UNIVERSITY OF TARTU
Faculty of Science and Technology
Institute of Computer Science
Computer Science Curriculum

Marten Vainult

ESP32 Based Mesh Network Solution for Managing Practical Lessons

Bachelor's thesis (9 ECTS)

Supervisor: Kaido Reivelt, PhD

Tartu 2024

ESP32 Based Mesh Network Solution for Managing Practical Lessons

Abstract: Basic and secondary education natural science lessons can be enhanced by conducting experiments or using an audience response system for polling. The issue with conducting practical experiments with electronic devices is the high cost and the lack of control for managing it all at once if provided to all the students. The polling sessions are usually done with the help of smartphones, but they're proven to cause distractions. The aim of this thesis was to design and build a prototype system which allows to conduct experiments while having a connection to the teacher interface. The result was a monitoring interface with an ESP32 based mesh network solution. The mesh network was built on ESP-Mesh-Lite. The monitoring interface was developed in Vue.js and connected to the mesh network with a USB cable. The monitoring interface displayed the information of all devices on the mesh network and was able to broadcast messages to the network. The resulting prototype filled the requirements by being easy to use and having the ability to exchange data with all of the devices.

Keywords: Mesh network, ESP32, SoC, Serial connection, Monitoring, LAN

CERCS: T180 Telecommunication engineering; P170 Computer science, numerical analysis, systems, control

ESP32 mikrokontrolleril põhineva lahenduse loomine praktiliste tundide läbiviimiseks

Abstract: Põhi- ja keskkooli loodusainete tunde saab võimendada praktiliste katsete või küsitlustega. Praktiliste katsete üldiseks murekohaks on elektrooniliste katsevahendite kõrge hind ja nende haldamine kui on välja jagatud terve klassi peale. Interaktiivsete küsitluste murekohaks on tähelepanu hajuvus kui kasutatakse personaalseid nutiseadmeid. Lõputöö eesmärgiks oli projekteerida ja valmis ehitada prototüüplahendus praktiliste tundide läbiviimiseks, mida oleks lihtne üles seada ja on õpetaja poolt ühest kohast hallatav. Töö tulemusena valmis veebirakendus, mis ühildub USB kaabli abil ESP32 mikrokontrollerite poolt loodud silmvõrguga (*mesh network*). Veebirakendusel on vaade kõikide seadmete info ja küsitlussessioonide tulemuste kohta. Töö tulemus täitis määratud tingimusi nii kasutusmugavuse kui ka võimekuse poolest.

Keywords: Silmvõrk, ESP32, süsteemikiip, *serial* ühendus, haldus, kohtvõrk

CERCS: T180 Telekommunikatsioonitehnoloogia; P170 Arvutiteadus, arvutusmeetodid, süsteemid, juhtimine (automaatjuhtimisteooria)

Contents

1. Introduction 5
1.1. Thesis structure 6
2. Concepts, terms and abbreviations 7
3. Design requirements and analog solutions 9
3.1. Sample experiment 9
3.2. General requirements 9
3.3. Monitoring interface requirements 9
3.4. Software requirements 9
3.5. Hardware requirements 9
4. Prototype solution design 10
4.1. Hardware options 10
Microcontroller 10
Ambient light sensor 11
Temperature sensor 11
4.2. Data exchange options 11
Bluetooth 11
Wi-Fi 11
Wireless mesh networks 12
No-router example 14
4.3. Software 15
Prototype web application 16
5. Developing the prototype system 17
5.1. Analog sensors 17
Photogate experiment 18
Continous mode ADC 19
5.2. External devices setup 20
RGB LED setup 20
Button setup 20
Temperature shield 21
5.3. Esp-Mesh-Lite setup 22
Event handling 23
UART connection setup 24
5.4. Monitoring interface 25
Opening a connection with USB 25
Connection with devices 26
Audience response system 27
6. Conclusions 29
6.1. Development experience 29
7. Acknowledgments 30
8. References 31

9. Appendix ... 35
I. Access to the code ... 35
II. Licence ... 36

1. Introduction

Real life lessons teach us more than theoretical ones. It is important to know and understand core concepts, but the acquired knowledge remains with us longer if we experience them to actually be true in the physical world. In the school system of Estonia, the natural science experimentation happens mainly on the teacher’s table while the students are watching. This is better than no experimentation at all, but there should be a way for students to conduct the experiments themselves for better experience and learning. The gap is mainly derived from the high cost of experiment equipment, but it’s also amplified by the difficulty of managing and assisting the whole classroom that conducts the experiments. This issue can be alleviated by a low-cost and easy to manage system for schools that offers teachers and students easy to conduct experiments and learn the core concepts of natural sciences.

Modern classrooms are also suffering from the overuse of smartphones as the presence of an external device decreases the attentiveness of the student and also its peers. Even in situations, where the students are required to use the smartphone as a polling device, it is reported to cause distractions by the students dwelling into other applications in the polling phase. While the polling system enhances students' understanding of core concepts, the teacher also gets a better understanding of whether there is a need to explain covered topics more thoroughly. The need for a smartphone can be diminished by a physical audience response system. Although there are no-cost and easy solutions for it, the main issue is that it should be anonymous for the students.

The problems could be solved by creating an easy to use system that enables students to conduct experiments while also offering the teacher an interface to manage the classroom. The main requirements for the system was the ease of use by having no external devices, software or configuring required while achieving low latency connection and being small in size. The main expectation for this work was to confirm that it’s possible to create an easy to manage interconnected system with smart devices that empower practical lessons.

1.1. Thesis structure

The following section illustrates the format in which the work will be presented. The topics of each chapter are summarized as follows:

- Chapter 1: Introduction and background information of the thesis
- Chapter 2: Explanation of concepts, terms and abbreviations used.
- Chapter 3: Setting the requirements on the prototype solution.
- Chapter 4: Designing the prototype system based on hardware and software available.
- Chapter 5: Testing the user device processing capability with a sample experiment and developing the prototype solution.
- Chapter 6: Summary on the end result and sharing the experience received in the design and development phase.
- Chapter 7: Acknowledgments of contributors

The source code for the prototype solutions can be found in the Appendix section.

2. Concepts, terms and abbreviations

ADC	Analog-to-Digital converter
API	Application programming interface
Broadcasting	Distributing information across the whole network.
Child node	If node X is connected to node Y and X travels more layers to the root node than Y, then X is said to be a child of Y
Downstream Connection	Connection from a parent node to one of its child nodes
GPIO port	General-Purpose Input/Output port - handles both incoming and outgoing digital signal
Heap memory	Dynamic memory allocated by function calls
Host user	The user that is managing the system. The teacher in our system
IDE	Integrated Development Environment
IoT	Internet of Things
JSON	Javasvript Object Notation - lightweight data-interchange format
LAN	Local Area Network – A collection of devices connected together
Latency	Time delay between the cause and the effect of some physical change in the system being observed
Leaf node	Node that has a parent node and no child nodes
LED	Light-emitting diode
Mac address	12-digit hexadecimal number assigned to each device connected to the network
Mesh network	Wireless mesh network (WMN) is a communications network made up of radio nodes organized in a mesh topology.
MCU	Microcontroller Unit
ms	Millisecond. 1/1000 of a second.
Node	Any device that is part of or can be part of the mesh network.
OTA	Over-the-air update
RGB	Red-Green-Blue color model.
Root node	Nodes at the top of the network
SDK	Software development kit

Singlecasting	Sending information to only one target
Serial (communication)	Serial communication is the process of sending data one bit at a time, sequentially, over a communication channel or computer bus
UART	A Universal Asynchronous Receiver-Transmitter is a peripheral device for asynchronous serial communication in which the data format and transmission speeds are configurable.
Upstream connection	Connection from a node to its parent node
VSC	Visual Studio Code, source code editor developed by Microsoft

3. Design requirements and analog solutions

The following section gives an overview on the requirements set for developing the prototype system. As the main users of the system are people without a technical background, there is a strong emphasis on the ease of use and requiring no pre-configuration for setup.

3.1. Sample experiment

We are going to use measuring the speed of objects via two photogates as a sample experiment to test the processing power of our device. Measuring the speed of objects is usually done with photogates that detect the passing of an object and a processing unit that calculates the time that took the object to get from point A to point B as illustrated.

3.2. General requirements

1. The devices can be set up and used without having to configure the connections.
2. Two-way data transfer, where the host can send and receive data with user devices.
3. The host user uses a browser on a computer for opening the interface and must not need to adjust the internet connection.
4. System is able to handle at least 24 devices.

3.3. Monitoring interface requirements

1. Overview consisting of all the devices connected.
2. State information for each device in the network.
3. Button for broadcasting info to the devices.
4. Audience response system functionality.
5. Low latency connection with the student devices.

3.4. Software requirements

1. Open-source frameworks in either the latest or stable release.
2. Support for OTA (over-the-air) updates

3.5. Hardware requirements

1. Battery powered
2. Wireless connection with monitoring interface
3. Expansion ports for external equipment
4. Processing capability of conducting the sample experiment

The decision to build the prototype solution came from not finding analog systems on the market that fit our requirements and also from the need of having a highly customizable solution.

4. Prototype solution design

The design of the prototype solution starts with a comparison on different data exchanging options based on our requirements. The software and hardware solution is then chosen based on the data exchange method.

4.1. Hardware options

Microcontroller

The ESP32 is a microcontroller manufactured by Espressif Systems [1] with built-in Wi-Fi and Bluetooth capabilities. It is a popular choice for Internet of Things (IoT) applications [2] as it includes a dual-core processor, rich I/O interfaces such as UART, SPI, and I2C, support for low-power modes and over-the-air (OTA) updates. There are different ESP32 based chips and development boards available on the market that allow out-of-the-box prototyping and production capable systems.

The Firebeetle 2 ESP32-E from DFRobots is a ESP-WROOM-32E-based controller board that offers built-in wireless connectivity, a broad selection of GPIO ports and a built-in Li-ion battery port with USB charging capability. [3] It is selected as our main device used by meeting the requirements and having a good documentation and product quality.

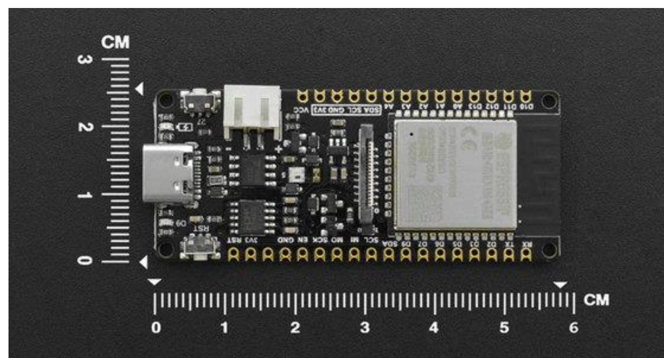

Figure 1. Top side of Firebeetle 2, powered by ESP32-WROOM-32E [3].

Other ESP32 based development boards are also included in this project for the purpose of testing the networking capability. Boards also used for this project:

- AZ-Delivery ESP32 DevKitC V2
- Wemos S2 Mini 1.0.0

Ambient light sensor

The WCMCU-101 breakout board is an ambient light sensor module based on the OPT101 photodiode [4]. Although the WCMCU-101 module lacks documentation, it was chosen for being at hand and being able to output analog signals with high frequency. For schematics, we are going to reference the CJMCU-101 [5], which seems to be identical to the WCMCU-101.

Figure 2. The ambient light sensor WCMCU-101 with a OPT101 photodiode. Photograph of the autor.

Temperature sensor

The WeMos DS18B20 shield is a digital temperature sensor based on the DS18B20 sensor [6]. This shield adds the ability to sense temperature to an accuracy of +/- 0.3 C and relative humidity to +/-3% . It is compatible with the Wemos S2 mini used in our system

4.2. Data exchange options

As our prototype is intended to work only in classroom scenarios, where the devices are usually in a line of sight and have no connection obstacles, we are going to look into solutions that create a local area network. Meaning that we are not going to cover cellular options.

Bluetooth

Bluetooth is a short-range wireless technology standard that is used for exchanging data between devices over short distances. Bluetooth connection is not a viable solution for us as the connection speed, which can reach up to 90 KB/s might be too slow for us later. Bluetooth also requires configuring for each session.

Wi-Fi

The ESP32 microcontroller boasts robust Wi-Fi capabilities, supporting both 2.4 GHz and 5 GHz bands for versatile connectivity options. With features such as Wi-Fi Direct and SoftAP, the ESP32 enables peer-to-peer communication and acts as an access point, facilitating seamless device-to-device connections. Furthermore, the ESP32's advanced Wi-Fi capabilities, coupled with its low power consumption, make it a preferred choice for Internet of Things (IoT) applications requiring reliable wireless connectivity. [7]

The root node of our system could create a local network by acting as a router and having other devices connect to it. Due to the connection limit of 15 devices the ESP32 based access point doesn’t fit our requirements. [8]

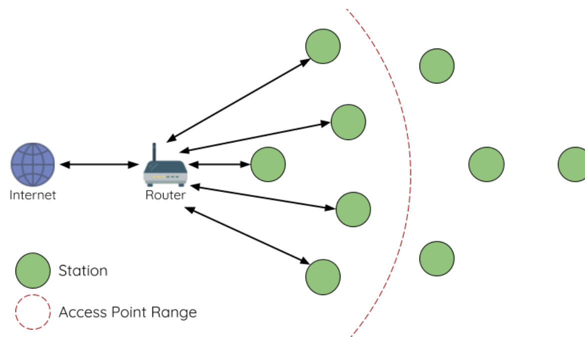

Figure 3. Visualization of Traditional Wi-Fi Network Architecture [9].

Wireless mesh networks

A mesh network is a local area network in which the nodes (devices) connect directly, dynamically and non-hierarchically to as many other nodes as possible and cooperate with one another to efficiently route data between each-other. This method allows larger networking systems as each device that exchanges data

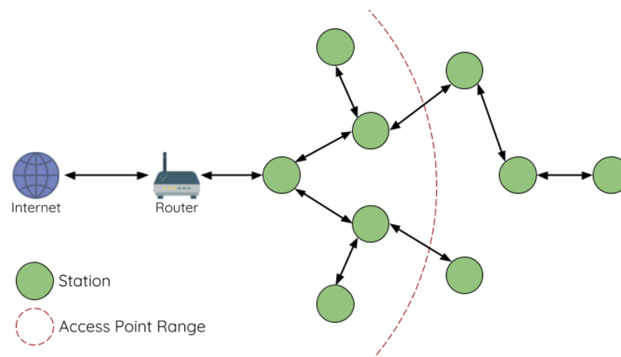

Figure 4. Visualization of a mesh network [9].

The difference between Esp-Mesh-Lite and traditional Wi-Fi networks is that the nodes in Esp-Mesh-Lite do not need to be connected to a central node, but can be connected to neighboring nodes. Each node is responsible for data forwarding of adjacent nodes through Wi-Fi connection. Without being limited by the distance to the central node, the Esp-Mesh-Lite network is able to cover a wider area. Similarly, without being limited by the capacity of the central node, Esp-Mesh-Lite allows more connections and is less prone to overload. At the same time, each node gets an IP address assigned by the parent node, hence accessing the network in the same way a single device accesses a router. During the process, the parent node only forwards this data on the network layer and ignores the application layer. [9]

In a Wi-Fi network, a station is limited to a single connection to an AP at any given time (upstream connection), while an AP can be simultaneously connected to multiple stations (downstream connection). However, the Esp-Mesh-Lite network allows a node to act as both station and AP, so a node in Esp-Mesh-Lite can establish multiple downstream connections using its SoftAP interface and one upstream connection using its station interface. This will naturally result in a tree network topology consisting of multiple layers of parent-child structures. [10]

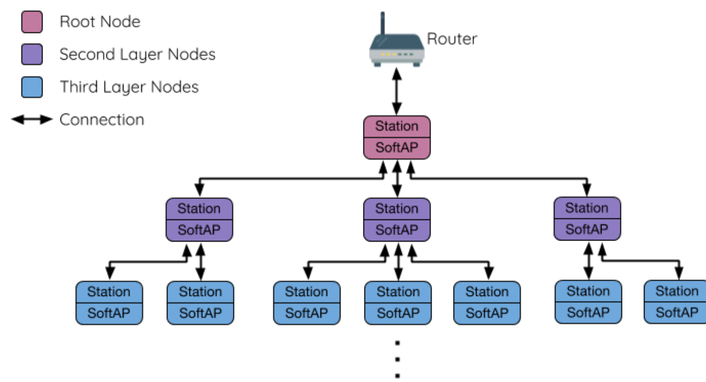

Figure 5. Tree topology in mesh network. [11].

Espressif published the ESP-MDF (Espressif Mesh Development Framework) in 2018 which enabled the development of a mesh network on ESP32 chips. It was an upgrade to the ESP8266-based mesh network published in 2015 that aimed to increase the connection reliability and data security. [12] Since the release of ESP-MDF v1.0 in 2020, the framework hasn't received any updates and is in limited maintenance.

In the February of 2023, Espressif published the first release of Mesh-Lite component v0.1.0. The biggest difference between Esp-Mesh-Lite and ESP-WIFI-MESH is that Esp-Mesh-Lite allows sub-devices in the network to independently access the external network, and the transmission information is insensitive to the parent node, which greatly reduces the difficulty to develop the application layer. Esp-Mesh-Lite is self-organizing and self-healing, which means the network can be built and maintained autonomously. [13] Although having a possibility of nodes connecting to external networks, it is not needed in our project.

The Esp-Mesh-Lite component has the following features that fit our requirements:

1. Bi-directional data flow - It's possible to broadcast a message to all of the nodes on the network and at the same time, send requests to the root node.
2. An average of 8 to 12 ms per-hop delay (in test conditions).
3. Supports up to 1000 devices, but has a recommended limit of 512 devices.
4. Has a self-healing ability when a node disconnects from the network.
5. The ESP-Mesh-Lite network does not require a gateway or border router, and any device in the network can connect to adjacent devices, making it easy to expand the network coverage.
6. OTA (Over-the-air) updates

The test numbers are based on the common performance metrics for the Esp-Mesh-Lite [14]. It is important to note that the performance depends on the environment that it's used in.

No-router example

Espressif has created an example Esp-Mesh-Lite project that doesn't require a router within the network. This project will be used as a starting point in the development of our project.

We are going to use JSON data format in the project as data payload are required to be cJSON objects. This is no problem for us as JSON format is the superior selection when handling request.

Although the example project console output indicates the possible size of the data packet (at least 140 characters), it would be good to know the upper byte limit for better data handling. Researching the web and documentation yielded no result on the Esp-Mesh-Lite payload size limit, but going through the open-source code it was certain that the payload size limit is comes from the fixed payload length limit set in the IoT-bridge framework, that is a building block for the mesh network. As uint16_t sets a maximum value of 65 535 characters for the payload, this is more than enough to fit our need of data transmission on the network.

```
typedef struct {
    union {
    #if defined(CONFIG_BRIDGE_EXTERNAL_NETIF_SDIO) ||
        defined(CONFIG_BRIDGE_DATA_FORWARDING_NETIF_SDIO)
        sdio_slave_buf_handle_t sdio_buf_handle;
    #endif
        wlan_buf_handle_t   wlan_buf_handle;
        void *priv_buffer_handle;
    };
    uint8_t if_type;
    uint8_t if_num;
    uint8_t *payload;
    uint8_t flag;
    uint16_t payload_len;
    uint16_t seq_num;

    void (*free_buf_handle)(void *buf_handle);
} interface_buffer_handle_t;
```

Figure 6. The interface of the buffer handler in the IoT-bridge.

4.3. Software

The ESP32 devices are programmed in ESP-IDF framework as Esp-Mesh-Lite library is not implemented on other frameworks. The ESP-IDF, Espressif IoT Development Framework, provides toolchain, API, components and workflows to develop applications for ESP32 using Windows, Linux and macOS operating systems. The prototype devices are programmed using the VSCode IDE for the reason of previous experience.

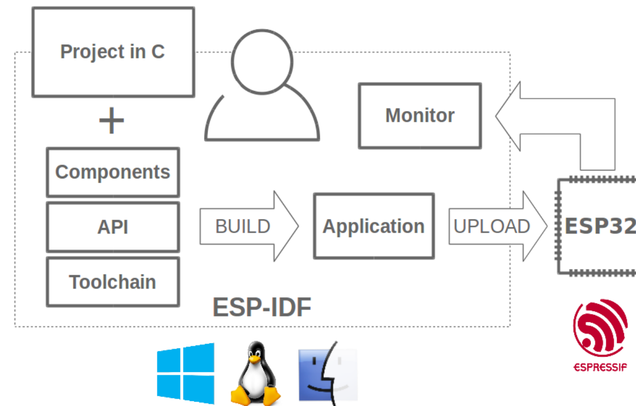

Figure 7. The functional scheme of the ESP-IDF [14].

Our prototype will be done on ESP-IDF version 5.1.2, which was the stable release version at the time of setting up the framework. It is maintained up to the end of 2025 [15] and after that it is advised to upgrade to a new version.

The Esp-Mesh-Lite component requires at least version 4.3.0 to be used on the main ESP32 chips. The chips used in our project, ESP32 and ESP32-S2 are both supported with the currently selected release.

Chip	ESP-IDF Release/v4.3	ESP-IDF Release/v4.4	ESP-IDF Release/v5.0	ESP-IDF Release/v5.1
ESP32	Supported	Supported	Supported	Supported
ESP32-C3	Supported	Supported	Supported	Supported
ESP32-S2	Supported	Supported	Supported	Supported
ESP32-S3		Supported	Supported	Supported
ESP32-C2			Supported	Supported
ESP32-C6				Supported

Figure 8. The Esp-Mesh-Lite release dates for chip support [9].

The original FreeRTOS, which runs the ESP-IDF, is a compact and efficient real-time operating system supported on numerous single-core devices. However, to support dual-core ESP targets, such as ESP32, ESP32-S3, and ESP32-P4, ESP-IDF provides a unique implementation of FreeRTOS with dual-core symmetric multiprocessing (SMP) capabilities.

By utilizing FreeRTOS tasks we can configure and create tasks that work on our selected core and have the right amount of memory allocated. This is very important when we need functions that work simultaneously and with different priorities. In our case we need to optimise the networking ability to have a stable connection with the host machine. The main system configuration can be set by using the SDK configuration editor that generates sdkconfig files. This is important as some packages provide default configs that help with the implementations.

Prototype web application

The web application interface for the host user will be developed in Node.js and Vue.js. Node.js is an asynchronous event-driven JavaScript runtime designed to build scalable network applications. We are going to use the release v20.12.2. Vue is a JavaScript framework for building user interfaces. It builds on top of standard HTML, CSS, and JavaScript and provides a declarative, component-based programming model that helps you efficiently develop user interfaces of any complexity. Node.js with Vue.js was chosen for the reason of previous experience and for being the industry standard for developing reactive web pages.

The serial connectivity through USB cable is achieved with Web Serial API by MDN Web Docs [16]. Although the API is only supported on Chromium web browsers [16], there is an implementation with WebUSB API that allows serial connection for Android devices also [17]. In the scope of this project we will be using the MDN Web Docs API.

As our prototype project does not store any data and will be used only in a local environment, we are not going to implement a database for it.

5. Developing the prototype system

The development of our prototype system sections begins by testing the ESP32-E capability to act as a processing unit for conducting classroom experiments followed by the hardware implementation for the end result. The implementation and connection of Esp-Mesh-Lite network with the monitoring interface is followed right after.

5.1. Analog sensors

Analog sensors detect and convert physical phenomena, such as temperature, pressure, light intensity, or sound, into proportional electrical signals. These sensors typically utilize various principles, including resistance, capacitance, inductance, or voltage, to measure and represent the input variables.

An analog-to-digital converter changes an analog signal that's continuous in terms of both time and amplitude to a digital signal that's discrete in terms of both time and amplitude. The analog input to a converter consists of a voltage that varies among a theoretically infinite number of values.

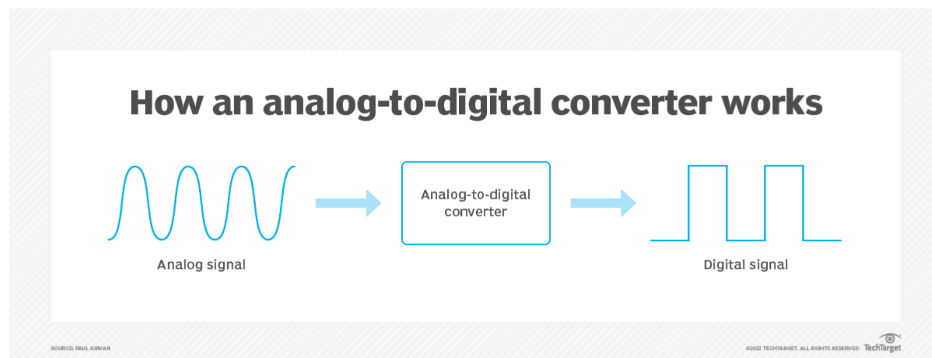

Figure 9. Analog-to-digital converter explanation [18]

The Analog to Digital Converter is integrated on the espressif chip and is capable of measuring analog signals from specific analog IO pins. [19] The ESP32 has two ADC units (channels), which can be used in to:

- generate one-shot ADC conversion result - one-time read of a sensor value
- generate continuous ADC conversion results - a conversion frame full of many reads. Also called DMA mode as it writes straight into memory and is rather used for transferring large chunks of data

A specific ADC unit can only work under one operating mode at any time, either in continuous mode or oneshot mode.

Photogate experiment

To conduct our sample experiment of measuring the speed of an object via photogate, the ambient light sensor was modified to take in light from only one path. To test the ESP analog sensor processing capability we are going to measure the speed of an index finger that is performing a flick.

ADC oneshot mode is suitable for low-frequency sampling operations like reading a temperature value every second. For high-frequency applications it's better to use the continuous mode for better accuracy. We are going to compare whether our prototype photogate's result can compete with the PASCO Wireless Smart Gate that is said to have a timing resolution of 3 ms. [20]

The ADC unit 1 on channel 6 (GPIO34) was selected for reading sensor data from the WCMCU-101 and we are going to test it in the continuous mode

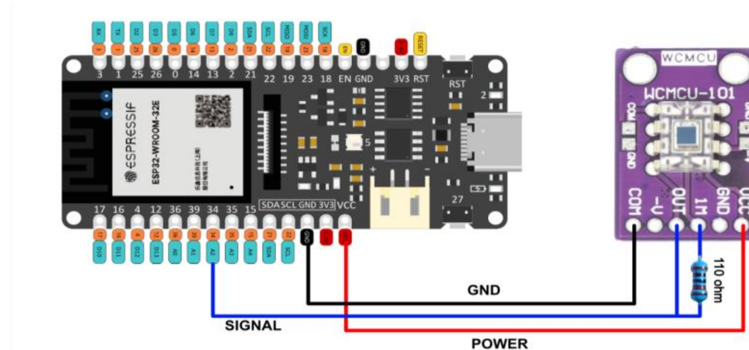

Figure 10. Firebeetle 2 ESP32-E Pinout for reading photosensor values. Scheme from the author.

Figure 11. Prototype photogate solution with a battery powered LED. Photograph of the autor.

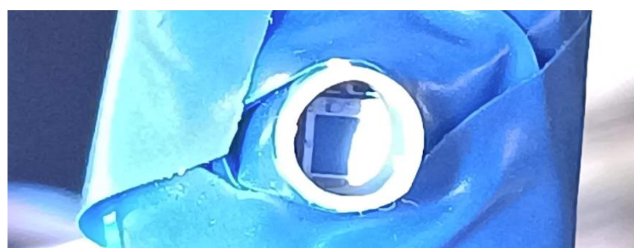

Figure 12. Modification on the light sensor to block the ambient light of the test environment. Photograph of the autor.

Continous mode ADC

For the testing base we are going to use ESP-IDF continuous read example [21].

After implementing a way to read and compare the sensor values, we get the following output after an average flick:

```
I (3683) TASK: ret is 0, ret_num is 512
I (3693) TASK: ret is 0, ret_num is 512
I (3703) TASK: ret is 0, ret_num is 512
I (3713) TASK: ret is 0, ret_num is 512
I (3723) TASK: ret is 0, ret_num is 512
I (3723) TASK: Result < 1000 value: 371
I (3723) TASK: Result < 1000 value: 535
I (3723) TASK: Result < 1000 value: 228
I (3723) TASK: Result < 1000 value: 274
I (3733) TASK: Result < 1000 value: 176
I (3733) TASK: Result < 1000 value: 190
I (3743) TASK: Result < 1000 value: 159

. . .
62 lines of reading values with similar results
. . .
I (4013) TASK: Result < 1000 value: 117
I (4023) TASK: Result < 1000 value: 135
I (4023) TASK: Result < 1000 value: 125
I (4033) TASK: Result < 1000 value: 364
I (4033) TASK: Result < 1000 value: 231
I (4043) TASK: Result < 1000 value: 981
I (4043) TASK: Result < 1000 value: 590
I (4053) TASK: ret is 0, ret_num is 512
I (4063) TASK: ret is 0, ret_num is 512
I (4073) TASK: ret is 0, ret_num is 512
I (4083) TASK: ret is 0, ret_num is 512
I (4093) TASK: ret is 0, ret_num is 512
```

Figure 13. Console output of ADC continuous mode read after flicking a finger through the prototype photogate.

To calculate the speed of an object going through the photogate we need the following values:

1. Sum of read values - The amount of values that prove there was an object in front of the sensor. In our case it’s 76.
2. Conversion frame read time - The time needed to read one conversion result. By checking the console output, that is in milliseconds, we get the conversion frame read time of 10 ms.
3. Conversion frame size - The total amount of readings in one conversion frame. The length of our conversion buffer was set to 512 values.
4. Object length - The distance covered by the object in a straight path when passing through the photogate. In our case the middle section of the finger is about 16 mm in diameter

$$t - time$$
$$s - distance\ covered\ (in\ metres)$$
$$v - speed$$

$$t = \frac{Sum\ of\ read\ values \times Conversion\ frame\ read\ time}{Conversion\ frame\ size}$$

$$v = \frac{s}{t} \Rightarrow v = \frac{distance\ covered\ (m) \times conversion\ frame\ size}{sum\ of\ read\ values \times conversion\ frame\ read\ time\ (s)}$$

$$v = \frac{0.016\ m \times 512}{76 \times 0.01} = 10.78\ \text{m/s}$$

By going through the calculations of other people [22]. The ~11 m/s read is a considerable answer. This means that the ESP chip can compete with other sensors available on the market in the scope of measuring accuracy. The implementation can be found from the source code link added to the extras

5.2. External devices setup

RGB LED setup

The built-in WS2812 RGB LED, controlled by pin IO5 on the ESP32-E board, was used for indicating the connection status. The LED setup was done with ESP-IDF LED Control API [23].

Button setup

Each of the ESP32 boards used had a built-in addressable button. As the IO channels were different on the boards an extra configuration check was added in the beginning of the main file. Even though there was a way to detect the board for building files via configuration, the ESP32 and ESP32-E were both on the same chip framework and it was easier to manually configure the right GPIO rather than create a custom setup.

For the prototype solution we used only the development boards' built-in button in GPIO mode for an easy setup. The advantage of the GPIO mode is that each button occupies an independent IO and therefore does not affect each other, and has high stability; however when the number of buttons increases, it may take too many IO resources. When we are in need of more buttons in the future, it is possible to utilize the ADC mode. The advantage of using the ADC button is that one ADC channel can share multiple buttons and occupy fewer IO resources. The disadvantages include that you cannot press multiple buttons at the same time, and instability increases due to increase in the closing resistance of the button due to oxidation and other factors.

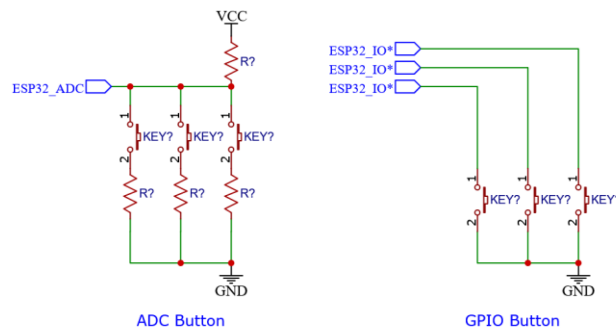

Figure 14. Button connection schemes of ADC and GPIO modes [23].

Temperature shield

As a way to showcase the system's ability to exchange and visualize data, the Firebeetle ESP32 and Wemos D1 Mini boards were fitted with a 1-wire temperature shield DS18B20. Unfortunately there were not enough temperature sensors available for all boards for testing and the Ez-delivery boards didn't send a temperature reading.

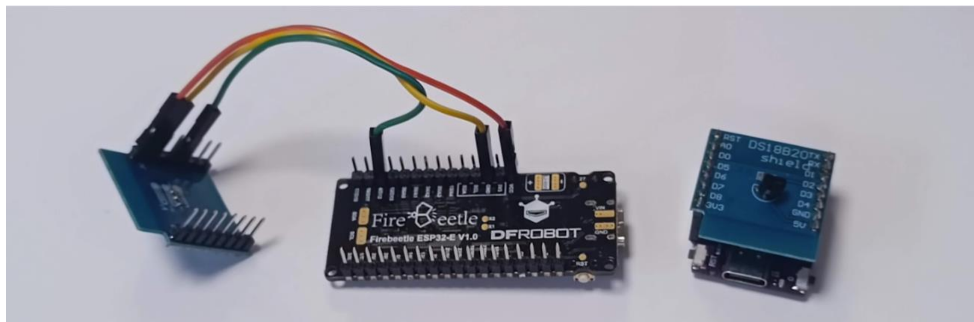

Figure 15. Firebeetle 2 ESP32-E and ESP32-S2 Wemos D1 boards with the temperature shields attached. Photograph of the autor.

The implementation of the DS18B20 temperature sensor was done with the help of the DS18B20 Device driver library [24]. The setup code was modified in the way that the sensor object was globally accessible for other functions.

Although the library was intended to communicate with a 1-wire bus setup, it manages to read the temperature value from the DS18B20 shield.

5.3. Esp-Mesh-Lite setup

The setup of the Esp-Mesh-Lite example was straight-forward and it was important to understand what configuration parameters adjust the overall performance. The network information was not modified as it was not needed in the scope of this project. It’s possible to change the ssid and password of the network to make it more secure in the future if needed.

To start the mesh network it’s essential to use the following functions in the main function in the correct order.

```
void app_main() {
esp_storage_init();
    ESP_ERROR_CHECK(esp_netif_init());
    ESP_ERROR_CHECK(esp_event_loop_create_default());
    esp_bridge_create_all_netif();
    wifi_init();
    esp_mesh_lite_config_t mesh_lite_config = ESP_MESH_LITE_DEFAULT_INIT();
    mesh_lite_config.join_mesh_ignore_router_status = true;

#if CONFIG_MESH_ROOT
    mesh_lite_config.join_mesh_without_configured_wifi = false;
#else
    mesh_lite_config.join_mesh_without_configured_wifi = true;
#endif

    esp_mesh_lite_init(&mesh_lite_config);
    app_wifi_set_softap_info();

#if CONFIG_MESH_ROOT
    ESP_LOGI(TAG, "Root node");
    esp_mesh_lite_set_allowed_level(1);
#else
    ESP_LOGI(TAG, "Child node");
#endif

    esp_mesh_lite_start();
}
```

Figure 16. Esp-Mesh-Lite setup code.

The no-router example project provided by Espressif contained a non-fatal bug on the root node. The error was published as a github issue and it was said to be fixed later. [26]

```
I (2523) [mesh-lite-espnow]: Start espnow task
I (2527) [ESP_Mesh_Lite_Comm]: msg action add success
I (2535) no_router: Root node
I (2537) Mesh-Lite: Mesh-Lite connecting
I (2541) main_task: Returned from app_main()
I (7317) wifi:new:<11,2>, old:<11,2>, ap:<11,2>, sta:<255,255>, prof:11
I (7319) wifi:station: fc:b4:67:53:39:a0 join, AID=1, bgn, 40D
I (7335) bridge_wifi: STA Connecting to the AP again...
I (7362) esp_netif_lwip: DHCP server assigned IP to a client, IP is: 192.168.4.2
I (8376) wifi:<ba-add>idx:2 (ifx:1, fc:b4:67:53:39:a0), tid:0, ssn:0, winSize:64
E (32493) [ESP_Mesh_Lite_Comm]: ESP_Mesh_Lite_Comm
E (37493) [ESP_Mesh_Lite_Comm]: ESP_Mesh_Lite_Comm
E (42493) [ESP_Mesh_Lite_Comm]: ESP_Mesh_Lite_Comm
```

Figure 17. Console output of the ESP32 that is configured as the root node. Screenshot by the author.

The number of maximum levels and downstream connections were tested and in all conditions the devices managed to form a functioning mesh network. Default settings for connection were the following:

```
CONFIG_MESH_LITE_MAXIMUM_LEVEL_ALLOWED=5
CONFIG_BRIDGE_SOFTAP_MAX_CONNECT_NUMBER=6
```

Figure 18. Default settings for mesh network.

Aside from the default config file that was attached with the example project, the following items were also modified to empower the connectivity and processing speeds.

```
CONFIG_ESP_DEFAULT_CPU_FREQ_MHZ_240
CONFIG_LWIP_TCPIP_TASK_PRIO=3

CONFIG_LWIP_TCP_SND_BUF_DEFAULT=14360
CONFIG_LWIP_TCP_WND_DEFAULT=14360
CONFIG_LWIP_TCP_RECVMBOX_SIZE=12
CONFIG_TCP_SND_BUF_DEFAULT=14360
CONFIG_TCP_WND_DEFAULT=14360
CONFIG_TCP_RECVMBOX_SIZE=12
```

Figure 19. Modified default config values.

A key discovery was increasing the CONFIG_LWIP_TCPIP_TASK_PRIO (*LWIP tcpip task priority*) from the default 18 to a level 3. In the documentation it is advised that in case of high throughput, this parameter could be changed up to (configMAX_PRIORITIES-1). Raising the task priority makes the processing unit handle this task before the other ones that have a lower priority. I didn't see the need to raise it any further as it may increase the system instability. This enabled for faster network forming and data exchanging.

Event handling

To exchange data between nodes on the network, it was needed to create and register listeners on the network layer according to the following interface.

```
/*** @brief Mesh-Lite message action parameters passed to
esp_mesh_lite_msg_action_list_register call.*/
typedef struct esp_mesh_lite_msg_action {
    const char* type;          /**< The type of message sent */
    const char* rsp_type;      /**< The message type expected to be received*/
                               /**< When a message of the expected type is
                                            received, stop retransmitting*/
                               /**< If set to NULL, it will be sent until the
                               maximum number of retransmissions is reached*/
    msg_process_cb_t process; /**< The callback function when receiving the
                               'type' message, The cJSON information in the type
                               message can be processed in this cb*/
} esp_mesh_lite_msg_action_t;
```

Figure 20. The messaging interface for mesh-lite connection.

Unfortunately there is no documentation available for the Esp-Mesh-Lite functions at the time of writing and the data exchange methods were created by reverse engineering the built-in functions of the example project. This enabled me to create callback functions for handling the events.

```
static cJSON* event_info_process(cJSON *payload, uint32_t seq) {
    ESP_LOGD( "event_info_process", "reached");
        // Do something when this has been reached
    return NULL;
}

static cJSON* event_info_ack_process(cJSON *payload, uint32_t seq) {
    ESP_LOGD("event_info_ack_process", "event callback received");
        // Something has been done and this is a callback for it
    return NULL;
}

static const esp_mesh_lite_msg_action_t node_event_action[] = {
    // Message to listen to,
    {"event_info", "event_info_ack", event_info_process},
    {"event_info_ack", NULL, event_info_ack_process},

    {NULL, NULL, NULL} /* Must be NULL terminated */
};

void app_main() {
    esp_mesh_lite_msg_action_list_register(node_event_action);
}
```

Figure 21. Callback functions for messaging in Esp-Mesh-Lite.

UART connection setup

The easiest and most secure data exchange method between the monitoring interface and root node is by creating a serial connection via regular data transferring USB cable. The root device has a Universal Asynchronous Receiver/Transmitter set up for reading and writing messages from the serial connection.

An additional event handling had to be implemented to publish messages coming from the monitoring interface as the Mesh-lite didn't have a function for publishing messages to the leaf nodes at the ends of the mesh network.

The main issue that would need a future upgrade is the self-healing ability of the mesh network. As the main use cases for the ESP32 mesh networks are systems that are either always online and in a fixed position, the self-healing part takes some time to happen.

5.4. Monitoring interface

The prototype solution for the monitoring interface was built on Vue.js by using Node.js for hosting. Pinia.js was used for managing the Vue.js states and objects in a reactive and efficient way. Vue router was included for navigation. The functional part is written in typescript for type-checking of objects and allows to catch errors more effectively. Web Serial API was used for data exchange with ESP32 through the USB port. Although the Web Serial API isn't compatible with Firefox, Safari and mobile devices right now, but it was still chosen for the reason of being fully supported on chromium based browsers.

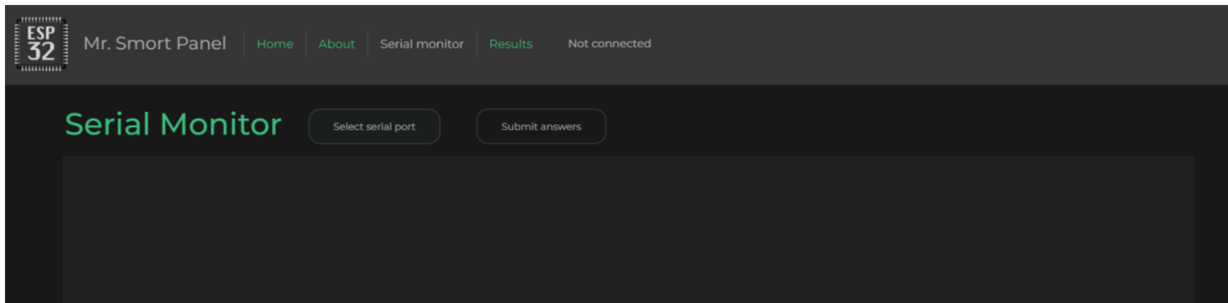

Figure 22. Serial monitor view when opening the interface. Screenshot by the author.

Opening a connection with USB

To create a connection with the ESP32 the user has to connect the computer with the ESP32 with an USB cable. By pressing the "Select serial port" button, the user can select the ESP32 from the pop-up menu. The serial port selection is filtered to only show serial ports that have a specific vendor Id. This helps the user not to select other USB peripherals connected to the computer.

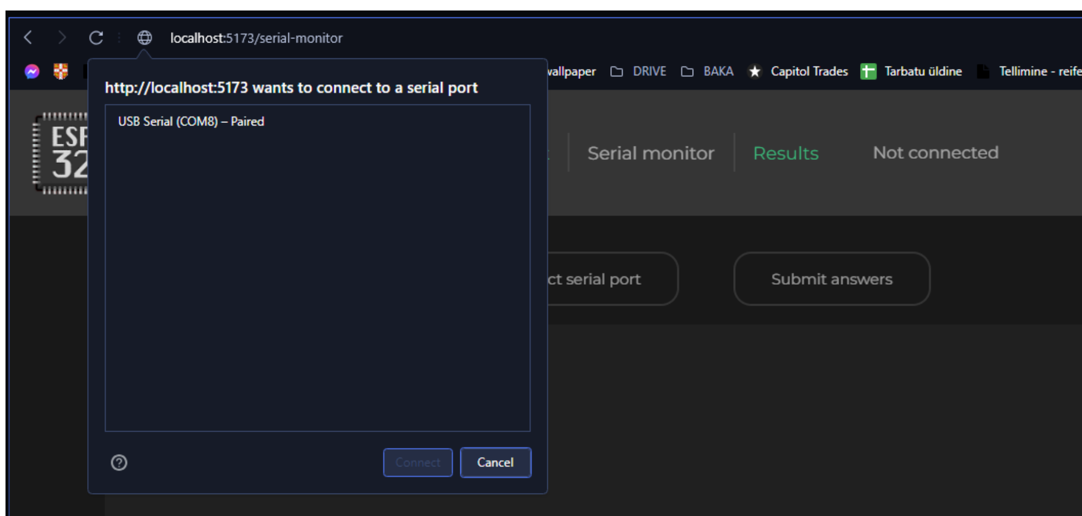

Figure 23. Serial port selection pop-up by the Web Serial API. Screenshot by the author.

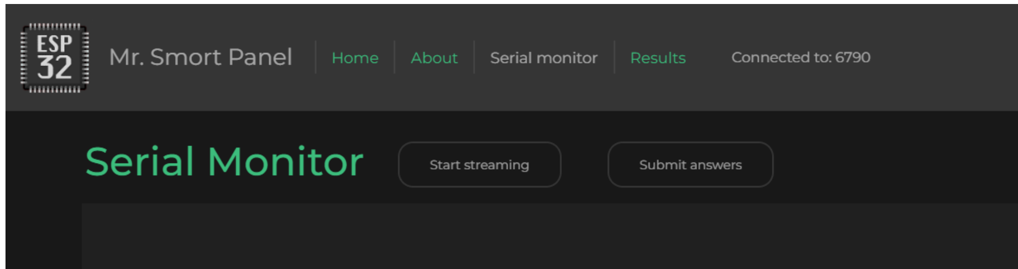

Figure 24. Serial monitor view after selecting the port. Screenshot by the author.

After selecting the port and pressing “Start streaming” the interface is populated with all of the devices on the mesh network that are in the same network as the root node. The root node device connected via USB is not in the shown devices list as it’s used solely for connecting the interface with the mesh network.

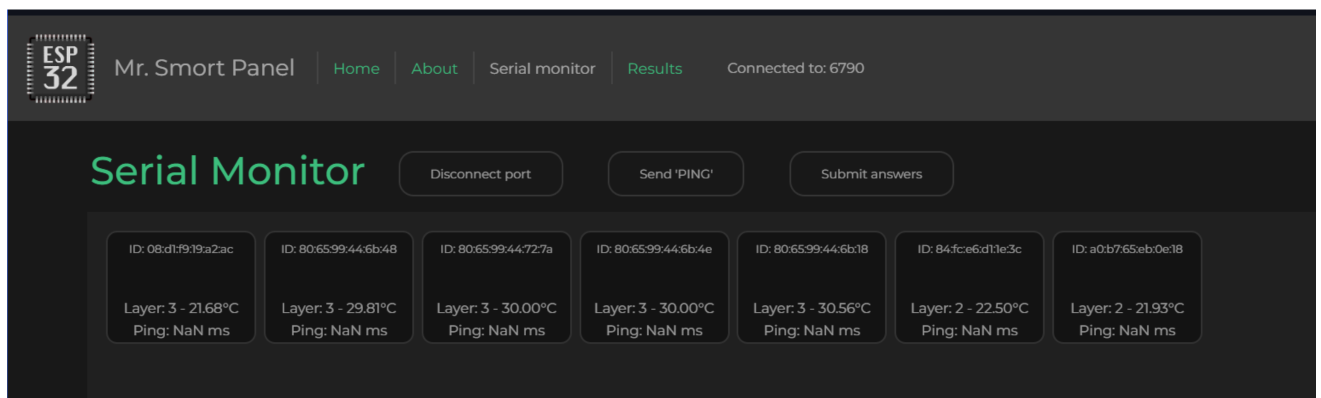

Figure 25. Serial monitor view after opening the connection through USB. Screenshot by the author.

Connection with devices

The mesh network devices send JSON payloads to the root node that get parsed by the connection store built with Pinia. The payload from the devices contain the following key and value pairs:

1. **event :** request type as a string value
2. **s_mac:** mac address of the ESP32 node as a string value
3. **p_mac:** mac address of the ESP32 parent node as a string value
4. **lev**: mesh level of the node as a number
5. **opt**: answer option of the user as a string value
6. **temp**: sensor value as a string

The “send PING" button is available after opening a serial connection and showcases the ability of publishing messages to the network. Each node replies to the event by selecting the next answer option and publishing it back to the interface. This is useful for testing the response

time of the devices. Latency is calculated in the backend by attaching two timestamps for each device object. The first timestamp is saved at the time of sending the “ping” and the second after receiving a reply from the device. Implementation is added to extras.

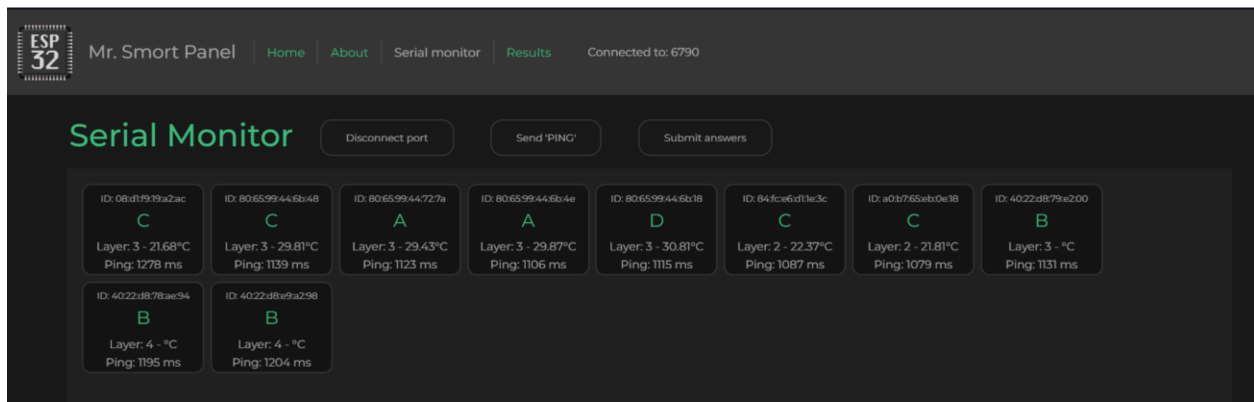

Figure 26. Ping results on the Serial monitor tab. Screenshot by the author.

This implementation fills our requirement for bi-directional data exchange between the interface and mesh network. We can broadcast messages to the whole network or singlecast to specific nodes by using their mac address. The connection latency depends on the processing power of node devices and has a minimum latency of 1000 ms in our test conditions. As our web application is locally hosted, the latency is also affected by other processes happening on the computer.

Audience response system

Teachers can use the ESP powered devices to conduct polling sessions for active feedback. The Serial monitor view outputs the selected options on the main view. The event of an ESP32 pressing a button reaches the interface with an unnoticeable delay even when being on the last layer of the network.

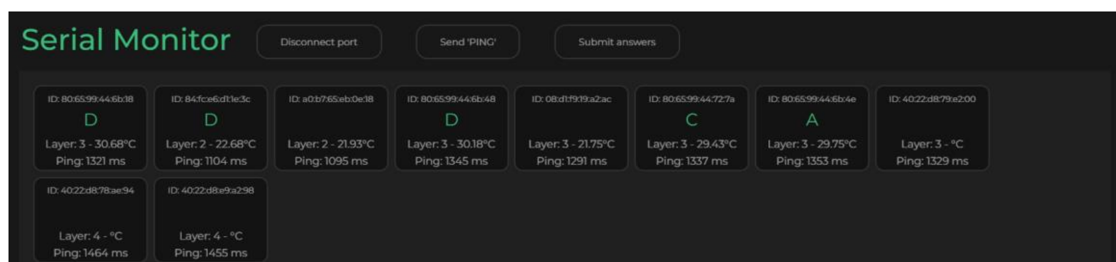

Figure 27. Result of pressing the button on the student devices. Screenshot by the author.

The results are saved and cleared after the teacher presses the “Submit answers” button. Results are then accessed from the “Results” tab in the header.

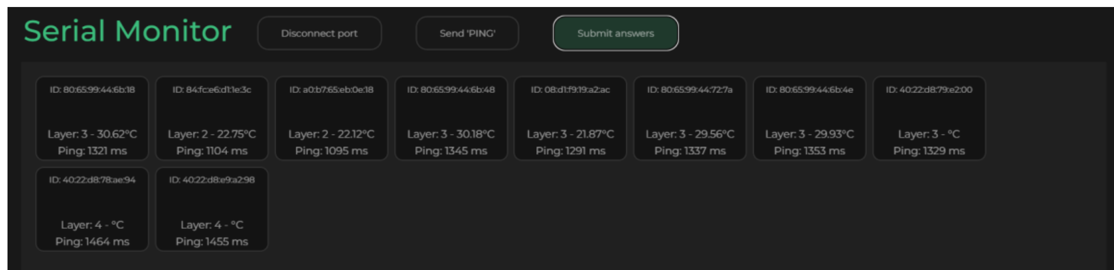

Figure 28. Serial monitor view after submitting the answers with the “Submit answers” button. Screenshot by the author.

The latest polling session is automatically selected after opening the “Results” view. The teacher can select the polling session which they like to visualize on the interface.

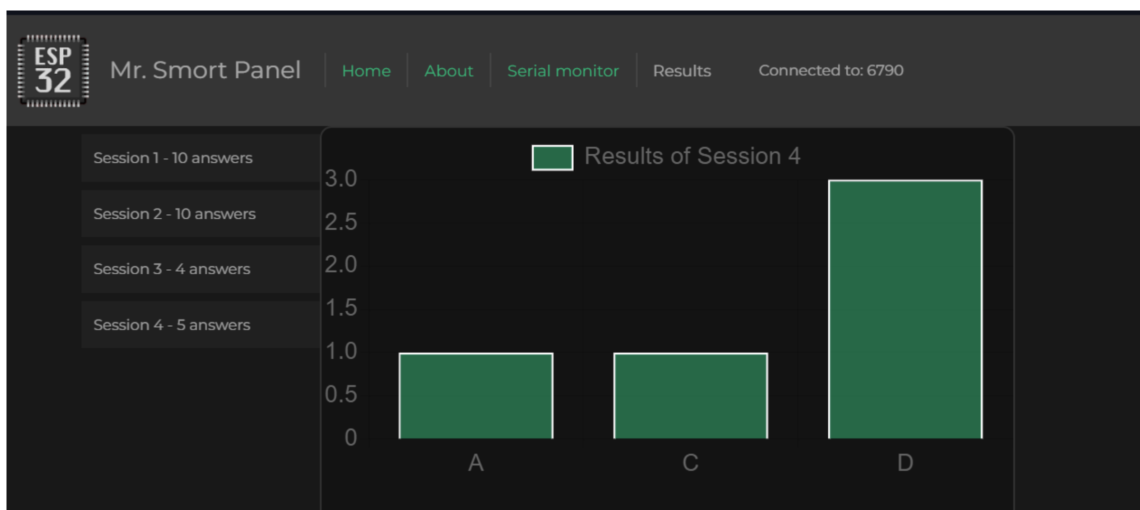

Figure 29. Bar charts created from the saved polling sessions. Screenshot by the author.

The chart visualization of each polling session is implemented with ChartJS [27]. Thanks to the reactiveness build we can use the same graph component in the Serial monitor vue to have a live graph view.

The monitoring interface met all of the requirements set and had no issues with testing.

6. Conclusions

The aim of this thesis was to design and build a prototype system which helps to manage practical lessons either in the way of conducting experiments or by being used as an audience response system. An important requirement for the solution was to have an interface for the teacher to control all of the devices being used in the practical lesson.

The prototype system filled all the general requirements as the student devices don't require any configuration for connection with the teacher's interface. The teacher interface connects to the network in any chromium browser by using the root ESP32 device and an USB cable. The mesh network between the ESP32 devices is self-healing and supports connecting at least 24 devices to the network.

The requirements for the monitoring interface used by the teacher were also filled as the serial connection paired with the Esp-Mesh-Lite mesh network allows bi-directional data flow, which was successfully tested. The overview and stages on the teacher interface was reactive and achieved an average latency of 1-1.2 seconds in our locally hosted environment.

The software and hardware requirements were achieved by choosing ESP32-E development board as the main microcontroller, which has a built-in battery charging function and enough processing capability for conducting experiments. Over-the-air updates weren't tested in the scope of this thesis, but the Esp-Mesh-Lite system allows the implementation.

6.1. Development experience

The most timeconsuming task of the thesis was designing the end solution that would fit all of the requirements. This meant anaysing every possible solution that could achieve the desired result.

There were different problems that had to be tackled during the development phase. The first and largest was to achieve a fully working and bi-directional data exchanging mesh network. As the documentation for the Esp-Mesh-Lite package didn't have any good examples and similar solutions on the internet, the best way to gather information was by reverse engineering the built-in components.

Although developing the system in ESP-IDF provided many benefits, there was also great responsibility for handling memory allocation on the board's resources. It was enough to have just one line of code with leaking memory to make the system crash after being alive for ~100-300 seconds. The `heap_caps_get_free_size()` method was used for memory leak debugging.

The serial data buffering from the browser's side required a custom solution for reading JSON data as the serial buffer chopped up each JSON into 3-5 uneven parts. And for the reason that there might be two different JSON messages coming through the serial at the same time, there was a need for a custom handler. The custom solution can be found in the *connection.ts* file from the monitoring interface source code repository.

It was great to see that the speed of data transfer from the ESP32 to the teacher's interface was almost seamless. It would be great to test the system limits with up to 100 devices working simultaneously.

7. Acknowledgments

A great thanks goes to **Dr. Ulrich Norbisrath,** who contributed to the thesis by providing knowledge on the capability of ESP32 controllers and sharing information on different networking options available. Ulrich also contributed by providing devices and peripherals needed to test the capability of the system from the user's side (Wemos ESP32-S2 minis, power cables, DS18P20 temperature shields).

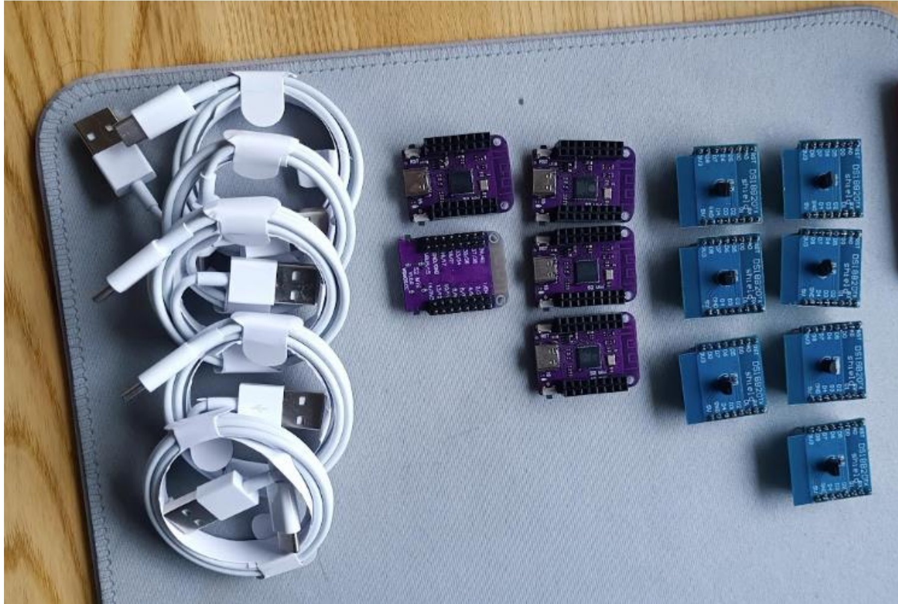

Figure 30. Components provided by Ulrich. Photograph of author.

I would also like to thank **Dr. Kaido Reivelt**, who offered me the topic for this thesis and provided me with equipment to start with the development (AZ-Delivery Dev Kit C, ambient light sensors). Kaido also contributed with his great expertise in the field of practical teaching by setting the right requirements for the prototype solution.

8. References

[1] Espressif Systems: Wireless SoCs, Software, Cloud and AIoT Solutions. Retrieved May 15, 2024, from https://www.espressif.com

[2] Marković, M. (2023, November 16). *Why is ESP the most used IoT connectivity MCU*. Byte Lab. Retrieved May 15, 2024, from https://www.byte-lab.com/why-is-esp-the-most-used-iot-connectivity-mcu/

[3] *FireBeetle_Board_ESP32_E_SKU_DFR0654-DFRobot*. (n.d.). DFRobot WIKI. Retrieved May 15, 2024, from https://wiki.dfrobot.com/FireBeetle_Board_ESP32_E_SKU_DFR0654

[4] *OPT101 Monolithic Photodiode and Single-Supply Transimpedance Amplifier datasheet (Rev. B)*. (n.d.). Texas Instruments. Retrieved May 15, 2024, from https://www.ti.com/lit/ds/symlink/opt101.pdf

[5] T.K. Hareendran (2022, October 20) *OPT101 Photodiode and CJMCU – 101 Module* - ElectroSchematics.com. Retrieved May 15, 2024, from https://www.electroschematics.com/photodiode/

[6] *DS18B20 - Programmable Resolution 1-Wire Digital Thermometer*. (n.d.). Analog Devices. Retrieved May 15, 2024, from https://www.analog.com/media/en/technical-documentation/data-sheets/ds18b20.pdf

[7] Wi-Fi - ESP32 ESP-IDF Programming Guide v5.2.1 documentation. Retrieved May 15, 2024, from https://docs.espressif.com/projects/esp-idf/en/stable/esp32/api-reference/network/esp_wifi.html

[8] Wi-Fi Driver - ESP32 - — ESP-IDF Programming Guide v5.2.1 documentation. Retrieved May 15, 2024, from https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-guides/wifi.html#ap-basic-configuration

[9] GitHub. Retrieved May 15, 2024, from https://github.com/espressif/esp-mesh-lite/blob/master/components/mesh_lite/User_Guide.md#introduction*esp-mesh-lite/components/mesh_lite/User_Guide.md at master · espressif/esp-mesh-lite*.

[10] GitHub. Retrieved May 15, 2024, from https://github.com/espressif/esp-mesh-lite/blob/master/components/mesh_lite/User_Guide.md#tree-topology*esp-mesh-lite/components/mesh_lite/User_Guide.md at master · espressif/esp-mesh-lite.*

[11] Espressif Systems: Wireless SoCs, Software, Cloud and AIoT Solutions. Retrieved May 15, 2024, from https://www.espressif.com/en/news/esp-mesh-development-framework-released

[12] GitHub. Retrieved May 15, 2024, from https://github.com/espressif/esp-mesh-lite/blob/master/components/mesh_lite/User_Guide.md#overview*esp-mesh-lite/components/mesh_lite/User_Guide.md at master · espressif/esp-mesh-lite.*

[13] GitHub. Retrieved May 15, 2024, from https://github.com/espressif/esp-mesh-lite/blob/master/components/mesh_lite/User_Guide.md#performance*esp-mesh-lite/components/mesh_lite/User_Guide.md at master · espressif/esp-mesh-lite*.

[14] Espressif Systems: Wireless SoCs, Software, Cloud and AIoT Solutions. Retrieved May 15, 2024, from https://docs.espressif.com/projects/esp-idf/en/v5.1.2/esp32/about.html

[15] GitHub. Retrieved May 15, 2024, from https://github.com/espressif/esp-idf?tab=readme-ov-file#esp-idf-release-support-schedule

[16] *Web Serial API - Web APIs | MDN*. (2024, April 3). MDN Web Docs. Retrieved May 15, 2024, from https://developer.mozilla.org/en-US/docs/Web/API/Web_Serial_API

[17] GitHub. Retrieved May 15, 2024, from https://github.com/google/web-serial-polyfill

[18] *What is analog-to-digital conversion (ADC)?,* (updated July 2022). TechTarget. Retrieved May 15, 2024, from https://www.techtarget.com/whatis/definition/analog-to-digital-conversion-ADC

[19] *Analog to Digital Converter (ADC) Continuous Mode Driver*. ESP-IDF Programming Guide v5.1.4 Retrieved May 15, 2024, from https://docs.espressif.com/projects/esp-idf/en/v5.1.4/esp32/api-reference/peripherals/adc_continuous.html

[20] Jonathan H. - *Wireless Smart Gate • PS-3225* . Home / Products / Sensors / Wireless Wireless Smart Gate. Retrieved May 15, 2024, from https://www.pasco.com/products/sensors/wireless/ps-3225#documents-panel

[21] GitHub. Retrieved May 15, 2024, from https://github.com/espressif/esp-idf/tree/d29e53dc0c59d1b1b01c5e2c3ca1b22113828587/examples/peripherals/adc/continuous_read

[22] *Finger snapping: Maximum fingertip speed and generated pressure change*. (2015, August 20). Physics Stack Exchange. Retrieved May 15, 2024, from https://physics.stackexchange.com/questions/201618/finger-snapping-maximum-fingertip-speed-and-generated-pressure-change

[23] *LED Control (LEDC) - ESP32 - — ESP-IDF Programming Guide v5.2.1 documentation*. (n.d.). Technical Documents. Retrieved May 15, 2024, from https://docs.espressif.com/projects/esp-idf/en/stable/esp32/api-reference/peripherals/ledc.html

[23] *Button* - ESP-IoT-Solution latest documentation. Retrieved May 15, 2024, from https://docs.espressif.com/projects/esp-iot-solution/en/latest/input_device/button.html

[24] *DS18B20 Device Driver* - GitHub. Retrieved May 15, 2024, from https://github.com/espressif/esp-bsp/tree/master/components/ds18b20

[25] *RMT Transmit Example -- LED Strip*. (2024, February 8). GitHub Pages. Retrieved May 15, 2024, from https://github.com/espressif/esp-idf/tree/release/v5.1/examples/peripherals/rmt/led_strip

[26] *Github issue: Repeating error from no_router example root node: E (32493) [ESP_Mesh_Lite_Comm]: ESP_Mesh_Lite_Comm (AEGHB-639) #82* - GitHub. Retrieved May 15, 2024, from https://github.com/espressif/esp-mesh-lite/issues/82

[27] Chart.js | Open source HTML5 Charts for your website. Retrieved May 15, 2024, from https://www.chartjs.org

9. Appendix

I. Access to the code

The source code is published to the github of the author of this thesis under GPL-3.0 license for the following parts:

1. Monitoring interface: https://github.com/martenainult/smort-panel
2. Esp-Mesh-Lite implementation: https://github.com/martenainult/esp-mesh-lite-smort-network
3. ESP32 Sample experiment: https://github.com/martenainult/DMA_photogate_ESP32

II. Licence

Non-exclusive licence to reproduce the thesis and make the thesis public

I, Marten Vainult,

(*author's name*)

1. grant the University of Tartu a free permit (non-exclusive licence) to

reproduce, for the purpose of preservation, including for adding to the DSpace digital archives until the expiry of the term of copyright, my thesis

ESP32 Based Mesh Network Solution for Managing Practical Lessons,

(*title of thesis*)

supervised by Kaido Reivelt.

(*supervisor's name*)

2. I grant the University of Tartu a permit to make the thesis specified in point 1 available to the public via the web environment of the University of Tartu, including via the DSpace digital archives, under the Creative Commons licence CC BY NC ND 4.0, which allows, by giving appropriate credit to the author, to reproduce, distribute the work and communicate it to the public, and prohibits the creation of derivative works and any commercial use of the work until the expiry of the term of copyright.

3. I am aware of the fact that the author retains the rights specified in points 1 and 2.

4. I confirm that granting the non-exclusive licence does not infringe other persons' intellectual property rights or rights arising from the personal data protection legislation.

Marten Vainult
15/05/2024