog** view the user can see what has been going on in the system. The Python server logs down messages when something happens in the system, for example when devices go on or off, or if there's an error, it is reported. The eventlog messages are colour coded. Green indicates everything is OK, yellow indicates actions such as something being added or removed and red is an indication about something failing and it should arouse the attention of the user.

Settings view gives the user control over parameters the system is functioning on. Currently these parameters concern intervals between actions such as reading sensor data or generating sensor graphs or how many lines of eventlog should be retrieved from the server.

Docs is short for documentation. It contains a written document that tries to explain how each view in the system works and for example how to add new tasks and what different parameters in the add new tasks form.

7 Testing

Testing is always an important part of developing applications. Its purpose is to ensure the wanted software quality is achieved. The program was tested by the author and by the Finnish chili entrepreneur Jukka "Fatalii" Kilpinen. The help of Jukka must be stressed here because not only did he extensively test the product's functionalities, he also provided good ideas for new features of which some are already integrated in, such as the sensor control system.

Jukka's contribution to this project came after he was looking for a solution to automate his greenhouse. He grows grows hundreds of chilies annually and the work burden without automation is heavy. After I heard him mention the kind of software he needed, it was apparent that this project could help him out. In the end the benefit was mutual because he had many good ideas about how to improve the system. In this way his testing has been highly valuable. He has also reported when some features are not working as expected and together we've figured out how the software misbehaves.

This project was not developed for Jukka per se. He actually uses an alternative version of the software where the microcontroller layer is substituted with Telldus Technologies' home control system. In short, Telldus' devices allow the user to wirelessly control their lights and electric sockets and read temperature and humidity sensor data. Telldus provides a free API which can be used to interface with Telldus hardware. [17.|18. TellStick – Awaken Your Gadgets & Welcome to Telldus API.] In future one aim is to merge these two projects so the software can handle both hardware interface layers simultaneously. During development some of the features were developed solely for the Telldus version of the software and later merged to the thesis version which is discussed in this document.

The testing I've done personally has mostly been performed as new features have been implemented. Some test cases have always been developed and tested to see that the program behaves as it was designed to do. Sometimes especially in the microcontroller layer the testing proved to be difficult, but in the end the code should be at a point where no real big issues are hiding.

8 Further development

The prospects for the future development of the project are planned well ahead of current progress. Now that the core functionalities are mostly there where they should be, it is time to start extending what the system is capable of. A part of this work will still be refactoring the current codebase because when this project was undertaken, I did not have a solid idea about how the system should be defined internally. The experience came from working on this project and now that there's a clearer picture about how the systems should be designed so that they last for years to come and allow also the expansion of the system, it is time to redesign some features before the codebase grows too big. For the project this means dividing every module more clearly in their respective functionalities and also moving more responsibility from the web user interface generation to the back end server side. Currently the data needed by the web user interface is fetched from the server when the user accesses the page and the data is globally accessible by every page. Data is also refreshed when needed, but the current model for example loads all the sensor information when it is needed for just one sensor. In future the data will be tailored for each page specifically and it will not be stored in the browser globally. This will be handled by the web user interface controllers in the server side. This also makes it easier to load only the needed information from the server. For example if the user wants to open a sensor to see data about it, the data for that sensor only can be loaded straight to the page and discarded immediately when the user leaves the page. Currently the data persists.

Once the refactoring is done new features can be developed. In the early phase the sister project's Telldus layer should be merged to this project and it should be made possible for the user to manage multiple clients dynamically. This means that the user could concurrently control devices and sensors through Telldus and one or more microcontrollers. Some kind of logic to distinguish these different clients will be needed. Probably the biggest concern about this feature is how to make the two different layers compatible with each other. Work on this field has already been done and the project's sensor and device data model has been adapted to resemble more the model Telldus uses.

The sister project also makes it possible to get email alerts from the system if the sensors indicate alarming readings. This feature will be ported to this project. It will not require a lot of work, but the aforementioned internal changes should be adapted first. Also the way the email alerts work could be further refined to work more comprehensively.

Jukka Kilpinen has often requested a feature where the user could fetch sensor graphs from history. This is a good feature and on paper it is also possible to implement.

Configurability has been mentioned to be important with this project. It is often not the case with proprietary products, but in the open world configurability is often regarded as a good goal. The idea is to develop one settings manager that takes care of handling the configuration files. In the web user interface the user would have a similar view to Firefox's “about:config” page which lists the configurable items with their values and lets the user change them easily. Once one setting is changed, it takes effect immediately. Having one robust core for changing settings makes it easier to let the users configure more and more things about the system. For example this could allow different configurations for certain features. Currently the user can define how many minutes is the interval between generating sensor graphs, but Jukka Kilpinen has requested a feature that only generates the graphs when the user tries to access the sensor page. The problem with this approach is that Raspberry Pi is not powerful enough to do this in real time, although a modern personal computer CPU is. If settings like this are easy to make, then they are more likely to be added.

One interesting feature might be integrating plants view to the system's interface. The users could keep track of the plants growing in their home by creating a plant object. Sensors and devices could be associated with each plant. The association would then add data from the sensors and buttons for the devices on the plant's page. This kind of view would help especially with watering plants potted in soil.

Numerous, smaller changes have been planned. Some of them make debugging easier, like making each method call print the call time in microseconds and the name of the called object and method. Once the system grows, it is easier to hunt down problems through the debug messages when the developer can see which methods the system calls.

Other changes could concern messaging and multiuser support. Currently messages are hardcoded, but this is a bad practice. Eventually they will be moved to configuration files that are loaded when needed. Multiuser support appears to be a good concept on paper, but no plans about the schedule have been made. It would make sense to allow more users in the system if the user wants to use the system with other people, but does not want to share the password. The users could have different access settings, so some users might not see all the pages that the main users sees.

After describing so many various ideas it is good to look at the big picture. On a broader level the idea is to improve the system to use more useful features and give the user a more streamlined and logical user experience. More control will also be given to the user with the new configuration system which makes it easy for the user to alter the system's parameters.

In future the system will also hopefully develop in a way that makes it more suitable for normal home automation, because considering the level of modern technology, home automation is surprisingly unpopular and little thought of. The potential to achieve great things in this field is high. The single biggest thing that allows home automation approach to be adapted is supporting wireless devices. Today there are many hardware suppliers that produce wireless electric sockets, wall mountable light switches and even sensors and these devices are the perfect solution for automating one's home. No ugly and inconvenient cables need to be installed. The Naga Automation System already contains logic that makes home automation possible

9 How to commercialise

One of the base ideas of this project was to develop an open system which anyone could use freely. This idea will remain, however it would make sense if the system was commercialised. Great thought was put into this concept and working on the side project with Jukka Kilpinen gave good ideas for commercialisation through the Telldus business model which was studied while getting to know how to interface with the Telldus API.

The way Telldus works is that they release almost all of their code openly, but they sell closed hardware which lets people use the code with real life applications. This kind of business model could work for this project as well. The market for such devices is not congested either, so there would be space for innovations. For example Telldus does not provide any automation functionality with their code. The company concentrates on making it easy for the users to interface with their devices. Where this project could potentially excel is providing good automation functionalities for the user so in addition to their garden they could automate things around their house, such as when heating goes on, which would also save energy.

Like it has been mentioned many times before, this software is open source and will stay that way. Therefore it does not make sense to try sell it. After all, the goal was to offer people a free tool which lets them automate their gardens with little effort using the hardware of their choice. This kind of approach requires some kind of understanding about computers and electronics so it might be a limitation for some people to adapt the system in their use cases. For this kind of people a ready solution engineered specially for this software could still allow them to use it for their benefit. In theory all that needs to be done is to develop a hardware bundle that can perform the functionalities the system is capable of.

It is potentially possible to cut off the microcontroller layer and substitute it with a wireless 433 MHz protocol which many home appliances support today. This way everything needed by the system could be built inside one small box with Ethernet and power cables running in it. Software could be preinstalled and the system could be readily configured through the web user interface when the system boots up. The rest of this chapter is written around this idea because it seems like the most realistic solution.

To make the device more tempting for the end user, there could be some special features which are not available in the free source code. The free code would not have any customer support, using it leaves all the responsibility for the user, not the developer.

9.1 Licensing

Even though the code is open source, it is still important to protect it from proprietary software. For this purpose a software license is used which protects the code is used. The popular open source license, GPLv3 was elected because of its open nature. It permits the free use of the code in other people's projects. It even allows the commercial use of the code as long as the source code stays open and the work deriving from the software uses the same license. What it does not permit is using the code in a proprietary system without opening its source code. [19. GNU GENERAL PUBLIC LICENSE.]

9.2 Safety concerns

Commercialising something also brings legal bindings which require the service provider to ensure the safety of the service they offer and in problem situations carry the responsibilities caused by possible damage. Therefore it would be essential to make sure the system is safe to use.

There are two main categories that need to be taken care of. The first is electrical safety of the device. The second is safeguarding the system against malfunctions and potential problem situations.

For the first case National, European and international electrical safety standards exist that ensure the device is safe to use. Usually devices can not even be sold in the country if the device has not passed the country's safety standard tests. With this kind of project the safety concerns arise mostly from controlling the relays; but the problems are not big if normal quality parts are used and they are shielded from the elements according to the IP standard (Ingress Protection rating). IP45 certification should be enough if the device controls relays physically attached to it. This rating means the device is "protected against solid objects over 1 mm (tools, wires and small wires)" and it is "protected against low pressure jets of water from all directions". [20. Ingress Protection Ratings.] The device would be designed to be used like a normal household appliance which does not need higher protection rating. However it might make sense to use 433 MHz wireless relays and sensors instead of physically wired ones, which would also mean that there is no need to place the device anywhere near potential water sources.

Certain things can be do for the second case, but in the end there's not much that can be done about it. The rest can be assured by the responsible use of the system by the end user. Like explained in the microcontroller software section, safeguards against hardware malfunctions are important. One way is to develop special device classes for devices such as water pumps which contain protecting logic. For example if the user adds a pump device, it would automatically contain logic that allows the pump to be in the on setting for a certain amount of time before it is forced to go off. However there are so many different hardware devices so that one universal setting will not work well. The system needs to be configurable which means giving control to the end user. This also makes it possible to misuse the system by setting far too long maximum on-time for the pumps which can cause trouble like over-flooding. However a case like this is clearly an user error and the responsibility over possible damages falls over the user. To protect the company manufacturing the devices legally, it can be explained in the user manual how the system should be used and tell that all usage that does not follow the guidelines also voids warranty.

9.3 Potential cost

Cost is an important factor with every product's development and commercial success. It makes sense to drive down manufacturing costs for more appealing price and a higher profit cover. Some of the possibilities are to develop hardware tailored specifically for the project's needs. On the other hand it could be possible to buy some existing hardware that might contain something useless for the needs of the project, but still be an affordable solution because it is mass-produced.

Developing your own hardware is a costly and a long process and requires lots of knowledge about hardware and extensive testing. It is not really an ideal option for a new startup company - which a company manufacturing the devices for this project would most probably be.

From the existing hardware solutions Raspberry Pi is probably the most serious contender. It would be possible to use cheaper, lower performance hardware, but then the entire codebase would need to be rewritten in some other language than Python. Raspberry Pi on the other hand is a full-blown computer and performs like one. Using it would mean that the hardware could easily run the system itself and leave space for a lot more. Hardware limitations should not be a concern in the future and in theory it would be possible to let the users use the device as their own home Linux server machine. However giving too much control for the user also means that a lot can go wrong, so extreme caution should be put into designing. As an idea, giving the user more control over the device sounds good.

Raspberry Pi would also be a cheap option. Currently there hardly exists any other device that can compete with the performance and low price it offers. There are other similar boards such as PandaBoards or BeagleBoards, but only BeagleBone Black can compete in Raspberry Pi's price and performance category. [21.|22. PandaBoard & BeagleBoard.]

Raspberry Pi is open hardware which means it is possible for anyone to acquire the schematics for free and produce their own boards. The production could be done by mass-producing the boards and peripherals in China with an estimated cost of 50 € per board. This would be a really cheap way to produce hardware, but still would need a relatively large sum of startup capital to get everything running. Producing 10 000 units would require an investment of 500 000 €, which in modern business world is not much itself, but for a young startup company it would be. Therefore finding good investors would be one important part in the process of starting up the business.

The cost was estimated with the $35 price for the Raspberry Pi and then adding the additional parts to the price, such as the module needed for controlling 433 MHz devices and packaging for the hardware.

10 Conclusions

While perhaps not the most common subject for a thesis, the project has been highly rewarding and satisfying. The author feels that the process has been successful and the main goals have mostly been fulfilled – the system is dynamic, open, easy to use and inexpensive to build with the reference hardware.

The goal of being dynamic was achieved by making it possible to change many aspects of the software during runtime, including various settings changes and adding and removing sensors, devices and tasks. It is open because the source code is freely available for anyone to download from GitHub, which also means the people are free to alter it as long as they obey the GPLv3 license. Easy to use might depend on the subjective view of the author, but the user interface should be quite straightforward and the same design logic is used on every page for displaying and altering data which usually makes it easier for the users to use the software. The system is also inexpensive because the user can easily build their own setup with less than 100 € using the reference hardware, while the software is of course completely free of charge.

Currently the field which covers this thesis project is not too crowded with similar projects, yet there has recently been a lot of talk about the possibilities and prospects of systems that perform similar functionalities. Hopefully in future this project gets featured in the media in some way so it can for its part benefit the modern trend of bringing the automation to everyone's home.

There will be a big amount of work ahead in this project, but this is certainly a positive thing because it means the project has still lots of potential to develop. At least in the current form the system is usable and stable, so hopefully some people get to take advantage of the features it provides. It requires a lot of effort to get support for this project and get people to adapt it in their own homes, but hopefully some progress will be made. The progress will be slow, but at least there's not much competition.

It is been a relief to get to this point. Now the pressure to get things done is off and the author can finally concentrate on making improvements that did not fit in the schedule of this project.

11 References

1. What is GardenBot? http://gardenbot.org/about/ [Read 23.8.2013]
2. Arduino Greenhouse mrk 2 http://www.youtube.com/watch?v=qmSyfodR7E8 [Read 23.8.2013]
3. Web based automation courtesy of Raspberry Pi http://hackaday.com/2013/07/12/web-based-automation-courtesy-of-raspberry-pi/ [Read 23.8.2013]
4. Greenhouse Automation: Climate Control Systems Inc. http://climatecontrol.com/ [Read 25.8.2013]
5. Arduino. http://www.arduino.cc/. [Read 2.7.2013.]
6. Priya Ganapati, Why Arduino Is a Hit With Hardware Hackers. http://www.wired.com/gadgetlab/2010/07/hardware-hobbyists-arduino/. [Read 2.7.2013.]
7. Arduino. Arduino/Processing Lanaguage Comparison. http://arduino.cc/en/Reference/Comparison. [Read 2.7.2013.]
8. Arduino Mega http://arduino.cc/en/Main/arduinoBoardMega [Read 27.7.2013.]

9. FAQs | Raspberry Pi. http://www.raspberrypi.org/faqs 26.8.2013.
10. Multitier architecture https://en.wikipedia.org/wiki/Multitier_architecture [Read 25.8.2013]
11. Understanding Login Authentication http://docs.oracle.com/javaee/1.4/tutorial/doc/Security5.html [Read 18.7.2013.]
12. Dan Goodin. Anatomy of a hack: How crackers ransack passwords like "qeadzcwrsfxv1331". http://arstechnica.com/security/2013/05/how-crackers-make-minced-meat-out-of-your-passwords/ [Read 26.8.2013.]
13. Cory Janssen. Modularity. http://www.techopedia.com/definition/24772/modularity [Read 26.8.2013.]
14. Python (programming language) http://en.wikipedia.org/wiki/Python_(programming_language) [Read 24.7.2013.]
15. PythonSpeed http://wiki.python.org/moin/PythonSpeed [Read 13.8.2013.]
16. Introducing JSON http://www.json.org/ [Read 24.7.2013.]
17. TellStick – Awaken Your Gadgets. http://www.telldus.se/products/why [Read 13.8.2013.]
18. Welcome to Telldus API http://api.telldus.com/ [Read 26.8.2013.]
19. GNU GENERAL PUBLIC LICENSE http://www.gnu.org/licenses/gpl.html [Read 13.8.2013.]
20. Ingress Protection Ratings http://www.engineeringtoolbox.com/ip-ingress-protection-d_452.html [Read 19.8.2013.]
21. PandaBoard http://en.wikipedia.org/wiki/PandaBoard [Read 19.8.2013.]
22. BeagleBoard http://en.wikipedia.org/wiki/BeagleBoard [Read 19.8.2013.]