Project Report

On

CANDrive Smart Dashboard

Submitted
In partial fulfilment
For the award of the Degree of

PG-Diploma in Embedded Systems and Design (PG-DESD)

C-DAC, ACTS (Pune)

Guided By:
Mr. Arafat Khan

Submitted By:
Ajay Singh 240840130002
Astitva Srivastava 240840130006
Hansraj Sahani 240840130009
Rohan Jagtap 240840130011
Nivya Ashokkumar 240840130023

Centre for Development of Advanced Computing(C-DAC), ACTS

(Pune- 411008)

ABSTRACT

The CANDrive Smart Dashboard is an advanced IoT-based vehicle monitoring system designed for real-time data acquisition and visualization. It integrates FreeRTOS for efficient task management and leverages ThingsBoard and MQTT for cloud-based analytics. The system consists of two STM32 Discovery boards communicating via CAN bus, with an ESP32 handling data transmission to the cloud. Key sensors include:

- TMP103 for motor temperature monitoring
- IR sensor for wheel revolutions, speed, and distance calculation
- Reed switch for door status detection

Using FreeRTOS, tasks such as sensor data acquisition, CAN communication, and UART processing run concurrently, ensuring reliable and efficient operation. The first STM32 collects sensor data and transmits it over CAN to the second STM32, which processes and relays it to the ESP32 via UART. The ESP32 formats the data into JSON and transmits it over MQTT to ThingsBoard for real-time dashboard visualization. This system enhances vehicle monitoring precision, enables predictive maintenance, and improves overall operational efficiency.

.

Table of Contents

Chapter 1
Introduction

1.1 Introduction

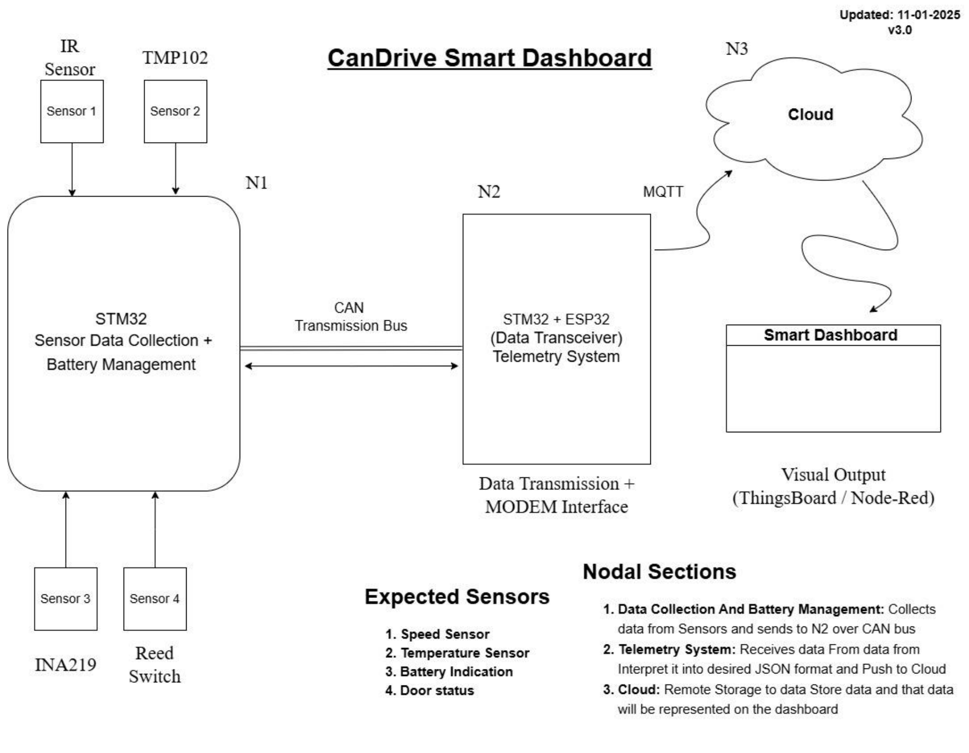
The automotive industry has witnessed a significant transformation over the decades, driven by advancements in embedded systems, communication technologies, and the Internet of Things (IoT). Early vehicle monitoring systems relied on basic onboard diagnostics (OBD), which provided limited fault detection through simple error codes. As vehicles became more sophisticated, the need for real-time monitoring of critical parameters such as engine temperature, fuel consumption, and vehicle speed became evident. This led to the integration of Controller Area Network (CAN bus) technology, which allowed different electronic control units (ECUs) within a vehicle to communicate efficiently.

With the rise of IoT and cloud computing, modern vehicle monitoring systems have evolved to offer remote diagnostics, predictive maintenance, and advanced data analytics. These advancements have not only improved vehicle safety and efficiency but have also become crucial in industries such as fleet management, logistics, and autonomous driving. Recognizing the growing demand for real-time vehicle monitoring, the CANDrive Smart Dashboard was developed as a comprehensive solution that leverages CAN bus communication, FreeRTOS, MQTT, and Thingsboard to provide live data acquisition and visualization.

CANDrive Smart Dashboard is an advanced IoT-enabled vehicle monitoring system designed to collect, process, and visualize critical vehicle parameters in real time. It combines embedded systems, cloud integration, and wireless communication to provide detailed insights into vehicle performance, operational efficiency, and maintenance requirements. By utilizing STM32 microcontrollers, an ESP32 module, and various sensors, the system ensures accurate and reliable data collection. The collected data is processed using FreeRTOS, transmitted via CAN bus, and displayed on a cloud-based Thingsboard dashboard using MQTT communication.

This system is particularly beneficial for automobile manufacturers, fleet operators, logistics companies, and vehicle owners who require real-time insights into vehicle health and performance. With its ability to provide continuous monitoring and predictive analytics, the CANDrive Smart Dashboard helps reduce unexpected breakdowns, improve fuel efficiency, and enhance overall safety.

In our project, we use a combination of advanced technologies to provide real-time vehicle monitoring and data visualization. The system is built around STM32 microcontrollers, which handle sensor data acquisition and communication over the CAN bus protocol for reliable and efficient data transfer between components. The ESP32 microcontroller acts as a communication gateway, transmitting the collected data to the cloud via MQTT using Wi-Fi. The project leverages FreeRTOS, a real-time operating system that enables efficient multitasking, allowing the system to handle concurrent tasks such as sensor data collection, CAN communication, and cloud transmission. Key sensors, including the TMP103 temperature sensor, IR sensor and Reed switch, monitor essential vehicle parameters such as motor temperature, wheel revolutions and door status. This data is then visualized in real-time on the Thingsboard cloud platform, offering an intuitive dashboard for continuous monitoring of vehicle health and performance.



1.2 Objective

The objectives of the project work are as -

- Real-Time Data Collection: Uses STM32 microcontrollers and CAN bus for efficient data acquisition and communication.
- Wireless Data Transmission: The ESP32 sends sensor data to the cloud via MQTT over Wi-Fi, ensuring remote accessibility.
- Efficient Task Management: Powered by FreeRTOS, enabling smooth multitasking and optimized system performance.
- Comprehensive Sensor Integration: Includes TMP103 for temperature, IR sensor for speed, Reed switch for door status.
- Cloud-Based Monitoring: Data is visualized on Thingsboard, allowing real-time insights for predictive maintenance and performance tracking.

Chapter 2

LITERATURE REVIEW

[1] Research in IoT-based automotive systems has demonstrated substantial advancements in vehicle monitoring and diagnostics. According to Lee et al. (2019), real-time data acquisition through IoT enhances vehicle safety and operational efficiency by enabling predictive maintenance and early fault detection. Additionally, cloud computing plays a crucial role in expanding remote monitoring capabilities, making vehicle diagnostics more accessible and efficient.
Our project builds on this foundation by integrating **ThingsBoard** as a cloud-based visualization platform, allowing real-time monitoring of key vehicle parameters and ensuring proactive diagnostics.

[2] Several studies highlight the significance of sensor-based vehicle monitoring. Choi et al. (2020) explored how **IR sensors** measure speed, **temperature sensors** track motor health, and **door sensors** enhance vehicle security.
In our system, we employ:

- **IR Sensor**: Calculates vehicle speed and distance by detecting wheel revolutions.
- **TMP102 Temperature Sensor**: Monitors motor temperature to prevent overheating.
- **Reed Switch**: Detects door status (open/closed), improving vehicle security. These sensors enable real-time monitoring of critical parameters, ensuring enhanced diagnostics and safety.

[3] Research by Srinivasan et al. (2018) underscores the role of real-time embedded systems in vehicle monitoring applications. STM32 microcontrollers are widely adopted due to their **low power consumption**, **high-speed processing**, and **robust communication capabilities**.
Our project incorporates:

- **Two STM32 boards**: One for speed and distance calculations, the other for temperature and door status monitoring.
- **CAN Protocol**: Facilitates STM32-to-STM32 communication, ensuring efficient data exchange between controllers.
  By leveraging **CAN bus technology**, the system provides **interference-free, reliable, and high-speed** communication between microcontrollers, a key requirement for real-time embedded systems.

[4] Studies on data communication in IoT systems emphasize the importance of **efficient and lightweight protocols**. Research by Zhang et al. (2021) discusses how the **UART protocol** is commonly used for short-distance microcontroller communication, whereas **MQTT** is preferred for cloud-based IoT communication due to its **low bandwidth** and **high reliability**.
Our system employs a hybrid communication strategy:

- **UART Protocol**: Transfers data from STM32 to ESP32.
- **MQTT Protocol**: Ensures low-latency, real-time data transmission from ESP32 to **ThingsBoard**.
  This architecture provides seamless **real-time data transfer** with minimal overhead.

[5] Additional research by Gupta et al. (2022) highlights the significance of **power monitoring** in IoT-based embedded systems. Power monitoring solutions, such as **INA219**, enable real-time current and voltage measurements to optimize power management and ensure uninterrupted system performance. Our future enhancements include integrating **INA219** to enhance power efficiency and improve system reliability.

[6] In the context of real-time IoT dashboards, research by Kumar et al. (2023) explores **cloud-based data visualization tools**, emphasizing ThingsBoard's capability to handle high-frequency sensor data and provide **customizable analytics dashboards** for improved decision-making. This aligns with our project's goal of providing an intuitive dashboard for vehicle monitoring.

Chapter 3
Methodology and Techniques

3.1 Approach and Methodology:

The development of the CANDrive Smart Dashboard follows a systematic approach that integrates IoT, embedded systems, cloud computing, and real-time data processing to achieve efficient vehicle monitoring. The methodology involves multiple stages, from hardware selection to data visualization and system testing.

3.1.1 System Architecture & Design

The project follows a modular approach, where different components are responsible for specific tasks:

- Motor & IR Sensor: Used to calculate vehicle speed and distance.
- TMP102 Temperature Sensor & Reed Switch: Measure motor temperature and door status (open/closed).
- Dual STM32 Board Setup:
  - First STM32: Calculates speed and distance.
  - Second STM32: Processes temperature and door status, then transmits data to the node1 STM32 using CAN protocol.
- ESP32 Microcontroller:
  - Receives accumulated data from the node1 STM32 via UART protocol.
  - Sends data to the Thingsboard cloud via MQTT over Wi-Fi for real-time monitoring.

This modular architecture allows for efficient communication, scalability, and ease of future upgrades.

Hardware Implementation

The hardware components are carefully selected for their efficiency and compatibility:

- IR Sensor to detect motor revolutions and calculate speed & distance.
- TMP102 Temperature Sensor to monitor motor temperature.

- Reed Switch to check door status (open/closed).
- Two STM32 Microcontrollers for processing and CAN-based communication.
- ESP32 for wireless data transmission to the cloud.

Software Development & Implementation

The software development process involves:

- Embedded C programming for STM32 firmware.
- FreeRTOS for Real-Time Task Scheduling, ensuring efficient multitasking in ESP32.
- UART Protocol for STM32 to ESP32 communication.
- CAN Protocol for STM32 to STM32 data transfer.
- MQTT Protocol for wireless data transmission from ESP32 to Thingsboard Cloud.
- Dashboard Development on Thingsboard for real-time data visualization.

Data Communication & Cloud Integration

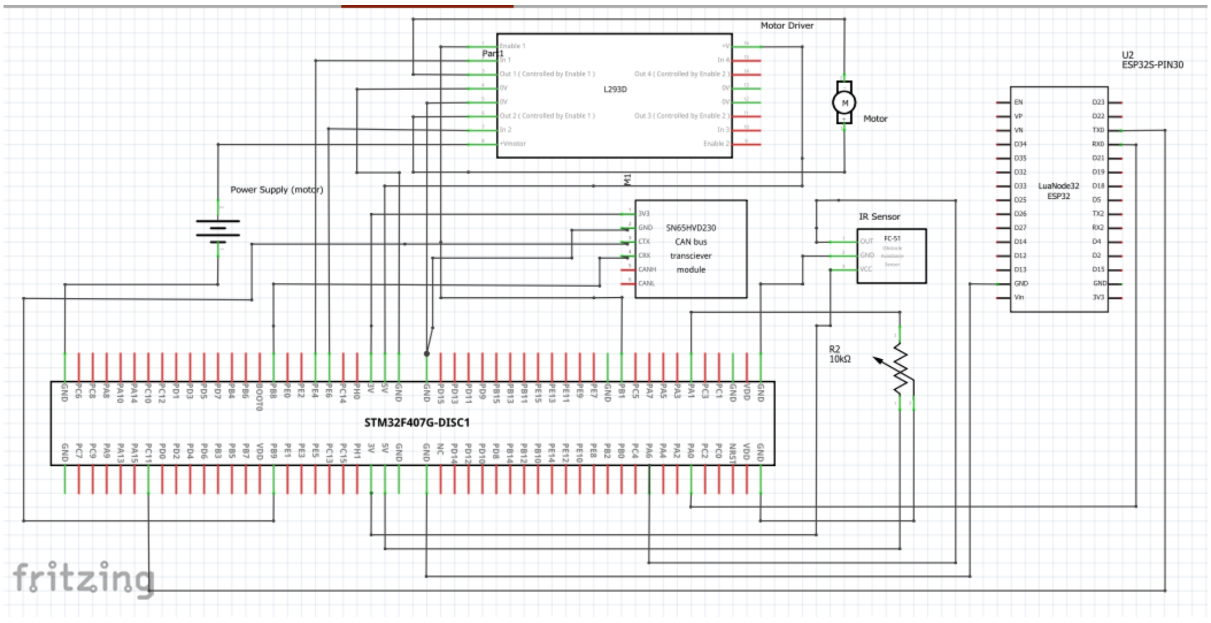
- The node2 STM32 board transmits processed temperature and door sensor data to the node1 STM32 via CAN protocol.
- The node1 STM32 board consolidates all data and transmits it to ESP32 via UART protocol.
- The ESP32 uses MQTT over Wi-Fi to send data to the Thingsboard cloud for remote monitoring.
- The Thingsboard dashboard displays vehicle parameters in real-time, enabling users to track performance and detect anomalies.

System Testing & Validation

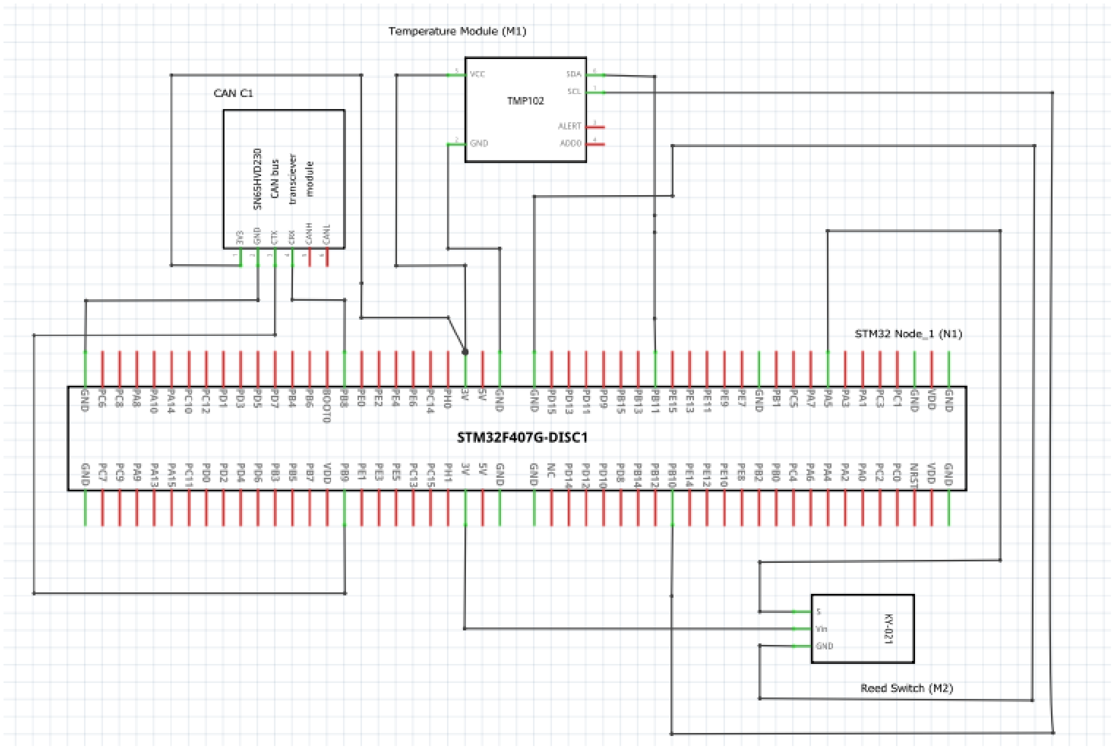
The system undergoes rigorous testing to ensure reliability:

- Hardware Testing: Verifying sensor accuracy and microcontroller performance.
- Software Testing: Ensuring correct data processing and real-time task execution.
- Communication Testing: Validating CAN, UART, and MQTT protocols for efficient data transfer.
- Cloud & Network Testing: Checking seamless data transmission and Thingsboard dashboard responsiveness.

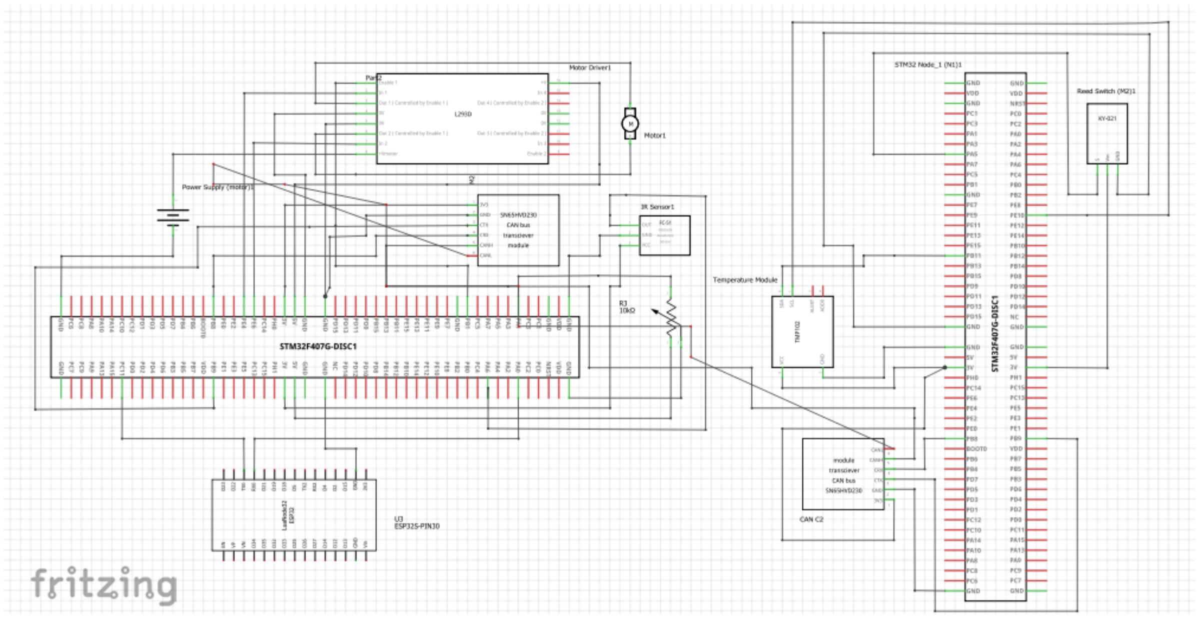
Circuit Diagrams:



Node 1 (STM32F407XX + ESP32)



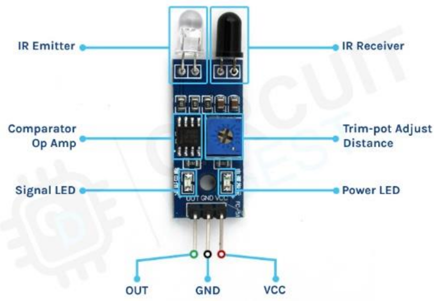
Node 2 (STM32F407xx + peripherals)



Node1 + Node2 (Full System)

3.2 Hardware Components:

3.2.1 Infrared (IR) sensor



An Infrared (IR) sensor is an electronic device that detects infrared radiation in its environment. It typically consists of an IR transmitter (usually an IR LED) and an IR receiver (such as a photodiode or phototransistor). The IR transmitter emits infrared light, which, when reflected off an object, is detected by the receiver. The sensor then processes this information to determine the presence or proximity of objects.

Working Principle:

When the IR transmitter emits infrared light, it travels until it encounters an object. Depending on the object's surface characteristics, the light is either absorbed or reflected. The IR receiver captures the reflected light; the amount of light received indicates the presence and sometimes the distance of the object. For instance, white surfaces reflect more IR light, leading to higher receiver signals, while black surfaces absorb more IR light, resulting in lower signals.

Types of IR Sensors:

1. **Active IR Sensors:** These sensors have both an IR transmitter and receiver. They emit IR light and detect the reflection, making them suitable for proximity sensing and object detection.
2. **Passive IR Sensors (PIR):** These sensors detect the IR radiation emitted by warm objects, such as humans and animals. They are commonly used in motion detection applications.

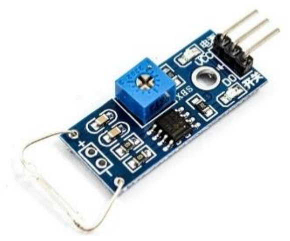
Interfacing with Microcontrollers:

IR sensors can be easily interfaced with microcontrollers like the Arduino. A typical IR sensor module has three pins: VCC (power), GND (ground), and OUT (output). The output pin provides a digital signal indicating the presence (LOW) or absence (HIGH) of an object. By connecting the output pin to a digital input on the microcontroller, you can write code to respond to the sensor's readings.

Considerations:

- **Ambient Light:** Strong ambient light can interfere with IR sensor performance. Using modulated IR light (e.g., at 38 kHz) can help mitigate this issue.
- **Surface Characteristics:** The color and texture of the target surface can affect the sensor's readings due to varying reflectivity.
- **Range:** The effective range of an IR sensor depends on factors like the power of the IR LED, sensitivity of the receiver, and environmental conditions.

3.2.2 Reed Switch



A reed switch is an electromechanical device that operates in response to an applied magnetic field. It consists of two ferromagnetic flexible metal contacts, known as reeds, sealed within a hermetically sealed glass envelope. These contacts are typically open and close when exposed to a magnetic field, though configurations with normally closed contacts that open upon magnetic field application also exist. The reed switch was first developed in 1922 by Professor Valentin Kovalenkov and later refined into the reed relay by Walter B. Ellwood at Bell Telephone Laboratories in 1936.

Construction and Operation

In its simplest form, a reed switch comprises two ferromagnetic reeds positioned with a small gap between them inside a glass tube. When an external magnetic field, either from a permanent magnet or an electromagnetic coil, is applied, the reeds become magnetized and attract each other, closing the gap and completing an electrical circuit. Upon removal of the magnetic field, the reeds' inherent elasticity causes them to separate, opening the circuit. The hermetic sealing of the glass envelope protects the contacts from environmental factors such as dust, dirt, and moisture, ensuring reliable operation even in harsh conditions.

Types of Reed Switches

Reed switches are primarily categorized into two types based on their default state:

1. **Normally Open (NO):** In this configuration, the contacts remain open (disconnected) in the absence of a magnetic field. When a magnetic field is applied, the contacts close, allowing current to flow.
2. **Normally Closed (NC):** Here, the contacts are closed (connected) when no magnetic field is present. Introducing a magnetic field causes the contacts to open, interrupting the current flow.

The choice between NO and NC configurations depends on the specific requirements of the application.

Advantages

Reed switches offer several benefits that make them suitable for a wide range of applications:

- **Hermetic Sealing:** The sealed glass envelope protects the contacts from environmental contaminants, ensuring reliable operation in various conditions.
- **Low Power Consumption:** Reed switches require no power to operate, making them ideal for battery-powered devices.
- **High Reliability and Longevity:** With no mechanical parts subject to wear and tear, reed switches can operate for millions of cycles without degradation.
- **Fast Switching Speed:** They can switch states rapidly, which is essential in

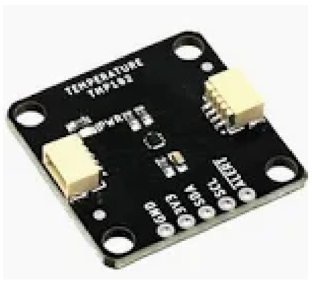
applications requiring quick response times.

Considerations

While reed switches are robust and versatile, certain factors should be considered in their application:

- **Sensitivity to Magnetic Fields:** Proper alignment and strength of the actuating magnet are crucial for reliable operation.
- **Contact Protection:** For applications involving high inrush currents or inductive loads, protective measures such as snubber circuits may be necessary to prevent contact damage.
- **Mechanical Stress:** The glass envelope can be fragile; therefore, care must be taken during installation to avoid breakage.

3.2.3 TMP102



The TMP102 is a low-power digital temperature sensor manufactured by Texas Instruments. It communicates using the I²C (Inter-Integrated Circuit) protocol and provides high accuracy temperature measurements in a compact package.
The TMP102 device is a digital temperature sensor designed for NTC/PTC thermistor replacement where high accuracy is required. The device offers an accuracy of ±0.5°C without requiring calibration or external component signal conditioning. Device temperature sensors are highly linear and do not require complex calculations or lookup tables to derive the temperature. The on-chip 12-bit ADC offers resolutions down to 0.0625°C.
The TMP102 device features SMBus™, two-wire and I2C interface compatibility, and allows up to four devices on one bus. The device also features an SMBus alert function. The device is specified to operate over supply voltages from 1.4V to 3.6V with the maximum quiescent current of 7.5µA over the full operating range.
The TMP102 device is designed for extended temperature measurement in a variety of communication, computer, consumer, environmental, industrial, and instrumentation applications. The device is specified for operation over a temperature range of –40°C to 125°C.

Features of TMP102

- Temperature Range: –40°C to +125°C
- Accuracy: ±0.5°C (typical)

- Resolution: 12-bit (0.0625°C per LSB)
- Low Power Consumption: 10µA in active mode, 1µA in shutdown mode
- I²C Communication: Supports multiple devices on the same bus
- Programmable Alert Function: Allows setting high/low temperature thresholds

TMP102 Pinout and Connection
The TMP102 sensor typically has the following pins:

- VCC - Power Supply (1.4V - 3.6V)
- GND - Ground
- SCL - I²C Clock Line
- SDA - I²C Data Line
- ALERT - Interrupt Output (optional)

I²C Communication in TMP102
The TMP102 communicates via I²C, which is a two-wire protocol (SCL and SDA). Each device on the I²C bus has a 7-bit address, allowing multiple devices to share the same lines.

TMP102 I²C Address

- Default: 0x48 (1001000 in binary)
- Can be changed to 0x49, 0x4A, or 0x4B by modifying the ADD0 pin.

Reading Temperature from TMP102

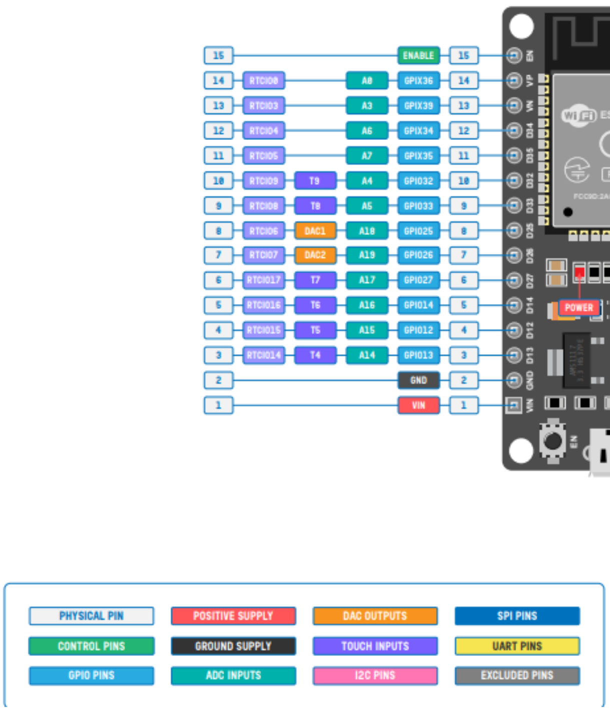
- Master (Microcontroller) sends a Start Condition.
- Master sends TMP102 Address (0x48) with Read bit.
- TMP102 acknowledges and sends the temperature data (2 bytes).
- Master sends Stop Condition.

Example Temperature Data:

Received Data: 0x19 0x00
Convert to Celsius:
Temperature = $(0x1900)/16 = 25.0°\ C$

3.2.3 ESP32

The ESP32 DevKit V1 is a powerful and versatile microcontroller development board based on the ESP32-WROOM-32 module. It is widely used in embedded systems, IoT (Internet of Things) applications, automation, and industrial projects due to its built-in Wi-Fi and Bluetooth capabilities. The ESP32 DevKit V1 comes in different pin configurations, with the 30-pin version being one of the most commonly used variants. This report provides a detailed description of its architecture, pin configuration, and functionalities.



- GPIO pins 34, 35, 36 and 39 are input only.
- TX0 and RX0 (**Serial0**) are used for serial programming.
- TX2 and RX2 can be accessed as **Serial2**.
- Default SPI is VSPI. Both VSPI and HSPI pins can be set to any GPIO pins.
- All GPIO pins support PWM and interrupts.
- Builtin LED is connected to **GPIO2**.
- Some GPIO pins are used for interfacing flash memory and thus are not shown.

Features of ESP32 DevKit V1 (30-Pin Version)

The key features of the **ESP32 DevKit V1** board are:

- **Microcontroller:** ESP32-WROOM-32
- **CPU:** Dual-core Xtensa® 32-bit LX6, up to 240 MHz
- **Memory:** 520 KB SRAM, with external SPI flash support
- **Storage:** Supports external flash memory
- **Wireless Connectivity:**
    - **Wi-Fi:** IEEE 802.11 b/g/n (2.4 GHz)
    - **Bluetooth:** Bluetooth v4.2 (Classic + BLE)
- **Power Supply:** 3.3V operation, with 5V input support
- **GPIO (General Purpose Input/Output):** 30 available pins
- **Analog Inputs:** 12-bit ADC (18 channels)
- **Analog Outputs:** 8-bit DAC on specific pins
- **Digital Interfaces:** UART, SPI, I2C, PWM, I2S, CAN
- **Security Features:** Secure boot, Flash encryption
- **Low Power Modes:** Deep sleep, light sleep, and modem sleep for energy efficiency

ESP32 DevKit V1 Pinout Description (30-Pin Version)

The **ESP32 DevKit V1** (30-pin version) has a variety of power, communication, and I/O pins that support multiple peripherals. Below is a detailed description of the available pins.

Power Pins

Pin	**Function**	**Description**
3V3	Power	Provides a regulated 3.3V output to peripherals
GND	Ground	Connects to system ground
VIN	Power Input	External power input (5V) for powering the board

General Purpose Input/Output (GPIO) Pins

The **ESP32** has **30 GPIO pins**, which can be used for input, output, and communication. However, some GPIOs have specific functionalities and restrictions.

GPIO Pins

The ESP32 development board has 25 GPIO pins that can be assigned different functions by programming the appropriate registers. There are several kinds of GPIOs: digital-only, analog-enabled, capacitive-touch-enabled, etc. Analog-enabled GPIOs and Capacitive-touch-enabled GPIOs can be configured as digital GPIOs. Most of these digital GPIOs can be configured with internal pull-up or pull-down, or set to high impedance.

GPIO Pin	**Functionality**	**Remarks**
GPIO0	Boot Mode Selection	Must be LOW during boot for flashing
GPIO1	TX (UART0)	Used for debugging, avoid using for other functions
GPIO2	ADC, Touch Sensor	Internal pull-down resistor
GPIO3	RX (UART0)	Used for debugging, avoid using for other functions
GPIO4	ADC, PWM, I2C	General-purpose GPIO

GPIO5	ADC, PWM, I2C	Used for SPI SS in some cases
GPIO12	ADC, Touch Sensor	Bootstrapping pin (avoid pull-up/down at boot)
GPIO13	ADC, Touch Sensor	Used as an SPI interface
GPIO14	ADC, PWM, I2C	General-purpose GPIO
GPIO15	ADC, Touch Sensor	Bootstrapping pin
GPIO16	Digital I/O	Supports wake-up from deep sleep
GPIO17	Digital I/O	General-purpose GPIO
GPIO18	SPI Clock (SCK)	Used in SPI communication
GPIO19	SPI MISO	Used in SPI communication
GPIO21	I2C SDA	Used in I2C communication
GPIO22	I2C SCL	Used in I2C communication
GPIO23	SPI MOSI	Used in SPI communication
GPIO25	DAC1	Supports digital-to-analog conversion
GPIO26	DAC2	Supports digital-to-analog conversion
GPIO27	ADC, Touch Sensor	General-purpose GPIO
GPIO32	ADC, Touch Sensor	Supports analog input
GPIO33	ADC, Touch Sensor	Supports analog input
GPIO34	ADC Input	Cannot be used as output
GPIO35	ADC Input	Cannot be used as output
GPIO36	ADC Input	Cannot be used as output
GPIO39	ADC Input	Cannot be used as output

Analog-to-Digital Converter (ADC) Pins

ESP32 has multiple **12-bit ADC (Analog-to-Digital Converter) channels** that can be used for analog signal measurements.

ADC Pin	**Resolution**	**Remarks**
GPIO32	12-bit	Supports ADC functions
GPIO33	12-bit	Supports ADC functions
GPIO34	12-bit	Input only, no output
GPIO35	12-bit	Input only, no output

GPIO36	12-bit	Input only, no output
GPIO39	12-bit	Input only, no output

Digital-to-Analog Converter (DAC) Pins

The **ESP32** features two **8-bit DAC (Digital-to-Analog Converter) channels**, useful for generating analog signals like audio output.

DAC Pin	**Function**
GPIO25	DAC Channel 1
GPIO26	DAC Channel 2

Communication Protocol Pins

The ESP32 supports various communication interfaces, including **UART, SPI, I2C, and I2S**.

Protocol	**Pins Used**	**Description**
UART0	GPIO1 (TX), GPIO3 (RX)	Used for serial debugging
UART1	GPIO9 (TX), GPIO10 (RX)	Can be reassigned to other pins
UART2	GPIO16 (TX), GPIO17 (RX)	General-purpose UART
SPI	GPIO18 (SCK), GPIO19 (MISO), GPIO23 (MOSI), GPIO5 (SS)	Used for SPI communication
I2C	GPIO21 (SDA), GPIO22 (SCL)	Used for I2C communication
I2S	GPIO25, GPIO26, GPIO27	Used for digital audio processing

PWM (Pulse Width Modulation) Pins

ESP32 supports **PWM on almost all GPIOs**, which can be used for motor control, LED dimming, and other applications.

Special Function Pins

Pin	Function	Description
EN (Enable)	Chip Enable	Must be HIGH for normal operation
BOOT (GPIO0)	Boot Mode Selection	Used for flashing firmware

Power Consumption and Low Power Modes

The **ESP32 DevKit V1** is designed to be power-efficient, offering multiple low-power modes:

- **Active Mode:** 160mA current draw
- **Modem Sleep:** 30mA (Wi-Fi/Bluetooth disabled)
- **Light Sleep:** 0.8mA
- **Deep Sleep:** 10µA

3.3 Communication Protocol:

3.3.1 CAN (Controlled Area Network)

The Controller Area Network (CAN) protocol is a robust, high-speed communication standard designed for real-time, embedded systems, primarily used in automotive and industrial applications. It enables microcontrollers and devices to communicate with each other in a network without a host computer.

Overview of CAN Protocol

CAN is a multi-master serial communication protocol, which means any device (called "node") can initiate communication. It was developed by Bosch in 1986 for automotive applications to enable robust and efficient communication between microcontrollers in vehicles.

Key features:

- Real-time communication: Suitable for applications that require quick responses, such as vehicle control systems.
- High reliability: CAN includes error detection and error handling mechanisms to ensure the integrity of communication.
- Multi-node communication: Allows communication between multiple devices on a single bus.
- Differential signaling: CAN uses two wires (CAN_H and CAN_L) for data transmission, providing noise immunity and enabling longer communication distances.

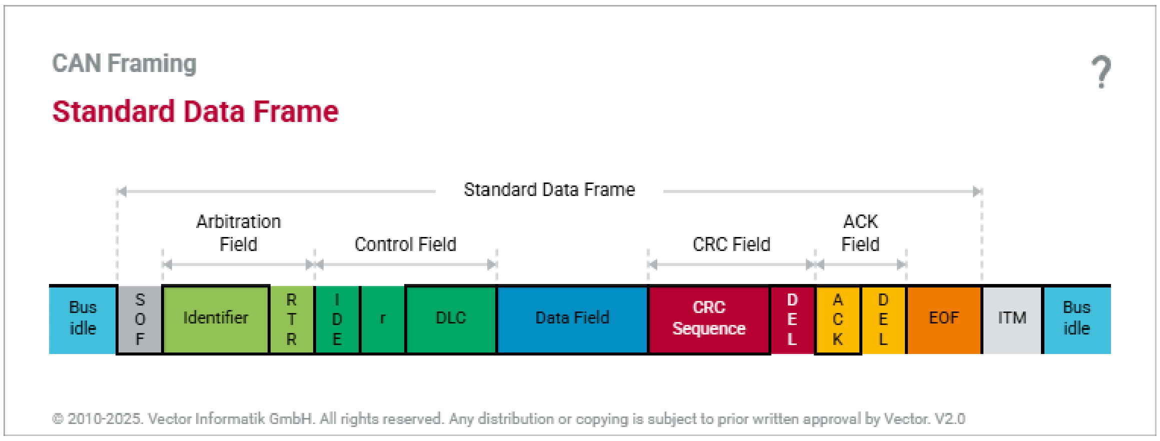
CAN Protocol Layers

CAN operates on the OSI model at the data link layer (Layer 2). Below are the main components:

- Physical Layer: Defines the electrical characteristics of the communication bus, such as voltage levels and bit rates.
- Data Link Layer: Handles message framing, error detection, and access control (responsible for CAN protocol implementation).

  Frame Types: CAN defines several types of frames:

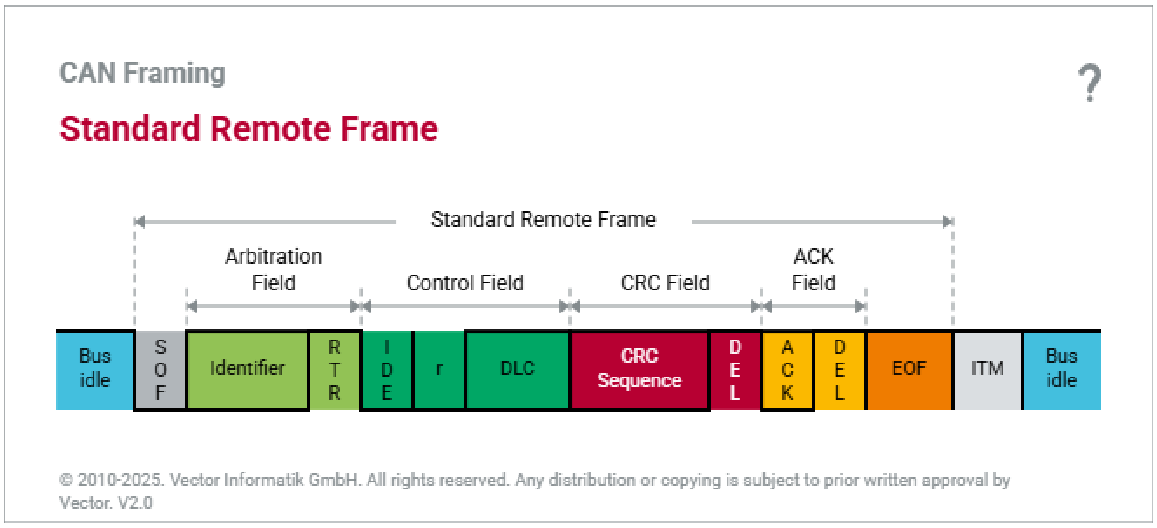
  - Data Frame: Used for transmitting data between nodes.
  - Remote Frame: Used to request data from another node.
  - Error Frame: Indicates errors in the network.
  - Overload Frame: Indicates temporary overload.
  - Status Frame: Used to report the status of the CAN network
  - Transport Layer: This is not explicitly defined in the standard but can be implemented for specific higher-layer protocols.



Standard Data Frame

A Data Frame is used to transmit actual data between nodes. It consists of the following fields:

- Start of Frame (SOF) – Indicates the beginning of a frame.
- Identifier – Defines the message priority (11-bit or 29-bit in extended format).
- Control Field – Specifies the data length (DLC).
- Data Field – Contains the actual data (0 to 8 bytes).
- CRC (Cyclic Redundancy Check) – Ensures data integrity.
- ACK (Acknowledgment) – Receivers send an ACK bit if they successfully received the frame.
- End of Frame (EOF) – Marks the end of the frame.



Remote Frame:

A Remote Frame is used to request data from another node. Instead of carrying data, it only contains a request for a specific identifier.

Key differences between Remote Frame and Data Frame:

- The RTR (Remote Transmission Request) bit is dominant (0) in a Data Frame but recessive (1) in a Remote Frame.
- A Remote Frame has no Data Field – it only requests data.
- The DLC (Data Length Code) field must match the expected Data Frame's DLC.

Connection Overview:

Each STM32 microcontroller interfaces with a CAN transceiver that converts the microcontroller's TX/RX signals into differential signals for the CAN bus. The transceivers communicate over the CANH and CANL lines, ensuring reliable data exchange. The STM32 microcontrollers are configured using the HAL (Hardware Abstraction Layer) library.

Data Transmission and Reception

- Transmission Process:
  - The sender STM32 prepares a CAN frame with a unique identifier.
  - The data is transmitted via the CAN transceiver onto the CAN bus.
  - The receiver STM32 reads the frame and processes the data.
- Reception Process:
  - The receiver listens for incoming frames using interrupts or polling.
  - Once a message is received, the STM32 retrieves the data and performs the necessary operations.

3.3.2 UART (Universal Asynchronous Receiver Transmitter)

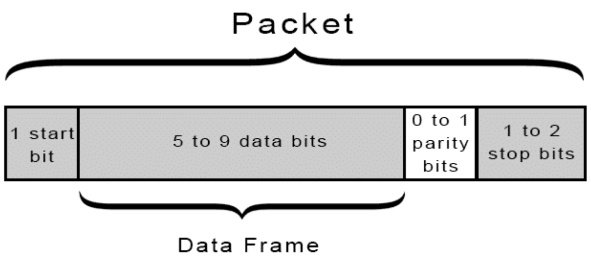
Universal Asynchronous Receiver-Transmitter (UART) is a serial communication protocol widely used for interfacing microcontrollers and peripherals.

How UART Works:

- Two-Wire Communication: UART uses two main data lines:
  - TX (Transmit) – Sends data from one device to another.
  - RX (Receive) – Receives incoming data.
- Full-Duplex Communication: Data can be transmitted and received simultaneously.

- No Clock Signal: Synchronization is achieved by setting a common baud rate between the sender and receiver.

UART Frame Structure



- Start Bit: Indicates the beginning of data transmission (always 0).
- Data Bit: 5 to 9 bits representing the actual data.
- Parity Bit (Optional): Used for error detection (Even, Odd, or None).
- Stop Bit(s): 1 or 2 bits marking the end of the transmission.

UART Modes of Operation

- Polling Mode: The microcontroller continuously checks if data is available.
- Interrupt Mode: The processor is notified when data is received, improving efficiency.
- DMA (Direct Memory Access) Mode: Data is transferred between memory and UART without CPU involvement, enhancing speed and reducing overhead.

UART Characteristics

- Baud Rate: The speed of communication (e.g., 115200 bps).
- Full-Duplex Communication: UART can send and receive data simultaneously.
- Asynchronous Nature: No clock signal is needed; devices must agree on the baud rate.

Advantages and Disadvantages of UART

- Advantages:
  - Simple to Implement – Requires only two wires (TX, RX).
  - Reliable Communication – Error detection via parity bits.
  - No Clock Synchronization Needed – Devices operate independently.
- Disadvantages:
  - Limited Speed – Typically slower than SPI or I2C.

- Short Distance Communication – Affected by noise over long distances.
- Only Supports Two Devices – Unlike SPI or I2C, UART is not suitable for multi-device communication.

3.3.3 I2C (Inter Integrated Circuit)

I²C (Inter-Integrated Circuit) is a serial communication protocol designed for low-speed communication between microcontrollers and peripherals like sensors, EEPROMs, and displays.

Features of I²C

- Uses Two Wires: SCL (Clock) and SDA (Data).
- Supports Multiple Devices: Each device has a unique 7-bit or 10-bit address.
- Master-Slave Communication: One master controls multiple slaves.
- Clock Synchronization: The master generates the clock signal.

I²C Communication Process

- Start Condition: The master pulls SDA low while SCL is high to indicate the beginning of communication.
- Device Address Transmission: The master sends the 7-bit slave address followed by a Read (1) or Write (0) bit.
- Acknowledgment (ACK): The slave acknowledges (pulls SDA low) if it recognizes the address.
- Data Transfer:
  - Write Mode: Master sends data, slave acknowledges.
  - Read Mode: Slave sends data, master acknowledges.
  - Stop Condition: The master pulls SDA high while SCL is high, ending communication.

I²C Speed Modes

- Standard Mode - 100 kHz
- Fast Mode - 400 kHz
- Fast Mode Plus - 1 MHz
- High-Speed Mode - 3.4 MHz

Advantages and Disadvantages of I²C

- Advantages
  - Uses only two wires (SDA, SCL)
  - Supports multiple devices on the same bus
  - Built-in error detection with ACK/NACK
  - More efficient than UART for multi-device communication
- Disadvantages
  - Slower than SPI (max 3.4 MHz vs. 50 MHz)
  - Limited communication distance (short-range applications)
  - Requires pull-up resistors

3.3.4 Message Queuing Telemetry Transport (MQTT)

MQTT was developed in 1999 by Andy Stanford-Clark (IBM) and Arlen Nipper (Eurotech) to create a lightweight and efficient messaging protocol for low-bandwidth and unreliable networks. The main motivation was to enable communication between remote sensors and control systems over satellite networks with minimal bandwidth usage.

Since then, MQTT has become the standard messaging protocol for IoT (Internet of Things), industrial automation, and embedded systems, providing low power consumption, scalability, and reliability.

Key Features of MQTT

1. **Lightweight & Efficient** – Uses minimal bandwidth, making it ideal for IoT and embedded devices.
2. **Publish-Subscribe Model** – Devices communicate through a central **broker**, reducing direct connections and improving efficiency.
3. **Quality of Service (QoS) Levels** – Ensures message delivery based on different reliability levels.
4. **Persistent & Stateful Communication** – Supports retained messages and session persistence, ensuring reliability even after connection loss.
5. **Security & Scalability** – Supports authentication, encryption (SSL/TLS), and can scale across thousands of devices.

Quality of Service (QoS) Levels in MQTT

QoS defines how MQTT handles message delivery reliability between publishers and subscribers. There are three levels of QoS:

1. **QoS 0 – At most once (Fire and Forget)**
   - Message is sent once without acknowledgment.
   - No guarantee of delivery.
   - Suitable for non-critical data (e.g., temperature updates).
2. **QoS 1 – At least once (Guaranteed Delivery)**
   - Message is delivered at least once but may be sent multiple times.
   - Requires acknowledgment from the receiver.
   - Used in applications where data loss is unacceptable, but duplication is manageable.
3. **QoS 2 – Exactly once (Guaranteed and Unique Delivery)**
   - Message is delivered exactly once, ensuring no duplication.
   - Requires multiple handshakes between sender and receiver.
   - Used in financial transactions, critical alerts, and control systems.

Why MQTT is Popular in IoT & Embedded Systems

- **Minimal Overhead:** Works well on resource-constrained devices like ESP32, Raspberry Pi, and STM32.
- **Efficient Power Usage:** Ideal for battery-powered IoT sensors and devices.
- **Reliable Communication:** Ensures data integrity even in unstable networks.
- **Cloud Integration:** Supports platforms like AWS IoT, Google Cloud IoT, ThingsBoard, etc.

MQTT has become the de facto standard for IoT applications due to its efficiency, scalability, and reliability in handling real-time device communication over constrained networks.

Real-Time Sensor Data Acquisition and Cloud Integration Using UART & MQTT

This project involves the use of an ESP32 DevKit V1 to collect sensor data from Node 2 (STM32) via UART (Universal Asynchronous Receiver-Transmitter). The collected data is processed and transmitted to ThingsBoard, an IoT platform, using the MQTT

(Message Queuing Telemetry Transport) protocol. The goal is to enable real-time monitoring and analysis of the sensor data over the cloud.

System Architecture:

The system consists of the following components:

- **STM32 (Node 2):** Collects sensor data and transmits it to ESP32 via UART.
- **ESP32 DevKit V1:** Acts as a bridge between STM32 and MQTT, handling data processing and transmission.
- **UART Communication:** Enables serial data transfer between STM32 and ESP32.
- **WiFi Connection:** ESP32 connects to the internet for data transmission.
- **MQTT Protocol:** A lightweight communication protocol for IoT applications.
- **ThingsBoard:** A cloud-based IoT platform that receives and visualizes sensor data.

Working Principle:

1. **Data Collection from STM32 via UART:**
   - The STM32 microcontroller collects real-time sensor data.
   - The sensor data is sent to ESP32 via UART communication.
   - ESP32 reads the incoming data using Serial.read() and processes it accordingly.
2. **Data Processing on ESP32:**
   - The received data is verified and converted into a suitable format.
   - Any necessary filtering or mathematical operations are performed before transmission.
3. **MQTT Communication Setup:**
   - ESP32 connects to a WiFi network using WiFi.begin().
   - It then establishes a connection with an MQTT broker using the PubSubClient library.
   - If the connection is lost, ESP32 attempts to reconnect every 5 seconds.
4. **Formatting Data in JSON:**
   - The processed sensor data is converted into JSON format using the ArduinoJson library.

- Example JSON structure:

```
{
    "temperature": 25.4,
    "door": 60 ,
      "speed": 1012
}
```

5. **Publishing Data to ThingsBoard via MQTT:**
   - The JSON data is serialized and published to ThingsBoard via MQTT topic.
   - ThingsBoard receives the data, stores it, and visualizes it on dashboards.
6. **Real-time Monitoring & Reconnection Handling:**
   - ESP32 prints sensor values to the Serial Monitor for debugging.
   - A reconnection mechanism ensures continuous data transmission even if the network is disrupted.

Key Features & Advantages:

- **Seamless Data Acquisition:** Data is continuously received from STM32 via UART.
- **MQTT-based Cloud Transmission:** Efficient and lightweight data transmission.
- **ThingsBoard Integration:** Enables real-time monitoring and visualization.
- **Auto-reconnect Mechanism:** Ensures reliable data transmission.
- **Scalability:** Additional sensors and processing can be incorporated as needed.

3.4 Thingsboard

ThingsBoard is an open-source Internet of Things (IoT) platform that enables efficient data collection, processing, visualization, and device management. It supports various communication protocols such as MQTT, CoAP, and HTTP, making it an ideal choice for real-time monitoring and control applications. ThingsBoard provides a scalable and flexible architecture that allows seamless integration of IoT devices, analytics, and dashboards for effective data visualization.

Features of ThingsBoard

1. **Device Management**: ThingsBoard enables remote provisioning, configuration, and control of IoT devices, ensuring efficient device lifecycle management.
2. **Data Collection & Storage**: The platform supports multiple protocols for data acquisition and ensures secure and scalable data storage.
3. **Real-time Data Visualization**: ThingsBoard provides customizable dashboards, widgets, and charts to display real-time sensor data.
4. **Rule Engine & Automation**: Users can define rules and triggers for automated actions, such as sending alerts or executing commands based on predefined conditions.
5. **Security & Access Control**: Role-based access control ensures that only authorized users can access specific functionalities.
6. **Scalability & Deployment Flexibility**: ThingsBoard can be deployed on-premises, in the cloud, or as a hybrid solution to meet different scalability needs.

Role of ThingsBoard in the Project

For the **IR Sensor-based Motor RPM Measurement** project, ThingsBoard serves as the central platform for monitoring and analyzing real-time RPM data. The IR sensor detects motor shaft rotations, and the corresponding pulse data is sent to a microcontroller, which then transmits the data to ThingsBoard via MQTT or HTTP.

Key Implementation Steps:

1. **Sensor Data Acquisition**: The IR sensor captures motor speed in terms of pulses per second.
2. **Microcontroller Processing**: A microcontroller processes the pulses to calculate RPM and formats the data for transmission.
3. **Data Transmission**: The microcontroller sends the RPM data to ThingsBoard using a wireless or wired communication protocol.
4. **Dashboard Configuration**: ThingsBoard visualizes the data through graphs, gauges, and alerts, providing real-time insights into motor performance.
5. **Alerts & Notifications**: The rule engine can trigger notifications if the RPM exceeds or falls below predefined thresholds.

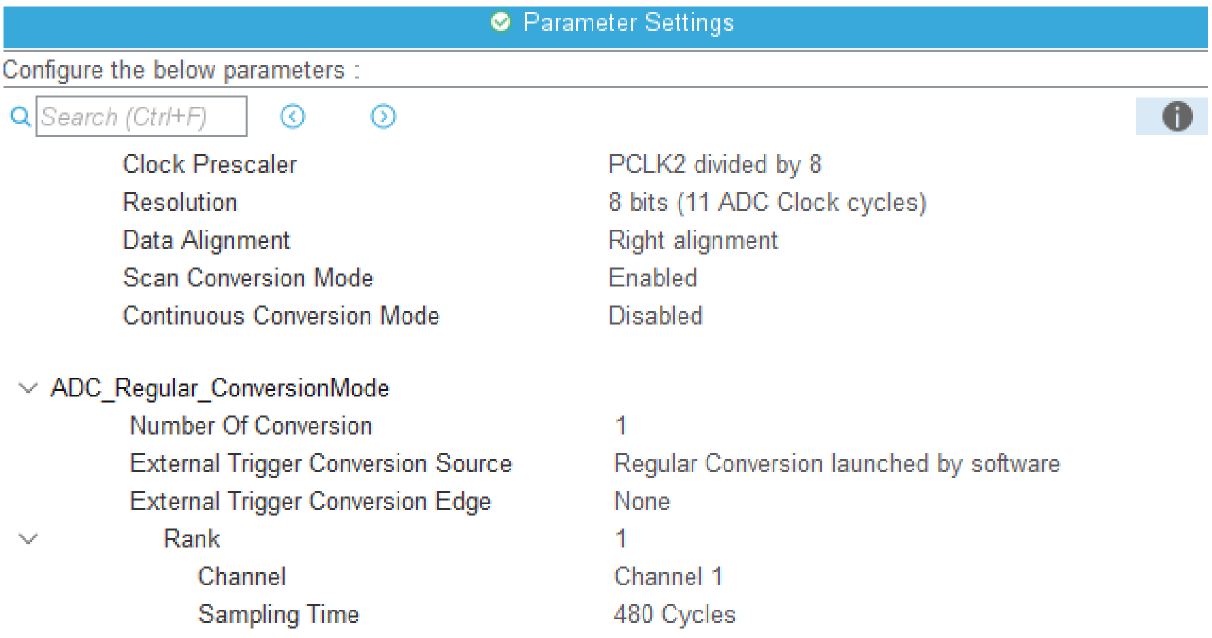
Chapter 4

Implementation

1. 2 STM board configured as CAN node1 and CAN node2. ESP32 interfaced with CAN node1 for IoT capability.
2. As mentioned in the previous sections, sensor interfacing is equally divided among both the nodes. A speed knob (10K potentiometer), motor driver (L293D), Infrared sensor (IR) attached to node1 and temperature sensor (TMP102), door sensor (magnetic reed switch).

Sensor and Peripheral Configurations:

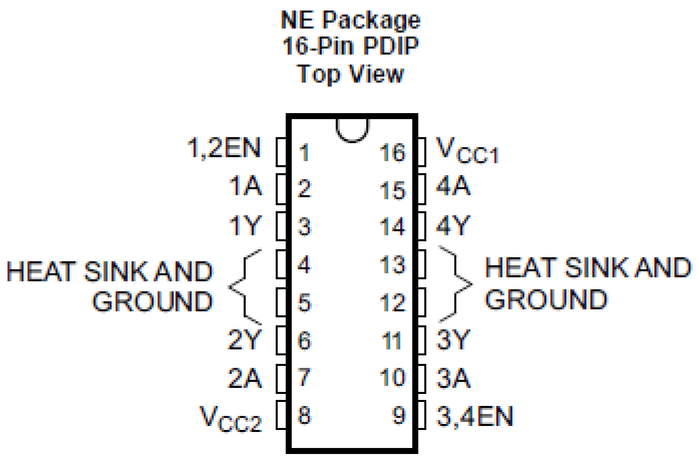
- **Motor Speed Knob:** Implemented via a rotary 10k potentiometer, which was supported by ADC3. Configurations of ADC used are given below:



Clock Prescaler	PCLK2 divided by 8
Resolution	8 bits (11 ADC Clock cycles)
Data Alignment	Right alignment
Scan Conversion Mode	Enabled
Continuous Conversion Mode	Disabled
ADC_Regular_ConversionMode	
Number Of Conversion	1
External Trigger Conversion Source	Regular Conversion launched by software
External Trigger Conversion Edge	None
Rank	1
Channel	Channel 1
Sampling Time	480 Cycles

Since, resolution of ADC is 8 bits, the range of our speed knob was from 0 to 255. Another important factor is sampling time. 480-cycle sampling time is a tradeoff that ensures the ADC can capture a stable, accurate representation of the input signal, avoiding common issues like incomplete charging of the sample-and-hold capacitor and ensuring that the ADC's internal circuits are properly settled for an accurate conversion.

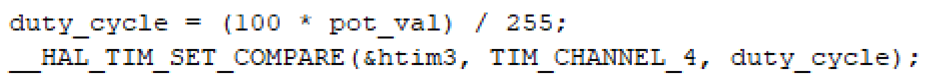
- **Motor driver (L293D)**: It is a quad half-H-bridge motor driver IC, which was used for driving the DC motor. It is commonly used in robotics and various automation applications due to its ease of use and built-in protection features. L293D is designed to provide bidirectional drive currents of up to 600-mA at voltages from 4.5 V to 36 V.



Pins concerning CAN Drive Smart dashboard are:

Pin 1	To enable the motor
Pin 2,7	For direction control of motor. 3 conditions possible: forward, reverse and stop
Pin 3,6	Provides supply to motor terminals
Pin 4,5	Ground motor, battery and STM board together
Pin 8	Actual positive supply from the battery
Pin 16	5V to IC for logic translation

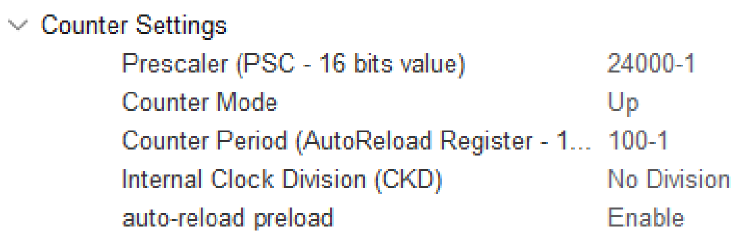
Success of motor speed control is subjected to ADC conversion of potentiometer and pulse width modulation (PWM) signal provided to pin 1 of L293D. Since, PWM timer is based on duty cycles, it would vary as we change the position of speed knob. ADC values were linearly used for configuring the PWM duty cycles. Logic applied is as follows:



```
duty_cycle = (100 * pot_val) / 255;
__HAL_TIM_SET_COMPARE(&htim3, TIM_CHANNEL_4, duty_cycle);
```

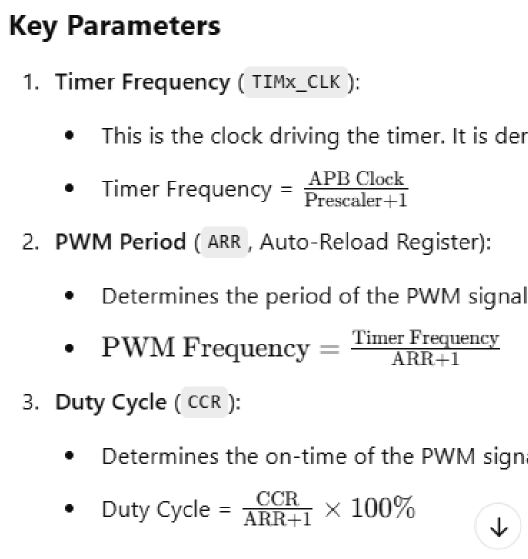
So, duty cycle ranging from 0 to 100 will define the motor speed, 0 being the lowest (motor is stopped) and 100 for the motor running at its highest potential.

Configuration of PWM timer:



Counter Settings

Prescaler (PSC - 16 bits value)	24000-1
Counter Mode	Up
Counter Period (AutoReload Register - 1...	100-1
Internal Clock Division (CKD)	No Division
auto-reload preload	Enable



Key Parameters

1. **Timer Frequency** (`TIMx_CLK`):
   - This is the clock driving the timer. It is derived from the system clock and prescaler settings.
   - Timer Frequency = $\frac{\text{APB Clock}}{\text{Prescaler}+1}$
2. **PWM Period** (`ARR` , Auto-Reload Register):
   - Determines the period of the PWM signal.
   - $\text{PWM Frequency} = \frac{\text{Timer Frequency}}{\text{ARR}+1}$
3. **Duty Cycle** (`CCR`):
   - Determines the on-time of the PWM signal.
   - Duty Cycle = $\frac{\text{CCR}}{\text{ARR}+1} \times 100\%$

APB clock is configured to 24MHz using Stm32cubeIDE's clock tree. The defined parameters result in timer frequency as 1000 Hz and PWM frequency of 10Hz.

- **Infrared Sensor**: It was configured on a GPIO as an external interrupt. Its purpose was to count the revolutions of wheel. Hence, critical for speed and distance measurement.

- **Temperature Sensor (TMP102):** Interfaced via I2C in standard mode (100KHz). Slave address of this sensor was by-default, 0x48. Register address to access the temperature data from this module was 0x00. Register addresses of other important and relevant data is given below:

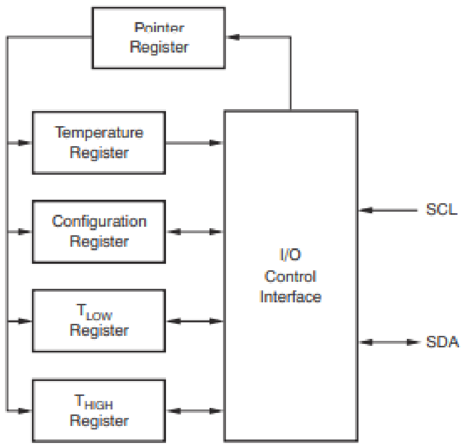


Figure 6-6. Internal Register Structure

Table 6-6. Pointer Register Byte

P7	P6	P5	P4	P3	P2	P1	P0
0	0	0	0	0	0	Register Bits	

Table 6-7. Pointer Addresses

P1	P0	REGISTER
0	0	Temperature Register (Read Only)
0	1	Configuration Register (Read/Write)
1	0	T_{LOW} Register (Read/Write)
1	1	T_{HIGH} Register (Read/Write)

Temperature register holds 2 bytes of data, out of which only 12 bits are significant, i.e., byte 1 (D7-D0) and byte 2 (D7-D4).

Table 6-8. Byte 1 of Temperature Register(1)

D7	D6	D5	D4	D3	D2	D1	D0
T11 (T12)	T10 (T11)	T9 (T10)	T8 (T9)	T7 (T8)	T6 (T7)	T5 (T6)	T4 (T5)

(1) Extended mode 13-bit configuration shown in parenthesis.

Table 6-9. Byte 2 of Temperature Register(1)

D7	D6	D5	D4	D3	D2	D1	D0
T3 (T4)	T2 (T3)	T1 (T2)	T0 (T1)	0 (T0)	0 (0)	0 (0)	0 (1)

(1) Extended mode 13-bit configuration shown in parenthesis.

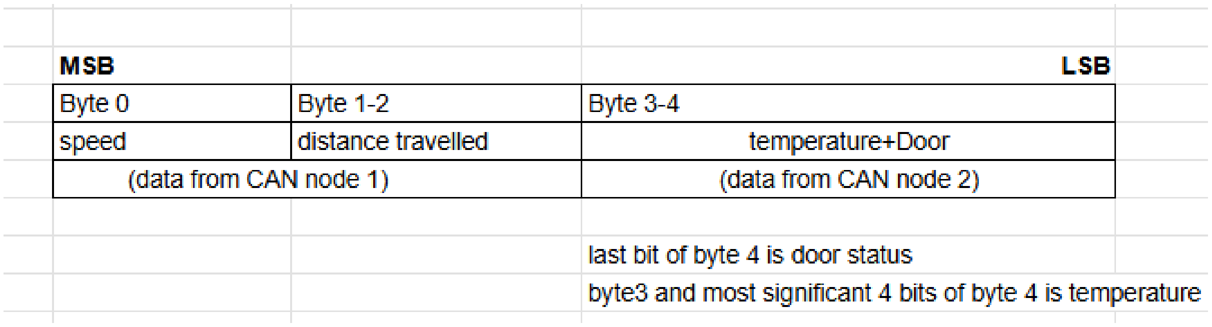
In TMP102, One LSB equals 0.0625°C. So, 12-bit data received from this sensor is to be multiplied by LSB value to get the actual temperature.

- **Reed Switch:** This sensor module was configured to GPIO external interrupt. It was used to demonstrate the vehicle's door status (open/close) as a safety mechanism. In proximity of magnet attached to the door, the circuit would be completed and it would break, after the magnet was detached.

Software Architecture Model of Node 1:

Application Layer
• Task 1 - Processing speed knob • Task 2 - Speed & distance measurement • Task 3 - CAN packet reception & processing • Task 4 - Transmitting vehicle parameter packet to ESP via UART
System service calls
• GPIO (IR sensor, Motor driver control) • UART (STM-ESP32 communication) • CAN (STM node 1 – STM node 2 communication) • Timers (PWM, periodic)
Embedded OS (FreeRTOS)
Hardware Abstraction Layer (HAL)

- Task 1 (Highest priority) and Task 2 are responsible for measuring speed and distance of vehicle in real-time.
- Task 3 is supported by CAN interrupts and receives the packet from node 2. Then it processes that data and combines to form the entire parameter payload. The format of packet is as follows:



MSB		**LSB**
Byte 0	Byte 1-2	Byte 3-4
speed	distance travelled	temperature+Door
(data from CAN node 1)		(data from CAN node 2)
		last bit of byte 4 is door status
		byte3 and most significant 4 bits of byte 4 is temperature

- Task 4 (Lowest priority) transmits this packet to ESP32 for IoT connectivity and dashboard functionality.

Software Architecture Model of Node 2:

Application Layer

- Task 1 - Read temperature
- Task 2 - CAN packet transmission to Node 1

System service calls

- GPIO (Reed switch)
- I2C (TMP102)
- CAN (STM node 1 – STM node 2 communication)

Embedded OS (FreeRTOS)

Hardware Abstraction Layer (HAL)

- Node 2 is relatively simpler interfacing. Collects the temperature and door status data via respective sensors.
- CAN packet transmitted to node 1 is only 2 bytes of data. Least significant bit of that data indicates the status of door , whether opened or closed.

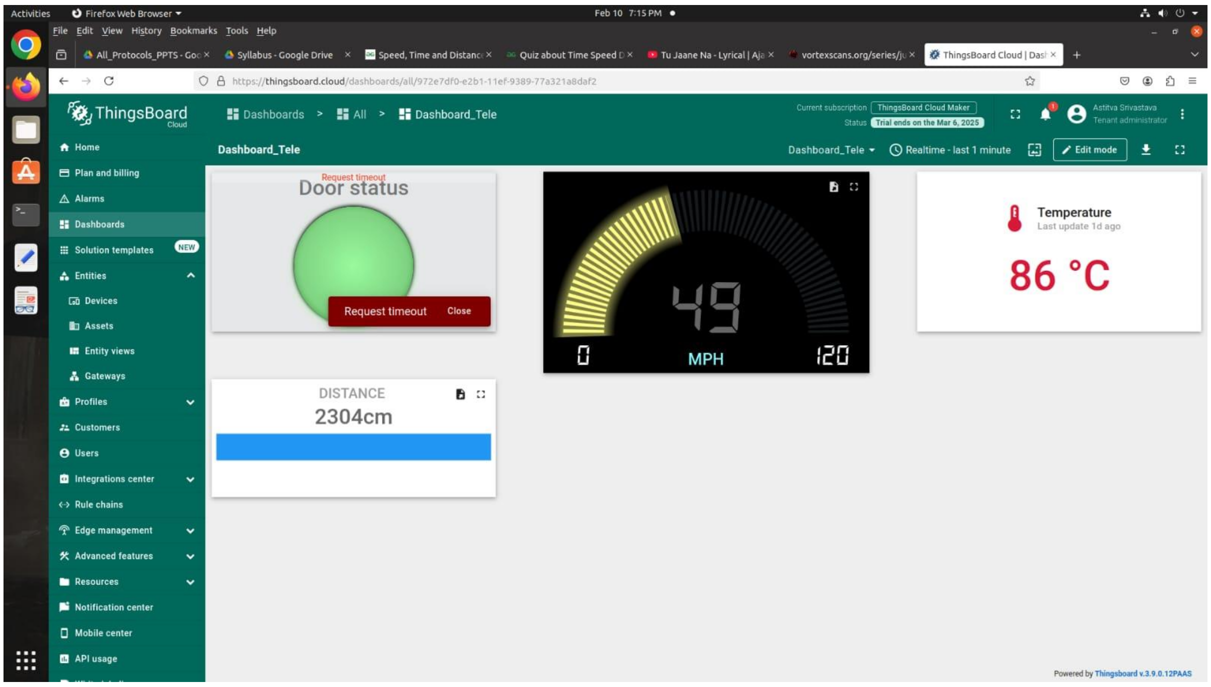
ESP32 (Data Reception and Processing):

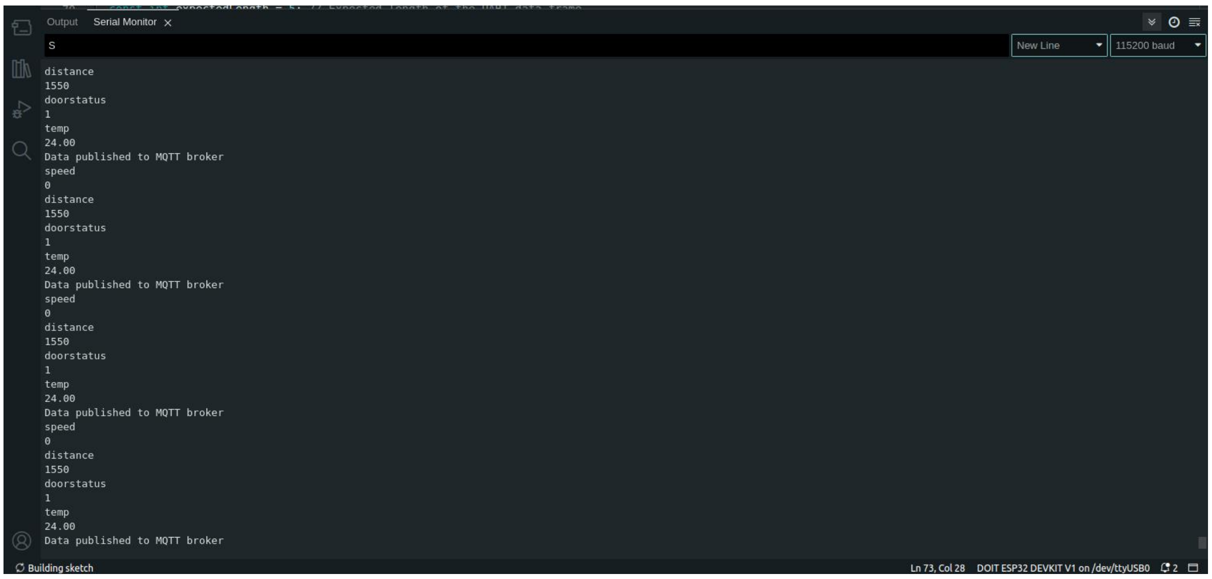
- **Data Reception**: ESP32 receives the data packet from the STM32 over UART communication protocol. Baud rate maintained was standard 115200 bps.
- **Data Decoding**: ESP32 decodes the incoming data packet into individual parameters (speed, distance, temperature, door status).
- **Data Transmission**: ESP32 sends the processed data to the **ThingsBoard** platform via **MQTT**. The ESP32 acts as an MQTT client, transmitting the data to the ThingsBoard MQTT broker.

Chapter 5

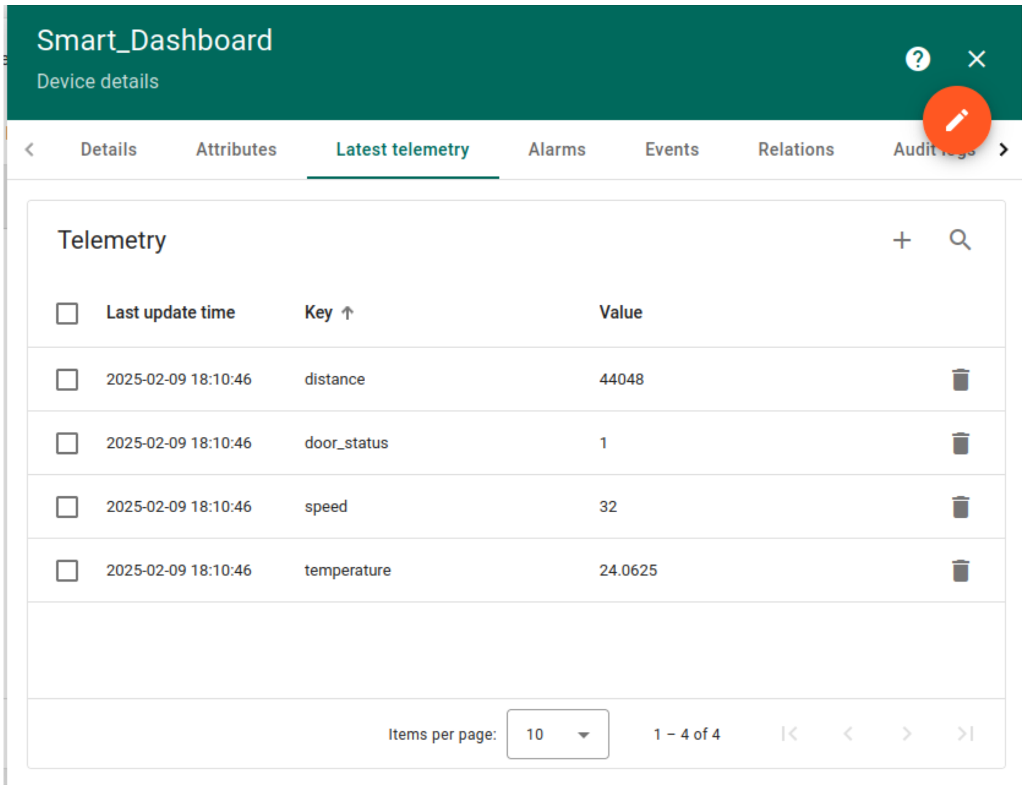
Results

- Data acquisition from various sensors via serial protocols was achievable in real-time.
- Pre-emptive RTOS scheduling of tasks based on priority would schedule and update the parameters properly in sequence without any race condition. Synchronization mechanisms like semaphore and for inter-process communication notifications and queues were used.
- CAN reception and transmission was successful and verified at both the CAN nodes.
- ESP32 successfully receives data from the STM32 and processes it in real-time.
- **ThingsBoard** dashboard displays updated information about the vehicle's speed, temperature, distance, and door status in real-time.





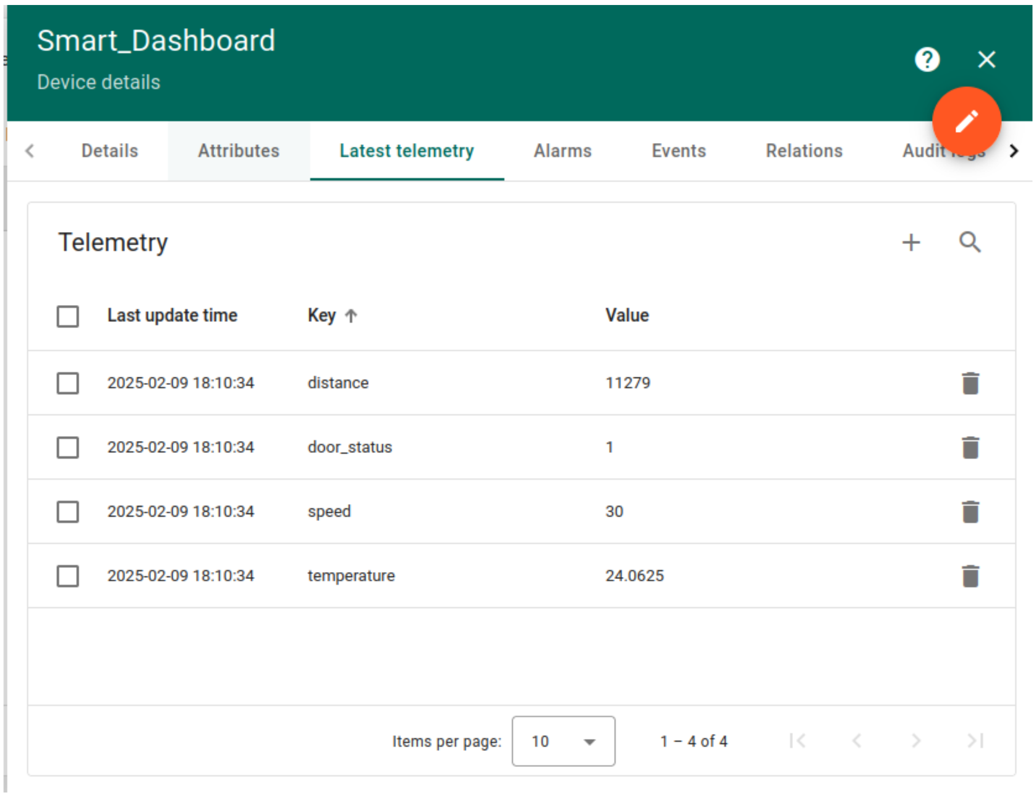
```
s
distance
1550
doorstatus
1
temp
24.00
Data published to MQTT broker
speed
0
distance
1550
doorstatus
1
temp
24.00
Data published to MQTT broker
speed
0
distance
1550
doorstatus
1
temp
24.00
Data published to MQTT broker
speed
0
distance
1550
doorstatus
1
temp
24.00
Data published to MQTT broker
```



Telemetry

Last update time	Key	Value
2025-02-09 18:10:46	distance	44048
2025-02-09 18:10:46	door_status	1
2025-02-09 18:10:46	speed	32
2025-02-09 18:10:46	temperature	24.0625

Items per page: 10 1 – 4 of 4



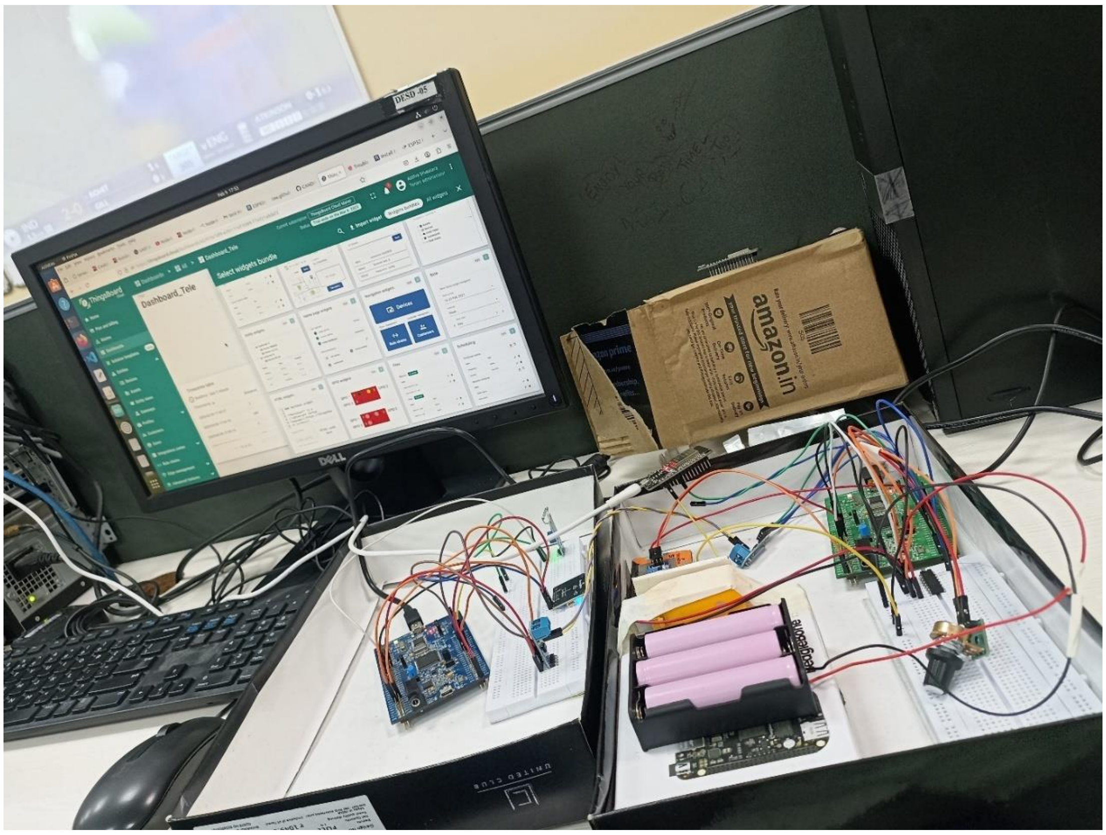
Smart_Dashboard

Device details

Telemetry

Last update time	Key	Value
2025-02-09 18:10:34	distance	11279
2025-02-09 18:10:34	door_status	1
2025-02-09 18:10:34	speed	30
2025-02-09 18:10:34	temperature	24.0625

Items per page: 10 1 – 4 of 4



Chapter 6
Conclusion

6.1 Conclusion

As the automotive industry embraces digital transformation, real-time vehicle monitoring has become a necessity for improving performance, safety, and efficiency. The CANDrive Smart Dashboard is a cutting-edge IoT-based solution that integrates CAN bus communication, FreeRTOS, MQTT, and cloud-based analytics to deliver a reliable, real-time, and scalable vehicle monitoring system. By leveraging modern embedded systems and IoT technologies, this system provides a comprehensive, data-driven approach to vehicle diagnostics, helping both individual users and fleet managers optimize vehicle performance and maintenance.

With its ability to deliver real-time insights and predictive analytics, the CANDrive Smart Dashboard represents the future of smart vehicle monitoring, offering a powerful, scalable, and customizable solution for next-generation automotive applications.

6.2 Future Enhancement

To further enhance the capabilities of the CANDrive Smart Dashboard, the following advancements can be integrated:

- Battery Management System: Utilizing the INA219 sensor to accurately monitor battery voltage, current, and power consumption, ensuring efficient power management and early fault detection.
- Fuel Level Monitoring: Implementing a capacitive fuel level sensor or an ultrasonic fuel sensor to provide precise real-time fuel measurements, improving fuel efficiency tracking.
- GPS-Based Location Tracking: Integrating a GPS module (such as NEO-6M or u-blox M8) using GNSS (Global Navigation Satellite System) technology, which includes GPS, GLONASS, and Galileo for accurate and reliable vehicle positioning.

Chapter 7

References

[1] J. Lee, M. Kim, and S. Park, "IoT-Based Real-Time Vehicle Monitoring and Diagnostics Using Cloud Computing," *IEEE Internet of Things Journal*, vol. 6, no. 5, pp. 7854-7862, Oct. 2019.

[2] H. Choi, K. Jung, and Y. Lee, "Sensor-Based Vehicle Performance Monitoring: Applications of IR, Temperature, and Door Sensors," *Sensors*, vol. 20, no. 12, p. 3456, June 2020.

[3] R. Srinivasan, A. Kumar, and P. Das, "Real-Time Embedded Systems for Vehicle Monitoring Using STM32 Microcontrollers," *International Journal of Embedded Systems*, vol. 10, no. 3, pp. 215-225, 2018.

[4] X. Zhang, Y. Lin, and B. Wang, "Efficient Data Communication Protocols in IoT: A Comparative Analysis of UART and MQTT," *IEEE Transactions on Communications*, vol. 69, no. 4, pp. 2401-2412, Apr. 2021.

[5] A. Gupta, V. Mehta, and S. Roy, "Power Monitoring in IoT Systems: Implementation of INA219 for Real-Time Power Management," *IEEE Transactions on Industrial Electronics*, vol. 69, no. 8, pp. 8567-8575, Aug. 2022.

[6] P. Kumar, R. Sharma, and N. Verma, "Cloud-Based Data Visualization for IoT Applications: Analyzing ThingsBoard and Other Frameworks," *Journal of Cloud Computing*, vol. 11, no. 2, pp. 101-118, Mar. 2023.

- TMP102 Datasheet
- STM32F407xxx Datasheet
- ESP32 WROOM Documentation
- docs.espressif.com
- L293D Datasheet