A REPORT

ON

"Development of Modbus slave node using Arduino mega 2560".

BY

K VED KARTHIK MANIKANTA

B.Tech Electronics and Communication Engineering with specialisation in biomedical engineering

AT

QA (Maintenance and Development),
NUCLEAR FUEL COMPLEX,
DEPARTMENT OF ATOMIC ENERGY,
HYDERABAD.

Under the guidance of
A.Hari Babu,
SO/F,
Manager, Maint (C.Lab and Char.Lab)

DEPARTMENT OF ELECTRONICS AND COMMUNICATION ENGINEERING,
VELLORE INSTITUTE OF TECHNOLOGY,
VELLORE, TAMILNADU.

Government of India
Department of Atomic Energy
NUCLEAR FUEL COMPLEX

BONAFIDE CERTIFICATE

This is to certify that Mr. K VED KARTHIK MANIKANTA has done his Project Work under my guidance during the period from 21st Aug 2023 to 20th Sep 2023 on the topic entitled "Development of Modbus slave node using Arduino mega 2560" with reference to Nuclear Fuel Complex.

It is ensured that the report does not contain classified or Plant operational live data in any form.

Signature :
Name :
Desgn.Of Guide :
Plant :

Hyderabad
20th Sep 2023

Approved by
The Manager of the Plant

ACKNOWLEDGEMENT

I express my sincere thanks to Mr. A.Hari Babu, SO/F, Manager, Maint(C. Lab and Char Lab), Nuclear Fuel Complex who has guided and helped me obtain an in-depth knowledge about the project.

I express my thanks to Mr. A Someshwar Rao, P.G.Srinivas Rao, K.Anil Kumar for spending his valuable time and for helping me understand the process.

I express our sincere thanks to Shri. C. Chinna Sailoo, AGM-QA (Maint and Development) for allowing me to do the project. I express our thanks to Shri, D.Srinivas , DGM(HR) for helping us throughout our training period at NFC and also conducting awareness programme on DAE, NFC activities at HRD.

I express my gratitude to Prof. Ms.Dr VIDHYA S, Head of the department, Electronics and Communication Engineering with spl in Biomedical Engineering, Vellore Institute of technology , Velmurugan V Associate Professor, Vellore Institute of technology for accepting my request to undergo training at Nuclear Fuel Complex

Abstract:

The project "Development of a Modbus Slave node Using Arduino Mega 2560" presents the design, implementation, and validation of a Modbus slave system using an Arduino Mega 2560 microcontroller. Modbus, a widely adopted communication protocol in industrial automation and control systems, serves as the backbone of the project, enabling seamless data exchange between the Arduino-based Modbus slave and other devices in the industrial ecosystem.

The primary objectives of the project are to create a robust Modbus slave system capable of interfacing with sensors, actuators, and Modbus masters while adhering to the RS 485 protocol standards. The Arduino Mega 2560 serves as the central processing unit, executing Modbus communication and data handling tasks efficiently.

First, by looking into the respective machinery we have designed the circuit diagram for the Modbus slave along with the Arduino mega 2560 as an internal structure of the circuit. By using cadence software, we have designed the circuit digitally and then gave the order for the fabrication of the printed circuit board.

Meanwhile we have developed the code for the Modbus slave using Arduino using Embedded C later as the board has arrived, we have obtained all the components and soldered it to the PCB. Then we have checked all the connections and verified it with designed circuit diagram.

Later we have connected the PCB along with Arduino to the machines HMI display with the RJ45 cable and then tested it with all the inputs and the expected output. By thorough testing we have obtained the required outputs.

Extensive testing and validation are performed to ensure the reliability and functionality of the Modbus slave system. Test cases cover a range of scenarios, including data reading, writing, error handling, and performance assessment under varying conditions. The validation process aims to confirm that the Modbus slave system meets project requirements and operates flawlessly within industrial or automation applications.

INDEX

1) INTRODUCTION TO NFC

The Nuclear Fuel Complex (NFC), established in the year 1971 is a major industrial unit of Department of Atomic Energy, Government of India. The complex is responsible for the supply of nuclear fuel bundles and reactor core components for all the nuclear power reactors operating in India. It is a unique facility where natural and enriched uranium fuel, zirconium alloy cladding and reactor core components are manufactured under one roof starting from the raw materials.

The Nuclear Fuel Complex set up at Hyderabad, will supply nuclear fuel bundles to the nuclear power reactor, Zircaloy reactor core component, seamless stainless steel and ball bearing tubes core sub-assemblies for fast breeder test reactor and high purity materials for electronic and allied industries.

Need for Nuclear Energy

Thermal Power produced causes a lot of pollution due to the release of carbon dioxide gas into the atmosphere which leads to breathing problems. Also, the Carbon dioxide Gas released to the atmosphere forms a layer around the earth which leads to the global warming and greenhouse effect. On the other hand, hydro-electric power involves huge investments for its set-up and the construction of dams etc. So, considering all these disadvantages nuclear power proves to be a better source of power. Nuclear energy is released mainly due to nuclear instability of the atoms. One gram of Uranium on complete fission gives energy equivalent to 2.5 tons of coal. So, Uranium emits about two and a half billion times more energy than that emitted by coal. The reactor is a horizontal vessel called Calandria Vessel which consists of 306 Channels. The fuel bundles are present in Zirconium cover. Each Channel takes two bundles. Three phase reactions can be carried out of which the first phase consists of the basic reaction making use of Uranium for the production of energy. The Second Phase consists of utilizing Plutonium for the production of energy. Third Phase consists of using Thorium for the production of nuclear energy. Uranium-235 is mainly used as the nuclear fuel which constitutes about 0.7% of the earth's crust and the rest 99.3% is Uranium-238.

REACTIONS:

$_{92}U^{235} + {}_{0}n^{1} \rightarrow {}_{92}U^{236}$

$_{88}Ra^{226} \rightarrow {}_{2}He^{4} + {}_{88}Ra^{226}$

$_{2}He^{4} + {}_{4}Be^{9} \rightarrow {}_{6}C^{12} + {}_{0}n^{1}$

Role of NFC

NFC carries out all the basic steps required for the production of the nuclear fuel bundles till the finished product is obtained. The pressurized tubes of the reactor used to carry the coolant heavy water is also produced by NFC. They are mainly made up of Niobium. The Complex is made of the following constituent units:

1. Zirconium Plant**:**

This is an integral plant consisting of three units:

a) Zirconium Oxide Plant

b) Zirconium Sponge Plant

c) Zircaloy Fabrication Plant.

Zircon sand obtained from the Indian Rare Earths Limited (Kerala) is processed into high purity Hafnium – free Zirconium Oxide at the Zirconium Oxide Plant and later into Zirconium Sponge metal at the Sponge Plant. The nuclear grade sponge is then converted into fuel sheathing and other reactor core components at the Zircaloy Fabrication Plant.

(a) Zirconium Oxide Plant:

Zircon Sand ($ZrSiO_4$) obtained from Kerala in the form of dry Powder is dissolved in nitric acid at $80^{o}C$ to obtain a Crude Solution which is taken for Slurry extraction using (try butyl phosphate) TBP + Kerosene as solvent. Extract is sent to scrubbing unit and then to strip solvent using DM Water (De-mineralized Water) to obtain pure $ZrNO_3$ solution which is precipitated using

NH_4OH to obtain $Zr(OH)_2$ which is filtered by Rotary Vacuum drum filter, dried, calcined, crushed and sieved to get nuclear grade Zirconium Oxide

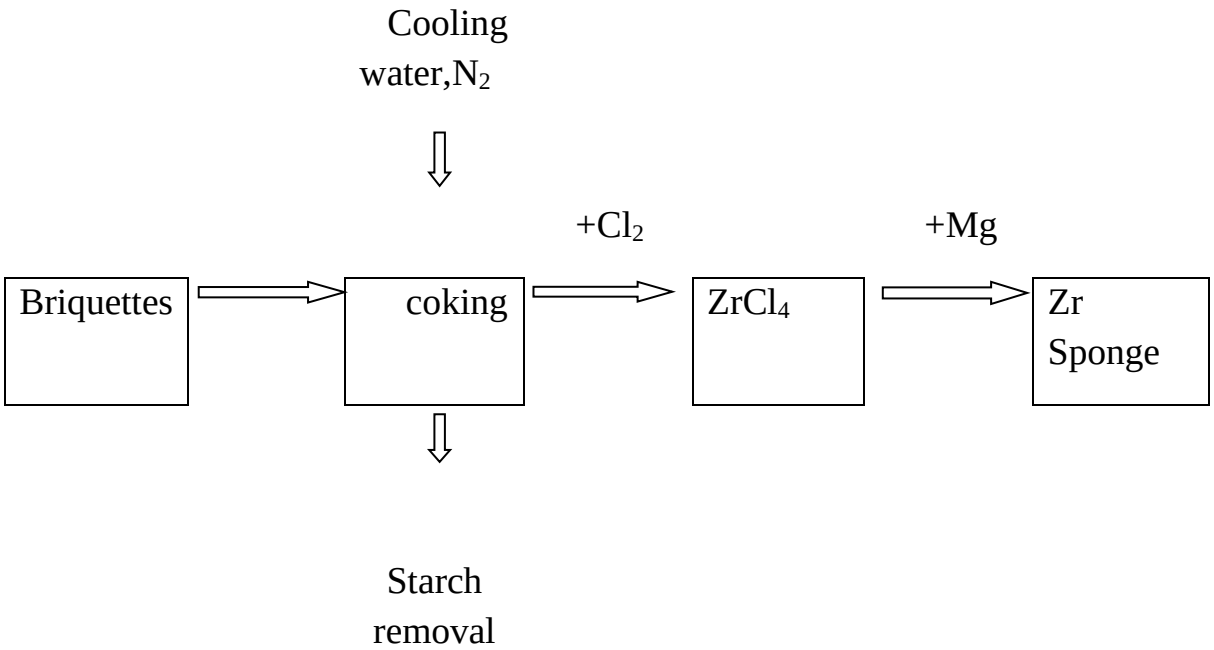
(b) Zirconium Sponge Plant:

Zirconium sponge plant processes $ZnCl_4$ (intermediate) to give pure Zirconium which is called as Reactor Grade sponge.

Reaction:

$$ZrO_2 + 2Cl_2 – ZrCl_4$$

Petroleum Coke, Starch solution, Zirconium Oxide Powder obtained from Zirconium Oxide Plant are mixed in a sigma mixer and thick mixture so formed is passed through extrusion from which briquettes are obtained. These briquettes are coked with cooling. water and N_2, which on chlorination produces $ZrCl_4$, which on reducing with Magnesium metal produces Zirconium metal.



Reaction:

$ZrCl_4 + 2Mg \rightarrow Zr + 2MgCl_2$

This Zr is coated with argon to prevent corrosion.

(c) Zircaloy Fabrication Plant:

- Ignot Making
- Zircaloy 2 for BWR (Boiling Water Reactor)
- Zircaloy 4 and Zr 2.5 Nb for PHWR (Pressurised Heavy Water Reactor)

2. Uranium Oxide Plant:

To process the Uranium ore concentrates obtained from the Uranium Corporation of India Limited (Jaduguda, Bihar) to ceramic quality, Reactor-Grade Uranium Oxide Powder.

3. Ceramic Fuel Fabrication Plant:

The Ceramic grade Uranium Oxide Powder obtained from the Plant at Uranium Oxide Plant above is made into sintered pellets at the Ceramic Fuel Fabrication Plant and then loaded and scaled inside Zircaloy tubes obtained from Zirconium Plant and the tubes are assembled into fuel element bundles.

4. Fast Reactor Facility:

In this facility various components required for fast breeder for reactors are manufacture and supplied to Kalpakkam (Tamilnadu)

5. Melt Shop, Billet Preparation Plant, Extrusion Press Plant, Zircaloy Fabrication Plant:

In these plants reactor grade Zirconium is converted into structural components required for nuclear reactors.

6. Quality Control Laboratory:

For Chemicals, spectroscopic and Vacuum fusion analysis, metallographic examination, mechanical and corrosion testing, in order to keep a close check on impurity levels in the various stages of production and to meet stringent reactor-grade specifications.

7. Seamless Steel Tube Plant:

In order to utilize the capacity available in the extrusion press of the seamless stainless tubing to meet the country's requirement additional machinery have been installed for the production of Zircaloy tubing.

In addition to this, NFC also has a health physics unit which plays a main role in the safety aspects. Periodic tests are done and the contamination levels are taken care to be maintained at an appropriate level which is harmless. They monitor the exhaust levels, surface contamination level, radiation level, transport of fuels etc.

2) NFC And Future Power Scenario:

Thermal and hydropower productions have many inherent drawbacks in terms of the availability and purity of raw material and the pollution problems that arise from their operations. The major problems that arise from the thermal plants are the non-availability of good grade coal. The Indian coal has high sulfur content and low calorific value, which is actually resulting in an uneconomical production of electricity. The coal reserves are also fast depleting. The amount of pollutants is also very high, ash, carbon dioxide and sulfur dioxide being the major pollutants. The hydropower generation is second and regional.

The answer to all these problems is the nuclear source. Just one gram of U^{235} would give the same amount of energy as 2.5 tons of coal. Thus, the coal consumption can be reduced drastically and this will automatically result in reduced effluents. The treatment of radioactive waste is not an easy task, but the amounts would make this easier.

The abundance of Uranium is very less and is only 0.04% of which the fissionable material is only 0.7%. Therefore, the consumption of the ore poses a greater threat than that of coal. This problem has been solved. The U^{235} is converted to fissionable U^{233} in the reactor. Therefore, the Uranium is consumed as well as manufactured in an in-situ manner.

Being the only plant in India that supplies fuel and fuel bundles for the nuclear installations in the country NFC is therefore, an important component in country's progress towards increased nuclear power production.

3.1) Data Communication

Data communication refers to the exchange of data or information between two or more devices through a medium such as wires, cables, Optical Fibers, or wireless channels. It is a fundamental aspect of modern technology and plays a crucial role in enabling the flow of information in various forms, including text, audio, video, and more. Here are some key concepts and components related to data communication:

Data: Data can be any piece of information that needs to be transmitted, such as text messages, files, images, or videos. Sender: The sender is the device or entity that initiates the communication by generating or providing the data.

Receiver: The receiver is the device or entity that receives the data from the sender and processes it. Medium: The medium is the physical or logical path through which data is transmitted. It can be wired (e.g., copper cables), wireless (e.g., radio waves), or optical (e.g., Fiber optics).

Channel: The channel is the specific communication path within the chosen medium that data follows from sender to receiver. In some cases, it may refer to a specific frequency or wavelength in wireless communication.

Protocols: Communication protocols are a set of rules and conventions that govern how data is formatted, transmitted, received, and processed. Examples include TCP/IP for internet communication and HTTP for web browsing.

Data Rate or Bandwidth: Data rate, often referred to as bandwidth, is the amount of data that can be transmitted in a given time period, typically measured in bits per second (bps).

Duplex Communication: Data communication can be half-duplex (one-way at a time) or full- duplex (two-way communication simultaneously).

Modulation: In wireless communication, data is often modulated onto carrier waves to transmit it over the air. Modulation changes some properties of the carrier wave (e.g., amplitude, frequency, or phase) to represent the digital data.

Error Detection and Correction: To ensure the integrity of transmitted data, various techniques are used to detect and correct errors that may occur during transmission, such as checksums and parity bits.

Multiplexing: Multiplexing is the technique of combining multiple data streams onto a single channel or medium to maximize its use and efficiency. Time-division multiplexing (TDM) and frequency-division multiplexing (FDM) are common multiplexing methods.

Networks: Data communication often involves multiple devices connected in a network. This can range from local area networks (LANs) within a single building to global-scale networks like the internet.

Security: Data communication also involves securing data to protect it from unauthorized access, interception, or tampering. Encryption and authentication are essential for data security.

Latency and Delay: These terms refer to the time it takes for data to travel from the sender to the receiver. Low latency is critical for real-time applications like voice and video calls.

Routing: In network communication, routing involves determining the best path for data to travel from the source to the destination. Routers play a vital role in this process.

Effective data communication is essential for the functioning of modern society, as it enables the exchange of information across the globe, supports businesses, facilitates research, and connects people through various communication devices and services.

3.2) Protocols

Protocols are sets of rules and conventions that define how data is formatted, transmitted, received, and processed in data communication systems. They are essential for ensuring that devices and systems can communicate effectively with each other, regardless of their underlying hardware or software. Here's everything you need to know about protocols:

Definition and Purpose: Protocols define a standardized way for devices and systems to exchange data and communicate. They ensure that information is transmitted reliably and consistently. Protocols enable interoperability, allowing devices and systems from different manufacturers to work together seamlessly.

Key Components: Protocols consist of various components, including message formats, data encoding schemes, rules for error detection and correction, and procedures for establishing, maintaining, and terminating communication sessions.

Standardization: Protocols are often developed and standardized by organizations like the Internet Engineering Task Force (IETF), IEEE, and ITU-T. Standardization ensures that protocols are widely accepted and can be implemented uniformly across different systems and platforms.

Protocol Stacks: Many communication systems use a stack of protocols, where each layer of the stack handles specific functions. The OSI (Open Systems Interconnection) model and the TCP/IP model are commonly used reference models for understanding protocol stacks. Protocols in a stack work together to enable end-to-end communication. For example, in the TCP/IP stack, the Internet Protocol (IP) operates at the network layer, while the Transmission Control Protocol (TCP) operates at the transport layer.

Packet Structure: Protocols often define the structure of data packets or frames, including headers, data fields, and sometimes trailers. Headers typically contain information such as source and destination addresses, sequence numbers, and control flags.

Handshaking: Many protocols involve a handshaking process during the establishment of a connection. This process allows both communicating parties to agree on communication parameters and verify each other's readiness to exchange data.

Error Handling: Protocols include mechanisms for detecting and handling errors that can occur during data transmission. This may involve checksums, acknowledgments, and retransmissions.

State Machines: Some protocols use finite state machines to define the possible states a communication session can be in and the transitions between these states.

Evolution and Versions: Protocols can evolve over time to address new requirements or security concerns. New versions may be developed to improve performance, security, or functionality.

Examples of Common Protocols: HTTP (Hypertext Transfer Protocol): Used for web communication. SMTP (Simple Mail Transfer Protocol): Used for sending email. FTP (File Transfer Protocol): Used for transferring files over a network. TCP (Transmission Control Protocol): Provides reliable, connection-oriented data transmission. UDP (User Datagram Protocol): Provides connectionless, low-latency data transmission. DNS (Domain Name System): Resolves domain names to IP addresses.

3.3) Types of Protocols

There are many types of protocols used in various fields of computing and communication to facilitate data exchange and interaction. Here are some common types of protocols, categorized based on their purposes:

Communication Protocols: These protocols govern how data is transmitted over a network or communication medium.

Examples include:

- TCP/IP (Transmission Control Protocol/Internet Protocol): The fundamental protocol suite of the internet, responsible for reliable data transmission and addressing.
- UDP (User Datagram Protocol): A connectionless protocol that offers low-latency but not guaranteed delivery.
- HTTP (Hypertext Transfer Protocol): Used for web communication and transferring hypertext documents.
- FTP (File Transfer Protocol): Used for transferring files over a network.
- SMTP (Simple Mail Transfer Protocol): Used for sending email.
- POP3 (Post Office Protocol version 3) and IMAP (Internet Message Access Protocol): Used for retrieving email.

Security Protocols: These protocols focus on securing data during transmission or authentication.

Examples include:

- SSL/TLS (Secure Sockets Layer/Transport Layer Security): Provides encryption and authentication for secure web communication.
- IPsec (Internet Protocol Security): Ensures secure network communication.
- SSH (Secure Shell): Used for secure remote access and file transfers.
- Kerberos: Provides strong authentication for network services.

Routing Protocols: These protocols determine the best path for data to travel within a network.

Examples include:

- RIP (Routing Information Protocol): A distance-vector routing protocol.
- OSPF (Open Shortest Path First): A link-state routing protocol.
- BGP (Border Gateway Protocol): Used in internet routing between autonomous systems.

Application Layer Protocols: These operate at the application layer of the OSI model and facilitate specific types of applications.

Examples include:

- DNS (Domain Name System): Resolves domain names to IP addresses.
- SNMP (Simple Network Management Protocol): Used for managing and monitoring network devices.
- NTP (Network Time Protocol): Synchronizes computer clocks over a network.

Data Link Layer Protocols: These manage data transfer between devices on the same physical network.

Examples include:

- Ethernet: A common wired LAN protocol.
- Wi-Fi (IEEE 802.11): A protocol for wireless LANs.

Transport Layer Protocols: These provide end-to-end communication and error recovery.

- Examples include:
- TCP (Transmission Control Protocol): Provides reliable, connection-oriented data transmission.
- UDP (User Datagram Protocol): Provides connectionless, low-latency data transmission.

VoIP Protocols: These are specialized protocols for voice over IP (VoIP) communication.

Examples include:

- SIP (Session Initiation Protocol): Used to establish, modify, and terminate VoIP sessions.
- RTP (Real-time Transport Protocol): Used for transmitting audio and video over IP networks.

Network Management Protocols:

- These are used for monitoring and managing network devices and resources.
- Examples include SNMP and NetFlow.

File Transfer Protocols:

- These protocols are used for transferring files between systems.
- Examples include FTP, SFTP (SSH File Transfer Protocol), and SCP (Secure Copy Protocol).

Peer-to-Peer Protocols:

- These facilitate direct communication between peers on a network without the need for a central server.
- Examples include BitTorrent and Direct Connect.

IoT (Internet of Things) Protocols:

- These are designed for communication within IoT ecosystems.
- Examples include MQTT (Message Queuing Telemetry Transport) and CoAP (Constrained Application Protocol).

4) Communication Protocols

Communication protocols are sets of rules and conventions that govern how data is formatted, transmitted, received, and processed in data communication systems. These protocols ensure that devices and systems can communicate effectively, reliably, and consistently. Here are some key aspects of communication protocols:

Standardization: Communication protocols are often developed and standardized by organizations like the Internet Engineering Task Force (IETF), IEEE, ITU-T, and others. Standardization ensures that protocols are widely accepted and can be implemented uniformly across different systems and platforms.

Types of Communication Protocols:

- Network Communication Protocols: These govern how data is transmitted over a network or communication medium. Examples include TCP/IP (Transmission Control Protocol/Internet Protocol) and UDP (User Datagram Protocol).
- Application Layer Protocols: These operate at the application layer of the OSI model and facilitate specific types of applications, such as HTTP (Hypertext Transfer Protocol) for web communication and SMTP (Simple Mail Transfer Protocol) for sending email.
- Transport Layer Protocols: These provide end-to-end communication and error recovery. Examples include TCP (Transmission Control Protocol) and UDP (User Datagram Protocol).
- Data Link Layer Protocols: These manage data transfer between devices on the same physical network, like Ethernet and Wi-Fi protocols.
- Security Protocols: These focus on securing data during transmission and may include SSL/TLS for secure web communication and IPsec for secure network communication.
- Routing Protocols: These determine the best path for data to travel within a network, such as RIP (Routing Information Protocol), OSPF (Open Shortest Path First), and BGP (Border Gateway Protocol).

Packet Structure: Protocols often define the structure of data packets or frames, including headers, data fields, and sometimes trailers. Headers typically contain information such as source and destination addresses, sequence numbers, and control flags.

Handshaking: Many protocols involve a handshaking process during the establishment of a connection. This process allows both communicating parties to agree on communication parameters and verify each other's readiness to exchange data.

Error Handling: Protocols include mechanisms for detecting and handling errors that can occur during data transmission. This may involve checksums, acknowledgments, and retransmissions.

State Machines: Some protocols use finite state machines to define the possible states a communication session can be in and the transitions between these states.

Evolution and Versions: Protocols can evolve over time to address new requirements, security concerns, or performance improvements. New versions may be developed to enhance functionality.

Application and Industry Specific: Some communication protocols are tailored for specific applications or industries. For example, Modbus is used in industrial automation, while MQTT is common in IoT (Internet of Things) applications.

Open Standards vs. Proprietary Protocols: Open standard protocols have publicly available specifications and are typically more interoperable, while proprietary protocols are developed and controlled by specific organizations or companies.

5) Modbus

Introduction:

Modbus is primarily a communication protocol used for industrial automation and control systems. It falls under the category of "Communication Protocols" and is often used to facilitate data exchange between various devices and sensors in industrial settings. Modbus is not limited to industrial applications, but it is most commonly associated with them.

Here are some key points about Modbus:

Purpose: Modbus is designed for transmitting data between devices in supervisory control and data acquisition (SCADA) systems, programmable logic controllers (PLCs), and other industrial automation equipment.

Types: There are different variants of Modbus, including Modbus RTU (used over serial connections) and Modbus TCP/IP (used over Ethernet networks). Modbus RTU is a serial communication protocol, while Modbus TCP/IP is used over Ethernet and is based on the TCP/IP protocol suite.

Message Structure: Modbus messages have a structured format, including addressing information, function codes, data, and error-checking fields.

Master-Slave Architecture: In Modbus communication, there is typically a master device (e.g., a SCADA system or PLC) that initiates requests to read or write data to slave devices (e.g., sensors or actuators). The slave devices respond to these requests.

Usage: Modbus is widely used in various industries, including manufacturing, energy, and building automation, to monitor and control processes and equipment.

Open Standard: Modbus is an open standard, which means that its specifications are publicly available, making it easier for different manufacturers to implement and support.

6) History

Modbus is a widely used communication protocol in industrial automation and control systems. Its history dates back to the late 1970s and early 1980s when it was initially developed. Here is a brief history of Modbus:

Development at Modicon:

- Modbus was developed by Modicon, a company that later became part of Schneider Electric. Modicon was a pioneer in the field of programmable logic controllers (PLCs), which are widely used in industrial automation.
- Modicon developed Modbus in 1979 as a proprietary communication protocol for its PLCs. The goal was to create a simple and efficient way for PLCs to communicate with other devices, such as sensors, actuators, and other controllers.

Early Versions:

- The original version of Modbus was a serial communication protocol, known as Modbus RTU (Remote Terminal Unit).
- Modbus RTU used binary encoding for data and was initially designed for RS-232 communication.

Standardization:

- Modbus gained popularity due to its effectiveness and simplicity. Recognizing its potential as an industry-standard protocol, Modicon allowed the protocol's specifications to be published.
- This openness led to the development of various implementations of Modbus by different manufacturers, making it a de facto standard for industrial communication.

Modbus ASCII and Modbus TCP/IP:

- In addition to Modbus RTU, Modicon introduced Modbus ASCII, a text-based variant, in the early 1980s.
- The emergence of Ethernet and TCP/IP networks in the 1990s prompted the development of Modbus TCP/IP, which allowed Modbus to be used over modern Ethernet networks.

Formal Standardization:

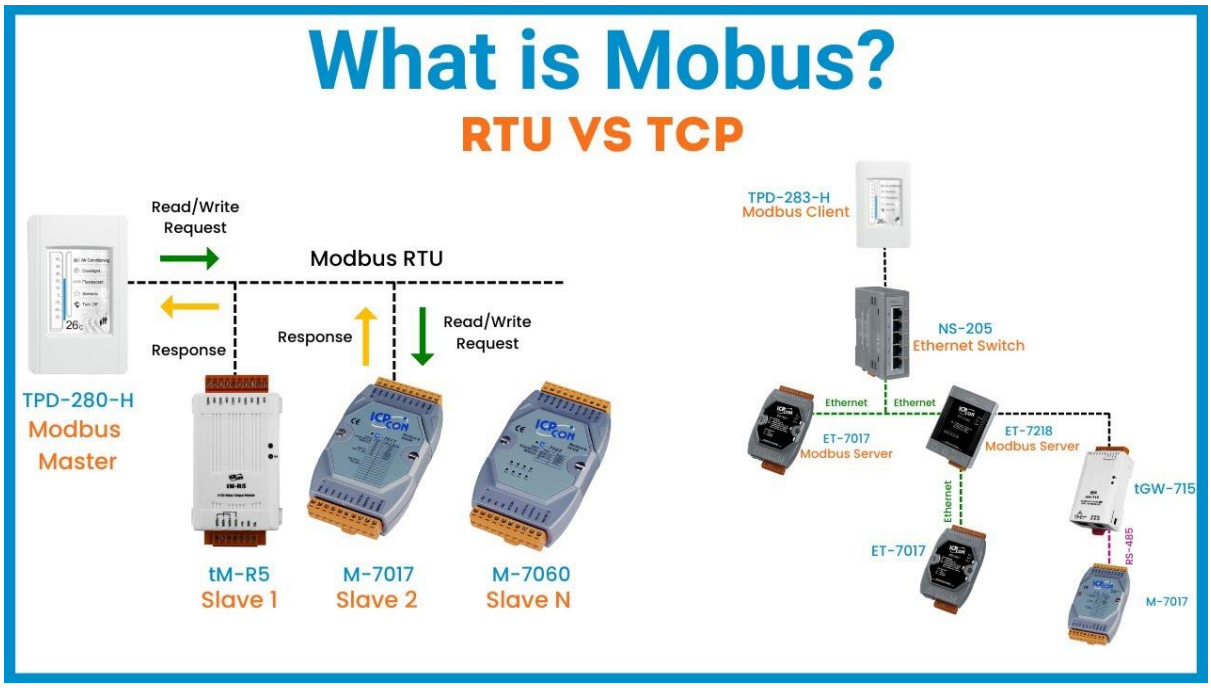
- In 2006, Modbus was formally standardized as Modbus Application Protocol (Modbus-AP) under the IEC 61158 standard. This formal standardization further solidified its status as an industrial communication protocol.

Ongoing Evolution:

- Modbus continues to evolve with the changing needs of industrial automation and control. Newer versions and extensions have been introduced to enhance its capabilities and security.

Open Modbus Community:

- The Modbus protocol has a strong community of users and developers who collaborate to improve and expand its applications.
- Various organizations and groups, such as the Modbus Organization, provide resources and support for Modbus users and developers.



7) Architecture

Modbus, a widely used communication protocol in industrial automation and control systems, follows a client-server or master-slave architecture, depending on the specific variant of Modbus being used. Below, I'll describe the typical architecture for Modbus RTU (Remote Terminal Unit) and Modbus TCP/IP:

1)Modbus RTU (Remote Terminal Unit) Architecture:

Physical Layer: Modbus RTU typically operates over serial communication, using RS-232 or RS-485 as the physical layer. RS-485 is common for longer-distance communication in industrial environments.

Topology: It often employs a multi-drop or daisy-chained topology, where multiple slave devices are connected to a single master device over the same communication line.

Master-Slave Model:

- Master: The master device initiates communication by sending requests to one or more slave devices.
- Slave: Slave devices respond to requests from the master. Each slave has a unique address, allowing the master to identify and communicate with specific slaves.

Message Structure:

Modbus RTU messages consist of a header, address, function code, data, and error-checking fields. The data is often in binary format.

Communication:

Communication in Modbus RTU is typically half-duplex, meaning only one device can transmit at a time.

Error Handling:

Error detection and correction are essential in Modbus RTU to ensure data integrity. Parity bits or checksums may be used for error checking.

Example Scenario:

In a typical Modbus RTU scenario, a master PLC (Programmable Logic Controller) may send a request to multiple slave devices (sensors, actuators, etc.) to read sensor values or control actuators.

2)Modbus TCP/IP Architecture:

Physical Layer:

Modbus TCP/IP operates over Ethernet networks, making use of standard Ethernet cabling and hardware.

Topology:

It can use various Ethernet topologies, including star, bus, or ring topologies.

Master-Slave Model:

- Master: In Modbus TCP/IP, the master device initiates communication with slave devices by sending requests over the IP network.
- Slave: Slave devices, which have unique IP addresses, respond to requests from the master.

Message Structure:

Modbus TCP/IP messages are encapsulated within TCP/IP packets. The protocol is binary and typically uses TCP port 502 for communication.

Communication:

Modbus TCP/IP supports full-duplex communication, allowing simultaneous transmission and reception of data.

Error Handling:

TCP/IP provides error detection and correction mechanisms, making it suitable for reliable data transmission.

Example Scenario:

In a Modbus TCP/IP scenario, a master computer or controller may send requests to remote PLCs or industrial devices over an Ethernet network to monitor processes or control equipment.

In both Modbus RTU and Modbus TCP/IP, communication is typically based on a request-response mechanism. The master or client device sends a request to a specific slave or server device, which processes the request and sends a response back to the master.

The choice of Modbus variant depends on the specific requirements of the industrial application, including factors like distance, network infrastructure, and the need for real-time control and monitoring.

8) Modbus TCP/IP

Modbus TCP/IP is a widely used industrial communication protocol that allows devices to exchange data and control information over Ethernet networks. It is an extension of the Modbus communication protocol, which was originally developed for serial communication but was later adapted for Ethernet-based communication. Here's an overview of Modbus TCP/IP:

1. **Protocol Type:** Modbus TCP/IP is part of the Modbus family of protocols and specifically designed for communication over Ethernet networks, including local area networks (LANs) and the internet.

2. Client-Server Architecture: Modbus TCP/IP uses a client-server architecture, where one device (the client) sends requests to another device (the server) to read or write data. The server responds to these requests.

3. Transport Layer: It typically operates using the Transmission Control Protocol (TCP) as the transport layer protocol, which ensures reliable and ordered data delivery.

4. Port Number: Modbus TCP/IP uses port 502 as the default port number for communication.

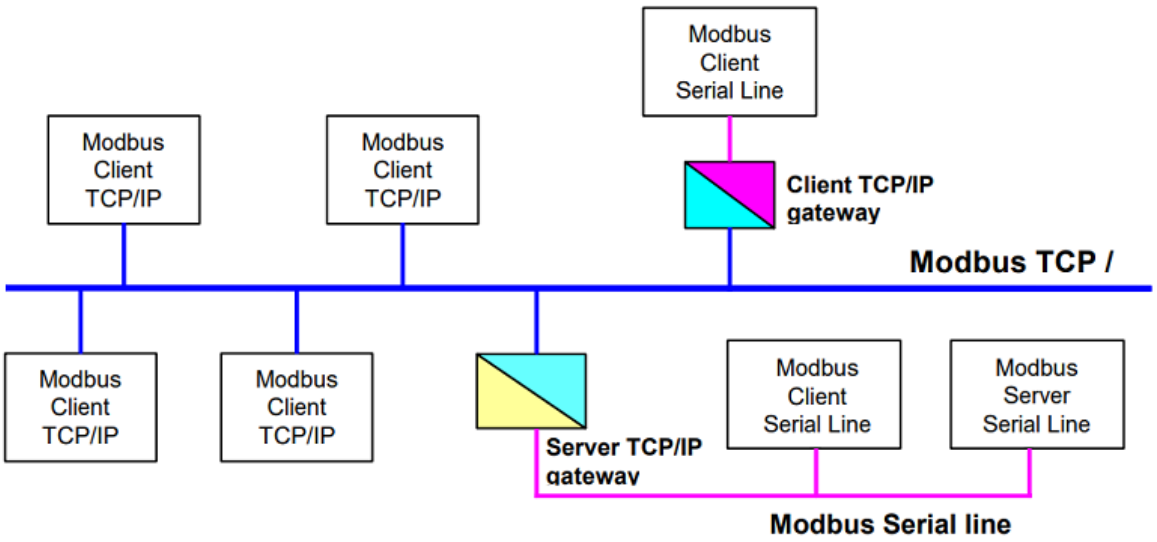
5. Messaging: The Modbus TCP/IP messages consist of a series of function codes that define various operations, such as reading registers, writing data, or performing diagnostics.

6. Data Types: It supports various data types, including binary inputs, binary outputs, analog inputs, and analog outputs. The data is organized into registers, which can be read from or written to.

7. Addressing: Modbus TCP/IP uses a unique device identifier called an IP address (or hostname) and a Unit Identifier (ID) to address devices on the network. The IP address identifies the device, and the Unit ID specifies the function or device within the device.

8. Error Checking: Modbus TCP/IP includes error-checking mechanisms to ensure data integrity and robust communication. This includes CRC (Cyclic Redundancy Check) for error detection.

9. Security: While Modbus TCP/IP itself does not provide strong security features, it can be used in conjunction with VPNs (Virtual Private Networks) or other security measures to protect data and control systems from unauthorized access.



10. Applications: Modbus TCP/IP is commonly used in industrial automation and control systems. It allows devices like PLCs (Programmable Logic Controllers), HMIs (Human-Machine Interfaces), sensors, and actuators to communicate and exchange data in manufacturing, process control, and energy management systems.

11. Interoperability: One of the advantages of Modbus TCP/IP is its widespread adoption and compatibility with a wide range of devices from different manufacturers.

12. Libraries and Implementations: There are various libraries and software tools available that enable the implementation of Modbus TCP/IP communication in devices and systems. These libraries can simplify the integration of Modbus functionality into your applications.

9) Modbus RTU

Modbus RTU (Remote Terminal Unit) is a popular variant of the Modbus communication protocol used in industrial automation and control systems. Within Modbus RTU, there are different types of messages and functions that allow communication between a master device (e.g., a PLC or computer) and slave devices (e.g., sensors or actuators). Here are some common types of messages and functions in Modbus RTU:

Read Holding Registers (Function Code 03): This function code is used by the master to request data from one or more holding registers (data storage locations) in a slave device. Typically used to read data such as sensor values, setpoints, or control parameters.

Read Input Registers (Function Code 04): Similar to Function Code 03, this code is used to read data from input registers in a slave device. Input registers may contain data that is read-only and not meant to be modified by the master.

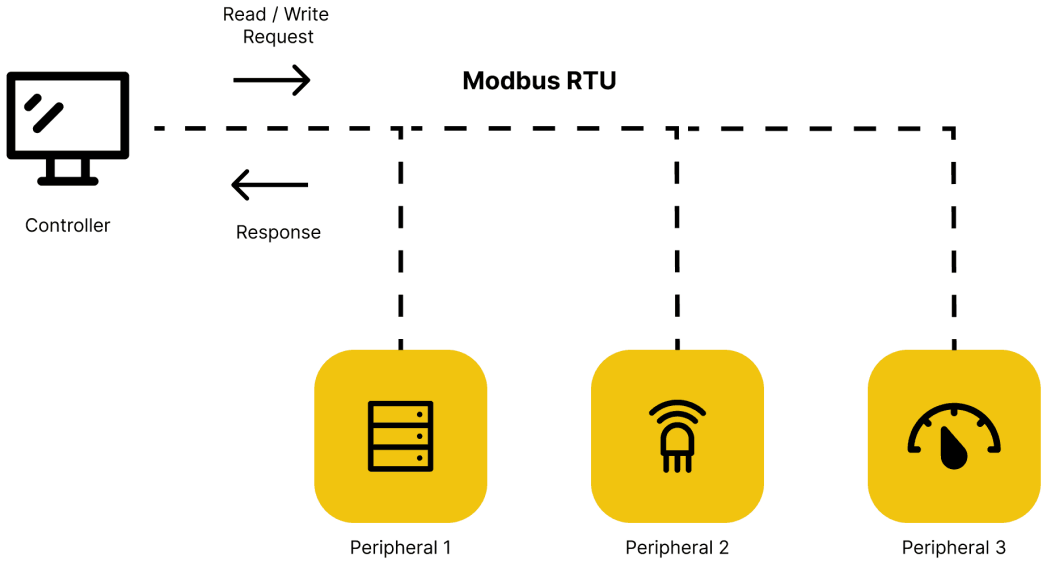
Write Single Coil (Function Code 05): This function code allows the master to write a single coil (a single binary output) in a slave device. Often used for controlling a specific binary output or actuator.

Write Single Register (Function Code 06): Function Code 06 is used to write a single holding register in a slave device. Typically employed to set a specific parameter or control value.

Write Multiple Coils (Function Code 15): With Function Code 15, the master can write multiple coils (binary outputs) in a single request. Useful for controlling multiple binary outputs simultaneously.

Write Multiple Registers (Function Code 16): Function Code 16 allows the master to write multiple holding registers in a single request. Often used to set multiple parameters or control values in a single transaction.

Read Exception Status (Function Code 07): This function code is used to read the exception status of a slave device. Exception status indicates various conditions or error states in the slave.



Diagnostics (Function Code 08): Function Code 08 includes various diagnostic subfunctions that allow the master to perform diagnostic tasks on the slave, such as clearing counters or restarting the device.

Report Slave ID (Function Code 17): Function Code 17 is used by the master to request information about the slave's identity, including its vendor, product, and version information.

Custom Function Codes: Modbus RTU allows for custom function codes to be implemented by device manufacturers to support specific application requirements. Custom function codes are often used for proprietary or specialized features.

10) RS485

RS-485 is a physical layer standard for serial communication, and it is commonly used as the transmission medium for Modbus RTU, which is a variant of the Modbus communication protocol. In other words, RS-485 is often used as the hardware interface over which Modbus RTU messages are transmitted between devices.

RS-485, also known as TIA-485 or EIA-485, is a widely used standard for serial communication. It defines the electrical and mechanical characteristics of the communication interface, allowing for reliable data transmission over long distances and in noisy environments. RS-485 is commonly used in industrial and commercial applications for connecting various devices and systems. Here are key features and characteristics of RS-485:

Differential Signaling: RS-485 uses differential signaling, which means that it transmits data as a voltage difference between two wires (one positive and one negative). This differential signaling provides better noise immunity and allows for longer cable runs compared to single-ended signaling like RS-232.

Balanced Communication: RS-485 supports balanced communication, where data is transmitted simultaneously in both directions (full-duplex) or in one direction at a time (half-duplex). This enables bidirectional communication between devices.

Multi-Point and Multi-Drop: RS-485 supports multi-point or multi-drop configurations, allowing multiple devices to be connected to the same communication bus. Each device has a unique address, and devices can take turns transmitting data.

Common Mode Rejection: RS-485 is designed to reject common mode noise, which is noise that affects both signal wires equally. This makes it suitable for use in electrically noisy industrial environments.

Distance and Data Rate: RS-485 can support communication over relatively long cable distances, often up to 4,000 feet (1,200 meters) or more, depending on factors like cable quality and data rate. Data rates can vary from a few hundred bits per second (baud) to several megabits per second.

Termination: Proper termination is essential in RS-485 networks to prevent signal reflections and ensure signal integrity. Termination resistors are typically used at both ends of the communication bus.

Point-to-Point and Multi-Drop Configurations: RS-485 can be used in both point-to-point (two devices communicating directly) and multi-drop (multiple devices connected to a single bus) configurations.

Applications: RS-485 is commonly used in various applications, including industrial automation, process control, building automation, HVAC systems, access control, and more.

Connectors: RS-485 connectors are typically two-wire connections, and common connectors include twisted-pair cables with terminal blocks or RJ-45 connectors.

Standard: RS-485 is a standard defined by organizations like the Telecommunications Industry Association (TIA) and the Electronic Industries Alliance (EIA).

11) RS-232

RS-232, which stands for "Recommended Standard 232," is a popular standard for serial communication between devices. It defines the electrical and mechanical characteristics of the serial interface and is commonly used for communication between computers and various peripheral devices. Here are key features and characteristics of RS-232:

Serial Communication: RS-232 is a serial communication standard, meaning data is sent sequentially, one bit at a time, over a single pair of wires (typically, one for transmission, one for reception).

Voltage Levels: RS-232 uses voltage levels to represent binary data. Typically, a positive voltage represents binary 1, and a negative voltage represents binary 0. Common voltage levels are ±3 to ±15 volts.

Asynchronous Communication: RS-232 communication is often asynchronous, which means there is no separate clock signal. Instead, both the sender (transmitter) and receiver must agree on a common baud rate (bits per second) for data transmission.

Data Bits and Parity: RS-232 allows for various configurations of data bits, stop bits, and optional parity bits. Common configurations include 8 data bits, 1 stop bit, and no parity (8N1).

Simplex or Duplex: RS-232 can be used in simplex mode (one-way communication) or duplex mode (two-way communication). In duplex mode, there is a separate pair of wires for transmitting and receiving.

Distance Limitations: RS-232 is suitable for relatively short-distance communication, typically up to 50 feet (15 meters) without additional signal boosting or conditioning. Beyond this distance, signal degradation may occur.

Connector: RS-232 connectors are typically 9-pin D-subminiature connectors (DB-9) or 25-pin connectors (DB-25). The DB-9 connector is more common for modern applications.

Use Cases: RS-232 was widely used for connecting computers to peripherals such as modems, mice, printers, and serial terminals. It has been used for configuration and communication with embedded systems, industrial equipment, and scientific instruments.

Challenges and Limitations: RS-232 is sensitive to noise and interference, which limits its use in noisy industrial environments. The short cable distances and the lack of built-in error-checking can lead to data transmission issues over longer distances.

Legacy Status: RS-232 has largely been replaced by more modern serial communication standards, such as USB and Ethernet, for computer peripherals and networking. However, it is still used in various specialized applications and legacy systems.

12) PCB Designing

Creating a PCB (Printed Circuit Board) involves designing a circuit layout and then manufacturing the physical board with the circuit traces and components. Here's a simplified overview of the process for making a PCB:

1. **PCB Design:**
   - Schematic Design: Start by creating a schematic diagram of your circuit using specialized software like KiCad, Eagle, Altium Designer, or others. This diagram represents the connections between components and their values.
   - PCB Layout: Using the same software, transfer the schematic to a PCB layout. Place components on the PCB, route traces to connect them, and define the board's dimensions.
   - Layer Design: PCBs can have multiple layers (top, bottom, inner). Define which layers will be used for traces, ground planes, power planes, and signal layers.

2. **Component Selection:**
   - Choose electronic components, such as resistors, capacitors, integrated circuits (ICs), connectors, etc., and determine their footprints.
   - Ensure that the selected components are suitable for your circuit's requirements, including voltage, current, and temperature specifications.

3. **PCB Design Rules:**

Define design rules for your PCB, including trace widths, spacing between traces, and minimum drill sizes. These rules ensure manufacturability and electrical integrity.

4. **PCB Routing:**
   - Manually route traces to connect components according to your layout. The routing process involves optimizing signal paths, avoiding crossovers, and minimizing trace lengths.
   - Use ground and power planes when necessary for better signal integrity and heat dissipation.

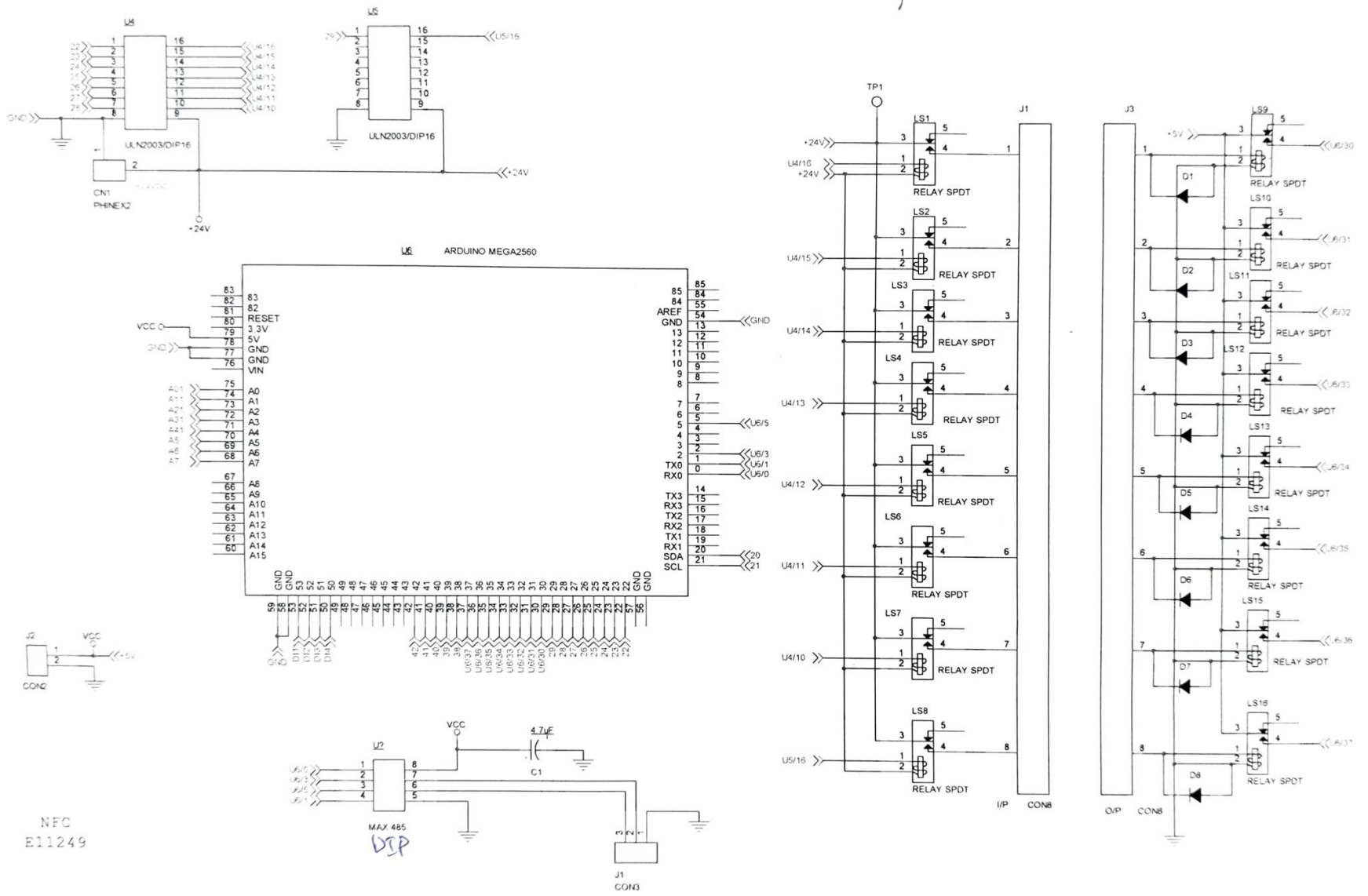
5. **Design Verification:**
    - Run design rule checks (DRC) and electrical rule checks (ERC) within your PCB design software to identify and correct any errors.
    - Verify your design by simulating it if needed to ensure proper functionality.

6. **Generate Gerber Files:**
    - Gerber files are a set of standard files that describe the PCB's layers, traces, and other details. Export these files from your PCB design software.

7. **Order PCB Fabrication:**
    - Send your Gerber files to a PCB fabrication service (e.g., PCB manufacturer). There are many online services that accept Gerber files and offer PCB manufacturing at various price points and turnaround times.
    - Specify your PCB specifications, such as board material, thickness, and the number of layers.

8. **PCB Manufacturing:**

- The PCB fabrication service will manufacture your board based on your Gerber files. This process involves etching copper layers, drilling holes, adding solder mask and silkscreen, and more.


U4
ULN2003/DIP16
U5
ULN2003/DIP16
GND
+24V
CN1
PHINEX2
U6 ARDUINO MEGA2560
RESET
3.3V
5V
GND
GND
VIN
AREF
TX0
RX0
TX3
RX3
TX2
RX2
TX1
RX1
SDA
SCL
J2
CON2
U7
MAX 485
C1
4.7uF
J1
CON3
TP1
J1
I/P CON8
J3
O/P CON8
LS1
LS2
LS3
LS4
LS5
LS6
LS7
LS8
LS9
LS10
LS11
LS12
LS13
LS14
LS15
LS16
RELAY SPDT
D1
D2
D3
D4
D5
D6
D7
D8
U4/16 +24V
U4/15
U4/14
U4/13
U4/12
U4/11
U4/10
U5/16
NFC
E11249
DIP

9. **Component Placement:**

- Once you receive the fabricated PCB, you'll need to assemble it. This involves soldering components onto the board. You can do this by hand or, for complex or high-volume production, by using pick-and-place machines.

10. Soldering and Testing:

- Carefully solder each component onto the PCB, following proper soldering techniques.
- After assembly, thoroughly test the PCB to ensure it functions as expected. This may include continuity tests, functional tests, and debugging if necessary.

10. Integration and Finalization: -

- Integrate the PCB into your overall project, connecting it to power sources, sensors, actuators, and other components as required.
- Ensure that the PCB performs its intended function within the larger system.

13) Arduino Mega 2560

The Arduino Mega 2560 is a microcontroller board based on the ATmega2560 microcontroller chip. It is one of the popular members of the Arduino family, known for its extensive I/O capabilities and memory, making it suitable for a wide range of complex projects. Here are some key features and specifications of the Arduino Mega 2560:

1. **Microcontroller:** The Arduino Mega 2560 is powered by the ATmega2560 microcontroller, which is an 8-bit AVR microcontroller clocked at 16 MHz. It has 256 KB of flash memory for program storage, 8 KB of SRAM, and 4 KB of EEPROM.

2. Digital I/O Pins: It features a total of 54 digital input/output pins, of which 15 can be used as PWM (Pulse Width Modulation) outputs.

3. Analog Inputs: The board has 16 analog input pins, labeled A0 through A15, for reading analog sensor values.

4. Communication Interfaces: UART: There are 4 hardware UART (serial) communication ports for serial communication with other devices. SPI: The board supports SPI (Serial Peripheral Interface) communication. I2C/TWI: It also supports I2C (Inter-Integrated Circuit) communication via the SDA and SCL pins.

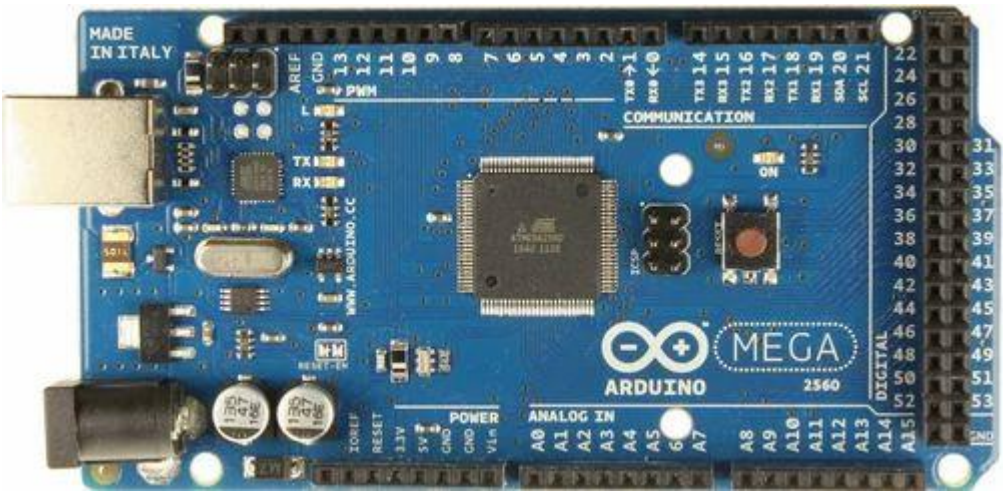
5. USB Interface: The Arduino Mega 2560 can be connected to a computer or power source via a USB-B connector. It can be used both for programming and for serial communication with a computer.

6. Power Supply: The board can be powered through the USB connection or an external power supply. The voltage range for external power is typically 7V to 12V. It also has a built-in voltage regulator that can provide a stable 5V supply for external components.

7. Reset Button: The board includes a reset button for restarting the microcontroller.

8. Operating Voltage: The Arduino Mega 2560 operates at 5V.

9. Flash Memory: It has 256 KB of flash memory for storing your Arduino sketches (programs).



10. SRAM and EEPROM: The board has 8 KB of SRAM for variables and 4 KB of EEPROM for non-volatile data storage.

11. Compatibility: The Arduino Mega 2560 is compatible with the Arduino IDE, which makes it easy to program and upload code to the board.

12. Shields: Like other Arduino boards, the Mega 2560 can be expanded using various "shields," which are add-on boards that provide additional functionality.

14) Arduino IDE

The Arduino IDE (Integrated Development Environment) is a software platform used for programming and developing applications for Arduino microcontroller boards and compatible devices. It provides a user-friendly interface and tools that simplify the process of writing, compiling, and uploading code (known as sketches) to Arduino boards. Here are some key features and components of the Arduino IDE:

Code Editor: The Arduino IDE includes a code editor where you write your Arduino sketches. It supports the C and C++ programming languages.

Integrated Compiler: The IDE has an integrated compiler that translates your code into machine-readable instructions (binary code) that can be executed by the microcontroller on the Arduino board.

Library Manager: Arduino provides a library manager that allows you to easily install, manage, and use libraries of pre-written code for various sensors, displays, and peripherals.

Serial Monitor: The Serial Monitor is a tool within the IDE that allows you to send and receive data between your computer and the Arduino board via the USB connection. It's useful for debugging and monitoring your code.

Board Manager: Arduino supports a wide range of boards. The Board Manager lets you select the specific board you are using, and it provides the necessary hardware definitions for compiling and uploading code to that board.

Code Examples: The IDE includes a collection of code examples that cover a variety of tasks and components. These examples serve as templates and learning resources.

Sketch Management: You can create, open, save, and manage multiple sketches within the IDE. Each sketch is a separate Arduino project.

Upload Tools: The IDE provides tools to upload your compiled code (hex file) to the Arduino board. You can also specify the COM port and other upload settings.

Serial Plotter: In addition to the Serial Monitor, there's a Serial Plotter tool that allows you to graphically visualize data sent to or received from the Arduino.

Preferences: You can configure various preferences in the IDE, such as setting the default location for sketch files, enabling verbose output during compilation and uploading, and more.

Cross-Platform: Arduino IDE is available for multiple operating systems, including Windows, macOS, and Linux, making it accessible to a wide range of users.

Open Source: The Arduino IDE is open-source software, which means that its source code is available for review and modification by the community.

15) Arduino Mega 2560 along with Modbus

The Arduino Mega 2560 can be used in conjunction with the Modbus communication protocol to create industrial automation and control systems, data acquisition systems, and other applications that require communication between devices. Here's a general overview of how you can use an Arduino Mega 2560 with Modbus:

1. **Modbus Communication Library:**
   - To implement Modbus communication on an Arduino Mega 2560, you'll need a Modbus communication library. There are various libraries available for Arduino that support Modbus, such as "SimpleModbus" and "ModbusMaster."
   - You can install these libraries using the Arduino IDE's Library Manager.

2. **Wiring and Hardware:**
   - Connect your Arduino Mega 2560 to other devices in your Modbus network. This may involve RS-232, RS-485, or TCP/IP communication, depending on your specific setup.
   - If using RS-232 or RS-485, ensure you have the appropriate level-shifters, transceivers, and termination resistors as needed for your communication protocol.

3. **Role in the Modbus Network:**
   - Determine whether your Arduino Mega 2560 will act as a Modbus master or slave device. The role will depend on your application's requirements.
   - As a Modbus master, your Arduino can initiate requests to read or write data to Modbus slave devices.
   - As a Modbus slave, your Arduino can respond to requests from a Modbus master.

4. **Programming:**
   - Write Arduino sketches (programs) using the Modbus library to implement the desired Modbus functionality.

- For example, if your Arduino Mega 2560 is acting as a Modbus master, you can write code to send requests to Modbus slave devices and process their responses.
- If it's a Modbus slave, you can write code to respond to Modbus requests from a master and perform actions based on those requests.

5. **Modbus Registers and Data Mapping:**
   - Define Modbus registers within your Arduino sketch. Modbus uses registers to read and write data.
   - Map these registers to specific data points or variables in your Arduino program.
   - Depending on your use case, you might map registers to sensor values, setpoints, or control parameters.

6. **Testing and Debugging:**
   - Test your Modbus communication by connecting your Arduino to other Modbus devices and verifying that data can be exchanged correctly.
   - Use tools like Modbus diagnostics software to monitor and troubleshoot your Modbus communication if needed.

7. **Real-World Applications:**
   - Integrate your Arduino Mega 2560 with Modbus into your specific application. This could include industrial control systems, data logging, remote monitoring, or any application that requires communication with Modbus-compatible devices.

16) HMI

HMI stands for "Human-Machine Interface." It refers to a user interface or dashboard that allows humans to interact with and control machines, devices, systems, or processes. HMIs are commonly used in various industries and applications to monitor and manage complex systems, such as industrial automation, manufacturing, robotics, and more. Here are key aspects of HMI:

1. Display Interface: HMIs typically feature a visual display, which can be a touchscreen panel, monitor, or other display devices. Users interact with the system by interacting with this visual interface.

2. Input Methods: Users interact with HMIs using various input methods, which can include touchscreens, keyboards, keypads, buttons, sliders, knobs, and even voice commands, depending on the application and technology used.

3. Information Presentation: HMIs present information to users in a graphical format, often using charts, graphs, gauges, and other visual elements to convey data and system status.

4. Control and Operation: In addition to displaying information, HMIs allow users to control and operate machines or processes. This can include starting or stopping equipment, adjusting settings, and executing commands.

5. Data Visualization: HMIs provide real-time data visualization, which is critical for monitoring and managing complex systems. Operators can see the status of various sensors, processes, and components in real time.

6. Alarms and Alerts: HMIs can generate alarms and alerts based on predefined conditions or thresholds. These notifications help operators respond quickly to critical situations or malfunctions.

7. Historical Data Logging: Many HMIs can log historical data, allowing users to review past performance and identify trends or anomalies for analysis and optimization.

8. Communication: HMIs often support communication with other devices and systems. They can integrate with PLCs (Programmable Logic Controllers), SCADA (Supervisory Control and Data Acquisition) systems, and other control systems.

9. Industrial HMIs: In industrial settings, industrial HMIs are used to monitor and control manufacturing processes, machines, and equipment. They play a central role in factory automation, ensuring efficient production and process management.

10. IoT and IIoT Integration: With the rise of the Internet of Things (IoT) and Industrial Internet of Things (IIoT), HMIs can connect to a broader network of devices and cloud-based services, enabling remote monitoring and control.

11. Customization: HMIs can often be customized to suit the specific needs of an application. Users can tailor the interface, data displays, and controls to their requirements.

12. Software and Hardware: HMIs can be implemented using dedicated hardware, like touchscreen panels, or as software applications running on general-purpose computers or mobile devices.

13. Safety and Security: Ensuring the safety and security of HMIs is crucial, especially in industrial and critical infrastructure applications, where unauthorized access or tampering can have serious consequences

17) Program Code:

```
#include <avr/interrupt.h>
#include <avr/io.h>
#define myubbr (16000000/16/9600-1)
volatile char ReceivedChar;
unsigned char SC;  // slave code
unsigned char SerialData[8]; //incoming data from UART
unsigned char i=0;
byte QA; // quantity of outputs
byte in=0;
unsigned short int checkSum; //is returned in CRC16
byte checkSumHigh; // high byte of checksum
byte checkSumLow; // low byte of checksum
byte coilLength;
byte startNumberFirstCoil;
byte sendData[54];
unsigned char digital[24];
byte digital_output[2];
unsigned char sendDataLength = 0;
unsigned char j=0;
unsigned char readHoldingRegisterNumber=0;
unsigned char readHoldingRegiserStart=0;
unsigned char readCoil_FC = 1; //modbus readCoil register number
unsigned char readInputstatus_FC = 2; //modbus read input status
```

unsigned char forcesinglecoil_FC = 5; //modbus force single coil

byte RxGo=0; // if RxGo = 1 then communication is available without exception

byte readCoilGo = 0;

byte read_input = 0;

byte force_coil = 0;

byte readHoldingRegisterGo=0;

char a=240;

double remainder; // to find how many coil as a byte to read (for instance 14 coil is need to read, that is 2 byte)

// all possible reminders for checksum

unsigned char auchCRCLo[] = {

0x00, 0xC0, 0xC1, 0x01, 0xC3, 0x03, 0x02, 0xC2, 0xC6, 0x06, 0x07, 0xC7, 0x05, 0xC5, 0xC4,

0x04, 0xCC, 0x0C, 0x0D, 0xCD, 0x0F, 0xCF, 0xCE, 0x0E, 0x0A, 0xCA, 0xCB, 0x0B, 0xC9, 0x09,

0x08, 0xC8, 0xD8, 0x18, 0x19, 0xD9, 0x1B, 0xDB, 0xDA, 0x1A, 0x1E, 0xDE, 0xDF, 0x1F, 0xDD,

0x1D, 0x1C, 0xDC, 0x14, 0xD4, 0xD5, 0x15, 0xD7, 0x17, 0x16, 0xD6, 0xD2, 0x12, 0x13, 0xD3,

0x11, 0xD1, 0xD0, 0x10, 0xF0, 0x30, 0x31, 0xF1, 0x33, 0xF3, 0xF2, 0x32, 0x36, 0xF6, 0xF7,

0x37, 0xF5, 0x35, 0x34, 0xF4, 0x3C, 0xFC, 0xFD, 0x3D, 0xFF, 0x3F, 0x3E, 0xFE, 0xFA, 0x3A,

0x3B, 0xFB, 0x39, 0xF9, 0xF8, 0x38, 0x28, 0xE8, 0xE9, 0x29, 0xEB, 0x2B, 0x2A, 0xEA, 0xEE,

0x2E, 0x2F, 0xEF, 0x2D, 0xED, 0xEC, 0x2C, 0xE4, 0x24, 0x25, 0xE5, 0x27, 0xE7, 0xE6, 0x26,

```
0x22, 0xE2, 0xE3, 0x23, 0xE1, 0x21, 0x20, 0xE0, 0xA0, 0x60, 0x61, 0xA1,
0x63, 0xA3, 0xA2,

0x62, 0x66, 0xA6, 0xA7, 0x67, 0xA5, 0x65, 0x64, 0xA4, 0x6C, 0xAC, 0xAD,
0x6D, 0xAF, 0x6F,

0x6E, 0xAE, 0xAA, 0x6A, 0x6B, 0xAB, 0x69, 0xA9, 0xA8, 0x68, 0x78, 0xB8,
0xB9, 0x79, 0xBB,

0x7B, 0x7A, 0xBA, 0xBE, 0x7E, 0x7F, 0xBF, 0x7D, 0xBD, 0xBC, 0x7C,
0xB4, 0x74, 0x75, 0xB5,

0x77, 0xB7, 0xB6, 0x76, 0x72, 0xB2, 0xB3, 0x73, 0xB1, 0x71, 0x70, 0xB0,
0x50, 0x90, 0x91,

0x51, 0x93, 0x53, 0x52, 0x92, 0x96, 0x56, 0x57, 0x97, 0x55, 0x95, 0x94,
0x54, 0x9C, 0x5C,

0x5D, 0x9D, 0x5F, 0x9F, 0x9E, 0x5E, 0x5A, 0x9A, 0x9B, 0x5B, 0x99, 0x59,
0x58, 0x98, 0x88,

0x48, 0x49, 0x89, 0x4B, 0x8B, 0x8A, 0x4A, 0x4E, 0x8E, 0x8F, 0x4F, 0x8D,
0x4D, 0x4C, 0x8C,

0x44, 0x84, 0x85, 0x45, 0x87, 0x47, 0x46, 0x86, 0x82, 0x42, 0x43, 0x83,
0x41, 0x81, 0x80,

0x40

};

unsigned char auchCRCHi[] = {

0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0, 0x80, 0x41, 0x01, 0xC0, 0x80, 0x41,
0x00, 0xC1, 0x81,

0x40, 0x01, 0xC0, 0x80, 0x41, 0x00, 0xC1, 0x81, 0x40, 0x00, 0xC1, 0x81,
0x40, 0x01, 0xC0,

0x80, 0x41, 0x01, 0xC0, 0x80, 0x41, 0x00, 0xC1, 0x81, 0x40, 0x00, 0xC1,
0x81, 0x40, 0x01,

0xC0, 0x80, 0x41, 0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0, 0x80, 0x41, 0x01,
0xC0, 0x80, 0x41,
```

0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0, 0x80, 0x41, 0x00, 0xC1, 0x81, 0x40, 0x00, 0xC1, 0x81,

0x40, 0x01, 0xC0, 0x80, 0x41, 0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0, 0x80, 0x41, 0x01, 0xC0,

0x80, 0x41, 0x00, 0xC1, 0x81, 0x40, 0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0, 0x80, 0x41, 0x01,

0xC0, 0x80, 0x41, 0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0, 0x80, 0x41, 0x00, 0xC1, 0x81, 0x40,

0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0, 0x80, 0x41, 0x01, 0xC0, 0x80, 0x41, 0x00, 0xC1, 0x81,

0x40, 0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0, 0x80, 0x41, 0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0,

0x80, 0x41, 0x01, 0xC0, 0x80, 0x41, 0x00, 0xC1, 0x81, 0x40, 0x00, 0xC1, 0x81, 0x40, 0x01,

0xC0, 0x80, 0x41, 0x01, 0xC0, 0x80, 0x41, 0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0, 0x80, 0x41,

0x00, 0xC1, 0x81, 0x40, 0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0, 0x80, 0x41, 0x00, 0xC1, 0x81,

0x40, 0x01, 0xC0, 0x80, 0x41, 0x01, 0xC0, 0x80, 0x41, 0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0,

0x80, 0x41, 0x00, 0xC1, 0x81, 0x40, 0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0, 0x80, 0x41, 0x01,

0xC0, 0x80, 0x41, 0x00, 0xC1, 0x81, 0x40, 0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0, 0x80, 0x41,

0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0, 0x80, 0x41, 0x01, 0xC0, 0x80, 0x41, 0x00, 0xC1, 0x81,

0x40

};

boolean ADCgo = 0;

```
unsigned char ba = 0;
const int Dpin1 =  22; // digital input pin 1
const int Dpin2 =  23; // digital input pin 2
const int Dpin3 =  24; // digital input pin 3
const int Dpin4 =  25; // digital input pin 4
const int Dpin5 =  26; // digital input pin 5
const int Dpin6 =  27; // digital input pin 6
const int Dpin7 =  28; // digital input pin 7
const int Dpin8 =  29; // digital input pin 8
const int Dpin9 =  30; // digital input pin 9
const int Dpin10 =  31; // digital input pin 10
const int Dpin11 =  32; // digital input pin 11
const int Dpin12 =  33; // digital input pin 12
const int Dpin13 =  34; // digital input pin 13
const int Dpin14 =  35; // digital input pin 14
const int Dpin15 =  36; // digital input pin 15
const int Dpin16 =  37; // digital input pin 16
const int ena1 = 3;
const int ena2 = 2;
const int output1 = 6;
const int output2 = 7;
const int output3 = 8;
const int output4 = 9;
const int output5 = 10;
```

```
const int output6 = 11;

const int output7 = 12;

const int output8 = 13;

void setup()

{

  SC = 3;

  UBRR0H = (unsigned char)(myubbr>>8);

  UBRR0L = (unsigned char)myubbr;

   //UCSR0C |= (1 << UCSZ00) | (1 << UCSZ10); // Use 8-bit character sizes

  // UCSR0C |= (1 << UCSZ10) | (1 << UCSZ11) | (0 << UCSZ12);

  UCSR0B |= (1 << RXEN0) | (1 << TXEN0) | (1 << RXCIE0);   // Turn on the transmission, reception, and Receive interrupt

    interrupts();

    pinMode(Dpin1, INPUT_PULLUP);

    pinMode(Dpin2, INPUT_PULLUP);

    pinMode(Dpin3, INPUT_PULLUP);

    pinMode(Dpin4, INPUT_PULLUP);

    pinMode(Dpin5, INPUT_PULLUP);

    pinMode(Dpin6, INPUT_PULLUP);

    pinMode(Dpin7, INPUT_PULLUP);

    pinMode(Dpin8, INPUT_PULLUP);

    pinMode(Dpin9,  INPUT_PULLUP);

    digitalWrite(Dpin1, HIGH);

    digitalWrite(Dpin2, HIGH);

    digitalWrite(Dpin3, HIGH);
```

```
digitalWrite(Dpin4, HIGH);
digitalWrite(Dpin5, HIGH);
digitalWrite(Dpin6, HIGH);
digitalWrite(Dpin7, HIGH);
digitalWrite(Dpin8, HIGH);
digitalWrite(Dpin9, HIGH);
pinMode(Dpin10, OUTPUT);
pinMode(Dpin11, OUTPUT);
pinMode(Dpin12, OUTPUT);
pinMode(Dpin13, OUTPUT);
pinMode(Dpin14, OUTPUT);
pinMode(Dpin15, OUTPUT);
pinMode(Dpin16, OUTPUT);
pinMode(output1, OUTPUT);
pinMode(output2, OUTPUT);
pinMode(output3, OUTPUT);
pinMode(output4, OUTPUT);
pinMode(output5, OUTPUT);
pinMode(output6, OUTPUT);
pinMode(output7, OUTPUT);
pinMode(output8, OUTPUT);
pinMode(ena1, OUTPUT);
pinMode(ena2, OUTPUT);
//pinMode(RS485_EN_R, OUTPUT);
```

```
    digitalWrite(output1, LOW);
    digitalWrite(output2, LOW);
    digitalWrite(output3, LOW);
    digitalWrite(output4, LOW);
    digitalWrite(output5, LOW);
    digitalWrite(output6, LOW);
    digitalWrite(output7, LOW);
    digitalWrite(output8, LOW);
    digitalWrite(ena2, LOW);   // receive
    digitalWrite(ena1, HIGH);   // transmit
}
void loop()
{
  digital[0]  = digitalRead(Dpin1);
  digital[1]  = digitalRead(Dpin2);
  digital[2]  = digitalRead(Dpin3);
  digital[3]  = digitalRead(Dpin4);
  digital[4]  = digitalRead(Dpin5);
  digital[5]  = digitalRead(Dpin6);
  digital[6]  = digitalRead(Dpin7);
  digital[7]  = digitalRead(Dpin8);
  digital[8]  = digitalRead(Dpin9);
  digital[9]  = digitalRead(Dpin10);
  digital[10] = digitalRead(Dpin11);
```

```
  digital[11] = digitalRead(Dpin12);
  digital[12] = digitalRead(Dpin13);
  digital[13] = digitalRead(Dpin14);
  digital[14] = digitalRead(Dpin15);
  digital[15] = digitalRead(Dpin16);
  digital[16] = 1;
  digital[17] = 0;
  digital[18] = 0;
  digital[19] = 0;
  digital[20] = 1;
  digital[21] = 1;
  digital[22] = 0;
  digital[23] = 1;
 //digitalWrite(output8, HIGH);
// delay(10);
// digitalWrite(output8, LOW);
// delay(10);
    if(RxGo == 1) // transmission will start
    {
      //if (SerialData[1] == readCoil){
      RxGo = 0;
      sendData[0] = SerialData[0]; // assign first element of the receiving data ( Slave Code ) to the first element of the tramsmitted data
      sendData[1] = SerialData[1]; // assing second element of the receiving data ( Function Code ) to the second element of the transmitted data
```

```
   /*----------------------------- Start Of ReadCoil ------------------------------------
------ */
     if(readCoilGo == 1){  // start of readCoil
     readCoilGo = 0;
   //  remainder = (SerialData[4]*256 + SerialData[5])*0.125; // to find how
many coils as a byte
     remainder = (QA)*0.125; // to find how many coils as a byte
     if((remainder) == 0)
     {
     sendData[2] = remainder;
     }
     else if(remainder  <= 1)
     {
      sendData[2] = 1;
      }
     else if(remainder  == 2)
     {
      sendData[2] = 2;
      }
     else if(remainder  == 3)
     { sendData[2] = 3;
     }
     else
     {
```

```
    sendData[2] = (unsigned short int)(remainder+1);
    }

    sendDataLength=3; // SerialData[0]+Serialdata[1]+SerialData[2]
  // Read Digital Inputs - start - Depending on sendData[2] ( number of byte of the sending data)
    coilLength = SerialData[4]*256+SerialData[5]; // number of coil wanted to read
    startNumberFirstCoil = SerialData[2]*256+SerialData[3]; // first number of coil (-1 çıkartıldı)
    if(coilLength > 24 ){coilLength = 24;}
    if(sendData[2] == 1){
      sendDataLength=4;
      for(i=0;i < (coilLength); i++){
        digital[( (startNumberFirstCoil) + i )] = digital[( (startNumberFirstCoil) + i)] << ( i );
          if(i == ( (coilLength) -1) ){
            for(i=0; i<(coilLength); i++ ){
              sendData[sendDataLength-1] = sendData[sendDataLength-1] + digital[(startNumberFirstCoil)  + i];
            }
          }
      }
    }
    else if(sendData[2] == 2){
      sendDataLength = 5;
```

```
      for(i=0;i < (coilLength); i++){
        if(i<8)
        {
          digital[( (startNumberFirstCoil) + i )] = digital[( (startNumberFirstCoil)
+ i)] << ( i );
        }
        else
        {
          digital[( (startNumberFirstCoil) + i )] = digital[( (startNumberFirstCoil)
+ i)] << ( i-8 );
        }
           if(i == 7 ){
             for(j=0; j<8; j++ ){
               sendData[sendDataLength-2] = sendData[sendDataLength-2] +
digital[(startNumberFirstCoil)  + j];
             }
           }
           if(i == ( (coilLength) -1) ){

             for(i=8; i<(coilLength); i++ ){

               sendData[sendDataLength-1] = sendData[sendDataLength-1] +
digital[(startNumberFirstCoil)  + i ];
             }
           }
      }
```

```
    }
    else if(sendData[2] == 3){
      sendDataLength = 6;
      for(i=0;i < (coilLength); i++)
      {
      if(i<8)
      {
            digital[( (startNumberFirstCoil) + i )] = digital[(
(startNumberFirstCoil) + i )] << ( i );
      }else if(i>7 && i<16){
            digital[( (startNumberFirstCoil) + i  )] = digital[(
(startNumberFirstCoil) + i )] << ( i-8 );
      }else if(i>15){ digital[( (startNumberFirstCoil) + i  )] = digital[(
(startNumberFirstCoil) + i )] << ( i-16 );
      }
          if(i == 7 ){
            for(j=0; j<8; j++ ){
            sendData[sendDataLength-3] = sendData[sendDataLength-3] +
digital[(startNumberFirstCoil)  + j];
            }
          }
          if(i==15){
            for(j=8; j<16; j++ ){
            sendData[sendDataLength-2] = sendData[sendDataLength-2] +
digital[(startNumberFirstCoil)  + j];
            }
```

```
            }
            if(i == (coilLength-1) )  {
              for(i=16; i<(coilLength); i++ ){
              sendData[sendDataLength-1] = sendData[sendDataLength-1] +
digital[(startNumberFirstCoil)  + i ];
              }
            }
          }
      }
      // Read Digital Inputs - end
      } //end of readCoil  -- readCoilGo is assigned to zero (readCoilgo = 0)
     if(read_input == 1){  // start of read digital input
      read_input = 0;
      sendData[2] = 1;
      sendData[3] = 0x00;
      //sendData[3] = 0xFF;
      sendDataLength = 4;
        for(i=0;i < 8; i++){
         digital[i] = digital[i] << (i);
          }
            for(i=0; i<8; i++ ){
             sendData[3] = sendData[3]^digital[i];
            }
      //sendData[3] = digital[7];
```

```
  /*-------------- Start CheckSum - CRC16 ------------------------------ */
    checkSum = mbCRC16(&sendData[0],sendDataLength);
    checkSumLow = checkSum & 0xFF; // low byte of chechsum
    checkSumHigh = checkSum>>8 & 0xFF; // high byte of checksum
    sendData[sendDataLength] = checkSumLow;
    sendData[sendDataLength+1] = checkSumHigh; // MSB
    /* ------------- End of CheckSumm - CRC16 ------------------------*/
    sendDataLength = sendDataLength + 2; // longer ( +2 )  because of checkSum values ( +2 )
    digitalWrite(ena1, HIGH);   // RECEIVE DISABLE
    digitalWrite(ena2, HIGH);   // transmit ENABLE mode
    delay(4);
    for(i=0;i<sendDataLength;i++){
    UDR0 = sendData[i];
    while ( !( UCSR0A & (1<<UDRE0)) );
    }
    //digitalWrite(output8, LOW);
    delay(5);
    digitalWrite(ena1, LOW);    // TRANSMIT DISABLE
    digitalWrite(ena2, LOW);   // ENABLE Receive
    for(i=0;i<sendDataLength;i++){
    sendData[i] = 0;
    }
    i=0;
```

```
SerialData[0]=0;
SerialData[1]=0;
//ES0 = 1; // enable uart interrupt
i=0;  // when interrupt is ready to get the data i should be setted to 0
} //end of read digital input
if(force_coil == 1){  // start of forcesingle coil
 force_coil = 0;
 sendData[2] = SerialData[2];
 sendData[3] = SerialData[3];
 sendData[4] = SerialData[4];
// if(sendData[4] == 0xFF){
   if(sendData[3] == 0){
    if(sendData[4] == 0xFF){
        digitalWrite(output1, HIGH);
       }
     else{
        digitalWrite(output1, LOW);
       }
      }
  if(sendData[3] == 1){
    if(sendData[4] == 0xFF){
        digitalWrite(output2, HIGH);
       }
     else{
```

```
        digitalWrite(output2, LOW);
       }
       }
  if(sendData[3] == 2){
    if(sendData[4] == 0xFF){
        digitalWrite(output3, HIGH);
       }
     else{
        digitalWrite(output3, LOW);
       }
  }
if(sendData[3] == 3){
    if(sendData[4] == 0xFF){
        digitalWrite(output4, HIGH);
       }
     else{
        digitalWrite(output4, LOW);
       }
  }
 if(sendData[3] == 4){
    if(sendData[4] == 0xFF){
        digitalWrite(output5, HIGH);
       }
     else{
```

```
        digitalWrite(output5, LOW);
       }
 }
if(sendData[3] == 5){
   if(sendData[4] == 0xFF){
        digitalWrite(output6, HIGH);
       }
    else{
        digitalWrite(output6, LOW);
       }
 }
if(sendData[3] == 6){
   if(sendData[4] == 0xFF){
        digitalWrite(output7, HIGH);
       }
    else{
        digitalWrite(output7, LOW);
       }
 }
if(sendData[3] == 7){
   if(sendData[4] == 0xFF){
        digitalWrite(output8, HIGH);
       }
    else{
```

```
          digitalWrite(output8, LOW);
         }
       }
     // }
     /* else{
      digitalWrite(output1, LOW);
      digitalWrite(output2, LOW);
      digitalWrite(output3, LOW);
      digitalWrite(output4, LOW);
      digitalWrite(output5, LOW);
      digitalWrite(output6, LOW);
      digitalWrite(output7, LOW);
      digitalWrite(output8, LOW);
     }*/
     sendData[5] = SerialData[5];
     sendDataLength = 6;
            /*-------------- Start CheckSum - CRC16 ------------------------------ */
    checkSum = mbCRC16(&sendData[0],sendDataLength);
    checkSumLow = checkSum & 0xFF; // low byte of chechsum
    checkSumHigh = checkSum>>8 & 0xFF; // high byte of checksum
    sendData[sendDataLength] = checkSumLow;
    sendData[sendDataLength+1] = checkSumHigh; // MSB
    /* ------------- End of CheckSumm - CRC16 ------------------------*/
    sendDataLength = sendDataLength + 2; // longer ( +2 )  because of
checkSum values ( +2 )
```

```
    digitalWrite(ena1, HIGH);   // RECEIVE DISABLE
    digitalWrite(ena2, HIGH);   // transmit ENABLE mode
    delay(4);
    for(i=0;i<sendDataLength;i++){
    UDR0 = sendData[i];
    while ( !( UCSR0A & (1<<UDRE0)) );
    }
    //digitalWrite(output8, LOW);
    delay(5);
    digitalWrite(ena1, LOW);    // TRANSMIT DISABLE
    digitalWrite(ena2, LOW);   // ENABLE Receive
    for(i=0;i<sendDataLength;i++){
    sendData[i] = 0;
    }
    i=0;
    SerialData[0]=0;
    SerialData[1]=0;
    } //end of forcesinglecoil
    /*----------------------------- End Of ReadCoil ------------------------------------
------- */
   } // end of RxGo == 1
 }
ISR(USART0_RX_vect)
{
  //digitalWrite(output8, HIGH);
```

```
 SerialData[ba]  = UDR0;                    // Read data from the RX buffer
 while ( !( UCSR0A & (1<<UDRE0)) );
if(SerialData[0] == SC){ // SC slave code   st+ng address
 /*   if( SerialData[1] == 1){
        if(SerialData[2] == 0){
          if(SerialData[3] == 19){
            if(SerialData[4] == 0){
             if(SerialData[5] == 37){
               if(SerialData[6] == 14){
                 if(SerialData[7] == 132){
                  RxGo = 1;
                 }
               }
             }
            }
          }
        }
     }  */
Read_Coil();
Read_Input();
force_singlecoil();
//digitalWrite(output8, HIGH);
//Holding_Register();
ba = ba + 1;
```

```
  if(ba==8)
  {  ba=0;  }
    }
  else  {
  ba=0;
  //SerialData[0]= 0; /* if slave code and number of function code are */
      //SerialData[1]= 0; /*  not equal, wait first bytes to start         */
      }
   // UDR0 = ReceivedChar;
  }
void Read_Coil(void){
  if(SerialData[1] == readCoil_FC){
      if(ba==7){
        QA = SerialData[4]*256+SerialData[5]; // Quantity of coils
        checkSum = mbCRC16(&SerialData[0],6);
        checkSumLow = checkSum & 0xFF; // low byte of checksum
        checkSumHigh = checkSum>>8 & 0xFF; // high byte of checksum
          if(SerialData[1] == readCoil_FC){
              if(QA>=1 && QA<=2000){
                if(((SerialData[2]*256+SerialData[3])+QA) <25 ){
                  if(SerialData[6] == checkSumLow && SerialData[7]==
checkSumHigh){
                      checkSum = 0;
                      RxGo = 1; // ready for transmitting true info
                      readCoilGo = 1;
```

```
            //ES0 = 0; // disable uart interrupt
                }
             }
           else{
              //SerialData[0] = 0;
              //exception02 = 1; // exception 2
              ba=0;
              }
           }
  else {
    //Exception  3
    //SerialData[0] = 0;
   //exception03 = 1;
      ba=0;
    }
}
else   {
        //SerialData[0] = 0;
   //exception01 = 1;
   ba=0;
  }
   } // end of i==7
 }
else  {
```

```
        //exception 1
       //i=0; // if any bit in the byte take address of the slave then stop i = 0
        //SerialData[0] = 1;
       //exception01 = 1;
        }
     }// end of Read_Coil()
void Read_Input(){
    if(SerialData[1] == readInputstatus_FC){
      if(ba==7){
       checkSum = mbCRC16(&SerialData[0],6);
       checkSumLow = checkSum & 0xFF; // low byte of checksum
       checkSumHigh = checkSum>>8 & 0xFF; // high byte of checksum
         if(SerialData[1] == readInputstatus_FC){
                if(SerialData[6] == checkSumLow && SerialData[7]==
checkSumHigh){
                    checkSum = 0;
                    RxGo = 1; // ready for transmitting true info
                    read_input = 1;
                   //ES0 = 0; // disable uart interrupt
                      }
       }
       else   {
                //SerialData[0] = 0;
          //exception01 = 1;
          ba=0;
```

```
 }
         } // end of i==7
        }
       else  {
        //exception 1
       //i=0; // if any bit in the byte take address of the slave then stop i = 0
        //SerialData[0] = 1;
       //exception01 = 1;
        }
 }
void force_singlecoil(){
   if(SerialData[1] == forcesinglecoil_FC){
     //digitalWrite(output8, HIGH);
     if(ba==7){
       checkSum = mbCRC16(&SerialData[0],6);
       checkSumLow = checkSum & 0xFF; // low byte of checksum
       checkSumHigh = checkSum>>8 & 0xFF; // high byte of checksum
         if(SerialData[1] == forcesinglecoil_FC){
                if(SerialData[6] == checkSumLow && SerialData[7]==
checkSumHigh){
                    checkSum = 0;
                    RxGo = 1; // ready for transmitting true info
                    force_coil = 1;
                   //ES0 = 0; // disable uart interrupt
                      }
```

```
      }
     else   {
              //SerialData[0] = 0;
        //exception01 = 1;
        ba=0;
       }
        } // end of i==7
       }
     else  {
      //exception 1
     //i=0; // if any bit in the byte take address of the slave then stop i = 0
      //SerialData[0] = 1;
     //exception01 = 1;
      }
}
```

/* CRC16 calculation. Do not try to use uchCCRCHi and uchCRCLo, they should be swapped.

Instead use them, just use directly checksum variable */

```
unsigned short int mbCRC16(unsigned char* puchMsg, unsigned char usDataLen)
{
  unsigned char uchCRCHi = 0xFF ; /* high byte of CRC initialized */
  unsigned char uchCRCLo = 0xFF ; /* low byte of CRC initialized */
  unsigned char  uIndex ;
```

```
  uchCRCHi = 0xFF ;
  uchCRCLo = 0xFF ;
while (usDataLen--)
  {
  uIndex = uchCRCHi ^ *puchMsg++;
    uchCRCHi = uchCRCLo ^ auchCRCHi[uIndex];
    uchCRCLo = auchCRCLo[uIndex];
  }
  return ( uchCRCLo<< 8 | uchCRCHi) ;
   }
```

18) Conclusion

In this project "Development of a Modbus Slave node Using Arduino Mega 2560" I have designed the required circuit diagram of the Modbus slave along with Arduino mega 2560 as its component and have given all the required connections and then implemented the same using the code from Arduino IDE connecting it to HMI.

I have performed all the required testing's and all the validations are being conducted thoroughly it runs successfully with all the sensors and actuators. Many test cases are being completely checked for all kinds of inputs and verified the outputs. This aims to confirm that the Modbus slave system meets project requirements and operates flawlessly within industrial,or automation applications and further development can be done based on the required conditions.

19) References

Research papers

https://www.researchgate.net/publication/284724085_Design_and_Implementation_of_Modbus_Slave_Based_on_ARM_Platform_and_FreeRTOS_Environment

https://www.researchgate.net/publication/286812167_Communication_between_PLC_and_Arduino_Based_on_Modbus_Protocol

https://www.ripublication.com/ijeer17/ijeerv9n4_16.pdf

Manuals:

https://descargas.cetronic.es/Manual.pdf

https://camatsystem.com/wp-content/uploads/2015/12/Modbus-manual-TD80.pdf