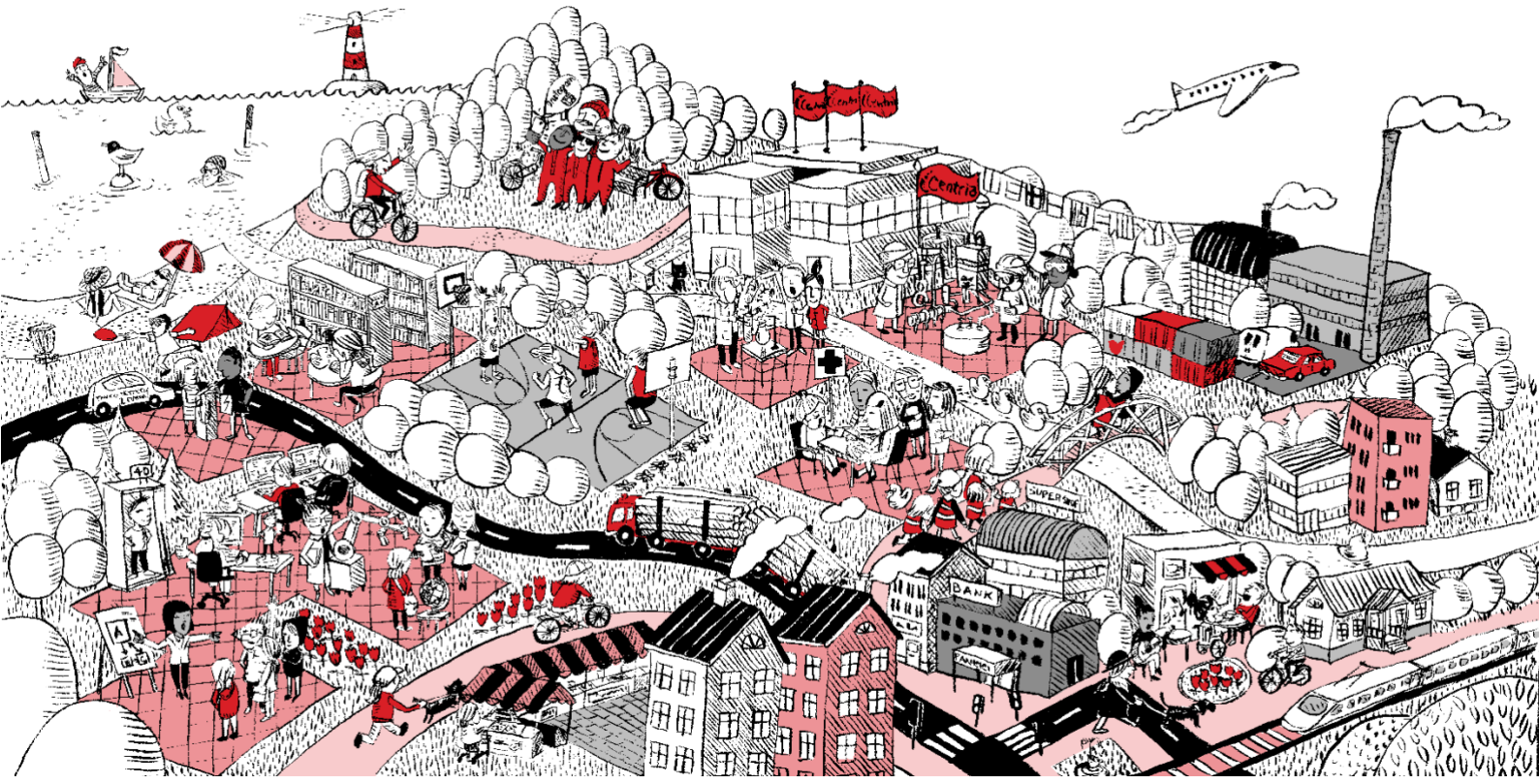
Carl-Oscar Ahlsved

EMBEDDED SYSTEMS

Project report

CENTRIA UNIVERSITY OF APPLIED SCIENCES
Information technology
April 2022



CONTENTS

1 INTRODUCTION

This project report is about a smarter engine block heater controller created for the course embedded systems. The idea for this project is that traditional engine block heaters might waste electricity by having power on for unnecessarily long when the outside temperature is not that cold. This seemed like an interesting project to do and would include a fair number of challenges as well as different technologies.

This report does not go into detail on all small technicalities as the writer already has some experience with electronics and embedded systems. Therefore, it focuses more on higher-level things than exact pinouts and details. When doing the project a project diary was written while working, to help remember what was done as well as document problems and challenges. This report has a more theoretical part first containing different parts of the project, followed by a more informal recap of the project diary where the actual process is described.

2 PROJECT PLAN

The aim of the project was to make a “smarter” car block heater controller. Typically, a timer is used for this purpose that is set a few hours before departure to warm up the engine. The controller in this case should instead use a departure time and adjust how long the block heater is on according to outside temperature.

The initial plan was to implement basic functionality first and foremost and then add more depending on time availability. Optional extras could be using a wireless temperature sensor instead of wired one, using a camera for car recognition or building a physical UI parts. (LCD etc)

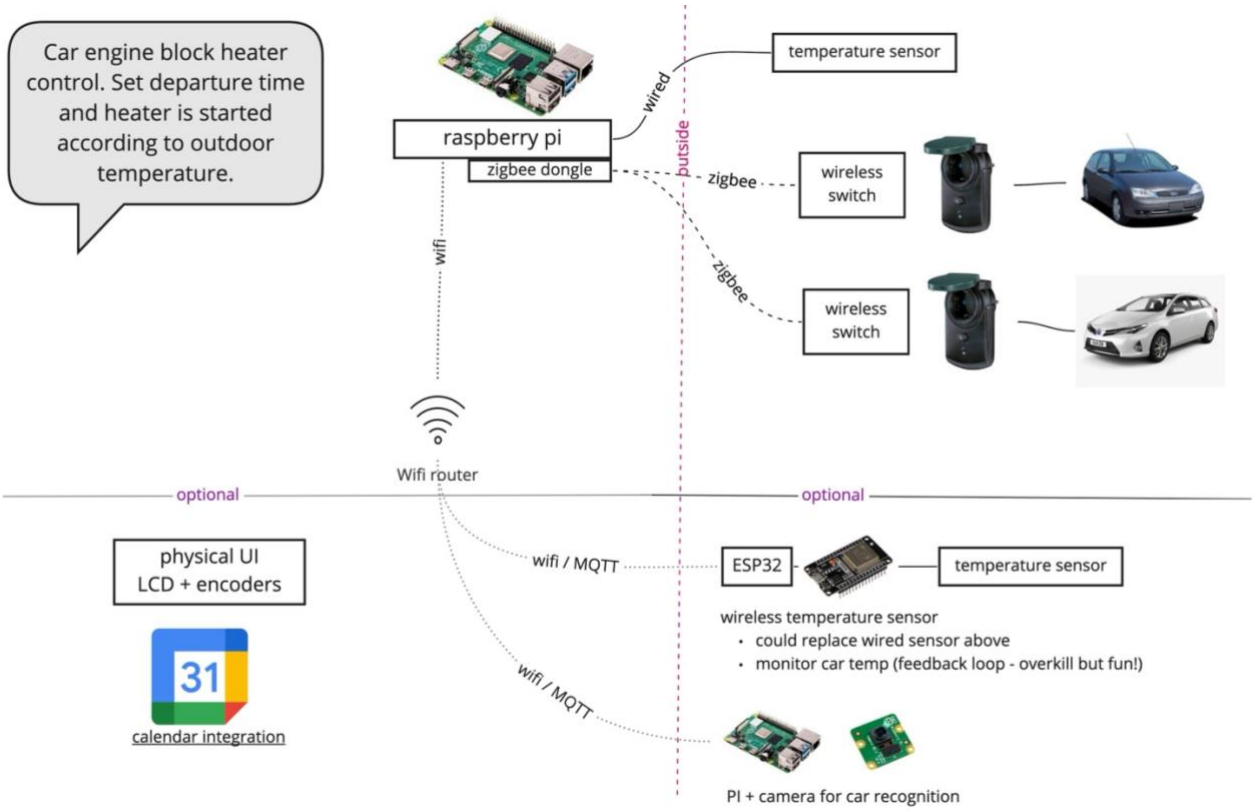


Figure 1 - Initial project plan

3 OVERVIEW OF THE PROJECT

This section will in short present the project with its hardware and software parts. Later some of the parts will be covered in more detail. Not everything in the initial project plan was finished, although that was also not really the plan either. The basic functionality of starting a wireless switch depending on departure time and the outside temperature was achieved. Furthermore, a wireless sensor was implemented as well as an LCD that shows the time, temperature, and status. The part totally omitted was the camera recognition part.

3.1 Conceptual overview

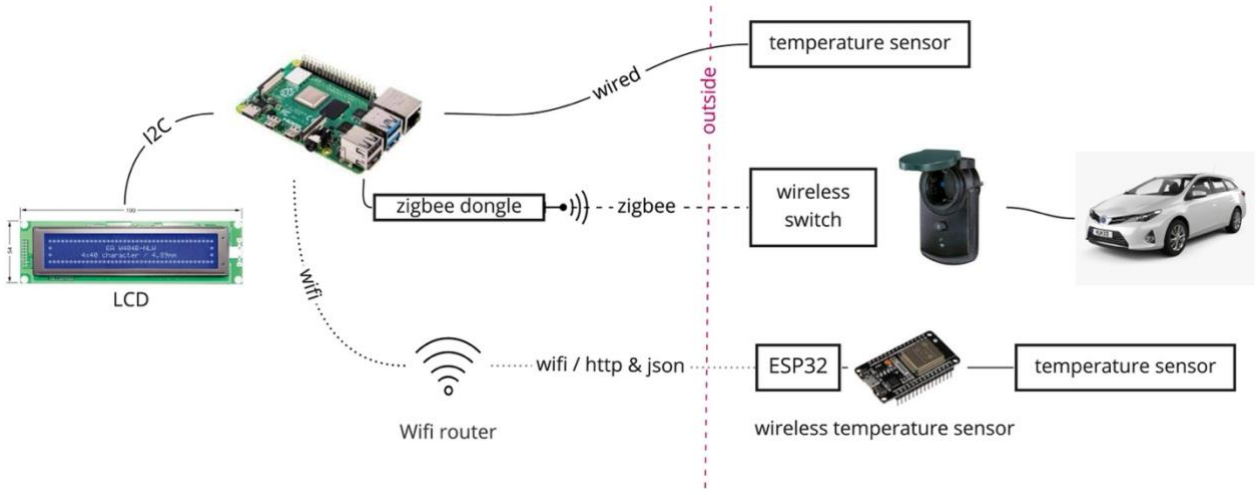


Figure 2 - Overview of implemented parts

3.2 Hardware overview

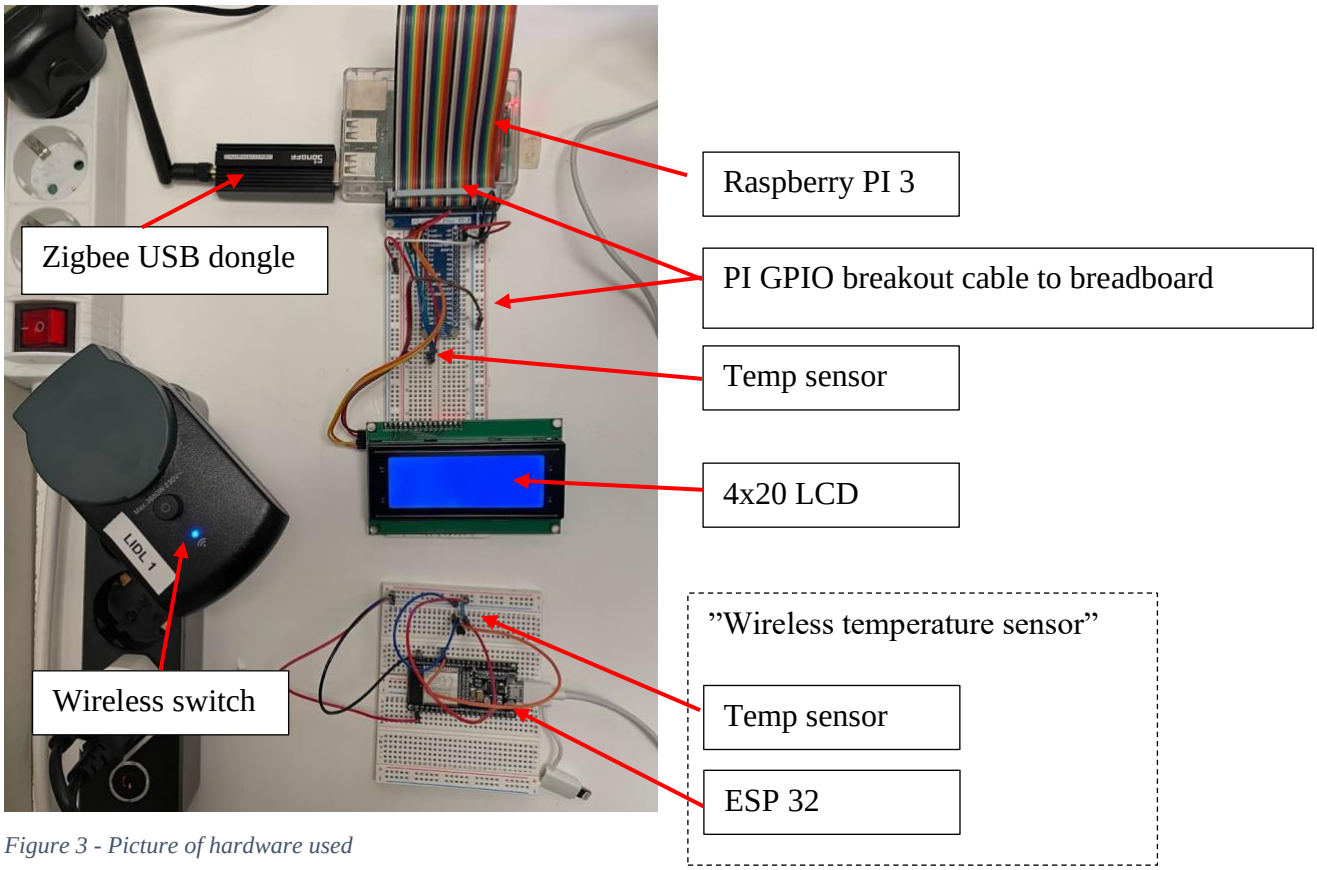


Figure 3 - Picture of hardware used

3.3 Software overview

The language used was C and C++ and that is because the main code was based on an earlier project for a C-programming course. (authors own code though!) That project was a simple block heater program, but only implemented in software that ran on a desktop PC. By using some ready code there was more time to focus on the overall system. Not all software/code solutions mentioned in the project diary were implemented in the main code due to time restraints but were tested in separate files.

This report will not go into detail on the programming and the main code is available on github with plenty of comments. Git version control was used for this project, but there were hardcoded passwords in the files so the commits are not visible as a new repository had to be made where passwords had been removed.

The figure to the right is the overall flowchart of the main code, where it can be seen that most revolve around a continuous loop. For testing purposes, there is an “emulation” mode that runs time much faster than in real-time.

The system reads departure times from a text file into an array, and this file is regularly re-read. The format used for time is seconds since midnight to make the heater logic easier.

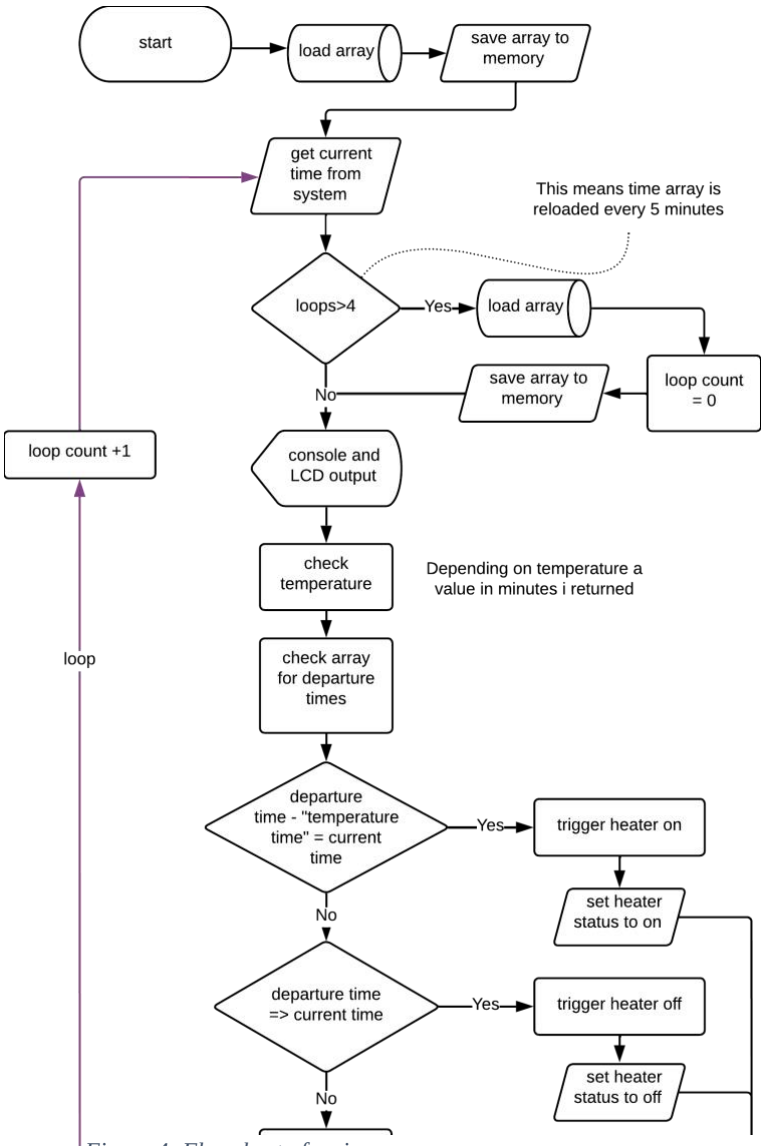


Figure 4- Flowchart of main program

As noted earlier not all code were implemented in the main program, but the figure below represents how the libraries and code are connected to the different parts and hardware.

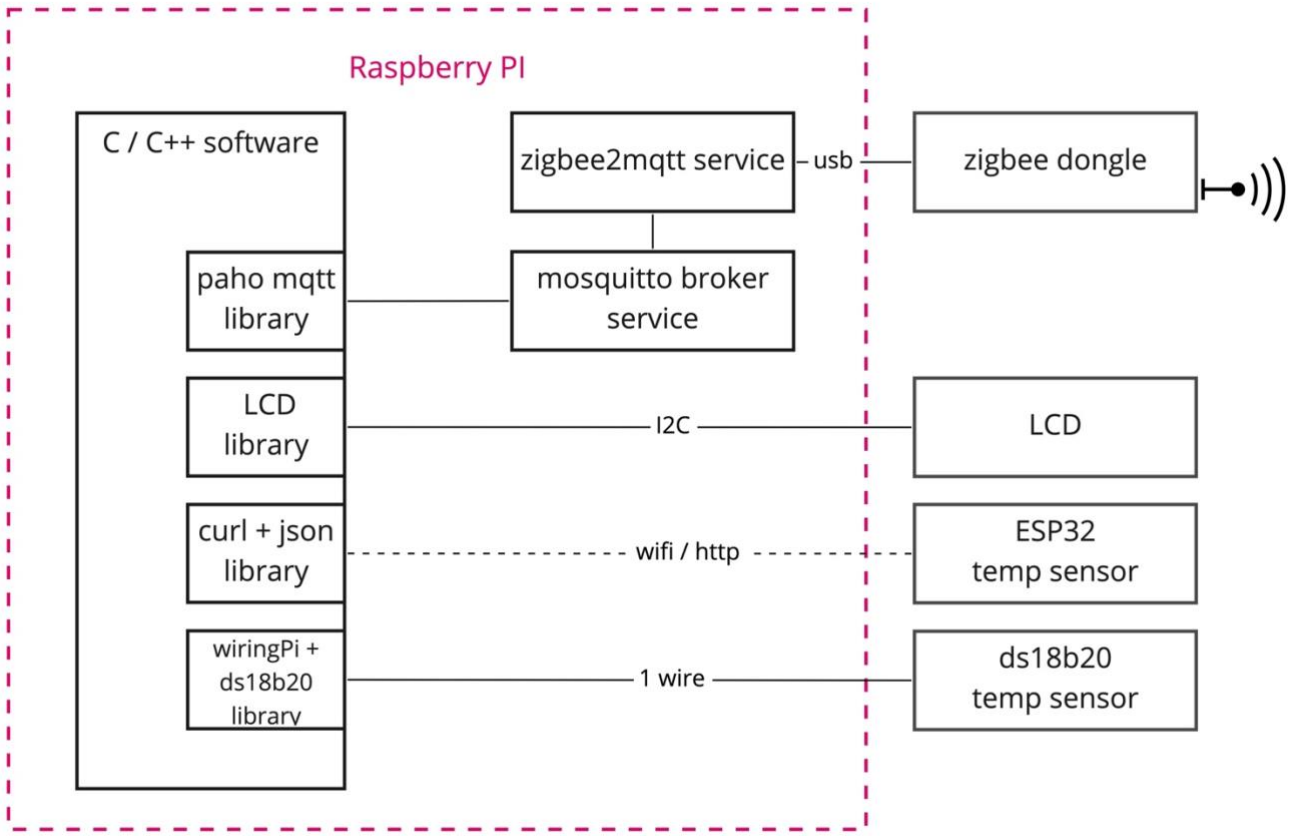


Figure 5 - Overview of software "parts"

Github address:

https://github.com/ahlsvedc/embedded-systems-report

Main code:

embedded-systems-report/block-heater-report-code/block-heater-2000-deamon-v2.cpp

4 DETAILS OF KEY COMPONENTS AND CONCEPTS

As already stated earlier this report does not go into details with everything, but this chapter does describe some of the key components and concepts a bit more as they are not properly described in the project diary part.

4.1 Raspberry PI

As the Raspberry Pi is already included in the course material it is assumed to be a known device that does not need much further explanation. The Raspberry Pi device mainly used was the Pi 3B even though some tests were carried out also with a Pi4. The raspbian 32-bit operating system was used.

4.2 ESP32

The ESP32 is a low power SoC (system on chip) microcontroller from the Chinese company Espressif. It has integrated Wi-Fi and Bluetooth and has several variations on the microprocessor used. The processor is available both in single core and dual core variations and operates up to 240 MHz. The ESP32 has a host of GPIO pins for different purposes, including I2C, SPI, and analogue IO. (Espressif 2022).

It is truly versatile and explaining all the functionality is out of the scope for this report. The version used in this project has a dual core processor and 4 MB of storage and 320 KiB of RAM. It was used as the heart of the wireless temperature sensor due to its wireless capabilities.

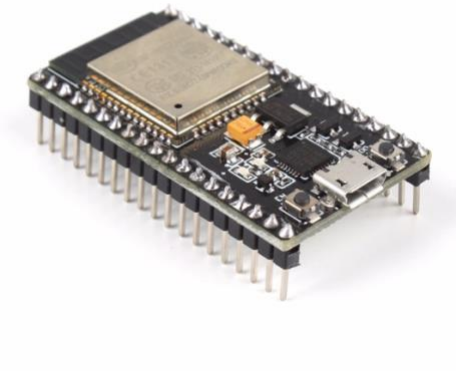


Figure 6 - ESP32

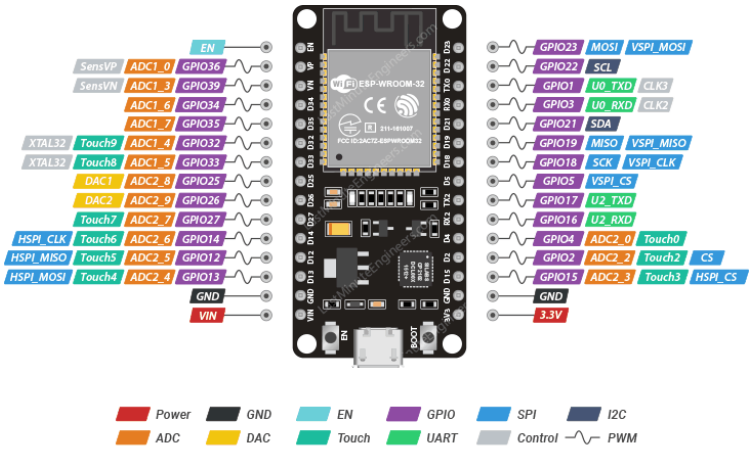
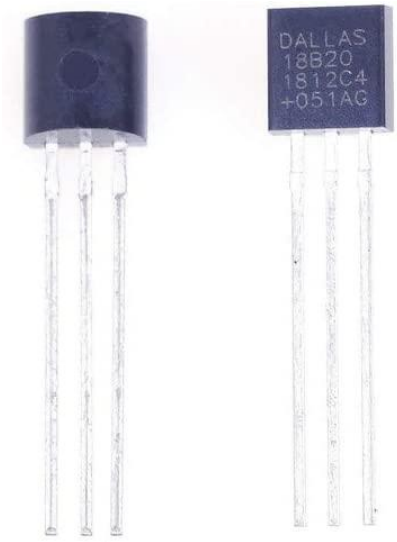


Figure 7 - ESP32 GPIO functionality

4.3 DS18B20 Digital Temperature Sensor

The DS18B20 is a digital thermometer chip from Maxim. It is quite popular and libraries and code can readily be found for many platforms. It uses a 1-wire interface and is very simple to connect. (Maxim 2019.) It is used for both the wired and wireless temperature sensor in this project.



- Unique 1-Wire® Interface Requires Only One Port Pin for Communication
- Reduce Component Count with Integrated Temperature Sensor and EEPROM
  - Measures Temperatures from -55°C to +125°C (-67°F to +257°F)
  - ±0.5°C Accuracy from -10°C to +85°C
  - Programmable Resolution from 9 Bits to 12 Bits
  - No External Components Required
- Parasitic Power Mode Requires Only 2 Pins for Operation (DQ and GND)
- Simplifies Distributed Temperature-Sensing Applications with Multidrop Capability
  - Each Device Has a Unique 64-Bit Serial Code Stored in On-Board ROM
- Flexible User-Definable Nonvolatile (NV) Alarm Settings with Alarm Search Command Identifies Devices with Temperatures Outside Programmed Limits
- Available in 8-Pin SO (150 mils), 8-Pin µSOP, and 3-Pin TO-92 Packages

Figure 8 - Data from spec sheet

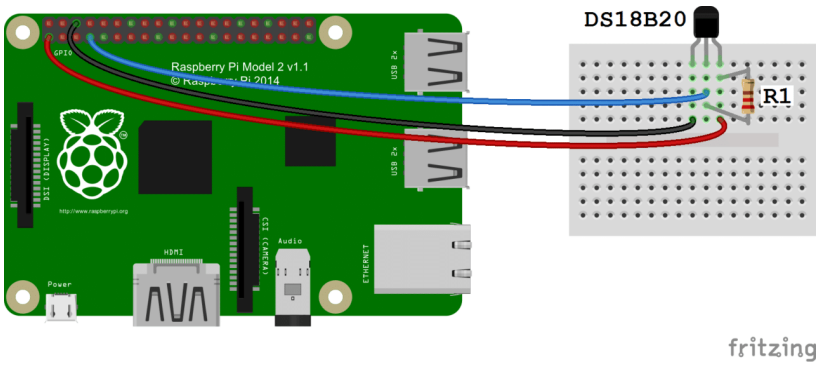


Figure 9 - How to connect from https://www.circuitbasics.com/raspberry-pi-ds18b20-temperature-sensor-tutorial/

4.4 MQTT

MQTT stands for Message Queuing Telemetry Transport protocol and is a lightweight messaging protocol that is very popular in the IoT world. It is extensively used for telemetry of sensor data of IoT devices to back-end services in the cloud, but has a range of other uses too. It is regarded to be very stable due to different QoS levels. MQTT is a publisher/subscriber type of technology. It requires a MQTT broker as a middleman that relays messages between devices. A device could subscribe to a certain topic and when another device publishes data on that topic to the broker, the broker then relays that message to devices subscribing to that same topic. (MQTT 2022.)

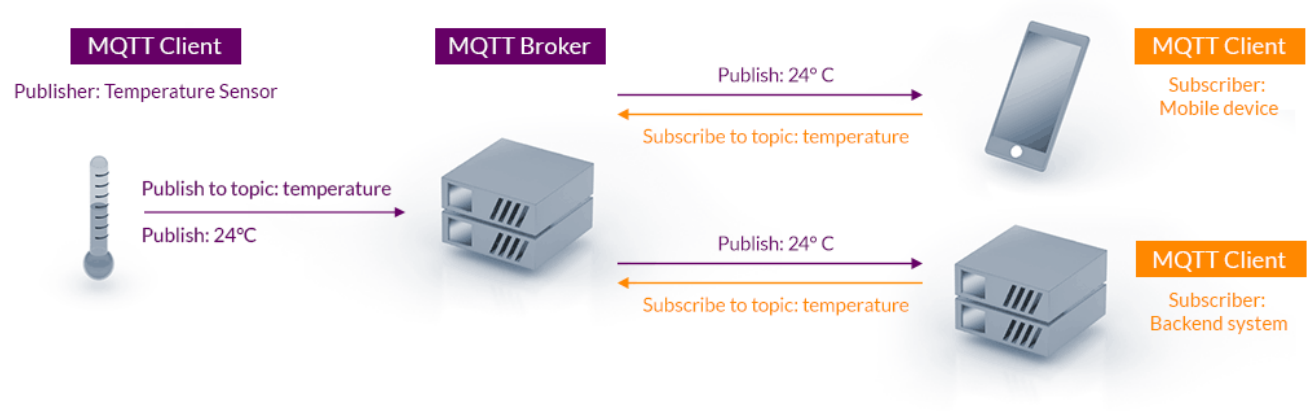


Figure 10 - publisher - broker - subscriber

The broker used in this project was mosquitto, which is a popular open source broker. For this project the guide at: https://randomnerdtutorials.com/how-to-install-mosquitto-broker-on-raspberry-pi/ was used to set it up. The whole point of using MQTT for this project was to be able to both send and receive messages.

4.5 ZigBee

Zigbee is a wireless close proximity mesh network, which means the nodes in the network may relay messages in between them. It can also operate as a point-to-point network. It requires very low power and can as such even run on batteries. It mainly operates on the unlicenced 2.4 GHz band. (CSA 2022.)

It is very popular in the home automation market and used for things like ”smart bulbs” (Ikea trådfri range for example) In this project Zigbee is used to control the outdoor power plug that is feeding power to the block heater. For that a Zigbee “dongle” is needed on the raspberry to send data as well as a Zigbee controlled wireless power socket. To do this a software called Zigbee2MQTT is also used to bridge between MQTT and Zigbee. The wireless switch is paired to the dongle through this software and makes it possible to assign a MQTT topic for it. This can be done with an easy-to-use GUI.

Figure 11 - ZigBee dongles used in the project. Above "noname" based on cheap CC2531 chipset. Below proper name brand Sonoff based on CC2652P chipset.

Figure 12 - LIDL Zigbee based outdoor plug

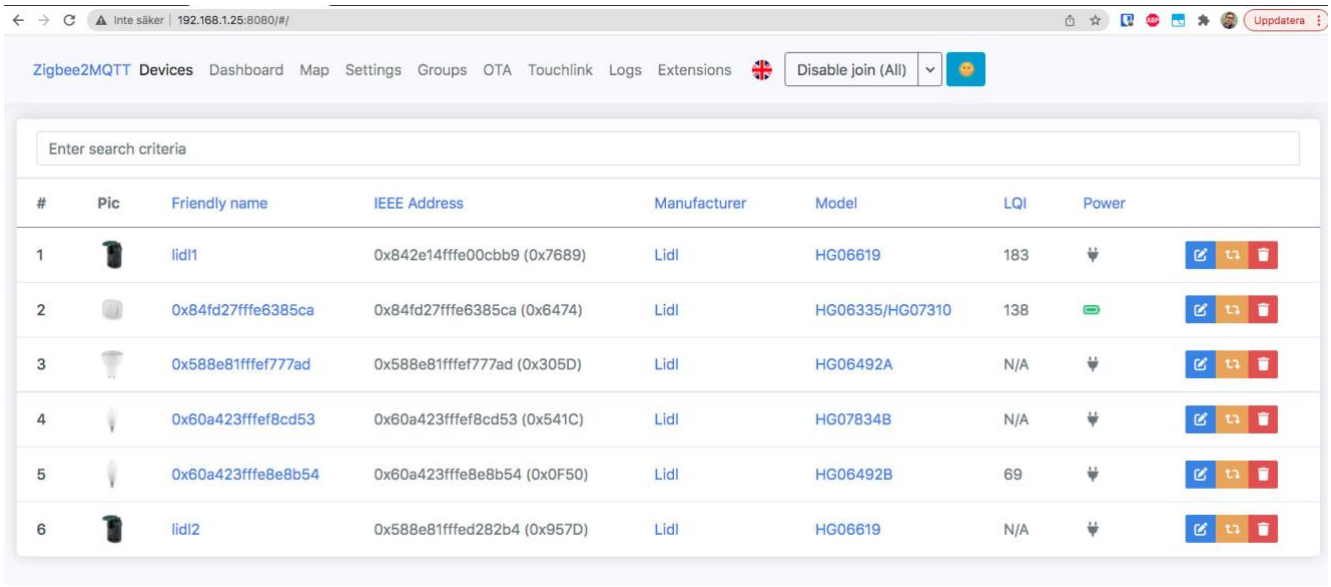


Figure 13 - Zigbee2MQTT GUI

4.6 LCD

The LCD used in this project was a very basic 20x4 LCD based on the HD44780 controller but with a I2C "backpack" attached so it can be connected with the I2C protocol. (https://uk.beta-layout.com/download/rk/RK-10290_410.pdf) These 20x4 LCDs actually behave like a 40x2 LCD when it comes to displaying data on the screen which at first was a bit confusing until the chart below was used. It was connected using this resource: https://tutorials-raspberrypi.com/control-a-raspberry-pi-hd44780-lcd-display-via-i2c/

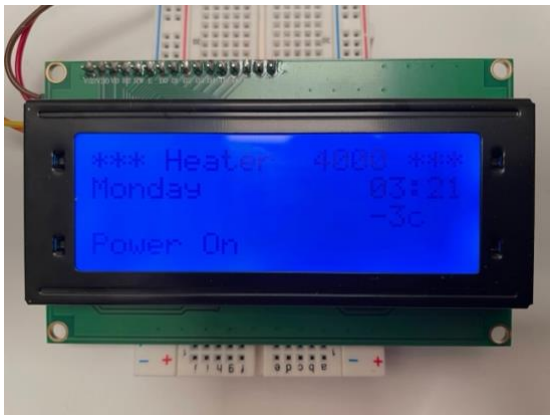


Figure 14 – LCD in action

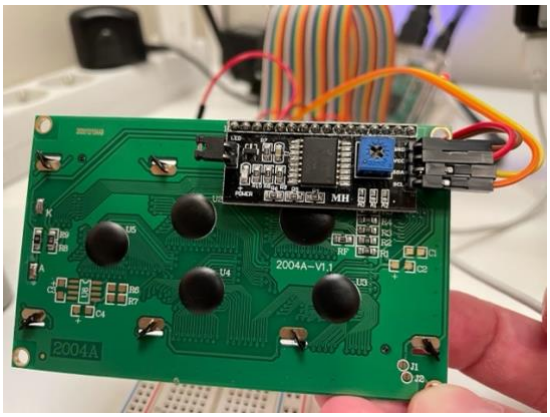


Figure 15- LCD I2C "BackPack"soldered on

Display Character Address Code

Display Position	1	2	3	4	5	6	7	8	9	10
DD RAM Address - Row 1	80	81	82	83	84	85	86	87	88	89
DD RAM Address - Row 2	C0	C1	C2	C3	C4	C5	C6	C7	C8	C9
DD RAM Address - Row 3	94	95	96	97	98	99	9A	9B	9C	9D
DD RAM Address - Row 4	D4	D5	D6	D7	D8	D9	DA	DB	DC	DD

Display Position	11	12	13	14	15	16	17	18	19	20
DD RAM Address - Row 1	8A	8B	8C	8D	8E	8F	90	91	92	93
DD RAM Address - Row 2	CA	CB	CC	CD	CE	CF	D0	D1	D2	D3
DD RAM Address - Row 3	9E	9F	A0	A1	A2	A3	A4	A5	A6	A7
DD RAM Address - Row 4	DE	DF	E0	E1	E2	E3	E4	E5	E6	E7

Figure 16 - Address chart of LCD

5 DEVELOPMENT ENVIRONMENT

This section will briefly showcase the development environment for this project. Primarily visual studio code was used, but with a few extensions. Firstly, the remote SSH extension by Microsoft. This extension allows you to use any remote machine with a SSH server as your development environment. This approach simplifies development and troubleshooting greatly. For the Raspberry Pi it was great as you did the compiling on the Pi itself very easily, while still being able to use the real estate of desktop PC monitors etc. It was like working locally even though everything happened on the Raspberry Pi itself. Things like version control with Git also worked very well. The software could then also easily be run in the terminal window in VS code. For the compile process the arguments needed to link the libraries could be added in configuration files.

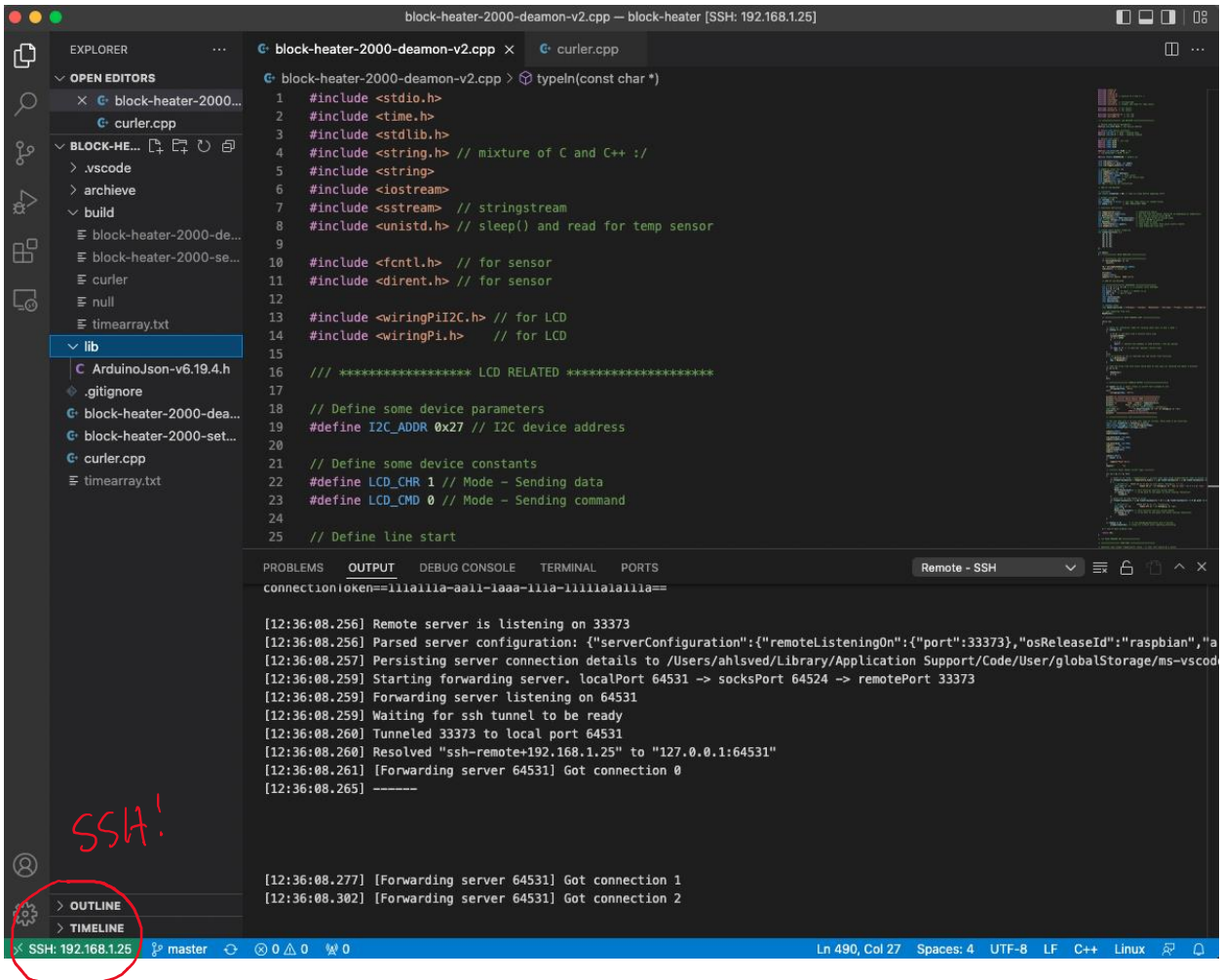


Figure 17 - SSH extension running in VS code. Looks like normal besides the SSH status in the bottom.

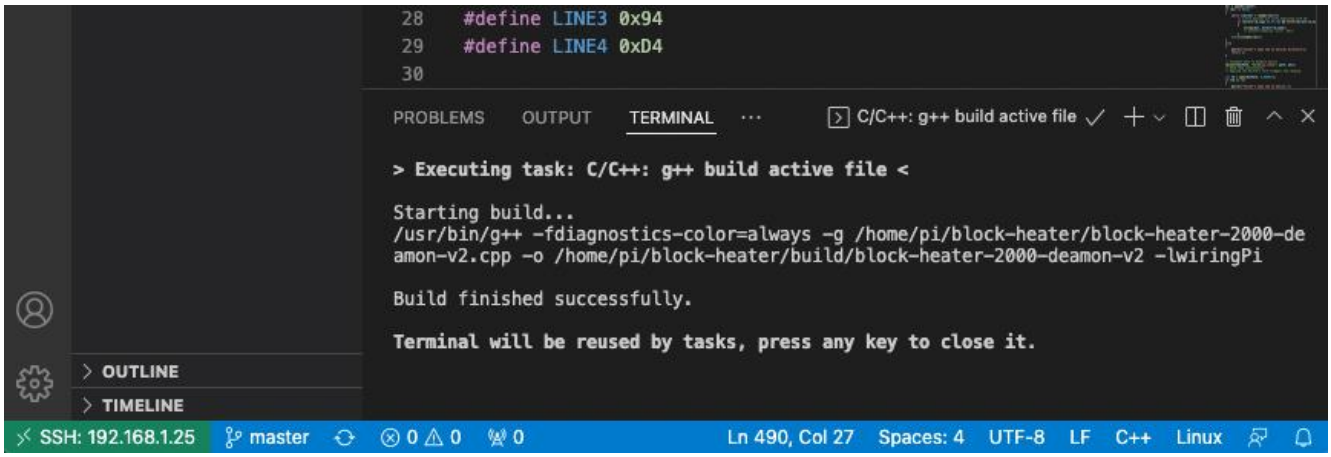


Figure 18 - Build task running on the Pi itself

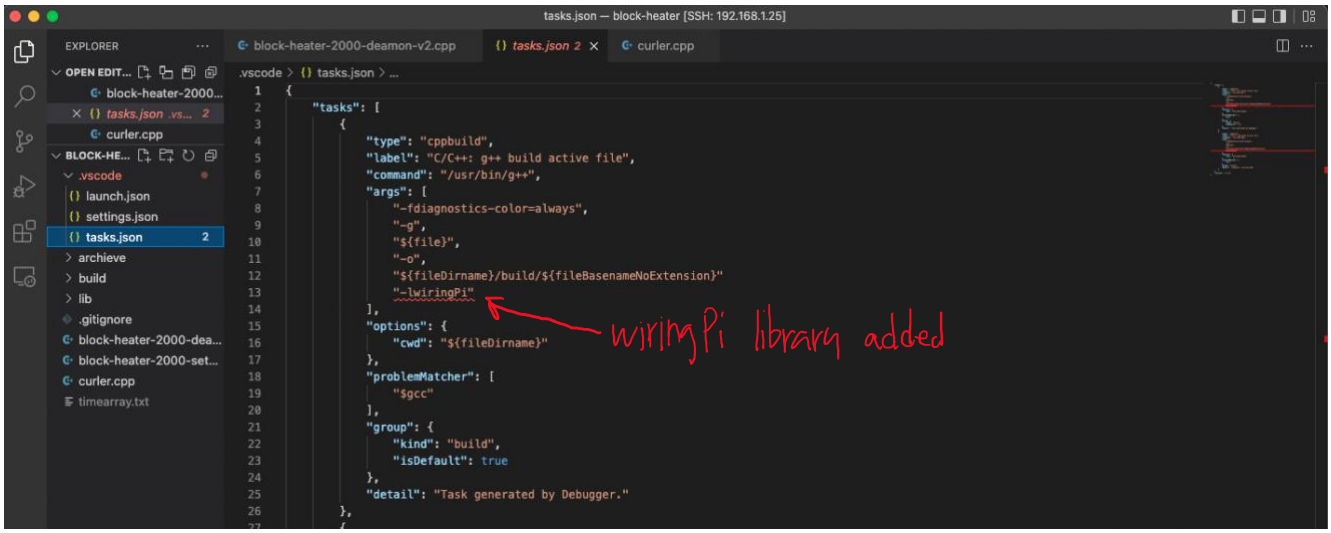


Figure 19 - tasks.json - file to specify compile arguments in VS code. Here the various libraries used could be added.

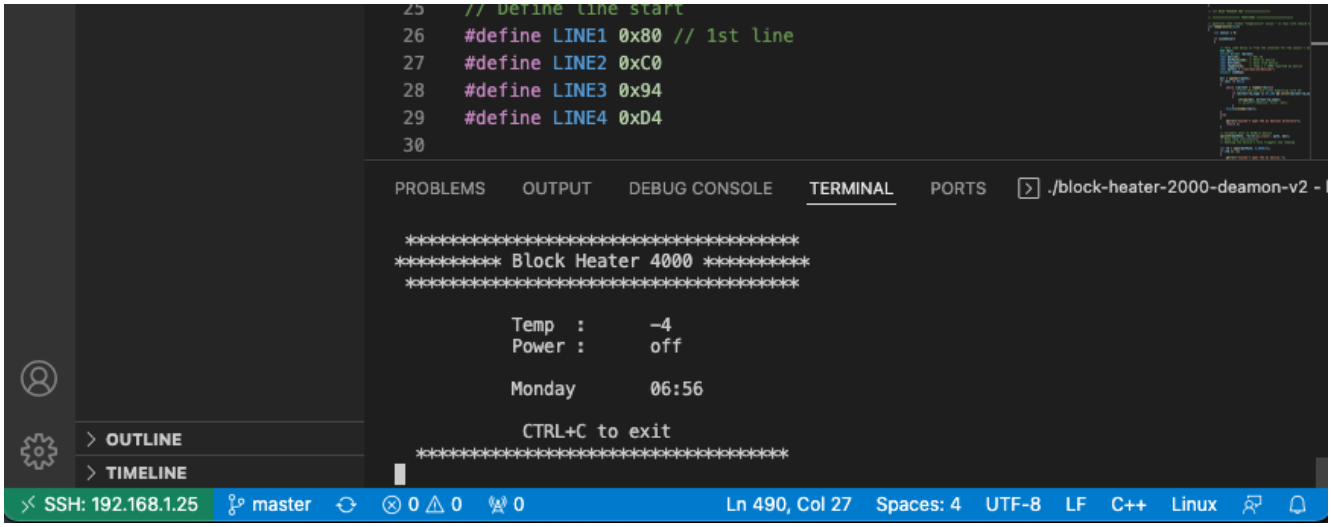


Figure 20 - the code running on Pi

The other extension used was platformIO which is intended for embedded use. It offers more modern build tools than the Arduino IDE and libraries can be managed on a per project basis instead of being installed globally. When starting a new project from the platformIO home screen the necessary folder structure is automatically created. There was not too much time to play around with it during this project, but it looks very promising. It also has a serial console and other tools like easy one-click upload of code to the device.

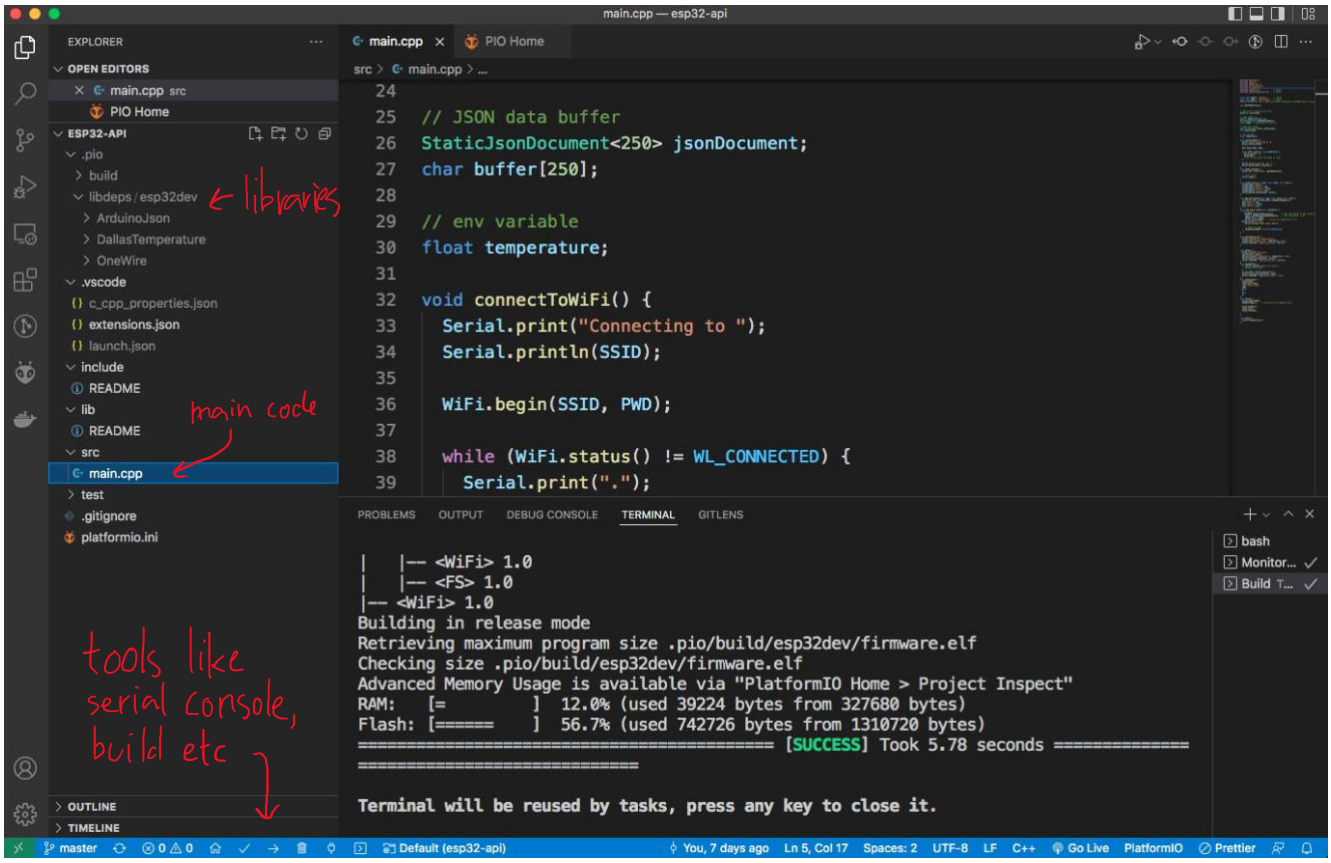


Figure 21 - platformIO

6 PROJECT DIARY

The project diary follows a chronological order to depict some of the major challenges and phases in the project. It again focuses more on the process than on technical details and is very informally written. There is no code but that can as earlier stated be found on github. Initially the project was more hardware focused but in the later stages the software side took more time and effort.

6.1 LCD testing

The project started with me testing some of the parts that I already had lying around to see what could be useful. Tested a bunch of OLED displays as well as a 40x4 LCD display. I had earlier bought a "backpack" for the LCD to use it with I2C which I wanted to test. These were all tested with an Arduino though as it was easy to find libraries that worked right away and the voltage levels were right. The use of level converters between 3.3 and 5v were something I stumbled upon at this time. In my project I actually later used the 3.3V from the PI to drive the LCD for this reason.

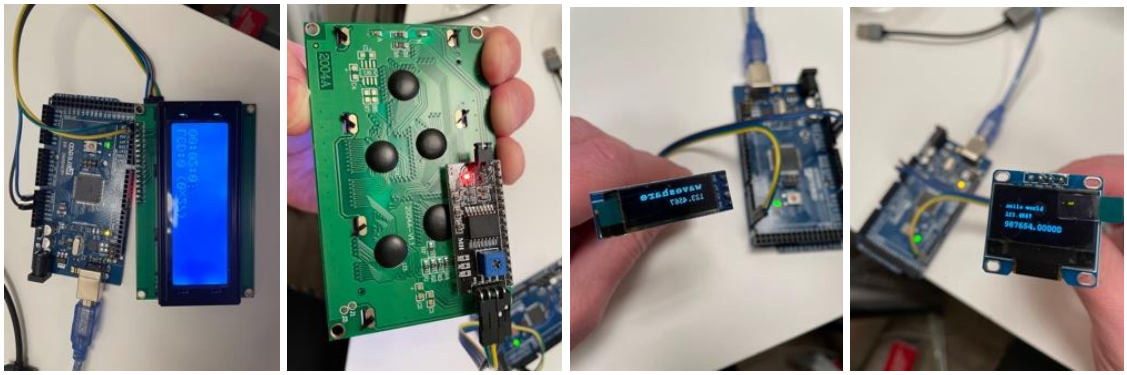


Figure 22- Some LCD being tested

The I2C "backpack" for the LCD worked fine but one could notice that it was slightly slower than connecting all wires, which was to be expected.

6.2 Early software adventures

I had at this point done a fresh install of Raspbian on the raspberry pi and was ready to test some software. The project was partly based on some previous c code I did for another course. The next step was to try to revive some parts of it. It was written on a mac with an intel processor so it would not run "as is" on the PI or my new M1 macbook. I got some segmentation faults and but after some debugging I got it running. Some of the faults were found in *char string arrays not behaving nicely on the Pi.

At this point I got a bit frustrated in trying to debug things on the raspberry Pi so I experimented with using a SSH extension for visual studio code. It made life much easier to be able to remotely code and build on the Pi itself. I also made a git repository for the code.

6.3 First temperature sensor attempts

I installed a library called wiringPi which is quite popular (though deprecated) to interface with an analogue temperature sensor. I had previously used analogue sensors with Arduino successfully. It was at this point I realized that there were no analogue inputs on the PI itself. I had a dht11 sensor from an Arduino kit, but I could not get it working on the PI although it worked when connected to an Arduino. One issue I read about was that on the PI the readings are polled directly on the GPIO which is very hard to do as other processes running on the PI might disrupt the timings. (it runs a full OS) There were workarounds but the dht11 does not measure below zero anyway, so I had to order another sensor in any case.

6.4 More hardware implementations

I had previously ordered a cheap ZigBee dongle that was popular in the home automation field that should work with the software I had planned to use. At this point I had been playing around with a software called "home assistant" that runs on the PI and verified that the dongle worked with my lidl power plugs that I bought. In the process I also started exchanging many of the lights at home as I found the technology so interesting, much to the joy of my wife. Now I also installed the mosquitto MQTT broker as well as the zigbee2mqtt software and got all that working. Cheap is not always durable though, because after a while the dongle suddenly died and I had to buy another (proper one).

As I now had no dongle I turned my attention to the wired temperature sensor again as it had arrived. This time I had more luck to get a temperature reading on the PI although it still took a while. It felt kind of “hacky” since as you read the sensor directly from a directory, but hey it works!

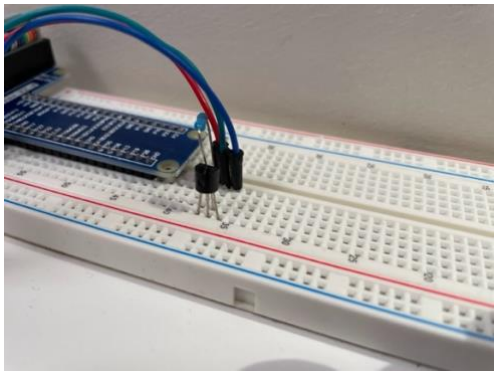


Figure 23 – Temperature sensor on breadboard

Figure 24- new Sonoff ZigBee dongle connected

6.5 More software

As I now had temperature readings I wanted to get the control of the power outlet to work which was more of a software problem at this point. I really struggled with trying different MQTT libraries and to get them to work on the raspberry pi. It was quite a steep learning curve as I had not needed to compile and link external libraries with C before like this. As I already had the mosquitto broker installed and I had learned how to send MQTT commands directly in the command line I did a hacky implementation of sending system commands from C itself. Not recommended I guess but at least I had a working solution. (in the end what is actually used in the main code) At this stage I also tried to refactor some of the spaghetti code and trying to make the build process better. Also experimented with external class files, but that did not go too well. Overall, I now had some kind of running software that could measure temperature and control the power socket by sending MQTT messages.

6.6 LCD

After all the software troubles I felt I wanted to focus on hardware. A quite simple thing to do was to connect an LCD as I already had done prior testing, or so I though. I again had problems with libraries and was a real hassle to get I2C to work on the PI. Quite quickly found examples for python that I got working so I knew that the hardware was ok, but for C/C++ it was much harder. I eventually found a library that I managed to get to work.
(https://github.com/Szpillmann/LCD1602A/blob/master/LCD1602a.CPP)

Now added the code to the project only to find out that the library used C and did not like C++ strings. The project was initially written in C and then I had converted some parts like some strings to C++ implementation… only to again have to do a fix to the fix.

Figure 25 - Having some LCD issues

Figure 26- Some real characters showing up. Here trying to learn how to properly place text

Figure 27 - Finding out I2C address of LCD

I mostly tested the software running in a ”debug mode” as waiting for time to pass is quite boring. One issue that started more and more started to be apparent that some kind of threading would be good. It took quite long to poll the temperature sensor so I just used a random generator to generate temperature readings.

6.7 Software and libraries again

I had not had much success with compiling my software and linking with various MQTT C / C++ libraries so far. At this point I had done a bit more research and finally got a MQTT library called paho to compile. An example of the compile command below:

g++ -DOPENSSL -I.. -I/usr/local/include -D_NDEBUG -Wall -std=c++11 -O2 -o async_publish async_publish.cpp -L../../lib -L/usr/local/lib -lpaho-mqttpp3 -lpaho-mqtt3a -pthread

For someone new like myself the learning curve to get to this point is very steep. When searching for answers many assume that you know how to write compiler commands like this and what it all means. This also made me start looking into cmake and makefiles to further streamline the build process, but I felt had not the time to fully dive into this topic at this point.

Happy actually being able to experiment with sending real MQTT I hit the next problem. The software I used for MQTT relies heavily on JSON. My solution at this time was to just use an online tool to get JSON "payloads" strings. Now by using the code examples included in the library I succeeded to send messages to myself.

The big problem now was that while it is quite easy to send a message. How to do if you want to receive a message? I was wanting to have a wireless temperature sensor and that was the whole point with MQTT in the first place, to able to both send and receive. The sensor would send MQTT messages about its temperature to my software. The problem is that you can't have the software wait for messages as that would block all other operations. For this I started experimenting with threading and reading about interprocess communication. This is for sure the road to take but it has another steep learning curve. I wanted to have something that I could do right now for the project, so I did another solution.

6.8 ESP32 with REST API

As it is way easier to read data from somewhere as threading is not absolutely necessary, I decided to make a REST API instead running on the ESP32. The net is full of examples for the ESP32 which is the controller I wanted to use as it has Wi-Fi built in (and I had a bunch of them). I merged two different projects I found on the net. One for REST API on the ESP32 and one regarding the temperature sensor I was using. https://microcontrollerslab.com/esp32-rest-api-web-server-get-post-postman/ and https://randomnerdtutorials.com/esp8266-ds18b20-temperature-sensor-web-server-with-arduino-ide/. It actually was refreshingly simple to work with the ESP32 after all the struggles with the raspberry Pi. Was much more examples and no need for complicated build commands when using the Arduino IDE. For this part it was more or less to combine the two projects, and not much problem solving was required.

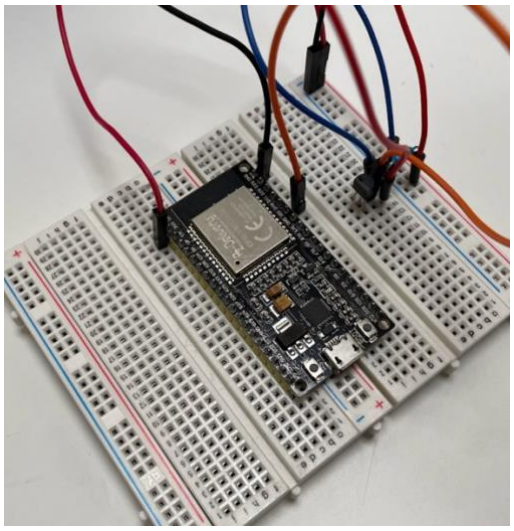


Figure 28 - ESP32 and DS18B20 sensor on breadboard

```
18:43:54.458 -> ??????.....Connected. IP: 192.168.1.104
18:43:57.079 -> 24.62°C
18:43:57.117 -> Read sensor data
18:44:57.149 -> 24.56°C
18:44:57.149 -> Read sensor data
18:46:01.704 -> 24.50°C
18:46:01.704 -> Read sensor data
18:47:02.442 -> 24.50°C
18:47:02.442 -> Read sensor data
18:47:13.448 -> Get temperature
18:48:06.420 -> 24.50°C
```

Figure 29 – Serial console output.

Figure 30- JSON data in web browser

One problem with the Arduino IDE was that it took quite some time to compile the code. I had previously checked out something called platformIO that could also be used as an extension in visual studio code. PlatformIO has a much more modern build system and did only compile parts that changed which meant that the first compile was 30s but the later ones where 5s, whereas the Arduino IDE always kept slow. Maybe the Arduino IDE can perform much better, but I was happy with platformIO. It was not as forgiving as the Arduino IDE though and wanted proper C code as having function declarations at the beginning of the code.

Inspired by the speed increase I wanted to see if using a PI4 instead of a 3+ would be worth it. I cloned the SD card to a SSD and booted the PI4 from that. On the PI4 the compile time was 3s instead of 5s so I switched back to the PI3 as it felt that the 2s was not worth it. For bigger projects it probably would make a bigger difference.

Figure 31 - PI4 with attached USB SSD

If the ESP32 part was easy it took more effort to have the raspberry pi code working. To access the ESP32 web server I used a popular library called CURL (https://niranjanmalviya.wordpress.com/2018/06/23/get-json-data-using-curl/) It was really hard to find a JSON library for C / C++ that was simple and easy to use for a situation like this. In the end I stumbled upon an Arduino library called ArduinoJson. They mentioned that it should compile on other platforms as well, and yes it did. This is something I really have to look into, as it seems many of the C libraries for Arduino are simpler to use than their desktop counterparts.

At this point it was very late in the project and I did not integrate all my findings into my main program but tested the concepts using smaller test programs, as the “curler.cpp” test program console output shown below.

```
^C
pi@raspberrypi:~/paho.mqtt.cpp $ g++ curler.cpp -o curler -lcurl
pi@raspberrypi:~/paho.mqtt.cpp $ ./curler
{"type":"temperature","value":24.625,"unit":"°C"}
```

Figure 32 - Compiling test program and fetching JSON data from ESP32

```
pi@raspberrypi:~/paho.mqtt.cpp $ g++ curler.cpp -o curler -lcurl
pi@raspberrypi:~/paho.mqtt.cpp $ ./curler
24.625
pi@raspberrypi:~/paho.mqtt.cpp $
```

Figure 33 - Running async in a thread and getting raw value from JSON

7 CONCLUSIONS

When the project was started there were ambitious goals on where it was supposed to end. Even though it did not get there, still a lot was learned, especially on the software side. It would have been interesting to work with image recognition and OpenCV, and some research was even started for it. In the end though, I it was better to focus on the core parts of the system instead.

There was not an exact order for how the project plan would be executed, and in the end, a lot of external factors affected that anyway. Things like parts breaking, forcing a change of focus. As the author already had experience with embedded systems and electronics the hardware part of the project was not that hard, but the software took much more time. The main issue was to link external libraries when compiling. The problem probably was a combination of a lack of skills in the area and that most resources assumed that this knowledge already was known. Furthermore, there were not too many libraries available for C / C++, but there were plenty available for some other platforms like python.

If I this kind of project would be repeated, researching platform support before starting definitely would be done, and python would probably been the better choice for this kind of project. The reason for sticking to C / C++ was the already written code. That code had to be modified quite extensively though, and in the end, it would probably have been easier to rewrite everything from scratch again to fit the project better and get rid of all the spaghetti code. Preferably, learning to do it not only in a procedural way, but to use object-oriented programming. (the PI is more or less like a PC anyway) It is also a bit confusing for a beginner to mix C and C++ in the same project.

For the build process and linking of all libraries, it would be good to learn to use CMake or makefiles to speed up the compiling process and maybe learn to cross compile. When it comes to the coding the next step for the project really would be to master the use of threading and interprocess communication. This would enable to make some kind of service that would handle the MQTT part of the software to be able to receive messages as well. Although basic Arduinos are still single core, the ESP32 used for this project has two cores and does multithreading as well, so learning skills like these absolutely makes sense for the future as well. Learning interrupts, to be able to interact with the code directly from the hardware would also be interesting. Like having a button that would instantly turn on the power for a few hours.

REFERENCES

The references really got messy. Half of them are in the text itself due to the informal style of text used in this report.

CSA 2022. *ZigBee*. Available: https://csa-iot.org/all-solutions/zigbee/

Espressif 2022. *ESP32*. Availible: https://www.espressif.com/en/products/socs/esp32

Maxim 2019. *Programmable Resolution 1-Wire Digital Thermometer*. Available: https://datasheets.maximintegrated.com/en/ds/DS18B20.pdf.

MQTT 2022. *MQTT: The Standard for IoT Messaging.* Availible: https://mqtt.org/