CHAPTER - 1
INTRODUCTION

1.1 INTRODUCTION

The Smart Energy Meter Gateway connects voltage and current sensors, an STM32 microcontroller, and a cloud platform, enabling real-time data acquisition, processing, and monitoring. It plays a key role in industrial environments where energy data is crucial for decision-making, predictive maintenance, and efficiency. By integrating voltage (ZMPT101B) and current (ACS712) sensors, the gateway optimizes power consumption and ensures system reliability.

With the rise of Industry 4.0, smart automation and remote energy monitoring are essential. The gateway acts as a bridge between sensors, STM32F446RE, a rugged A5D2X board, and cloud platforms, ensuring secure data transmission via USART. Local monitoring via LCD and cloud-based visualization provide real-time insights, helping industries reduce energy wastage, prevent failures, and improve operations.

By integrating ThingsBoard or a custom cloud platform, the system enables data visualization, alerts, and analytics. Users can track voltage and current levels, detect power fluctuations, and take preventive actions. This project showcases how Industrial IoT Gateways drive intelligent, data-driven energy management, enhancing productivity and sustainability.

1.2 SCOPE OF THE PROJECT

The Smart Energy Meter Gateway project focuses on real-time data collection, processing, and transmission using an interconnected system of voltage and current sensors, an STM32 microcontroller, and a rugged A5D2X board. It provides a scalable and efficient solution for industrial energy monitoring, enabling seamless communication between sensors and the cloud for remote access and actionable insights.

Key Features and Capabilities

- **Real-time Data Collection**: The system gathers voltage and current data using ZMPT101B and ACS712 sensors, ensuring accurate and timely energy monitoring.
- **Wired Communication**: The STM32 microcontroller transmits processed sensor data to the rugged A5D2X board via USART, ensuring stable and reliable communication.
- **Cloud Integration**: The gateway connects to ThingsBoard or a custom cloud platform, enabling real-time data visualization, analysis, and remote management.
- **Scalability**: The system can be expanded to support additional sensors and devices as per industrial needs.
- **Predictive Maintenance**: Continuous data analysis helps identify power fluctuations, predict failures, and schedule maintenance, reducing downtime and operational costs.
- **Alerts and Notifications**: The system generates alerts for abnormal voltage or current variations, ensuring timely intervention and preventing potential failures.

Applications

The Smart Energy Meter Gateway is applicable across various industries, including:

- **Manufacturing**: Real-time energy monitoring, equipment protection, and fault detection.
- **Logistics & Warehousing**: Power consumption tracking for efficient operations.
- **Energy Management**: Monitoring voltage and current patterns to optimize

energy usage and reduce waste.

- **Smart Grids**: Enhancing grid stability through real-time power quality monitoring.

1.3 OBJECTIVE OF THE PROJECT

The primary objective of the Smart Energy Meter Gateway project is to design and develop a reliable system for real-time energy monitoring, data collection, and transmission in industrial environments. By providing accurate voltage and current data with actionable insights, the system aims to enhance energy efficiency, minimize downtime, and optimize power consumption.

Specific Objectives:

- **Real-Time Data Acquisition**: Develop a system to measure voltage and current using ZMPT101B and ACS712 sensors, ensuring precise energy monitoring.
- **Seamless Data Transmission**: Establish a USART-based communication framework to send processed sensor data from STM32F446RE to the rugged A5D2X board.
- **Cloud Integration**: Implement secure data transfer to ThingsBoard or a custom cloud platform for visualization and analysis.
- **Remote Monitoring and Management**: Enable real-time access to energy data via a user-friendly cloud-based dashboard.
- **Predictive Maintenance**: Use collected power data to detect anomalies, prevent failures, and reduce maintenance costs.
- **Alert Generation**: Configure the system to send notifications for abnormal voltage or current variations, ensuring timely intervention.
- **Scalability and Flexibility**: Design the system to support additional sensors and features, making it adaptable for various industrial energy management applications.

1.4 METHODOLOGY

The methodology for developing the Smart Energy Meter Gateway follows a structured approach to ensure accurate data acquisition, seamless communication, and efficient cloud-based monitoring. The process involves multiple stages, including hardware selection, software development, testing, and deployment.

Step 1: Requirement Analysis

The project begins with identifying the essential components and technologies for voltage and current measurement, data processing, and cloud integration. Key aspects analyzed include:

- Selection of sensors: ZMPT101B (voltage) and ACS712 (current) for energy monitoring.
- Communication protocols: USART for wired data transmission between STM32F446RE and the A5D2X rugged board.
- Cloud platform selection: ThingsBoard or a custom cloud solution for data storage, visualization, and analytics.

Step 2: Hardware Integration

The hardware components are assembled and interconnected for seamless data flow:

- Sensors (ZMPT101B & ACS712) are interfaced with the STM32F446RE microcontroller for real-time voltage and current data acquisition.
- USART communication is established between the STM32F446RE and the rugged A5D2X board for data transmission.
- LCD display integration with A5D2X to show real-time sensor readings locally.

Step 3: Software Development

The software framework is developed to enable sensor data acquisition, processing, and cloud transmission:

- Embedded firmware is developed on the STM32 to collect, process, and send voltage/current data via USART.
- Python-based scripts on the A5D2X rugged board handle received data and transmit it to the cloud.
- A cloud dashboard is created for real-time visualization, alerts, and analytics.

Step 4: Testing and Validation

The system undergoes rigorous testing to ensure reliable performance and data accuracy:

- Unit testing verifies sensor readings and USART communication.
- Integration testing ensures seamless data flow from sensors to STM32, A5D2X, and the cloud.
- Performance testing evaluates data transmission efficiency, cloud response time, and system stability.

Step 5: Deployment and Monitoring

After successful testing, the system is deployed for real-world evaluation:

- The Smart Energy Meter Gateway is installed, with sensors connected to the STM32F446RE.
- Continuous monitoring is performed to analyze system performance, detect anomalies, and optimize efficiency.
- User feedback is collected to implement improvements and additional features.

CHAPTER - 2

LITERATURE SURVEY

The literature survey provides a comprehensive review of existing research, technologies, and methodologies related to Smart Meters and Gateway Communication in Industrial IoT (IIoT). The focus is on sensor integration, communication protocols, cloud-based data management, real-time monitoring, and predictive maintenance. Understanding previous developments helps identify gaps and formulate an effective approach for the proposed Smart Meter and Gateway Communication project.

2.1 Smart Meters and IoT Gateways

Smart meters and IoT gateways play a crucial role in industrial automation by ensuring efficient data transmission between sensors, devices, and cloud platforms. Research indicates that these gateways are essential for real-time data acquisition, processing, and secure cloud integration. They also enhance operational efficiency and predictive maintenance.

- **Smith et al. (2023)** developed an intelligent IoT gateway with edge computing capabilities, improving real-time decision-making and reducing cloud dependency.
- **Johnson and Lee (2022)** proposed a multi-layer IoT gateway architecture, integrating BLE and Ethernet for efficient industrial data transmission, ensuring low latency and high reliability.

2.2 Sensor Integration and Data Acquisition

Accurate sensor integration is vital for smart meters to monitor electrical parameters such as voltage, current, and power consumption. Efficient data acquisition ensures enhanced monitoring and predictive maintenance.

- **Patel et al. (2022)** implemented a sensor-based smart metering system using STM32 microcontrollers, reducing energy wastage by 30%.

- **Garcia and Wang (2021)** highlighted the role of calibration techniques in improving sensor accuracy for industrial energy monitoring applications.

2.3 Communication Protocols for Smart Meters

Communication protocols ensure seamless data transmission between smart meters, gateways, and cloud servers. The selection of the appropriate protocol is crucial for reliable and efficient data transfer.

- **Singh and Sharma (2023)** compared MQTT, CoAP, and HTTP for IIoT applications, concluding that MQTT's lightweight nature and publish-subscribe model performed best in large-scale deployments.
- **Wang and Li (2022)** explored BLE's role in short-range industrial communication, reporting efficient data transmission with minimal interference.

2.4 Cloud Integration and Data Management

Cloud platforms enable scalable data storage, real-time visualization, and advanced analytics for smart meters and IIoT systems. Integrating IIoT gateways with cloud services enhances predictive maintenance and operational efficiency.

- **Chen et al. (2023)** implemented a ThingsBoard-based smart metering system, achieving a 35% improvement in real-time energy monitoring accuracy.
- **Lee and Johnson (2022)** designed a cloud-integrated IIoT system for remote anomaly detection, reducing energy losses by 40%.

2.5 Predictive Maintenance and Real-Time Monitoring

Predictive maintenance plays a crucial role in IIoT, allowing industries to detect potential equipment failures and optimize maintenance schedules using real-time data from smart meters.

- **Smith et al. (2023)** developed a predictive maintenance system using machine learning on real-time smart meter data, reducing downtime by 50%.
- **Johnson and Kim (2022)** applied deep learning techniques to analyze sensor

data, achieving a 92% accuracy in predicting power system failures.

- **Garcia and Lee (2021)** integrated predictive analytics with ThingsBoard for remote diagnostics, reducing operational costs and improving efficiency.

2.6 Security in IoT Gateways

Ensuring secure data transmission and storage is essential for smart metering systems. Recent research has focused on encryption, authentication, and anomaly detection for industrial applications.

- **Wang et al. (2023)** proposed a blockchain-based security framework for smart meters, enhancing data integrity and preventing cyber threats.

- **Chen and Zhang (2022)** developed a lightweight encryption algorithm for resource-constrained IoT devices, reducing security risks without compromising performance.

CHAPTER - 3

SYSTEM DEVELOPMENT

3.1 HARDWARE REQUIREMENTS

The Smart Meter with Gateway communication requires a set of hardware components for sensor data collection, processing, communication, and cloud integration. Each component is selected to ensure reliable performance, efficient data handling, and real-time monitoring. The primary hardware elements include sensors, microcontrollers, communication modules, and network interfaces.

1. STM32 Nucleo F446RE

- **Role:** Acts as the central microcontroller, processing sensor data and managing communication.
- **Features:**

- ARM Cortex-M4 processor with 180 MHz clock speed.
- Multiple GPIOs for sensor integration.
- Supports UART, I2C,ADC, and SPI interfaces.
- Low power consumption, suitable for industrial environments.

2. ZMPT101B Voltage Sensor

- **Role:** Measures AC voltage levels for power monitoring.
- **Features:**

- High accuracy in detecting AC voltage fluctuations.
- Compatible with ADC inputs of microcontrollers.
- Compact and easy to integrate into circuits.

3. ACS712 Current Sensor

- **Role:** Measures AC/DC current for energy consumption analysis.
- **Features:**

- Provides analog output proportional to current.
- Isolated measurement for safety.
- Suitable for real-time monitoring applications.

4. A5D2X Rugged Board

- **Role:** Acts as an edge device to receive sensor data from the STM32 and forward it to the cloud.
- **Features:**

- Built for industrial applications with robust construction.
- Supports Ethernet for cloud connectivity.
- Provides data storage and pre-processing capabilities.

5. Ethernet Module

- **Role:** Enables reliable long-range communication by transmitting sensor data to the cloud platform.
- **Features:**

- Fast and stable connectivity.
- Supports real-time data transfer.
- Compatible with industrial networks.

6. LCD Display

- **Role:** Displays real-time sensor readings and system status information.
- **Features:**

- Clear visibility with adjustable brightness.
- I2C communication for easy interfacing.
- Compact size suitable for panel mounting.

7. Power Supply

- **Role:** Provides the necessary power to all hardware components.
- **Features:**

- Stable and regulated 5V/3.3V output.
- Over-voltage and short-circuit protection.

8. Connectors and Wires

- **Role:** Establish electrical connections between sensors, modules, and the microcontroller.
- **Features:**
- Reliable signal transmission.
- Flexible and durable for industrial applications.

3.2 SOFTWARE REQUIREMENTS

The Smart Meter with Gateway Communication relies on various software components for sensor data acquisition, communication, data processing, and cloud integration. The software ensures smooth operation, real-time monitoring, and secure data transmission. The following are the key software requirements for the project:

1. STM32CubeIDE

- **Purpose:** Used for writing, compiling, and debugging embedded C code for the STM32 Nucleo F446RE microcontroller.
- **Features:**
  - Integrated development environment with GCC compiler.
  - Provides HAL (Hardware Abstraction Layer) for simplified hardware interfacing.
  - Supports real-time debugging and simulation.

2. STM32CubeMX

- **Purpose:** Configuration and code generation tool for STM32 microcontrollers.
- **Features:**
  - Allows graphical configuration of peripherals.

- Generates optimized initialization code.
- Reduces development time by automating routine tasks.

3. ThingsBoard

- **Purpose:** Cloud platform for data visualization, storage, and management.
- **Features:**
  - Provides real-time dashboards and customizable widgets.
  - Supports MQTT and HTTP for data transmission.
  - Offers remote monitoring and alert generation.

4. Keil µVision (Optional)

- **Purpose:** Alternative development environment for STM32 firmware development.
- **Features:**
  - Advanced debugging features.
  - ARM Compiler for optimized code.
  - Supports RTOS development for multitasking applications.

5. MQTT Protocol

- **Purpose:** Facilitates lightweight and reliable data transmission between the gateway and ThingsBoard.
- **Features:**
  - Supports publish/subscribe messaging model.
  - Low bandwidth consumption.

6. UART and I2C Drivers

- **Purpose:** Enables serial communication between sensors, the STM32
- **Features:**

- Provides stable and error-free data transmission.
- Supports multi-device communication.
- Integrated within the STM32 HAL library.

7. C Programming Language

- **Purpose:** Primary language used for firmware development on the STM32 microcontroller.
- **Features:**
  - Efficient low-level meAdvanced debugging features.
  - ARM Compiler for optimized code.
  - Supports RTOS development for multitasking applications.Supports publish/subscribe messaging model.
  - Low bandwidth consumption.
  - Ensures secure communication using TLS.Provides stable and error-free data transmission.
  - Supports multi-device communication.
  - Integrated within the STM32 HAL library.Supports direct hardware access.
  - Compatible with embedded real-time systems.Efficientmory management.
  - Supports direct hardware access.
  - Compatible with embedded real-time systems.

8. Python

- **Purpose:** Used for creating scripts for data analysis and visualization.
- **Features:**
  - Compatible with ThingsBoard API for data extraction.
  - Supports data processing libraries like NumPy and Pandas.
  - Enables automated report generation.

9. Minicom

- **Purpose:** Serial terminal software for monitoring UART communication and debugging.
- **Features:**
  - Provides real-time data logging.
  - Supports SSH and Telnet for remote access.
  - Simple user interface for serial communication.

10. Operating System

- **Purpose:** Provides a development environment for software installation and coding.
- **Features:**
  - Compatible with **Windows**, **Linux**, or **macOS**.
  - Supports development tools like STM32CubeIDE and Python.

3.3 SYSTEM DESIGN

The system design of the Smart Meter with Gateway Communication outlines how various components work together to collect sensor data, process it, transmit it to the cloud, and visualize it for monitoring and analysis. The design is divided into different modules, including hardware design, communication design, cloud integration, and user interface design.

3.3.1 Overview of System Desig

The Industrial IoT Gateway system consists of three main layers:

1. **Sensing Layer:** This layer consists of sensors like the ZMPT101B and ACS712 that collect real-time data on Voltage and Current. The data is transmitted to the STM32 Nucleo F446RE microcontroller using I2C communication.

2. **Processing Layer:** The STM32 processes the received data, applies necessary algorithms for data validation and preprocessing, and prepares it for transmission.
3. **Communication Layer:** Data is transmitted using USART wired communication to A5D2X board And A5D2X to Cloud By Ethernet
4. **Cloud Layer:** The data is sent to ThingsBoard, which provides storage, analysis, and visualization using customizable dashboards.
5. **User Interface Layer:** The data is displayed on an LCD screen for local monitoring and accessible remotely through the ThingsBoard web dashboard.

3.3.2 Working Process

1. **Data Collection:**
    - The zmpt101b and acs712 sensors continuously monitors Voltage and Current.
    - It transmits data to the STM32 Nucleo F446RE via the I2C protocol.
2. **Data Processing:**
    - The STM32 processes the sensor data to remove noise and apply necessary calibrations.
3. **Data Transmission:**
    - For long-range data transmission, Ethernet is used to send data directly to ThingsBoard.
4. **Cloud Integration:**
    - ThingsBoard receives the sensor data using the MQTT protocol.
    - The data is stored, analyzed, and visualized using dashboards and graphs.
5. **Data Visualization:**
    - The user can monitor sensor readings locally on the LCD display.
    - Remote monitoring is enabled using the ThingsBoard web interface, which also provides notifications and alerts in case of anomalies.

3.4 MODEL DEVELOPMENT

The model development phase involves the systematic design, implementation, and integration of various components to build a fully functional Smart Meter with Gateway Communication. The development process is divided into hardware interfacing, firmware programming, communication setup, cloud integration, and testing.

3.4.1 Hardware Interfacing

The first step in model development is assembling and interfacing the hardware components.

- **STM32 Nucleo F446RE:** Acts as the core processing unit. It reads data from the zmpt101b and acs712 sensor via ADC

- **ZMPT101b and ACS712 Sensor:** Measures Voltage and Current values and sends the data to the microcontroller using ADc communication

- **A5D2X :** Recive values from STM32 by USART communication

- **Ethernet Module:** Enables long-range data transmission directly to the cloud using MQTT.
- **LCD Display:** Displays real-time sensor data for local monitoring.

Hardware Connections:

- ZMBT101B and ACS712 sensors connected to STM32 via ADC
- STM32 connected to A5D2X using UART.
- Ethernet module connected to A5D2X2 Ethernet pins.
- LCD connected via I2C using GPIO pins.

3.4.2 Firmware Development

The firmware for STM32 is developed using **STM32CubeIDE**. The primary tasks include:

- **Sensor Data Acquisition:** Implementing ADC driver functions to read data from the ZMPT101B and ACS712
- **Data Processing:** Filtering and formatting sensor data for transmission.
- **USART Communication:** Using UART protocols to sending data via wired
- **Ethernet Communication:** Configuring MQTT protocols for cloud data transfer using ThingsBoard.
- **LCD Display Control:** Displaying real-time sensor readings using appropriate driver functions.

Programming Languages: C and Embedded C

3.4.3 Communication Setup

- **USART Communication:** The USART Communication module is configured Data is transmitted using USART.
- **Ethernet Communication:** Ethernet connectivity is used to send data to the cloud platform through the MQTT protocol. The data is sent securely using authentication mechanisms.
- **MQTT Protocol:** Data is published to specific topics on ThingsBoard, and alerts are generated based on threshold values.

3.4.4 Cloud Integration

- The ThingsBoard platform is used for storing, visualizing, and analyzing sensor data.
- Data is received via MQTT using a unique device token.
- Real-time dashboards display Voltage and Current.
- Alerts and notifications are configured to trigger in case of abnormal sensor readings.

3.4.5 Testing and Debugging

- **Unit Testing:** Each module (sensors, Ethernet, and cloud) is tested individually to ensure proper functionality.

- **Integration Testing:** The entire system is tested to validate data flow from sensor to cloud

- **Performance Testing:** Response time, data accuracy, and transmission reliability are evaluated.**Error Handling:** Error detection mechanisms are implemented to identify faulty sensors, communication failures, or cloud connectivity issues.

3.5 PERIPHERAL EXPLAINATION

In the Industrial IoT Gateway, various peripherals are used to perform specific functions like data acquisition, communication, processing, and display. Each peripheral is carefully selected to ensure efficient and reliable operation. This section explains the role and functionality of each peripheral in the system.

3.5.1 STM32 Nucleo F446RE

- **Role:** Acts as the main microcontroller for processing and managing the entire system.

- **Functionality:**
  - Collects data from sensors using ADC communication.
  - Processes sensor data to filter noise and perform necessary calculations.
  - Transmits data using USART to the Rugged Board A5D2X And directly to the cloud.
  - Controls the LCD for displaying sensor readings.
- **Key Features:**
  - ARM Cortex-M4 processor.
  - Multiple GPIO pins for peripheral interfacing.
  - Supports I2C, UART, and SPI communication.

3.5.2 ZMPT101B Voltage Sensor

- **Role**: Measures AC voltage in electrical systems.
- **Functionality**:
  - Uses an op-amp and transformer-based circuit for voltage measurement.
  - Provides isolation between high voltage and microcontroller.
  - Outputs an analog signal proportional to the measured voltage.
- **Key Features**:
  - Measurement Range: 0V to 250V AC.
  - High accuracy and stability.
  - Compact design for easy integration.

3.5.3 ACS712 Current Sensor

- **Role**: Measures AC and DC current in electrical circuits.
- **Functionality**:

- Uses Hall-effect technology for non-intrusive current measurement.
- Provides an analog voltage output proportional to the current.
- Can measure both positive and negative currents.

➢ **Key Features**:

- Measurement Range: ±5A, ±20A, or ±30A (depending on variant).
- Low resistance path for minimal power loss.
- High sensitivity and isolation from high voltage circuits.

3.5.4 Rugged Board A5D2X

➢ **Role:** Acts as a data receiver and edge processing device.

➢ **Functionality:**

- Receives sensor data from the STM32 using
- Processes data locally to perform initial analysis.
- Forwards data to the cloud for further monitoring and storage.

➢ **Key Features:**

- Industrial-grade design.
- Supports Ethernet and wireless communication.

3.5.5 Ethernet Module

➢ **Role:** Provides long-range wired communication.

➢ **Functionality:**

- Establishes a stable connection to the cloud using the MQTT protocol.
- Ensures secure and fast data transfer.
- Ideal for industrial environments requiring reliable connectivity.

➢ **Key Features:**

- Supports 10/100 Mbps Ethernet.
- Low power consumption.

3.5.6 LCD Display

- **Role:** Displays real-time sensor data for local monitoring.
- **Functionality:**
  - Connected to STM32 using I2C communication.
  - Displays temperature, humidity, and system status messages.
- **Key Features:**
  - Clear visibility with adjustable brightness.
  - Supports 16x2 or 20x4 character display.

3.5.7 Power Supply

Role: Provides the necessary power to all components.

- **Functionality:**
  - Supplies 5V and 3.3V to the STM32, sensors, and other modules.
  - Ensures stable and reliable power delivery.
- **Key Features:**
  - Overvoltage and short-circuit protection.

3.6. DATA FLOW

A Data Flow Diagram (DFD) provides a visual representation of the flow of data within a system. It outlines the sources of data input, destinations of data output, the transformation processes it undergoes, and where the data is stored. DFDs focus on the logical flow of information rather than operational details like timing, sequence, or parallelism. They are crucial in understanding and analyzing the functional aspects of information systems.

External Entity

An external entity can represent a human, system or subsystem. It is where certain data comes from or goes to. It is external to the system we study, in terms of the

business process. For this reason, people used to draw external entities on the edge of a diagram.

Process

A process is a business activity or function where the manipulation and transformation of data takes place. A process can be decomposed to finer level of details, for representing how data is being processed within the process.

Data Store

A data store represents the storage of persistent data required and/or produced by theprocess. Here are some examples of data stores: membership forms, database table.

Data Flow

A data flow represents the flow of information, with its direction represented by an arrow head that shows at the end(s) of flow connector.

Data flow diagram

Level 0:

Figure 3.6.1.Smart Meter System Operation

Level 1:

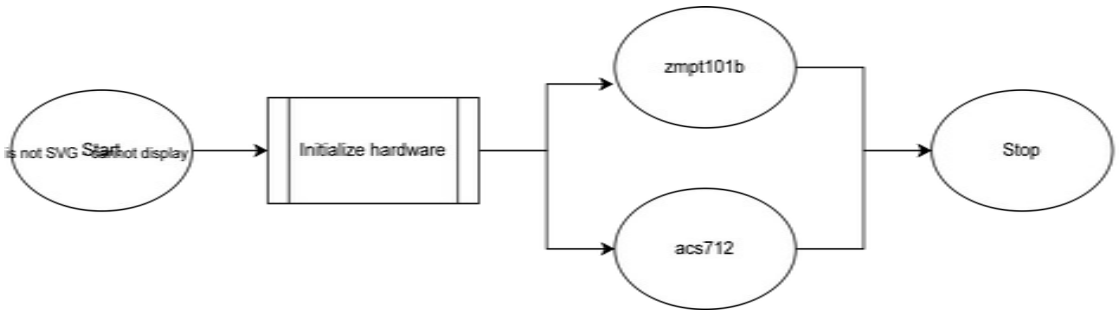

Figure 3.6.2. Sensor Data Acquisition Flowchart,STM32 Data Processing

Level 2:

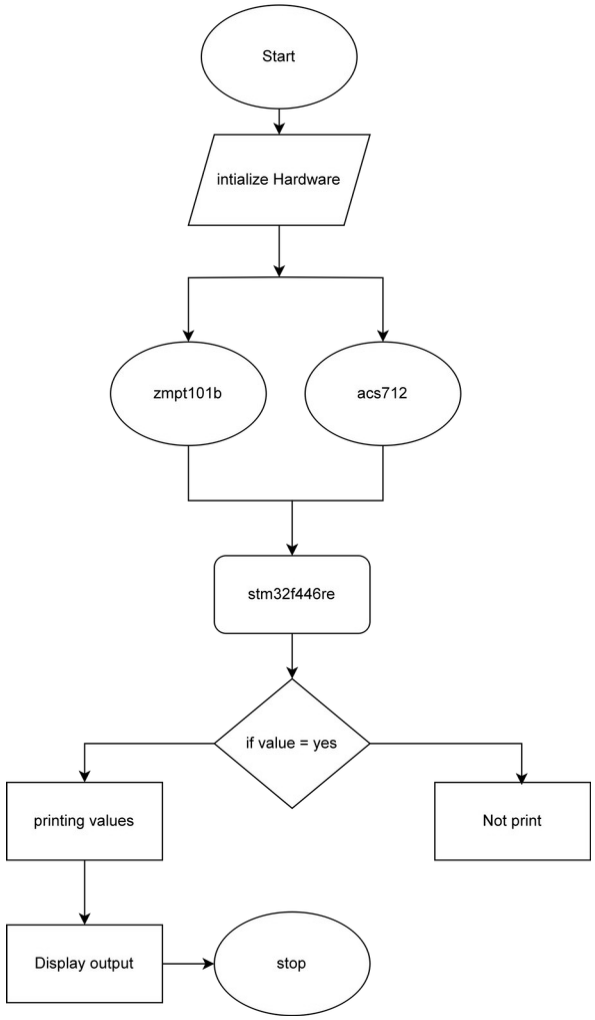

Figure 3.6.3. Representing Cloud-Based Data Visualization, Analytics, and Alert Generation

Level 3:

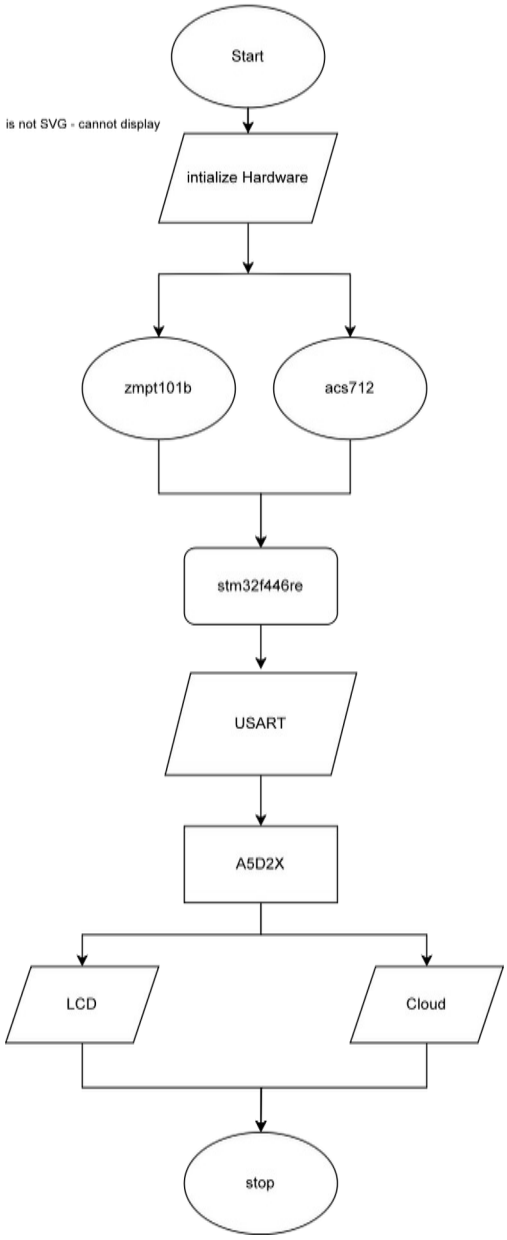

Figure 3.6.4. End-to-End Data Flow of the Smart Energy Meter Gateway from Sensor Acquisition to Cloud Monitoring and Alerting

CHAPTER -4
PROPOSED SYSTEM

PROPOSED SYSTEM

The proposed system is an Smart meter with Gateway Communication that enables real-time monitoring of environmental parameters using sensors, microcontrollers, and cloud integration. The system efficiently collects sensor data, processes it, transmits it wirelessly or through Ethernet, and visualizes it on a cloud platform using ThingsBoard. This allows industries to monitor and analyze data remotely, ensuring predictive maintenance and operational efficiency.

4.1 ALGORITHM

The algorithm for the Smart Meter with Gateway Communication consists of the following steps:

Step 1: Initialization

- Initialize the STM32 Nucleo F446RE microcontroller.
- Configure ADC, USART
- Initialize the ZMPT101B and ACS712 sensors for Volatage and Current data collection.
- Declaring GPIO pins For Data Transmission

Step 2: Sensor Data Acquisition

- Read real-time Voltage and Current data from the ZMPT101B and ACS712 sensors using ADC communication.
- Store the sensor data in variables for processing.

Step 3: Data Processing

- Perform basic validation to ensure data accuracy.
- Apply filtering algorithms to remove noise.
- Format the data for transmission using a suitable protocol (MQTT).

Table 4.1. Summary of Literature Survey on Smart Meters and Gateway Communication in IoT

Sl. No.	Author(s) & Year	Focus Area	Key Contributions / Findings
1	Smith et al. (2023)	IoT Gateway & Edge Computing	Developed intelligent IoT gateway with edge computing, enabling faster decision-making.
2	Johnson and Lee (2022)	IoT Gateway Architecture	Proposed multi-layer gateway using BLE & Ethernet for efficient low-latency industrial data.
3	Patel et al. (2022)	Sensor-Based Smart Metering	Used STM32 microcontrollers, reducing industrial energy wastage by 30%.
4	Garcia and Wang (2021)	Sensor Accuracy & Calibration	Emphasized calibration techniques to enhance sensor precision in energy monitoring.
5	Singh and Sharma (2023)	Communication Protocols in IIoT	Compared MQTT, CoAP, HTTP; MQTT found most suitable for large-scale IIoT deployments.
6	Wang and Li (2022)	Short-Range Communication (BLE)	Demonstrated BLE's efficiency in industrial environments with low interference.
7	Chen et al. (2023)	Cloud Integration & Visualization	Implemented ThingsBoard-based system; improved monitoring accuracy by 35%.
8	Lee and Johnson (2022)	Cloud-Based Anomaly Detection	Designed cloud-integrated IIoT for anomaly detection; reduced energy losses by 40%.
9	Smith et al. (2023)	Predictive Maintenance Using ML	Used ML for analyzing real-time meter data; reduced system downtime by 50%.

Step 4: Communication

- **Option 1:** For short-range communication, transmit data using USART
- **Option 2:** For long-range communication, send the data using the Ethernet module to the ThingsBoard cloud platform.

Step 5: Data Visualization

- Display real-time sensor values on the LCD screen for local monitoring.
- Send data to ThingsBoard for remote monitoring using customized dashboards.

Step 6: Alert Generation

- Continuously monitor data for abnormal values using threshold conditions.
- Generate alerts and notifications through ThingsBoard if necessary.

Step 7: Data Logging

- Log data at regular intervals for further analysis and predictive maintenance.

Step 8: Error Handling

- Implement error detection mechanisms to handle sensor failures, communication issues, or data transmission errors.

Step 9: Repeat

- Continuously repeat steps 2 to 8 in a loop for real-time monitoring.

CHAPTER 5
RESULTS AND DISCUSSION

The Figure 5.1 shows the STM32CubeMX tool, a graphical configuration and code generation utility provided by STMicroelectronics, being used to configure the ADC (Analog-to-Digital Converter) on the STM32F446RE microcontroller. In the **Pinout & Configuration** tab, several ADC input channels (IN0, IN1, IN4, IN5) are selected under **ADC1**, indicating the microcontroller will be sampling analog signals from these pins.

The right side of the screen displays the **pin diagram** of the STM32F446RE in LQFP64 package format, with active pins highlighted in green, reflecting their assigned functionalities. The configuration supports projects like smart metering or energy monitoring, where multiple analog signals must be read and processed in real-time. The tool simplifies the initialization process by auto-generating boilerplate code for the STM32CubeIDE, streamlining embedded development.

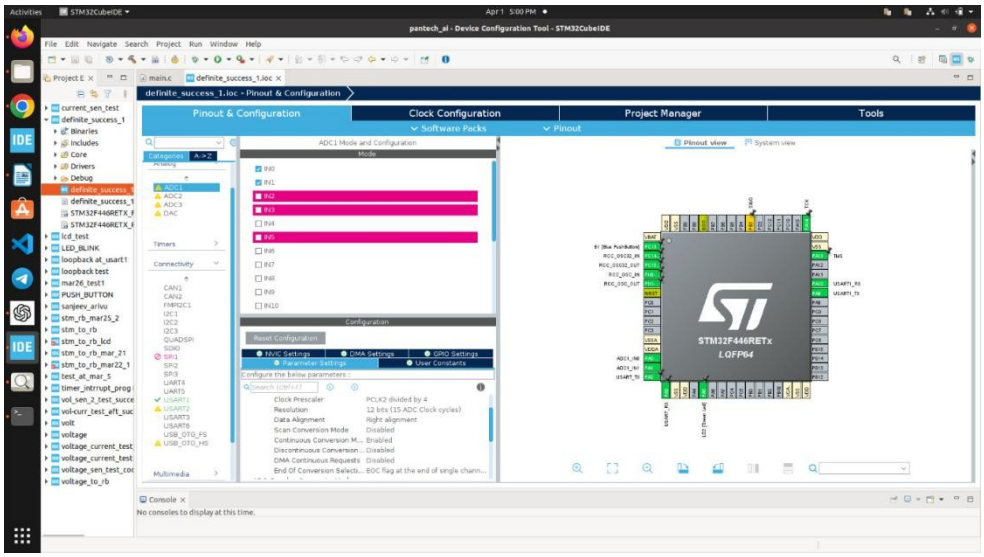

Figure 5.1 STM32CubeMX Configuration for ADC on STM32F446RE

The Figure 5.2 shows real-time voltage and current data received from the STM32F446RE via a Python script on the RuggedBoard A5D2X (left), and the corresponding firmware code running on STM32CubeIDE (right). The microcontroller reads sensor data (from ZMPT101B and ACS712) and sends it via USART. This setup demonstrates a smart energy monitoring system with live data acquisition and serial communication.

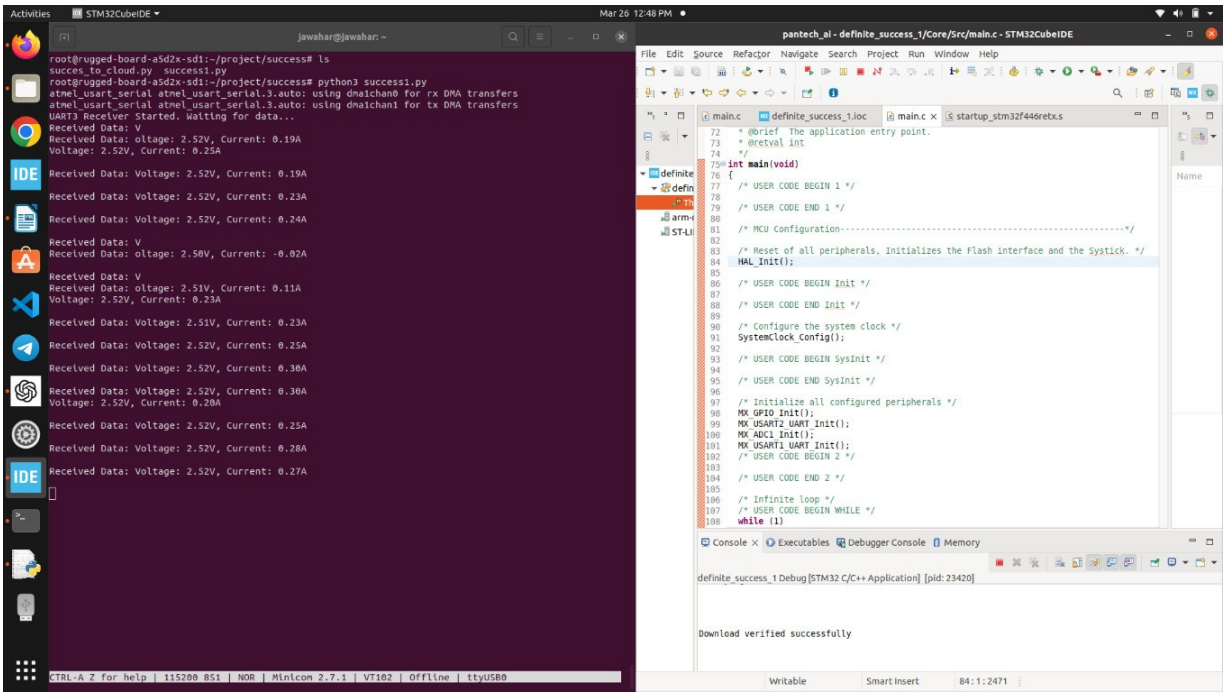

Figure 5.2. Live Data Monitoring and Firmware Execution for Smart Meter using STM32 and RuggedBoard A5D2X

The Figure 5.3 displays the physical setup of a smart meter system prototype. The components include an STM32F446RE Nucleo board connected to a RuggedBoard A5D2X, with sensors like the ZMPT101B voltage sensor and ACS712 current sensor interfaced. An I2C-based 16x2 LCD shows real-time voltage and current values for local monitoring. Multiple jumper wires and a breadboard indicate sensor interfacing and signal routing. Red LEDs on the boards confirm active power and communication. This setup illustrates the integration of embedded hardware for real-time energy data acquisition, processing, and display in a smart metering application.

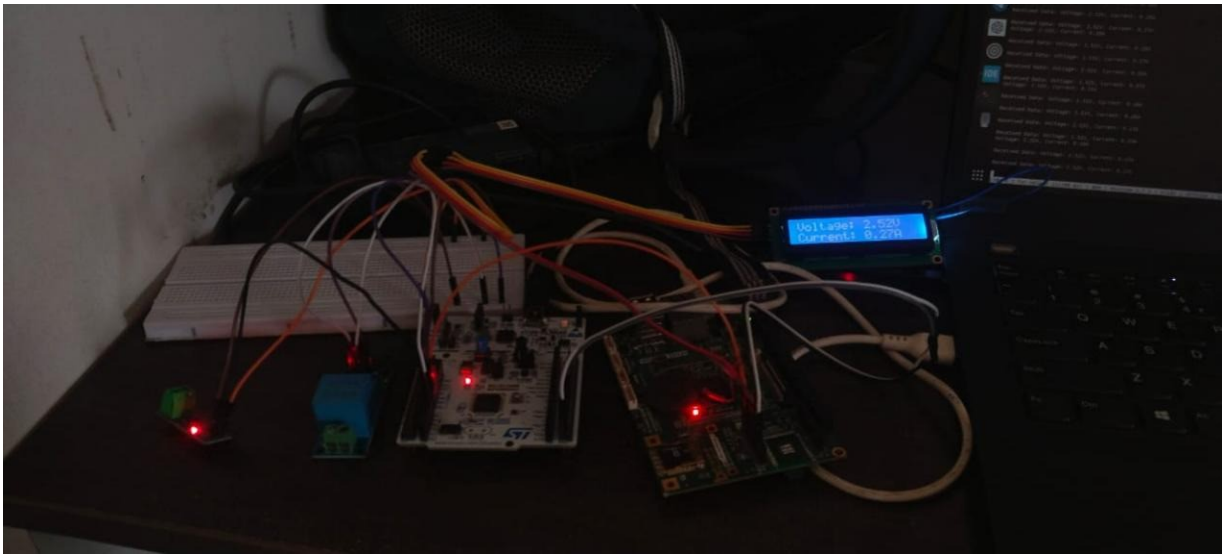

Figure 5.3 Hardware Setup of Smart Energy Monitoring System with STM32 and RuggedBoard A5D2X

The Figure 5.4 shows a laptop screen displaying real-time energy monitoring data received via serial communication and visualized on a cloud-based IoT dashboard. The terminal (left) logs voltage and current values from the STM32F446RE system, while the open dashboard (right) titled "smart_meter_gateway_24" presents a live line chart showing power consumption trends. This setup confirms successful MQTT-based data transmission from the RuggedBoard A5D2X to the cloud, enabling users to monitor energy metrics remotely and in real-time. It reflects the system's capability for remote analytics and integration with smart energy management platforms.

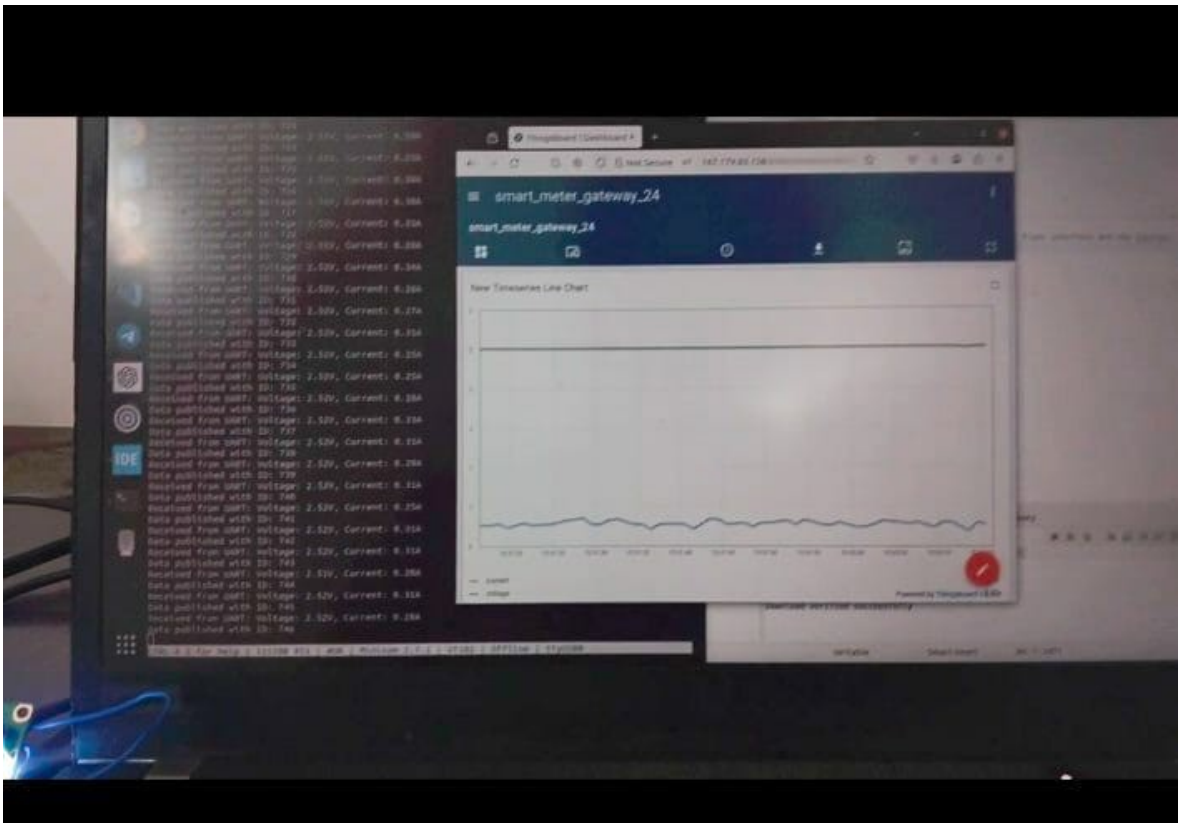

Figure 5.4. Cloud-Based Visualization of Smart Meter Data using IoT Dashboard

CHAPTER -6
CONCLUSION AND FUTURE ENHANCEMENT

CONCLUSION

The Smart Meter with Gateway Communication is an efficient, scalable solution for real-time energy monitoring and management, integrating ZMPT101B and ACS712 sensors with an STM32F446RE microcontroller for accurate voltage and current measurement. Data is transmitted to the A5D2X rugged board via USART, with a local LCD display for real-time monitoring. Designed for both industrial and residential use, it supports cloud connectivity for remote access, analytics, and alert generation through secure MQTT communication. With predictive maintenance capabilities and Industry 4.0 alignment, the system promotes energy efficiency, reduces downtime, and supports sustainable energy management.

FUTURE ENHANCEMENT

The Smart Meter with Gateway Communication is a comprehensive solution for real-time voltage and current monitoring. However, to further enhance its functionality and efficiency, the following future improvements can be considered:

1. AI-Powered Predictive Analytics

- Implement advanced Artificial Intelligence (AI) and Machine Learning (ML) algorithms to analyze historical data.
- Provide predictive insights to detect equipment failures and optimize maintenance schedules.
- Enhance anomaly detection with AI models that continuously improve over time.

2. Edge Computing

- Integrate edge computing capabilities to process data locally on the STM32 Nucleo F446RE or Rugged Board A5D2X.

- Reduce latency by performing real-time analysis and decision-making at the edge before transmitting data to the cloud.

3. Multi-Sensor Integration

- Expand the system to support a variety of industrial sensors, including vibration, pressure, gas, and proximity sensors.
- Enable comprehensive monitoring for complex industrial processes.

4. Enhanced Security Features

- Implement advanced security protocols such as data encryption, secure boot, and device authentication.
- Utilize blockchain technology for secure and transparent data logging.

5. Remote Control Capabilities

- Develop remote control functionalities to allow users to adjust system settings and control connected devices through ThingsBoard.
- Implement automated responses based on sensor data, such as triggering emergency shutoffs or activating cooling systems.

6. Energy Management System

- Integrate energy management algorithms to optimize energy consumption in industries.
- Provide energy usage reports and recommendations for energy-saving practices.

7. Mobile Application Support

- Develop a mobile application for real-time monitoring and notifications.
- Provide remote access to sensor data, alerts, and system diagnostics.

8. Scalability for Large-Scale Deployments

- Design the gateway to manage large-scale sensor networks efficiently.

- Implement data aggregation techniques to consolidate data from multiple gateways.

9. Integration with Other Platforms

- Enable interoperability with other Industrial IoT platforms through standardized APIs.
- Support integration with enterprise resource planning (ERP) and manufacturing execution systems (MES) for complete industrial automation.

10. Advanced Visualization Tools

- Develop customizable dashboards with augmented reality (AR) or virtual reality (VR) support.
- Provide immersive data visualization for better situational awareness in control rooms.

BIBLIOGRAPHY

The following sources were referred to during the research, design, and development of the Smart meter With Gateway Communication

Books and Textbooks:

- ahga, A., & Madisetti, V. (2019). *Internet of Things: A Hands-On Approach*. Universities Press.
- Buyya, R., & Dastjerdi, A. V. (2021). *Internet of Things: Principles and Paradigms*. Elsevier.
- Raj, P., & Raman, A. C. (2020). *The Internet of Things: Enabling Technologies, Platforms, and Use Cases*. CRC Press.

Journals and Research Papers:

- Kumar, P., & Sharma, R. (2023). *Secure Communication in IoT-Based Smart Metering Systems Using MQTT and Edge Computing*. International Journal of Computer Science & Network Security.

- Singh, R., & Agarwal, N. (2023). *Implementation of Wireless Sensor Networks in Smart Energy Metering Using STM32 Microcontrollers*. IEEE Transactions on Industrial Informatics.

Web Resources:

- STMicroelectronics. STM32 Nucleo F446RE Datasheet and User Guide. Retrieved from: https://www.st.com

- ThingsBoard. IoT Platform for Data Collection and Visualization. Retrieved from: https://thingsboard.io

1. **Conference Proceedings:**

- International Conference on Industrial Internet of Things and Smart Manufacturing (IIOTSM).
- IEEE International Conference on Internet of Things (ICIoT).

2. **Datasheets and Technical Manuals:**

- **ZMPT101B Voltage Sensor Datasheet** – Available at wwanalog.com
- **ACS712 Current Sensor Datasheet** – Available at www.allegromicro.com

- **STM32 Nucleo F446RE Technical Reference Manual** – Available at www.st.com

- **Rugged Board (A5D2X) Datasheet** – Available at https://developer.ruggedboard.com/

3. **Software Documentation:**
    - STM32CubeIDE User Guide.
    - Ruggerd Board User Guide
    - MQTT Protocol Documentation by OASIS.
    - ThingsBoard API Documentation.

1. STM32 Nucleo F446RE Tutorials

- www.youtube.com/watch?v=8S78Ih4SaiE
- www.youtube.com/watch?v=rfBeq-Fu0hc

2. A5D2X Rugged Board Information

- www.ruggedboard.com/product/ruggedboard-a5d2x

3. ThingsBoard IoT Platform

- www.thingsboard.io
- https://ieeexplore.ieee.org/document/9936826
- https://www.datasheethub.com/fc-28-soil-moisture-sensor-module/#google_vignette

APPENDIX

Coding:

```
/* USER CODE BEGIN Header */
  * @file           : main.c
  * @brief          : Main program body
  ******************************************************************************
  * @attention
  * Copyright (c) 2025 STMicroelectronics.
  * All rights reserved.
  * This software is licensed under terms that can be found in the LICENSE file
  * in the root directory of this software component.
  * If no LICENSE file comes with this software, it is provided AS-IS.
/* USER CODE END Header */
/* Includes ------------------------------------------------------------------*/
#include "main.h"
#include <string.h>
#include <stdio.h>
#include "stm32f4xx_hal.h"
/* Private includes ----------------------------------------------------------*/
/* USER CODE BEGIN Includes */
```

```
/* USER CODE END Includes */
/* Private typedef -----------------------------------------------------------*/
/* USER CODE BEGIN PTD */
/* USER CODE END PTD */
/* Private define ------------------------------------------------------------*/
/* USER CODE BEGIN PD */
/* USER CODE END PD */
/* Private macro -------------------------------------------------------------*/
/* USER CODE BEGIN PM */
/* USER CODE END PM */
/* Private variables ---------------------------------------------------------*/
ADC_HandleTypeDef hadc1;
UART_HandleTypeDef huart1;
UART_HandleTypeDef huart2;
/* USER CODE BEGIN PV */
char buffer[100];
uint32_t adc_value;
float voltage, current;
/* USER CODE END PV */
/* Private function prototypes -----------------------------------------------*/
void SystemClock_Config(void);
static void MX_GPIO_Init(void);
static void MX_USART2_UART_Init(void);
static void MX_ADC1_Init(void);
static void MX_USART1_UART_Init(void);
```

```
/* USER CODE BEGIN PFP */
/* USER CODE END PFP */
/* Private user code ---------------------------------------------------------*/
/* USER CODE BEGIN 0 */
/* USER CODE END 0 */
/**
  * @brief  The application entry point.
  * @retval int
  */
int main(void)
{
  /* USER CODE BEGIN 1 */
  /* USER CODE END 1 */
  /* MCU Configuration--------------------------------------------------------*/
  /* Reset of all peripherals, Initializes the Flash interface and the Systick. */
  HAL_Init();
  /* USER CODE BEGIN Init */
  /* USER CODE END Init */
  /* Configure the system clock */
  SystemClock_Config();
  /* USER CODE BEGIN SysInit */
  /* USER CODE END SysInit */
  /* Initialize all configured peripherals */
  MX_GPIO_Init();
  MX_USART2_UART_Init();
```

```
 MX_ADC1_Init();
 MX_USART1_UART_Init();
 /* USER CODE BEGIN 2 */
 /* USER CODE END 2 */
 /* Infinite loop */
 /* USER CODE BEGIN WHILE */
 while (1)
 {
  /* USER CODE END WHILE */
        HAL_ADC_Start(&hadc1);
           HAL_ADC_PollForConversion(&hadc1, HAL_MAX_DELAY);
           adc_value = HAL_ADC_GetValue(&hadc1);
             // Convert ADC Value to Voltage (Assuming 3.3V reference and 12-bit ADC)
           voltage = (adc_value * 3.3) / 4095.0;
           // Start ADC conversion for Current Sensor (ACS712)
           HAL_ADC_PollForConversion(&hadc1, HAL_MAX_DELAY);
           adc_value = HAL_ADC_GetValue(&hadc1);
               // Convert ADC Value to Current (ACS712 30A version, 66mV/A sensitivity)
           float sensor_voltage = (adc_value * 3.3) / 4095.0;
           current = (sensor_voltage - 2.5) / 0.066;
           // Format and Send Voltage and Current via USART1
           snprintf(buffer, sizeof(buffer), "Voltage: %.2fV, Current: %.2fA\n", voltage, current);
                      HAL_UART_Transmit(&huart1,  (uint8_t*)buffer,  strlen(buffer),
```

```
HAL_MAX_DELAY);  // Corrected function
            HAL_Delay(1000);
   /* USER CODE BEGIN 3 */
  }
  /* USER CODE END 3 */
}
/**
  * @brief System Clock Configuration
  * @retval None
  */
void SystemClock_Config(void)
{
  RCC_OscInitTypeDef RCC_OscInitStruct = {0};
  RCC_ClkInitTypeDef RCC_ClkInitStruct = {0};
  /** Configure the main internal regulator output voltage
  */
                                        __HAL_RCC_PWR_CLK_ENABLE();
__HAL_PWR_VOLTAGESCALING_CONFIG(PWR_REGULATOR_VOLTAGE_S
CALE3);
  /** Initializes the RCC Oscillators according to the specified parameters
  * in the RCC_OscInitTypeDef structure.
  */
  RCC_OscInitStruct.OscillatorType = RCC_OSCILLATORTYPE_HSI;
  RCC_OscInitStruct.HSIState = RCC_HSI_ON;
  RCC_OscInitStruct.HSICalibrationValue = RCC_HSICALIBRATION_DEFAULT;
  RCC_OscInitStruct.PLL.PLLState = RCC_PLL_ON;
```

```
  RCC_OscInitStruct.PLL.PLLSource = RCC_PLLSOURCE_HSI;
  RCC_OscInitStruct.PLL.PLLM = 16;
  RCC_OscInitStruct.PLL.PLLN = 336;
  RCC_OscInitStruct.PLL.PLLP = RCC_PLLP_DIV4;
  RCC_OscInitStruct.PLL.PLLQ = 2;
  RCC_OscInitStruct.PLL.PLLR = 2;
  if (HAL_RCC_OscConfig(&RCC_OscInitStruct) != HAL_OK)
  {
    Error_Handler();
  }
  /** Initializes the CPU, AHB and APB buses clocks
  */
                RCC_ClkInitStruct.ClockType      = RCC_CLOCKTYPE_HCLK|
RCC_CLOCKTYPE_SYSCLK
                        |RCC_CLOCKTYPE_PCLK1|RCC_CLOCKTYPE_PCLK2;
  RCC_ClkInitStruct.SYSCLKSource = RCC_SYSCLKSOURCE_PLLCLK;
  RCC_ClkInitStruct.AHBCLKDivider = RCC_SYSCLK_DIV1;
  RCC_ClkInitStruct.APB1CLKDivider = RCC_HCLK_DIV2;
  RCC_ClkInitStruct.APB2CLKDivider = RCC_HCLK_DIV1;
if    (HAL_RCC_ClockConfig(&RCC_ClkInitStruct,    FLASH_LATENCY_2)    !=
HAL_OK)
  {
    Error_Handler();
  }
}
/**
```

```
  * @brief ADC1 Initialization Function
  * @param None
  * @retval None
  */
static void MX_ADC1_Init(void)
{
  /* USER CODE BEGIN ADC1_Init 0 */
  /* USER CODE END ADC1_Init 0 */
  ADC_ChannelConfTypeDef sConfig = {0};
  /* USER CODE BEGIN ADC1_Init 1 */
  /* USER CODE END ADC1_Init 1 */
  /** Configure the global features of the ADC (Clock, Resolution, Data Alignment and number of conversion)
  */
  hadc1.Instance = ADC1;
  hadc1.Init.ClockPrescaler = ADC_CLOCK_SYNC_PCLK_DIV4;
  hadc1.Init.Resolution = ADC_RESOLUTION_12B;
  hadc1.Init.ScanConvMode = DISABLE;
  hadc1.Init.ContinuousConvMode = ENABLE;
  hadc1.Init.DiscontinuousConvMode = DISABLE;
  hadc1.Init.ExternalTrigConvEdge = ADC_EXTERNALTRIGCONVEDGE_NONE;
  hadc1.Init.ExternalTrigConv = ADC_SOFTWARE_START;
  hadc1.Init.DataAlign = ADC_DATAALIGN_RIGHT;
  hadc1.Init.NbrOfConversion = 1;
  hadc1.Init.DMAContinuousRequests = DISABLE;
  hadc1.Init.EOCSelection = ADC_EOC_SINGLE_CONV;
```

```
  if (HAL_ADC_Init(&hadc1) != HAL_OK)
  {
    Error_Handler();
  }
   /** Configure for the selected ADC regular channel its corresponding rank in the sequencer and its sample time.
  */
  sConfig.Channel = ADC_CHANNEL_0;
  sConfig.Rank = 1;
  sConfig.SamplingTime = ADC_SAMPLETIME_3CYCLES;
  if (HAL_ADC_ConfigChannel(&hadc1, &sConfig) != HAL_OK)
  {
    Error_Handler();
  }
  /* USER CODE BEGIN ADC1_Init 2 */
  /* USER CODE END ADC1_Init 2 */
}
/**
  * @brief USART1 Initialization Function
  * @param None
  * @retval None
  */
static void MX_USART1_UART_Init(void)
{
  /* USER CODE BEGIN USART1_Init 0 */
  /* USER CODE END USART1_Init 0 */
```

```
  /* USER CODE BEGIN USART1_Init 1 */
  /* USER CODE END USART1_Init 1 */
  huart1.Instance = USART1;
  huart1.Init.BaudRate = 115200;
  huart1.Init.WordLength = UART_WORDLENGTH_8B;
  huart1.Init.StopBits = UART_STOPBITS_1;
  huart1.Init.Parity = UART_PARITY_NONE;
  huart1.Init.Mode = UART_MODE_TX_RX;
  huart1.Init.HwFlowCtl = UART_HWCONTROL_NONE;
  huart1.Init.OverSampling = UART_OVERSAMPLING_16;
  if (HAL_UART_Init(&huart1) != HAL_OK)
  {
    Error_Handler();
  }
  /* USER CODE BEGIN USART1_Init 2 */
  /* USER CODE END USART1_Init 2 */
}
/**
  * @brief USART2 Initialization Function
  * @param None
  * @retval None
  */
static void MX_USART2_UART_Init(void)
{
  /* USER CODE BEGIN USART2_Init 0 */
```

```
  /* USER CODE END USART2_Init 0 */
  /* USER CODE BEGIN USART2_Init 1 */
  /* USER CODE END USART2_Init 1 */
  huart2.Instance = USART2;
  huart2.Init.BaudRate = 115200;
  huart2.Init.WordLength = UART_WORDLENGTH_8B;
  huart2.Init.StopBits = UART_STOPBITS_1;
  huart2.Init.Parity = UART_PARITY_NONE;
  huart2.Init.Mode = UART_MODE_TX_RX;
  huart2.Init.HwFlowCtl = UART_HWCONTROL_NONE;
  huart2.Init.OverSampling = UART_OVERSAMPLING_16;
  if (HAL_UART_Init(&huart2) != HAL_OK)
  {
    Error_Handler();
  }
  /* USER CODE BEGIN USART2_Init 2 */
  /* USER CODE END USART2_Init 2 */
}
/**
  * @brief GPIO Initialization Function
  * @param None
  * @retval None
  */
static void MX_GPIO_Init(void)
{
```

```
  GPIO_InitTypeDef GPIO_InitStruct = {0};
  /* GPIO Ports Clock Enable */
  __HAL_RCC_GPIOC_CLK_ENABLE();
  __HAL_RCC_GPIOH_CLK_ENABLE();
  __HAL_RCC_GPIOA_CLK_ENABLE();
  __HAL_RCC_GPIOB_CLK_ENABLE();
  /*Configure GPIO pin Output Level */
  HAL_GPIO_WritePin(LD2_GPIO_Port, LD2_Pin, GPIO_PIN_RESET);
  /*Configure GPIO pin : B1_Pin */
  GPIO_InitStruct.Pin = B1_Pin;
  GPIO_InitStruct.Mode = GPIO_MODE_IT_FALLING;
  GPIO_InitStruct.Pull = GPIO_NOPULL;
  HAL_GPIO_Init(B1_GPIO_Port, &GPIO_InitStruct);
  /*Configure GPIO pin : LD2_Pin */
  GPIO_InitStruct.Pin = LD2_Pin;
  GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
  GPIO_InitStruct.Pull = GPIO_NOPULL;
  GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
  HAL_GPIO_Init(LD2_GPIO_Port, &GPIO_InitStruct);
}
/* USER CODE BEGIN 4 */
/* USER CODE END 4 */
void Error_Handler(void)
{
  /* USER CODE BEGIN Error_Handler_Debug */
```

```
  /* User can add his own implementation to report the HAL error return state */
  __disable_irq();
  while (1)
  {
  }
  /* USER CODE END Error_Handler_Debug */
}

#ifdef  USE_FULL_ASSERT
/**
  * @brief  Reports the name of the source file and the source line number
  *         where the assert_param error has occurred.
  * @param  file: pointer to the source file name
  * @param  line: assert_param error line source number
  * @retval None
  */
void assert_failed(uint8_t *file, uint32_t line)
{
  /* USER CODE BEGIN 6 */
  /* User can add his own implementation to report the file name and line number,
     ex: printf("Wrong parameters value: file %s on line %d\r\n", file, line) */
  /* USER CODE END 6 */
}
#endif /* USE_FULL_ASSERT */
```

```
import paho.mqtt.client as mqtt
import serial
import time
from smbus2 import SMBus
# Constants
THINGSBOARD_HOST = '167.179.83.158'
ACCESS_TOKEN = 'lnL2jGaehkHK5jC8Ge1m'  # Replace with your actual access token
SERIAL_PORT = '/dev/ttyS3'
BAUD_RATE = 115200
 # I2C LCD Constants
I2C_ADDR = 0x27  # I2C address of the LCD (may vary depending on your LCD)
LCD_WIDTH = 16   # Max characters per line on the LCD
LCD_CHR = 1      # Mode - Sending data
LCD_CMD = 0      # Mode - Sending command
LCD_LINE_1 = 0x80  # LCD RAM address for the 1st line
LCD_LINE_2 = 0xC0  # LCD RAM address for the 2nd line
LCD_BACKLIGHT = 0x08  # On
# Initialize I2C (SMBus)
bus = SMBus(0)
# LCD Functions
def lcd_init():
    """Initialize LCD display"""
    lcd_byte(0x33, LCD_CMD)  # Initialize
    lcd_byte(0x32, LCD_CMD)  # Initialize
    lcd_byte(0x06, LCD_CMD)  # Cursor move direction
```

```
    lcd_byte(0x0C, LCD_CMD)  # Turn cursor off
    lcd_byte(0x28, LCD_CMD)  # 2 line display
    lcd_byte(0x01, LCD_CMD)  # Clear display

def lcd_byte(bits, mode):
    """Send byte to data pins (bits) via I2C"""
    bits_high = mode | (bits & 0xF0) | LCD_BACKLIGHT
    bits_low = mode | ((bits << 4) & 0xF0) | LCD_BACKLIGHT
    bus.write_byte(I2C_ADDR, bits_high)
    lcd_toggle_enable(bits_high)
    bus.write_byte(I2C_ADDR, bits_low)
    lcd_toggle_enable(bits_low)

def lcd_toggle_enable(bits):
    """Toggle enable pin on LCD"""
    time.sleep(0.0005)
    bus.write_byte(I2C_ADDR, (bits | 0b00000100))
    time.sleep(0.0005)
    bus.write_byte(I2C_ADDR, (bits & ~0b00000100))
    time.sleep(0.0005)

def lcd_string(message, line):
    """Display string on LCD at specified line"""
    message = message.ljust(LCD_WIDTH, " ")
    lcd_byte(line, LCD_CMD)
```

```
    for i in range(LCD_WIDTH):
        lcd_byte(ord(message[i]), LCD_CHR)

# MQTT Callbacks
def on_connect(client, userdata, flags, rc):
    if rc == 0:
        print("Connected to ThingsBoard")
    else:
        print("Failed to connect, return code %d" % rc)

def on_publish(client, userdata, mid):
    print("Data published with ID: %d" % mid)

# Establish MQTT connection
def mqtt_connect():
    client = mqtt.Client()
    client.username_pw_set(ACCESS_TOKEN)
    client.on_connect = on_connect
    client.on_publish = on_publish
        client.connect(THINGSBOARD_HOST, 1884, 60)  # Use port 1884 for ThingsBoard
    client.loop_start()  # Start the loop to process network traffic
    return client

# Parse voltage and current from received UART data
def parse_uart_data(data):
```

```
    try:
        # Assuming the format is "Voltage: 2.52V, Current: 0.34A"
        voltage = float(data.split('Voltage: ')[1].split('V')[0].strip())
        current = float(data.split('Current: ')[1].split('A')[0].strip())
        return voltage, current
    except Exception as e:
        print("Error parsing data: %s" % str(e))
        return None, None

# Read data from UART
def read_uart3():
    try:
        ser = serial.Serial(SERIAL_PORT, BAUD_RATE, timeout=1)
        mqtt_client = mqtt_connect()

        while True:
            if ser.in_waiting > 0:
                data = ser.readline().decode('utf-8').strip()
                print("Received from UART: %s" % data)

                # Parse the voltage and current from the UART data
                voltage, current = parse_uart_data(data)

                if voltage is not None and current is not None:
                    # Display voltage on line 1 and current on line 2
```

```
                lcd_string("Voltage: {:.2f}V".format(voltage), LCD_LINE_1)
                lcd_string("Current: {:.2f}A".format(current), LCD_LINE_2)

                # Prepare telemetry data for ThingsBoard
                telemetry_data = {'voltage': voltage, 'current': current}
                mqtt_client.publish('v1/devices/me/telemetry', str(telemetry_data))
                time.sleep(1)  # Pause for a bit
    except Exception as e:
        print("Error: %s" % str(e))
    finally:
        if 'ser' in locals() and ser.is_open:
            ser.close()
# Main program
if __name__ == "__main__":
    lcd_init()  # Initialize LCD
    read_uart3()  # Read from UART and process data
```