VRIJE
UNIVERSITEIT
BRUSSEL

Master thesis submitted in partial fulfillment of the requirements for the degree of Master of Science in de Toegepaste Informatica

A MODULAR HARDWARE PLATFORM FOR DYNAMIC DATA PHYSICALISATION

Milan Ilić

Academic Year 2023/2024

Promotor: Prof. Dr. Beat Signer
Advisors: Xuyao Zhang & Ingela Rossing

Faculty of Sciences and Bio-engineering Sciences

VRIJE UNIVERSITEIT BRUSSEL

Proefschrift ingediend met het oog op het behalen van de graad van Master of Science in de Toegepaste Informatica

A MODULAR HARDWARE PLATFORM FOR DYNAMIC DATA PHYSICALISATION

Milan Ilić

Academiejaar 2023/2024

Promotor: Prof. Dr. Beat Signer
Advisors: Xuyao Zhang & Ingela Rossing

Faculteit Wetenschappen en Bio-ingenieurswetenschappen

Abstract

Analysing data in an effective way is becoming harder and harder. The amount of data that one needs to comprehend can be too much for non-experts with existing tools. Data physicalisations exist to explore data comprehension beyond just the visual modality. Going one step further and making data physicalisations applicable to multiple problems and the result is a dynamic data physicalisation. With this being a relatively new field, a way to prototype and quickly create solutions is an interesting subject. That is especially true because it has been shown to have many cognitive benefits. This is mainly due to being a physical tool where users could explore and use a combination of modalities instead of just vision as for example charts on a monitor.

The goal of this thesis is to create a modular hardware platform for dynamic data physicalisation. The tool or platform that will be created has to be modular because its main use case is prototyping for developers and researchers. The solution consists of a platform and modules. This platform has multiple connectors where modules can plug in at any moment, hence the modularity. A module is an encoding of one or more physical variable(s). It would be impossible to create all modules that represent all possible combinations of physical variables. However, it is possible to support all possible combinations. This means that anyone could create a new module with their physical variables of choice by just following a set of rules. Finally, the used hardware is easily available and is not too complicated to replicate.

Acknowledgements

I would like to express my deepest gratitude to my advisor Xuyao Zhang for guiding me through this thesis. Her aid can be found in every aspect of this project. She always provided me with a clear path to the end.

I am extremely grateful to my promotor Prof. Dr. Beat Signer. First, for providing me with this opportunity and secondly for all the aid, guidance and feedback. Because of his course "Information Visualisation" at the VUB (Vrije Universiteit Brussel) I became knowledgeable about this interesting and still growing research field.

Thanks should also go to my other advisor Ingela Rossing. It was a pleasure working with this team of talented and charming people.

I am also grateful to the WISE (Web & Information Systems Engineering) lab for letting me use all of the required electronic parts, hardware components and 3D printer.

Lastly, special thanks to my family for their patience and emotional support. They always gave me the motivation that I needed. I was always met with support throughout my academic path, which began with studying chemistry and now ends with computer science/toegepaste informatica.

Contents

Chapter 1

Introduction

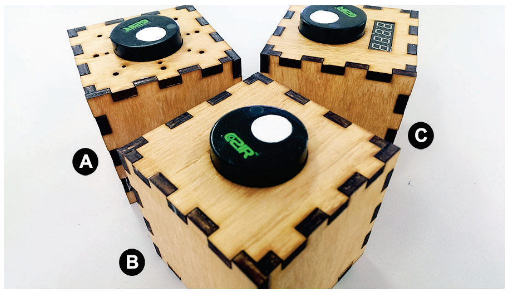
The amount of generated data increases every year [21]. This is due to the further digitisation of our world. Companies nowadays gather lots of information about their customers to later analyse. Not only companies but also researchers or governments have an increasing need in specific data for instance gathered via a survey, an interview, online behaviours or observations.

Analysing big, complex datasets can often be difficult to understand. This is even more true for non-expert people who are not specialised in a specific field. This gap will keep increasing unless good visualisations, explanations and physicalisations exist. Making this bridge to fully understand complex data is one of the focuses of this thesis, but not the main goal. The main goal is to create a platform or recipe for developers so that they can solve this issue of closing the gap that exists between complex datasets and non-expert people. Furthermore, this platform could also be used by researchers to create their own solution for a better understanding of data.

One solution to better understand a dataset is a data physicalisation. Examples of such physicalisations are Relief as shown in Figure 1.1a and EMERGE as shown in Figure 1.1c. These physicalisations consist of actuated rods that push either a flexible screen to form different landscapes or to set the rods to different heights and colour values to represent some dataset as one would do with a bar chart on a monitor. Another completely different example is shown in Figure 1.1b, where these 3 boxes represent different modalities, namely auditory by using sound, haptic with a vibration motor and visual with a screen. As the name implies, it is a physical artefact that represents data but is also dynamic. In the paper named "Opportunities and challenges for data physicalization" by Jansen et al. [13], a data physicalisation has been defined as "*a research area that examines how computer-supported, physical representations of data (i.e., physicalisations), can support cognition, communication, learning, problem-solving, and decision-making*".

Before going further into the details, let us think of some simpler examples to represent data to better understand them. One could make a data visualisation and plot data in some kind of chart. This would be a quick way to show how data has some kind of relationship or just to have a visual representation. This would typically be shown on a monitor and thus only use one modality, namely vision. The next step would be to create a data physicalisation. If the previously mentioned chart on a monitor was for example a bar chart, one could build a physical artefact of it through 3D printing, thereby extending the visual modality because it is now a 3D form in the real world and thus adding the tactile modality. A user would thus be able to understand even more, but a new problem has been introduced: we need to make a new physical artefact for every change in the original dataset. This can be very wasteful and inefficient. That is why we finally arrive at dynamic data physicalisations, which achieve the same advantages while being able to change with the dataset. It is thus not dependent on one fixed dataset, as before with the 'static' data physicalisation.

(a) Relief [16]



(b) Auditory, haptic and visual data representations [11]

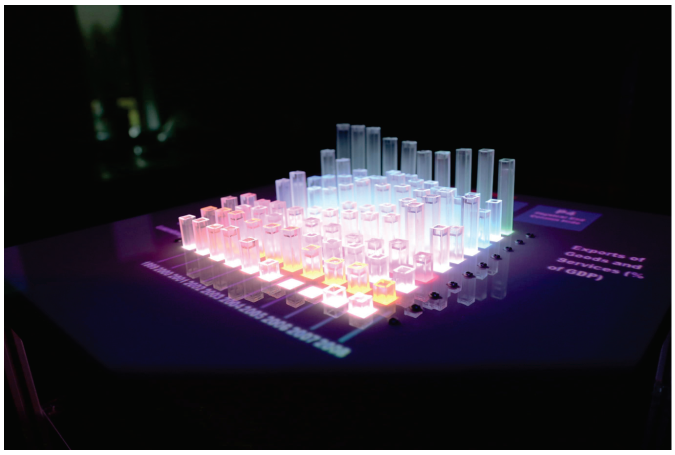

(c) EMERGE [25]

Figure 1.1: Various examples of dynamic data physicalisations

One of the reasons that data physicalisations were chosen for this thesis and to build a modular platform for it, is because it has many cognitive and learning benefits. The first one is being able to inspect, rotate and play with the physicalisation. This is known as "*active perception skills*" [13]. It also encourages one to explore what is in front of them. It is more engaging than just looking at visualisations on a monitor. Not being limited to only using the visual sense is also a benefit. That in particular applies to visually-impaired people.

Although dynamic data physicalisations are a relatively new field, many implementations already exist, including project FEELEX [12], Relief [16], EMERGE [25] and Laina [17] just to name a few. These examples will be discussed later.

Creating effective data physicalisations brings some challenges as described by Jansen et al. [13]. The first challenge is making digital data readable for humans. As with visual representations like a chart, a way is needed to map data to this representation. In the paper by Jansen et al., they proposed to combine multiple sensory variables so that they can complement one another. A second challenge is knowing how effective the used sensory variables or combinations of sensory variables are. Good constraints and guidelines are needed to effectively create a solution. Another challenge is the limited machinery and materials to represent certain physical variables, while also being cost-effective. These are problems that are holding back further research. It is therefore a goal of this thesis, to solve or better understand these mentioned challenges.

1.1 Problem Statement

The goal of this thesis is to develop a platform that serves for easy prototyping of dynamic data physicalisations. The solution will be a general platform with multiple modules. The modules each represent one or more physical variables, for example, temperature. They can be plugged into and out of the platform at any moment. Any low-level software and hardware are abstracted away. This will be referred to as plug-and-play. It correlates to the "*type 3*" as described by Lambrichts et al. [14] and the "*level 3 abstraction*" by Blickstein [2]. In both papers, this correlates to an electronics toolkit [2, 14]. In some ways, our platform will follow some similar ideas as a toolkit. Modularity is one of them.

Lastly, we will be able to create our own modules by just following a set of rules. These rules are hardware and software-related. The hardware correlates to the physical design of a module. This means for example what circuitry should be used to be compatible. The software correlates to a self-describing language. This is needed so that the platform knows what type of data is mapped onto this specific module.

The research questions that aim to solve the existing problems with dynamic data physicalisations and data representations in general are:

RQ1 How can plug-and-play capabilities be implemented in a platform that serves to create a dynamic data physicalisation?

RQ2 How can modules be made for a dynamic data physicalisation platform by following a set of rules and a self-describing language?

1.2 Thesis Outline

In Chapter 2, we dive deeper into the world of dynamic data physicalisations, by not only reviewing some examples and discussing them but also reviewing relevant work in terms of plug-and-play capabilities. This initial research gives ideas and shows what is (im)possible. The final section of this Chapter contains background information about state-of-the-art communication protocols. This practical information is needed to understand the following Chapters of this thesis.

Next up is Chapter 3 in which relevant communication protocols will be discussed. The advantages and disadvantages will be mentioned as well as argumentation why certain protocols are or are not used. By the end of this chapter, it should become clear what direction this thesis takes. The end product of this chapter is a solution that serves as the base for the implementation.

Chapter 4 will implement the design choices made in Chapter 3. It will start by explaining what hardware parts were chosen to implement the design solution of Chapter 3. Thereafter the software components will be explained. An important feature is that anyone will be able to create a module, that is why there will also be a discussion on how to make your own module. The Chapter will conclude with a technical evaluation to show what it is capable of.

In Chapter 5 the contributions will be discussed and some limits in the implemented prototype will be exposed. If someone wants to contribute this will be their starting point. Everything that can be improved will be mentioned here. The thesis ends with some concluding remarks in Chapter 6.

Chapter 2

Related Work

2.1 Plug and Play

The key aspects of the proposed platform are modularity and plug-and-play capabilities. Let us begin by looking at some state-of-the-art modular systems that already exist. This does not necessarily have to be about data physicalisation, since it is the modular part that is of interest here. Modularity and plug-and-play both correlate to the ease of use and playful nature of the model, that engages the user to explore.

One such modular system is called MorphMatrix, which is a shape-changing interface as seen in Figure 2.1. It consists of a 8×8 matrix where each block or 'pixel' can move up and down [7]. In actuality, it contains 2 modules of 4×8 matrices, that when put next to each other form one bigger 8×8 matrix. Each module is guaranteed to be a separate entity because they each have their own motor, camshaft, etc for moving the pixels. It is modular in the sense that one could keep adding modules and it would still work. This implementation is limited in the physical variables that it can represent. It is a first step towards the solution that is envisioned in this thesis.

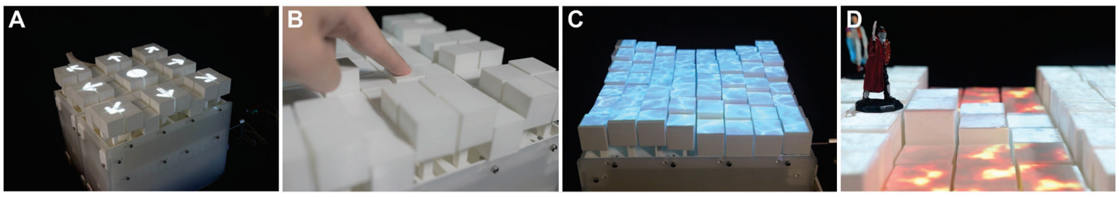


Figure 2.1: MorphMatrix [7]

Another example of a modular system is shown in Figure 2.2 and is called SensorBricks. It is an IoT (Internet of Things) toolkit for non-expert people to interact with data in a fun and interactive way [4]. It consists of input and output bricks that look like and are compatible with Lego bricks. The input bricks are the source of data that is then in turn provided to output bricks. For example, one user picks out a temperature input brick that measures the temperature in the room and then uses the digital output brick to show the temperature on a segmented display. All the controls happen via a dashboard that is provided with the toolkit. Input and output bricks are linked via this dashboard. Bricks can be placed anywhere and are charged when placed in the toolkit box. The toolkit is limited in the complexity of data that it can represent since it can only represent data from the input bricks. Because SensorBricks uses wireless communication, the location of each brick is unknown. Most importantly relative locations of bricks that are connected, either with Lego or on top of each other, are not known.

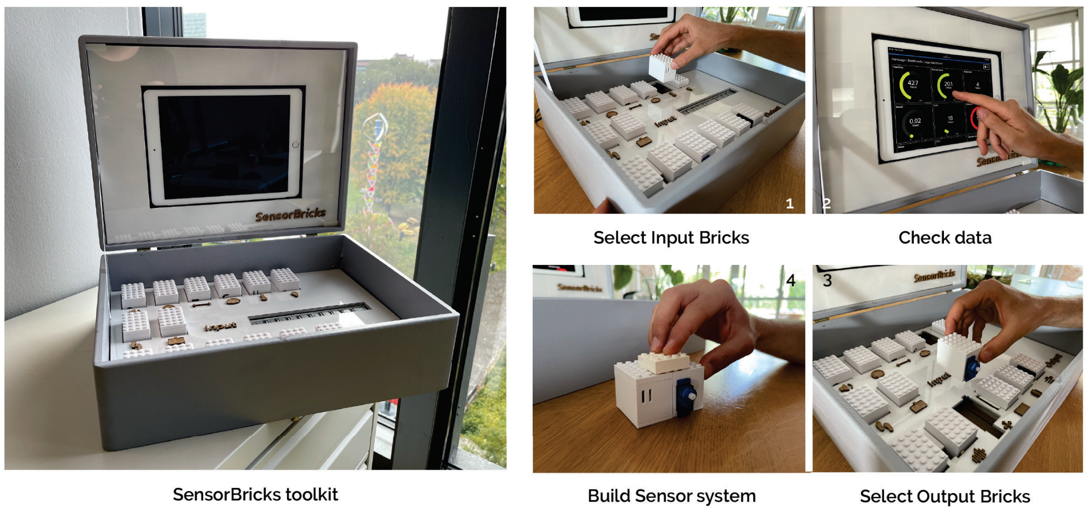


Figure 2.2: SensorBricks [4]

Another interesting plug-and-play example is called Zanzibar [27], which is a mat that can communicate with and determine the position of tangible objects that are placed on it as well as sense touch by a user as shown in Figure 2.3. Due to the design of this project, the placement of tangible objects is limited to the size of the mat. If one needs two tangible objects that are further away from each other, a big mat would have to be designed with lots of wasted space due to the nature of the design itself. The idea of using a surface to interact with tangible objects is certainly interesting but not further looked into because of the limitation on this flat surface.

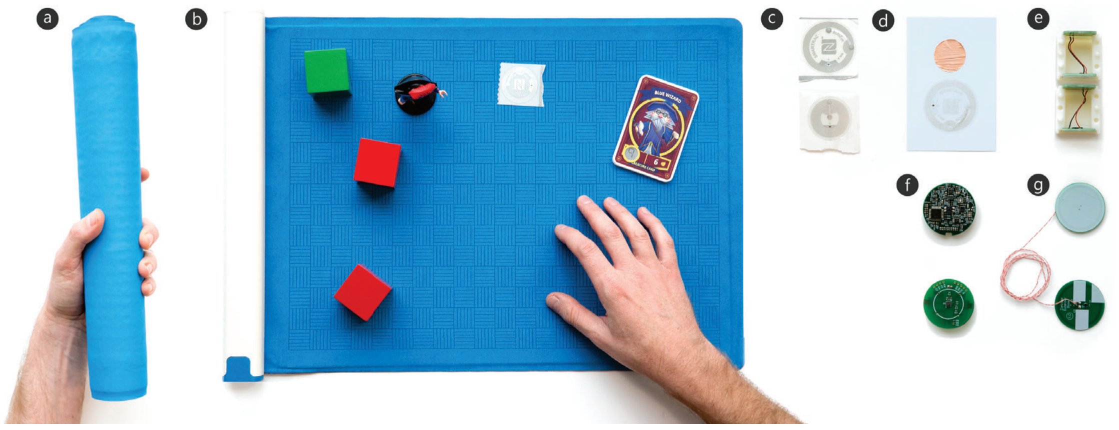


Figure 2.3: Project Zanzibar [27]

Bloctopus is a general USB hub, as shown in Figure 2.4, to which multiple sensors and actuators could be attached [22]. This general USB hub can be plugged into a computer or micro-controller. The advantage of this design is that it highly encourages to explore what the different modules can do. Because of the plug-and-play architecture knowing what each module does and how it works can easily be handled. The Bloctopus is not a standalone product or toolkit because a computer is still needed to interact with the modules. For this thesis, another philosophy is followed, where the end product acts more like a toolkit or stand-alone product without the need of an extra computer.

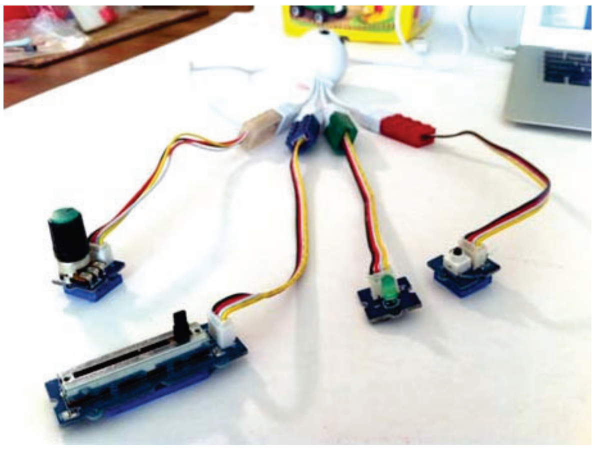


Figure 2.4: Bloctopus with multiple modules [22]

The .NET gadgeteer project from Microsoft, which is shown in Figure 2.5, is also interesting to look at [28]. It consists of a main board where different modules can be connected via a physical wire. A module is connected to the main board via a socket. This socket supports many different communication protocols like UART, I^{2}C or SPI. By supporting this amount of different connections, hardware compatibility is no longer an issue. One downside with this design is that all the sockets are located on the main board, which means that when more connections are needed a bigger main board with more sockets on it will be required. When using a main bus such as in the design proposed in this thesis, this problem is mitigated. By using a main bus, the project becomes more scalable in the sense that we do not have to worry about designing this main board with a set amount of connectors.

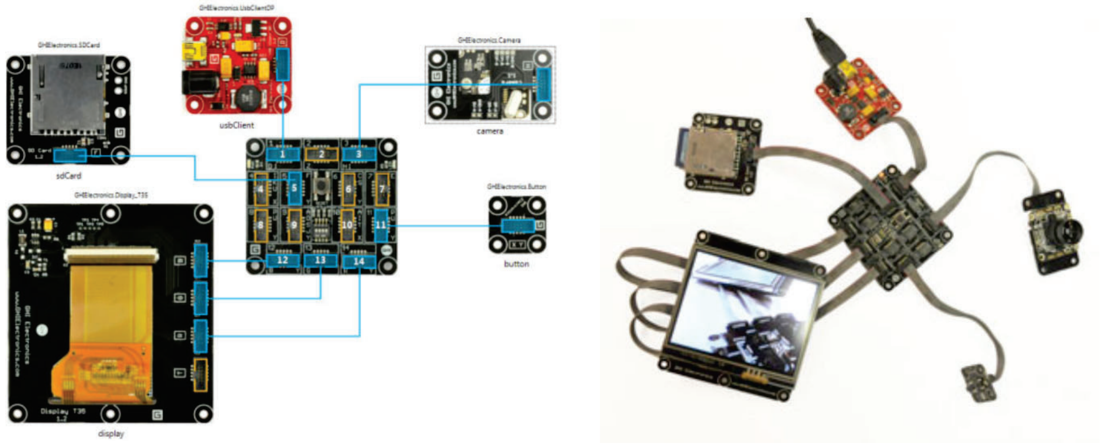


Figure 2.5: Gadgeteer in designer view (left) and assembled electronics (right) [28]

Another type of modular plug-and-play architecture exists where the modules are interconnected. An example of this is SoftMod [15]. The user can shape the prototype how they want as illustrated in Figure 2.6. A SoftMod prototype always consists of a master module and one or more slave modules. In a way, it is similar to the earlier discussed SensorBricks, as shown in Figure 2.2, because it also uses the concept of input and output modules that interact with each other. SoftMod uses an I^{2}C as a communication bus between modules. It is the master module that assigns addresses to each of the slave modules either directly or via other slave modules. Another interesting feature is the tracking of the topology. The topology or spatial placement of modules with regard to one another is gathered and visualised. Because the modules have four connectors (top, bottom, left and right) their placements and connections can be visualised.

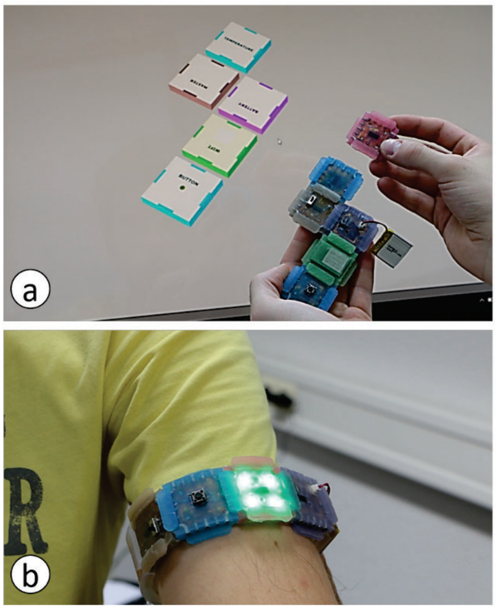
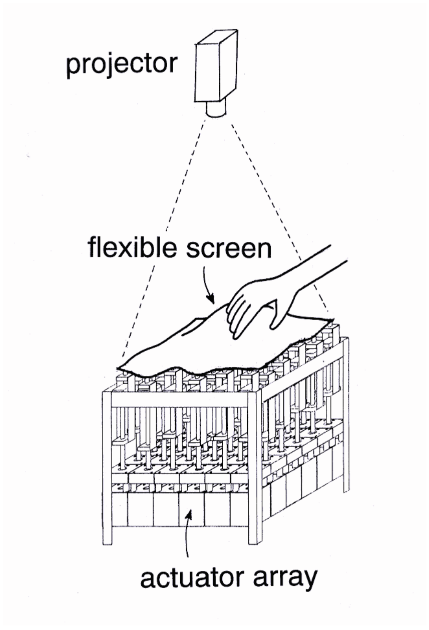


Figure 2.6: SoftMod [15]

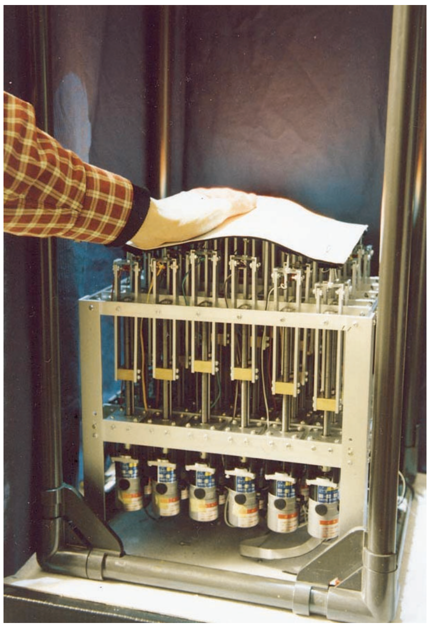
2.2 Dynamic Data Physicalisation

Now that the modularity of the proposed system has been explained, it is possible to start explaining what solutions would look like. By using this modular platform, a developer should be able to create a dynamic data physicalisation of their choice. Let us look at and review some existing dynamic data physicalisations. Since this is a relatively new field, there are not that many implementations as for example the 'static' data physicalisations. Nonetheless, some relevant projects will be looked at.

Let us start by looking at project FEELEX which consists of actuated rods that push a flexible screen into a shape [12]. An image could also be projected onto this flexible screen, as shown in Figure 2.7. This allows a user to feel and see at the same time, using the tactile and vision modalities for a better understanding. This is called a shape display. The FEELEX project is already 20+ years old. But it still laid the foundation to build upon. Some of the next projects/papers will extend this idea to the modern world.



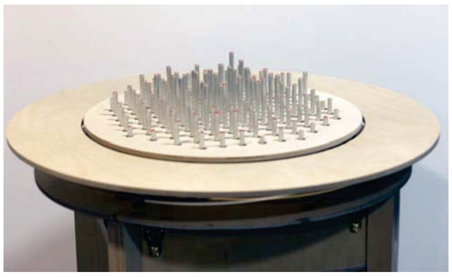
(a) Feelex schematic representation



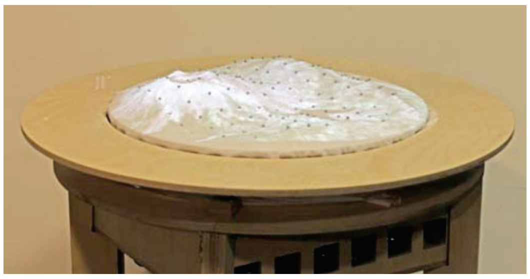
(b) Feelex

Figure 2.7: Project FEELEX: Adding haptic surface to graphics [12]

Modern variants like Relief [16] can for example be used to display a landscape in more detail. An example of what this landscape in Relief looks like can be seen in Figure 2.8. Some areas that Relief tries to improve upon are scalability and cost. Decreasing the cost by making the project from commercially available parts.



(a) Relief aluminum pins



(b) Relief with surface over the pins

Figure 2.8: Relief: A scalable actuated shape display [16]

Actuated rods can also be used to implement a dynamic physical bar chart like EMERGE [25] in Figure 1.1c. EMERGE is a physical three-dimensional bar chart.

It is implemented using actuated rods that can push each individual LED bar vertically up and down. The rods and surrounding area have touch detection. This also allows a user to do data exploration. With this data exploration, it becomes possible to select a subset of data, filter data, reorganise data and many more operations. A user study was performed to demonstrate its effectiveness. Some interesting findings were that due to the speed of the actuators and noise generated, some participants became "*hesitant to interact with the system*" [25]. A user study was performed on 17 participants to determine if the benefits of static data physicalisations carried over to this implementation. Further research on this topic is needed to form conclusive results.

In the paper named 'Investigating the Use of a Dynamic Physical Bar Chart for Data Exploration and Presentation' [26] a user study was performed with EMERGE, where the participants had no time limit and were provided with an unknown dataset. In this paper, they concluded from the user study that "*Our findings provide promising evidence that physicalizations encourage people to engage in data exploration, to support them in data-based presentations, and have the potential to support thinking about data in ways that are different from non-physical visualizations*". This is another reason why we should continue this research in dynamic data physicalisations.

To end the related work papers, let us look at something completely different. In the paper titled 'The visual and beyond: Characterising experiences with auditory, haptic and visual data representations' [11], 3 probes were made representing different modalities: auditory, haptic and visual. As shown in Figure 1.1b, the probes are small boxes that fit in an adult's hand. For the auditory modality, the frequency of sound was used: a low data value corresponds to a low frequency. The output sound is generated by a small speaker inside the cube. The haptic modality is represented by vibration motors where data is mapped to a certain motor speed. The visual modality was implemented to show familiarity. A number is displayed on the segmented LED display. To test the effectiveness of each probe and gather the user's experiences, a repgrid study and micro-phenomenological interview were performed.

These were just some examples of how dynamic data physicalisations could be implemented. Many more examples of actuated rods exist like Laina [17], where a run plan is shown as data on a more artistic use of actuated rods.

The main takeaway from these examples is that data can indeed be mapped to different modalities. This can be done in many different ways as we have just seen in the examples above. Going further than only the visual modality is the driving force.

However, these papers mainly followed different approaches. One can imagine that coming up with such designs and actually creating them takes a lot of time.

That is exactly why the proposed modular platform with modules is essential. It allows developers to quickly prototype and build their envisioned solution. Some initial testing can be done and it will be the first step towards their final design.

2.3 Background

Now that we have seen some examples of both modularity and dynamic data physicalisation, it becomes possible to start reasoning about a potential architecture for this thesis. Before going into the details about the design choices of this architecture, it is important to lay the foundation of communication protocols. That is why this section is dedicated to explaining some background knowledge that is needed to understand the next Chapter, namely Chapter 3 which uses some of the described communication protocols. These protocols are needed for the communication between hardware components.

First, it is important to distinguish between the two existing types of serial communication, namely synchronous and asynchronous. With synchronous communication, there is always one wire that sends timing signals to synchronise the clocks on both ends [9]. Asynchronous communication does not have this extra wire because it relies on start and stop bits to indicate when a message starts and ends [1].

There are many types of synchronous and asynchronous serial communication protocols. We will look at some state-of-the-art protocols, and later, in Chapter 3 determine which is the best fit for this project.

2.3.1 Universal Serial Bus

Let us start by looking at a communication medium that is well known, the universal serial bus or USB for short. It has become a standard connector in our daily lives: from charging a smartphone to using a USB stick for data transfer. By looking at these examples, it already becomes obvious that USB does not only serve for the transmission of data, like with a USB stick but it also provides power to the connected device. This is a master-slave architecture, meaning that there is always one host or master and one or more connected devices or slaves [9]. The USB protocol is illustrated in Figure 2.9. It consists of four wires, one for power, one for ground and two for the transmission of data.

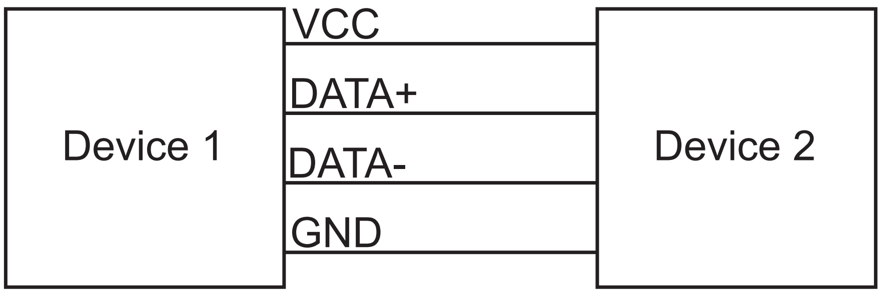


Figure 2.9: USB protocol

2.3.2 Inter Integrated Circuit

The I^2C (Inter Integrated Circuit) protocol uses 2 wires or signal lines. One for data (SDA) and one for the clock speed (SCL) [9]. A representation of this protocol is shown in Figure 2.10. The communication is synchronous because the clock speeds are connected via a wire and synced. It is a master-slave architecture, where multiple masters are possible (but only one master at a time). Only the master can send messages to the slaves and it does not have an address like the slaves.

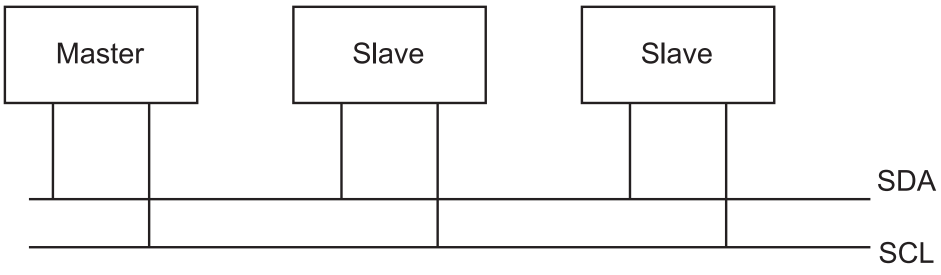


Figure 2.10: I^2C protocol

System management bus (SMBus) and power management bus (PMbus) are two well-known protocols that are based on I^2C [20]. Both use the same principle as I^2C .

2.3.3 Serial Peripheral Interface

SPI (Serial Peripheral Interface) generally uses four wires to communicate between two devices as explained by Dhaker in the article named ‘Introduction to SPI interface’ [8]. It is also synchronous and thus has a wire for the clock signal and is full-duplex, meaning that it can send data in both directions at the same time. The chip select (CS) connector selects one device as the main device. This main device is responsible for generating the clock on the SCK wire. The other device, known as the subnode, synchronises with this clock so that they both have the right timing for sending messages to each other. Now it becomes possible to send and receive data via the MISO (Main in, subnode out) and MOSI (Main out, subnode in) wires. These connections are illustrated in Figure 2.11.

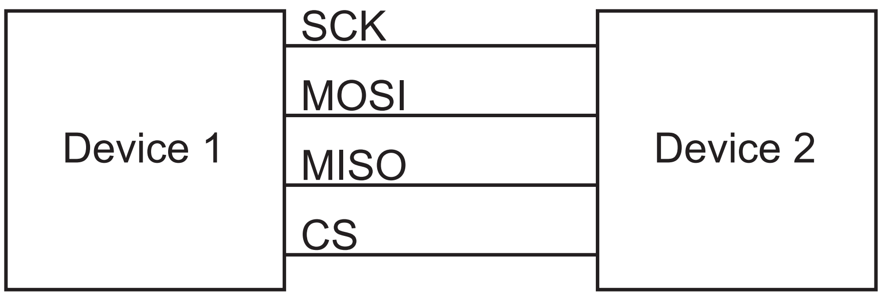


Figure 2.11: SPI protocol

2.3.4 Universal Asynchronous Receiver-Transmitter

UART (Universal asynchronous receiver-transmitter) uses, as the name implies, asynchronous serial communication. As previously explained, asynchronous communication means that there is no connection between the two devices to synchronise. The two connected wires that UART uses are RX which stands for receiver and TX which stands for transmitter [19]. Logically the transmitter of one device is connected to the receiver of the other as shown in Figure 2.12. The UART protocol takes as input a data bus that sends data in parallel. This is then converted to serial and transmitted to the other device where it is again sent to a data bus in parallel.

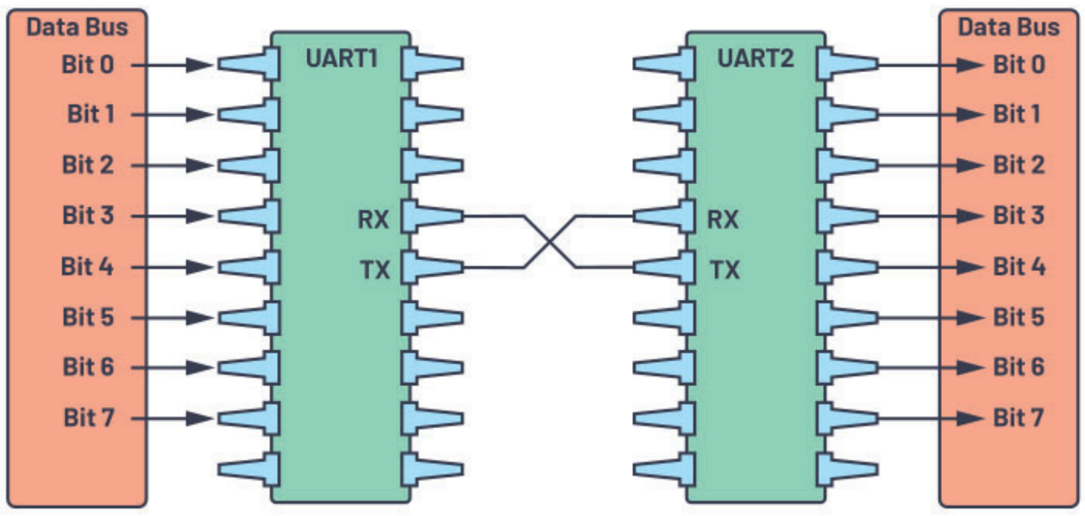


Figure 2.12: UART with data bus [19]

2.3.5 Controller Area Network

The CAN (controller area network) bus is mainly used in the automotive industry for cars. The relation between the CAN bus and a car is illustrated in Figure 2.13. The red line represents the CAN bus where all the data will be sent over [18]. Many different parts use this bus to communicate and address messages only to those that need it. Of course, every connected device can see what is happening on the bus but can only read a message if it is addressed to said device. There is a main bus where a device can connect to. Every connected device needs an address and receives messages via that address.

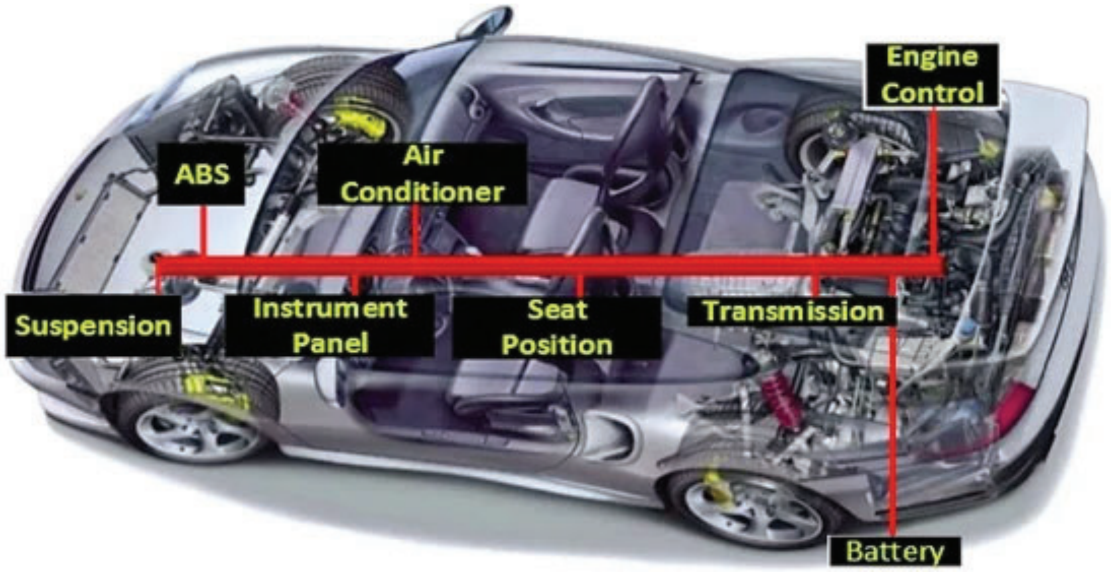


Figure 2.13: CAN used in a car [18]

CAN uses two wires and is half-duplex [29]. Half-duplex means that data can be sent in both directions but not at the same time. The signalling is message-based and has a maximum rate of 1Mb per second [6].

Because CAN was designed for the automotive industry, it has some benefits like no short distance limitation and no crosstalk can occur. In cars, for example, many moving parts create noise that could interfere with and change the value of some bits sent over the CAN bus. This is why the CAN bus has a "*high immunity to electrical interference, and an ability to self-diagnose and repair data errors*" [5]. It is robust because there are many error-checking protocols.

In order to have a working CAN connection, a CAN transceiver and controller are required. A CAN controller is present in some microprocessors. It is also possible to use an external CAN controller. The transceiver and controller are the 'workhorses' of the CAN bus. Sending a message over the CAN bus works as follows. Suppose we have an external CAN controller. A microprocessor sends a message to the controller. The controller checks if the bus is available. If so, the message is sent to the transceiver. The transceiver will convert the message to the CAN bus level. Receiving a message works in a similar way. Every message contains the unique identifier of the receiver. A message is sent over the bus and can be seen by all connected devices but can only be read by the receiver.

In the application report of Texas Instruments by Corrigan the CAN protocol and structure of a CAN message is explained in full detail [6]. A CAN message or frame has a maximum length of 128 bits with the actual data being 8 bytes (64 bits) maximum. As shown in Figure 2.14, the actual data is only one part of the actual message. The first part of a message namely SOF (single dominant start of frame) is one bit that indicates when a message starts. The next part is an 11-bit identifier. This is to whom it is addressed. RTR, IDE and r0 are 1 or 2 bits long and are not relevant but they are mentioned because they need to be part of the message to be valid. DLC or data length code is 4 bits long and describes how many data bytes are used. Now comes the 8 bytes or 64 bits of data that we can transfer. Next comes the CRC or cyclic redundancy check. This is used to detect errors because it is a checksum. ACK stands for acknowledge and is used to verify that the CAN message has indeed been received by the receiver. Lastly, EOF or end of frame marks the end of the CAN frame and IFS or inter-frame space is the time between two frames. As we have seen, a CAN message or frame exists out of many parts and not only the data that we want to send.

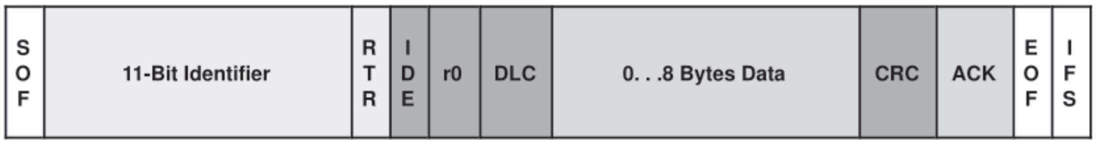


Figure 2.14: CAN message [6]

An example use case of the CAN bus is driving multiple stepper motors [30]. In this paper, a microprocessor is used to communicate between a CAN transceiver and a stepper motor. The microprocessor has a built-in CAN controller and is connected to the stepper motor driver that controls the stepper motor.

2.3.6 Power Line Communication

Another interesting protocol is called power line communication or PLC for short. PLC sends data over a power cable. It becomes possible to power a device and transmit data via one power cable [23].

Chapter 3

Solution

Going back to the goal of this thesis to design a platform for rapid prototyping of dynamic data physicalisations. Many possible solutions for such a system can be created as seen in Section 2.1. Modules in combination with the platform should be seen as a complete solution to some problem. Meaning that the modules should be adjacent to one another rather than scattered around for example.

A project like EMERGE could be the outcome of prototyping with the proposed platform and modules. Each actuated LED rod or group of multiple could be seen as a module. It is then possible to have multiple such modules and connect them to the platform to create a solution. In this particular case, the proposed platform could be the first step in creating a project like EMERGE. A developer could create this with the platform and modules to test it out and see how their original idea would translate to the real world. One could already see why this rapid prototyping makes sense, instead of building this project from scratch.

A first idea could be to implement the modules as IoT (Internet of Things) devices. This would mean that their communication is wireless and there is no spatial awareness (i.e. no easy way of knowing where a module is placed relative to the other modules). It could be done with some GPS tracking or camera tracking, but there are better solutions to this problem. Also as just mentioned the modules should not be scattered around which promotes the use of a wired connection. When using a wired connection between the platform and modules the connectors could be addressed and thus we would instantly know the positions of modules relative to each other.

Adding more possible connectors to the platform should be an easy task. Because every project that is built using this platform will have different requirements, adaptability is the key. The easiest way to add connectors is to have a main communication bus where modules could plug into.

The idea of having a platform where modules could plug into is illustrated in Figure 3.1. The platform will contain a main bus to communicate with the modules. Each module represents one or more physical variables as we can see in Figure 3.1 where module 1 encodes temperature and module 2 the size. In essence, a module encodes data in some way or form. A 'tactile module' with a vibration motor inside can, for example, vary in vibration pattern and strength. Let us start by looking at some options for the main bus so that our platform can indeed communicate with these modules. Note that in Figure 3.1 the bus is illustrated as one line but will of course be more complex than this simplified illustration.

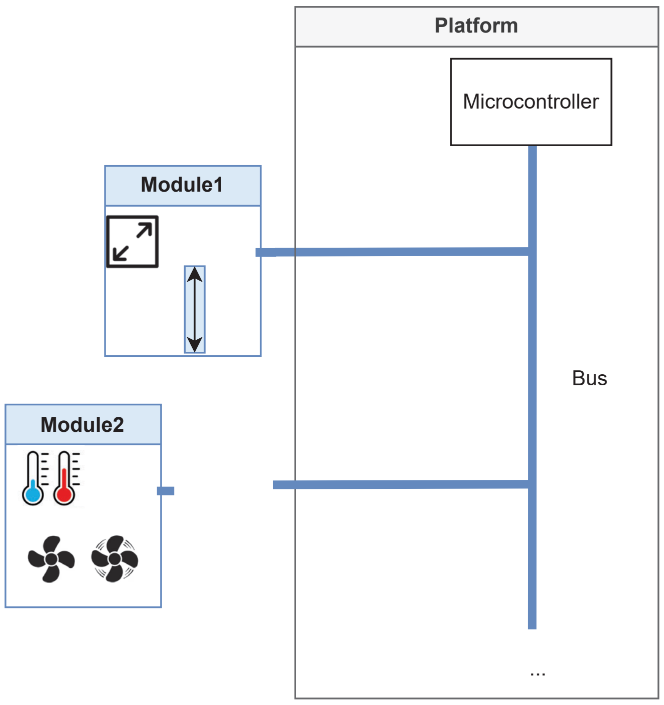
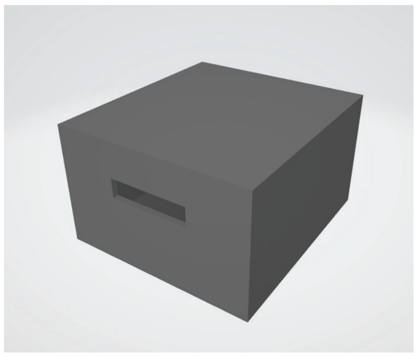


Figure 3.1: Architecture of platform and modules

An example of a module in 3D is illustrated in Figure 3.2a. Every module has holes on the side for cooling and to displace air if a fan were connected. In Figure 3.2b, the platform is illustrated with two such modules plugged in. This platform can fit exactly six modules.


(a) 3D model of one module with cooling holes

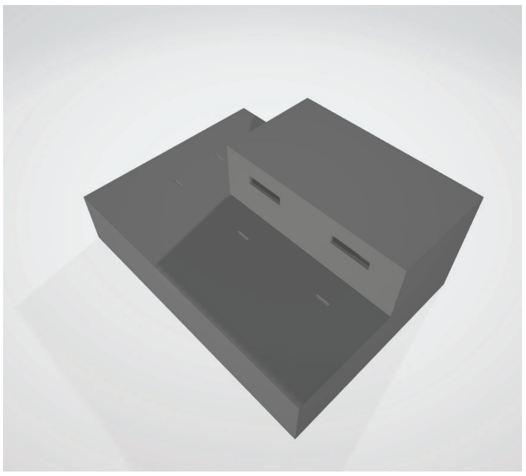

(b) 3D model of the platform with two connected modules

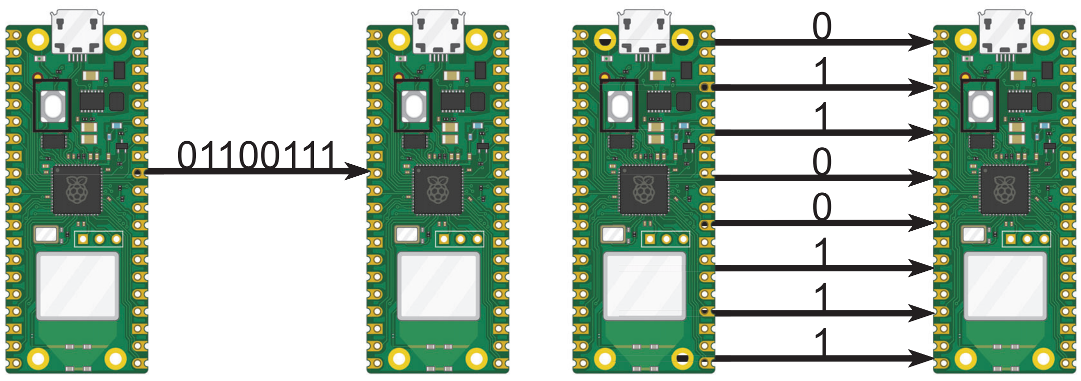
Figure 3.2: 3D model of the platform and modules

As just mentioned, a communication bus is needed for the communication between the platform and all of the modules. When a module is plugged into the platform, it needs to send a message to the platform to tell what kind of physical variable the module encodes. The platform needs to know when modules are still plugged in or if they are taken out. The platform can set the module to a certain data value and read what value it contains at the moment. All of this communication needs to happen over a bus where multiple modules can be attached to. It has to be scalable not only in the sense of adding modules but also in adding platforms together so that it becomes one bigger platform. It also has to implement plug-and-play so that any module can be plugged in or out at any moment. The platform also has to know where a certain module is plugged in so that it has a kind of spatial awareness. This correlates to the platform having multiple connectors where modules can plug into. The platform will then be able to distinguish between these different connectors.

IoT devices were already disregarded but what about Ethernet? By definition, Ethernet is not only hardware but also software [24]. A LAN (Local Area Network) would be needed to communicate between devices using Ethernet. Each connected module would have a MAC (medium access control) address assigned to it as a unique identifier. With PoE (Power over Ethernet) it also becomes possible to provide power over this same Ethernet cable. On paper, this all seems like a good solution but we also need to take hardware components into account. A microcontroller of a popular brand will be chosen to use inside both the platform and the module. This solution was chosen because we do not want to overcomplicate the project, hardware compatibility, the many available projects and software libraries.

With such requirements, it already becomes harder to implement a LAN network with modules connected to the platform via Ethernet. Using a simpler more low-level protocol seems like a better fit for these hardware requirements.

In general, there exist two types of buses: serial and parallel. Serial communication sends all bits over a single bus, while parallel communication uses multiple busses to send bits as shown in Figure 3.3 [3]. Using parallel requires many more wires which adds costs and would unnecessarily complicate the project. This is the case because every module has to be connected to the platform. If every connection would be parallel it would require many more wires than a serial connection. Furthermore, the length of this connection is limited due to crosstalk (interference between the parallel connections). The effects of crosstalk increase with cable length. Due to the mentioned limitations of parallel communication, it is not further looked into. In what follows we will look at serial communication and why it is the better choice for this project. In short, serial communication is chosen because it is easier to implement and more practical for this implementation.



(a) Serial communication (b) Parallel communication

Figure 3.3: Serial and parallel communication

3.1 Potential Communication Buses

We will look at the protocols that were explained in Chapter 2.3, and determine which is the best fit as a main bus for this project, while also highlighting some of the other protocols used in this thesis.

3.1.1 Universal Serial Bus

At first sight, USB looks like a promising start because the platform needs to provide power to a module as well as send and receive data. But this is a master-slave architecture, meaning that there is always one host or master and one or more connected devices or slaves [9]. Something that has yet to be mentioned is that micro-controllers will be used for these communication protocols that we are investigating. Unfortunately, most of these micro-controllers and the one that we ultimately select are not capable of acting as a USB host.

3.1.2 Inter Integrated Circuit

Again, the master-slave nature of this protocol does not match the requirements for the communication between a module and the platform. Both have to be able to initiate contact without relying on others to send a first message. Because only one master can ‘control’ the bus at a time, it would also be possible to switch between masters. However, this is not the intended way to use this architecture and can be cumbersome to implement and make it work. Because SMBus and PMBus use the same principle as I^2C, the same arguments can be applied here and it will not be further looked into.

3.1.3 Serial Peripheral Interface

This communication protocol already uses more wires, namely four, and this without providing power. Each connected device would also require an additional wire for the clock. Another possibility is to interconnect all subnodes in a daisy chain. However, this would mean that data flows from the master to the first subnode, then to the second and so on. This is not ideal and does not scale well for this project. The many wires and additional wire for every connected device and master-slave architecture make it not a good solution for this thesis.

However, some hardware is configured to communicate with SPI. As we will later see in this thesis, the SPI communication protocol will be used with certain hardware because they were designed this way. It is important to note that the arguments and conclusion that were given are only relevant in this project when using it for the main communication bus. When using SPI for communication between only two devices this is no longer relevant.

3.1.4 Universal Asynchronous Receiver-Transmitter

As mentioned before serial was chosen over parallel communication. More importantly, the UART protocol is a master-slave architecture. That is also why this protocol is not used as a main bus for this thesis. It is however relevant to mention it because some hardware requires the UART protocol to communicate with.

3.1.5 Controller Area Network

As explained in Section 2.3.5, there is a main bus where a device can connect to. Every connected device needs an address and receives messages via that address. In this situation, modules could send an initial message when they are plugged in. The platform can also address messages to the modules. The initial inspiration for using CAN came from its use in the automotive industry where different hardware parts use it to communicate to for example send data to the instrument panel [18].

The message-based CAN protocol seems a good fit for this project. By using either a micro-processor with a CAN controller built-in and an external CAN transceiver or a micro-processor with an external CAN controller and transceiver it is possible to communicate over this bus. Anyone is able to initiate contact or send a message at any given point. That is exactly the problem that we were trying to solve. Adding more modules to the system only requires plugging into the main bus to be able to communicate.

3.1.6 Power Line Communication

PLC could be useful to detect if devices are connected and also to provide a unique address to this spot. It is not message-based or protocol-based so all modules need to communicate via different power wires with the platform. PLC is increasingly becoming more popular for the implementation of smart homes and smart grids [10]. Where the existing power grid in a house is used as the communication medium for smart appliances. As high voltage could be dangerous, PLC for this project must be used with care and low enough voltages.

3.2 Potential Architectures

Multiple communication protocols were explained each with arguments why or why not to use it. Taking into account these advantages and disadvantages multiple solutions were created. In the end, one solution will be implemented, and the other solutions build up to the final one.

The idea is to have a platform where different kinds of modules can be plugged into. These modules represent one or more physical variables for example temperature and size. In Figure 3.1 a simple illustration of the platform and modules was shown. The three most important parts when designing this architecture will now be discussed. The platform needs to know where a module is located spatially. This refers to having set addresses at each of the connectors of the platform where modules can plug into. This is essential to, for example, change the value of a specific module. As a second point, the platform also needs to know if a module is still plugged in or not, to know if it is still possible to set the value of that module and also if it is still the same module that is connected. When we physically switch module 1 and module 2 in Figure 3.1, the platform should be informed that firstly they are no longer there and secondly that a new module has been plugged in. Otherwise, the platform could think it is setting some height or size instead of the temperature. The third and last point that we need to focus on is what physical variable a module encodes and how this is done. When a module is plugged in via some connector it needs to send what kind of module it is to the platform. This is done so that the platform in turn knows how it can use or set this module to some ordinal or nominal value. In what follows, I will show four different architectures. Each of these has its advantages and disadvantages. I will also keep the structure of this simple representation, with a platform, a module and the connections and a main bus as in Figure 3.1. A module needs power, for this, we have 2 solutions either by battery power or with a power connector. For now, a power cable is chosen.

3.2.1 Solution 1: Power Line Communication

Figure 3.4 depicts the first solution. It is the simplest solution as it only uses PLC between the modules and platform. The red wire is for power and the black is ground. Thus the power line is also used as a communication channel to send data over. Here we have a main bus that is PLC. The platform will always be some kind of micro-controller. The module will also consist of a micro-controller and a physical variable. The module and platform communicate via PLC. A special PLC controller is needed for this to be able to detect and send messages via this medium. This architecture only requires two wires and is the most minimal solution. However, this would be hard to implement and bring some shortages. First, the platform has no way of knowing where a module is. This means that the connector where the module is plugged into has no address. Furthermore, it is not possible to connect multiple devices to this bus because there is no way of knowing who needs to receive which message. We will build further upon this solution because the idea of receiving power and sending data over that same wire is certainly interesting.

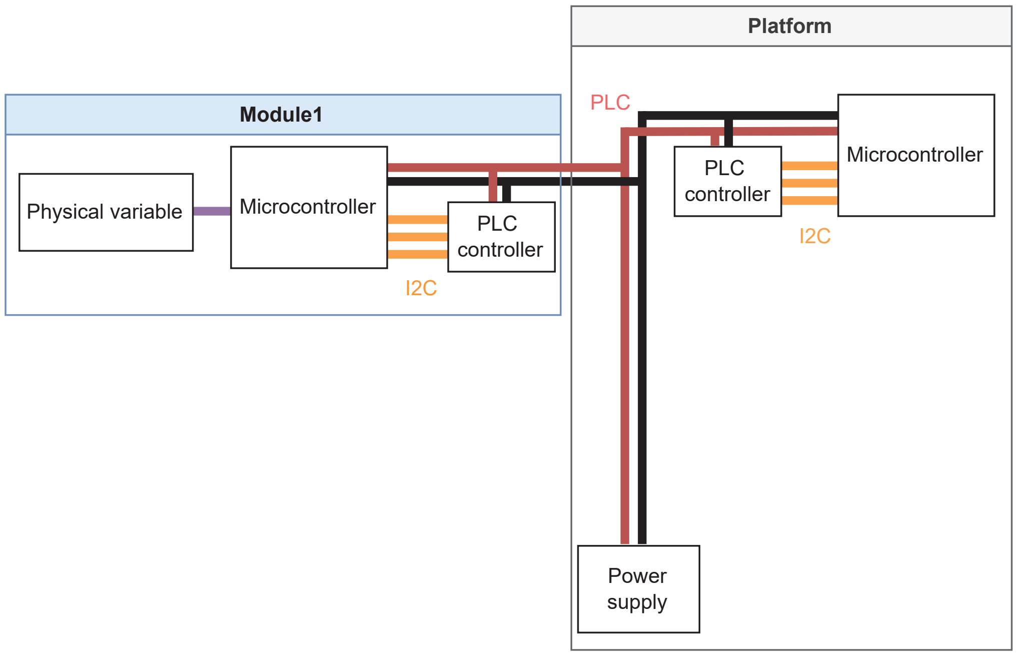


Figure 3.4: Architecture solution 1

3.2.2 Solution 2: Message-based Power Line Communication

The second architecture is almost the same but with one difference. This is because the communication is now message-based because of a controller that converts it to a CAN-like message, which is shown in Figure 3.5. This means that a connected module can send a message to the platform without being asked to send a message. The DCAN500[1] is capable of this message-based PLC. This already solves some issues but the addressing problem remains. Also, the implementation of DCAN500 proved to be very hard because of limited information.

[1]https://yamar.com/product/dcan500/

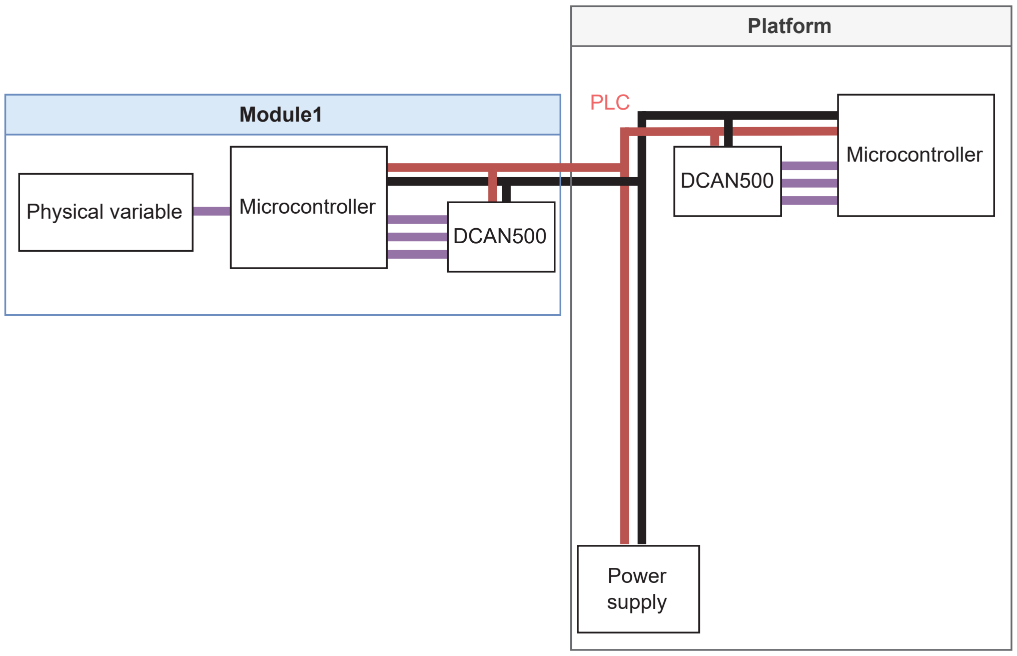


Figure 3.5: Architecture solution 2

3.2.3 Solution 3: CAN as Main Bus and PLC Addressing

The third architecture is shown in Figure 3.6. It is more complicated than the other solutions and has a four-wire connector (two for CAN and two for power) now instead of the 2 wires from before. PLC will still be used but now together with CAN. Because the CAN protocol is message based an identifier is needed for every connected module. This identifier or address will be provided when the module is plugged into the connector. The idea is to send the address via PLC to the module. In Figure 3.6 this correlates to the red and black wires that connect the micro-controller (in this case an Arduino) to a power supply of the platform. Both the module and platform have a PLC to be able to communicate via this medium. When a module is plugged in it will first receive an address via this PLC bus, and then the module can now use this to communicate on the main bus. This main bus uses CAN and requires 2 wires: CANH and CANL. This correlates to the green and yellow wires in Figure 3.6. The module will send what kind of module it is to the platform via a self-describing language that will be explained later. Because it is message-based the module can do this. Everything is then ready for the platform to set the module to certain values. The reason that CAN was used is because it is message-based, only 2 wires are needed and it is a very robust communication channel. This means that if some electrical interference occurs it does not affect this bus. Also because of the availability and cheap cost of a CAN transceiver and controller. To see if a module is still plugged in, it will send some value to inform the platform that it is still there. When the platform does not receive this special message, it will assume that the module is plugged out. This solution requires a four-wire connector and 1 micro-controller per connector because of the PLC. The underlying problem that we have to keep in mind is to not overload the CAN bus with so-called ‘hearthbeat messages’. These are small CAN messages that a module sends to the platform to inform it that the module is still there. If many such modules do this the load could become too much to handle.

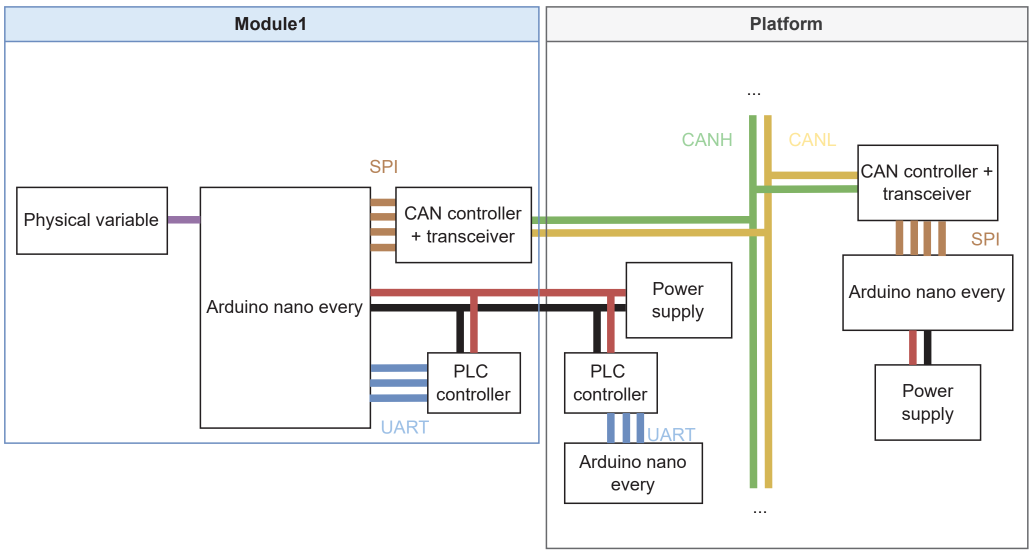


Figure 3.6: Architecture solution 3

3.2.4 Final Solution: CAN as Main Bus with Addressing Arduino

Lastly, a fourth solution was created without the use of PLC. It still uses similar principles to the previous solution but with some minor adaptations. The main idea remains the same: there is a main CAN bus where all the communication will happen. The only use of the PLC in solution 3 was to send the unique address of the connector where the module is plugged in. This task will now be solved by a ‘address micro-controller’. In Figure 3.7 we can see that the PLC is omitted and there is an extra wire because it is now a five-wire connector. The extra micro-controller in this case an Arduino will send the address to the connected module.

One such micro-controller can solve addressing for multiple connectors. In Chapter 4 the solution as illustrated in Figure 3.7 will be implemented. This solution was chosen because of the limited availability of the specific PLC controller. Most commercially available PLC controllers require to work with high voltages because they were mainly designed for communication over the existing power cables as for example inside a house, which requires 220 Volts. Furthermore, solution 3 requires two PLC controllers and one Arduino per connector. By using one extra wire for the connector as illustrated in Figure 3.7, all of this additional hardware can be omitted.

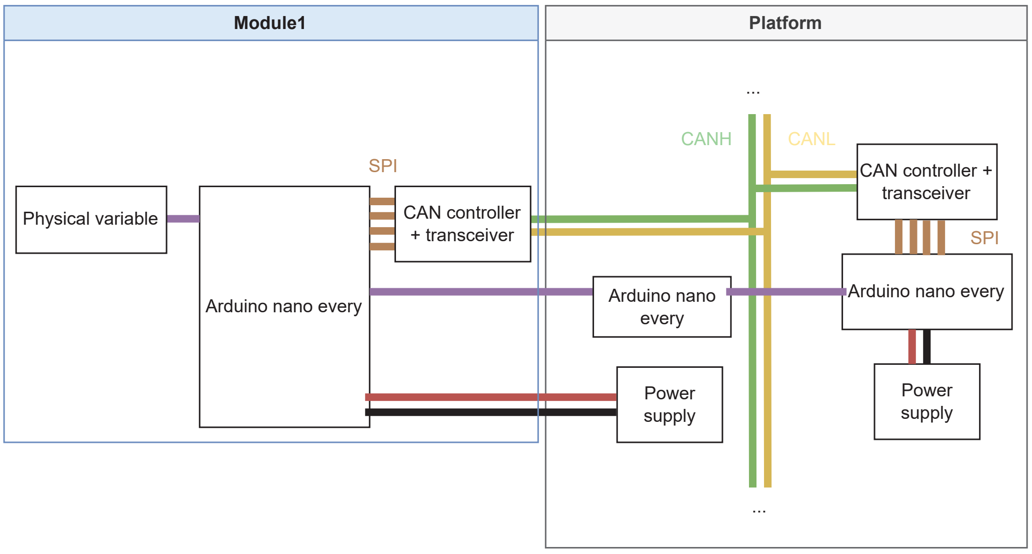


Figure 3.7: Architecture final solution

3.3 Modules

Before implementing the proposed architecture shown in Figure 3.7, the behaviour of a module must be explained. There can exist many different types of modules, each with one or a combination of physical variables. When a module is plugged in, it will send what physical variable(s) it represents to the platform. After this, the module will wait until the platform sets it to a certain value. This behaviour can be seen in the state diagram in Figure 3.8 and in the sequence diagram in Figure 3.9.

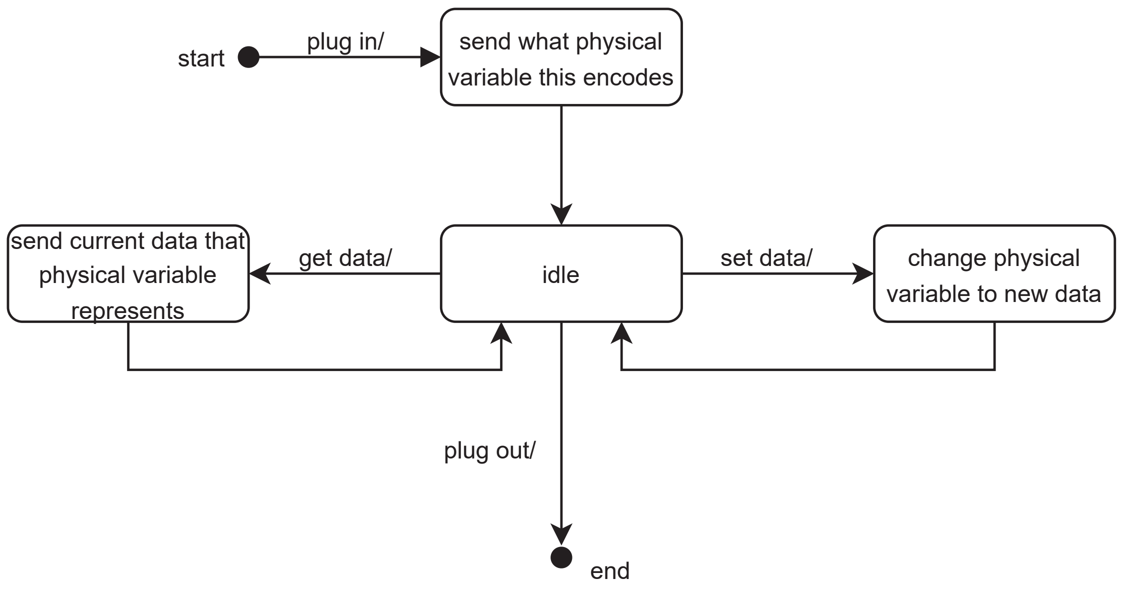

start
plug in/
send what physical variable this encodes
send current data that physical variable represents
get data/
idle
set data/
change physical variable to new data
plug out/
end

Figure 3.8: State diagram of a module

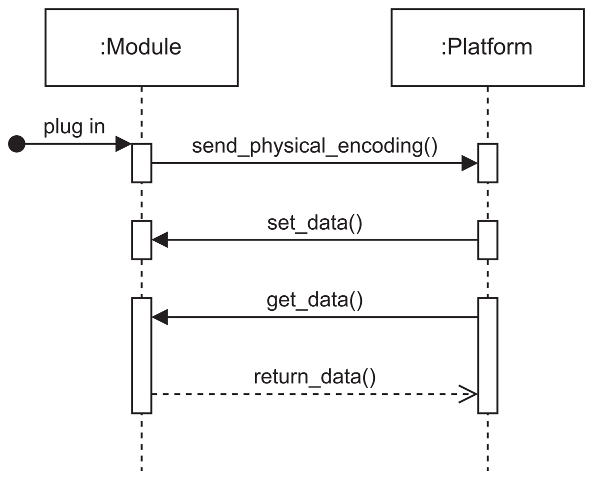


Figure 3.9: Sequence diagram module and platform

In addition to the mentioned behaviours of a module, a heartbeat is also used. This is a small message that will be sent over the CAN bus to inform the platform that a module is still plugged in. It has to be done every so often. When the platform does not receive such a message after a set amount of time, the module will be regarded as plugged out. This will now be more explained in Chapter 4 alongside the full implementation of the chosen solution, which was illustrated in Figure 3.7.

Chapter 4

Implementation

4.1 Hardware

To implement the suggested architecture shown in Figure 3.7, there are multiple options. The solution should of course implement all the mentioned choices, for example, plug-and-play. If a new developer wants to contribute to this platform by making a module of their own, they should be able to do so. That is why the hardware parts will be chosen such that they are easy to come by as well as being affordable. This means that in theory, anyone with the required knowledge will be able to make a contribution.

Modules have to be able to communicate with the platform. The logical choice to implement this is by using a micro-controller. Generally, this is a small computer board with enough computational power to solve basic tasks. The 2 most known brands are Raspberry pi[1] and Arduino[2]. Each module will have at least one physical variable. These can be represented by, for example, fans, actuators or LEDs. To have the best compatibility with these hardware components and because of the low-level nature of the needed micro-controller Arduino was chosen. Still in the Arduino family there exist many different types. The first one that seemed interesting for this project was the 'Arduino UNO R4 minima' because of the built-in CAN controller, many I/O pins and support for the most commonly known communication protocols. As mentioned before, to have a working connection via CAN, a CAN controller and a CAN transceiver are needed. The 'Arduino UNO R4 minima' in combination with a CAN transceiver would work for this thesis. Let us also look at another interesting Arduino before making a hasty decision. The 'Arduino nano every' has similar benefits as the 'UNO R4 minima' while also being much smaller and cheaper. The only disadvantage when compared with the

[1]`https://www.raspberrypi.com/`
[2]`https://www.arduino.cc/`

previous Arduino is that it does not have a built-in CAN controller. At first, one might think that this is a clear-cut decision. In reality, there are arguments for using both models. The advantage of having a built-in CAN controller proved to be none existent in reality. This is because of the availability of a CAN controller and transceiver on one board. It is also worth mentioning that it is not an either/or problem. Both of these micro-controllers with extra hardware to communicate with the CAN bus are compatible with each other. In the end, I chose the 'Arduino nano every' because of its smaller form factor and cheaper cost. For convenience and because of the same arguments as for the module, the platform will also have this same Arduino. The pin layout of the 'Arduino nano every' is shown in Figure 4.1, and the specifications are shown in Table 4.1.

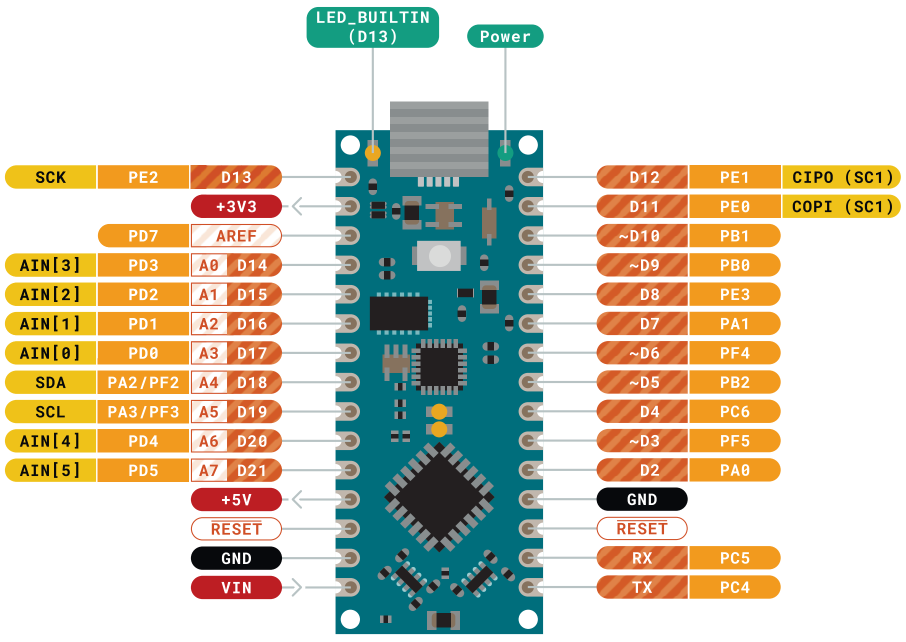


Figure 4.1: Arduino nano every[3]

[3]https://store.arduino.cc/products/arduino-nano-every

Micro-controller:	ATMega4809
Operating Voltage:	5V
Clock Speed:	20MHz
SRAM:	6KB
EEPROM:	256byte
Size:	45 × 18mm

Table 4.1: Arduino nano every specifications[4]

For the communication with the CAN bus, a CAN controller/transceiver named 'MCP2515 CAN-bus module' is used. From now on this will be referred to as simply 'MCP2515'. This board contains a CAN transceiver and controller. An Arduino will connect to this board with the SPI connection protocol and also power and ground. It requires 5V as power input. At the other end of the board, there are 2 connections: CANH and CANL. This is for the communication with the CAN bus. It is a small board that can easily fit inside a module. In Figure 4.2 we can see the board. The Arduino and MCP2515 are the base hardware components for both the platform and the modules.

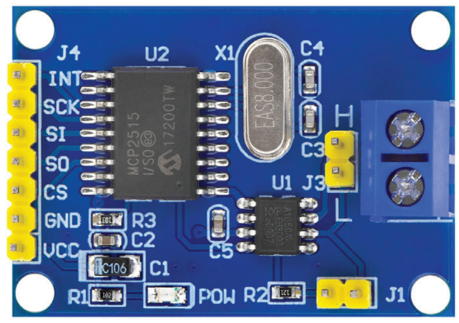


Figure 4.2: MCP2515 CAN bus module[5]

As illustrated in Figure 3.7 there is one cable for power that connects to a module. This power cable will power the Arduino as well as a small power supply. It is 9V DC power that will be provided by the platform. The power supply will supply 5V and 3.3V as these are the most commonly used voltages for hardware

[4]https://store.arduino.cc/products/arduino-nano-every
[5]https://www.az-delivery.de/en/products/mcp2515-can-bus-modul

used with Arduino. This means that all the hardware will be powered by this power supply which is shown in Figure 4.3. It is a breadboard power supply. From now on the 'MB102 breadboard power supply' will simply be referred to as 'MB102'.

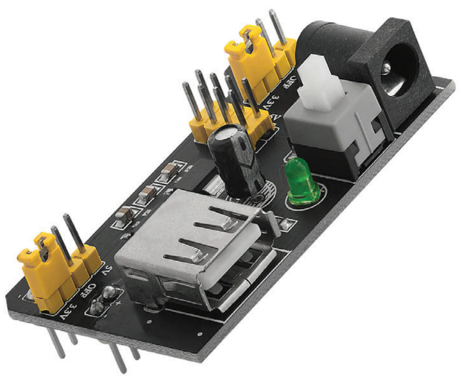


Figure 4.3: MB102 breadboard power supply[6]

Now that all of the hardware components of a module have been explained, a case is needed to keep everything neatly together and to also not expose the user to the internals. A 3D printer was chosen to make the case. Firstly, a 3D model was made using the Freecad software[7]. The 3D model was then printed using the Ultimaker S5 3D printer[8].

4.1.1 Platform

Let us start with the design of the platform before looking at the modules. The platform consists of 2×Arduino's, 1×MCP2515 and batteries for power. The platform is a 3D printed box that has the size of exactly 6 modules in a 2×3 grid as illustrated in Figure 4.4. At the top of the platform, there are 6 connectors that a module can plug into. These connectors consist of 5 wires. The power and ground go to the power and ground bus that the platform provides. The CANH and CANL wires plug into a main bus where all of the CAN wires will be plugged into. There is also a 5th wire for the determination of the addresses of modules. This wire plugs into the 'addressing' Arduino. It is responsible for sending the

[6]`https://www.az-delivery.de/en/products/mb102-breadboard`
[7]`https://www.freecad.org/`
[8]`https://ultimaker.com/nl/3d-printers/s-series/ultimaker-s5/`

addresses of the connector where a module is plugged into. The inside of the platform, with all the wiring and hardware components, is shown in Figure 4.5.

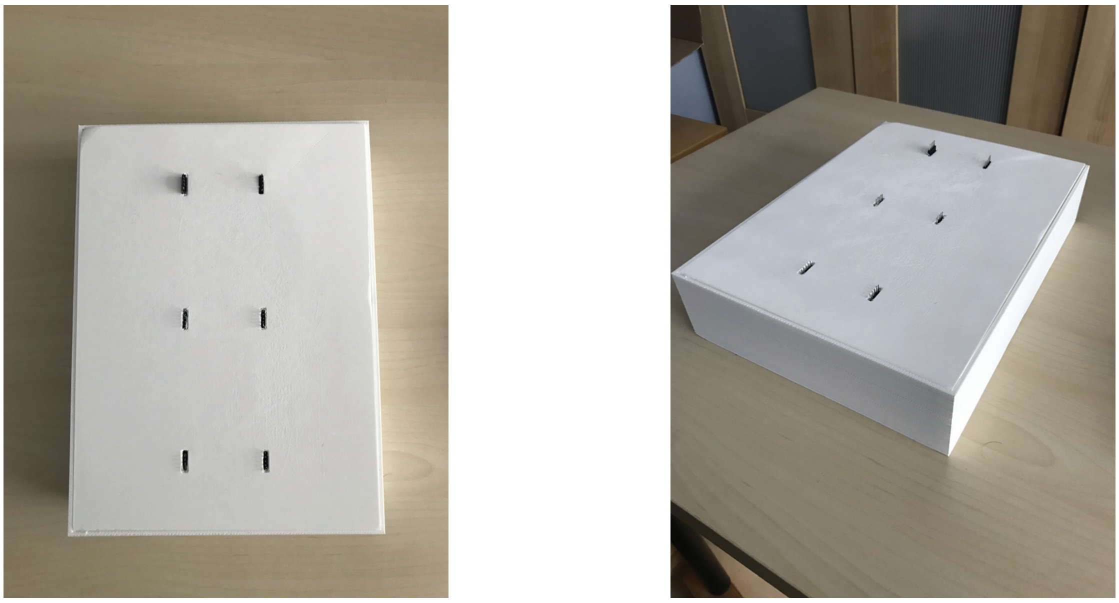

(a) Platform top (b) Platform side

Figure 4.4: The platform

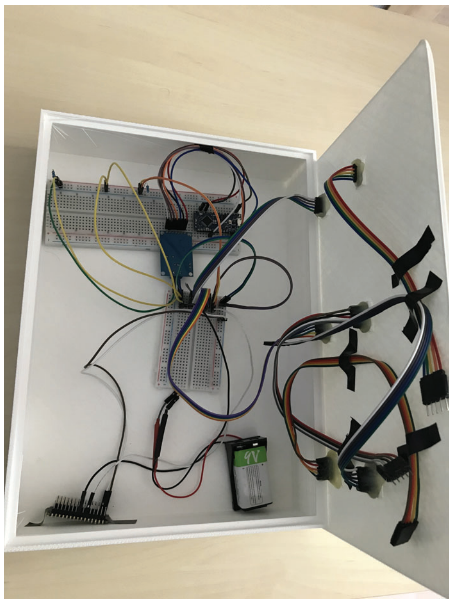


Figure 4.5: Inside of the platform

The complete wiring and hardware components of the platform are shown in Figure 4.6 alongside two connectors (the two bundles of five wires on the left side of Figure 4.6) that show how and where the connector wires plug into. This is exactly what is inside the 3D printed case that was represented in Figure 4.4 and Figure 4.5.

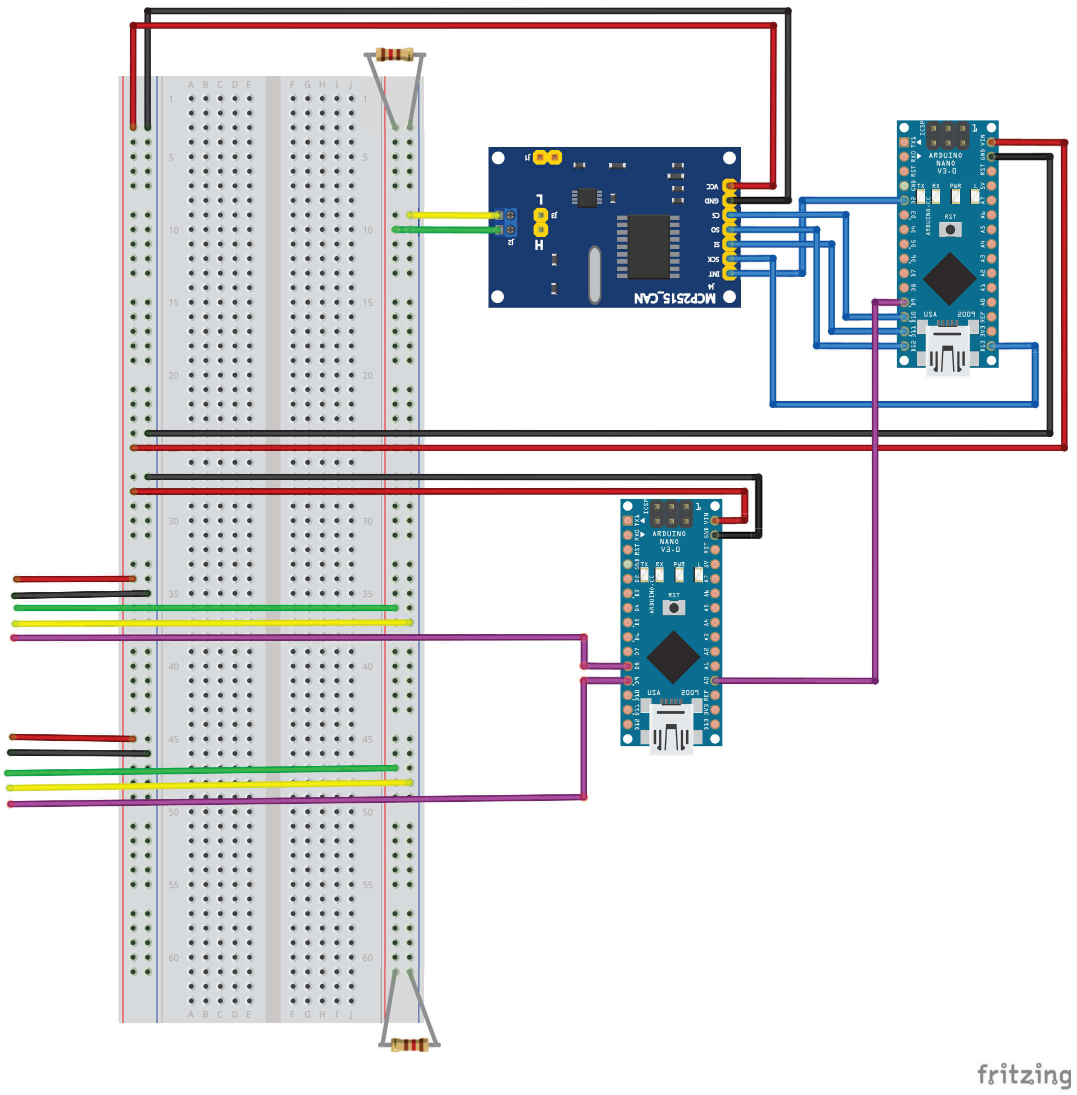


Figure 4.6: Wiring of the platform with 2 connectors

4.1.2 Module

As mentioned before the basic hardware of a module consists of an Arduino, MCP2515 and MB102. It is then possible to add any hardware that represents one or more physical variable(s). This hardware must either be powered by 3.3 Volts or 5 Volts because that is what the power supply inside a module provides. An example implementation is illustrated in Figure 4.9. The case for a module is shown in Figure 4.7. This is a small box that can just fit the required hardware. There is a small hole at the bottom. This is for the wires that are needed to connect to the platform. The rectangular holes on both sides of the module have two functions. First, it provides airflow so that modules do not overheat. Secondly, this was necessary for a fan because it needed to pull air from somewhere. This part of the module is meant to be the same for every physical variable. For the top part of the module, one can either use the provided top parts or design one of their own. The module in Figure 4.7 has a fan on top that blows air. This fan represents the temperature physical variable. It can exactly fit one fan of $40 \times 40 \times 20$mm. The wiring and hardware components are shown in Figure 4.8.

Note that the module shown in Figure 4.8 does not include the 'MB102' to power all of the hardware except for the Arduino. In an ideal situation, it would be best to implement it as illustrated in Figure 4.9, but for the proof-of-concept, this makes no difference.

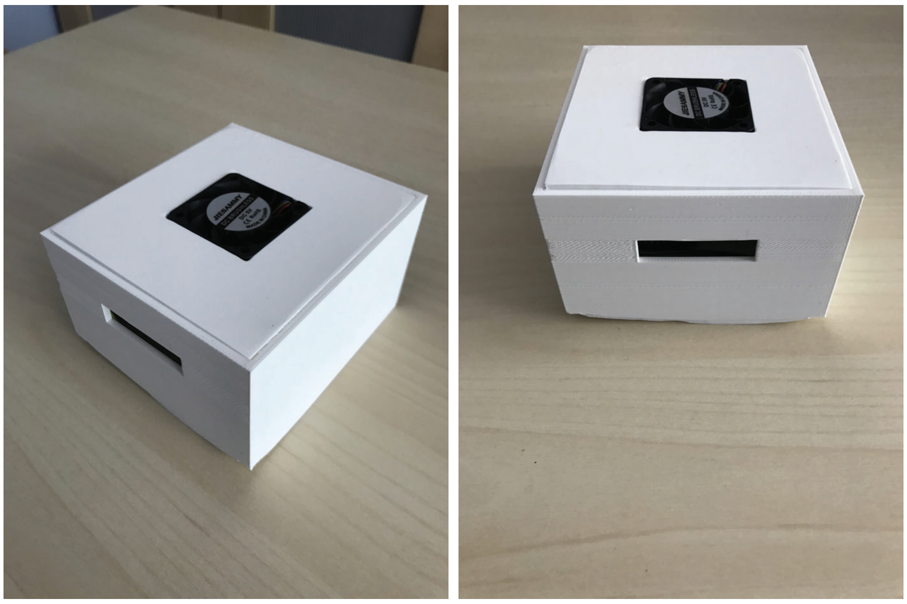

(a) Module top (b) Module side

Figure 4.7: Module with a fan

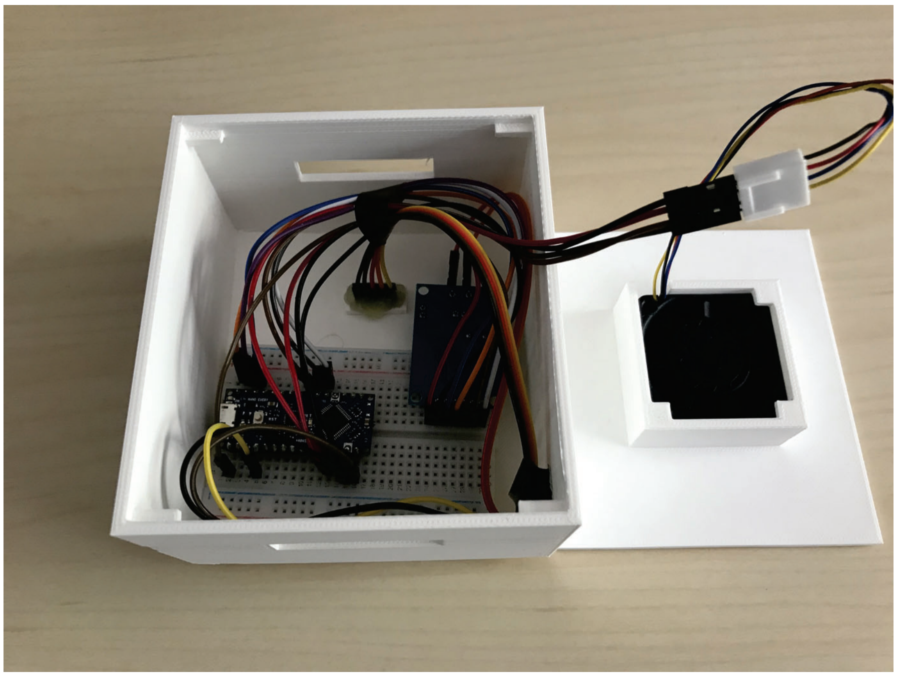


Figure 4.8: Inside of the temperature module

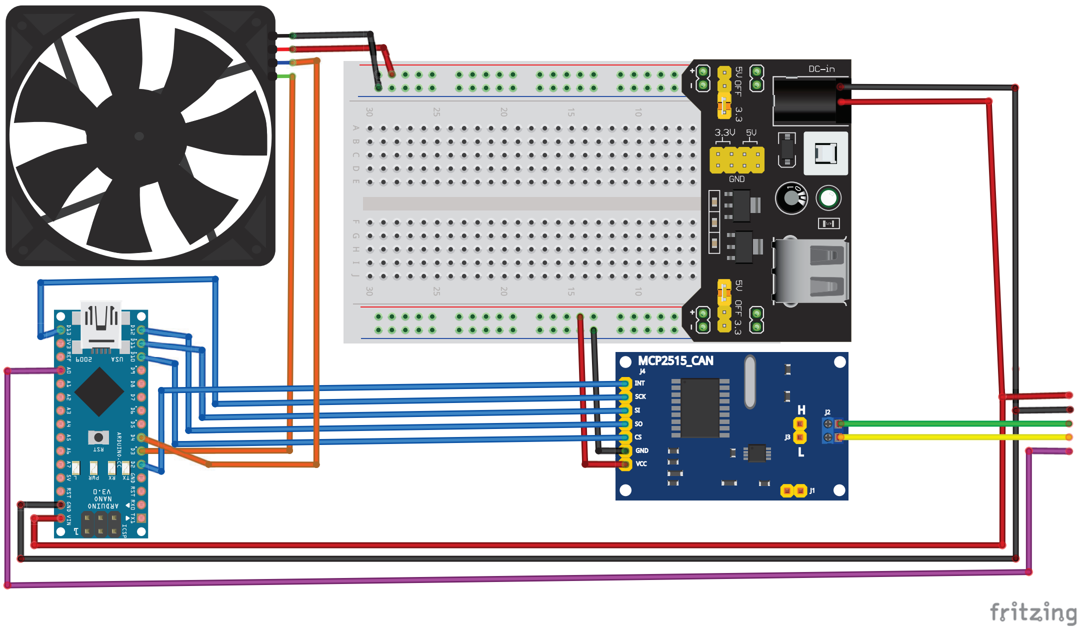


Figure 4.9: Wiring of an example module with a fan

4.2 Software

The programming language that Arduino uses is C++. Because many different projects have already been created, there is a large amount of libraries available to use for specific hardware parts. This support and compatibility is also one of the reasons why Arduino was chosen in the first place. An Arduino program consists of a setup and a loop. The setup is code that runs once when it is powered on, while the loop gets repeatedly called after the setup.

Before a module is able to send messages over the CAN bus, a unique address is needed. Remember that this address was sent to the module via an 'Address Arduino'. A one-wire protocol is needed to send the address. This is not as trivial as it initially sounds. An Arduino has digital I/O pins that send and receive data bit by bit. Sending and receiving a message bit by bit is not an option because there is no way of knowing where the message starts and ends. This could be solved by using a start and stop bit, but still, this proved difficult to implement. Because the connection only uses one wire, no clock is synced and it is possible to miss some bits that were sent, thus making this method unreliable. The Arduino also has analog I/O pins. The solution is to use PWM (Pulse Width Modulation). This is a wave that is HIGH for a certain amount of time and thereafter LOW for a set amount of time, as illustrated in Figure 4.10. It is then possible to measure the time between either two minimums or two maximums. The value that this returns is the address of the connector. If not used with care, this would also be an unreliable way of sending the address, because the measured time will not always be exactly the same, an error margin is used to make this method reliable.

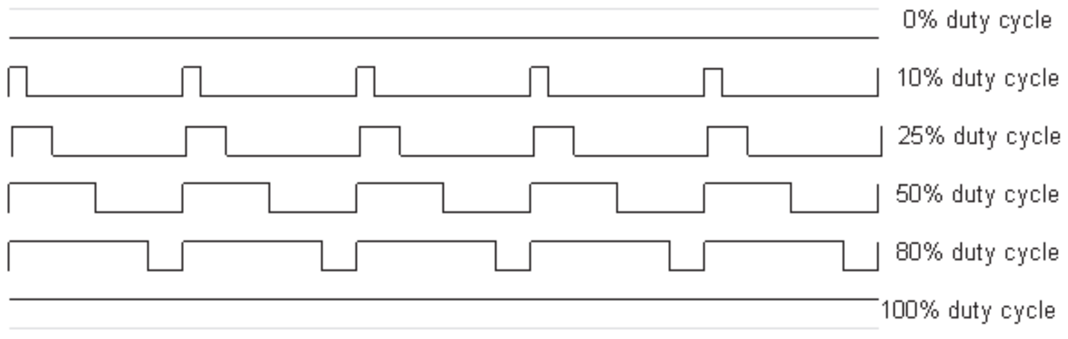


Figure 4.10: Pulse Width Modulation[9]

[9]https://docs.arduino.cc/tutorials/generic/secrets-of-arduino-pwm/

A CAN message contains 8 bytes of data as explained in Section 2.3.5. This refers to the maximum amount of data bytes; less bytes could also be used. The general structure of a CAN message is shown in Figure 4.11. The first byte of data will be used for the address of the sender. The second byte indicates what kind of message it is. This means “`n`” for a newly plugged-in module as illustrated in Figure 4.12, “`a`” for an additional physical variable of a newly plugged-in module as illustrated in Figure 4.13, “`h`” for the heartbeat of a module as illustrated in Figure 4.14 and “`s`” to set a specific physical variable a module to a new value as illustrated in Figure 4.15. The remaining values in Figure 4.11 are “`min`”, “`max`” and “`increment`” that describe the first physical variable, and “`#physvar`” for the total amount of physical variables. The “`min`” field refers to the minimum value, “`max`” to the maximum value and “`increment`” to how the value of a module is incremented. This is always on a linear scale. The “`#physvar`” field is in additional messages replaced by that number of the physical variable, as shown in Figure 4.13.

When a module is plugged into the platform, it first sends its name, min, max and increment values to the platform as shown in Figure 4.12. Additional physical variables are sent via the additional kind as illustrated in Figure 4.13. The module will then send a heartbeat to let the platform know that it is still plugged in, this is shown in Figure 4.14. If the value of a module needs to be changed, the platform will send a set message, which is shown in Figure 4.15.

addr	kind	name	min	max	interval	#phys var	

Figure 4.11: Newly plugged in module message general

1	n	t	0	100	10	1

Figure 4.12: Newly plugged in module message

1	a	t	0	100	10	2

Figure 4.13: Additional physical variable message

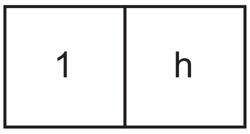


Figure 4.14: Heartbeat message

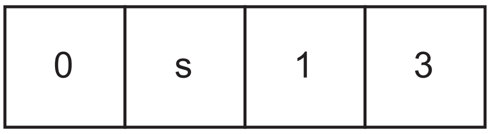


Figure 4.15: Set message

4.2.1 Platform

The platform has to keep track of all the currently plugged-in modules, not only in terms of time but also the encoding of that module. As previously mentioned, a module can have multiple physical variables. The platform will receive all of the required information and store this until the module is removed.

The source code of the platform consists of an array of ‘modules’, which are represented as structs. The address field is a fixed number that represents the connector where this module is located. The time value is updated to the current time when a heartbeat is received. By doing this it becomes possible to see if a module is removed. The potentially multiple physical variables are stored in an array (`module_sdl_array`). This is an array of `ModuleSDL objects`, which will be explained in section 4.2.3. Only the array of modules has a fixed size, all the other arrays are dynamically allocated because the size of, for example, the amount of physical variables of a module is not yet known at compile time. Lastly, this module struct also contains a value called `amount_phys_var`, which represents the number of physical variables that a module has.

```
class ModuleSDL {
  public:
    char name;
    byte min, max, increment;
};

typedef byte ADDR;

struct module {
  ADDR address;
  unsigned long time;
  byte amount_phys_var;
  ModuleSDL** module_sdl_array;
};

struct module* module_array[60];
```

When a module is plugged in, it sends a message to the platform that this is a newly plugged-in module, as was illustrated in Figure 4.12. To receive a CAN message, the CAN Arduino library[10] allows to read a packet byte by byte. The result of the receive procedure is a dynamically allocated array with the received values.

```
char* receive() {
    ...
        int i = 0;
        while (CAN.available()) {
          if (i < packetSize){
            result[i] = CAN.read();
            i++;
          }
        }
      }
    }
  }
  return result;
}
```

Every message that a module sends follows the same template, with the difference that it sends less or more bytes. The received value that is now stored in an

[10]https://github.com/sandeepmistry/arduino-CAN

array is then typecast to the `struct CANmsg` so that the information can easily be stored in the array of modules.

```
struct CANmsg {
  ADDR sender_address;
  char kind;
  ModuleSDL module_sdl;
  byte amount_phys_var;
};
```

To set or change the value of a module the serial input of the Arduino IDE needs to be used. First, the platform has to be plugged into a computer so that a serial message can be sent. The format of this 'set' message is illustrated in Figure 4.16.

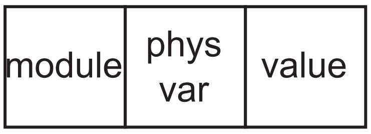


Figure 4.16: User input set message

4.2.2 Module

Receiving CAN messages for a module works exactly the same. The template of such a message is similar as shown in the `struct CANmsg` here:

```
struct CANmsg {
  ADDR sender_address;
  char kind;
  byte value;
  byte physical_variable_index;
};
```

Because the Arduino nano every is a micro-controller, it is not trivial to use multiple threads. Still, a module has to send a heartbeat every so often and constantly peek at the CAN bus to see if there is a message available. The solution was to implement the heartbeat procedure as follows. The Arduino starts counting from the moment that it is turned on. This can be used to keep track of time since a heartbeat was last sent. According to Arduino, the maximum amount of time it can count is 50 days[11], but this limit does not matter. A module could constantly

[11]`https://www.arduino.cc/reference/en/language/functions/time/millis/`

send this heartbeat message, but this would overload the CAN bus. Which is why a delay is added to the heartbeat.

In the heartbeat procedure, it becomes clear how a CAN message is sent. Such a message always begins with the address of the receiver and thereafter the address of the sender, in this case, a message is sent from the module to the platform. Lastly, the character "h" is added to the message. Calling the `CAN.endPacket()` procedure will send the message.

```
unsigned long heartbeat_time = 0;

void heartbeat(){
  char heartbeat_char = 'h';
  if(millis() > heartbeat_time || heartbeat_time == 0){
    CAN.beginPacket(platform_address);
    CAN.write(module_address);
    CAN.write(heartbeat_char);
    CAN.endPacket();
    heartbeat_time = millis()+500;
  }
}
```

4.2.3 Encoded Physical Variables in Modules

It should be the case that the platform knows how to set or increment specific physical variables. That is why we need a self-describing language to encode the attributes and properties of a physical variable. Remember that when a module is plugged in, it sends its 'type' to the platform via the CAN bus. After this step, the module waits for the platform to set it to certain values. A first idea could be to send the entire encoding of the physical variable of the module. A message on the CAN bus is limited to 8 bytes of data. That is why an alternative method was created. The self-describing language is represented as a C++ class. This is an example of a temperature module that has one fan encoded:

```
class ModuleSDL {
  public:
    char name;
    byte min, max, increment;
};

class Temperature : public ModuleSDL {
  public:
    char name = 't';
    byte min = 0;
    byte max = 100;
    byte increment = 10;
};

Temperature module;
```

The fields of this class are `min`, `max` and `increment`. When a new module is created, these values need to be specified by its creator. Upon plugging a module into the platform, it will send a special message containing these values. The platform will create a new object containing exactly these values. It can keep track of multiple such objects per module, meaning that multiple physical variables per module are supported. Every so often the module will send a small message called the heartbeat to indicate that it is still plugged in. When the platform does not receive such a message after a certain amount of time the module will be regarded as plugged out. At that moment the platform can throw away any information, like the physical variable objects, that is kept about that module.

Now that all of the components have been explained in terms of hardware and software, the result is shown in Figure 4.17. There are 2 modules on top of the platform, one temperature and one tactile module. The temperature module was previously shown. The tactile module has a vibration motor on top. The intensity of this vibration motor can be changed, just like the speed of the fan of the temperature module.

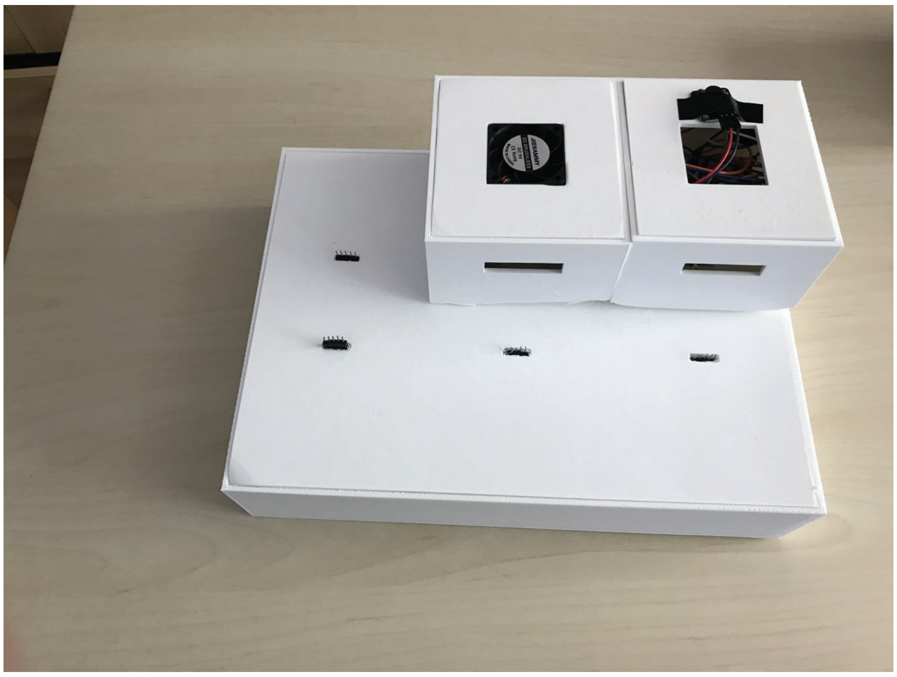


Figure 4.17: Platform with 2 modules plugged in

4.3 Creating a New Module

As said before, anyone should be able to make a new module by following a set of rules. This will now be discussed in detail. Any size for the platform and module can be chosen but let us assume that someone wants to create a new module for this platform. The exact measurements of a module are 100 × 110 × 63mm, although the height of 63mm could be changed. The connector hole at the bottom must be 5 × 14mm and must be placed exactly as shown in Figure 4.18. Next, the `ModuleSDL` class must be implemented by creating a subclass and filling in all the fields.

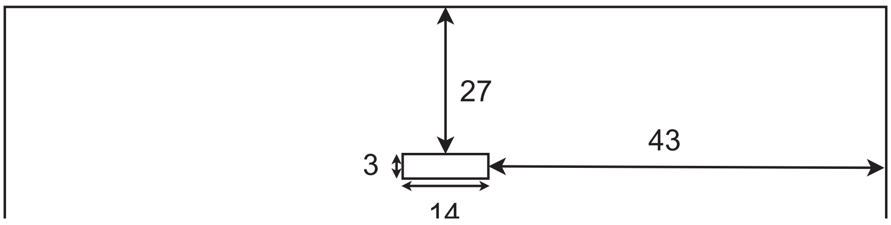


Figure 4.18: Dimensions in millimetres of the connector of a module

```
class ModuleSDL {
  public:
    char name;
    byte min, max, interval;
};
```

The two following procedures must be implemented for the setup and changing of a physical variable. The code below shows the implementation for the temperature module that encodes a fan.

Finally, the module must also contain all of the 'standard' hardware of a module, for example, an Arduino. The exact wiring of a module is shown in Figure 4.9.

```
void setup_physical_variables(){
  fan.begin();
  fan.setDutyCycle(0);
}

void set_physical_variable(int value){
  fan.setDutyCycle(value);
}
```

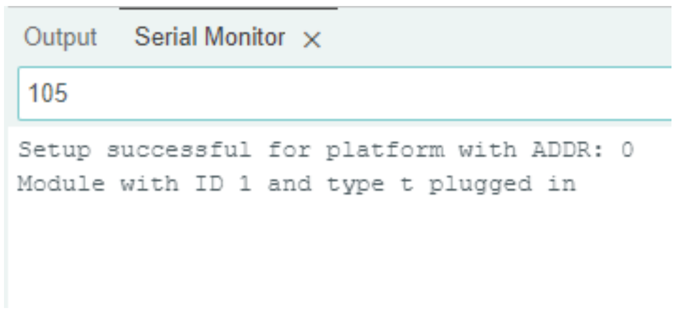
4.4 Technical Evaluation

4.4.1 Experiment 1: Set the Value of a Module

When the platform is plugged into a computer via USB, a user is able to communicate with it. Via the Arduino IDE, it is possible to send a message from a computer to the platform. This message contains the address of the module and value(s) of the physical variable(s). The platform will send the value(s), via the CAN bus, to the specified module. The module itself is the only one that knows how to handle this value(s). The user can set the fan speed of a module with a fan to a certain value. The module will change the fan speed to this value.

Let us start with a simple technical evaluation. Plug in one module and change the value of this module a couple of times. In Figure 4.19 the output of the Arduino IDE is shown. In Figure 4.19a, the platform setup was successful and a module has been plugged into the connector with address 1. Every module has a 1-byte type, which is represented here with the 't' character. In the serial input box, the number `105` has been typed. This input will be sent to the platform when the user presses enter. Remember that the `1` represents the address of the module, `0` represents the physical variable (which is numbered beginning from 0) and `5` represents the number of increments that the physical variable has to change. In Figure 4.19b we see the result of the input message `105`. This specific module has an `increment` of 10 so $5 \times 10 = 50$. The output of inputting `100` is shown in Figure 4.19c.

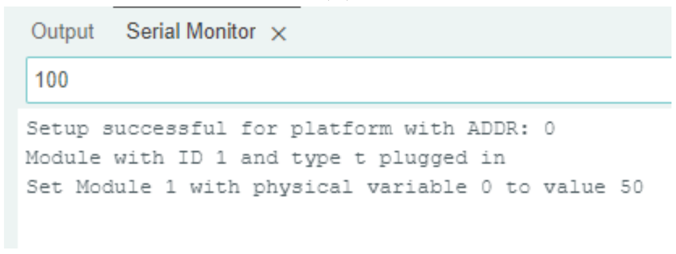
When plugging the module into the platform it was noticed how hard it is to plug in a module. This is because the connectors are fragile and not perfectly aligned.



```
105
```

```
Setup successful for platform with ADDR: 0
Module with ID 1 and type t plugged in
```

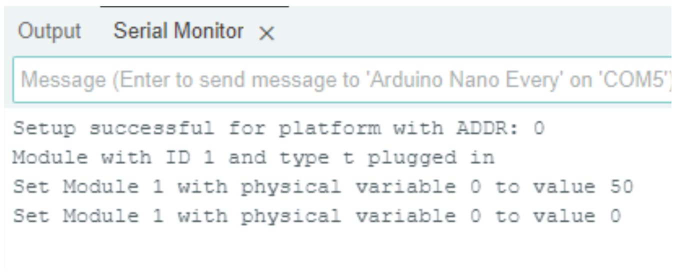
(a)



```
100
```

```
Setup successful for platform with ADDR: 0
Module with ID 1 and type t plugged in
Set Module 1 with physical variable 0 to value 50
```

(b)


Output
Serial Monitor ×
Message (Enter to send message to 'Arduino Nano Every' on 'COM5')

```
Setup successful for platform with ADDR: 0
Module with ID 1 and type t plugged in
Set Module 1 with physical variable 0 to value 50
Set Module 1 with physical variable 0 to value 0
```

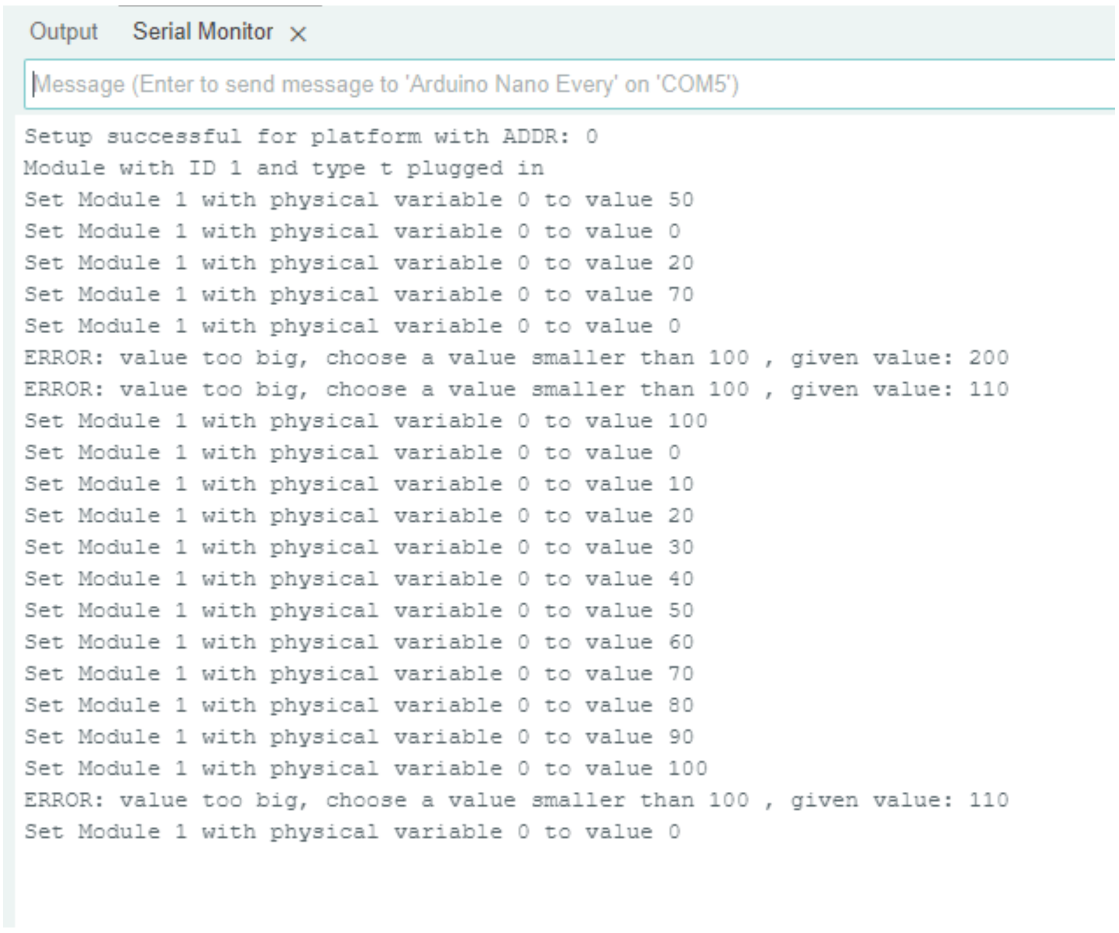
(c)

Figure 4.19: Arduino IDE output of setting the value of a module

4.4.2 Experiment 2: Set an Invalid Value of a Module

The setup is exactly the same as experiment 1, with the difference being that the received value is outside the range of values of that physical variable. Every physical variable has a minimum and maximum range of values. Receiving a value outside this range will result in an error.

The experiment is an extension of experiment 1. As shown in Figure 4.20, the initial setup is again the same. The first physical variable of a module is set to a valid value multiple times. Then a value is used that is outside of the range of that physical variable. An error message is printed saying that a value was given that is too big and shows the maximum range and the given value.

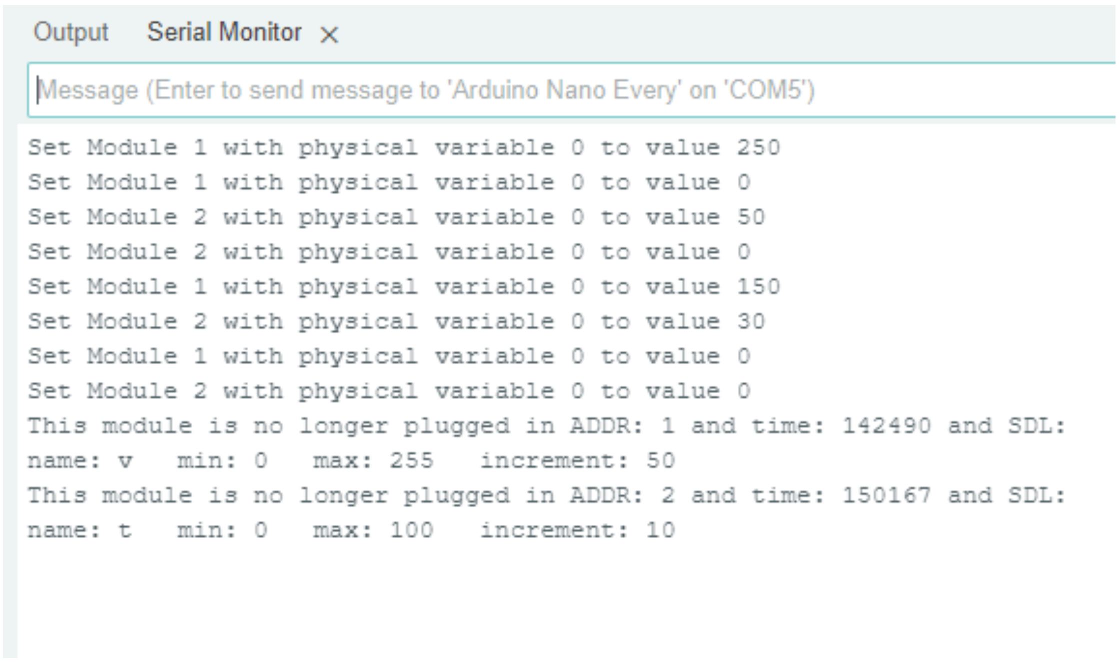

Output
Serial Monitor
Message (Enter to send message to 'Arduino Nano Every' on 'COM5')

```
Setup successful for platform with ADDR: 0
Module with ID 1 and type t plugged in
Set Module 1 with physical variable 0 to value 50
Set Module 1 with physical variable 0 to value 0
Set Module 1 with physical variable 0 to value 20
Set Module 1 with physical variable 0 to value 70
Set Module 1 with physical variable 0 to value 0
ERROR: value too big, choose a value smaller than 100 , given value: 200
ERROR: value too big, choose a value smaller than 100 , given value: 110
Set Module 1 with physical variable 0 to value 100
Set Module 1 with physical variable 0 to value 0
Set Module 1 with physical variable 0 to value 10
Set Module 1 with physical variable 0 to value 20
Set Module 1 with physical variable 0 to value 30
Set Module 1 with physical variable 0 to value 40
Set Module 1 with physical variable 0 to value 50
Set Module 1 with physical variable 0 to value 60
Set Module 1 with physical variable 0 to value 70
Set Module 1 with physical variable 0 to value 80
Set Module 1 with physical variable 0 to value 90
Set Module 1 with physical variable 0 to value 100
ERROR: value too big, choose a value smaller than 100 , given value: 110
Set Module 1 with physical variable 0 to value 0
```

Figure 4.20: Arduino IDE output of setting a value outside of the range of a physical variable of a module

4.4.3 Experiment 3: Setting 2 Modules at the Same Time

In this experiment, 2 modules are set to some value at the same time. This is illustrated in Figure 4.21. At the end, it is also shown when a module is removed. The information that the module sent when plugged in is displayed to show that it was stored correctly.


Output
Serial Monitor ×
Message (Enter to send message to 'Arduino Nano Every' on 'COM5')

```
Set Module 1 with physical variable 0 to value 250
Set Module 1 with physical variable 0 to value 0
Set Module 2 with physical variable 0 to value 50
Set Module 2 with physical variable 0 to value 0
Set Module 1 with physical variable 0 to value 150
Set Module 2 with physical variable 0 to value 30
Set Module 1 with physical variable 0 to value 0
Set Module 2 with physical variable 0 to value 0
This module is no longer plugged in ADDR: 1 and time: 142490 and SDL:
name: v   min: 0   max: 255   increment: 50
This module is no longer plugged in ADDR: 2 and time: 150167 and SDL:
name: t   min: 0   max: 100   increment: 10
```

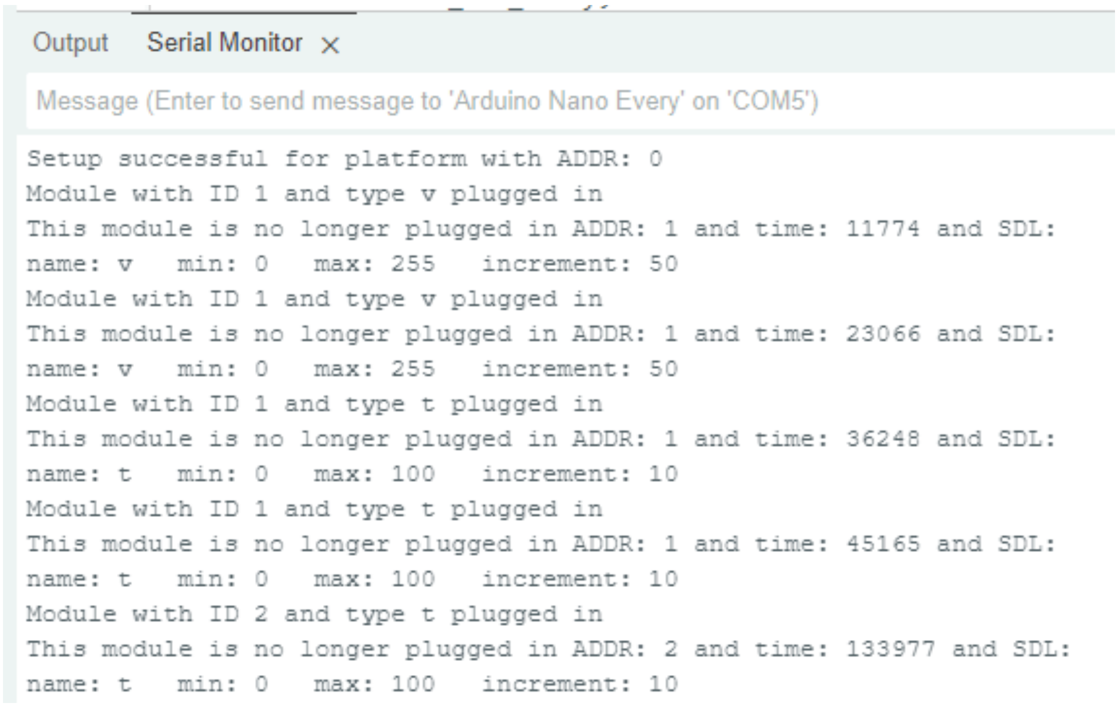
Figure 4.21: Arduino IDE output of setting values of physical variables of 2 modules

In this experiment, it was noticed that plugging in a module adjacent to another module was difficult. The reason behind this is that the platform was designed to fit exactly six modules. Because of this tight margin, it was not easy to plug in a module when a connector did not perfectly align.

4.4.4 Experiment 4: Hot-Plugging Modules

Modularity was defined as a key aspect of this platform. To illustrate that this works as intended, modules are plugged in and out of the platform. The platform detects whenever a module is plugged out that it no longer sends the heartbeat and recognises newly plugged-in modules. As mentioned before, plugging a module into the platform was more difficult than envisioned.

In Figure 4.22, the result of this experiment are shown. Modules were plugged in and out of 2 connectors. Again, most connectors were difficult to plug into, but when finally plugged in everything worked as described.


Output
Serial Monitor
Message (Enter to send message to 'Arduino Nano Every' on 'COM5')

```
Setup successful for platform with ADDR: 0
Module with ID 1 and type v plugged in
This module is no longer plugged in ADDR: 1 and time: 11774 and SDL:
name: v   min: 0   max: 255   increment: 50
Module with ID 1 and type v plugged in
This module is no longer plugged in ADDR: 1 and time: 23066 and SDL:
name: v   min: 0   max: 255   increment: 50
Module with ID 1 and type t plugged in
This module is no longer plugged in ADDR: 1 and time: 36248 and SDL:
name: t   min: 0   max: 100   increment: 10
Module with ID 1 and type t plugged in
This module is no longer plugged in ADDR: 1 and time: 45165 and SDL:
name: t   min: 0   max: 100   increment: 10
Module with ID 2 and type t plugged in
This module is no longer plugged in ADDR: 2 and time: 133977 and SDL:
name: t   min: 0   max: 100   increment: 10
```

Figure 4.22: Arduino IDE output of hot-plugging modules

Chapter 5

Discussion and Future Work

5.1 Discussion

Remember that these are the research questions that this thesis aims to answer:

RQ1 How can plug-and-play capabilities be implemented in a platform that serves to create a dynamic data physicalisation?

RQ2 How can modules be made for a dynamic data physicalisation platform by following a set of rules and a self-describing language?

The first research question is partially answered. The thesis begins with a summary of existing projects and papers about modularity and dynamic data physicalisations. Their relevancy was explained and arguments were given as to why certain features were interesting. A solution for such a platform was created and discussed. The platform with modules serves as a stand-alone toolkit that can be used by any researcher or developer. The described solution can easily be replicated because of the availability and cost of the hardware components. The platform can be used to quickly and efficiently prototype to create a dynamic data physicalisation. A researcher or developer does not have to start from scratch but can instead use the platform described in this thesis. As discussed in the technical evaluation, the connectors of the platform were difficult to plug a module into. The whole idea and the connectors themselves work, but because standard fragile wires were used, this came at the cost of easily plugging in. The difficulty of using this connector limits the real plug-and-play capabilities. Furthermore, the actual dynamic data physicalisations that could be made with this platform are limited because of the modules, which will now be discussed when talking about the second research question.

The second research question also remains partially unanswered. Researchers and developers can also make their own modules by following the described rules.

This new module should work as expected with the platform. It can encode multiple physical variables. However, the modules are limited in what and how they can represent physical variables. The modules now use an encoding of physical variables, which is not a self-describing language.

5.2 Contributions

1. The first contribution is related to the research on existing approaches that was conducted. Existing modular plug-and-play systems were reviewed, in combination with interesting and relevant dynamic data physicalisations. Furthermore, the most relevant and well-known communication protocols were discussed and explained why they could or could not be used.

2. The second contribution are the different solution architectures for the platform and module. Especially the final two solutions. Coming to these solutions meant taking various decisions along the way. Any direction could have been followed, but, for example, the CAN bus was used as the main communication bus and this was heavily discussed.

3. Lastly, the third contribution is the whole implementation. Creating a theoretical solution is one thing, but actually implementing the envisioned architecture has multiple challenges. This includes the design of the platform and module, the hardware components and source code. All of the components were chosen so that anyone could recreate this thesis without being overly complex or expensive. The source code for the module and platform is made so that when someone makes a new module, they only have to fill in specific values about the chosen physical variables, and it should just work. Any amount of physical variables are supported.

5.3 Future Work

With this modular platform, it now becomes possible for developers and researchers to quickly prototype dynamic data physicalisations. Coming to this solution, however, meant taking various choices along the way. Some of which could be improved upon. These improvements include hardware, software and validation of the project as a whole.

5.3.1 Wiring and 3D Printing

For starters the wiring. The 3D printed cases were made so that they could fit a small breadboard so that there was no soldering involved. While this benefits the group of people who cannot solder, the better solution would be to solder everything so that the platform base and the encasing of the modules can be much smaller.

5.3.2 Validation and Use Case

An extensive evaluation and use case is necessary to validate the effectiveness of this solution. A comprehensive evaluation using developers and researchers and letting them use the platform. This way we can validate if this proof of concept does indeed serve as a quick prototyping tool. This would test the system as a whole. Another needed validation should be done to test how easy and effective it is to make a contribution. By letting developers and researchers create their own module of choice and validating that this module works as intended.

5.3.3 Module Examples

The modules were made so that anyone could create their own by just following the self-describing language and using the same hardware components with their own physical variables. While some examples of modules were created this is not a big enough pool to support starting developers and researchers that would just want to use simple existing modules. Having a big set of pre-designed modules allows users to have a good starting point.

5.3.4 User Interface

The most important improvement has to do with user interaction with the platform. Now there is no screen for a user to interact with. This means that users have to set the values of modules by using the Arduino interface. To really complete this thesis, creating a user interface on a touch screen where users could select the values of specific modules would by far be the most needed improvement. Such a touch screen should be connected to the platform and just inform it what module has to be set to what value. However, the platform could also send information about modules to this touch screen. Because the platform has the knowledge of when a module is plugged in and also what type of module this is, this information could all be shown on the screen. Furthermore, the connectors of the platform have addresses, so the platform knows where a module is plugged in. This is again useful information that could be presented on the screen.

5.3.5 Self-describing Language

An important feature of the modules was the self-describing language. In the current implementation, this was replaced by a simpler idea: this encoding contains some values about the physical variable. However, complex physical variables that require not only increments on a linear scale are not supported. Changing the way in which a module sends its encoding so that it becomes applicable to a possible endless amount of supported physical variables would be a major improvement. The platform has to be able to handle all of this.

5.3.6 Platform-module connector

As already mentioned in the technical evaluation and discussion, the connector is difficult to plug into. This means that a new type of connector has to be researched and implemented. Various options should be looked at and explain why the chosen solution is the best. This new connector can then be used in the top of the platform and at the bottom of a module. The modules now have a female connector and the platform has the male connector. This idea should best be kept so that a module can still lay flat on a surface.

Chapter 6

Conclusion

The modular platform that was presented in this thesis can be used to create dynamic data physicalisations. By first looking at plug-and-play systems that already exist, an idea was gathered on how to approach this thesis. Thereafter dynamic data physicalisations were reviewed to already see what the outcome of a solution with this platform should be. It was an important part to see what is already out there in both fields. Doing this provided an immediate idea of what was needed to build a modular platform that could create dynamic data physicalisations in such a way that it promotes prototyping for both developers and researchers.

Dynamic data physicalisations provide many learning and cognitive benefits. It goes beyond only the visual modality. This means that it is a real-world object that we can interact with and play with.

Modularity and plug-and-play were two of the most important requirements. This has been achieved by making the modules so that they can be plugged in and out at any moment and that the platform can handle this hot plugging. Anyone with the required knowledge should be able to make a contribution by making a module with the physical variable(s) of their choice using the required module hardware and the self-describing language. This newly created module should work with the platform. If other configurations of the platform or modules are required, one can create a new case and add the required hardware.

After researching existing approaches, a solution was created that followed all of the mentioned design choices, for example, plug-and-play and consisting of well-known hardware parts like Arduino. This theoretical solution with the mentioned hardware parts was implemented. A 3D printer was used to create an encasing for the platform and modules so that users would not be exposed to any internal parts. To verify this implementation a technical evaluation was performed. This did indeed verify the most important features, although it has some limitations. These limitations include the lack of a user interface, the simple encoding of a module instead of a self-describing language and the difficulty of using the connector between a module and the platform. Many modules with multiple physical variables could be created and should work with the platform. The benefits are that it is possible to recreate the platform and implement new modules. The new modules could implement any physical variable and can communicate with and be set by the platform. This platform sets a new baseline for rapid development and prototyping for researchers and developers so that they can create their envisioned dynamic data physicalisation with this platform instead of starting from scratch.

Bibliography

[1] Jan Axelson. *Serial Port Complete: COM Ports, USB Virtual COM Ports, and Ports for Embedded Systems*. Lakeview Research, December 2007.

[2] Paulo Blikstein. Computationally Enhanced Toolkits for Children: Historical Review and a Framework for Future Design. *Foundations and Trends in Human-Computer Interaction*, 9(1):1–68, December 2015.

[3] William Bolton. Chapter 4 - I/O Processing. In *Programmable Logic Controllers*, pages 75–109. Boston, USA, July 2009.

[4] Hans Brombacher, Rosa Van Koningsbruggen, Steven Vos, and Steven Houben. SensorBricks: A Collaborative Tangible Sensor Toolkit to Support the Development of Data Literacy. In *Proceedings of the Eighteenth International Conference on Tangible, Embedded, and Embodied Interaction*, pages 1–17, New York, USA, February 2024.

[5] Hanxing Chen and Jun Tian. Research on the Controller Area Network. In *2009 International Conference on Networking and Digital Society*, pages 251–254, Guiyang, Guizhou, China, June 2009.

[6] Steve Corrigan. Introduction to the Controller Area Network (CAN). pages 1–17, August 2002.

[7] Sida Dai, Brygg Ullmer, and Winifred Elysse Newman. MorphMatrix: A Toolkit Facilitating Shape-Changing Interface Design. In *Proceedings of the Eighteenth International Conference on Tangible, Embedded, and Embodied Interaction*, pages 1–12, New York, USA, February 2024.

[8] Piyu Dhaker. Introduction to SPI Interface. *Analog Dialogue*, 52(3):49–53, September 2018.

[9] Louis E. Frenzel. *Handbook of Serial Communications Interfaces: A Comprehensive Compendium of Serial Digital Input/Output (I/O) Standards*. Newnes, 2016.

[10] Shaimaa Mudhafar Hashim and Israa Badr Al-Mashhadani. Adaptation of Powerline Communications-based Smart Metering Deployments With IoT Cloud Platform. *Indonesian Journal of Electrical Engineering and Computer Science*, 29(2):825–837, February 2023.

[11] Trevor Hogan, Uta Hinrichs, and Eva Hornecker. The Visual and Beyond: Characterizing Experiences with Auditory, Haptic and Visual Data Representations. In *Proceedings of the 2017 Conference on Designing Interactive Systems*, pages 797–809, New York, USA, June 2017.

[12] Hiroo Iwata, Hiroaki Yano, Fumitaka Nakaizumi, and Ryo Kawamura. Project FEELEX: Adding Haptic Surface to Graphics. In *Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques*, pages 469–476, New York, USA, August 2001.

[13] Yvonne Jansen, Pierre Dragicevic, Petra Isenberg, Jason Alexander, Abhijit Karnik, Johan Kildal, Sriram Subramanian, and Kasper Hornbæk. Opportunities and Challenges for Data Physicalization. In *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems*, pages 3227–3236, New York, USA, April 2015.

[14] Mannu Lambrichts, Raf Ramakers, Steve Hodges, Sven Coppers, and James Devine. A Survey and Taxonomy of Electronics Toolkits for Interactive and Ubiquitous Device Prototyping. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies*, 5(2):1–24, June 2021.

[15] Mannu Lambrichts, Jose Maria Tijerina, and Raf Ramakers. SoftMod: A Soft Modular Plug-and-Play Kit for Prototyping Electronic Systems. In *Proceedings of the Fourteenth International Conference on Tangible, Embedded, and Embodied Interaction*, pages 287–298, New York, USA, February 2020.

[16] Daniel Leithinger and Hiroshi Ishii. Relief: A Scalable Actuated Shape Display. In *Proceedings of the Fourth International Conference on Tangible, Embedded, and Embodied Interaction*, pages 221–222, New York, USA, January 2010.

[17] Daphne Menheere, Evianne van Hartingsveldt, Mads Birkebæk, Steven Vos, and Carine Lallemand. Laina: Dynamic Data Physicalization for Slow Exercising Feedback. In *Designing Interactive Systems Conference 2021*, pages 1015–1030, New York, USA, June 2021.

[18] Stephen St. Michael. Introduction to CAN (Controller Area Network). February 2019.

[19] Eric Peña and Mary Grace Legaspi. Uart: A Hardware Communication Protocol Understanding Universal Asynchronous Receiver/Transmitter. *Analog Dialogue*, 54(4):1–5, December 2020.

[20] Eric Peña and Mary Grace Legaspi. I2C Communication Protocol: Understanding I2C Primer, PMBus, and SMBus. *Analog Dialogue*, 55(4):1–9, November 2021.

[21] David Reinsel, John Gantz, and John Rydning. The Digitization of the World from Edge to Core. *Framingham: International Data Corporation*, pages 1–28, November 2018.

[22] Joel Sadler, Kevin Durfee, Lauren Shluzas, and Paulo Blikstein. Bloctopus: A Novice Modular Sensor System for Playful Prototyping. In *Proceedings of the Ninth International Conference on Tangible, Embedded, and Embodied Interaction*, pages 347–354, New York, USA, January 2015.

[23] Nishant Sagar. *Powerline Communications Systems: Overview and Analysis.* PhD thesis, Rutgers University-Graduate School-New Brunswick, 2011.

[24] Charles E Spurgeon. *Ethernet: The Definitive Guide.* "O'Reilly Media", February 2000.

[25] Faisal Taher, John Hardy, Abhijit Karnik, Christian Weichel, Yvonne Jansen, Kasper Hornbæk, and Jason Alexander. Exploring Interactions with Physically Dynamic Bar Charts. In *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems*, pages 3237–3246, New York, USA, April 2015.

[26] Faisal Taher, Yvonne Jansen, Jonathan Woodruff, John Hardy, Kasper Hornbæk, and Jason Alexander. Investigating the Use of a Dynamic Physical Bar Chart for Data Exploration and Presentation. *IEEE Transactions on Visualization and Computer Graphics*, 23(1):451–460, August 2016.

[27] Nicolas Villar, Daniel Cletheroe, Greg Saul, Christian Holz, Tim Regan, Oscar Salandin, Misha Sra, Hui-Shyong Yeo, William Field, and Haiyan Zhang. Project Zanzibar: A Portable and Flexible Tangible Interaction Platform. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*, pages 1–13, New York, USA, April 2018.

[28] Nicolas Villar, James Scott, Steve Hodges, Kerry Hammil, and Colin Miller. .NET Gadgeteer: A Platform for Custom Devices. In *Pervasive Computing*, pages 216–233, Berlin, Germany, June 2012.

[29] Wilfried Voss. *A Comprehensible Guide to Controller Area Network*. Copperhill Media, 2008.

[30] Lieping Zhang, Huihao Peng, Liu Tang, Xu Cao, and Xiao-Guang Yue. A Control System of Multiple Stepmotors Based on STM32 and CAN Bus. In *Proceedings of the 2020 2nd International Conference on Big Data and Artificial Intelligence*, pages 624–629, New York, USA, January 2021.