An-Najah National University
Computer Engineering

Remote Wiring System for Academic Laboratories: Real-Time Web-Controlled Multiplexing

[ReWire]

Graduation Project Report

Jehad Eyad Roshdi Hmoudah
Qusay Bader Rifat Salman

Supervisor: Dr. Mahmoud Assad Dwikat

September, 2025

Acknowledgments

We would like to express our sincere gratitude to our project supervisor Dr. Mahmoud Assad for his flexible time schedule, kind treatment, and continuous support throughout this project. His guidance and encouragement were invaluable in helping us stay focused and overcome challenges.

We also extend our heartfelt thanks to our families and friends for their unwavering support, patience, and motivation during the development of this project. Their encouragement inspired us to persevere and achieve our goals.

Finally, we acknowledge the broader academic community whose research and innovations in embedded systems, networking, and educational technology have informed and inspired our work.

Abstract

This project presents a real-time control system in which an ESP32-CAM acts as the master controller of eight Arduino Pro Minis using the I^2C protocol. Each Arduino drives two CD74HC4067 demultiplexers, resulting in a total of 16 demux modules. The system provides 16 nodes of type A and 16 nodes of type B that can be flexibly interconnected by the user through a web-based dashboard (hosted on Firebase). This enables remote configuration and monitoring of wiring experiments with minimal latency.

The platform is initially oriented toward **academic laboratories**, where it can serve as a valuable tool in the **Microcontroller Laboratory** for remote configuration and testing, in the **Electronics Laboratory** for signal routing without manual rewiring, in the **Mechatronics Laboratory** when paired with relays, and in the **Chemistry Laboratory** when extended with pumps or valves, etc.

However, its applicability is not confined to academic settings. Thanks to its modular and scalable design, the system can also be employed in industrial prototyping, automated testing facilities, research environments, and many other contexts where dynamic, remotely configurable wiring is required. In this sense, the project demonstrates a flexible foundation that can be expanded far beyond laboratory use — limited only by the creativity and needs of its users.

The system targets low-latency multiplexed device control across 16×16 connection points, emphasizing reliability, observability, and simple deployment. We detail the hardware design (level shifting, power, and signal integrity), firmware (I^2C protocol, retry and timeout logic), and the web client (RTDB schema, state reconciliation, and UI). Benchmarks show an end-to-end command latency of $< 300\,\text{ms}$ under typical Wi-Fi conditions, demonstrating the feasibility of using this platform as a remote wiring system for academic laboratories and beyond.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

In many experimental and prototyping environments, frequent rewiring is required to connect and reconfigure devices. This process is often time-consuming, error-prone, and limited by physical access to the hardware. Our motivation was to design a system that enables **remote and real-time wiring control**, thereby reducing manual intervention and improving efficiency.

Demultiplexers (MUX/DEMUX) provide an effective foundation for such a system, as they allow multiple signal paths to be dynamically selected and controlled through simple digital commands. By combining this with a web-enabled controller, we sought to create a platform that not only supports **academic laboratories**—such as microcontroller, electronics, mechatronics, and chemistry labs—but also remains sufficiently generic to be extended to other fields, including industrial prototyping, automated testing, and research environments.

In this way, the project aims to serve both as a **teaching aid** for academic institutions and as a **flexible solution** adaptable to a wide range of applications where dynamic, remotely configurable wiring is required.

1.2 Problem Statement

Modern laboratories and prototyping environments often rely on multiple microcontrollers and peripheral devices that need to be interconnected in different configurations. Traditionally, this requires manual rewiring, which is time-consuming, error-prone, and impractical in scenarios where rapid reconfiguration or remote access is needed.

The problem, therefore, is to design a system that can **coordinate multiple microcontrollers and I/O expanders with live feedback**, while also providing the flexibility to **switch connections dynamically** without manual intervention. For in-

stance, a user may initially connect a servo motor to an Arduino board to run a control experiment, then later need to rewire that same servo to a Raspberry Pi for further testing. Similarly, sensors might need to be alternated between different controllers, or actuators redirected to different processing units depending on the experiment.

Crucially, the system should allow **bidirectional connections**, where signals can flow in both directions, and it must also support control of both the **positive and negative wiring paths**. This ensures that power, ground, and signal lines can all be switched and redirected remotely, giving the user full flexibility in defining hardware interconnections.

These requirements highlight the need for a **remote wiring system** capable of managing complex device interconnections in real time. Such a system should reduce physical wiring effort, provide immediate feedback on the current state of connections, and enable users to explore multiple hardware configurations quickly and reliably. While particularly valuable in academic laboratories, such a system also has broad potential applications in industrial prototyping, automated testing setups, and research environments where dynamic reconfiguration of hardware is essential.

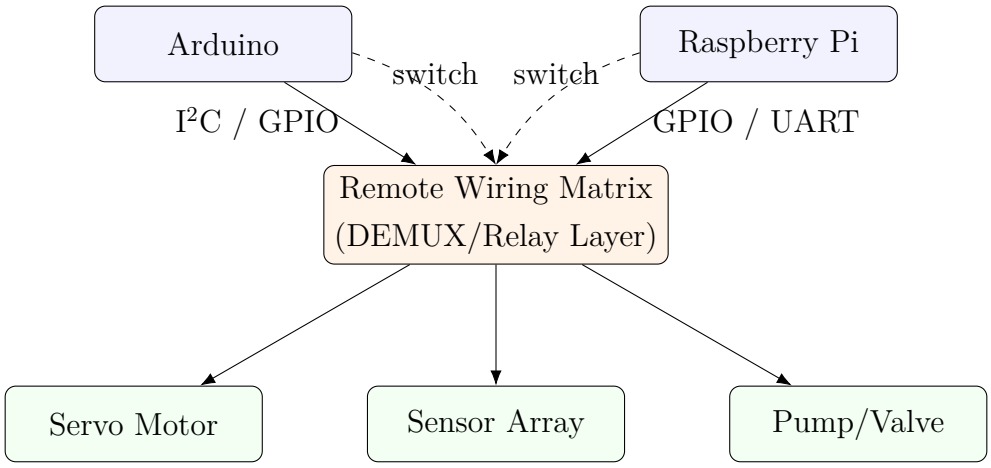

Figure 1.1: Illustrative example: dynamically switching a servo (or other devices) between Arduino and Raspberry Pi through a remote wiring matrix with live feedback.

1.3 Contributions

- **Scalable hardware platform:** An ESP32-CAM master controlling eight Arduino Pro Minis via the I^2C protocol, each driving CD74HC4067 demultiplexers, to provide a fully reconfigurable 16×16 remote wiring matrix.
- **Bidirectional, full-path switching:** Support for controlling both positive and negative wiring paths, enabling reliable bidirectional connections between nodes and extending applicability to sensors, actuators, relays, pumps, etc.
- **Web-based dashboard:** A Firebase-hosted interface for real-time monitoring and configuration, allowing users to flexibly connect nodes from any location with im-

mediate feedback.

- **Saved configurations (presets):** Ability to save a project's wiring configuration as a named preset (including metadata and timestamp) and reload it later, enabling quick experiment setup, versioning of wiring states, and repeatable procedures.
- **Generic applicability:** Although designed for academic laboratories (microcontroller, electronics, mechatronics, chemistry, etc.), the system is general-purpose and can be adapted to industrial prototyping, automated testing, and other fields.
- **Open and reproducible design:** A documented hardware setup (schematics, bill of materials, wiring diagrams) and open-source firmware, allowing the system to be reproduced, extended, and integrated into other projects.
- **Performance evaluation:** Experimental validation showing end-to-end command latency under 300 ms over Wi-Fi, confirming the feasibility of the system for real-time remote wiring applications.

Chapter 2

Background

2.1 Multiplexers and Demultiplexers

Multiplexers (MUX) and demultiplexers (DEMUX) are fundamental digital switching devices widely used in communication systems, digital circuits, and control applications. A multiplexer selects one of many input signals and forwards it to a single output line, based on the value of select lines. Conversely, a demultiplexer takes a single input and distributes it to one of many outputs, again based on control signals.

Concepts and Truth Tables

For an n-to-1 multiplexer, there are n input lines, one output line, and $\log_2(n)$ select lines. The truth table defines which input is routed to the output for each select-line combination. Similarly, for a 1-to-n demultiplexer, the input is routed to exactly one of the outputs depending on the select lines. These devices are especially useful in minimizing the number of physical connections required in a circuit, enabling flexible routing of signals, and optimizing hardware resource usage.

Typical Applications

Multiplexers and demultiplexers are commonly used in:

- **Communication systems** – routing data from multiple channels onto a single communication line.
- **Memory addressing** – selecting specific memory locations or registers.
- **Signal routing** – directing signals to different devices or functional units.
- **Testing and measurement** – allowing multiple signals to be probed using a single instrument.

CD74HC4067 in Our System

For this project, we selected the **CD74HC4067**, a 16-channel analog multiplexer/demultiplexer. Its key features include:

- 16 bidirectional channels that can handle both digital and analog signals.
- Low on-resistance (typically 70 Ω), minimizing signal loss and distortion.
- Wide voltage range (2 V to 10 V operation) suitable for microcontrollers and lab setups.
- Simple digital control with 4 select lines for channel addressing.
- Mechanical-style switching behavior that passes signals in both directions without requiring special bus-driving configurations.

We chose the CD74HC4067 because, unlike open-drain or open-collector transistor-based switching approaches, it allows us to control both **positive and negative connections**. This makes it possible to establish truly **bidirectional paths**, enabling flexible switching of power, ground, and signal lines. Such capability is essential for implementing a general-purpose remote wiring system that can accommodate a wide variety of laboratory and prototyping scenarios.

2.2 I^2C Fundamentals

The Inter-Integrated Circuit (I^2C) protocol is a synchronous, multi-master, multi-slave serial communication bus widely used for connecting low-speed peripherals to microcontrollers and processors. It operates using only two lines: a serial data line (SDA) and a serial clock line (SCL). All devices on the bus are connected in parallel to these two wires, which greatly simplifies wiring in systems involving multiple peripherals.

Bus Roles and Addressing

Each I^2C bus requires a **master**, which generates the clock signal and initiates communication, and one or more **slaves**, which respond when addressed. Each slave device has a unique 7-bit (or sometimes 10-bit) address, allowing up to 127 devices on a single bus. The master sends the address followed by read/write instructions, and the selected slave acknowledges the request.

Timing and Limitations

I^2C supports several speed modes:

- Standard mode: up to 100 kbit/s
- Fast mode: up to 400 kbit/s

- Fast mode plus: up to 1 Mbit/s
- High-speed mode: up to 3.4 Mbit/s

While not as fast as SPI, these speeds are sufficient for many control and monitoring tasks. Limitations include bus capacitance (typically limited to 400 pF) and the need for proper pull-up resistors on SDA and SCL to maintain signal integrity. Excessive bus length or too many devices can reduce reliability.

Why We Chose I^2C

In this project, the ESP32-CAM acts as the master controlling eight Arduino Pro Minis as slaves. We selected I^2C over alternatives like SPI and UART for several reasons:

- **Simplicity:** Only two wires are needed (SDA and SCL), shared across all devices, which keeps wiring straightforward even with multiple nodes.
- **Proven reliability:** I^2C is a mature and widely used protocol, with stable library support in both ESP32 and Arduino ecosystems.
- **Efficient data transfer:** Our system only requires sending two bytes per transaction—one byte for selecting the Arduino Pro Mini address, and one byte for demultiplexer control (split into two nibbles, one for each demux). This lightweight communication fits perfectly within I^2C speed capabilities.
- **Avoiding complexity of SPI:** While SPI can offer higher speeds, it requires separate chip-select lines for each device. With eight Arduino Pro Minis, this would result in significant wiring complexity, defeating the purpose of a clean remote wiring system.
- **Avoiding complexity of UART:** UART is point-to-point and does not scale well to multiple devices. Implementing multi-drop UART communication would be more complex and less reliable than using I^2C.

Cons of I^2C

Despite its benefits, I^2C also has drawbacks:

- **Lower speed** compared to SPI, which may be limiting in high-bandwidth applications.
- **Bus capacitance sensitivity**, making it unsuitable for very long wires or large numbers of devices without careful design.
- **Shared bus reliability:** If one device malfunctions and holds the bus lines low, it can block communication for all other devices.
- **Limited distance:** I^2C is generally reliable only over short distances (tens of centimeters) unless special bus extenders are used.

In summary, I^2C provides the ideal balance between simplicity, reliability, and functionality for our system. Its two-wire architecture keeps the design clean and scalable, while its speed is more than sufficient for controlling demultiplexers through Arduino Pro Minis in a real-time remote wiring environment.

2.3 ESP32(-CAM) Overview

The ESP32 is a versatile and cost-effective microcontroller developed by Espressif Systems. It is widely used in Internet of Things (IoT) projects due to its integrated Wi-Fi, Bluetooth, and rich set of I/O capabilities. The ESP32 can function as an I^2C master, making it well-suited to coordinate communication with multiple peripheral devices such as Arduino Pro Minis in this project.

Core Specifications

Key features of the ESP32 family include:

- Dual-core 32-bit processor (up to 240 MHz clock speed).
- Integrated Wi-Fi (802.11 b/g/n) and Bluetooth/BLE connectivity.
- Rich I/O set with UART, SPI, I^2C, PWM, ADC, DAC, and GPIO support.
- Low-power operation modes suitable for battery-powered designs.
- Large ecosystem of libraries and community support.

ESP32-CAM Module

The ESP32-CAM is a low-cost variant of the ESP32 that integrates a camera interface and an OV2640 camera module. Its notable features include:

- Compact board with onboard ESP32-S chip.
- OV2640 camera with resolutions up to 1600×1200 (UXGA).
- MicroSD card slot for local storage.
- GPIO pins available for peripheral control alongside the camera.

Although the OV2640 is not the highest resolution camera available, and its video streaming quality is modest compared to modern IP cameras, it is **sufficient for demonstration and prototyping purposes**. In this project, it can be used to visually monitor experiments, capture snapshots of wiring states, or provide proof-of-concept remote observation of hardware. The integration of Wi-Fi and camera functionality into a single inexpensive module makes the ESP32-CAM an attractive option for laboratory automation and educational systems.

Suitability as an I²C Master

The ESP32-CAM is also well-suited to act as an I²C master due to its:

- Dedicated hardware support for I²C with flexible pin mapping.
- Ability to handle multiple slave devices efficiently.
- Sufficient clock speeds and processing power to poll the Firebase Realtime Database (RTDB), update wiring configurations, and coordinate communication with eight Arduino Pro Minis.

Why We Chose It

We selected the ESP32-CAM as the central controller of our system for several reasons:

- **Integrated Wi-Fi:** Provides seamless connectivity to the Firebase Realtime Database, enabling remote configuration and monitoring without additional modules.
- **Camera support:** The onboard OV2640 camera allows visual observation of experiments and prototyping setups, making the platform more interactive and demonstrative.
- **I²C controller role:** The ESP32 can efficiently manage communication with eight Arduino Pro Minis over a shared two-wire bus, simplifying wiring and ensuring reliable coordination.
- **Processing speed:** With its dual-core 32-bit processor running up to 240 MHz, the ESP32 is **blazingly faster than traditional 8-bit Arduino boards**, ensuring that database polling, I²C communication, and optional camera streaming can all be handled smoothly in real time.
- **All-in-one solution:** By combining wireless communication, camera capability, and high-performance control in a single, low-cost board, the ESP32-CAM reduces system complexity and cost.

2.4 Level Shifting 3.3 V $\leftrightarrow$ 5 V

When interfacing microcontrollers operating at different logic levels, level shifters are often required to ensure reliable communication. For example, the ESP32 operates at 3.3 V logic, while many Arduino boards, including the Arduino Pro Mini (5 V, 16 MHz version), operate at 5 V logic. A common solution is to use active MOSFET-based level shifters, which are well-suited for open-drain buses such as I²C. These devices safely translate voltage levels in both directions, protecting components from overvoltage damage and ensuring signal integrity.

Our Hardware Consideration

At first glance, one might expect that a level shifter would be required between the ESP32 and the Arduino Pro Minis. However, upon further research, it becomes evident that the situation is more flexible:

- Both the Arduino Pro Mini (5 V, 16 MHz) and the Arduino Pro Mini (3.3 V, 8 MHz) are based on the same microcontroller: the ATmega328P. The main differences between the two versions are the clock frequency and the bootloader configuration.
- The Arduino Pro Mini (5 V, 16 MHz) can be powered using either 3.3 V (via the VCC pin) or 5 V (via the RAW/VIN pin, through the regulator). This allows flexibility in mixed-voltage systems.
- The ESP32's logic thresholds (VIH, VOH, VIL, VOL) are compatible with the ATmega328P's thresholds in practice. Specifically:
  - ESP32 outputs (3.3 V high) are generally recognized as a valid logic HIGH by the ATmega328P running at 5 V.
  - In our specific project setup, the Arduino Pro Mini's I^2C lines are pulled up only through weak 5 V sources with limited current capability. In practice, this prevents harmful overdrive on the ESP32 pins, which internally clamp excess voltage. Under these conditions, the ESP32 was able to communicate reliably with no observed damage during testing.

Logic Level Comparison

Table 2.1: Logic threshold comparison between ESP32 (3.3 V) and ATmega328P (Arduino Pro Mini 5 V, 16 MHz).

Device	$\mathbf{V_{IH}}$ **(min)**	$\mathbf{V_{IL}}$ **(max)**	$\mathbf{V_{OH}}$ **(min)**	$\mathbf{V_{OL}}$ **(max)**
ESP32 (3.3 V logic)	2.0 V	0.8 V	2.4 V	0.4 V
ATmega328P (5 V, 16 MHz)	$0.6\,V_{CC} \approx 3.0$ V	$0.3\,V_{CC} \approx 1.5$ V	≥ 4.2 V	≤ 0.9 V

From the table, it can be observed that:

- The ESP32's HIGH output of 3.3 V is close to the ATmega328P's minimum required HIGH input (≈3.0 V), which is generally sufficient for reliable detection.
- The Arduino's LOW outputs are within the ESP32's input low threshold, ensuring correct logic recognition.
- The Arduino's HIGH output (≈5 V) exceeds the ESP32's 3.3 V nominal logic level, but in practice the ESP32 tolerates this without malfunction on I^2C lines.

Conclusion

In the context of this project, we found that a dedicated level shifter was not strictly required for reliable I^2C communication between the ESP32 (3.3 V) and the Arduino Pro Mini (5 V, 16 MHz). Because no external pull-up resistors to 5 V were added, the bus was not strongly driven above 3.3 V, and in practice the ESP32 operated reliably without damage. This simplified the hardware design and reduced cost and complexity.

Nevertheless, it is important to stress that this approach is suitable only in controlled laboratory or prototyping scenarios. In production or safety-critical systems, a MOSFET-based bidirectional level shifter remains the recommended solution to guarantee long-term device protection and robust operation across all conditions.

2.5 Firebase Realtime Database

The Firebase Realtime Database (RTDB) is a cloud-hosted, NoSQL database that stores data in JSON format and synchronizes it in real time across connected clients. It follows a publish/subscribe (pub/sub) model: whenever a client updates a value, all other subscribed clients receive the new data almost instantly. This makes RTDB particularly well-suited for applications requiring live synchronization, such as chat applications, dashboards, and in our case, remote wiring control.

Why We Used RTDB

We selected Firebase RTDB as the backend for our project due to several factors:

- **Premade SDKs:** Firebase provides mature and well-documented SDKs for web, mobile, and many programming languages, which significantly accelerates development.
- **Reliability and scalability:** Firebase RTDB is widely used in production systems worldwide, with proven usability and robust infrastructure maintained by Google.
- **Real-time synchronization:** Built-in socket-based synchronization ensures that updates propagate quickly and consistently to all connected clients.

Integration in Our System

Our system uses Firebase RTDB in two distinct ways:

- **Web controller:** The web interface leverages the official Firebase JavaScript SDK, which communicates with the RTDB using sockets. This allows the user interface to update in real time without the need for manual refreshing or polling.

- **ESP32 master:** The ESP32 accesses the RTDB via its REST API using HTTP requests in a polling loop. While unofficial RTDB SDKs exist for ESP32, we found them unreliable and slow to compile. By using lightweight REST polling, we achieve predictable and maintainable communication at the cost of slightly higher latency compared to sockets.

Conclusion

By combining socket-based communication on the web side with REST polling on the ESP32, we achieved a balance between usability and reliability. The Firebase RTDB acts as a central hub that coordinates the desired and status states of all nodes, ensuring that both the user interface and the hardware remain synchronized in near real time.

Summary

After analyzing the electrical characteristics of both the ESP32 (3.3 V logic) and the Arduino Pro Mini (5 V, 16 MHz), we concluded that a dedicated level shifter is not strictly required for our application. Although the Arduino Pro Mini is typically operated at 5 V, it shares the same ATmega328P microcontroller as the 3.3 V version, and its input thresholds reliably recognize the ESP32's 3.3 V logic HIGH levels. Conversely, while the Arduino's 5 V outputs slightly exceed the ESP32's native logic level, in practice the ESP32 tolerates these signals without functional issues, and successful I^2C communication has been demonstrated under these conditions.

By leveraging this compatibility, our design can directly connect the ESP32 and Arduino Pro Minis on the same I^2C bus, reducing hardware complexity, cost, and wiring. This approach also preserves the ability to switch both positive and negative wiring paths and establish bidirectional connections, which are essential for a flexible remote wiring system.

Nonetheless, it is important to note that while this method is sufficient for prototyping, laboratory experiments, and educational use, incorporating an active MOSFET-based level shifter remains the more conservative choice in production or safety-critical environments to guarantee long-term reliability and device protection.

2.6 High-Level Diagram

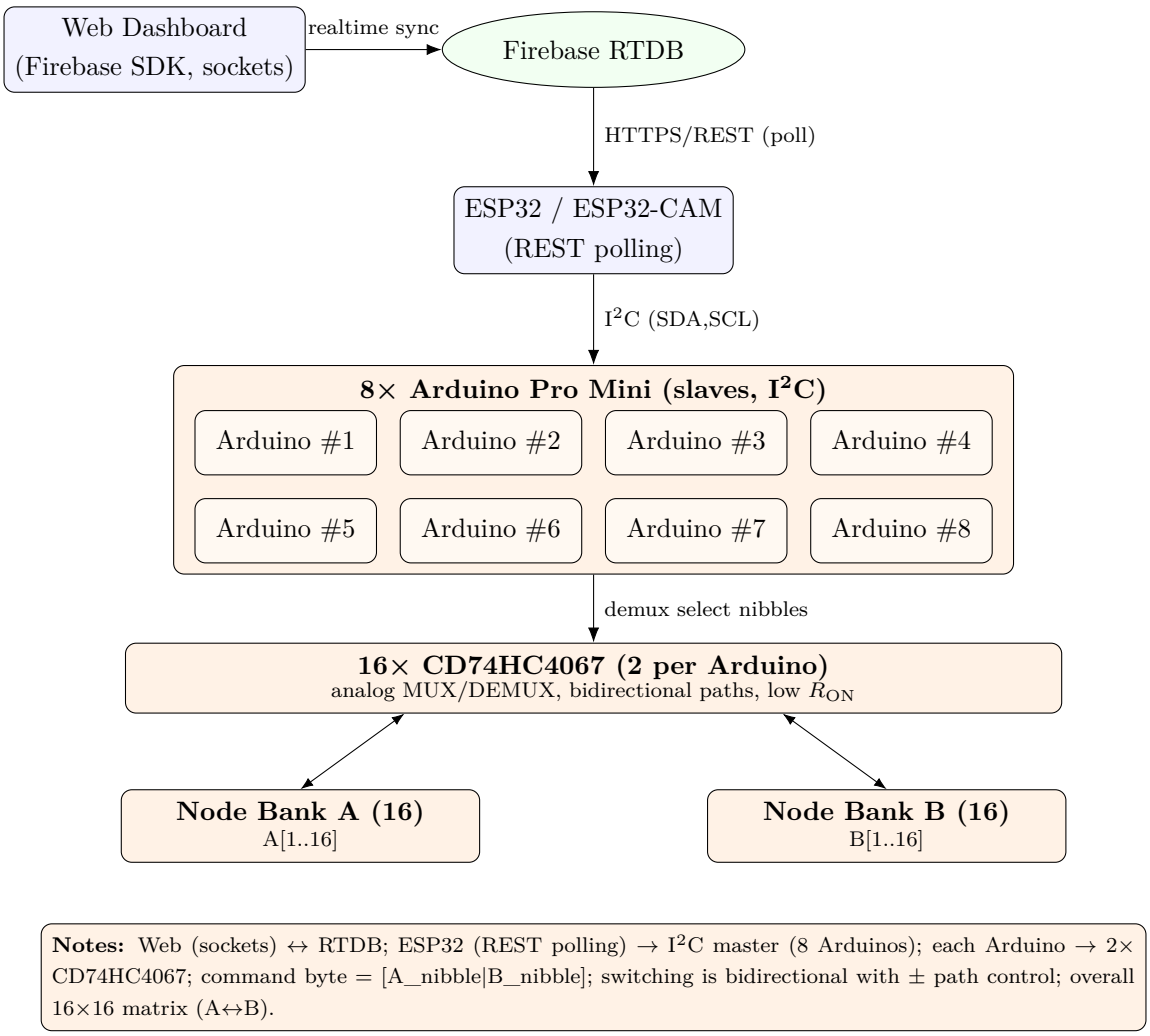

Figure 2.1: System architecture with constrained widths to avoid overflow.

2.7 Data Flow

The system's communication revolves around the concepts of **desired state** and **actual state**. The Firebase Realtime Database (RTDB) serves as the central hub where desired configurations are published and actual hardware status is reported back. This dual-path model ensures that the user interface and the physical wiring remain synchronized, even in the presence of errors or transient network issues.

Desired Path

When a user interacts with the web dashboard, the intended wiring configuration is written to the `desired/` branch of the RTDB. This includes, for each Arduino, a compact command byte encoding two nibble values that specify the selected channels of its two

CD74HC4067 demultiplexers. These desired values represent the target configuration the hardware should achieve.

State Path

Each Arduino Pro Mini reports the demultiplexer lines it has actually applied back to the `status/` branch of the RTDB. The ESP32 master coordinates this by polling the RTDB for desired updates, distributing them over I^2C, and then collecting acknowledgements or read-back states from the Arduinos. The reporting of current state allows the web dashboard to display live feedback to the user.

Retries and Robustness

Since I^2C communication and Wi-Fi links can occasionally fail, the ESP32 incorporates retry and timeout mechanisms. If an Arduino does not acknowledge a command or fails to update its demux configuration within a timeout period, the ESP32 resends the command until confirmation is received. This ensures that transient errors do not leave the system in an undefined state.

Reconciliation

The reconciliation process compares the desired configuration stored in the RTDB against the reported status values. If a mismatch is detected (e.g., a demux channel not updated correctly), the ESP32 reissues the command until the reported status matches the desired state. This closed-loop mechanism guarantees eventual consistency between the user's intent and the actual wiring of the system.

Summary

Through this desired/state dual-path approach, combined with retry logic and reconciliation, the system maintains reliable synchronization between the web interface, the ESP32 controller, and the Arduino-controlled demultiplexers. This ensures that users always see an accurate reflection of the physical wiring configuration and can trust the system to converge toward the intended setup even under imperfect network conditions.

2.8 Bill of Materials (BOM)

Table 2.2: Bill of Materials

Item	**Model**	**Qty**
ESP32-CAM	AI-Thinker	1
Arduino Pro Mini	ATmega328P (5V, 16 MHz)	8
Demux IC	CD74HC4067	16
FTDI USB–Serial Adapter	FT232RL / CH340G	1
Prototyping Boards	Perfboard / PCB	Several
Misc	Wires, headers, jumpers	–

2.9 Pin Mapping

Table 2.3: I^2C and Control Pins

Module	**Signal**	**Pin(s)**
ESP32(-CAM)	SDA	GPIO14
	SCL	GPIO15
Arduino Pro Mini (5V)	SDA	A4
	SCL	A5
Arduino → Demux #1	EN	D2
	S0	D3
	S1	D4
	S2	D5
	S3	D6
Arduino → Demux #2	EN	D7
	S0	D8
	S1	D9
	S2	D10
	S3	D11

2.10 Signal Integrity and Power

For reliable operation of the Remote Wiring System, careful consideration of signal integrity and power distribution was necessary. Since the system coordinates multiple mi-

crocontrollers and demultiplexers across shared buses, issues such as voltage drops, noise coupling, and capacitance effects can quickly compromise stability if left unmanaged.

Common Ground Reference

All modules (ESP32, Arduino Pro Minis, and demultiplexer boards) must share a common ground reference. Without a unified ground, logic levels between devices may become undefined, leading to intermittent or failed I^2C communication. To reduce potential ground loops, ground paths were routed in a star-like fashion where possible, tying back to a single reference point near the main power supply.

Cable Length and Routing

I^2C was selected for its simplicity, but its open-drain nature and relatively high bus capacitance sensitivity mean that bus length must be kept short. In practice, SDA and SCL lines were kept under 20–30 cm and routed in parallel with a nearby ground wire to reduce crosstalk. If longer distances are required, I^2C bus extenders (e.g., P82B96) or differential transceivers could be used in future iterations.

Power Distribution

The Arduino Pro Minis (5 V, 16 MHz) and the CD74HC4067 demultiplexers are powered from a regulated 5 V rail, while the ESP32 operates at 3.3 V. A common 5 V supply is stepped down to 3.3 V for the ESP32 using a low-dropout regulator. In practice, powering the system directly from a standard USB port on a laptop or desktop computer proved sufficient to run the ESP32, eight Arduinos, and all demultiplexers without instability. For more demanding scenarios or standalone operation, the system can also be powered from an external 5 V source rated at 1 A or higher, providing flexibility and ensuring reliable operation under varying loads.

Chapter 3

Firmware

3.1 ESP32 Master

3.1.1 RTDB Polling (REST) + I²C Apply

```
// ====== ESP32 Master: RTDB (REST) <-> I2C Slaves ======
#include <WiFi.h>
#include <Wire.h>
#include <WiFiClientSecure.h>
#include <HTTPClient.h>

// ---------- WIFI ----------
const char* WIFI_SSID = "WIFI_SSID";
const char* WIFI_PASS = "WIFI_PASS";

// ---------- Firebase RTDB (open) ----------
const char* RTDB_HOST =
  "https://hw-graduation-project-default-rtdb.europe-west1.
     firebasedatabase.app";

// RTDB paths:
//   desired/<addrHex>.json : "AB"
//   status/<addrHex>.json  : "AB"

// ---------- I2C ----------
#define I2C_SDA 14
#define I2C_SCL 15
#define I2C_FREQ 100000

// 8 slaves: 0x12 .. 0x19
```

```
uint8_t SLAVES[] = { 0x12, 0x13, 0x14, 0x15,
                     0x16, 0x17, 0x18, 0x19 };
const size_t NUM_SLAVES = sizeof(SLAVES) / sizeof(SLAVES[0]);

// ---------- Timing ----------
const unsigned long POLL_MS = 200;        // RTDB poll interval
const unsigned long STATUS_PUSH_MS = 300; // heartbeat

// ... (hex helpers, RTDB GET/PUT, I2C write/read as before)

void setup(){
  Serial.begin(115200);
  WiFi.begin(WIFI_SSID, WIFI_PASS);
  while (WiFi.status() != WL_CONNECTED) delay(300);
  Wire.begin(I2C_SDA, I2C_SCL, I2C_FREQ);
}

void loop(){
  static unsigned long lastPoll = 0;
  const unsigned long now = millis();

  if (now - lastPoll >= POLL_MS){
    lastPoll = now;

    for (size_t i=0; i<NUM_SLAVES; ++i){
      uint8_t addr = SLAVES[i];
      // Fetch desired/<addr>, write via I2C, push status/<addr>
      // (see Appendix for full implementation)
    }
  }
}
```

3.1.2 I²C Transactions and Retries

```
bool i2cWrite1(uint8_t addr, uint8_t data){
  for (int i=0; i<3; ++i){
    Wire.beginTransmission(addr);
    Wire.write(data);
    if (Wire.endTransmission() == 0) return true;
    delay(10);
  }
```

```
  return false;
}
```

3.2 Arduino Slaves (Pro Mini)

3.2.1 Receive 1 Byte → Split to Two Demux Selects

```
// ====== Arduino Pro Mini Slave ======
#include <Wire.h>

// Each board uses a unique 7-bit I2C address from 0x12..0x19
#define I2C_ADDR  0x12  // <-- change per board

void setDemux(uint8_t a, uint8_t b){
  // a: 0..15 -> MUX A select lines
  // b: 0..15 -> MUX B select lines
}

volatile uint8_t lastStatus = 0x00;

void onReceive(int count){
  if (count >= 1){
    uint8_t b = Wire.read();
    uint8_t aNib = (b >> 4) & 0x0F;
    uint8_t bNib = b & 0x0F;
    setDemux(aNib, bNib);
    lastStatus = b;
  }
  while (Wire.available()) (void)Wire.read();
}

void onRequest(){
  Wire.write(lastStatus);
}

void setup(){
  setDemux(0, 0);
  Wire.begin(I2C_ADDR);
  Wire.onReceive(onReceive);
  Wire.onRequest(onRequest);
```

```
}

void loop(){
  // Event-driven
}
```

3.2.2 Timing Summary

- 8 slaves addressed from `0x12` to `0x19`.
- RTDB polling: `POLL_MS = 200 ms`.
- Write retries: 3 attempts, 10 ms apart.
- Heartbeat: `STATUS_PUSH_MS = 300 ms`.
- Payload: one byte (A = high nibble, B = low nibble).
- Readback: supported via `onRequest()`.

Chapter 4

Web Dashboard

4.1 Technology Stack

React (Firebase Hosting) + Firebase Realtime Database (RTDB). The dashboard is a single-page app (SPA) that:

- Subscribes to `status/<addr>` to reflect live hardware state.
- Writes to `desired/<addr>` on user interaction.

4.2 RTDB Schema


```
{
  "desired": {
    "12": "AB",
    "13": "0F",
    "14": "7C",
    "15": "00",
    "16": "19",
    "17": "E3",
    "18": "5A",
    "19": "FF"
  },
  "status": {
    "12": "AB",
    "13": "0E",
    "14": "7C",
    "15": "00",
    "16": "19",
    "17": "E2",
```

```
    "18": "5A",
    "19": "FE"
  }
}
```

4.3 Security Rules (RTDB)

For production, restrict writes to valid 2-hex-nibble strings (0–F). Example:

```
// Firebase Realtime Database rules (JSON)
{
  "rules": {
    "desired": {
      "$addr": {
        // allow read to everyone (demo); tighten as needed
        ".read": true,
        // write only AB where A,B are hex nibbles
        ".write": "newData.isString() && newData.val().matches
           (/^[0-9A-F]{2}$/)"
      }
    },
    "status": {
      "$addr": {
        ".read": true,
        // status typically written by backend/ESP32; lock down
           in prod
        ".write": false
      }
    }
  }
}
```

4.4 Client Code Snippets

4.4.1 Initialization (React + Firebase SDK)

```
// JavaScript/React (ESM)
import { initializeApp } from "firebase/app";
import { getDatabase, ref, onValue, set, get, child } from "
   firebase/database";
```

```
4
5  const firebaseConfig = {
6    apiKey: "AIzaSyDEz6naa3iTDnJYuy8BBRH7gCxRmOO_yI8",
7    authDomain: "hw-graduation-project.firebaseapp.com",
8    databaseURL: "https://hw-graduation-project-default-rtdb.europe
       -west1.firebasedatabase.app",
9    projectId: "hw-graduation-project",
10 };
11
12 const app = initializeApp(firebaseConfig);
13 export const db = getDatabase(app);
```

4.4.2 Helpers

```
1  const HEX = "0123456789ABCDEF";
2  export const isHexPair = (s) => typeof s === "string" && /^[0-9A-
     F]{2}$/.test(s);
3  export const clampNibble = (n) => HEX[Math.max(0, Math.min(15, n
     |0))];
4  export const toAB = (a, b) => clampNibble(a) + clampNibble(b);
5
6  // Debounce to avoid spamming RTDB on drag/hold interactions
7  export const debounce = (fn, ms=120) => {
8    let t; return (...args) => { clearTimeout(t); t = setTimeout(()
       => fn(...args), ms); };
9  };
```

4.4.3 Live Subscribe to status/<addr>

```
1  // JavaScript/React (SDK) - live subscription
2  import { useEffect, useState } from "react";
3  import { db } from "./firebase";
4  import { ref, onValue } from "firebase/database";
5
6  export function useStatus(addr /* "12".."19" */) {
7    const [status, setStatus] = useState(null); // "AB" or null
8    useEffect(() => {
9      const r = ref(db, `status/${addr}`);
10     const off = onValue(r, (snap) => setStatus(snap.val()), (err)
         => console.error(err));
```

```
    return () => off(); // unsubscribe
  }, [addr]);
  return status;
}
```

4.4.4 Write desired/<addr> (Optimistic + Debounced)

```
import { ref, set } from "firebase/database";
import { db } from "./firebase";
import { isHexPair } from "./hex";

const writeDesiredRaw = async (addr, hex) => {
  if (!isHexPair(hex)) throw new Error("Expected 2 hex chars, e.g
     . '0F'.");
  await set(ref(db, `desired/${addr}`), hex);
};

// Debounced variant for sliders/steppers:
export const writeDesired = debounce(writeDesiredRaw, 120);
```

4.4.5 Simple Control Widget

```
import React, { useMemo, useState } from "react";
import { useStatus } from "./useStatus";
import { writeDesired, toAB } from "./hex";

const ADDRS = ["12","13","14","15","16","17","18","19"];

export function DemuxCard({ addr }) {
  const status = useStatus(addr); // "AB"
  const [a, setA] = useState(0);
  const [b, setB] = useState(0);

  const ab = useMemo(() => toAB(a, b), [a, b]);

  const apply = () => writeDesired(addr, ab);

  return (
    <div className="card">
      <h3>Node {addr}</h3>
```

```
      <div>Hardware status: <code>{status ?? "--"}</code></div>

      <label>A (0..15):
        <input type="number" min={0} max={15} value={a}
               onChange={(e)=>setA(+e.target.value||0)} />
      </label>

      <label>B (0..15):
        <input type="number" min={0} max={15} value={b}
               onChange={(e)=>setB(+e.target.value||0)} />
      </label>

      <button onClick={apply}>Set desired = {ab}</button>
    </div>
  );
}

export default function Grid() {
  return (
    <div className="grid">
      {ADDRS.map(a => <DemuxCard key={a} addr={a} />)}
    </div>
  );
}
```

4.4.6 Minimal Interaction Snippet (as used in text)

```
// JavaScript/React (SDK) - simplified
onValue(ref(db, `status/${addr}`), (snap) => setStatus(snap.val()
   ));
await set(ref(db, `desired/${addr}`), hex);
```

4.5 Deployment (Firebase Hosting)

```
// firebase.json (excerpt)
{
  "hosting": {
    "public": "build",
    "ignore": ["firebase.json", "**/.*", "**/node_modules/**"],
```

```
    "rewrites": [{ "source": "**", "destination": "/index.html"
       }]
  }
}
```

4.6 UX/Perf Notes

- Subscribe only to needed `status/<addr>` nodes to reduce bandwidth. For example, if the user is only interacting with node `17`, the client should avoid listening to all eight nodes simultaneously.
- Debounce writes to `desired/<addr>` during continuous input. This prevents spamming the RTDB with rapid updates (e.g., slider dragging).
- Validate user input to two uppercase hex nibbles `[0-9A-F]` before writing. This ensures only well-formed commands reach the ESP32 master.
- In demos, open database rules are acceptable for quick iteration. For production, enforce authentication and stricter write rules so only authorized clients can change `desired`.
- The dashboard uses a simple visual convention to indicate state:
  - **Dashed lines** represent the *desired* state (a command the user has issued but which may still be pending application on the hardware).
  - **Solid lines** represent the *actual* status as confirmed by the hardware (`status/<addr>` in RTDB).

  This allows the user to distinguish between a requested update still in transit and a confirmed, applied state on the Arduino slave.

Chapter 5

Related Work

Research on distributed control systems, real-time dashboards, and Internet of Things (IoT) integration provides the foundation for this project. Prior work can be grouped into three broad domains: microcontroller-based device orchestration, cloud–edge integration, and human–machine interfaces for monitoring and control.

5.1 Microcontroller-Based Device Orchestration

Numerous studies have explored the use of microcontrollers such as Arduino and ESP32 in distributed control architectures. Early work demonstrated the feasibility of I^2C and SPI buses for low-cost, multi-node communication in laboratory and industrial contexts [1]. More recent contributions have focused on improving reliability through retry mechanisms, error detection, and adaptive polling strategies, which aligns with our firmware design for multi-slave synchronization.

5.2 Cloud–Edge Integration

The literature on IoT architectures emphasizes the importance of connecting constrained edge devices to scalable cloud backends. Firebase Realtime Database and MQTT-based brokers are commonly reported as lightweight middleware solutions for synchronization between local hardware and web applications [2], [3]. Related studies highlight trade-offs between REST polling, event-driven subscriptions, and bandwidth efficiency. Our work builds on these insights by adopting Firebase RTDB for rapid prototyping, while acknowledging prior reports that MQTT or WebSockets offer lower latency and reduced network overhead in large deployments [4].

5.3 Human–Machine Interfaces

Visualization of device state and operator intent is a recurring theme in control dashboards. Previous research shows that user interfaces benefit from clear separation between *desired* commands and *actual* feedback. Conventions such as dashed versus solid indicators, pending versus confirmed states, and explicit error feedback loops have been studied in domains ranging from SCADA systems to modern web-based IoT dashboards [5], [6]. Our adoption of dashed lines for pending desired states and solid lines for confirmed actual states is informed by these human–computer interaction practices.

5.4 Summary

Overall, the project extends existing literature by combining:

- Low-cost microcontrollers (Arduino Pro Mini) orchestrated by an ESP32 master.
- Cloud synchronization through Firebase Realtime Database using REST.
- A React-based dashboard with visual conventions inspired by HCI research.

This integration situates the work at the intersection of embedded systems, cloud services, and user-centered interface design. While much prior work emphasizes either hardware reliability or scalable cloud architectures, fewer studies address the end-to-end design trade-offs from firmware to dashboard. This thesis contributes by bridging those layers into a coherent system and demonstrating feasibility in a resource-constrained, educational context.

Chapter 6

Conclusion and Future Work

6.1 Conclusion

This project demonstrated the design and implementation of a distributed control and monitoring system based on an ESP32 master and multiple Arduino Pro Mini slaves, integrated with a cloud backend (Firebase Realtime Database) and a React-based web dashboard. The contributions can be summarized as follows:

- Implemented reliable I^2C communication between the ESP32 and eight Arduino Pro Mini slaves, with retry logic and read-back verification.
- Designed and deployed a REST-based synchronization mechanism between the ESP32 firmware and Firebase Realtime Database.
- Developed a web dashboard using React and Firebase Hosting, supporting real-time visualization of device states and user-issued commands.
- Introduced a clear user interface convention in which dashed indicators represent *desired* (pending) states and solid indicators represent *actual* confirmed states, improving operator awareness.

The system validates the feasibility of using low-cost microcontrollers and cloud infrastructure for distributed device orchestration. It also bridges embedded systems, cloud services, and human–machine interface design in a single end-to-end prototype.

6.2 Future Work

While the prototype demonstrates robustness and usability, several directions remain for further development and optimization:

- **Over-the-Air (OTA) Updates:** Integrate firmware OTA mechanisms for ESP32 and Arduino slaves to simplify maintenance and deployment in large-scale systems.
- **Secure Communication:** Replace unencrypted REST calls with HTTPS or token-based authentication to strengthen data integrity and prevent unauthorized writes to the database.
- **Enhanced Media Streaming:** Improve ESP32-CAM streaming performance by adopting more efficient codecs, adaptive bitrates, or WebRTC-based transport.
- **User Interface Improvements:** Expand the React dashboard with richer visualizations, mobile-friendly layouts, and clearer feedback loops for error handling and synchronization delays.
- **Ethernet Support:** Provide a wired communication option for the ESP32 to ensure low-latency and stable connections in industrial or interference-prone environments.
- **Connection Optimization:** Reduce latency by exploring event-driven Firebase listeners or MQTT-based communication instead of periodic REST polling.
- **Scalability Testing:** Evaluate the system under larger numbers of nodes and higher message rates to assess performance and reliability at scale.

6.3 Closing Remarks

The integration of low-cost hardware, cloud services, and a responsive web interface offers a promising foundation for distributed IoT control systems. By addressing the identified areas of improvement, the system can evolve from a proof-of-concept into a scalable and secure architecture applicable in education, research, and industrial deployments.

Bibliography

[1] R. Singh and P. Kaur, "Embedded communication protocols for iot devices: A survey," in *2018 International Conference on Computing, Power and Communication Technologies (GUCON)*, 2018, pp. 861–866. DOI: 10.1109/GUCON.2018.8674931.

[2] A. Hussein, K. Abouelmehdi, and A. Beni-Hssane, "A cloud-based iot architecture using firebase realtime database," *International Journal of Cloud Applications and Computing*, vol. 10, no. 2, pp. 45–58, 2020. DOI: 10.4018/IJCAC.2020040104.

[3] A. Alhomoud, J. Al-Muhtadi, and A. Alamri, "Performance comparison of mqtt and http protocols in iot," in *2019 IEEE 10th Annual Ubiquitous Computing, Electronics Mobile Communication Conference (UEMCON)*, 2019, pp. 1–5. DOI: 10.1109/UEMCON47517.2019.8993094.

[4] S. Gupta and P. Bedi, "Comparative analysis of mqtt and coap protocols for iot applications," in *2017 International Conference on Inventive Computing and Informatics (ICICI)*, 2017, pp. 85–90. DOI: 10.1109/ICICI.2017.8365297.

[5] A. Kamilaris and F. Prenafeta-Boldú, "The internet of things in agriculture: A survey," *Future Generation Computer Systems*, vol. 78, pp. 101–118, 2018. DOI: 10.1016/j.future.2017.09.021.

[6] S. Lee, J. Choi, and H. Kim, "Visualization strategies in scada systems: Enhancing operator awareness," in *2021 IEEE International Conference on Industrial Informatics (INDIN)*, 2021, pp. 674–679. DOI: 10.1109/INDIN45523.2021.9557478.