

PROJECT REPORT

ON

DATA ACQUISITION SYSTEM OF ELECTRON BEAM MELTING FURNACE



Nuclear Fuel Complex, Hyderabad

Dept. of Atomic Energy, Government of India

Submitted by
Sarvagya Ranjan

Under the guidance of
Shri G. Pradeep Kumar
Sr. Manager (MZG)
NFC Hyderabad

Acknowledgment

First of all, I am extremely thankful to Dr. Dinesh Shrivastava, Chairman, NFC Board and Chief Executive, NFC for giving me this opportunity to carry out internship at Nuclear Fuel Complex.

I would like to express my sincere thanks to Shri. G. Pradeep Kumar (Sr. Manager, MZG) for accepting to be my guide and helping me throughout my project work period.

I also thank Mr.D.Sreedhar, DCE (CED, PD & EP) for supporting me during the initial stages of the project.

I would like to thank Shri. Vijay Kumar. N and Shri. Ashok Rao, for providing information regarding electron beam melting process.

I express my sincere thanks to Shri. Jawad Rasheed, AGM (HRD, Q.Cir., QIS) for helping me throughout my training period at NFC and also for conducting the Awareness Program on DAE/NFC activities at HRD.

I thank Prof. Shri G.S Nayak, HOD (Electronics and Communication Engg. MIT Manipal) who has given me this opportunity to intern at Nuclear Fuel Complex, Hyderabad.

I am grateful to my parents who have given me constant encouragement and inspiration to pursue my graduation.

Finally, I would like to thank everyone who has directly or indirectly helped me in the completion of this project.

Government of India
Department of Atomic Energy
NUCLEAR FUEL COMPLEX

BONAFIDE CERTIFICATE

This is to certify that Mr. Sarvagya Ranjan of Manipal Institute of Technology, Manipal has done his project work under my guidance from 13th December 2022 to 12th January 2023 on the topic titled *Data Acquisition System of Electron Beam Melting Furnace* with referenceto Nuclear Fuel Complex.

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

Hyderabad

Date:

G .Pradeep Kumar

Sr. Manager (MZG)

Synopsis

Title:

Data Acquisition System of Electron Beam Melting Furnace.

Abstract:

Electron Beam Melting is the process of passing high power electron beam that produces the energy needed for melting certain elements for their purification. Metals such as Niobium, Tantalum are purified using the technique of Electron Beam Melting. The surface of the melting chamber of electron beam melting furnace requires a continuous supply of water that acts as a coolant. The temperature of water is monitored via Resistor Temperature Detectors (RTDs). The RTDs and the temperature controller are connected using RS-485 interface and ModbusTCP is the medium to communicate between server and the temperature controller. Python programming language and its libraries are used to acquire data from the client that is stored in a database. While the data is being stored in the database, it is simultaneously fetched and plotted using another python program thereby performing real-time data analysis.

Expected Contribution:

To study various communication protocols & standards (RS-232, RS-485, Ethernet, and Modbus) involved in the process of establishing communication between the sensors and the computer.

Establishing Communication between a client and the server and acquiring data from the client and storing it in database. Utilizing the data from the database for real-time data analysis using Python programming language.

Contents

Chapters	Description	Page No.
	Title page	
	Acknowledgment	
	Certificate	
	Synopsis	
Chapter 1	Introduction	6
Chapter 2	Electron Beam Melting	8
2.1	<u>What is EBM?</u>	
2.2	<u>How does the process start?</u>	
Chapter 3	Communication between the Client and the Server	11
3.1	Serial Communication	
3.2	Parallel Communication	
3.3	Communication Protocols & Standards	
	➤ RS-232	
	➤ RS-485	
	➤ Modbus RTU	
	➤ Modbus TCP/IP	
	➤ Ethernet	
3.4	Block Diagram	
Chapter 4	Using PyModbus and other Python Libraries for plotting real-time data	28
4.1	Project Description	
4.2	PyModbusTCP	
	➤ ModbusClient	
	➤ ModbusServer	
	➤ ModbusUtils	
4.3	Other Python Libraries	
4.4	Python Code for fetching data from the sensors	
4.5	Python Code for plotting the real-time data	
4.6	Plot	

Introduction

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 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 the reactor core components are manufactured under one roof starting from raw materials.

The Fuel:

India is pursuing an indigenous three stage Nuclear Power Programmer involving closed fuel cycles of Pressurized Heavy Water Reactors (PHWRs) and Liquid Metal Cooled Fast Breeder Reactors for judicious utilization of relatively limited reserves of uranium and vast resources of thorium.

Uranium Refining and Conversion:

The raw material for the production PHWR fuel in NFC is Magnesium Di-uranate (MDU) and Sodium Uranate (SU) popularly known as “yellow cake”. The MDU concentrate is obtained from the uranium mines and milled at Jaduguda, Jharkhand, while the sodium uranate is obtained from the mines and mills located at Thumalapalle (AP) both operated by Uranium Corporation of India Limited (UICL). The impure MDU and SU are subjected to nitric acid dissolution followed by solvent extraction and precipitation with Ammonium Di-Urinate (ADU). By further steps of controlled calcinations and reduction, sinterable uranium di-oxide powder which is then compacted in the form of cylindrical pellets and sintered at high temperature to get high density uranium dioxide pellets. For BWRs the enriched uranium hexafluoride is subjected to pyro-hydrolysis and converted to ammonium di-urinate which is treated in the same way as natural ADU to obtain high density uranium dioxide pellets.

Zircaloy Production:

The source mineral for the production of zirconium metal is zircon (Zirconium Silicate) available in the beach sand deposits of Kerala, Tamil Nadu and Orissa and is supplied by the Indian Rare Earths LTD. Zircon sand is processed through caustic fusion, dissolution, solvent extraction, precipitation and calcinations steps to get zirconium oxide. Further, the pure zirconium oxide is subjected to high temperature chlorination, reactive metal reduction and vacuum distillation to get homogeneous zirconium sponge. The sponge is then briquetted with alloying ingredients and multiple vacuum arc melted to form homogeneous Zircaloy ingots which are then converted into seamless tubes, sheets and bars by extrusion, pilgering and finishing.

NFC Fuel Fabrication:

At NFC, the cylindrical UO_2 pellets are stacked and encapsulated in thin-walled tubes of zirconium alloy, both ends of which are sealed by resistance welding using Zircaloy end plugs. A number of such fuel pins are assembled to form a fuel bundle that can be conveniently loaded into the reactor.

Seamless Tubes, FBR Sub-Assemblies and Special Materials:

The Stainless-Steel Tubes Plant and Special tubes Plant at NFC produce a wide variety of stainless steel and titanium seamless tubes for both nuclear and non-nuclear applications. The Special Materials Plant at NFC manufactures high volume, low volume, and high purity Special Materials like tantalum, niobium, gallium, indium, etc. for application in electronics, aerospace and defense sectors.

Fabrication of Critical Equipment:

A notable feature at Nuclear Fuel Complex is that, apart from in-house process development, a lot of encouragement is given to the Indian industry for fabrication of plant equipment's and automated systems. Major sophisticated equipment's fabricated in-house at NFC include the slurry extraction system for purification of uranium, high temperature (1750°C), pellet sintering furnace, etc.

Waste Management, Health and Safety:

By means of an elaborately organized program of effluent management, NFC takes scrupulous care in protecting the environment. The Health Physics Unit, the Safety Engineering Division and Environment and Pollution Control Group keeps a continuous watch to ensure that the radioactive and chemical discharges are much below the threshold limits. Rich greenery has been developed at NFC which is being nourished with treated waste water from production plants.

Self-Reliance:

The Nuclear Fuel Complex is an outstanding example of a successful translation of indigenously developed processes to production scale operations. The strong base of self-reliance in the crucial area of nuclear fuel and core components is a great asset to the country in not only supporting the nuclear power program but also in developing a large number of allied and ancillary industries.

Electron Beam Melting



2.1 What is electron beam melting?

Electron beam melting is the process of passing high-power electron beam that produces energy needed for melting the metal. The machine that does this process is called Electron Beam Melting Furnace. The principle of conservation of energy is used in this process. Kinetic Energy of electrons is converted into heat energy, thereby melting the metal.



2.2 How does the process start?

The entire process is carried out in a vacuum. This is so because; if electrons interact with air particles, they lose energy or the interaction might also change their direction. For generation of vacuum, there are four motor pumps involved.

- (i) Rotary Pump— Provides Rough Vacuum ($\sim 10^{-2}$ bar)
- (ii) Roots Pump – Provides Medium Vacuum ($\sim 10^{-3}$ bar, Rotary pump must be switched on before using the Roots Pump.)

- (iii) Diffusion Pump – Provides High Vacuum ($\sim 10^{-5}$ bar, Roots pump must be switched on before using the Diffusion Pump.)
- (iv) Turbo Molecular Pump – Provides High Vacuum ($\sim 10^{-5}$ bar, Diffusion pump must be switched on before using the Turbo Molecular Pump.)

After these pumps are operated, a vacuum ($\sim 10^{-5}$ bar) is generated inside the chambers.

The machine is divided into three chambers:

1. Gun Chamber,
2. Intermediate Chamber and
3. Melting Chamber.



Gun Chamber:

The Gun Chamber consists of a filament generally of tungsten (because of its high melting point). The electrodes of the gun chamber are supplied with an AC voltage which excites the electrons in the filament. Electrons upon excitation leave the filament at a rather slow speed. Now, a high DC voltage is applied to the cathode thereby increasing the speed (~ 100000 km/s) and also the number of electrons emitted. These electrons emitted form a beam which is to be focused on the metal present in the crucible.

$$K.E = \frac{1}{2}mv^2 = \frac{1}{2} * 9.134 * 10^{-31} * (100000 * 1000)^2 = 4.1567 * 10^{-15} J = 25979.35 eV$$

Intermediate Chamber:

This chamber consists of magnetic lenses, which are also controlled using the PLC. The lenses are adjusted to focus the electron beam on the crucible holding the metal. The deflection coils deflect the beam in X-Y direction to cover the entire surface of the metal. Chamber also consists of a viewing glass through which the operator can observe the electron beam and modify the position of the lenses to focus the beam on the crucible.

Melting Chamber:

The melting chamber consists of a crucible (on which the metal is placed to be melted) covered by graphite (high melting point) around it. The electron beam hits the metal and melts it. The metal is placed on a retraction chamber which moves down after the metal is melted thereby taking the molten metal down and making space for the next batch to be worked upon. The retraction chamber is covered by water from all the sides. Therefore, when the molten metal is taken down it solidifies immediately.



Water is supplied as a coolant for the surface of the melting chamber. The water temperature is monitored continuously using Resistor Temperature Detectors (RTDs).

Thermocouples are used in various other operations which involve monitoring of temperature.

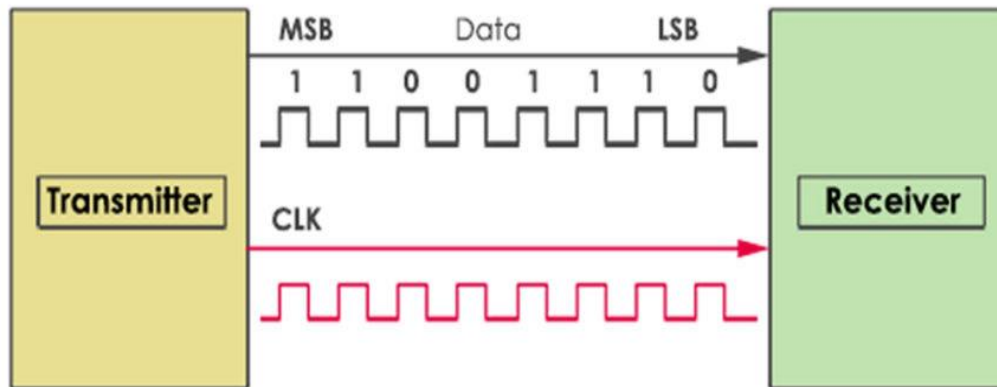
The thermocouple and other sensors send data to the Schneider PLC in the server room.

Communicating between the Client and the Server

Two different types of digital communication techniques are mentioned as follows. Interfaces and protocols for establishing communication between the server and the client are discussed.

3.1 Serial Communication

- It is a type of digital communication in which data is transmitted bit by bit through a common channel.
- As there is only one channel, devices using serial communication cannot send the whole byte at once; instead, 8 bits are sent one by one in series.

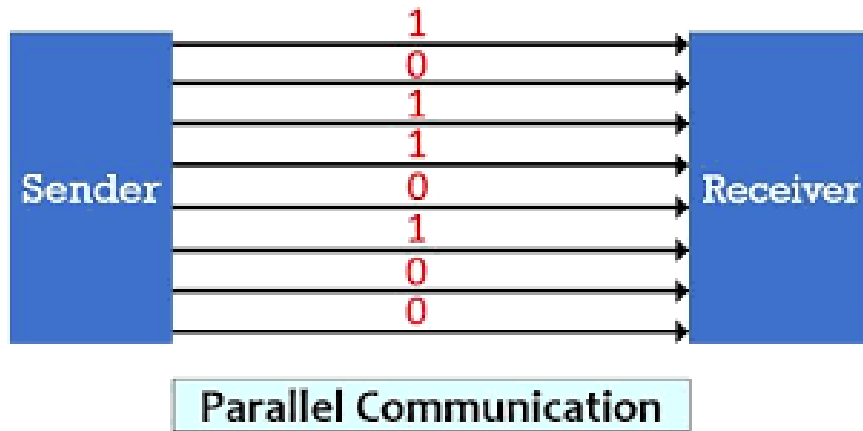


- It is slower but can be implemented in a simple manner. It is a good option for long-distance communication.
- Two channels are used for full-duplex communication. Full duplex communication is two-way communication where information or signal can be transmitted in both directions simultaneously.

3.2 Parallel Communication:

- Parallel communication needs eight separate channels to transfer 8 bits or a byte of data to the receiver.
- The whole byte gets transferred in a single clock cycle, making it relatively faster.
- It is complex when compared to Serial Communication.
- It is efficient for short distances.

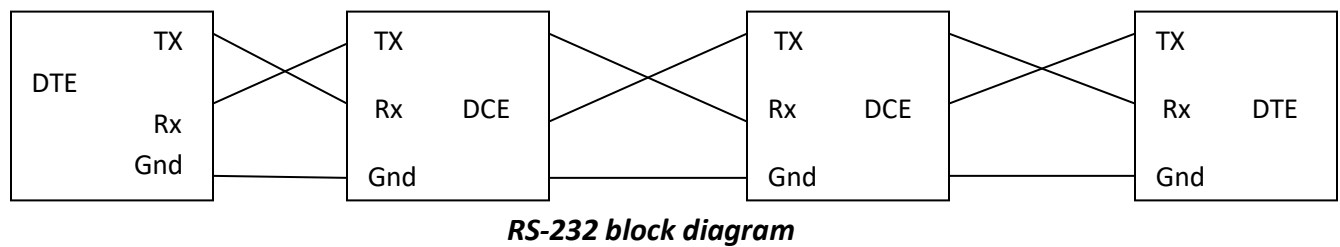
- Due to multiple channels, data is vulnerable to interference such as crosstalk or skewing.
 - **Crosstalk:** When the bits skip into another channel.
 - **Skewing:** The receiver has to wait for the slowest bit to complete the data transfer. This is so because in Parallel Communication each bit must be received in the same pattern.



3.3 Communication Protocols & Standards:

➤ RS-232:

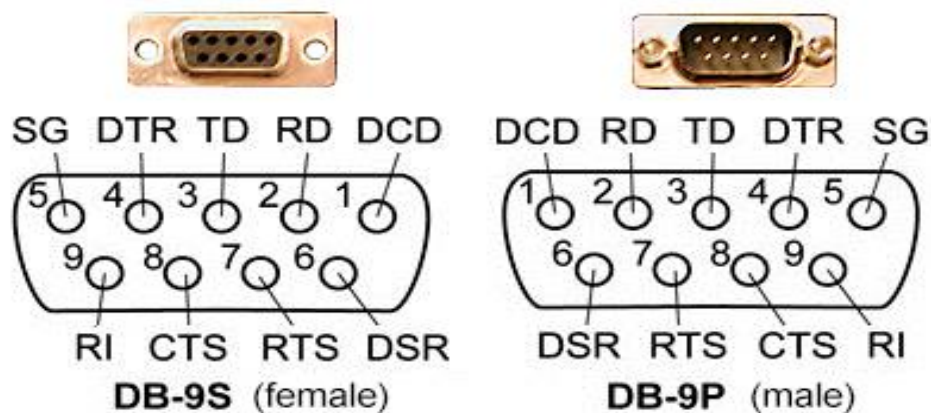
- RS-232, or Recommended Standard 232, was initially introduced in 1960 for serial communication data transmission.
- RS-232 is intended to perform the transmission of data between two terminals also called Data Terminal Equipment (DTE). Data Communication Equipment or Data Circuit-Terminating Equipment such as Modem is used to improve communication between the two DTEs.
- RS-232 interface operates in full-duplex mode, allowing us to send and receive information simultaneously as different lines are used for transmitting and receiving.



Information on the RS-232 interface is transmitted digitally by logical 0 and 1. Logical 1 corresponds to a voltage from **-3V to -15V** and logical 0 corresponds to a voltage from **+3 to +15V**.

For connection to RS-232, a DB-9 connector is used.

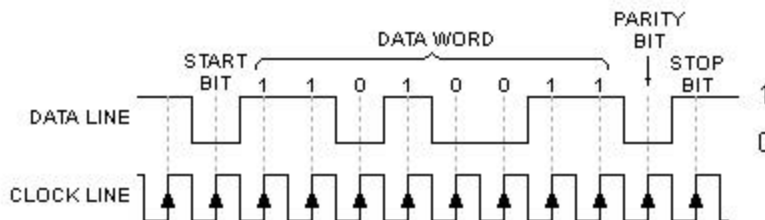
The standard does not define a maximum cable length but defines the maximum capacitance that a compliant drive circuit must tolerate. A widely used rule of thumb indicates that cables more than 15 m (50 ft) long will have too much capacitance unless special cables are used. By using low-capacitance cables, communication can be maintained over more considerable distances up to about 300 m.



Pin Configuration of DB-9

1. CD –Carrier Detect
2. RD – Receive Data
3. TD – Transmit Data
4. DTR – Data Terminal Ready
5. SG – System Ground
6. DSR – Data Set Ready
7. RTS – Request to Send
8. CTS – Clear to Send
9. RI – Ring Indicator

Structure of transmitted data in RS-232:

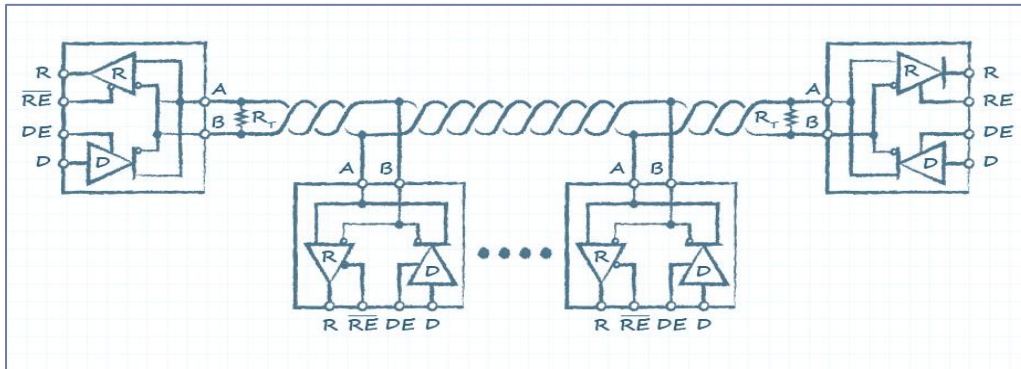


- The start bit is the denoting the beginning of the transmission.
- Data bits-5,6,7 or 8 bits of data.
- Parity – A bit intended for parity checking. Serves for detecting errors. It can take the following values:
 - i. **The parity (EVEN)** takes such a value that the number of units in the message is even
 - ii. **Oddness (ODD)**, takes on such a value that the number of units in the message is odd
 - iii. **Always 1 (MARK)**, the parity bit will always be 1
 - iv. **Always 0 (SPACE)**, the parity bit will always be 0
 - v. **Not used (NONE)**

➤ RS-485

- RS-485 is a standard defining the electrical characteristics of drivers and receivers for use in serial communication systems.

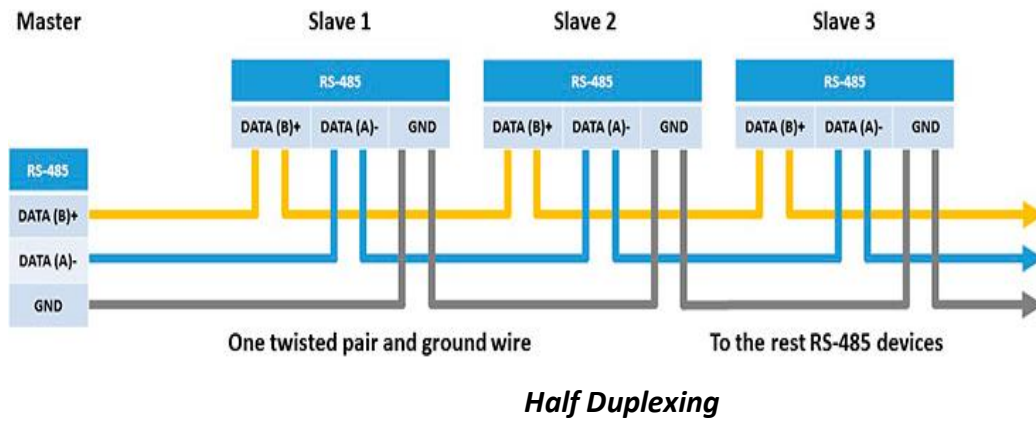
- Multiple devices can be connected at once over a communication bus. It can handle up to 32 devices at once.
- RS-485 can provide speeds of up to 10 Mbps with a maximum cable length of 4000ft.



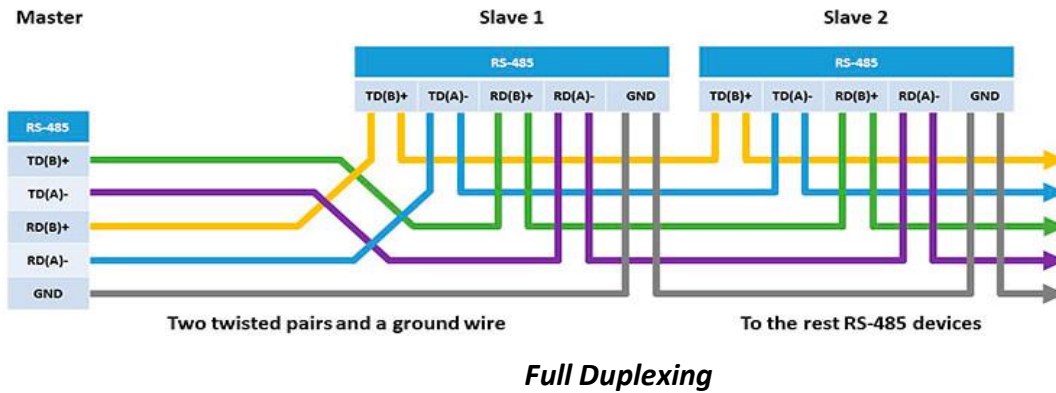
Typical RS-485 network topology

- Data is transmitted via two wires twisted together called the “*Twisted Pair Cable*”. This twisted pair cable provides immunity against electrical noise.
- RS-485 when used along with Modbus can be used with different devices from different manufacturers to be integrated into the main system.
- Most Modbus applications use RS-485 due to the allowance of longer distances, higher speeds and multiple devices on a single network.
- **Duplexing in RS-485:**
- With two contacts, RS-485 operates in half duplex mode and with four contacts; it operates in full duplex mode.
- 120 ohm network termination resistors placed at the ends of an RS-485 twisted-pair communications line help to eliminate data pulse signal reflections that can corrupt the data on the line. The terminations also include pull up and pull down resistors to establish fail-safe bias for each data wire for the case when the lines are not being driven by any device. This way, the lines will be biased to known voltages and nodes will not interpret the noise from undriven lines as actual data; without biasing resistors, the data lines float in such a way that electrical noise sensitivity is greatest when all device stations are silent or unpowered
- These are used in programmable logic controllers and on factory floors. RS-485 is used as the physical layer underlying many standard and proprietary automation protocols used to implement industrial control systems, including the most common versions of Modbus.

- Half Duplexing:



- Full Duplexing:



The OSI Model of RS-485:

The Open Systems Interconnection (OSI) model attempts to characterize the various layers of a communication system from the final application down through the electrical layers and lastly onto the physical layer.



1. Physical Layer of the OSI Model:

The physical layer of the OSI model is responsible for the transfer of raw data between a device and a physical transmission medium. It handles the conversion of electrical signals into digital data, while defining voltages, timing, data rates, etc.

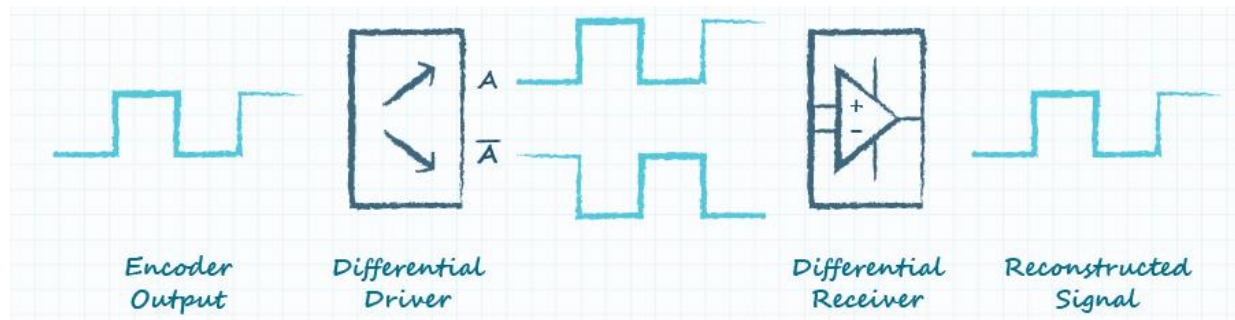
RS-485 uses two signal lines, 'A' and 'B', which must be balanced and differential. Balanced signals are two lines that share a pair in a twisted pair cable with the same impedance on each line. Along with matched impedance of the lines, there must also be matched impedance at the receiver and transmitter.

Balanced cabling allows for noise reduction when using differential signals. These signals, 'A' and 'B' are referred to as a differential pair; one of the signals matches the original signal while the other is entirely inverted, which is why it is sometimes referred to as a complementary signal.

Differential Signaling:

- When communicating over long cabling distances voltages tend to drop, slew rate decreases and signal error often occurs.

- Thereby we use a differential pair. In a differential application, the host generates the original single-ended signal which then goes into a differential transmitter. The transmitter creates the differential pair to be sent out over the cabling.
- With two signals generated, the receiver no longer references the voltage to ground, but instead references the signals to each other. The receiver is now always looking at the difference between the two signals.
- The differential receiver then reconstructs the pair of signals back into one single-ended signal that can be interpreted by the host. All of this is done to avoid signal degradation which would have occurred with single-ended application over long cabling distances.



Encoder output driven by a differential driver and reconstructed by receiver

- Signal degradation is not the only issue that arises over long cabling distances. The longer the cabling is within a system, the higher the chances that electrical noise and interference will make its way onto the cables and, ultimately, into the electrical system.
- When noise couples onto cabling, it shows up as voltages of varying magnitudes, but the benefit of using a balanced twisted pair cable is that the noise couples to the cable equally on each line. For example, a positive 1-volt spike would result in +1V on A and +1V on B. Because the differential receiver subtracts the signals from each other to get the reconstructed signal, it ignores the noise shown equally on both wires.



Differential receiver ignoring noise common to both signals

- One more significant advantage regarding the physical layer of RS-485 is the signal voltage specification. RS-485 does not require the use of a specific bus voltage but instead specifies the minimum required differential voltage, which is the difference between the signal A and B voltages. The bus requires a minimum differential voltage of $\pm 200\text{mV}$ at the receiver and generally, all RS-485 devices will have the same input voltage range despite transmitting at various voltages.

2. Data Link Layer of the OSI Model:

- RS-485 is a duplex communication system in which multiple devices on the same bus can communicate in both directions. RS-485 is most often used as a half-duplex, as shown in the figures above, with only a single communication line ('A' and 'B' as a pair). In half-duplex, devices take turns using the same line where the host will assert control of the bus and send a command with all other devices listening. The intended recipient will listen for its address and then that device will assert control and respond back.
- At the data layer, we use MODBUS RTU (Remote Terminal Unit) for the project.
- When controllers are set up to communicate on a Modbus network using RTU (Remote Terminal Unit) mode, each 8-bit byte in a message contains two 4-bit hexadecimal characters. The main advantage of this mode is that its greater character density allows better data throughput than ASCII mode for the same baud rate. Each message must be transmitted in a continuous stream.
- The format for each byte in RTU mode is:
 - **Coding System:** 8-bit binary, hexadecimal 0–9, A–F Two hexadecimal characters contained in each 8-bit field of the message
 - **Bits per Byte:** 1 start bit, 8 data bits, least significant bit sent first, 1 bit for even/odd parity; no bit for no parity, 1 stop bit if parity is used; 2 bits if no parity.
 - **Error Check Field:** Cyclical Redundancy Check (CRC)

Start	Address	Function	Data	CRC Check	END
T1–T2–T3–T4	8 BITS	8 BITS	n x 8 BITS	16 BITS	T1–T2–T3–T4

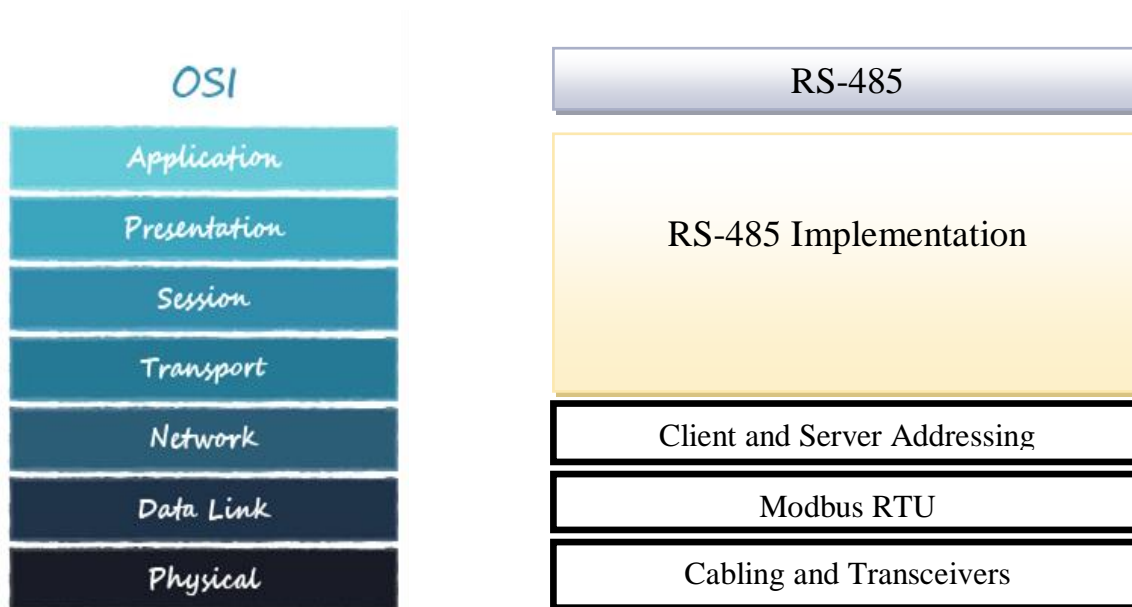
3. Network Layer of the OSI Model:

- The network layer deals with the actual communication between devices that occur on the RS-485 bus.

- There is no set specification for addressing the network layer, but the RS-485 bus must be appropriately managed by a server to avoid bus collisions. Bus collisions occur when multiple devices attempt to communicate at once, which can harm the network. When collisions occur, transmitters clash on both ends and effectively both create shorts. This causes each device to draw large amounts of current that could put the transceiver into thermal shutdown.
- To avoid collisions, the server controls the bus and will make calls to individual devices. This is achieved by having specific addresses for each device or specific commands for each individual device. Since the bus is shared between all devices, every device will see the command/address being sent by the server but will only respond when that individual device is asserted.

4. Application Layer of the OSI Model:

- RS-485 is well contained within the first three layers of the OSI model with the actual implementation of the bus being characterized within the application layer. This layer covers the addresses or command sets used by the devices as well as the interpretation of the data. Simply stated, the application layer is the implementation of the RS-485 bus.
- As the RS-485 standard only defines the physical and data link layers with an addressing requirement, the application layer can adopt various proprietary or open communication protocols.



- **Modbus RTU:** Modbus RTU consists of the address of the client or client ID, the function code, data, and CRC of the checksum.

Start	Address	Function	Data	CRC Check	END
T1–T2–T3–T4	8 BITS	8 BITS	n x 8 BITS	16 BITS	T1–T2–T3–T4

Client ID	Function Code	Data	CRC
-----------	---------------	------	-----

- **Client ID** is the address of the device. It can take a value from 0-247. 248-255 are reserved.
- The individual client devices or clients are assigned addresses in the range of 1-247. A server addresses a client by placing the client address in the address field of the message. When the client sends its response, it places its own address in this address field of the response to let the server know which client is responding.
- Address 0 is used for the broadcast address, which all client devices recognize.
- The **function code** field of a message frame contains eight bits in the case of RTU. Valid codes are in the range of 1-255 decimals. When a message is sent from a server to a client device, the function code field tells the client what kind of action it needs to perform. For example, the function code could ask the client device to read ON/OFF states of a group of discrete coils or inputs; to read the data contents of a group of registers; to read the diagnostic status of the client; to write the designated coils or registers; or to allow loading, recording or verifying the program within the client.
- In this project we will be reading the data received from the holding registers which has the function code of $(0000\ 0011)_2$ which is 03_{hex} .
- If the client device takes the requested action without error, it returns the same code in the response. If the exception occurs, it would return 1000 0011 as the function code if the function code initially was 0000 0011. In addition to its modification of the function code for an exception response, the client places a unique code into the data field of the response message which tells the server what kind of error has occurred or the reason for the exception.
- The **data field** is constructed using sets of two hexadecimal digits, in the range of 00 to FF.
- This field of messages sent from a server to client devices contains additional information the client must use to take the action defined by the function code. For example, it can

include items like discrete and register addresses, the quantity of items to be handled and the count of actual data bytes in the field.

- If the server requests the client to read a group of holding registers (function code 03_{hex}), the data field specifies the starting register and how many registers are to be read. If the server writes to a group of registers in the client (function code 10_{hex}), the data field specifies the starting register, number of registers to write, the count of data bytes that follow in the data field and the data to be written into the registers. If no error occurs, the data field of a response from a client to a server contains the data requested. If an error occurs, the field contains an exception code that the server application can use to determine the next action to be taken.
- In RTU mode, messages include an error-checking field that is based on a **Cyclical Redundancy Check (CRC)** method. The CRC field checks the contents of the entire message.
- The CRC field is of two bytes, containing a 16-bit binary value. The CRC value is calculated by the transmitting device, which appends the CRC to the message. The receiving device recalculates a CRC upon receiving the message and compares the calculated value to the actual value it received in the CRC field. If the two values are not equal, an error results.
- The CRC is started by first loading a 16-bit register with 1's. Then a process begins of applying successive 8-bit bytes of the message to the current contents of the register. Only the eight bits of data in each character are used for generating the CRC. Start, stop and parity bits do not apply to the CRC.
- Each 8-bit character is exclusive ORed with the register contents. Then the result is shifted in the direction of the least significant bit (LSB), with a zero filled into the most significant bit (MSB). The LSB is extracted and examined. If the LSB was 1, the register is then exclusive ORed with a preset, a fixed value. If the LSB was a 0, no exclusive OR takes place. This is repeated until eight shifts have been performed. After the last shift, the next 8-bit byte is exclusive ORed with the registers current value and the process repeats for eight more shifts. After all bytes of the message have been applied, the final content of the register is the CRC value.

Modbus Function Formats:

REGISTER NUMBER	REGISTER ADDRESS hex	TYPE	NAME
1-9999	0000 to 270E	Read-write	Discrete Output Coils
10001-19999	0000 to 270E	Read	Discrete Input Contacts
30001-39999	0000 to 270E	Read	Analog input registers
40001 – 49999	0000 to 270E	Read-write	Analog O/P holding registers

The holding registers 40001 is addressed as register 0000 in the data address field of the message. The function code already specifies a ‘holding register’ operation.

The following is an example of a request to read registers 40108-40110 from client device 17.

Query:

Field Name	Hex	RTU Field
Client Address	11	0001 0001
Function Code	03	0000 0011
Starting Address Hi	00	0000 0000
Starting Address Lo	6B	0110 1011
No of registers Hi	00	0000 0000
No of registers Lo	03	0000 0011
Error Check		-

Request: 11 03 006B 0003 (CRC)

$006B_{\text{hex}} = 107_{10} = (40108 - 40001)$

0003 = the number of required registers (40108 to 40110)

Response:

Field Name	Hex	RTU Field
Client Address	11	0001 0001
Function Code	03	0000 0011
Byte Count	06	0000 0110
Data Hi (Register 40108)	02	0000 0010
Data Lo (Register 40108)	2B	0010 1011
Data Hi (Register 40109)	00	0000 0000
Data Lo (Register 40109)	00	0000 0000
Data Hi (Register 40110)	00	0000 0000
Data Lo (Register 40110)	64	0110 0100
Error Check		-

Response: 11 03 06 022B 0000 0064 (CRC)

The following table shows the function codes supported by various controllers.

In the project we are using the function code 03 for reading the holding registers.

Code	Name
01	Read Coil Status
02	Read Input Status
03	Read Holding Registers
04	Read Input Registers
05	Force Single Coil
06	Preset Single Register
07	Read Exception Status
08	Diagnostics
09	Program 484
10	Poll 484
11	Fetch Comm. Event Ctr.
12	Fetch Comm. Event Log
13	Program Controller
14	Poll Controller
15	Force Multiple Coils
16	Preset Multiple Registers
17	Report Client ID
18	Program 884/M84
19	Reset Comm. Link
20	Read General Reference
21	Write General Reference
22	Mask Write 4X Register
23	Read/Write 4X Registers
24	Read FIFO Queue

For communicating between the client and the computer over an Ethernet network we need to encapsulate the Modbus RTU with **Modbus TCP/IP**.

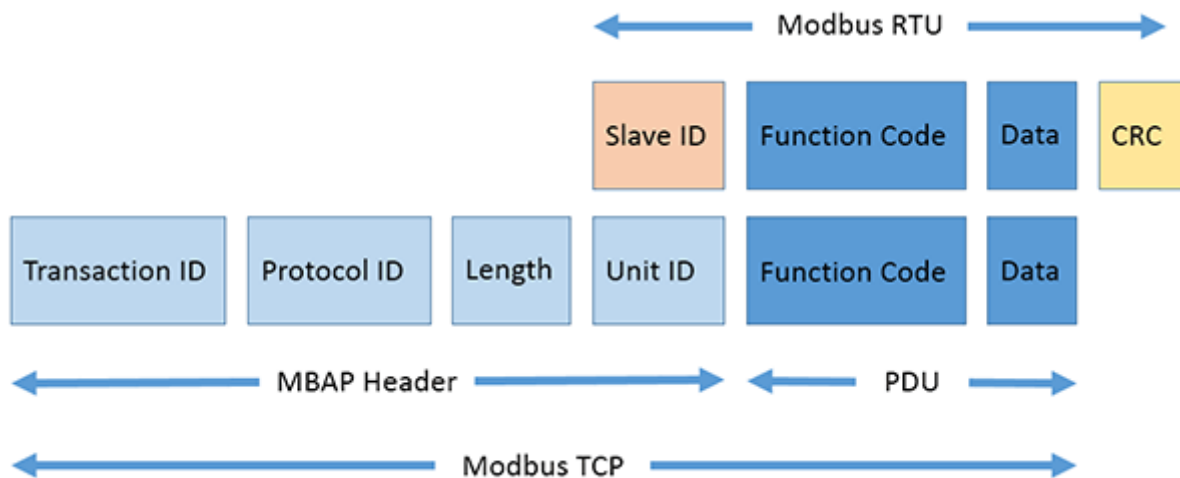
➤ **Modbus TCP/IP:**

In an Ethernet network, the device address is its IP address. Typically, devices are on the same subnet, where IP addresses differ by the last two digits 192.172.100.**100** when using the most common subnet mask 255.255.255.0. The interface is an **Ethernet** network; the data transfer protocol is **TCP / IP**. The TCP port used is: **502**.

The Modbus TCP command consists of a portion of Modbus RTU message and a special header.

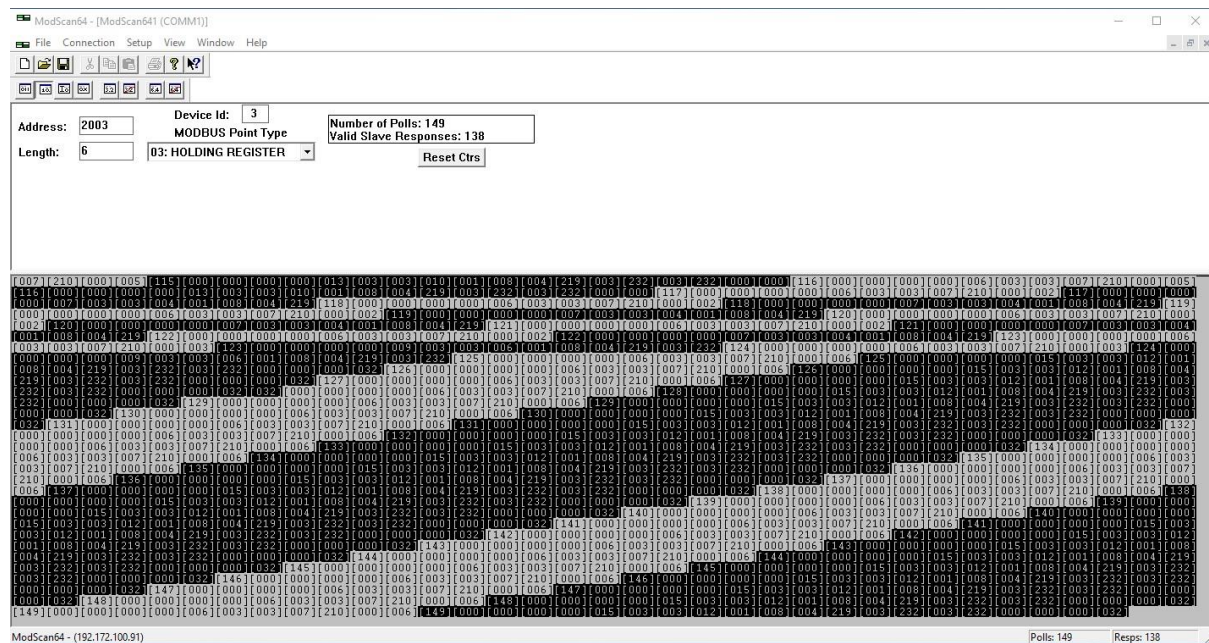
From the Modbus RTU message, if the client ID and the CRC are removed; we get the Protocol Data Unit (PDU).

At the beginning of the received PDU, a new 7-byte header is added which is called MBAP header (Modbus Application Header).



- **Transaction Identifier:** 2 bytes are set by the server to uniquely identify each request. These bytes are repeated by the client device in the response as the responses of the client device may not always be received in the same order as that of the requests.
- **Protocol Identifier:** 2 bytes are set by the server. It will always be 00 00 which corresponds to the Modbus protocol.
- **Length:** 2 bytes are set by the Server, identifying the number of bytes in the message that follow. It is counted from Unit Identifier to the end of the message.
- **Unit Identifier:** 1 byte is set to Server. It is repeated by the Client device to uniquely identify the Client device.

ModScan software can be used to check for communication between the client and the server.



The connection is established with the Modbus gateway. The address of the register from where data is to be fetched is written along with the client ID.

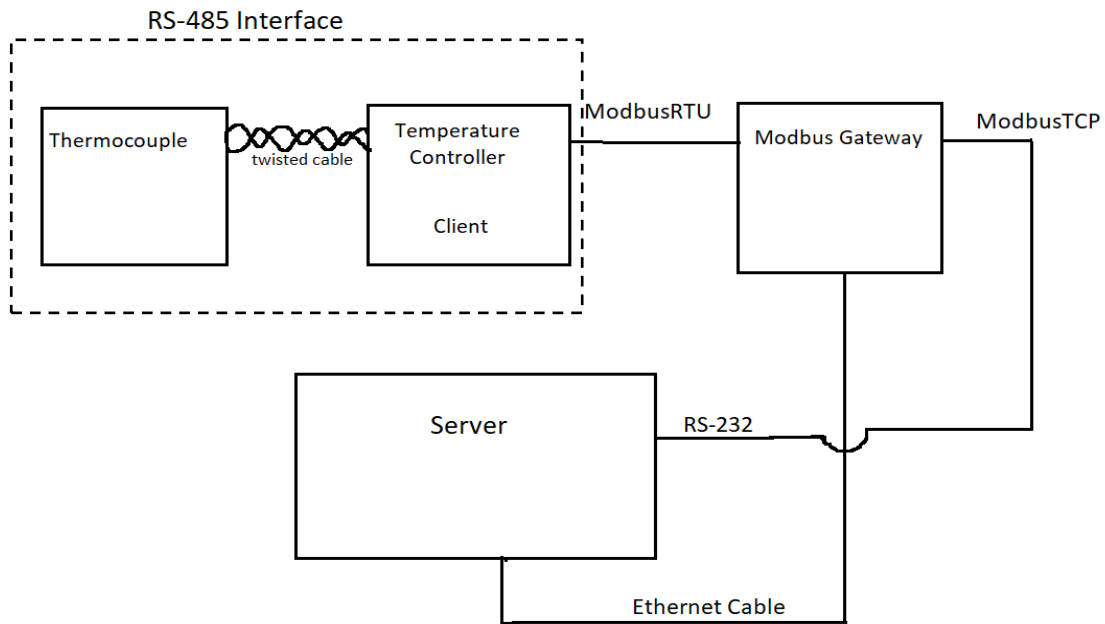
The above picture shows the data sent by the server and data received from the client.

➤ Ethernet:

- Ethernet is a network protocol that controls how data is transmitted over a LAN and is referred to as the IEEE 802.3 protocol. The protocol has evolved and improved over time to transfer data at the speed of more than a gigabit per second. It is a widely used LAN protocol, which is also known as Alto Aloha Network.
- It offers a simple user interface that helps to connect various devices easily, such as switches, routers and computers. A local area network (LAN) can be created with the help of a single router and a few Ethernet cables which enable communication between all linked devices.

The Modbus gateway is connected to the server via Ethernet.

3.4 Block Diagram:



PyModbusTCP and other Python Libraries for plotting real-time data received from client

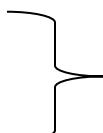
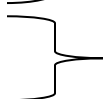
4.1

Python version 3.11.1 is used for the project.

Python programming language is used because of its flexibility, simplicity as well as its collection of libraries such as matplotlib, pandas, pymodbusTCP etc.

Python has found its extensive industrial applications in the areas of Electronics & Communication Engineering such as signal processing, image processing, control system engineering, embedded systems, data visualization, automation and the list is extensive.

The following libraries were used for obtaining the desired output:

- | | | |
|--------------------|---|---|
| ▪ PyModbusTCP |  | Require Installation |
| ▪ Matplotlib 3.5.2 | | |
| ▪ Pandas | | |
| ▪ csv |  | Comes inbuilt within the Python Package |
| ▪ time | | |

The project consists of two python files 1. *modbus.py* which fetches data from the temperature controller and stores the data in the csv files and 2. *plotting.py* which when run fetches data from the csv file while it is getting stored using Pandas and plots it simultaneously using Matplotlib.

There are two csv files in which the data is stored. One, through which the data is fetched for plotting *{real_time_data.csv}* and the other which stores data fetched from the sensor since the first ever operation *{all_time_data.csv}*.

The *real_time_data.csv* file has 4 headers i.e., **Data, Time, Temperature and Set Point**. Giving headers to the csv file enables the user to fetch data using pandas which is done in *plotting.py*.

The all_time_data.csv file stores data in the form of an object with parameters “{‘Date’, ‘Time’, ‘Temperature’, ‘Set-Point’}”.

	A	B	C	D	E	F	G	H
1	{‘Date’: ‘30/12/2022’, ‘Time’: ‘10:57:25’, ‘Temperature’: 27.6, ‘Setpoint’: 124.3}							
2								
3	{‘Date’: ‘30/12/2022’, ‘Time’: ‘10:57:27’, ‘Temperature’: 27.6, ‘Setpoint’: 124.3}							
4								
5	{‘Date’: ‘30/12/2022’, ‘Time’: ‘10:57:29’, ‘Temperature’: 27.6, ‘Setpoint’: 124.3}							
6								
7	{‘Date’: ‘30/12/2022’, ‘Time’: ‘10:57:31’, ‘Temperature’: 27.6, ‘Setpoint’: 124.3}							
8								
9	{‘Date’: ‘30/12/2022’, ‘Time’: ‘10:57:33’, ‘Temperature’: 27.6, ‘Setpoint’: 124.3}							
10								
11	{‘Date’: ‘30/12/2022’, ‘Time’: ‘10:57:35’, ‘Temperature’: 27.6, ‘Setpoint’: 124.3}							
12								
13	{‘Date’: ‘30/12/2022’, ‘Time’: ‘10:57:37’, ‘Temperature’: 27.6, ‘Setpoint’: 124.3}							
14								
15	{‘Date’: ‘30/12/2022’, ‘Time’: ‘10:57:39’, ‘Temperature’: 27.6, ‘Setpoint’: 124.3}							
16								
17	{‘Date’: ‘30/12/2022’, ‘Time’: ‘10:57:41’, ‘Temperature’: 27.6, ‘Setpoint’: 124.3}							
18								
19	{‘Date’: ‘30/12/2022’, ‘Time’: ‘10:57:43’, ‘Temperature’: 27.6, ‘Setpoint’: 124.3}							
20								
21	{‘Date’: ‘30/12/2022’, ‘Time’: ‘10:57:45’, ‘Temperature’: 27.6, ‘Setpoint’: 124.3}							

all_time_data.csv

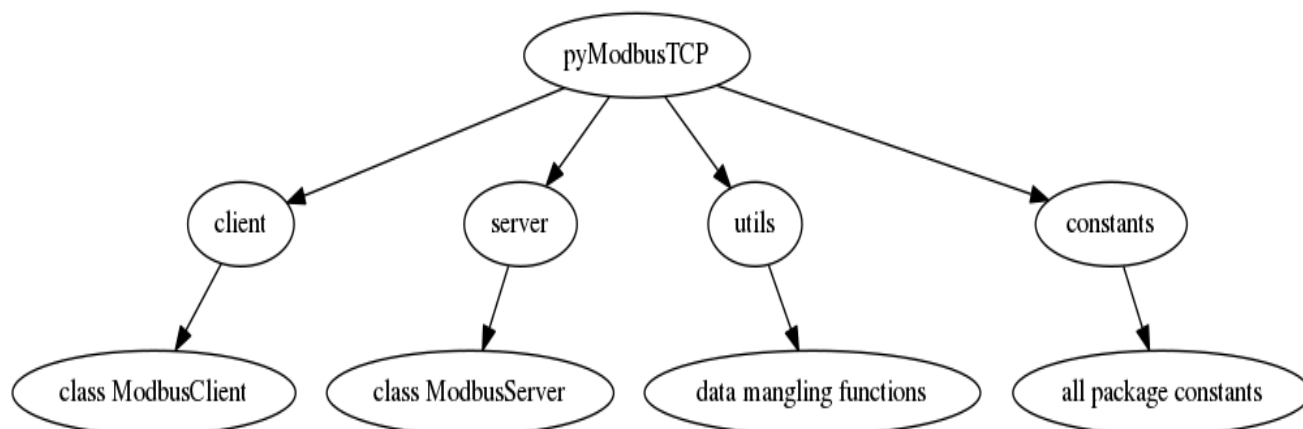
	A	B	C	D	E
1	Date	Time	Temperature	Set-Point	
2					
3	01-01-2023	10:38:25	27.1	124.3	
4	01-01-2023	10:38:35	27.9	124.3	
5	01-01-2023	10:38:45	28.2	124.3	
6	01-01-2023	10:38:55	28.9	124.3	
7	01-01-2023	10:39:05	29.5	124.3	
8	01-01-2023	10:39:15	29.9	124.3	
9	01-01-2023	10:39:25	30.5	124.3	
10	01-01-2023	10:39:36	30.9	124.3	
11	01-01-2023	10:39:46	28.6	124.3	
12	01-01-2023	10:39:56	28.7	124.3	
13					

real_time_data.csv

➤ 4.2PyModbusTCP:

PyModbusTCP gives access to Modbus/TCP server through the ModbusClient object.

PyModbusTCP is pure python code without any extension or external module dependency.



- `class pyModbusTCP.client.Modbus.client (host = 'localhost', port = 502, unit_id = 3, timeout = 30.0, debug = False, auto_open = True, auto_close = False)`

Parameters:	<ul style="list-style-type: none">• host (<i>str</i>) – hostname or IPv4/IPv6 address server address• port (<i>int</i>) – TCP port number• unit_id (<i>int</i>) – unit ID• timeout (<i>float</i>) – socket timeout in seconds• debug (<i>bool</i>) – debug state• auto_open (<i>bool</i>) – auto TCP connect• auto_close (<i>bool</i>) – auto TCP close)
Returns:	Object ModbusClient
Return type:	ModbusClient

The following objects can be called using the ModbusClient class:

- `read_coils(bit_addr, bit_nb=1)` function code 0x01

Parameters:	<ul style="list-style-type: none"> • bit_addr (<i>int</i>) – bit address (0 to 65535) • bit_nb (<i>int</i>) – number of bits to read (1 to 2000)
Returns:	bits list or None if error
Return type:	list of bool or None

- `read_discrete_inputs(bit_addr, bit_nb = 1)` function code 0x02

Parameters:	<ul style="list-style-type: none"> • bit_addr (<i>int</i>) – bit address (0 to 65535) • bit_nb (<i>int</i>) – number of bits to read (1 to 2000)
Returns:	bits list or None if error
Return type:	list of bool or None

- `read_holding_registers(reg_addr, reg_nb=1)` function code 0x03

Parameters:	<ul style="list-style-type: none"> • reg_addr (<i>int</i>) – register address (0 to 65535) • reg_nb (<i>int</i>) – number of registers to read (1 to 125)
Returns:	registers list or None if fail
Return type:	list of int or None

```
rr = client.read_holding_registers(0x7D2, 2, unit = 3)
```

Here, 0x7D2 is the address of the register from which data is being read. '2' is the number of registers that are to be read i.e., registers of address 0x7D2 and 0x7D3. Unit =3 is the client ID of the sensor.

- `write_multiple_coils(bits_addr, bits_value)` function code 0x0f

Parameters:	<ul style="list-style-type: none"> • bits_addr (<i>int</i>) – bits address (0 to 65535) • bits_value (<i>int</i>) – bits values to write
Returns:	True if write is done
Return type:	bool

- write_multiple_registers(*reg_addr*, *regs_value*) function code 0x10

Parameters:	<ul style="list-style-type: none"> • reg_addr (<i>int</i>) – register address (0 to 65535) • reg_nb (<i>int</i>) – number of registers to read (1 to 125)
Returns:	True if write ok
Return type:	bool

- write_single_coil(*bit_addr*, *bit_value*) function code 0x05

Parameters:	<ul style="list-style-type: none"> • bits_addr (<i>int</i>) – bits address (0 to 65535) • bits_value (<i>int</i>) – bits values to write
Returns:	True if write is done
Return type:	bool

- write_single_register(*reg_addr*, *reg_value*) function code 0x06

Parameters:	<ul style="list-style-type: none"> • reg_addr (<i>int</i>) – register address (0 to 65535) • reg_nb (<i>int</i>) – number of registers to read (1 to 125)
Returns:	True if write ok
Return type:	bool

- **class PyModbusTCP.server.ModbusServer**(host = 'localhost', port = 502, no_block = False, ipv6 = False, data_bank = None, data_hdl = None, ext_engine = None)

Parameters:

- **host** (*str*) – hostname or IPv4/IPv6 address server address (default is 'localhost')
- **port** (*int*) – TCP port number (default is 502)
- **no_block** (*bool*) – no block mode, i.e. start() will return (default is False)
- **ipv6** (*bool*) – use ipv6 stack (default is False)
- **data_bank** (DataBank) – instance of custom data bank, if you don't want the default one (optional)
- **data_hdl** (DataHandler) – instance of custom data handler, if you don't want the default one (optional)
- **ext_engine** (*callable*) – an external engine reference (ref to ext_engine(session_data)) (optional)

- **PyModbusTCP.utils:** Provides a set of functions for modbus data mangling. Data mangling is a technique by which we can resolve the object variable naming conflicts in classes and subclasses.

- **pyModbusTCP.utils.byte_length**(*bit_length*)

Parameters:	bit_length (<i>int</i>) – the number of bits
Returns:	the number of bytes
Return type:	int

- **pyModbusTCP.utils.get_bits_from_int (val_int, val_size=16)**

Parameters:	<ul style="list-style-type: none"> • val_int (<i>int</i>) – integer value • val_size (<i>int</i>) – bit length of integer (word = 16, long = 32) (optional)
Returns:	list of boolean “bits” (the least significant first)
Return type:	list

- **pyModbusTCP.utils.set_bit (value, offset)**

Parameters:	<ul style="list-style-type: none"> • value (<i>int</i>) – value of integer where set the bit • offset (<i>int</i>) – bit offset (0 is lsb)
Returns:	value of integer with bit set
Return type:	int

- **pyModbusTCP.utils.test_bit (value, offset)**

•

Parameters:	<ul style="list-style-type: none"> • value (<i>int</i>) – value of integer to test • offset (<i>int</i>) – bit offset (0 is lsb)
Returns:	value of bit at offset position
Return type:	bool

- **pyModbusTCP.utils.toggle_bit (value, offset)**

Parameters:	<ul style="list-style-type: none"> • value (<i>int</i>) – value of integer where invert the bit • offset (<i>int</i>) – bit offset (0 is lsb)
Returns:	value of integer with bit inverted
Return type:	int

4.3 Other Libraries used:

- **Matplotlib:**

Matplotlib is a comprehensive library for creating static, animated and interactive visualizations in Python. Most of the Matplotlib utilities lie under the pyplot submodule. Matplotlib consists of several plots like line, bar, scatter, histogram, pie-charts, etc. Matplotlib has numerous submodules which have various classes which can be used for plotting data. This project uses Animation subclass from TimedAnimation sub module of Matplotlib library for plotting of real-time data received from the sensors.

- **Pandas:**

Pandas is a python library which is used for data manipulation and analysis. Pandas is mainly used for data-analysis and associated manipulation of tabular data in DataFrames. Pandas allow importing data from various file formats such as comma-separated values, JSON, Parquet, SQL database table or queries and excel files. They allow various data manipulation operations such as merging, reshaping, data cleaning and data wrangling features. It is built upon another library i.e., NumPy which is a library for the Python programming language, adding support for large, multi-dimensional arrays and matrices, along with a large collection of high-level mathematical functions to operate on these arrays.

4.4 Python Code for fetching data from the temperature sensor:

```
'''
Sarvagya Ranjan
'''

from pymodbus.client import ModbusTcpClient
import time
import csv

host = '192.172.100.91' #ip adress of gateway
port = 502

#communication between client and server
client = ModbusTcpClient(host, port)
client.connect()

#header of csv file for real-time analysis
fieldnames = ["Date", "Time", "Temperature", "Set-Point"]

withopen
("C:/Users/DELL/AppData/Local/Programs/Python/Python311/real_time_data.csv
", 'w') as csv_file:
csvwriter = csv.writer(csv_file)

csvwriter.writerow(fieldnames)


while True:
    #read the holding registers
    rr = client.read_holding_registers(0x7D2, 2, unit = 3)
    time.sleep(10)
    print(rr)
    #print(rr.registers)

    t = rr.registers[0]
    sp = rr.registers[1]
    temp = t/10
    setp = sp/10
    #date and time of when the reading is taken from the sensor
    date = time.localtime().tm_mday
    month = time.localtime().tm_mon
    year = time.localtime().tm_year

    hrs = time.localtime().tm_hour
    mins = time.localtime().tm_min
```

```

secs = time.localtime().tm_sec

date_Arr = (str(date)+'/'+str(month)+'/'+str(year))
time_Arr = (str(hrs) + ':' + str(mins) + ':' + str(secs))

info = {
    "Date":date_Arr,
    "Time":time_Arr,
    "Temperature": temp,
    "Setpoint":setp
}

#real_time analysis csv file
withopen("C:/Users/DELL/AppData/Local/Programs/Python/Python311/
real_time_data.csv", 'a') as csv_file:
    writer = csv.writer(csv_file, delimiter = ',')
writer.writerow([date_Arr, time_Arr, temp,setp])

#all time data csv file
With open
("C:/Users/DELL/AppData/Local/Programs/Python/Python311/all_time_data
.csv", 'a') as csv_file:
    writer = csv.writer(csv_file, delimiter = ',')
writer.writerow([info])

```

4.4 Python code for plotting data while it is getting stored in a csv file:

```
'''
Sarvagya Ranjan
'''

from itertools import count
from matplotlib.animation import FuncAnimation
import csv
import pandas as pd
import matplotlib.pyplot as plt
import time

#for obtaining the date on label
date = time.localtime().tm_mday
month = time.localtime().tm_mon
year = time.localtime().tm_year

date_Arr = (str(date)+'/'+str(month)+'/'+str(year))

def animate (i):
    #read csv data using panda
    data =
pd.read_csv("C:/Users/DELL/AppData/Local/Programs/Python/Python311/real_time_data.csv")
    x = data["Time"]
    y1 = data["Temperature"]
    y2 = data["Set-Point"]

    #plotting parameters
plt.cla() #clear axis
plt.plot(x,y1, 'r-o', label = ("Temperature" + " " + date_Arr))
plt.xlabel("Time", fontsize = 10)
plt.ylabel("Temperature in °C", fontsize = 10)
plt.title("Temperature °C vs Time (Real-Time Analysis)", fontsize = 15)
plt.legend()

ani = FuncAnimation(plt.gcf(), animate, interval = 2000)

plt.tight_layout()
plt.show()
```

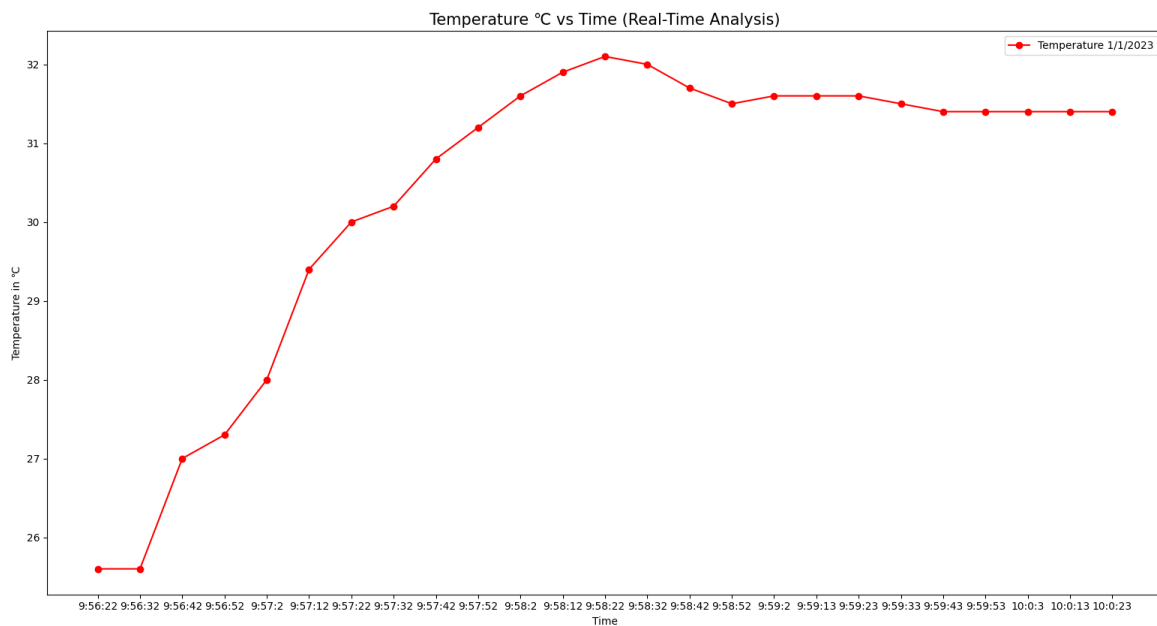
FuncAnimation which is a subclass of TimedAnimation from the matplotlib.animation, makes an animation by repeatedly calling a function func. Animate function is defined which fetches data from the csv file using pandas. The plotting parameters are defined within the function. FuncAnimation is called with the following parameters:

- plt.gcf() i.e. get current figure, shows the last figure that was created.
- animate function is called every 2 seconds for plotting the data obtained from the csv file.
- Plt.tight_layout () automatically adjusts parameters to give specified padding

```
class matplotlib.animation.FuncAnimation(fig, func, frames=None, init_func=None, fargs=None, save_count=None, *, cache_frame_data=True, **kwargs)
```

- **fig:** It is the figure object used for drawing, resizing or any other needed events. Any additional positional arguments can be supplied via the fargs parameter.
- **func:** It is the callable function that gets called each time. The next value in frames is given through the first argument. Any other additional positional arguments is given via the fargs parameter. If the blit value is equal to True then, func returns an iterable of all artists that are to be modified or created. This data is used by the blitting algorithm to decide which parts of the figure has to be updated. If blit== False then the value returned is unused or omitted.
- **frames:** It is either an iterable, an integer, a generator function or None. It is an optional argument. It is the data source to be passed to func and every frame of the animation.
- **init_func:** It is an optional callable function that is used to draw a clear frame.
- **fargs:** It is an optional parameter that is either a tuple or None, which is an additional arguments that needs to be passed to each call to func.
- **save_count:** It is an integer that acts as fallback for the number of values from frames to cache. This is only used if the number of frames cannot be inferred from frames, i.e. when it's an iterator without length or a generator. Its default is 100.
- **interval:** It is an optional integer value that represents the delay between each frame in milliseconds. Its default is 100.
- **repeat_delay:** It is an optional integer value that adds a delay in milliseconds before repeating the animation. It defaults to None.
- **blit:** It is an optional boolean argument used to control blitting to optimize drawing.
- **cache_frame_data:** It is an optional boolean argument used to control caching of data. It defaults to True.

4.6 Plot:



Plot of data received every 10 seconds from the sensor

Screen Recording of the plot: https://drive.google.com/file/d/1VU7qcInjorCEeoenxTqhD40CYfOhd1o/view?usp=share_link