Design Project

Blinking Lights Unit

Design Report

Authors:
Daniël van Andel - *s2520729*
Tim Brands - *2986876*
Anamaria Ceban - *s2800705*
Tieme van Enkhuizen - *s2576945*
Mariska Frelier - *s2494388*
Wander Stribos - *s2398443*

Supervisors:
Roland van Rijskwijk-Deij
Bernd Meijerink
Rick Fontein

November 9, 2025

UNIVERSITY OF TWENTE.

Contents

1 Introduction

OpenINTEL

OpenINTEL is an internet measurement initiative run by the DACS (Design and Analysis of Communication Systems) research group at the University of Twente. DACS' mission is to contribute to the development of the trusted and resilient internet that is needed by our society. The first step in doing this is to detect anomalies and attacks [1]. This is where OpenINTEL comes in. The goal of OpenINTEL is to measure, gather, analyse, and share large-scale data on the global Domain Name System (DNS) ecosystem. Every day, approximately 308 million domains are measured, resulting in 5 billion data points every day. These measurements start at midnight (Coordinated Universal Time) and end somewhere at the end of the afternoon or the start of the evening (usually). This program has been running and gathering data since 2015 [2].

The project

The goal of this project is to create a dashboard which displays the progress of the daily measurements of OpenINTEL. We have designed several modules that each display a different aspect of the progress of the data collection. At the end of this project, we have designed and implemented a dashboard which contains five different modules, all modularly designed and controlled by their own microcontroller (Arduino Nano). These five modules are all controlled by a small computer (Raspberry Pi 5), which queries the OpenINTEL database and calculates aspects of the data collection. This data is then converted to a form that the microcontrollers can receive, which is then sent towards them over a custom communication protocol. The microcontrollers then make sure that the incoming data is displayed correctly on their respective modules.

The report

In this document, you will find how we, as a project group, progressed through the different phases leading up to our final product: an operational dashboard. We started in the **Requirement Phase**, during which we interviewed our supervisors to get a clear idea of the functional requirements. After this, we progressed to the **Design Phase**, in which we presented several ideas to the DACS team to get an idea of what they had in mind for the dashboard's design. We then entered the **Implementation Phase**, which was also the **Testing Phase**, as we were required to implement both software and hardware simultaneously. Our progress through these phases is explained in this document.

At the end of the document, we have attached the following appendices:

- **Appendix A: Documentation Per Module** — An in depth technical explanation of how a module is wired, what software is running on each module, how the communication protocol between the computers and the microcontrollers work, and more.
- **Appendix B: Planning** — A clear overview of the planning we used to implement this project.
- **Appendix C: Meetings** — A short summary of the different meetings we had with our supervisors.

2 Requirement Phase

The first meeting we had with DACS was an introductory meeting to get familiar with the system we were going to design a product for: OpenIntel. We were shown what the system does and how it works, and its goal every day. After this first meeting, we got to work and set up a list of requirements, which we discussed during the following meeting. For this, we followed the MoSCoW prioritisation method; this list can be found below. Since we would have weekly meetings with the supervisor wherein we would also lay down our future plans, we decided not to formulate any *Won't*-requirements.

- Must
  - The system shall visualise the data collection, more specifically, the progress thereof, of the DACS research group on DNS.
  - The data will be visualised on an A1-sized mainframe with different components.
  - The system will consist of multiple modules, which are detailed underneath.
  - There will be one main module, which provides the power and data for the other modules.
  - The system will be connected to OpenIntel.
  - The system will have an overall good-looking front-end (the physical board)
- Should
  - The board should be at most 5 cm thick, this 5 cm is a sum of the mainframe, the modules & the module components themselves.
  - All the modules should function independently
- Could
  - The system should not cost a large amount of power.
  - We want to be able to hide specific data, specifically some ccTLDs that are scanned.
  - We want to be able to add new data to the board.
  - The system would preferably be connected to OpenIntel by Ethernet.

Additionally, we came up with multiple modules to (possibly) include, which we also decided to prioritise following the MoSCoW system.

- Must
  - The main module will have a reset button to be used when the system freezes and needs to be rebooted, which will be executed in an aesthetically pleasing manner.
  - The progress until the scan is expected to finish will be shown in a manner that appears to be live.
  - The current expected end time of the scan will be continually shown.
- Should

  - The expected end times of each different worker group should be visible.
  - The number of queries that have been completed will be shown, in such a way that it appears to update live.
  - The system should have an alarm indicator for when the OpenIntel anomaly alarm triggers, suggesting that the client take action.
  - The status of the different worker nodes should be shown.
- Could
  - A world map could be included, where the progress per country code top-level domain is shown.
  - The result of random DNS queries could be shown on the board.

3 Design Phase

In this section of the document, we will describe the various design challenges we encountered and the choices made. The main goal of this phase was to conceive various visual designs and low-fi prototypes. We also faced several hardware-related challenges during this phase.

3.1 Physical Attributes

The first design choice we faced was the physical backboard on which everything would eventually be placed, as we had to keep this in mind during the implementation of all future modules. In interviews with the clients, they expressed a desire for it to be a so-called “Blinking Lights Unit”, the kind of blinking LED panels in sci-fi movies that pretend to be complex technical dashboards and control panels.

We decided to divide the physical board into three layers:

1. **The Main Board** — A wooden panel with side panels to create space behind the board, which allows us to hide wires, power supply and the Raspberry Pi. This panel will contain small holes which allow the wiring of each separate module to go through.
2. **The Module Boxes** — 3D-printed boxes which can easily open and close for easy access for repairs or changes. They contain all the hardware of a module, with a small hole to allow the wiring from the hardware to the Arduino Nano to go through. Each box will be screwed into the main board.
3. **The Visible Hardware** — These are the parts of each module which display information. For example, the Predicted End Time Module Box has a hole in the front which fits the 40×8 matrix board exactly.

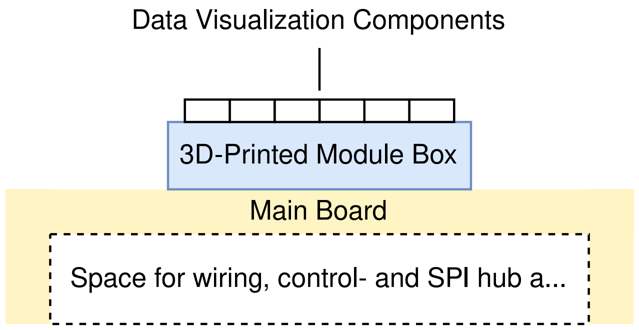

Data Visualization Components
3D-Printed Module Box
Main Board
Space for wiring, control- and SPI hub a...

Figure 1: Layered physical design of the dashboard.

All in all, these three layers together form a modular and visually layered dashboard.

3.2 Module Design

During the design phase of each module, we selected the hardware to be used, as ordering materials would take considerable time. We actively decided to use different visual components for each module to maximise diversity on our dashboard. For example, time is displayed both through 7-segment displays and through matrix boards.

Detailed technical information on every module can be found in Appendix E, and the final low-fidelity prototypes for each module are shown in Figure 2.

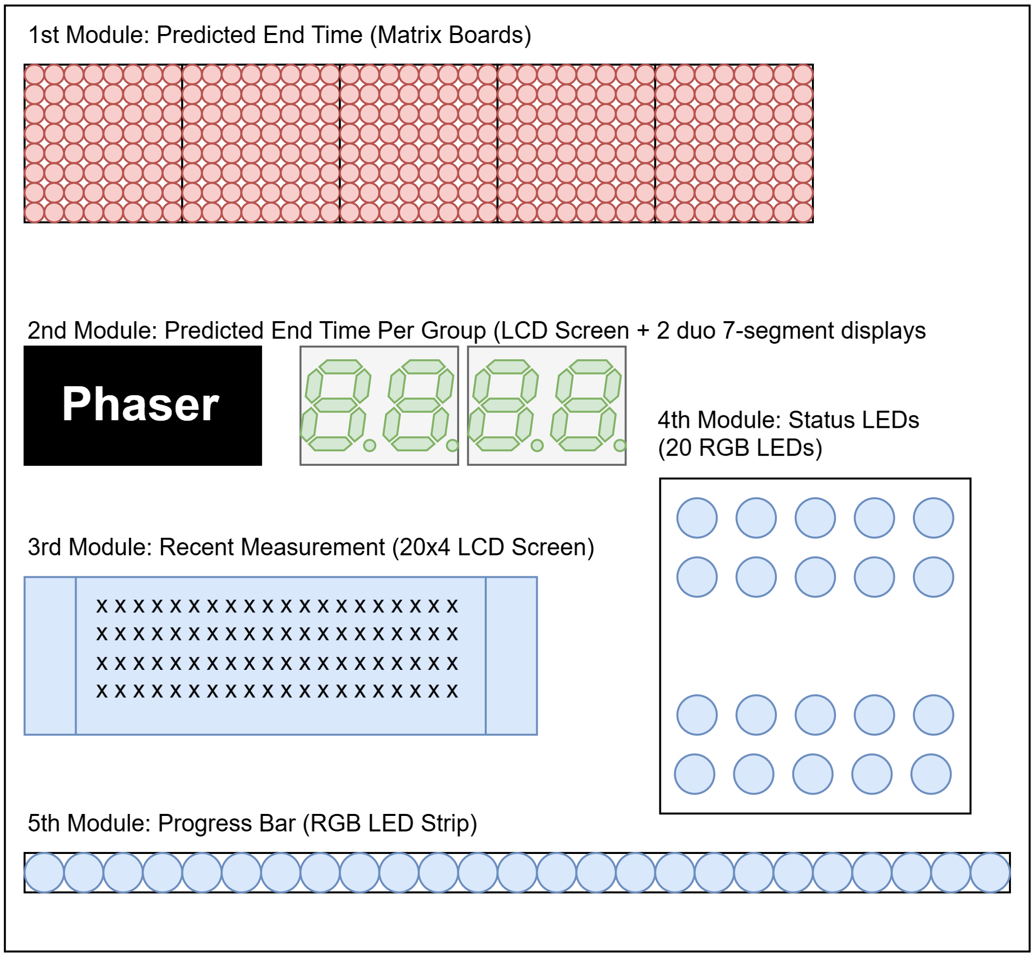

1st Module: Predicted End Time (Matrix Boards)
2nd Module: Predicted End Time Per Group (LCD Screen + 2 duo 7-segment displays
Phaser
4th Module: Status LEDs (20 RGB LEDs)
3rd Module: Recent Measurement (20x4 LCD Screen)
XXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXX
5th Module: Progress Bar (RGB LED Strip)

Figure 2: Low-fidelity prototypes of the individual modules.

3.3 System Design: Hardware

During the design phase, we had to decide on which microcontrollers and microprocessors to use. Initially, we debated whether to use multiple Raspberry Pis or only microcontrollers, such as ESP32Cs. Because one of the “Must Have” requirements was that the system should be as energy-efficient as possible, we investigated power usage and concluded that the most efficient way to implement the system was to use one Raspberry Pi, which would communicate to multiple Arduino Nanos that all controlled their own module.

Another issue we faced was the power supply, as we could not power all modules through a single Raspberry Pi 5, as it did not have enough power throughput (The LEDs would have to run at a fraction of their brightness). We estimated the power requirements for each module and explored supply options. During the peer feedback meetings, we received a suggestion to use a USB power hub, which we subsequently adopted. This way, the Raspberry Pi is powered separately (since it requires more power than the hub can deliver), while the other modules are powered through the USB hub.

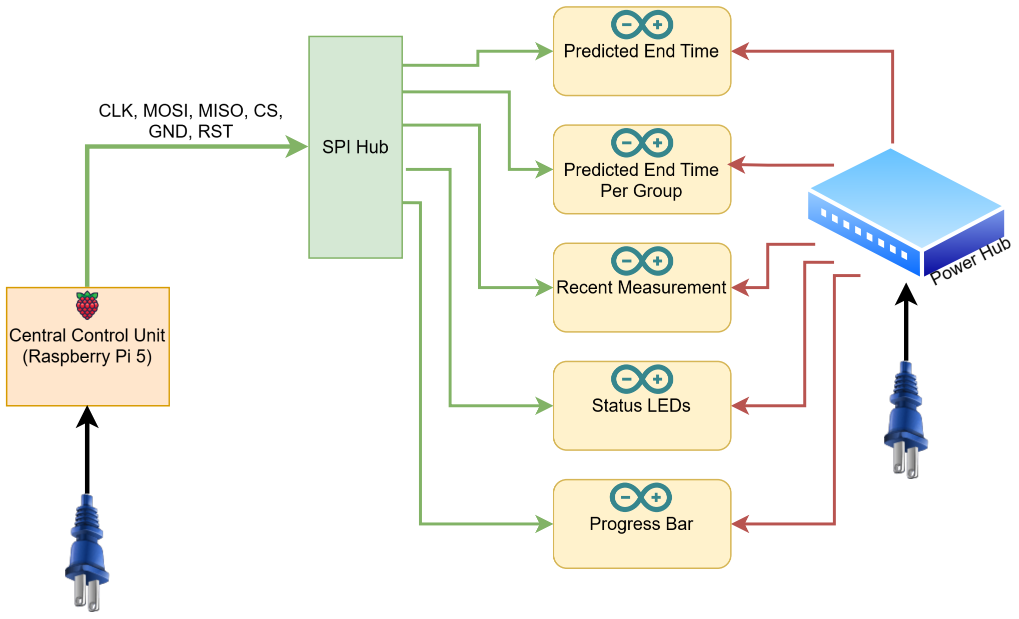

CLK, MOSI, MISO, CS, GND, RST
SPI Hub
Predicted End Time
Predicted End Time Per Group
Recent Measurement
Status LEDs
Progress Bar
Central Control Unit (Raspberry Pi 5)
Power Hub

Figure 3: System design showing the microcontroller and microprocessor layout.

3.4 System Design: Software

After we had figured out how the system would be built, we were required to investigate how the Raspberry Pi would communicate its instructions to the various modules. To enable a single Raspberry Pi to instruct several Arduino Nanos, we constructed a custom SPI communication protocol.

Initially, we wanted to maximise modularity by having each Arduino Nano register itself with the Raspberry Pi. However, an Arduino Nano only contained one SPI chip. Since some of them needed to use this chip to control their hardware, they could not send information back to the Raspberry Pi. More information on this bidirectional protocol can be found in Section 8.2.

We ended up designing a system where a JSON file on the Raspberry Pi defines which Arduino Nano is connected to which pin, allowing the Raspberry Pi to send instructions to each individual module.

4 Risk analysis

4.1 Data to get

After the design phase, we assessed the risks that could occur. As the system is only informative, the primary risk is the connection to the OpenINTEL system. First, their system makes queries to DNS servers every day, which should not be influenced in any way. To make sure this does not happen, all the tokens for the database are read-only. However, our system still has access to the database. This means that, when our system is compromised, an attacker could get access to all the data. Because of this, none of the raw data is stored in the database, except for pure metadata, such as the number of queries performed. The only exception is the random queries, but as they are already displayed on the board, this is not a problem.

4.2 Ways to get in

To get into the system, there are four main ways, which will be discussed below:

ssh

Because the Raspberry Pi is behind the board, it is not easy to access physically. As a result, we enabled SSH. During testing, password authentication is enabled. However, when the board is hung in the hallway, it will be disabled, and the board will only accept public key authentication.

Our code

Our own code could also be an entry point. Because the code needs to be able to shut down and reboot the Pi, it was given root access. This means that if our program gets compromised, it can have major consequences. However, we do not estimate this to be likely. Our program does not have any open ports; it only asks for data from the database. There is also no input from the user, except for the reboot-or-shutdown button.

Physical access

The main risk for our project is physical access. The Raspberry Pi is behind the board. It is not visible, but if one checks, it is possible to get either the whole Raspberry Pi or the SD card. This would allow access to all the tokens or to load different code onto the Raspberry Pi. Because fixing this will make the board less accessible, which was one of the key points, we decided to accept this risk.

Git

To make updating easy, the Raspberry Pi automatically pulls code from Git. To do this, it needs to store credentials, as they cannot be asked from the user. Because these credentials could be compromised through the methods above, we decided to put a read-only token on the Pi. This means that even when the token is compromised, the code on git cannot be changed. The risk of this is that when a bug or other issue is pushed to Git, it would automatically be put on the Pi. Since this risk is known to the users, who will manage the access to the git and are experienced in writing secure code, we deem this risk to be acceptable.

5 Implementation

5.1 Modules

The design group was divided into three smaller groups of two people, where each group would work on a part of the project, two of which would work on modules. This meant that two modules could be developed at the same time. This can also be seen in Appendix A. What will become immediately clear is that initially, way more modules were planned than eventually delivered. It became clear quite quickly that we underestimated the time it would take to work on certain modules. We subsequently did not have the time to implement all the modules we would have liked to and were forced to scrap some quite early on.

A related problem is that we lacked sufficient knowledge about working with hardware and how parts would fit together. For example, we made the mistake of getting drivers for common *anode* LEDs, but getting common *cathode* RGB LEDs for the "worker group status" module. We managed to resolve this issue by using the same drivers that drive the seven-segment displays in other modules, of which we still had a few left. This did have the effect of slightly altering how the module displays the information, as we no longer could adjust the brightness per module. However, with the limited number of different states to be displayed, simply using Red, Green, and Blue, and blinking versions of the three gave us sufficient different options.

Other than that, the implementation of the modules went according to design, and both hardware (once figured out) and the primary functionality of the software were implemented relatively according to plan.

5.2 Hardware

Aside from the change in module hardware named above, and the choice being made to use a powered USB hub as a power supply as mentioned in Section 3.3, most of the surrounding hardware, which is the choice of controllers and connecting the Raspberry Pi to the controllers, was done according to design; there was only one issue that turned up which required a slight change. With the SPI protocol to have the Pi communicate with the Arduinos, the reset line, and the shared ground, there was a large amount of wiring required for each module. We had built an SPI hub which split the wires over the system in a smaller form for earlier testing; However, for the final product, we needed to make the wires longer to be able to stretch across the entirety of the final board. Here, we noticed that the hub ceased to function, and modules no longer received any data, even though every cable was correctly connected. We soon figured out that this was due to interference on the slave/chip select pins, something that none of us had expected, given our computer science background. After some troubleshooting, we installed resistors at the end of these wires nearest to the module controllers, after which the system worked reliably again.

5.3 Software

As stated already in Section 3.4, it was unfortunately not possible for us to implement a bidirectional communication protocol. Other than this, the implementation of the software went entirely according to the initial design. There were some hiccups with each controller correctly accepting the data sent by the Raspberry Pi, but these issues were resolved with only slight adjustments to the code.

5.4 Requirement evaluation

5.4.1 General requirements

- Must
  - ✓ The system shall visualise the data collection, more specifically, the progress thereof, of the DACS research group on DNS.
  - ∼ The data will be visualised on an A1-sized mainframe with different components.

    The board is 63 by 97 cm, slightly larger than A1 (59 by 84 cm).
  - ✓ The system will consist of multiple modules, which are detailed underneath.
  - ✓ There will be one main module, which provides the power and data for the other modules.
  - ✓ The system will be connected to OpenIntel.
  - ✓ The system will have an overall good-looking front-end (the physical board)
- Should
  - ✗ The board should be at most 5 cm thick; this 5 cm is a sum of the mainframe, the modules & the module components themselves.

    The backboard is 7 cm thick, as we needed enough space to hide the long wires. Additionally, the largest module (the status LEDs) protrudes another 6 cm, also to allow sufficient space for wiring.
  - ✓ All the modules should function independently
- Could
  - ∼ The system should not cost a large amount of power.

    All the implemented modules draw a maximum of around 10 Watts, and around 15 Watts for the Raspberry Pi (under full load), for a total of 25 Watts. In reality, all modules will draw less, as some LEDs had their brightness limited by software. A more in-depth calculation can be found in Appendix B.
  - ✗ We want to be able to hide specific data, specifically some ccTLDs that are scanned.

    The main reason for this requirement was the world map module, as we did not wish for all the ccTLD scans to be shown off too much. Since the module was scrapped, this feature was no longer necessary.
  - ✓ We want to be able to add new data to the board.
  - ✓ The system would preferably be connected to OpenIntel by Ethernet.

5.4.2 Module evaluation

- Must
  - ✓ The main module will have a reset button to be used when the system freezes and needs to be rebooted, which will be executed in an aesthetically pleasing manner.

- ✓ The progress until the scan is expected to finish will be shown in a manner that appears to be live.

  The module updates every five minutes. However, since the scan takes around 15-16 hours, the updates sent to the progress bar every five minutes differ by a mere $\sim 0.5\%$ *and appear to mostly be continuous.*

- ✓ The current expected end time of the scan will be continually shown.

  The expected end time is updated every five minutes.

- Should
  - ✓ The expected end times of each different worker group should be visible.
  - ✗ The number of queries that have been completed will be shown, in such a way that it appears to update live.

    This module was nearly implemented, but had to be cancelled due to what we expect to be problems within the purchased parts. The required parts and schematics are included in the appendix, and implementation should be relatively straightforward.

  - ∼ The system should have an alarm indicator for when the OpenIntel anomaly alarm triggers, suggesting the client to take action.

    Originally, this module was supposed to consist of a very clear signal beacon, which would flash when the anomaly detection of OpenINTEL went off. However, we were then told we would not get access to this system, and the plan switched to having it go off if the end time was estimated to be after the next scan would begin. However, due to time and material constraints, this module was never implemented.
    Nonetheless, while it is not as flashy as a genuine beacon light, the status LEDs give the same information: If the expected end time is too late, then at least one of the worker nodes would have a too-late end time and, thus, their corresponding status LED would be flashing bright red instead of their usual green.

  - ✓ The status of the different worker nodes should be shown.
- Could
  - ✗ A world map could be included where the progress per country code top-level domain is shown.

    Originally due to time constraints, this module was never implemented. However, since the size of the screen required would likely have a large effect on the power consumption, we would advise against the implementation of this module.

  - ✓ The result of random DNS queries could be shown on the board.

6 Testing Phase

6.1 Unit test main module

In the main module, there are two parts which we have deemed complex enough to require active unit tests.

6.1.1 Message builder for random query

The message builder constructs the messages which are sent to the display. Since the protocol allows lines at most 20 characters long, this function was developed to split up the IP addresses and names across multiple lines if necessary.

First, the default case is tested, where the message does not need to be split up over multiple messages. Afterwards, we have a range of test cases with different IP address lengths. These are made to simulate various edge cases and situations that are likely to go wrong, such as a length of exactly 20 characters. Finally, the same is done for the hostnames. This one also tests for too-long names, in which case the name sent will consist of '...' followed by the last 54 characters in the name, as anything more would exceed the allocated memory on the Pi.

6.1.2 Time converter

Another part that is likely to have bugs is the time conversion. This takes the UTC+0 time from the DACS database and converts it to local time. Some testing for this has been implemented. However, it is not possible to test it thoroughly. The main reason for this is that we lack proper documentation from the database, and therefore do not know what data we get in edge cases and, thus, what to test for.

6.2 Modules

For the modules, testing what they were supposed to do was simultaneously easy and, at times, quite difficult. Since we were working with hardware, and we could not get data back from the Arduinos to the main module, there was no formal software testing that could be done. Nevertheless, we have tried some methods to verify the functionality of all the modules.

6.2.1 Hardware

First, we had to make sure that all the connected hardware was working. For the simpler modules where everything is connected directly to the Arduino, this was quite easy, as the only points of failure were switching around wires or faulty parts. For the modules that have extra connections on a perfboard, this required checking the continuity of connections between relevant parts on the board, and checking whether resistances were correct with a multimeter. Once we verified that the hardware was working (or, in the case of problems, that it's not the issue), we could look at the software side.

6.2.2 Software

When testing the software, one half was easy. We would start out programming and working on the code on a "local" Arduino sketch. This means in this case that all the code related to receiving and processing data from the main module was not present, so we could quickly visually verify on the components that what we wanted to show was showing. This was done with dummy data.

Once we could verify the underlying software was working correctly, we could connect it to the main module and add the protocol code back into the Arduino file. In theory, with the protocol working correctly, putting the written code in the predefined spots in the protocol, a module should work fully with correct data handling as well. This was, of course, not always the case, but at this point, we could always rule out anything but an issue with how the communication protocol hands data to the rest of the program. From here, the only way to test the program was by opening a Serial connection to the Arduino and seeing what data it receives. This would then diagnose what the issue is.

7 Evaluation & Conclusion

7.1 Responsibilities

At the beginning of the project, the group was divided into three subgroups of two members each. Two groups focused on implementing the modules, and a third worked on the main module. Tasks were divided mostly according to personal interest, although their distribution remained flexible as new challenges occurred. The responsibilities assigned to each person were:

- **Anamaria:** Expected finish time, group status LEDs, design module boxes, 3D printing, board assembly, graphic design board, SPI hub, power hub
- **Daniël:** Progress bar module, time per group module (software), poster design, debugging and documentation
- **Mariska:** Main module, communication protocol, graphic design board
- **Tieme:** Expected finish time, group status LEDs, random DNS query, debugging and documentation
- **Tim:** Main module, communication protocol, database connection
- **Wander:** Progress bar module, time per group module (hardware), group status LEDs (soldering), soldering specialist

All team members contributed to writing the report, and since the project was a group effort, it must also be noted that everyone had a say in the important decisions made for any module, and whenever needed, all members helped with troubleshooting beyond their assigned modules.

7.2 Team Evaluation

From a teamwork perspective, we are content with how the collaboration developed throughout the project. Dividing the team into subgroups allowed us to work on multiple parts of the system simultaneously and gave each member clear ownership of their tasks. We held bi-weekly scrum-like meetings to discuss progress on the modules, identify obstacles, align priorities, and discuss new ideas. This approach supported accountability and transparency, while still giving each subgroup autonomy in their workflow.

As the project progressed, members naturally built up specialisations on parts of the project. By dividing new tasks in accordance with these specialisations, work speed improved as there was no more time lost on exploring the code environment or learning the related skills before being able to work on the tasks. However, toward the end of the project, a key drawback of this structure became apparent. While it was true that we were not losing time on learning the skill anymore, it also meant that these tasks were dependent on the "specialist". If this person were then absent or busy with another task, it meant that progress on related tasks would stall until they returned, or another member would have to spend time learn about what had to be done.

In terms of planning our working hours, we agreed to keep work within working hours on most days, which provided structure and helped maintain consistency. While punctuality became a little more relaxed near the end, communication remained, and we were always aware of each other's responsibilities and progress. Overall, the team dynamic was positive—cooperative, supportive, and adaptable when schedules or priorities shifted.

Looking back, one of our key takeaways is the importance of striking a balance between specialisation and shared knowledge. While focusing on individual expertise improved efficiency, ensuring overlap in understanding would have made the final integration and troubleshooting stages smoother.

7.3 Evaluation

Reflecting on the project process itself, several aspects of our planning and execution stand out, both strengths and points for improvement. While dividing the project into sub-teams effectively distributed the workload, we underestimated the interdependence between components. In particular, during the first weeks, the module sub-teams were unable to complete their implementations until the internal communication protocol was developed.

In hindsight, it would have been more efficient to prioritise getting the hardware installed, soldered, and *ready for instruction* before focusing on module logic. Once the communication template and protocol were finalised, module development accelerated significantly, taking only a few days per module rather than over a week. This meant that our initial planning of three days per module and eight for the final product turned out to be rather optimistic.

We also underestimated the time and effort required for building the physical modules. Several factors contributed to delays:

- The parts ordered online arrived a week later than expected.
- Some of the modules required soldering to reduce cable clutter. However, none of us had prior experience with soldering, which meant one of us had to learn the technique first.
- In general, the Arduinos and hardware were more finicky than we had hoped, which meant that many modules required significantly more troubleshooting than planned.

Despite early challenges, the process improved significantly toward the end. Our workflow became more structured, we met our milestones, and we successfully integrated the built modules. Our supervisors supported us throughout the project by meeting daily, providing consistent feedback, and assisting with hardware purchases.

7.4 Conclusion

As a team, we are very positive about how the product turned out and the "Best Project Award" we received at the Project Market. We faced many challenges combining different types of hardware and software into a physical, aesthetically pleasing, and operational board. We have learned a great deal during our Design Project, and we are very pleased that our client is also content with our product, which was, of course, the ultimate goal of this module.

8 Future use and work

In its current state, the product functions can be hung up in the Zilverling as a viable product. However, there are, of course, still some points of improvement which could be implemented in the future.

8.1 Power hub

Currently, all Arduinos are powered by a USB hub. This is a robust and simple way to power all the modules, but it also means that a separate power outlet is needed for both the USB hub and the Raspberry Pi. Besides that, the number of modules is also limited by the number of USB ports in the power hub.

It would be a good idea to look into a single power supply to power both systems and how to connect it properly. It might be possible to use only the Raspberry Pi power supply to power the modules; However, we are unsure if this would be safe for the Pi. This supply would also, preferably, power the hardware connected to the Arduinos, as some of them are currently outputting a large amount of power, which might shorten the lifespans of the Arduinos. Specifically, the Predicted End Time Per Group could benefit from this, as currently the seven-segment displays are not at full brightness, as we believe it might exceed the current capacity of some of the Arduino components.

8.2 Bidirectional protocol (with ESP32)

The Arduinos are currently unable to reliably send any data back to the Raspberry Pi. This is because Arduinos only have one SPI bus, and a lot of the components also use SPI. This means the Arduino can only receive from the Raspberry Pi (as a slave), and send data to the modules' hardware (as a master). Theoretically, the Arduinos should be able to switch their mode of operation and communicate pins through software. But we weren't able to get that to work. Currently, the Pi can only identify the Arduinos by following a hardcoded list of where and how each module is connected to the Pi. This problem can be solved by switching the Arduinos to ESP32s, which have two hardware SPI buses. This allows the ESP32s to both stay in contact with their respective hardware and exchange information with the Pi. This way, the Raspberry Pi can ask connected modules for an identification and dynamically keep track of which pins connect to which modules, regardless of the order in which the modules are connected. This feature has already been implemented on the Pi, but has been disabled.

8.3 Additional modules

Not all of the conceived modules were implemented. There is still some space on the board and wiring available on the Raspberry Pi and USB hub to connect additional modules, if the client would ever like to connect more modules. This should be relatively easy, as the system was designed to be modular, so that a module can be added and removed as easily as possible. Adding another module would mean that some of the decorations would need to be removed, since they might be in the way.

8.4 Sudo access

Because the code needs to be able to reboot the Raspberry Pi, it runs as the user 'dacs', who is a member of the sudo group. However, we later found out there might be options to reboot and shut down without this, which would allow the user to be removed from the sudo group. One possible drawback could be that it would not be able to reboot if the Pi is stuck on something. It is advised to do further research into whether or not it is possible to give the reboot and shutdown commands without sudo rights, and what the potential side effects of this would be.

8.5 Synchronise 5 minutes with database

OpenINTEL updates the database every 5 minutes, so we decided to update the board every 5 minutes as well. However, they are currently not synchronised. This means that the total delay of the board can be up to 10 minutes. Additionally, there is some time drift, as our system is slightly slower, resulting in some data being lost. This can be fixed in two ways:

If the five-minute timer of OpenINTEL is synchronised with the system clock, the timer of our board can be synchronised with the systemclock as well, which already fixes the drift. It might also be possible to time it immediately after the database update.

If this is not possible, it might be possible to check the time at which the database was updated, and schedule the next data collection a bit over 5 minutes later.

References

[1] DACS. *About DACS*. 2025. URL: `https://www.utwente.nl/en/eemcs/dacs/about/`.

[2] DACS. *OpenINTEL*. 2025. URL: `https://openintel.nl/`.

Appendices

A Planning

After investigating how we would implement the different modules in terms of hardware, software, and physical design, we divided our project group into three sub-teams. The first team focused on the main control unit and on developing a communication protocol for every module. The other two teams focused on implementing one module at a time. Once the modules were implemented, we merged together again as a single project group to finalise the implementation and collaboratively work on the overall design. The full planning can be found in the image below.

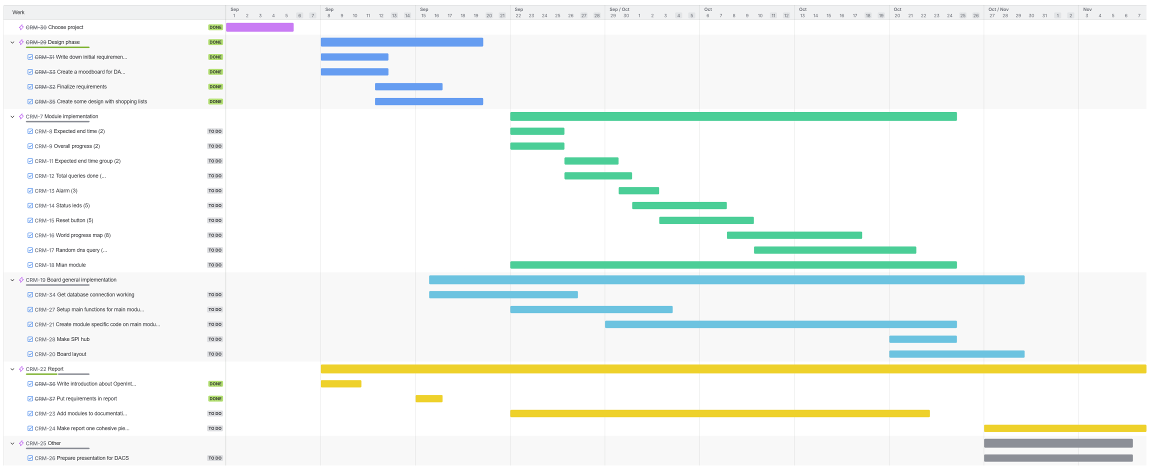
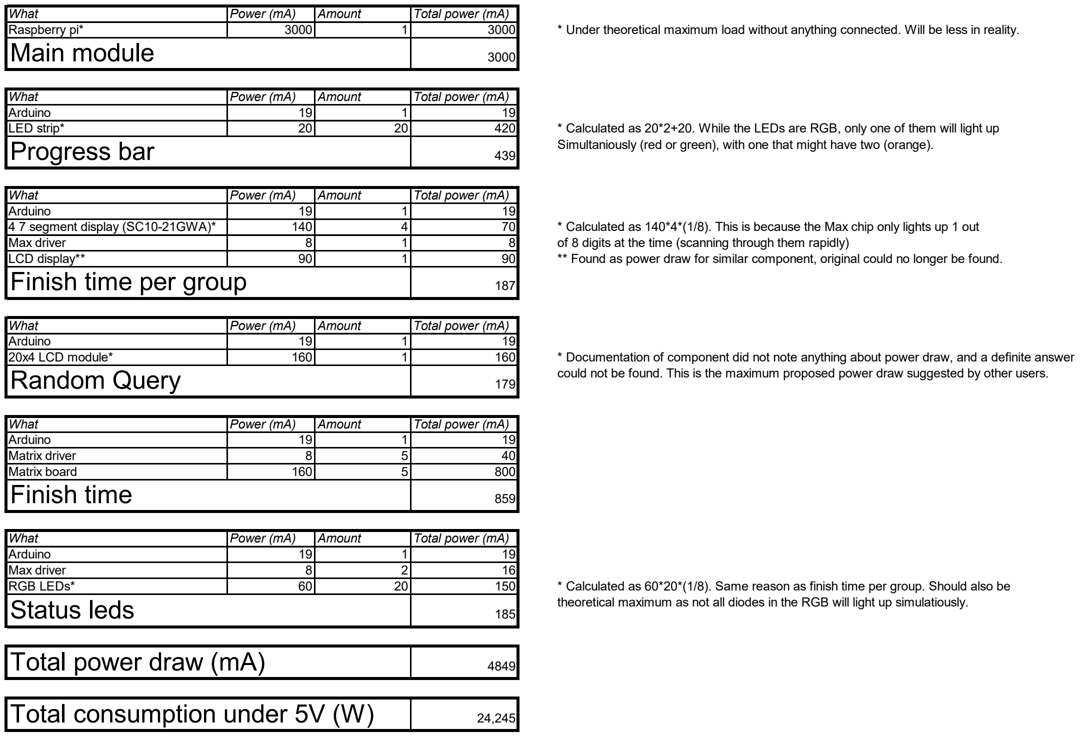


Figure 4: Project planning showing division of tasks and phases.

B Power

After figuring out what hardware we would use for the project, we could also calculate the power necessities of everything. The table below should give a quick overview of the theoretical power consumption of all the used modules. For each module, we aimed for accuracy or a worst-case scenario, so the power usage should be as listed below or better. Note that there are some anomalies in the table, which are described and explained with text next to the table. The seeming discrepancy in the calculation for the “time per group” and “status leds” modules is because of how the MAX7219 driver works (used for both). It quickly switches through its digit outputs, turning only one out of eight on at a time. By doing this quickly, it creates the illusion that the LEDs are on all the time, while they actually only are on for an eighth of the time. This also means only an eighth of the LEDs draw power at the same time.



What	Power (mA)	Amount	Total power (mA)
Raspberry pi*	3000	1	3000
Main module			3000

* Under theoretical maximum load without anything connected. Will be less in reality.

What	Power (mA)	Amount	Total power (mA)
Arduino	19	1	19
LED strip*	20	20	420
Progress bar			439

* Calculated as 20*2+20. While the LEDs are RGB, only one of them will light up Simultaniously (red or green), with one that might have two (orange).

What	Power (mA)	Amount	Total power (mA)
Arduino	19	1	19
4 7 segment display (SC10-21GWA)*	140	4	70
Max driver	8	1	8
LCD display**	90	1	90
Finish time per group			187

* Calculated as 140*4*(1/8). This is because the Max chip only lights up 1 out of 8 digits at the time (scanning through them rapidly)
** Found as power draw for similar component, original could no longer be found.

What	Power (mA)	Amount	Total power (mA)
Arduino	19	1	19
20x4 LCD module*	160	1	160
Random Query			179

* Documentation of component did not note anything about power draw, and a definite answer could not be found. This is the maximum proposed power draw suggested by other users.

What	Power (mA)	Amount	Total power (mA)
Arduino	19	1	19
Matrix driver	8	5	40
Matrix board	160	5	800
Finish time			859

What	Power (mA)	Amount	Total power (mA)
Arduino	19	1	19
Max driver	8	2	16
RGB LEDs*	60	20	150
Status leds			185

* Calculated as 60*20*(1/8). Same reason as finish time per group. Should also be theoretical maximum as not all diodes in the RGB will light up simulatiously.

Total power draw (mA)	4849
Total consumption under 5V (W)	24,245

Figure 5: A table quickly showing the calculated power necessities of the project.

C Meetings

Introductory Meeting: Monday, 8th of September 2025

When we learned we were going to be working on the *Blinking Lights* project by DACS, we planned a meeting with our supervisors right away. During this first meeting, we were introduced to the system of OpenIntel, the type of data we were going to be working with, and an initial requirement:

> "The final product must be functional, correct, and aesthetically pleasing."

Data

We learned that we are working with an *Influx Database* where data gets pushed by a measurement system. Currently, OpenIntel is migrating from a Postgres Database to an Influx Database, and we are required only to write queries for Influx. Because of this, we were not working with all available data yet, but eventually, more data will be fed into this database.

Physical Product

The product we have to create will be a *display board* in the style of a *sci-fi control panel*, which will be mounted on the wall near the DACS offices. The panel will feature several modules, each displaying different data on the progress of the daily queries performed by the OpenIntel system.

Microcontrollers

During the introductory meeting, we were tasked to investigate which *microcontroller* to use for the project. Suggested options included the Raspberry Pi, ESP32-C, and Arduino Nano. The chosen microcontroller must allow for an internet connection, as we need to query data from the DACS database.

First Meeting: Friday, 12th of September 2025

We used the first meeting to introduce our supervisors to possible requirements and asked for feedback on potential modules. The list of requirements can be found in Section **??** (*Requirement Specification*), and the possible modules we presented to our supervisors were:

1. A panel with several LEDs indicating a status per worker group.
2. An LCD screen displaying a random measurement and related data.
3. The expected finish time for the measurements of the day.
4. A progress bar displaying the progress of all queries (%).
5. A world map which becomes increasingly colored as more queries are completed for specific countries or regions.
6. A progress meter displaying the exact percentage of how many measurements have been completed.

7. A comparison panel between different queries (e.g. percentage of A measurements vs. AAAA measurements).
8. A panel displaying pie charts per region, showing what percentage of measurements use the DNS-Security Protocol.
9. An expected finish time per worker group (switching between groups, displaying one at a time).
10. The total number of measurements completed at that point in time for the given day.
11. The numbers displayed on the OpenIntel website (i.e. how many domains are measured daily, how many data points these produce, and how many have been collected since 2015).

Moodboard: Possible Modules

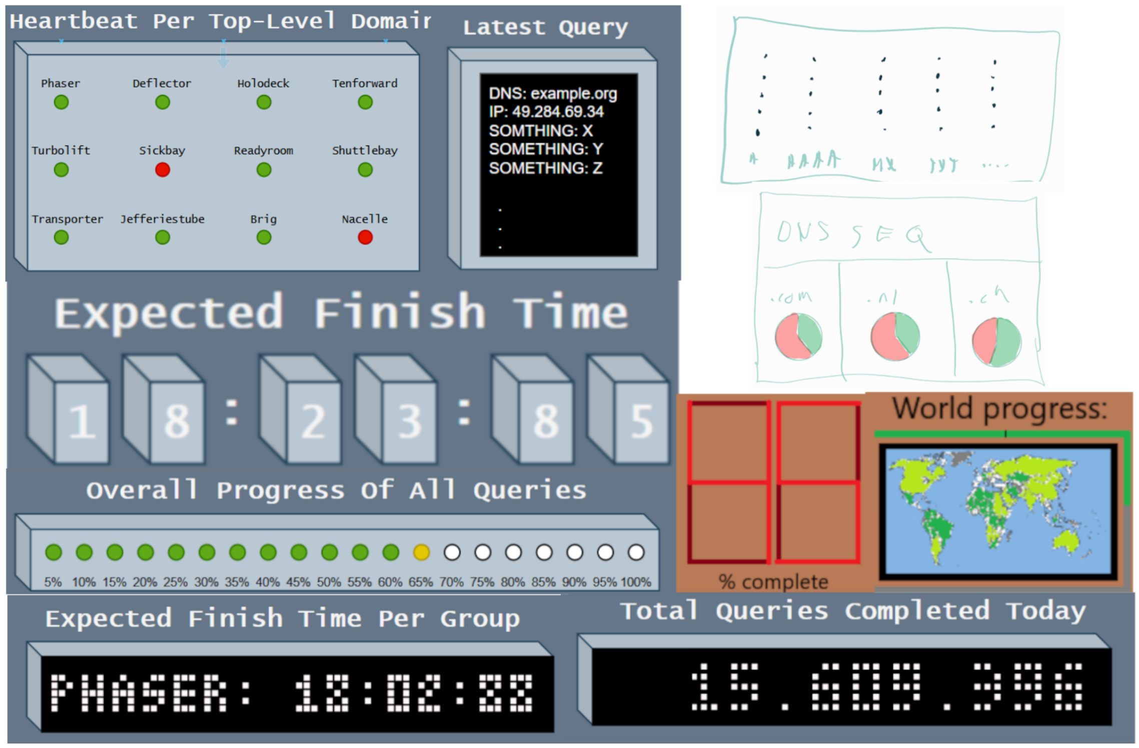


Figure 6: Moodboard for the possible modules.

Miscellaneous Ideas:

- A reset button to restart the hardware.
- An outage alarm to indicate when something is wrong and requires attention (not with the panel itself, but with the OpenIntel measurements).

Finally, we presented our idea for the construction of the physical board. It will consist of a main board on which several separate module boxes are installed, each containing the hardware for a specific module. This allows for a modular design and easy access to each module separately.

During this meeting, we received feedback on which modules DACS would like to see implemented and which modules could be omitted. From the list above, Modules 6, 7, 8, and 11 were deemed unnecessary by our supervisors, allowing us to focus on the remaining ones.

We had written structured requirements, and our supervisors instructed us to follow these; these are the same requirements that can be found in the corresponding section of this document.

Second Meeting: Friday, 19th of September 2025

Before the second meeting, we investigated which hardware we needed DACS to order for us to start implementing the first modules. Two days prior to this meeting, we sent them a request containing the following list of materials required for our project. The ordered components are listed in Table 1 and Table 2.

Table 1: Hardware requested from DACS before the second meeting (online order).

For Module	**What**	**Quantity**	**€/pc**	**Total (€)**
Total Queries Done Today	3-digit 7-segment display	3	4.29	12.87
Total Queries Done Today	MAX7219 Driver	5	1.50	7.50
Grouped Expected Finish Times	1-digit 7-segment display	4	4.29	17.16
All modules	Arduino Nano	5	2.75	13.75
All modules	Breadboard	3	4.56	13.68
Total				**€64.96**

Table 2: Hardware components obtained from Stores (Educafe).

For Module	**What**	**Quantity**	**€/pc**	**Total (€)**
Status LEDs	RGB LED (common cathode)	20	0.03	0.60
Status LEDs	TLC5916	10	1.02	10.20
Overall Progress of Queries	LED strip WS2812B	20	0.05	1.00
Overall Expected End Time	8×8 LED Matrix board (MAX7219)	5	2.38	11.90
Total				**€23.70**

We also asked for elaboration on how to implement the world map and requested our supervisors to provide a framework for the *Random DNS Query* module, since there was currently no way to access the necessary data from the database. DACS informed us that they would implement a dedicated *measurement bucket*, which would periodically send random measurements to a separate table in the database. This data can then be used to display the Random Measurement on our panel.

Finally, we were told that the ordered hardware would likely be delivered around the middle of the upcoming week.

Following Meetings

The following meetings took place on the 26th of September, the 5th of October, the 17th of October, and the 24th of October. During these meetings, we presented our supervisors with updates on the implementation phase. No new decisions were made during these sessions; they served purely to inform DACS of our ongoing progress.

D Central Control Unit

The entire board is controlled by one main module, the Raspberry Pi 5. This module is responsible for making queries to the database and processing that data. It then sends commands to all the other modules to display this data, using a protocol based on SPI.

D.1 Python Code

The code on the Raspberry Pi is made with Python. To make it easier to maintain, we have divided it into multiple files. There are three main files: ‘main.py’, ‘protocol.py’ and ‘database.py’. In the database file, all the queries for the database are written out. This is explained in more detail below.

The second file is ‘protocol.py’. This file contains a class for the protocol, which is used by all modules. It has functions for opening and closing the protocol and for sending data. It also supports self-identification of connected modules and maintaining a complete module list; however, this feature is currently disabled, as the Arduinos do not support it.

Third, we have the file ‘main.py’. This file manages all the modules and ensures they update every 5 minutes. Initially, it was planned to get a list of module names from the protocols. However, currently it simply reads from a JSON file instead.

D.1.1 Configurations files

To configure the system, a few files need to be configured. First, we have ‘pinConfigurations.json’. In this file, all the modules are listed, with the corresponding pins. The order in which the modules are listed matters, as this is also the order in which the modules are updated. Therefore, modules that take a long time to send data should not be placed at the top. This file is also included in GitLab.

Secondly, there is a file called ‘statusLeds.json’, which includes the mapping for the status LEDs. For this one, the order does not matter, but to make it easily readable, it is advised to use the same order as on the board. In this file, up to 20 (currently) worker groups can be added, which will then be displayed on the status LEDs. Pins that are not included in the file will be turned off; otherwise, they will get a colour (depending on their state). This file is also included on GitLab.

Third, there should be a file called ‘.env’. This file should contain the token for the database, in the following form:

```
TOKEN="ABC123"
```

Because this token gives access to the database, it is not included on GitLab and has to be added each time the repository is cloned.

Module files

To facilitate the addition of new modules, each module has its own file. These files all have a class for the module, with at least three functions: ‘__init__’, ‘update’, and ‘close’. The ‘__init__’ function creates the class for the module, and has two parameters, ‘pin’ and ‘database’. The pin parameter indicates which chip select the module is connected to, allowing the protocol to be established. The database parameter is a database object that can be used to collect the data needed for the module. The function ‘update’ is called by the main loop every 5 minutes. This function should make a query to the database and send a message to the module. The ‘close’ function is called when the system needs to shut down. Its primary function is to close the protocol, but for some modules, it also needs to complete a few additional tasks. We have created a file ‘testModule.py’, which contains an example of a complete module.

Libraries

We have used several libraries. This is the list of them:

- gpiozero
  This library is used to control the GPIO. Used to control the reset pin.
- influxdb_client
  This is the libary to collect data from the database
- spidev
  We use this library to control the SPI pins. These are used to communicate with the modules
- dotenv
  The token for the database is stored in a .env file, this library is used to read the token

D.2 Reset button and shutdown button

The Central Control Unit has two buttons, which are the reset button and the shutdown button. The buttons will trigger a software based reaction, meaning that in the case python crashes, they will not work. It could also be the case that other running code might delay the reaction for a bit. However, this is usually not the case, and the main goal is still to safely reset the pi. This is still achieved.
When the button is pressed it will start the corresponding python function (button-shutdown or button-reset). The GPIO pin 25 will be pulled low to trigger resets on all the arduino’s. The pin is connected through the SPI-hub with brown cables to the reset pin on all the arduinos. After this reset, the pi will either call a reboot or a shutdown of itself.
The connection of the buttons to the pi can be seen in 15 on pin 34, 36, and 38. The Ground is connected to the pin of reboot or shutdown when the corresponding button is pressed. This results in the python code being called.
It is important to note that in the final product both buttons will result in a reboot. This decision was requested because of the risk of passerby’s pressing the shutdown button.

D.3 Database connection

All the data comes from the DACS database. For this, they use InfluxDB with two Buckets, ‘ZabbixMirror’ and ‘MeasurementSamples’. Because most of the data comes from the ‘ZabbixMirror’, and uses a comparable query, we have decided to make a function (‘executeQuery’) that already has some default parameters. This function selects the data from ‘ZabbixMirror’, and the filter is on the time. By default, it selects all results from the last 20 minutes, but this can easily be changed.

When this function is used, it requires the second part of the query as a parameter. Most times, this consists of some filters, the last keyword, and finally a keep statement.

There is one query that is different, this is the one for the random queries. This one uses the bucket ‘MeasurementSamples’ instead of ‘ZabbixMirror’. Therefore, this query does not use the ‘executeQuery’ function.

D.4 Communication

For the communication between the Raspberry Pi and Arduinos, we based our protocol on SPI. We chose this interface because it was widely used and because we expected it to be easy to implement. The protocol used to communicate over the SPI bus is as follows:

To start, each message is split up into lines of at most 20 characters, which end with a ‘\n’.

Most messages consist of data to be sent to the Arduino. These start with the indicator ‘begindata\n’, notifying the Arduino that it should be expecting data. It then sends the actual data, which is formatted differently per module. How the data is formatted for each module can be found in the appendix at section E. After each batch of data, the indicator ‘enddata\n’ is sent, showing the message to be complete.

The various Chip Select pins for each Arduino are currently hardcoded into the Raspberry Pi, as the Arduinos are not able to send information back to the Pi for self-identification. However, support for this is implemented in the protocol on the Raspberry Pi side. In this case, the Pi will send ‘name\n’, to which the module will respond with its identifier, followed by ‘\n’. It is currently disabled; more information about this can be found in 8.2.

D.5 Physical wiring

First, all the modules are connected to the Raspberry Pi with SPI. For this, there are four cables which all modules have in common: GND, MOSI, MISO and CLK. Besides this, all modules have a CS, which means Chip Select. All of the Arduinos have their own CS pin on the Raspberry Pi, so it can send commands to only one of them. Besides this, there is one reset pin on the Raspberry Pi. When this is pulled to GND, all the arduinos will reset. Below are two schematics, one with the pin layout of the Raspberry Pi, and one with a connection to an Arduino.

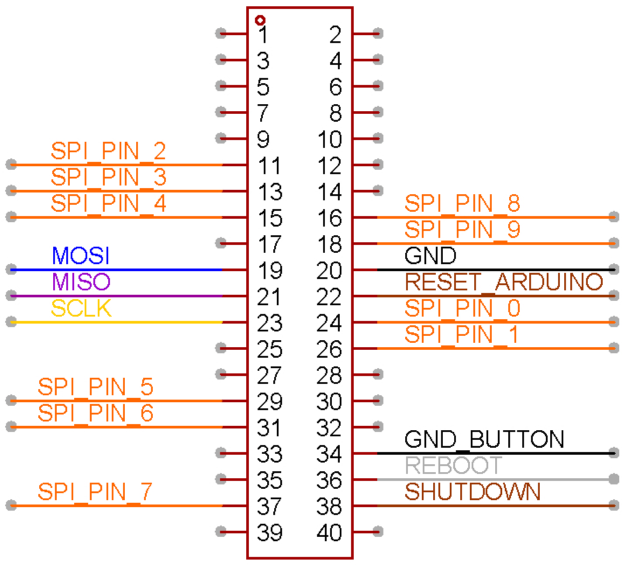


Figure 7: Connection scheme of the Raspberry Pi

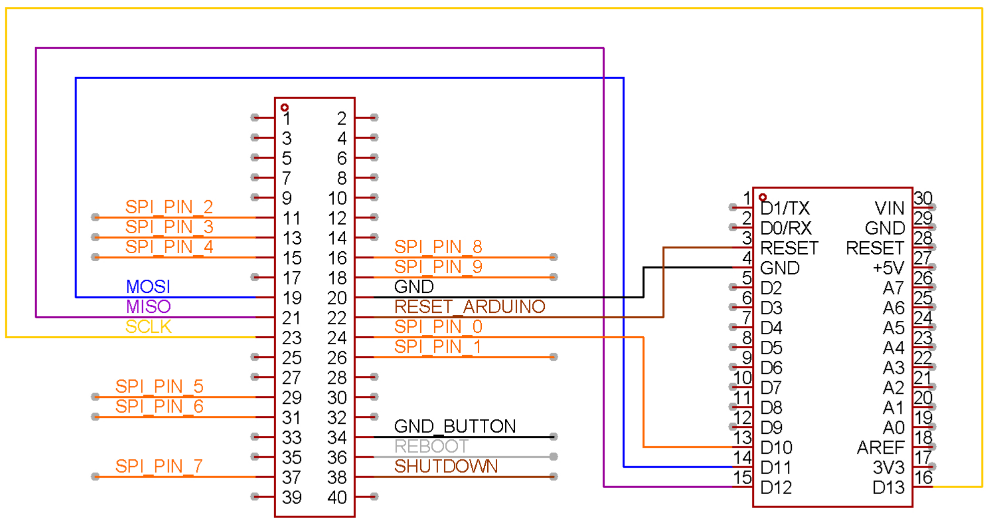


Figure 8: Scheme of connection between Raspberry Pi and Arduino

E Modules

E.1 Recently Measured

This module displays a randomly selected measurement from a specially created bucket (`MeasurementBucket`). The chosen measurement remains on the screen for **10 seconds**, after which the control unit sends a new measurement to display.

Hardware Involved

- 1× Arduino Nano
- 1× Joy-IT COM-LCD20x4 Display
- $1 \times 22\,k\Omega$ resistor
- 1× 2.2 kΩ resistor
- 1× 2.2 μF electrolytic capacitor

Idle State (After Boot)

The Idle State of this module displays "Waiting for data..." on the first line of the LCD screen and does not display anything else.

Software Involved

We use the `LiquidCrystal` library for Arduino (by Hans-Christoph Steiner), compatible with the Hitachi HD44780 controller used on the Joy-IT COM-LCD20x4 module. If a measurement contains an IPv6 address or a DNS name that exceeds the device's per-line limit of twenty characters, a horizontal scrolling mechanism is used to display the full content over time.

Internal Communication Protocol

Every data package received by this module via the SPI bus contains all the information required to display a single random DNS query. The incoming package is structured as a two-dimensional array, `Message[20][20]`, where each element is a null-terminated character string (up to 20 characters). The table below outlines the meaning and display mapping of each relevant index.

`setup()` — hardware initialization and idle state When the Arduino receives power (or after a reset), the module first configures SPI (as slave) and initialises the LCD screen.

```
pinMode(MISO, OUTPUT);
SPCR |= _BV(SPE);
SPI.attachInterrupt();

i = 0;
linecomplete   = false;
processingdata = false;
```

Index	Content Description	LCD Display Line
`Message[0]`	Hostname of the queried domain.	Line 1: `Hostname`
`Message[1--3]`	IP address (IPv4 uses only `Message[1]`, IPv6 spans up to `Message[3]`).	Line 3: `IP Address`
`Message[4]`	Geographic location code (2–3 letters, e.g., `NL`, `USA`).	Line 4: `Location: <code>`
`Message[5--7]`	DNS name (domain name); concatenated if longer than 20 characters.	Line 2: `DNS Name`

Table 3: Structure and display mapping of the `Message[20][20]` array.

```
lcd.begin(20, 4);
delay(100);
lcd.clear();
lcd.setCursor(0, 0);
lcd.print("Waiting for data...");
```

Listing 1: Essential parts of setup()

Global variables used by these mechanisms

- **Line/protocol assembly:** `line[20]`, `message[10][20]`, `i`, `linecomplete`, `processingdata`, `linenumber`
- **Delimiters:** `begindata`, `enddata`
- **LCD object:** `LiquidCrystal lcd(9, 8, 7, 6, 5, 4)`
- **DNS scrolling:** `dnsScroll`, `dnsFull[61]`, `dnsLen`, `dnsPos`, `dnsTick`, `dnsStepMs`
- **IP scrolling:** `ipScroll`, `ipFull[61]`, `ipLen`, `ipPos`, `ipTick`, `ipStepMs`

`processdata()` — **handling incoming data from the control unit** When a complete packet has been received, the function clears the LCD, displays the hostname, and reconstructs multi-line fields (DNS, IPv6) into single buffers (ipFull and dnsFull). Fields of 20 characters or fewer are printed directly; longer ones enable scrolling.

```
lcd.clear();
lcd.setCursor(0, 0);
lcd.print(message[0]);

lcd.setCursor(0, 3);
lcd.print("Location: ");
lcd.print(message[4]);
```

Listing 2: High-level flow of processdata()

An if else statement separates IPv4 from IPv6 by checking whether `Message[2]` is empty:

```
if (message[2][0] == '\0') {
    ipScroll = false;
    lcd.setCursor(0, 2);
    lcd.print(message[1]);
} else {
    int p = 0;
    for (int k = 0; k < 20 && message[1][k] != '\0' && p < (int)sizeof(
       ipFull)-1; ++k) ipFull[p++] = message[1][k];
    for (int k = 0; k < 20 && message[2][k] != '\0' && p < (int)sizeof(
       ipFull)-1; ++k) ipFull[p++] = message[2][k];
    ipFull[p] = '\0';
    ipLen    = p;

    ipScroll = true;
    ipPos    = 0;
    int steps = (ipLen - 20 + 1);
    if (steps < 1) steps = 1;
    ipStepMs = 10000 / steps;
    if (ipStepMs == 0) ipStepMs = 1;
    ipTick   = millis();
    displayWindow(ipFull, 0, 2);
}
```

Listing 3: IPv4 vs IPv6 selection in processdata()

DACS Database Query

This module uses a special bucket called `MeasurementSamples`. Here we give a range for 24 hours, which means we will take a random sample from the previous 24 hours. This is because if we were to take a random sample from, let's say, the last five minutes, there will not be enough measurements in the bucket at the end of the day, while at the beginning of the day they will be in abundance. When DACS migrates more worker nodes, there should be more data in the bucket, so then it can be changes, but they indicated that the migration will not be finished soon. From this bucket we take the `host`, `address`, `geo` and `name`, which are then sent to the Arduino. The Flux query can be found in the code section below:

Listing 4: Flux query used to retrieve measurements from the DACS database.

```
from(bucket: "MeasurementSamples")
|> range(start: -24h)
|> filter(fn: (r) => r["_measurement"] == "dnsresult")
|> filter(fn: (r) => r["_field"] == "address" or r["_field"] == "geo" or r["_field"] ==
|> pivot(rowKey: ["_time"], columnKey: ["_field"], valueColumn: "_value")
|> yield(name: "sample")
|> keep(columns: ["_time", "_measurement", "host", "address", "geo", "name"])
```

Arduino - Helper Function

`displayWindow()` — **20-character sliding window displayer** This method prints an exact 20-character window from a larger buffer to a given LCD row. The scroll logic is implemented in such a way that when the pos + 20th character is the final character, it starts over, so that it always uses the full line.

```
void displayWindow(const char *text, int pos, int row) {
    lcd.setCursor(0, row);
    for (int i = 0; i < 20; i++) {
        lcd.print(text[pos + i]);
    }
}
```

Listing 5: Viewport rendering

Hardware Limitations Because the LCD has only **20 characters per line**, we enable horizontal scrolling whenever the **DNS name** or the **IP address** exceeds 20 characters. For this, we have implemented two booleans, dnsScroll and ipScroll, which control whether or not each line requires horizontal scrolling.

For IPv6, the scroll speed is fixed. For DNS, the interval is computed so the full string traverses in about ten seconds:

$$\text{steps} = \text{DNSLength} - 20 + 1, \qquad \text{dnsStepMs} = \left\lfloor \frac{10000}{\max(1, \text{steps})} \right\rfloor$$

The visible 20-character window is printed by `displayWindow()`, which receives the full buffer (`dnsFull` or `ipFull`), the current window start (`dnsPos` or `ipPos`), and the LCD row (1 for DNS, 2 for IP).

```
if (dnsScroll) {
    unsigned long now = millis();
    if (now - dnsTick >= dnsStepMs) {
        dnsTick = now;
        int maxPos = dnsLen - 20;
        dnsPos = (dnsPos < maxPos) ? (dnsPos + 1) : 0;
        displayWindow(dnsFull, dnsPos, 1);
    }
}

if (ipScroll) {
    unsigned long now = millis();
    if (now - ipTick >= ipStepMs) {
        ipTick = now;
        int maxPos = ipLen - 20;
        ipPos = (ipPos < maxPos) ? (ipPos + 1) : 0;
        displayWindow(ipFull, ipPos, 2);
    }
}
```

Listing 6: Advancing the scroll window in the main loop

Schematic: Random DNS Measurement

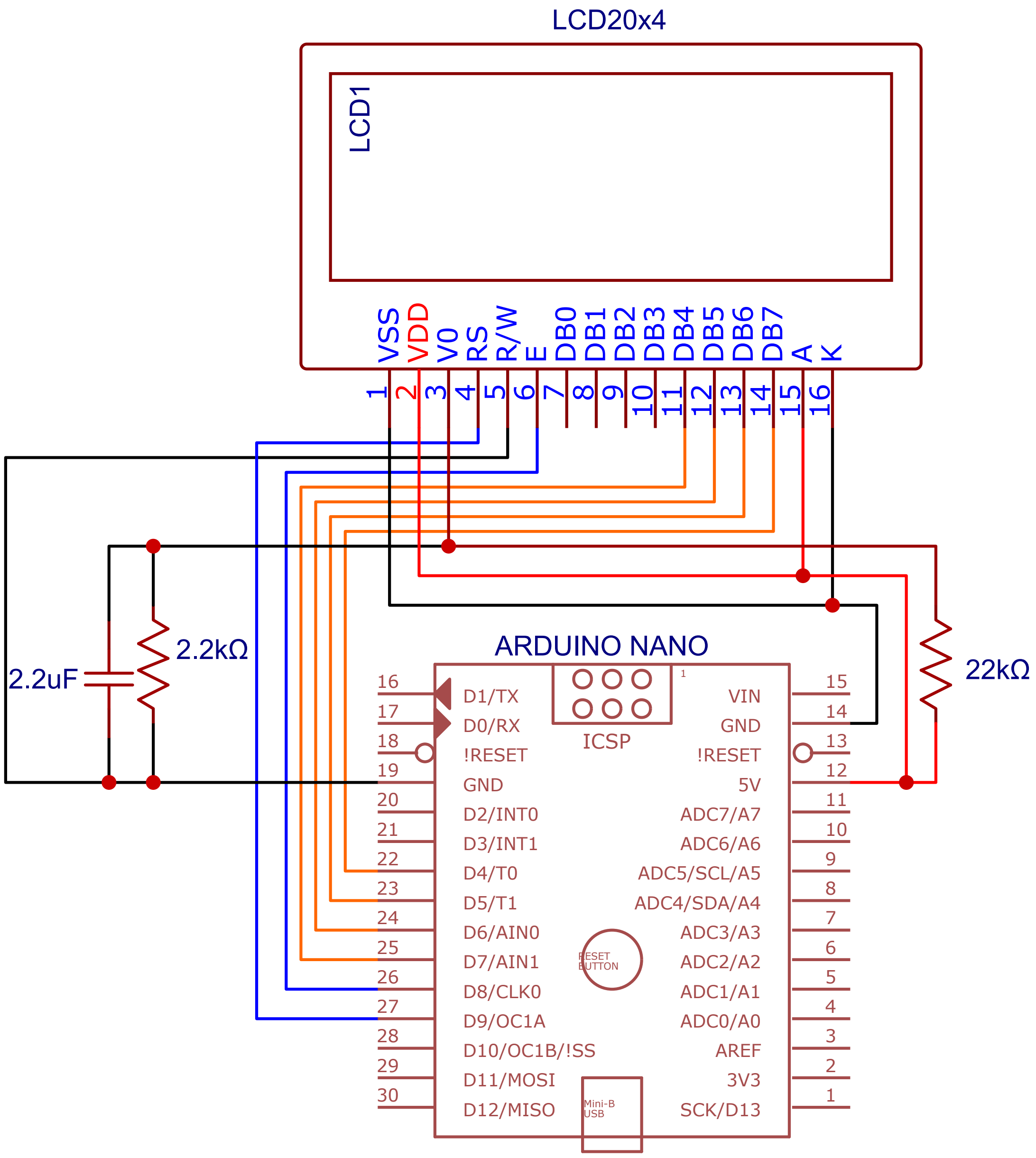


Figure 9: Schematic for the *Random Query* module.

E.2 Progress Bar

This module uses an LED strip to visualise the total progress of the current day's measurements in the form of a progress bar.

Hardware Involved

- 1× Arduino Nano
- 1× WS2812B LED strip (20 LEDs)

A good practice would be to add a 330Ω resistor in the data line to the LED strip; However, since this is not strictly necessary and for ease of connection, this was left out.

Idle State (After Boot)

The idle state of this module makes every LED turn blue until it receives its first data.

Software Involved

The total progress bar is quite a simple module. To communicate with the LED strip, we use the FastLED library. First, we need to set up an object for the Arduino to send commands to the LED strip. After this, the object must be initialised. This is done by the code below:

```
CRGB leds[NUM_LEDS];       //Constructor from the library for building
    the LEDs

void setup() {
    FastLED.addLeds<WS2812, LED_PIN, GRB>(leds, NUM_LEDS); //Setup of
        the LEDs by the library
}
```

Listing 7: Initialisation of the LEDs

Here, WS2812 signifies the type of LED, and LED_PIN and NUM_LEDS are macros signifying the digital pin used for data transfer and the number of LEDs in the strip, respectively.

The Arduino receives a number between 0 and 100 from the main module, which denotes the progress percentage. As the measurement progresses, the LEDs gradually change from red to green from left to right. With 20 LEDs, this means each green LED signifies 5% worth of progress. How much this is, is calculated for every LED and displayed every time the Arduino receives a new number with the following code:

```
percentage = atoi(message[c]);

for (int i = 0; i <= 19; i++) {
  float brightness = float(percentage / 5.0);
                                                              // get the
      brightness of the LED as a float
  if (brightness >= 1) {
    leds[i] = CRGB (0, MAX_BRIGHTNESS, 0);
                                                              // if the value
        of brightness is higher than 1, turn the LED on (green)
  }
  else if (brightness > 0 && brightness < 1) {
    leds[i] = CRGB (MAX_BRIGHTNESS * (1 - brightness), MAX_BRIGHTNESS *
        brightness, 0);   // if the value of brightness is between 0 and
         1, turn the LED on lower brightness (yellow)
  }
  else {
    leds[i] = CRGB (MAX_BRIGHTNESS, 0, 0);
                                                              // if the value
        of brightness is 0 or below, turn LED off (red)
  }

  percentage -= 5;

      // decrease percentage number for the next LED
  FastLED.show();

      // upload brightness data to the LED
}
```

Listing 8: Calculate color and brightness for each LED

Very simply, we loop through all the LEDs and check what colour each of them needs to be. Each LED indicates the status of 5 percentage points, and every LED will transition from red to orange to green to show how far the scan has progressed.

Internal Communication Protocol

This module only expects the total progress, as a number between 0 and 100.
Example:

```
begindata\n
35\n
enddata\n
```

Database query

Unfortunately, the database does not directly include a progress update, so this is computed by the Pi. To achieve this, the total queries done today are queried, along with the queries from yesterday. We then estimate a progress percentage by assuming that the total number of queries that need to be completed today equals the number completed yesterday. This is, of course, not entirely correct, but in practice, the difference is insignificant.

The queries used are listed below. We use the 'sum' function to sum the results from all groups.

Listing 9: Functions to retrieve domain counts from InfluxDB

```
def getDomsToday(self):
    flux_query = f'''
    |> filter(fn: (r) => r["item_key"] == "cms.doms_today")
    |> last()
    |> group(columns: ["host"])
    |> keep(columns: ["_value"])
    |> sum()
    '''
    result = self.executeQuery(flux_query)
    return(result[0].records[0].values.get("_value"))

def getDomsLastDay(self):
    flux_query = f'''
    |> filter(fn: (r) => r["item_key"] == "cms.doms_last_day")
    |> last()
    |> group(columns: ["host"])
    |> keep(columns: ["_value"])
    |> sum()
    '''
    result = self.executeQuery(flux_query)
    return(result[0].records[0].values.get("_value"))
```

Schematic: Total Progress Bar

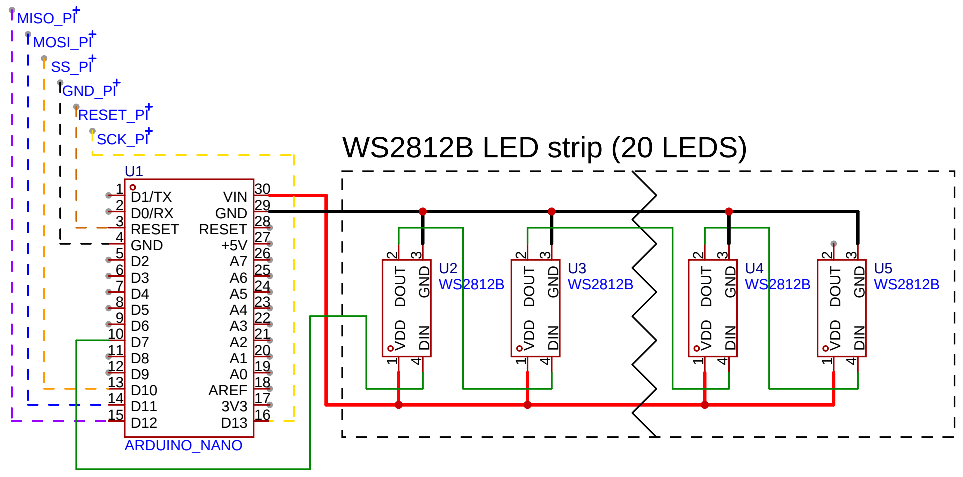


Figure 10: Schematic for the *Total Progress Bar* module.

E.3 Predicted End Time

This module displays the predicted end time of the entire scan. Each worker group has its own expected completion time, so to estimate the total expected finish time for today, we simply take the latest end time of a single worker group and display this time on a matrix board.

Hardware Involved

- 1× Arduino Nano
- 5× 8x8 LED Matrix Red with MAX7219 Driver (1088AS)

Idle State (After Boot)

This module displays "00:00:00" as the idle state until it receives its first data.

Software Involved

For this module, we use the library `LedControl` (by Eberhard Fahle). This allows us to individually control each LED on the five (daisy-chained) matrix boards.

Digit Font Definition

Each digit 0 to 9 uses 7 rows and 5 columns to display itself. We have divided the daisy-chained matrix board into six segments, three pairs of two divided by ":", which will indicate the hours, minutes and seconds of the estimated finish time. In the table below you can find at what column each segment starts. The digits themselves have been hardcoded in `digitFont[10][7]`.

Segment	Start Column	Represents
0	0	Hour tens
1	6	Hour ones
2	14	Minute tens
3	20	Minute ones
4	28	Second tens
5	34	Second ones

Table 4: Segment-to-column mapping for the time display.

Helpers

The functions `setLedMapped()` and `setLedGlobal()` set the orientation of the pixels and automatically calculate which device controls the pixel. We need this because we physically placed the matrix boards in a different order than the expected logical order. These helper functions thus remap the coordinates so that defining LED `(row 1, column 1)` corresponds to the top-left corner of the full display.

Listing 10: Helper functions for LED mapping and orientation.

```
inline void setLedMapped(int dev, int row, int col, bool on) {
    lc.setLed(dev, 7 - col, row, on);
}

inline void setLedGlobal(int row, int col, bool on) {
    int device = col / 8;       // which MAX7219 module
    int colInDevice = col % 8; // local column
    lc.setLed(device, 7 - colInDevice, row, on);
}
```

Database Query Involved

From the database we take the maximum value of `cms.estimated_end_time` and store the `_time`, `_value`, `host` and `host_name`. We then send the `_value` of this to the Arduino. This will be in the form of `x.y`, where `x` is the hours and `y` the minutes. The Raspberry Pi then converts this time to local time. For example, if we get `10.5` from the database, the Raspberry Pi knows it should first add one hour to adjust for our time zone (assuming wintertime), which means the display time becomes `11:30:00`.

Listing 11: Flux query for retrieving the estimated end time.

```
def getExpectedEndTime(self):
    flux_query = f'''
    |> filter(fn: (r) => r["item_key"] == "cms.estimated_end_time")
    |> last()
    |> highestMax(n: 1)
    |> keep(columns: ["_time", "_value", "host", "host_name"])
    '''
    result = self.executeQuery(flux_query)
    for table in result:
        for record in table.records:
            return(record.values.get("_value"))
```

Schematic: Total Expected Finish Time

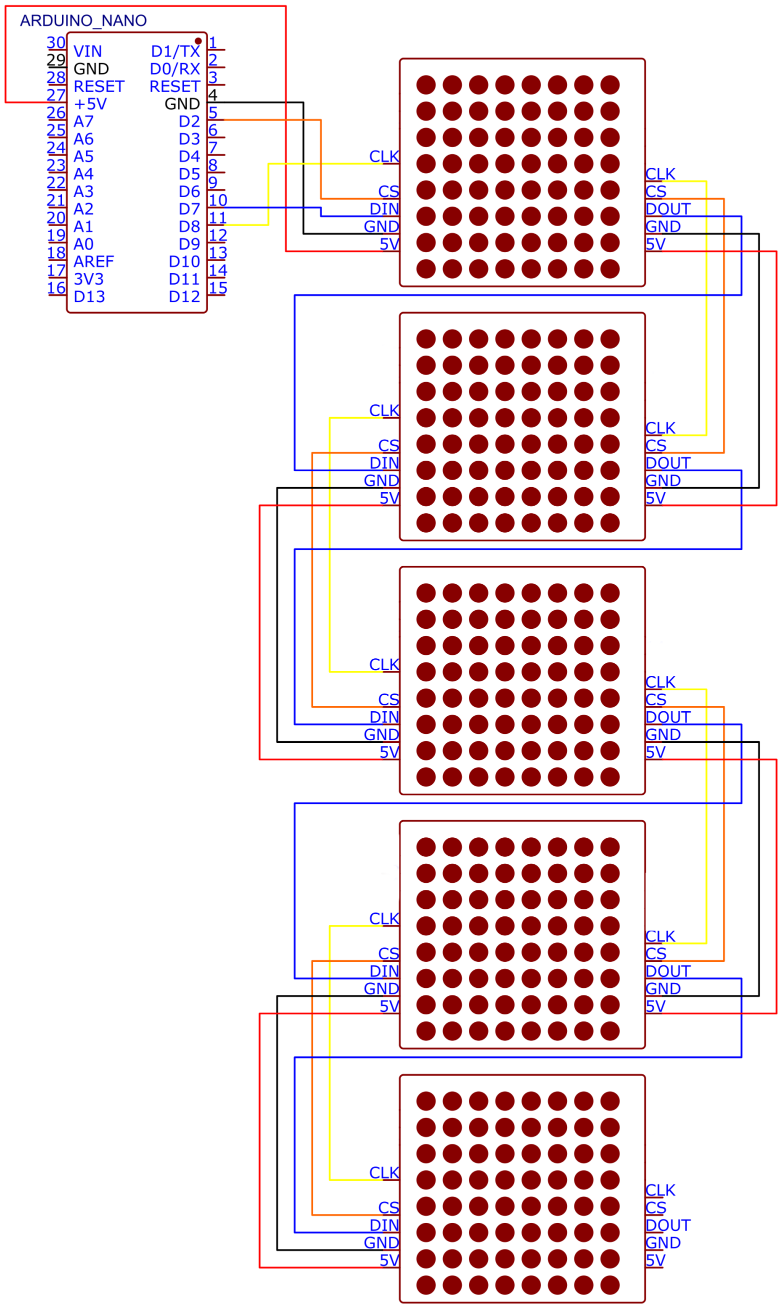


Figure 11: Schematic for the *Total Expected Finish Time* module.

E.4 Predicted End Time Per Group

This module displays the predicted time that every worker group will finish. This visualises which groups are still working and which have already finished, with an indication of when they are expected to finish. The module cycles through all the worker groups, displaying their respective predicted end times one at a time. These times are accurate to the minute and displayed on four seven-segment displays. The current group is displayed on a small LCD screen that is mounted in front of the seven-segment displays.

Hardware Involved

- 1× Arduino Nano
- 1× MAX7219CNG control chip
- 4× SC10-21GWA seven-segment display (green)
- 1× 240x320 ST7789 SPI TFT LCD touchscreen
- 1× 12 kΩ resistor
- 5× 330Ω resistor

Idle State (After Boot)

This module displays "Display Ready" on the LCD Screen and "12:34" on the 7-segment displays.

Software Involved

To control all the elements on the board, we have used two libraries. We have used a library called LedControl to send instructions to the MAX7219 chip to control the seven-segment displays. Additionally, a modified version of the Adafruit_ST7789 library was used to control the LCD screen, a library originally written by the manufacturer of the control chip in the display, but modified by us to fit our needs. The Arduino did not have enough program memory to contain the entire graphics library, so we have modified the library and the underlying graphics library to save on program memory. The largest change made was that some of the initialising functions were set to constants that our screen uses, and unnecessary characters were removed from the font bitmap. Other than the changes made and the removed functionality, the library operates exactly the same and all functions are interchangeable.

In total, this module uses eight pins on the Arduino Nano to send data to the components (3 for the seven-segment displays, and 5 for the LCD display), and two pins to supply (5V) power. Additional pins are also used for communication with the main module.

First, for both components, a software object must be created, and both also need some initialisation. Creating the object is done by supplying it with the pin numbers that are used. These are defined in macros and can be changed. All of this must be done when the Arduino powers on; otherwise, the components will not work. This is done in the following code block.

```
// Initialize the LCD screen object
```

```
Adafruit_ST7789 display = Adafruit_ST7789(DP_CS_PIN, DP_DC_PIN,
    DP_MOSI_PIN, DP_SCLK_PIN, DP_RESET_PIN);

// Initialize the seven segment display object
LedControl sevSeg = LedControl(SS_DIN, SS_CLK, SS_LOAD, 1);

void setup() {
  sevSeg.shutdown(0, false);            // Get the chip out of shutdown
      mode
  sevSeg.setIntensity(0, 15);           // Set sevseg brightness to max

  // Initialisation code for the things on the board
  display.init(SCREEN_H, SCREEN_W);     // Initialize the screen
  display.fillScreen(0x0000);           // Turn background of the
      display black
  display.setRotation(3);               // Set display orientation
  display.setTextColor(0xFFFF, 0x0000); // Set text color to white, and
      letter background black. Background black is important otherwise
      it will not overwrite previous text correctly
  display.setTextSize(TEXT_SIZE);       // Set the text size
  display.invertDisplay(false);         // Set color invert off, is on
      by default for some reason
}
```

Listing 12: Initialisation of the components

For most of the operations here, it is quite straightforward what they do, and the comments briefly explain them as well. What is important to note is that the MAX7219 chip has a shutdown mode, which it will initialise in. This is to save power, and it must be turned off at the start. The LCD also starts with colours inverted, and this must also be turned off by code. Finally, the LCD has a rotation setting, which must be set to the correct value as well to make sure the text faces the right direction. The macros SCREEN_H, SCREEN_W, and TEXT_SIZE set the screen height (240), screen width (320) and size of the text (here set at 7, which multiplies the original size of the font by this number).

Before we start uploading to the screen, there are a couple of other helper functions that are used. The first is displayCompletionTime. This function is similar to one used in the total finish time module, but slightly different. It receives an amount of minutes in the form of an integer, and calculates the time in hours and minutes based on that. Then, it displays the correct time on the seven-segment displays with a simple division and modulo operation as well. The numbers must be assigned to each seven-segment display separately, one by one.

The other helper function is PadRight. This function replicates a singular operation of sprintf(). This operation pads the string with a maximal amount of whitespace on the Right side of the text that we want to be printed. This is because we want the next name of the worker group to overwrite the current text on the screen, even if it is longer. This function was implemented instead of using sprintf() because there were initially some issues with fitting the program within the maximum program space available on the Arduino. This is because the graphics libraries for the display are quite demanding. This way, we could save a little bit of memory by only using what we need.

Then, finally, we get to the part of displaying everything we need. First, we need to get the data from an array of character arrays. Every character array is the input from one worker group. The data is formatted as “1234:name”, where the first segment is the time in minutes, and the second segment is the name of the group, separated by a colon. This separation is done by using pointers to the original character array and looping through it until the colon is found. Now that the two data parts are split, we can draw them on the components. For the LCD, text can be drawn using a built-in font. We must, however, first tell the screen where to start drawing. Since we want to draw the text in the centre of the screen, we do a small calculation to make sure the starting point is at the right spot. This can be seen in the code block below. With text background enabled as black, and the buffer making sure to fill up any of the spaces unused by whitespace, we ensure that all text will be drawn over the previous text displayed, and that nothing from a previous draw remains. With the print function, the text is simply displayed on the screen. The SPI connection is very slow in writing, causing a slow scrolling effect on the screen. Because of this slowness, text above a certain length will be wrapped to a new line instead of scrolling it over the screen, as it would not be possible to scroll the text over the screen. For the seven-segment displays, we just used the previously mentioned displayCompletionTime() function. The described code can be found below.

```
const int v_offset = SCREEN_H / 2 - (TEXT_H * TEXT_SIZE); // Calculate
    where to draw text on the vertical axis to display text in the
    center (assuming two lines)
const int h_offset = TEXT_W * TEXT_SIZE * 0.5;            // Calculate
    where to draw text on the horizontal axis
```

Listing 13: Calculate drawing point

```
void displaydata() {
  // It is of course possible that this might skip at some point, but
      otherwise an entire system would need to be built to detect
      overflow of the millis() function
  if (millis() % wait_time == 0) {

    char* colon = strchr(message[drawing_line], ':'); // Find where the
        colon is in the string
    if (colon != NULL) {                              // make sure we
        can find it
      char* name = NULL;                              // Pointer that
          will point to the name string
      int time_len = colon - message[drawing_line];   // Length of the "
          time" part

      char time[4];                                   // Buffer for time
           character array
      strncpy(time, message[drawing_line], time_len); // Copy the time
          part of the message to different character array
      time[time_len] = '\0';                          // Null terminate
          the string

      name = colon + 1;
      display.setCursor(h_offset, v_offset);    // Position to start
          drawing at the right pixel
      padRight(buffer, name, MOST_CHAR);        // Format the string in
          the buffer, might be worth changing the buffer to scale with
          character size, but idk how to do that.
      display.print(buffer);                    // Print the buffer.
          Note that unfortunately, no data can be received when drawing
          to the screen
      displayCompletionTime(atoi(time));        // Display the time on
          the seven segment displays. This will be done AFTER the name
          has changed
    }
    if (drawing_line < linenumber) drawing_line++;
    else drawing_line = 0;
  }
}
```

Listing 14: Calculate drawing point

Noteably, no built-in loops are used, and the "functional" part of the module (what is described above) runs in a different order than the other modules. Instead of calling processdata(), the above function is called in the Arduino loop *after* the SPI and data-related operations are done. It is done in such a way that the text will continue looping after showing the last data point. No loops are used to ensure data is handled correctly after an SPI interrupt. Instead, every wait_time in (15 seconds), the module will attempt to draw new information, and manually update an integer (drawing_line) until the maximum of linenumber, and then loop back to 0;

Database Query Involved

It was mentioned before, but for the sake of clarity, it will be repeated here. The Arduino communicates with the Raspberry Pi through an SPI protocol. The Arduino receives an array of (at most) 15 character arrays of (at most) 20 characters. Every character array is formatted in a way where the first (maximum of 4) characters will contain the time in minutes, followed by a colon to separate the two parts of the message, and followed by the rest of the characters containing the name of the worker group. An example would be "960:phaser", where phaser is the name of the worker group and 960 is the number of minutes past midnight, which would show 16:00 in this case. The time is in Central European Time or Central European Summer Time, and the Pi takes this change into account.

Listing 15: Flux query for retrieving the estimated end times.

```
def getExpectedEndTimes(self):
    flux_query = f'''
    |> filter(fn: (r) => r["item_key"] == "cms.estimated_end_time")
    |> last()
    |> keep(columns: ["_value", "host", "host_name"])
    '''
    result = self.executeQuery(flux_query)
```

```
print("Results")
returnResult = {}
for table in result:
    for record in table.records:
        host = record.values.get("host")
        time = record.values.get("_value")
        returnResult[host] = time
return(returnResult)
```

Schematic: Finish Time Per Group

Below is the wiring diagram for the module, in case there are any faults. The wires on the physical board look different, of course, but the end-to-end connections as laid out below are correct. Additionally, there is also a connection diagram to show in a simpler way how the module board is connected to the LCD and the Arduino. Note that the LCD, MAX7219 chip, and seven-segment displays are flipped in the wiring diagram, so the layout shows the wiring as seen from the back. Note that the specified connections to the Raspberry Pi follow the wire colour scheme on the physical board, but the connections between the Arduino and the hardware do not.

To fully connect the module to the Arduino and then to the main module, the following things must be done. The LCD screen must be slotted correctly into its socket. The board, its case, and the socket were designed for this specific LCD screen, but may be replaced with another as long as the pinout is the same. Then, as can be seen in the diagram, the following Arduino must be connected in a specific way to the board (as specified by the delivered code):

- Connect Arduino 5-volt pin to pin 1 on the board
- Connect an Arduino ground pin to pin 2 on the board
- Connect Arduino digital pin 5 to pin 3 on the board. This is the load pin for the MAX7219 chip.
- Connect Arduino digital pin 6 to pin 4 on the board. This is the clock pin for the MAX7219 chip.
- Connect Arduino digital pin 7 to pin 6 on the board. This is the data in pint for the MAX7219 chip.
- Connect Arduino digital pin 4 to pin 8 on the board. This is the chip select pin for the screen.
- Connect Arduino digital pin 3 to pin 9 on the board. This is the reset pin for the screen.
- Connect Arduino digital pin 2 to pin 10 on the board. This is the data/command pin for the screen.
- Connect Arduino transit/digital pin 0 to pin 11 on the board. This is the MOSO pin for the screen.
- Connect Arduino receiving/digital pin 1 to pin 12 on the board. This is the serial clock pin for the screen.

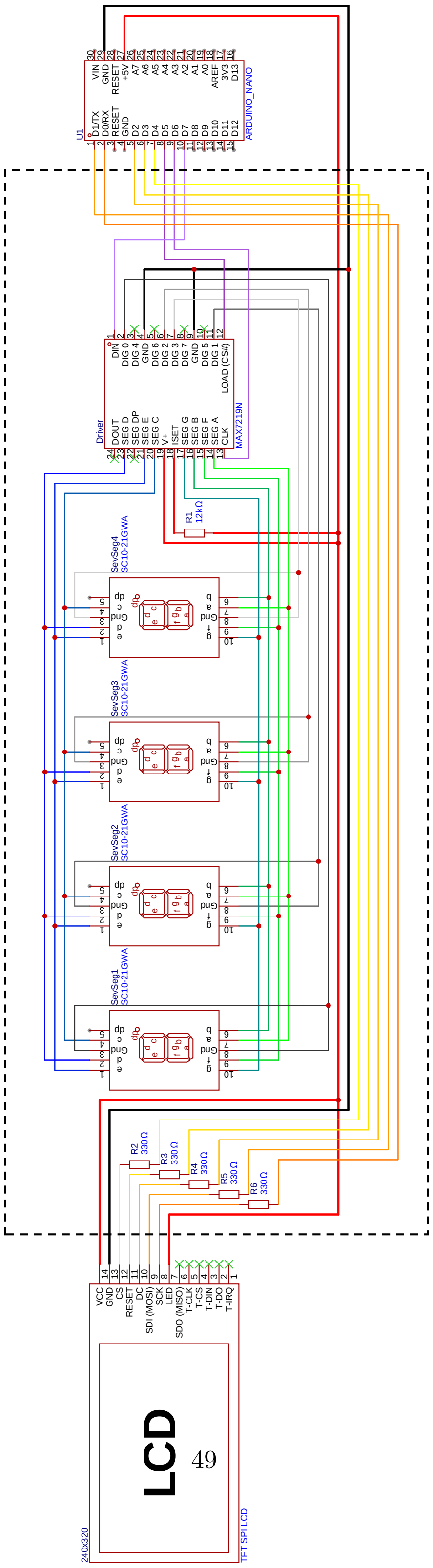


Figure 12: Schematic for the *Finish Time Per Group* module.

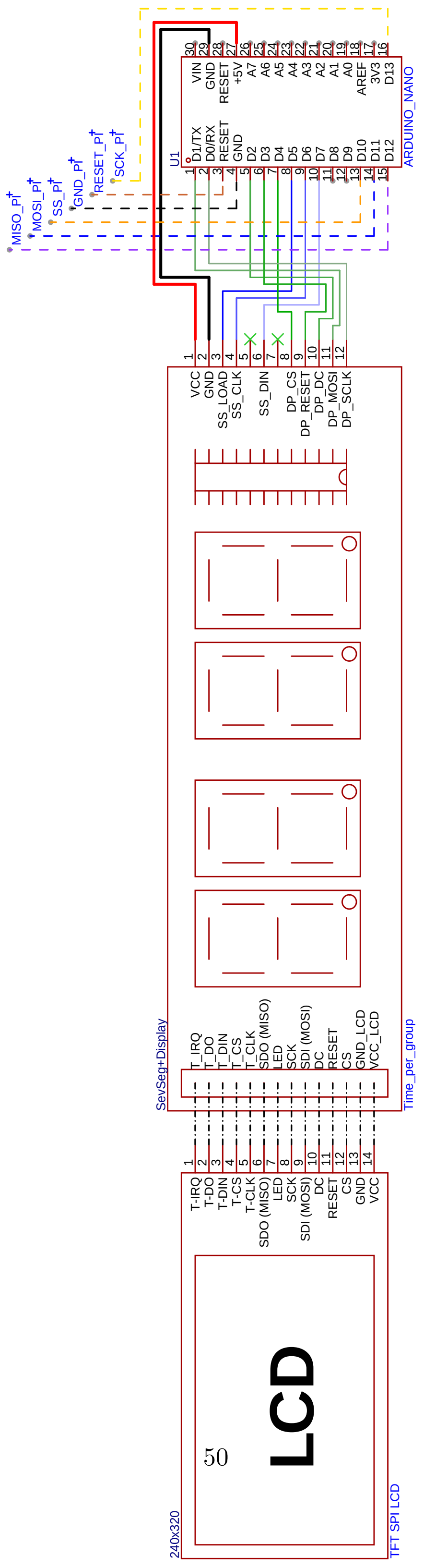


Figure 13: Connection diagram for the *Finish Time Per Group* module.

Note that all the used digital pins can be changed to different ones, if this is, of course, also reflected in the software. In the Arduino files, there are also comments on what goes where. Pins 5 and 7 are not in use.

E.5 Worker Group Status

This modules displays the the status of each worker group through coloured LEDs.

Hardware Involved

- 1× Arduino Nano
- 2× MAX7219 Display Driver
- 2× 12 kΩ resistor
- 20× RGB LED Common Cathode
- 1× 0.1 μF ceramic capacitor
- 1× 10 μF electrolytic capacitor

Idle State (After Boot)

On startup, every LED is set to the idle state (blue) until it receives further instructions.

Software Involved

To control all LEDs separately through the two MAX7219 drivers, we use the `LedControl` library (by Eberhard Fahl).

Database query

Since the statuses are not directly retrievable from the database, we had to create our own statuses. For this, we use three queries: expected end time, queries done today, and queries done yesterday.

Most statuses are determined by the expected end time. When a worker node is done (indicated by an expected end time of 00:00), the status is set to ‘done’. If it is still in progress, in most cases, the node will be set to ‘busy’. However, if the expected end time is too late, it will be set to either ‘slow’ or ‘too slow’. When a node has not made any queries yesterday and today, it is considered sleeping.

To get the data, the following code is used:

```
times = self.database.getExpectedEndTimes()
domsToday = self.database.getItemFromAllHosts("cms.doms_today")
domsLastDay = self.database.getItemFromAllHosts("cms.doms_last_day")



    def getItemFromAllHosts(self, item_key):
        flux_query = f'''
        |> filter(fn: (r) => r["item_key"] == "{item_key}")
        |> last()
        |> keep(columns: ["_value", "host", "host_name"])
        '''
        result = self.executeQuery(flux_query)
        print("Results")
        returnResult = {}
        for table in result:
            for record in table.records:
                host = record.values.get("host")
                value = record.values.get("_value")
                returnResult[host] = value
        return(returnResult)
```

Listing 16: Calculate drawing point

Internal Communication Protocol

Through the custom SPI bus, this module receives data packages in the format of `Message[20][20]`. Each package contains the current status of a specific host, to be displayed on the corresponding LED.

Currently, the system supports fifteen worker groups. However, the hardware and Arduino software support the addition of up to five more groups, so adding more groups only requires adjustment on the Raspberry Pi.

`setup()` — **hardware initialization and idle state** During setup, the Arduino first initialises the SPI (as a slave) and then initialises both the MAX7219 drivers. Afterwards, it will put each LED in the idle state, which is a solid blue light for this module.

```
void setup() {
  pinMode(MISO, OUTPUT);
  SPCR |= _BV(SPE);
  i = 0;
  linecomplete = false;
  SPI.attachInterrupt();

  for (int d=0; d<2; d++) {
    lc.shutdown(d, false);
    lc.setIntensity(d, 8);
    lc.clearDisplay(d);
  }

  for (int led = 1; led <= 20; ++led) {
    ledMode[led] = MODE_IDLE;
  }

  nextFrameAt = millis();
  refreshAll();

  nextFrameAt = millis();
}
```

Listing 17: Essential parts of setup()

Host name	**ID**
Phaser	1
Deflector	2
Holodeck	3
Tenforward	4
Turboliftex	5
Sickbay	6
Readyroom	7
Shuttlebay	8
Transport	9
Jefferiestube	10
Brig	11
Nacella	12
Saucer	13
Warpcore	14
Photontorpedo	15

Table 5: Registered hosts and their corresponding identifiers.

Global variables used by these mechanisms

```
LedControl lc(5, 6, 7, 2);

#define SEG_DP  0
#define SEG_A   1
#define SEG_B   2
#define SEG_C   3
#define SEG_D   4
#define SEG_E   5
#define SEG_F   6
#define SEG_G   7

char line[20];
char message[20][20];
volatile byte i;
volatile bool linecomplete;
volatile bool processingdata;
volatile int linenumber;

const char begindata[20] = "begindata";
const char enddata[20]   = "enddata";
```

```
enum LedMode : uint8_t {
  MODE_IDLE = 0,
  MODE_SLEEP,
  MODE_BUSY,
  MODE_DONE,
  MODE_SLOW,
  MODE_TSLOW,
  MODE_OFF,
  MODE_UNKNOWN
};

static LedMode ledMode[21];

enum Color { COL_BLUE, COL_RED, COL_GREEN };

const uint16_t BLINK_SLOW_MS = 1000;
const uint16_t BLINK_FAST_MS = 250;
const uint16_t FRAME_MS = 20;

uint32_t nextFrameAt = 0;
```

Listing 18: Global variables and configuration

`processdata()` — **handling incoming data from the central control unit** The `processdata()` function is executed once the complete data transmission from the Raspberry Pi has been received via the SPI bus. It processes all lines stored in the `message[20][20]` buffer and updates the internal LED states accordingly. Each line represents the update for a single LED, for example: `"1:busy"`, `"2:sleep"`, `"3:slow"`, or `"20:off"`.

The function splits each line at the colon character (':') using the `strtok()` function and copies the resulting parts into a temporary buffer `buf[]`. The first token (`numStr`) contains the LED number, while the second token (`modeStr`) contains the corresponding status. These values are then used to update the LED state array via `parseMode()`.

The helper function `parseMode()` translates the textual state received from the Raspberry Pi (for example, `"done"`) into the corresponding internal state constant (e.g., `MODE_DONE`). This internal mode is then stored in the `ledMode[]` array, which keeps track of the current operating mode of all LEDs in the system.

Once all entries have been processed, `processdata()` clears the contents of the `message[][]` buffer, clearing the memory to correctly handle the next data transmission cycle from the Raspberry Pi.

Physical LED Placement The 3D-printed box for this module contains ten LEDs on the top side, ten LEDs on the bottom side, and additional internal space to house the hardware components. The software considers the following physical alignment of LEDs:

```
1   2   3   4   5
6   7   8   9   10
<empty space>
```

```
11  12  13  14  15
16  17  18  19  20
```

Connection of LEDs to MAX7219 Drivers Each LED is grouped with the LED directly above or below it on a shared DIGx pin. This means that, for example, LED 1 and LED 6 share DIG0 of the first MAX7219 driver. The following table shows how each LED is connected to the two MAX7219 drivers. This mapping is also visualised in the corresponding schematic.

LED #	MAX Driver	DIGx	Red	Green	Blue
1	1	DIG0	segB	segDP	segA
2	1	DIG1	segB	segDP	segA
3	1	DIG2	segB	segDP	segA
4	1	DIG3	segB	segDP	segA
5	1	DIG4	segB	segDP	segA
6	1	DIG0	segE	segC	segD
7	1	DIG1	segE	segC	segD
8	1	DIG2	segE	segC	segD
9	1	DIG3	segE	segC	segD
10	1	DIG4	segE	segC	segD
11	2	DIG0	segB	segDP	segA
12	2	DIG1	segB	segDP	segA
13	2	DIG2	segB	segDP	segA
14	2	DIG3	segB	segDP	segA
15	2	DIG4	segB	segDP	segA
16	2	DIG0	segE	segC	segD
17	2	DIG1	segE	segC	segD
18	2	DIG2	segE	segC	segD
19	2	DIG3	segE	segC	segD
20	2	DIG4	segE	segC	segD

Table 6: LED-to-MAX7219 driver mapping.

States and Their LED Actions We have defined the following LED behaviours for each possible state:

State	LED Status	Description
OFF	Off	There is extra space for future groups to be added; these LEDs are not in use.
IDLE	Steady Blue	Default state during startup of the Arduino.
UNKNOWN	Steady Blue	The LED is linked to a worker group, but we did not receive status data from the database.
SLEEP	Blinking Blue	Today and yesterday, no measurements were performed by this worker group.
SLOW	Steady Red	The expected end time is between 20:00 and 24:00.
TOO SLOW	Blinking Red	The expected end time is after 24:00.
BUSY	Blinking Green	The worker group is running normally; the expected end time is within the day and before 20:00.
DONE	Steady Green	The measurements of the worker group have finished.

Table 7: Overview of all possible LED states and their meanings.

Schematic: Status LED Board

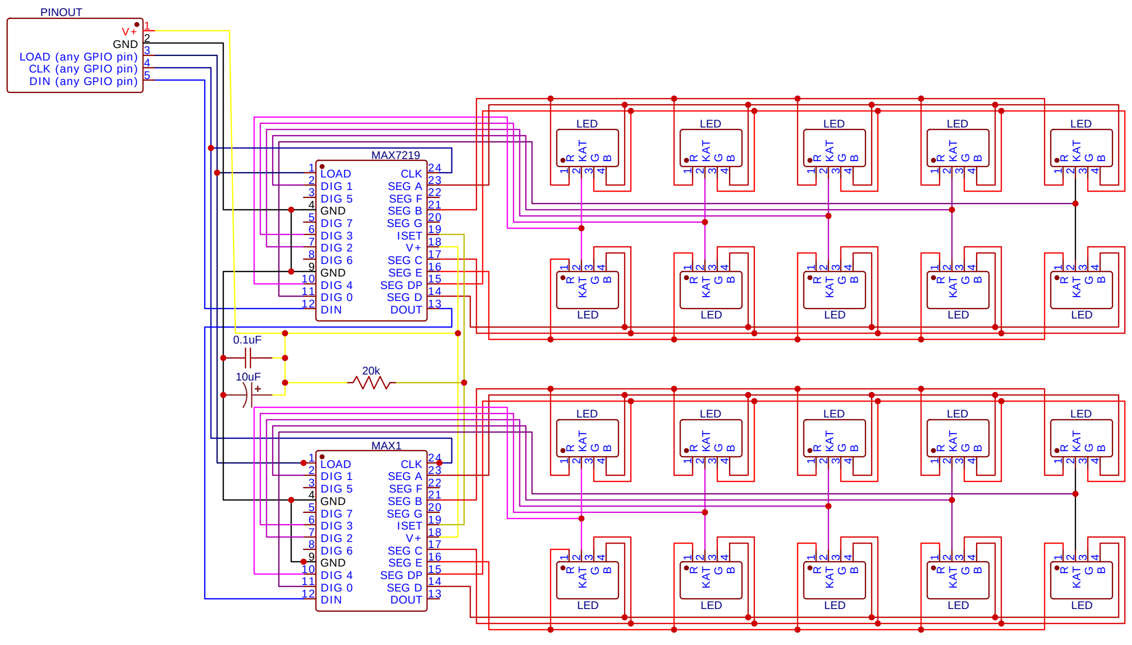


Figure 14: Schematic for the *Status LED* module.

E.6 Total Measurements Counter (scrapped)

This module would have displayed the total number of measurements completed that day. The maximum value shown would be 999,999,999, using **three** 3-digit 7-segment displays arranged in a 9-digit group. The control unit would have an updated total every five minutes; upon receiving this value, the module computes an increment rate so that the on-device counter smoothly reaches the provided total over the next five-minute interval.

Sadly, this module ended up being scrapped, as the top three segments seemed to continuously receive charge and light up at full power whenever the system was connected. We suspect this to be due to internal problems in either a driver or a seven-segment display, as extensive testing did not find any problems in either the software or the soldering. (With the exception of the connection to the ISET pin of the second driver not working, the fixing of which had no effect on the problem.) For this reason, we have decided to include the schematics for a possible future implementation.

Hardware Involved

- 1× Arduino Nano
- 2× MAX7219
- 3× BC56-12EW (3-digit 7-segment display)
- 2× 22 kΩ resistor

Software Involved

Not applicable.

Database Query Involved

Not applicable.

Schematic: Total Measurement Counter

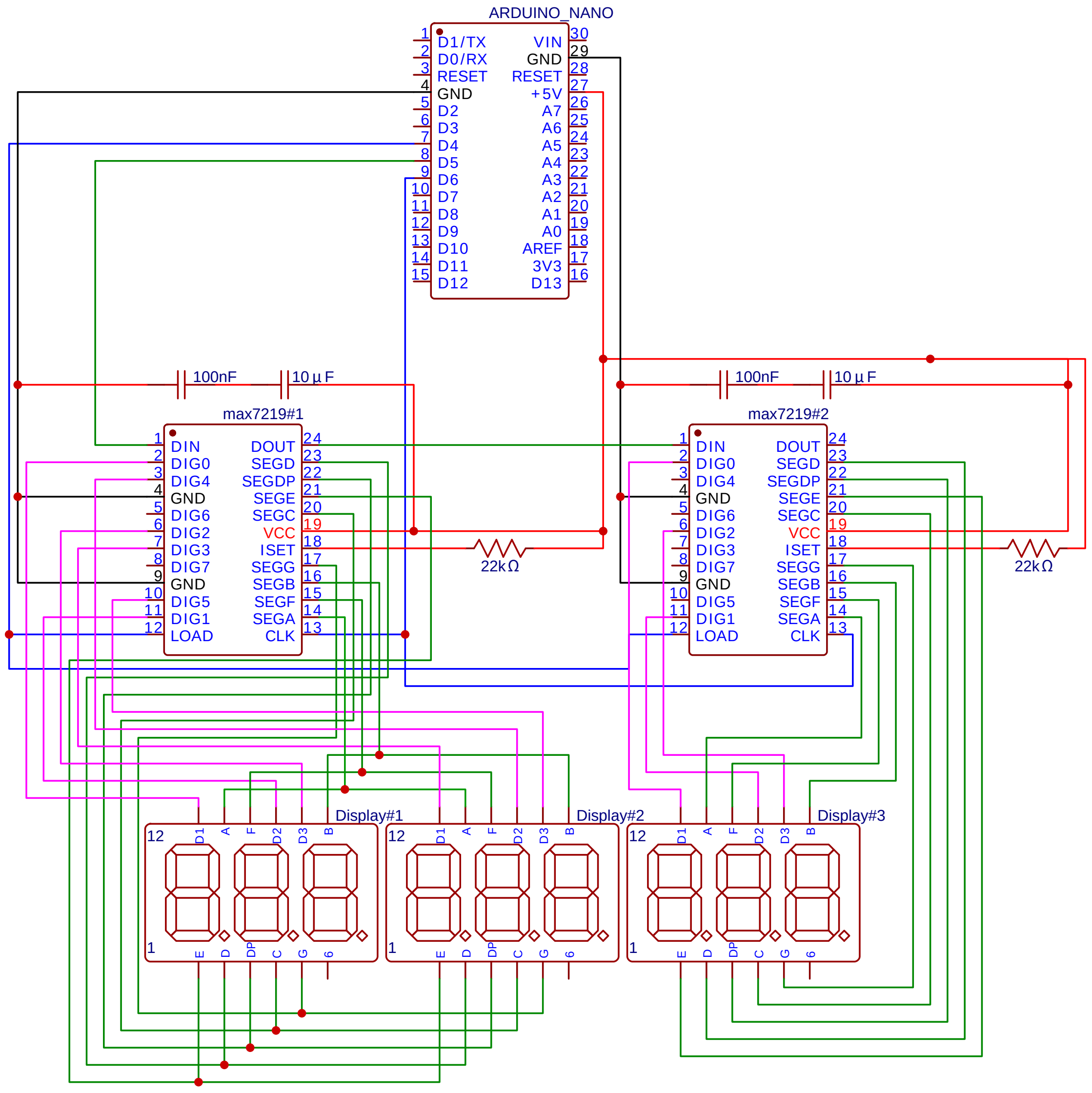
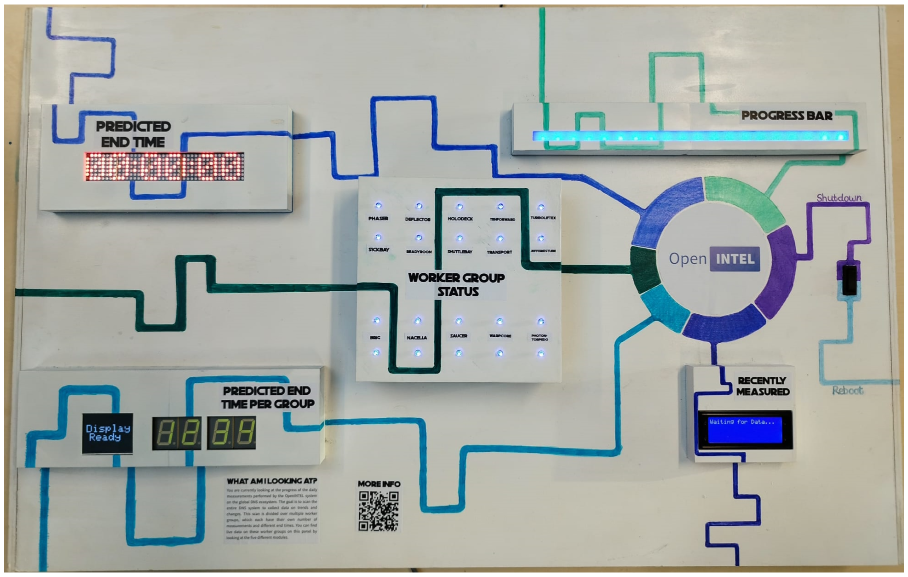


Figure 15: Schematic for the *Total Measurements Counter* module.

F Pictures

As the final result was not shown in another place in the report, below are some pictures of the final physical product for reference.



You are currently looking at the progress of the daily measurements performed by the OpenINTEL system on the global DNS ecosystem. The goal is to scan the entire DNS system to collect data on trends and changes. This scan is divided over multiple worker groups, which each have their own number of measurements and different end times. You can find live data on these worker groups on this panel by looking at the five different modules.

Figure 16: The front of the Blinking Lights Unit.

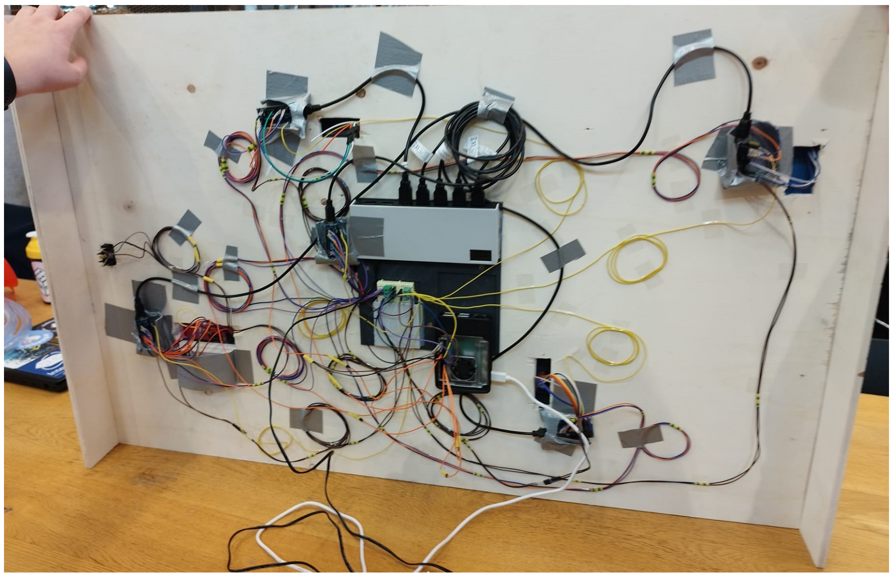


Figure 17: The back of the Blinking Lights Unit.