Accura
Tequipment

Embedded Full Stack IoT

PREFACE

In today's rapidly evolving technological landscape, embedded systems and the Internet of Things (IoT) are driving significant innovation across industries and shaping our everyday lives. This curriculum, tailored specifically for engineering students, provides a comprehensive exploration of both foundational concepts and advanced topics in embedded systems and IoT. It aims to equip aspiring engineers and technologists with the essential knowledge and skills needed to thrive in these dynamic fields.

Embedded systems are specialized computing systems designed to perform specific functions within larger systems. They are pervasive in modern devices, ranging from everyday appliances to sophisticated industrial machinery. The integration of hardware and software in these systems necessitates a deep understanding of both domains, emphasizing efficiency, performance optimization, and adherence to precise application requirements.

IoT extends the capabilities of embedded systems by connecting them to the internet, enabling seamless data exchange and remote control. This connectivity has paved the way for transformative innovations such as smart homes, automated industrial processes, and advanced healthcare solutions. The synergy between embedded systems and IoT is driving the forefront of technological advancements, making it essential for students to grasp the intricacies of these interconnected fields.

This curriculum covers a wide range of topics, including the architectural principles of embedded systems, their fundamental characteristics, and various specialized types tailored to specific applications. It also explores the pivotal role of microcontrollers and microprocessors, providing detailed insights into their functionalities, selection criteria, and integration into IoT solutions. The curriculum is designed not only to impart theoretical knowledge but also to offer practical insights, enabling students to apply their learning effectively in real-world scenarios.

As students embark on this educational journey, the goal is to empower them with the competencies required to excel in the dynamic landscape of embedded systems and IoT. By gaining a comprehensive understanding of the principles and applications of these technologies, students will be well-prepared to contribute innovatively and lead the development of solutions that will shape the future of technology.

This curriculum aims to serve as a valuable resource, inspiring students to explore the limitless potential within embedded systems and IoT. It fosters a deep appreciation for the complexity and impact of these technologies, preparing students for rewarding careers where they can make significant contributions to technological advancements and societal progress.

NOS STANDARDS

The Embedded Full Stack IoT Analyst course is designed to equip participants with essential skills for the rapidly evolving IoT landscape. The course covers the development and testing of IoT system designs (NOS: ELE/N1406), ensuring that participants can create robust designs based on detailed specifications, key component dependencies, and functional parameters. This foundational knowledge ensures that participants are capable of conceptualizing and implementing IoT systems that meet precise requirements and performance standards.

A key aspect of the program is building Graphical User Interfaces (GUIs) and applications for IoT frameworks (NOS: ELE/N1410). Participants will be trained to develop user-friendly and functional interfaces that enhance the usability and performance of IoT systems. This includes learning the principles of GUI design, usability testing, and application development within the IoT ecosystem, ensuring that the end products are both intuitive and efficient.T

The course also emphasizes the importance of testing and troubleshooting firmware (NOS: ELE/N1411). Participants will gain practical experience in validating IoT prototypes, identifying and resolving malfunctions, and conducting root cause analysis. This hands-on training is crucial for maintaining the integrity and reliability of IoT systems, ensuring that participants can effectively troubleshoot and maintain these systems.

In addition to technical skills, the course highlights the significance of health and safety practices (NOS: ELE/N1002). Participants will learn to adhere to essential health and safety protocols, identify potential hazards, and use safety equipment appropriately. This knowledge is vital for maintaining a safe working environment and ensuring compliance with workplace regulations, thus promoting a culture of safety within the workplace.

The course also focuses on enhancing employability skills (NOS: DGT/VSQ/N0102) by fostering strong communication abilities, effective teamwork, and proficient work management strategies. These skills are essential for professional competence and career advancement, enabling participants to navigate and succeed in dynamic work environments. By the end of the program, participants will be well-prepared to develop, implement, and manage IoT systems effectively, while ensuring safety and enhancing their professional competence, thus positioning themselves as valuable assets in the IoT industry.

Table of Contents

Introduction to embedded system

An embedded system is a specialized computing system that is designed to perform dedicated functions or tasks within a larger system. Unlike general-purpose computers, embedded systems are typically purpose-built for specific applications and are tightly integrated into the devices or products they serve. These systems are pervasive in our daily lives, found in a wide range of devices such as consumer electronics, automotive systems, medical devices, industrial machines, and more. Embedded systems play a crucial role in various aspects of modern life, providing dedicated and efficient solutions for a wide range of applications. Their design requires a balance between hardware and software considerations, taking into account performance, power efficiency, and the specific requirements of the intended application.

Characteristics of Embedded system

Dedicated Functionality: Embedded systems are designed to perform specific functions or tasks. Unlike general-purpose computers that can run a variety of applications, embedded systems are tailored for a particular purpose, contributing to their efficiency and reliability in performing specialized tasks.

Real-Time Operation: Many embedded systems operate in real-time or have real-time constraints, meaning they must respond to external events within a predetermined time frame. This is crucial for applications where timing and responsiveness are critical, such as in automotive control systems, industrial automation, or medical devices.

Embedded Hardware: Embedded systems typically have dedicated hardware components, such as microcontrollers or microprocessors, memory, and peripherals. The hardware is often designed to meet the specific requirements of the application, providing a cost-effective and optimized solution.

Integration into a Larger System: Embedded systems are integrated into larger systems or products. They work as a part of a larger entity, contributing to the overall functionality of the device or equipment in which they are embedded.

Resource Constraints: Embedded systems often operate under resource constraints, including limitations on processing power, memory, and storage. Designers must carefully optimize both hardware and software to meet performance requirements within these constraints.

Fixed Functionality: Once deployed, embedded systems typically have fixed functionality and are not easily reprogrammable by end-users. Updates or changes to the software are often performed during the manufacturing process or through specialized procedures.

Low Power Consumption: Many embedded systems are designed to operate with minimal power consumption, especially in applications where energy efficiency is critical. This is essential for battery-powered devices and environments where power availability may be limited.

Specific Environment: Embedded systems often operate in specific environments and conditions. For example, automotive embedded systems must withstand temperature variations and vibrations, while those in medical devices must comply with safety and regulatory standards.

Reliability and Stability: Embedded systems are expected to be reliable and stable over long periods of operation. They need to perform their tasks consistently and without errors, as failures could have significant consequences depending on the application.

Cost Sensitivity: Cost considerations are crucial in embedded system design. Manufacturers aim to produce cost-effective solutions that meet the performance and functionality requirements of the application without unnecessary overhead.

Security Considerations: Security is increasingly important in embedded systems, especially as they become more interconnected. Designers must address vulnerabilities and implement security features to protect against unauthorized access and potential attacks.ere are some key characteristics and components of embedded systems:

Examples of Embedded Systems:

- Microcontrollers in household appliances (washing machines, microwave ovens).
- Automotive control systems (engine control units, anti-lock braking systems).
- Medical devices (patient monitoring systems, infusion pumps).
- Industrial automation (PLCs - Programmable Logic Controllers, robotics).
- Consumer electronics (smartphones, smart TVs).

Working of Embedded System

The working of an embedded system involves the integration of hardware and software components to perform a specific set of functions within a larger system or product. Here is a general overview of the working principles of embedded systems:

System Initialization: The embedded system starts with a power-on or reset event. During initialization, the hardware components are configured, and the initial state of the system is set up. The processor's program counter is set to the starting address of the program memory.

Loading Software: The embedded system's software, often referred to as firmware, is loaded into the program memory. This software is specifically developed to perform the intended functions of the embedded system.

Execution of the Program: The processor begins executing instructions from the program memory. The program contains algorithms and logic to control the embedded system's behaviour and response to various inputs.

Input Processing: Embedded systems interact with the external world through input devices or sensors. Input signals, such as sensor readings or user inputs, are processed by the embedded system to make decisions or adjustments.

Control and Decision Making: The embedded software includes control algorithms that determine how the system responds to different inputs. Decision-making processes mayinvolve calculations, comparisons, and logical operations to control actuators or other output devices.

Output Generation: Embedded systems produce outputs to interact with the external environment. Output devices, such as motors, displays, or communication interfaces, are controlled based on the decisions made by the embedded software.

Real-Time Operation: In many cases, embedded systems operate in real-time, meaning they must respond to inputs within specific time constraints. Real-time tasks, such as sensor data processing or control loop adjustments, are crucial for applications like automotive control systems or industrial automation.

Communication: Embedded systems may communicate with other systems or devices within a network. Communication protocols, such as UART, SPI, I2C, Ethernet, or wireless protocols, are used to exchange data with external devices or a central control unit.

Error Handling and Fault Tolerance: Embedded systems often include mechanisms for error handling and fault tolerance. Error detection, recovery strategies, and mechanisms to handle unexpected events are implemented to enhance system reliability.

Low Power Operation: Many embedded systems are designed with a focus on low power consumption, especially in battery-operated devices. Power management techniques, such as sleep modes and dynamic voltage scaling, may be employed to optimize energy usage.

Security Measures: Depending on the application, embedded systems may incorporate security measures to protect against unauthorized access, data breaches, or malicious attacks.

Lifecycle Management: Embedded systems may have a defined lifecycle with considerations for software updates, maintenance, and end-of-life processes.

Structure of Embedded System

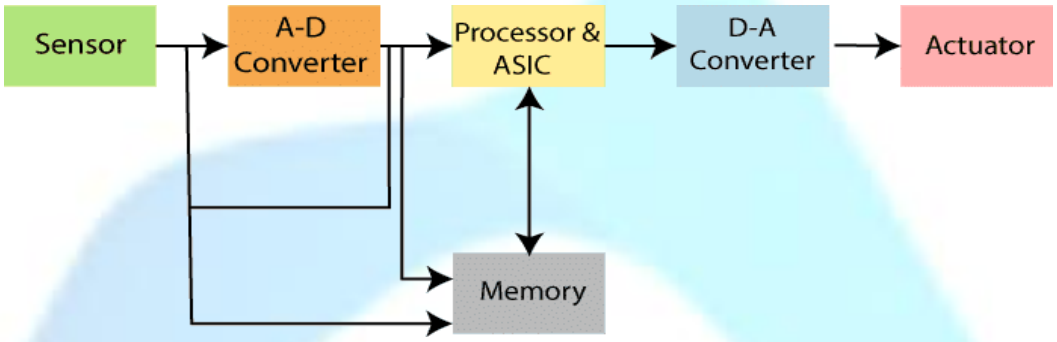


Fig.: Structure of Embedded System

Microcontroller or Microprocessor: The central processing unit (CPU) is at the core of the embedded system and can be either a microcontroller or a microprocessor. Microcontrollers often integrate the CPU, memory, and peripherals on a single chip, while microprocessors may require external components.

Memory:

Program Memory (Flash or ROM): Stores the firmware or software code.

Data Memory (RAM): Holds temporary data and variables during program execution.

Peripherals: Peripherals are hardware components that extend the capabilities of the embedded system.

Common peripherals include: Input/Output (I/O) ports for connecting sensors, actuators, and external devices. Timers and counters for time-sensitive operations. Analog-to-Digital Converters (ADC) for reading analog sensor signals. Communication interfaces such asUART, SPI, I2C, and USB.

Sensors and Actuators: Sensors capture data from the physical environment, providing input to the embedded system. Actuators are devices that carry out physical actions based on the system's output. Examples include temperature sensors, accelerometers, motors, and displays.

Power Supply: Embedded systems require a power supply, which may be provided through batteries, external power sources, or power management circuits. Power considerations are crucial, especially in applications where low power consumption is essential.

System Clock: A clock source provides timing for the operation of the embedded system. Clock frequency influences the system's processing speed and may impact power consumption.
Bus Architecture: Buses facilitate communication between different components of the embedded system. Address buses and data buses enable the transfer of information between the CPU, memory, and peripherals.

Firmware/Software: The embedded software, often referred to as firmware, is a crucial component. It includes the application code, control algorithms, and other software modules necessary for the system's operation.

Operating System (Optional): Some embedded systems use real-time operating systems (RTOS) to manage tasks, scheduling, and resource allocation. Many smaller embedded systems operate without a full-fledged operating system.

Communication Interfaces: Embedded systems may include communication interfaces to exchange data with other devices or systems. Examples include wired (Ethernet, CAN) or wireless (Wi-Fi, Bluetooth) communication.

Security Features: Depending on the application, security measures may be implemented to protect against unauthorized access and ensure data integrity. Security features can include encryption, authentication, and secure boot mechanisms.

Debugging and Development Tools: Embedded systems often rely on debugging and development tools to facilitate software development, testing, and troubleshooting. In-circuit emulators, debuggers, and compilers are common tools used by developers.

Enclosure and Packaging: The physical enclosure and packaging are essential considerations, especially for embedded systems deployed in harsh environments or consumer products. Environmental factors, like temperature and humidity, may influence the design of the enclosure.

Types of Embedded System

Embedded systems come in various types, each tailored to specific applications and requirements. Here are some common types of embedded systems based on their functionalities.

Real-Time Embedded Systems: Real-time embedded systems are designed to respond to events or inputs within a specific time frame. They are crucial in applications where timing is critical, such as automotive control systems, industrial automation, and medical devices.

Stand-Alone Embedded Systems: Stand-alone embedded systems operate independently and do not rely on external systems for their functionality. Examples include household appliances, consumer electronics, and embedded controllers in industrial machines.

Networked Embedded Systems: These systems are connected to a network and communicate with other devices or systems. Applications include IoT (Internet of Things) devices, smart home systems, and industrial automation networks.

Mobile Embedded Systems: Mobile embedded systems are designed for portable devices and often have power-efficient features. Examples include smartphones, tablets, wearable devices, and portable medical instruments.

Digital Signal Processors (DSPs): DSP embedded systems are optimized for processing and manipulating digital signals. Common applications include audio processing, imageprocessing, and communication systems.

Automotive Embedded Systems: Embedded systems are prevalent in the automotive industry for tasks such as engine control, anti-lock braking systems (ABS), airbag systems, and in-vehicle infotainment.

Avionics Systems: Avionics embedded systems are used in aircraft for navigation, communication, flight control, and monitoring of various systems.

Medical Embedded Systems: Medical devices and instruments often incorporate embedded systems for patient monitoring, diagnostics, and control of medical equipment.

Industrial Embedded Systems: Industrial automation relies heavily on embedded systems for tasks such as process control, robotics, and monitoring of manufacturing equipment.

Consumer Electronics: Embedded systems are pervasive in consumer electronics, including smart TVs, digital cameras, home theater systems, and gaming consoles.

Embedded Systems in IoT (Internet of Things): IoT embedded systems connect physical devices to the internet, enabling data exchange and remote control. Examples include smart thermostats, connected sensors, and home automation systems.

Embedded Control Systems: These systems are designed for controlling various processes and systems, such as traffic lights, elevators, and heating, ventilation, and air conditioning (HVAC) systems.

Embedded Systems in Robotics: Robotics heavily relies on embedded systems for control and coordination of robotic movements and functions.

Embedded Systems in Telecommunications: Telecommunication equipment, such as routers, switches, and base stations, often incorporates embedded systems for data processing and network management.

Embedded Security Systems: These systems are designed to provide security features, including surveillance cameras, access control systems, and biometric identification systems.

Embedded Systems in Smart Grids: Smart grid systems use embedded systems for efficient monitoring and control of electrical grids, optimizing energy distribution**.**

Microcontrollers

A microcontroller is a compact integrated circuit that contains a processor core, memory, and programmable input/output peripherals. It is designed to perform specific tasks in embedded systems. Unlike general-purpose computers, microcontrollers are optimized for real-time processing and control applications. Microcontrollers are widely used in embedded systems for a variety of applications, including robotics, industrial automation, smart appliances, automotive control systems, and many others. They are chosen for their compact size, low power consumption, and specialized functionality tailored to the specific requirements of the application. Popular microcontroller families include the AVR series from Atmel (now owned by Microchip), PIC from Microchip, ARM Cortex-M series, and many others.

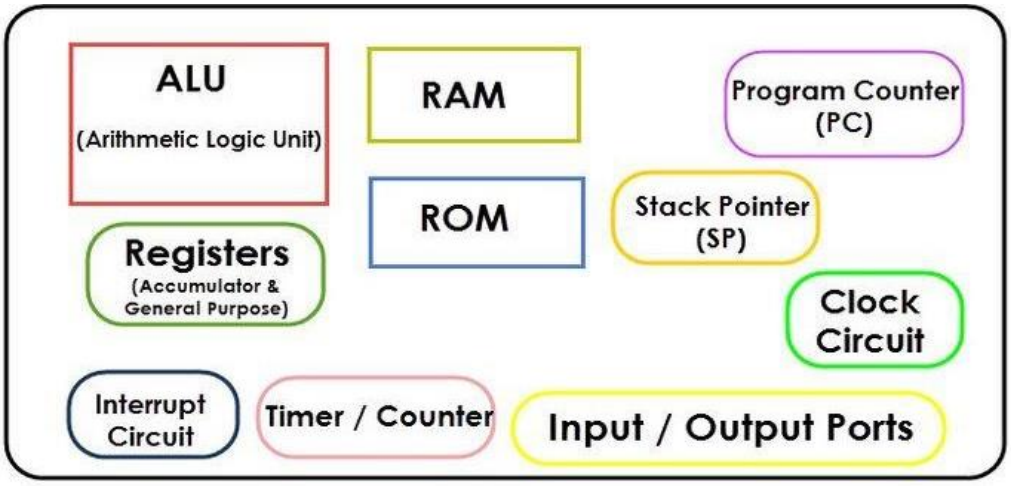


Fig.: Block Diagram of Microcontroller

Central Processing Unit (CPU): The processor core is the brain of the microcontroller, responsible for executing instructions.

Memory: Microcontrollers typically have both program memory (for storing the firmware or program code) and data memory (for storing variables and temporary data).

Input/Output (I/O) Peripherals: Microcontrollers have dedicated pins and circuits for interfacing with the external world. These can include digital and analog input/output pins, serial communication ports (UART, SPI, I2C), timers, counters, and more.

Clock Source: Microcontrollers require a clock signal to synchronize their internal operations. The clock speed influences the processing speed of the microcontroller.

Interrupt System: Microcontrollers often support interrupts, allowing them to respond quickly to external events or signals.

Analog-to-Digital Converter (ADC): Many microcontrollers have ADCs to convert analog signals (such as sensor readings) into digital values.

Communication Interfaces: Microcontrollers may include various communication interfaces like UART (Universal Asynchronous Receiver/Transmitter), SPI (Serial Peripheral Interface), I2C (Inter-Integrated Circuit), and others for connecting to other devices.

Peripheral Devices: Some microcontrollers have built-in peripherals like timers, PWM controllers, and more to facilitate specific tasks.

Selection of microcontrollers

Performance Requirements: Evaluate the processing power needed for your application. Consider factors such as clock speed, instruction set architecture, and the complexity of the tasks the microcontroller must perform.

Memory Requirements: Assess the program memory (Flash) and data memory (RAM) requirements. Ensure the selected microcontroller has sufficient memory for storing your code and handling data.

Peripherals and I/O Requirements: Identify the required peripherals and I/O features for your application. Consider the number and types of GPIO pins, communication interfaces (UART, SPI, I2C), timers, ADCs, PWM controllers, and other peripherals.

Power Consumption: Evaluate the power consumption characteristics of the microcontroller, especially if your application requires low power or battery-operated functionality.

Operating Voltage: Ensure that the microcontroller's operating voltage matches the voltage levels of the components in your system. Some microcontrollers support a wide range of voltages, offering flexibility in design.

Package Size and Form Factor: Consider the physical size and package type of the microcontroller. Ensure it fits within the constraints of your system design and can be easily integrated.

Development Tools and Ecosystem: Check the availability and quality of development tools, such as compilers, debuggers, and programming environments. A strong ecosystem and good documentation can simplify the development process.

Cost Considerations: Evaluate the overall cost of the microcontroller, including not only the component cost but also the development tools, support, and other associated costs.

Availability and Longevity: Consider the availability and long-term support for the chosen microcontroller. Ensure that the manufacturer provides a reliable supply and plans for long-term availability.

Community and Support: Check for a vibrant user community and online support forums. Having access to a community of developers can be valuable for troubleshooting, sharing knowledge, and finding solutions to common problems.

Security Features: If your application requires security features, such as encryption or secure boot, choose a microcontroller that includes the necessary hardware support for these features.

Which microcontroller is right for your iot?

Wireless Connectivity: Many IoT applications require wireless communication. Choose a microcontroller that supports the relevant wireless protocols such as Wi-Fi, Bluetooth, Zigbee, LoRa, or cellular connectivity based on the requirements of your IoT project.

Power Efficiency: IoT devices are often battery-powered or have strict power constraints. Select a microcontroller with low power consumption to extend battery life or minimize power usage in energy-efficient IoT applications.

Processing Power: Consider the processing power needed for your IoT application. While some IoT devices may require only basic processing capabilities, others, such as edge computing devices, may need more powerful microcontrollers to handle data processing locally.

Security Features: IoT devices often handle sensitive data, so security is crucial. Look for microcontrollers with built-in security features, such as hardware encryption, secure boot, and secure key storage.

Memory Size: Ensure that the microcontroller has sufficient memory for your IoT application, considering both program memory (Flash) and data memory (RAM) requirements.

Sensor Integration: IoT devices typically involve sensors to collect data. Choose a microcontroller with the necessary interfaces and ADC channels to connect and interface with the sensors in your application.

IoT Protocols and Standards: Consider whether the microcontroller supports common IoT communication protocols and standards such as MQTT, CoAP, or HTTP for seamless integration with IoT platforms and services.

Development Ecosystem: Look for a microcontroller with a well-supported development ecosystem, including software development kits (SDKs), libraries, and community support. This can make the development process more efficient.

Cost: Evaluate the overall cost of the microcontroller, considering not only the component cost but also development tools, support, and associated costs.

Form Factor: Consider the physical size and form factor of the microcontroller, especially if your IoT device has space constraints.

Microprocessor

A microprocessor is a computer processor that contains the arithmetic, logic, and control circuitry required to perform the functions of a computer's central processing unit. It's also known as a CPU or central processing unit.

A microprocessor is an electronic component that is used by a computer to do its work. It's a central processing unit on a single integrated circuit chip containing millions of very small components including transistors, resistors, and diodes that work together.

The first microprocessor was the Intel 4004, introduced in 1971. The 4004 was not very powerful — all it could do was add and subtract, and it could only do that 4 bits at a time.

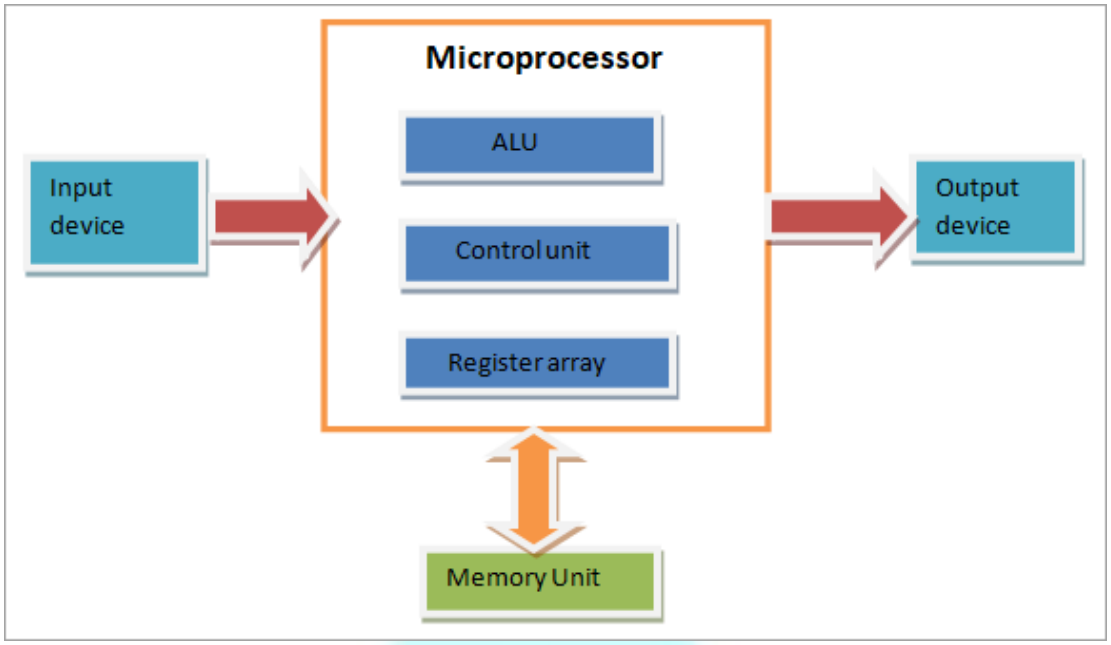


Fig.: Block Diagram Of Microprocessor

Arithmetic Logic Unit (ALU)

The ALU is responsible for performing arithmetic and logical operations on data. It can perform operations such as addition, subtraction, logical AND, logical OR, and more. The ALUoperates on 8-bit data and provides flags to indicate conditions such as zero, carry, sign, and parity.

Control Unit (CU)

The Control Unit coordinates and controls the activities of the other functional units within the microprocessor. It generates timing and control signals to synchronize the execution of instructions and manage data transfer between different units.

Instruction Decoder

The Instruction Decoder decodes the instructions fetched from memory. It determines the type of instruction being executed and generates control signals accordingly. The decoded instructions guide the microprocessor in executing the appropriate operations.

Registers

The 8085 microprocessor has several registers that serve different purposes:

Accumulator (A): The Accumulator is an 8-bit register used for storing intermediate results during arithmetic and logical operations.

General Purpose Registers (B, C, D, E, H, L): These are six 8-bit registers that can be used for various purposes, including storing data and performing operations.

Special Purpose Registers (SP, PC): The Stack Pointer (SP) is used to manage the stack in memory and the program counter (PC) keeps track of the memory address of the following instruction for fetching.

Address and Data Bus

The microprocessor uses a bidirectional address bus to specify the memory location or I/O device it wants to access. Similarly, it employs an 8-bit bidirectional data bus for transferring data between the microprocessor and memory or I/O devices.

Timing and Control Unit

The Timing and Control Unit generates the necessary timing signals to synchronize the activities of the microprocessor. It produces signals such as RD (Read), WR (Write), and various control signals required for instruction execution.

Interrupt Control Unit

The Interrupt Control Unit manages interrupts in the 8085 microprocessor. It handles external interrupt signals and facilitates interrupt-driven operations by interrupting the normal execution flow of the program and branching to specific interrupt service routines.

Memory Interface

The Memory Interface connects the microprocessor to the memory system. It manages the address and data transfers between the microprocessor and the memory chips, including Read and Write operations.

Difference Between Microprocessor and Microcontroller

Specification	Microcontroller	Microprocessor
Function	Geared towards specific control- oriented tasks. It integrates the CPU, memory, and peripherals on a single chip and is commonly used in embedded systems for dedicatedfunctions such as controlling a deviceor a system.	Primarily designed for general-purpose computing tasks. It serves as the central processing unit (CPU) in a computer system and is oftenused in applications where computational power is a priority.
Integration of Components	Integrates the CPU, memory, and peripherals (timers, counters, GPIO,etc.) on a single chip, making it a self-contained unit for specificapplications.	Typically needs external components like memory (RAM, ROM), input/output devices, and additional support chips to function as a complete system.
Applications	Used in embedded systems for tasks such as control, monitoring, and interfacing with the environment. Common applications include consumer electronics, automotive systems, industrial automation, andIoT devices.	Suited for applications that require significant computational power, multitasking, and handling complex software. Examples include personal computers, servers, and high-end systems.
Cost and Power Consumption	Designed for cost-effectiveness and efficiency, with lower power consumption. It is optimized for the specific tasks it is intended to perform.	Generally more expensive and consumes more power due toits higher processingcapabilities.
Instruction Set	Features a more specialized instruction set tailored to the specificapplications it is designed for, which can contribute to better performancein those tasks.	Often has a more extensiveand general-purpose instruction set to handle a variety of tasks.
Complexity	Simplified architecture with theessential components integrated,making it easier to use for specific applications.	Generally more complex in terms of architecture and capabilities.

Application of Microcontrollers

1. Consumer Electronics

- **Home Appliances:** Microcontrollers are used in washing machines, microwave ovens, air conditioners, and refrigerators to control functions, display information, and ensure energy efficiency.
- **Personal Gadgets:** Devices like remote controls, digital cameras, and smartwatches utilize microcontrollers for handling user inputs and managing internal processes.

2. Automotive Industry

- **Engine Control Units (ECUs):** Microcontrollers manage engine performance, fuel injection, and emissions control.
- **Infotainment Systems:** They provide control for audio, video, and navigation systems within vehicles.
- **Safety Features:** Airbag deployment, anti-lock braking systems (ABS), and electronic stability control (ESC) systems rely on microcontrollers for real-time processing and response.

3. Industrial Automation

- **Process Control:** Microcontrollers are employed in programmable logic controllers (PLCs) to automate manufacturing processes, ensuring precision and consistency.
- **Robotics:** They control robotic arms and automated guided vehicles (AGVs), enabling tasks like assembly, packaging, and material handling.

4. Medical Devices

- **Diagnostic Equipment:** Devices such as blood glucose meters, blood pressure monitors, and portable ECG machines use microcontrollers to process data and provide readings.
- **Therapeutic Devices:** Microcontrollers control devices like insulin pumps, pacemakers, and hearing aids to ensure proper operation and patient safety.

5. IoT (Internet of Things)

- **Smart Home Devices:** Microcontrollers are integral in smart thermostats, lighting systems, and security cameras, enabling connectivity and automation.
- **Wearable Technology:** Fitness trackers and health monitoring devices use microcontrollers to gather and process sensor data.

Application of Microprocessors

1. Personal Computers (PCs)

- **Desktops and Laptops:** Microprocessors are the brain of PCs, executing instructions, running applications, and managing system resources.
- **Servers:** High-performance microprocessors in servers handle large-scale processing, data storage, and network management tasks.

2. Mobile Devices

- **Smartphones and Tablets:** Microprocessors enable advanced functionalities such as multitasking, multimedia processing, and connectivity features.
- **Handheld Gaming Devices:** They provide the computational power needed for rendering graphics and running sophisticated gaming software.

3. Embedded Systems

- **Digital Signage:** Microprocessors drive the content management and display functionalities in digital billboards and interactive kiosks.
- **Automotive Systems:** Advanced driver-assistance systems (ADAS) and infotainment systems in vehicles rely on microprocessors for complex processing tasks.

4. Industrial Applications

- **Automation and Control Systems:** Microprocessors are used in industrial computers and controllers for managing large-scale industrial processes and machinery.
- **Data Acquisition Systems:** They process and analyze data from various sensors and instrumentation in real-time.

5. Consumer Electronics

- **Smart TVs:** Microprocessors enable smart features like internet connectivity, app integration, and voice control.
- **Gaming Consoles:** They power the high-speed processing required for running modern video games and providing immersive experiences.

6. Telecommunications

- **Network Routers and Switches:** Microprocessors manage data routing, network traffic, and communication protocols in networking equipment.
- **Base Stations:** They handle signal processing, data transmission, and connectivity management in cellular networks.

Sensors

A sensor is a device that detects and responds to some type of input from the physical environment. The input can be light, heat, motion, moisture, pressure or any number of other environmental phenomena. The output is generally a signal that is converted to a human-readable display at the sensor location or transmitted electronically over a network for reading or further processing.

Sensors play a pivotal role in the internet of things (IoT). They make it possible to create an ecosystem for collecting and processing data about a specific environment so it can be monitored, managed and controlled more easily and efficiently. IoT sensors are used in homes, out in the field, in automobiles, on airplanes, in industrial settings and in other environments. Sensors bridge the gap between the physical world and logical world, acting as the eyes and ears for a computing infrastructure that analyzes and acts upon the data collected from the sensors.

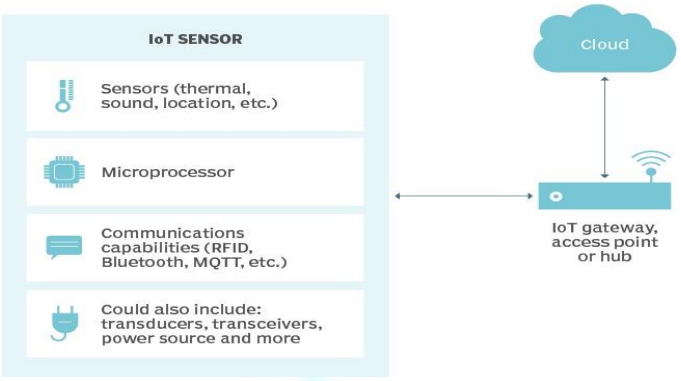


Fig.: IOT in action

Role of sensors in IoT

Data Acquisition: Sensors gather data from the surrounding environment. This data can include information about temperature, humidity, pressure, motion, light, sound, and more.

Environmental Monitoring: Sensors in IoT devices enable the continuous monitoring of environmental conditions. This is valuable in various applications such as agriculture (soil moisture, temperature), smart cities (air quality, noise levels), and industrial settings.

Remote Sensing: IoT sensors allow for remote sensing of physical parameters. This is particularly useful in scenarios where manual monitoring is difficult or dangerous.

Automation and Control: Sensors provide real-time information that can be used to automate processes and control devices. For example, sensors in smart home devices can adjust heating or cooling systems based on temperature readings.

Predictive Maintenance: IoT sensors are used for monitoring the health and performance of machinery and equipment. By analysing sensor data, predictive maintenance models can predict when equipment is likely to fail, allowing for timely maintenance and reducing downtime.

Energy Efficiency: Sensors help optimize energy usage by providing insights into energy consumption patterns. This is crucial for applications in smart buildings, smart grids, and energy management systems.

Health Monitoring: Wearable devices equipped with various sensors monitor vital signs, physical activity, and other health-related parameters. This data can be used for personal health tracking, as well as in healthcare applications.

Asset Tracking: IoT sensors can be used to track the location and status of assets in real-time. This is valuable in logistics, supply chain management, and inventory control.

Security and Surveillance: Sensors play a key role in security systems by detecting motion, monitoring access points, and capturing images or video. These applications are crucial for both residential and commercial security.

Smart Agriculture: Sensors in agriculture measure soil moisture, temperature, and other environmental factors, allowing farmers to make informed decisions about irrigation, fertilization, and crop management.

Data Analytics: The data collected by sensors is often processed and analysed to extract meaningful insights. This data analytics process is essential for making informed decisions and optimizing various processes.

Working of sensor in IoT

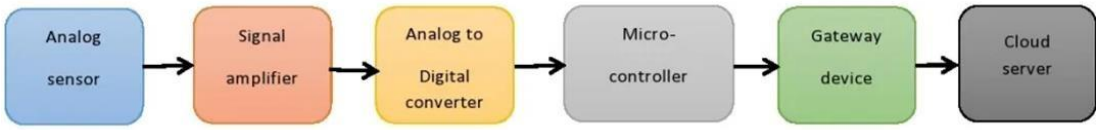


Fig.: IOT Sensor Workflow

IoT cloud servers and devices depend on sensors to collect real-time data.

Sensors detect changes in their environment and then convert it to digital data. Because, the physical parameters are present in the analog signal like the temperature in Fahrenheit, speed in miles per hour, distance in feet, etc.

Sensors need to connect to the cloud through some connectivity. Nowadays, we use wireless technologies such as Bluetooth, NFC, RF, Wi-Fi.

Classification of Sensor

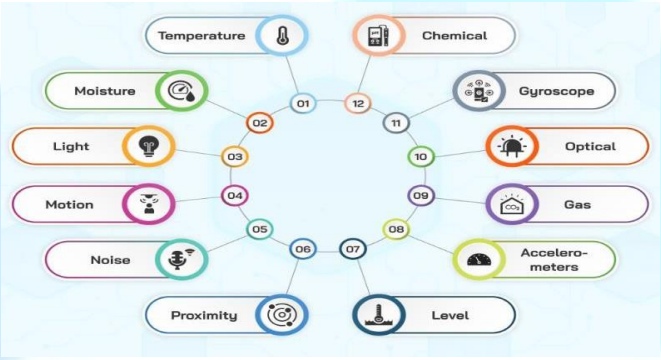


Fig.: Types of Sensors

- **Based on Measured Property**

Temperature Sensors: Measure temperature changes (e.g., thermocouples, thermistors).

Pressure Sensors: Measure pressure variations (e.g., barometers, piezoelectric sensors).

Proximity Sensors: Detect the presence or absence of an object without physical contact (e.g., infrared sensors, ultrasonic sensors).

Motion Sensors: Detect movement or acceleration (e.g., accelerometers, gyroscopes).

Light Sensors: Measure the intensity of light (e.g., photodiodes, phototransistors).

Humidity Sensors: Measure the moisture content in the air (e.g., hygrometers, capacitive humidity sensors).

Gas Sensors: Detect the concentration of gases in the environment (e.g., carbon monoxide sensors, methane sensors).

Biometric Sensors: Capture and analyze biological characteristics for identification

(e.g.,fingerprint scanners, iris scanners).

- **Based on Technology**

Active Sensors: Require an external power source to operate (e.g., active infrared sensors).

Passive Sensors: Do not require an external power source and operate based on external stimuli (e.g., passive infrared sensors).

- **Based on Output Signal**

Analog Sensors: Provide a continuous output signal that varies proportionally with the measured property.

Digital Sensors: Provide discrete output signals, often in the form of binary data.

- **Based on Application**

Industrial Sensors: Used in manufacturing and industrial automation (e.g., pressure sensors, proximity sensors).

Automotive Sensors: Used in vehicles for various purposes, such as engine control, safety, and navigation (e.g., temperature sensors, accelerometers).

Medical Sensors: Used in healthcare for monitoring and diagnostics (e.g., heart rate monitors, blood pressure sensors).

Environmental Sensors: Monitor parameters related to the environment, such as air quality and pollution (e.g., air quality sensors).

Consumer Electronics Sensors: Integrated into devices like smartphones, smartwatches, and cameras for various functionalities (e.g., accelerometers, gyroscopes).

- **Based on Working Principle:**

Resistive Sensors: Work based on changes in electrical resistance (e.g., thermistors).

Capacitive Sensors: Operate based on changes in capacitance (e.g., touch sensors, humidity sensors).

Optical Sensors: Use light to detect changes in the environment (e.g., photodiodes, phototransistors).

Piezoelectric Sensors: Generate an electrical charge in response to mechanical stress (e.g., pressure sensors).

- **Based on Range and Resolution:**

High-Resolution Sensors: Provide precise and detailed measurements.

Low-Resolution Sensors: Offer less detailed measurements but may be suitable for certain applications.

Actuator

A device that turns electrical energy into mechanical energy is known as an actuator. An actuator, in other terms, is a component that can move or control a mechanism or system. Actuators are commonly employed in industrial automation, robotics, and other applications requiring precise mechanical system control. Actuators come in a variety of configurations, including electric, hydraulic, and pneumatic actuators.

Types of Actuators

Fig.: Types Of Actuators

Actuators come in various forms, and each type serves a specific purpose depending on the application. Two main categories define actuators: the type of movement and the power source.

Actuator movement type

Linear actuators: Linear actuators move objects along a straight line and use a belt and pulley, rack and pinion, or ball screw to convert electric motor rotation into linear motion. Linear actuators stop at a fixed linear distance and are known for their high repeatability and positioning accuracy, easy installation and operation, low maintenance, and ability to withstand harsh environments. These actuators are commonly used in food processing, automotive, material handling, and more for tasks like pushing, pulling, lifting, and positioning.

Rotary actuators: Rotary actuators convert energy into rotary motion through a shaft to control equipment speed, position, and rotation. These actuators have a continuous rotational motor and are versatile in usage. An electric motor is a rotary actuator powered by an electric signal. They have high torque, constant torque during full angle rotation, compatibility with different diameters, hollow shafts with zero backlash, twice the output, low maintenance, and can achieve any degree of rotation. Rotary actuators are used in medical equipment, radar and monitoring systems, robotics, flight simulators, the semiconductor industry, special machine manufacturing, and defense.

Actuator power sources

Pneumatic actuators: Pneumatic actuators (Figure 2) use compressed air to generate motion. They can be used for various applications, such as moving machine parts or controlling valve positions. They are often preferred for applications that require high force, fast response times, or explosion-proof environments.

Hydraulic actuators: Hydraulic actuators use fluid pressure to generate motion. They are commonly used for heavy-duty applications such as construction equipment, manufacturing machinery, and industrial robots. Hydraulic actuators offer high levels of force, durability, and reliability.

Electric actuators: Electric actuators (Figure 3) use electrical energy to generate motion. They can be driven by AC or DC motors and are often used in applications that require precise control, low noise, and low maintenance. Electric actuators are commonly used in automation systems, medical devices, and laboratory equipment.

Magnetic and thermal actuators: Magnetic and thermal actuators are two types of actuators that use magnetic and temperature changes to generate motion, respectively. Magnetic actuators use magnetic fields to generate force. Thermal actuators use the expansion or contraction of materials in response to temperature changes. Both actuators are commonly used in micro-electromechanical systems (MEMS) and other miniaturized applications.

Mechanical actuators: Mechanical actuators use physical mechanisms such as levers, gears, or cams to generate motion. Mechanical actuators are commonly used in applications where low cost, simple operation, and durability are important. Examples include hand-crank machines, manual valve systems, and mechanical locks.

Applications of Actuator

Smart Home Automation: Actuators control smart devices such as smart locks, smart thermostats, and motorized blinds based on user preferences or automation rules.

Industrial Automation: Actuators play a crucial role in manufacturing processes, controlling machinery, valves, and robotic systems based on real-time data.

Healthcare: Actuators in medical devices may adjust the dosage of medication, control the flow of fluids, or move mechanical components in response to patient monitoring data.

Smart Cities: Actuators are used in various applications such as controlling traffic lights, adjusting street lighting based on environmental conditions, and managing irrigation systems.

Agriculture: Actuators control irrigation systems, open/close valves, and adjust equipment in response to environmental sensors and monitoring data.

Environmental Control: Actuators are involved in HVAC (Heating, Ventilation, and Air Conditioning) systems, adjusting airflow, temperature, and humidity based on sensor data.

Transportation: Actuators in vehicles control various functions, such as adjusting seat positions, managing engine components, and steering mechanisms in autonomous vehicles.

Introduction to IoT

IoT stands for Internet of Things. It refers to the interconnectedness of physical devices, such as appliances and vehicles, that are embedded with software, sensors, and connectivity which enables these objects to connect and exchange data. This technology allows for the collection and sharing of data from a vast network of devices, creating opportunities for more efficient and automated systems.

Internet of Things (IoT) is the networking of physical objects that contain electronics embedded within their architecture in order to communicate and sense interactions amongst each other or with respect to the external environment. In the upcoming years, IoT-based technology will offer advanced levels of services and practically change the way people lead their daily lives. Advancements in medicine, power, gene therapies, agriculture, smart cities, and smart homes are just a few of the categorical examples where IoT is strongly established.

IOT is a system of interrelated things, computing devices, mechanical and digital machines, objects, animals, or people that are provided with unique identifiers. And the ability to transfer the data over a network requiring human-to-human or human-to-computer interaction.

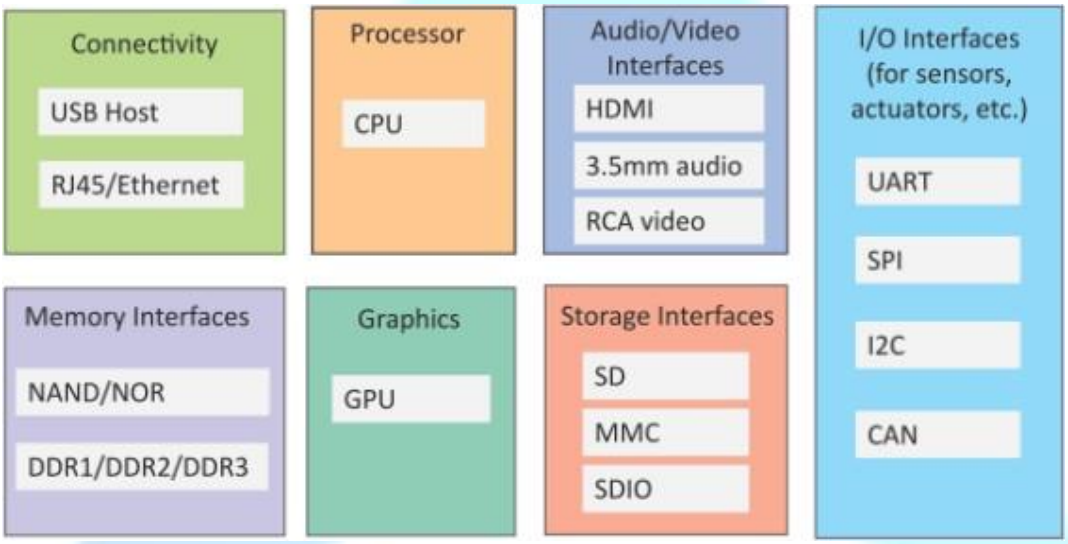


Fig.:Block Diagram of IOT

This diagram outlines the various interfaces and components that are typically found in an embedded system or IoT device. Each of these components and interfaces plays a crucial role in the functionality and performance of embedded systems and IoT devices, enabling them to connect to various peripherals, process data, and communicate with other devices and networks.

- **Connectivity**:
  - **USB Host**: An interface that allows the system to connect to USB devices such as keyboards, mice, storage devices, etc.
  - **RJ45/Ethernet**: A standard network interface for wired internet connections, facilitating communication over a local network or the internet.
- **Processor**:
  - **CPU (Central Processing Unit)**: The primary component of a computer that performs most of the processing inside an embedded system.

- **Audio/Video Interfaces**:
  - **HDMI (High-Definition Multimedia Interface)**: An interface for transmitting high-definition audio and video signals.
  - **3.5mm audio**: A standard audio jack for connecting headphones, microphones, and other audio devices.
  - **RCA video**: A type of analog video connector commonly used for video signals.
- **Memory Interfaces**:
  - **NAND/NOR**: Types of flash memory technologies used for storage in embedded systems. NAND flash is generally used for data storage, while NOR flash is used for code storage due to its faster read speeds.
  - **DDR1/DDR2/DDR3**: Types of dynamic random-access memory (DRAM) technologies, used for system memory in embedded systems. DDR (Double Data Rate) memory comes in different versions with varying speeds and efficiencies.
- **Graphics**:
  - **GPU (Graphics Processing Unit)**: A specialized processor designed to accelerate graphics rendering, commonly used in systems that require image and video processing.
- **Storage Interfaces**:
  - **SD (Secure Digital)**: A type of non-volatile memory card used for storing data in devices like cameras, smartphones, and embedded systems.
  - **MMC (MultiMediaCard)**: A memory card standard similar to SD cards, used for storage in various devices.
  - **SDIO (Secure Digital Input Output)**: An interface that allows additional functionality such as Wi-Fi or Bluetooth modules to be connected to an SD card slot.
- **I/O Interfaces (for sensors, actuators, etc.)**:
  - **UART (Universal Asynchronous Receiver-Transmitter)**: A hardware communication protocol used for serial communication between devices.
  - **SPI (Serial Peripheral Interface)**: A synchronous serial communication protocol used for short-distance communication, primarily in embedded systems.
  - **I2C (Inter-Integrated Circuit)**: A multi-master, multi-slave, packet-switched, single-ended, serial communication bus used for attaching lower-speed peripherals to processors and microcontrollers.
  - **CAN (Controller Area Network)**: A robust vehicle bus standard designed to allow microcontrollers and devices to communicate with each other without a host computer, commonly used in automotive applications.

Hardware Interfacing Protocol

- I2C Protocol
- UART Protocol
- SPI Protocol
- CAN Protocol
- MODBUD Protocol

I2C Protocol

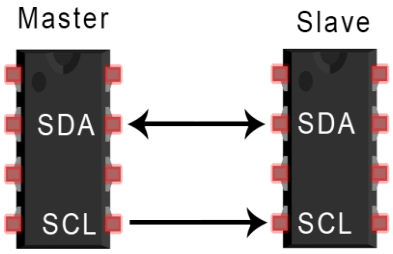


Fig.: I2C Protocol

The Inter-Integrated Circuit (I2C) protocol is a widely used serial communication protocol that allows multiple devices to communicate with each other over a short distance. It was developed by Philips (now NXP Semiconductors) and is commonly used in embedded systems, sensors, and various other electronic devices. I2C is widely used in applications where a moderate data rate, low pin count, and relatively short-distance communication are required. It is commonly found in sensors, memory devices, and other peripherals in embedded systems.

Features of I2C protocol

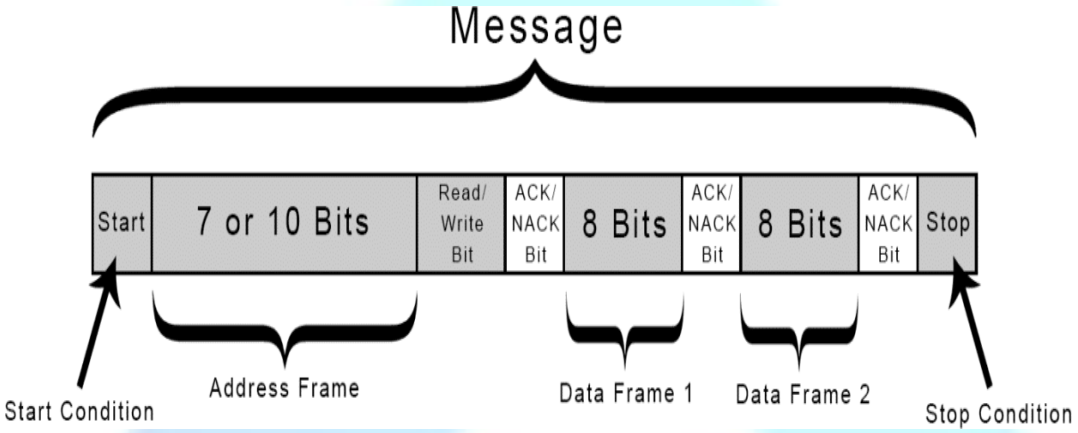


Fig.: I2C Frame Format

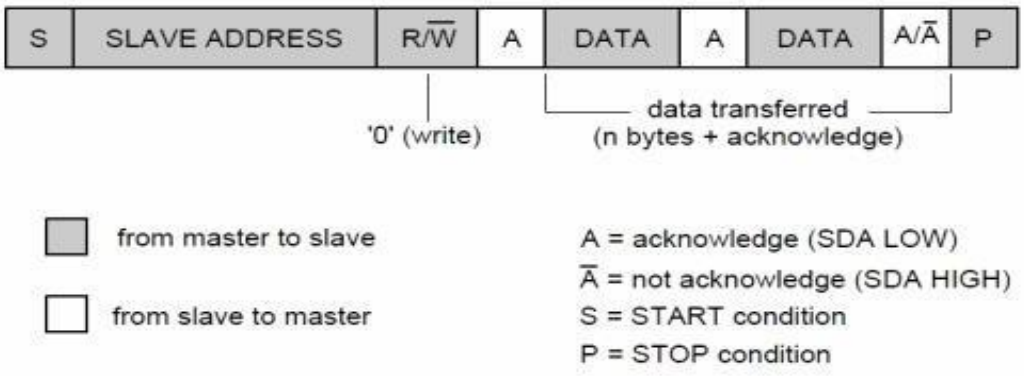


Fig.: Master-slave Frame Format

Two-Wire Communication: I2C uses only two wires for communication.

SDA (Serial Data Line): Carries the actual data being transmitted between devices.

SCL (Serial Clock Line): Carries the clock signal, which synchronizes the data transfer between devices.

Master-Slave Architecture: The I2C protocol operates in a master-slave architecture. One device (the master) initiates and controls the communication, while one or more devices (the slaves) respond to the master's commands.

Addressing: Each I2C device on the bus has a unique 7-bit or 10-bit address. The 7-bit addressing allows for up to 128 unique addresses, while the 10-bit addressing extends the address space for larger systems.

Start and Stop Conditions: Communication on the I2C bus begins with a start condition and ends with a stop condition. The start condition indicates the beginning of a communication session, and the stop condition indicates the end. These conditions help in framing the data transmission.

Data Frame Format: Data on the I2C bus is transmitted in 8-bit frames. Each byte is usually followed by an acknowledgment bit. The master generates the clock signal, and data on the SDA line is sampled on the rising or falling edge of the clock signal, depending on the device's specifications.

Clock Speeds: The I2C protocol supports different clock speeds, often expressed as the frequency of the SCL line. Common speeds include standard mode (up to 100 kHz), fast mode (up to 400 kHz), and high-speed mode (up to 3.4 MHz).

Acknowledge (ACK) Bit: After each byte of data, the receiver (whether master or slave) sends an acknowledge bit to confirm that it received the data successfully. If the acknowledge bit is not received, it indicates an error, and communication may be aborted.

Repeatability: The I2C protocol is designed to be repeatable, meaning that devices can be addressed multiple times within a single communication session, allowing for efficient communication with multiple devices on the bus.

Working of I2C Protocol

Initialization: The master device generates a start condition by pulling the SDA (Serial Data Line) from high to low while SCL (Serial Clock Line) remains high. This start condition indicates the beginning of a communication session.

Addressing: The master sends the address of the slave device it wants to communicate with. The address is typically 7 or 10 bits long. The 7-bit addressing allows for up to 128 unique addresses, while the 10-bit addressing extends the address space for larger systems.

Read/Write Bit: After sending the address, the master indicates whether it wants to read from or write to the slave by setting the Read/Write bit. If the Read/Write bit is 0, it indicates a write operation (master is sending data to the slave). If it's 1, it indicates a read operation (master is receiving data from the slave).

Data Transfer: The actual data transfer occurs in 8-bit frames. In a write operation, the master sends data to the slave, and in a read operation, the slave sends data to the master. The data is clocked on the rising or falling edge of the SCL line, depending on the device specifications.

Acknowledge (ACK) Bit: After each byte of data, the receiver (whether master or slave) sends an Acknowledge bit to confirm successful reception. If the Acknowledge bit is not received, it indicates an error, and the communication may be aborted.

Repeated Start or Stop Condition: After completing the data transfer for a particular transaction, the master can choose to either generate a repeated start condition or a stop condition. A repeated start allows the master to continue communication without releasing control of the bus, enabling multiple transactions in a single session. A stop condition indicates the end of the communication session.

Termination: The master generates a stop condition by pulling the SDA from low to high while SCL remains high. The stop condition signals the end of the communication session.

Advantages of I2C Protocol

Simplicity and Minimal Wiring: I2C uses only two wires (SDA and SCL) for communication, which simplifies the wiring and reduces the number of pins required on the devices. This is especially beneficial in space-constrained applications.

Multi-Master Capability: I2C supports a multi-master architecture, allowing multiple master devices to exist on the same bus. This feature is useful in systems where more than one device needs to initiate communication.

Addressing Flexibility: I2C supports both 7-bit and 10-bit addressing, providing flexibility in designing systems with a large number of devices. The 7-bit addressing allows for up to 128 unique addresses, while the 10-bit addressing extends the address space for larger systems.

Clock Synchronization: The master device generates the clock signal, ensuring synchronization between devices on the bus. This eliminates the need for precise clock synchronization between devices, simplifying the design.

Acknowledge Mechanism: I2C includes an Acknowledge (ACK) mechanism after each byte transfer. This allows the sender to confirm that the data was successfully received by the receiver. If a device fails to acknowledge, it indicates a potential issue.

Efficient for Short-Distance Communication: I2C is designed for short-distance communication within a single PCB or between closely located devices. It is well-suited for communication between components on a circuit board.

Low Power Consumption: I2C devices can be designed with low-power consumption, making it suitable for battery-operated devices or applications with strict power requirements.

Widely Adopted Standard: I2C is a well-established and widely adopted standard in the electronics industry. This widespread adoption ensures compatibility and interoperability between devices from different manufacturers.

Built-in Arbitration: I2C has built-in arbitration mechanisms that handle bus contention in multi-master configurations. If two masters attempt to communicate simultaneously, the protocol resolves the conflict.

Support for Repeated Start: I2C allows the master to generate a repeated start condition, enabling sequential communication with a device without releasing control of the bus. This is useful for efficient communication with multiple devices in a single session.

Disadvantages of I2C Protocol

Limited Distance: I2C is primarily designed for short-distance communication within a single PCB (Printed Circuit Board) or between closely located devices. It may not be suitable for applications requiring communication over longer distances.

Lower Data Transfer Rates: Compared to some other communication protocols, such as SPI (Serial Peripheral Interface) or UART (Universal Asynchronous Receiver-Transmitter), I2C typically has lower data transfer rates. This limitation can be a concern in applications that demand high-speed communication.

Clock Stretching Challenges: I2C supports clock stretching, where a slave device can hold the clock line low to slow down the master's clock. While this feature is beneficial, it can lead to challenges in timing and may require careful consideration in the design.

Complexity in Multi-Master Configurations: While I2C supports multi-master configurations, managing bus contention and potential collisions between masters can add complexity to the system. Implementing proper arbitration mechanisms is crucial in such scenarios.

Lower Noise Immunity: I2C may be more susceptible to noise and interference compared to differential signaling protocols. This can be a concern in environments where noise is a significant factor.

No Built-in Error Checking: I2C does not have built-in error-checking mechanisms, and it relies on the acknowledgment bit to confirm successful data reception. If an error occurs, the protocol may not detect it, and additional error-checking mechanisms may be needed at a higher level of the system.

Not Suitable for High-Density Systems: In systems with a large number of devices, the limited address space (even with 10-bit addressing) may become a limitation. Other protocols like SPI or CAN may be more suitable for high-density systems.

Complex Protocol for Simple Applications: I2C's feature-rich nature may be considered overkill for simple applications that do not require the advanced capabilities it offers. Simpler protocols like UART may be more straightforward for basic communication.

Incompatibility with 5V Systems: Some I2C devices operate at lower voltage levels, which may not be compatible with systems that use higher voltage levels (e.g., 5V). Level-shifting may be required in such cases.

UART Protocol

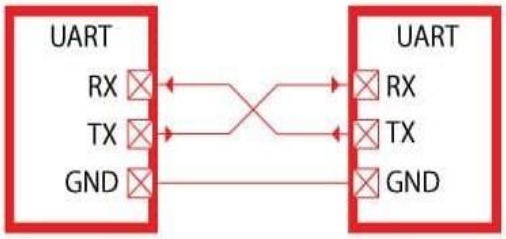


Fig.: UART Protocol

UART (Universal Asynchronous Receiver-Transmitter) is a popular serial communication protocol widely used for asynchronous communication between two devices. Unlike synchronous protocols, such as SPI or I2C, UART does not use a shared clock signal to synchronize data transfer. Instead, it relies on the sender and receiver agreeing on a specific baud rate (bits per second) for communication. UART is a flexible and straightforward protocol, making it a popular choice for various applications that require simple and reliable serial communication between devices. Its ease of implementation and widespread support make it a go-to solution for many embedded systems.

Asynchronous Communication: UART is asynchronous, meaning that there is no common clock signal shared between the communicating devices. Instead, the sender and receiver must agree on a specific baud rate to ensure proper timing for data transfer.

Two-Wire Communication: UART uses two wires for communication.

TX (Transmit): Carries data from the sender (transmitter) to the receiver.

RX (Receive): Carries data from the receiver (transmitter) to the sender.

Start and Stop Bits: Each data byte in a UART frame is framed by start and stop bits. The start bit indicates the beginning of the data byte, and the stop bit(s) signal the end of the byte. The most common configurations are 8-N-1 (eight data bits, no parity, one stop bit) or 8-E-1 (eight data bits, even parity, one stop bit).

Baud Rate: Baud rate represents the speed at which data is transmitted and received. Both the sender and receiver must operate at the same baud rate for successful communication. Common baud rates include 9600, 19200, 38400, 115200, and others.

Full-Duplex Communication: UART supports full-duplex communication, allowing simultaneous data transmission and reception. This is achieved through separate TX and RX lines.

Widely Used in Serial Communication: UART is commonly used for serial communication between devices, such as microcontrollers, sensors, GPS modules, and other embedded systems.

Simple Implementation: The simplicity of UART makes it easy to implement in both hardware and software. Many microcontrollers and communication modules include UART hardware interfaces.

No Master/Slave Distinction: Unlike protocols such as SPI or I2C, UART does not have a strict master or slave designation. Devices can communicate in a peer-to-peer fashion without a central controlling unit.

Versatile and Widely Supported: UART is a versatile protocol that is supported by a wide range of devices. It is often used for point-to-point communication but can also be used in multi-device scenarios with the proper wiring and addressing schemes.

Error Detection and Handling: UART does not have built-in error-checking mechanisms. However, error detection and handling can be implemented at higher levels of the communication stack, if needed.

Working of UART Protocol

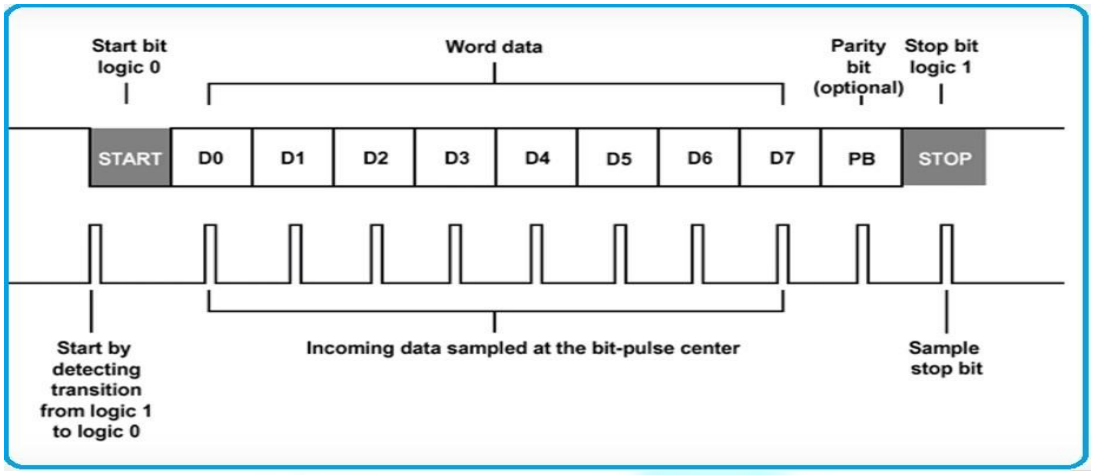


Fig.: UART Protocol Working

Data Frame Structure: UART communication involves the transmission of data in the form of frames. Each frame typically consists of a start bit, a specified number of data bits (usually 8 bits), an optional parity bit for error checking, and one or more stop bits. The start bit signals the beginning of a frame, and the stop bit(s) indicate the end. The start and stop bits help the receiving device synchronize with the transmitted data.

Asynchronous Communication: UART is asynchronous, meaning that there is no shared clock signal between the transmitting and receiving devices. Instead, both devices must agree on a specific baud rate, which represents the number of bits transmitted per second. The absence of a shared clock means that both devices need to be configured with the same baud rate to ensure proper communication.

Data Transmission: To transmit data, the UART sender takes each byte of data, adds the start bit, data bits, optional parity bit, and stop bit(s) to create a frame. The frame is then sent serially, one bit at a time, starting with the least significant bit (LSB) and ending with the stop bit(s). The receiving UART device continuously monitors the incoming data line for the start of a new frame, detects the start bit, and then samples the incoming bits based on the agreed-upon baud rate to reconstruct the transmitted byte.

Error Checking (Optional): Parity bit: Some UART implementations include a parity bit for error checking. The parity bit can be set to even or odd, and the receiving device checks whether the number of set bits in the received data matches the specified parity.

Flow Control (Optional): Flow control mechanisms, such as hardware (RTS/CTS) or software (XON/XOFF) flow control, may be used to manage the rate of data transmission between devices, preventing data overflow in the receiver's buffer. In summary, UART is a simple and versatile serial communication protocol that facilitates the transfer of data between devices. It is widely used due to its simplicity and effectiveness, especially in scenarios where a shared clock signal is not feasible or necessary.

SPI Protocol

The Serial Peripheral Interface (SPI) is a synchronous serial communication protocol used to transfer data between a master device and one or more peripheral devices. It is commonly used in embedded systems, microcontrollers, and other electronic applications. SPI is a versatile and widely used communication protocol in embedded systems due to its simplicity,speed, and support for multiple devices on a single bus. Developers often choose SPI when designing systems that require high-speed, full-duplex communication with multiple peripherals.

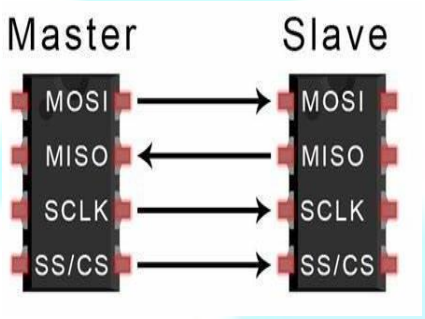


Fig.: SPI Protocol

Master-Slave Architecture: The communication involves one master device that initiates and controls the data transfer and one or more slave devices that respond to the master's commands.

Synchronous Communication: SPI is synchronous, meaning that data is transferred based on a clock signal (SCLK or SCK) shared between the master and the slave devices. The master generates the clock signal to synchronize the data transfer.

Full-Duplex Communication: SPI supports full-duplex communication, allowing data to be sent and received simultaneously. This is achieved through separate data lines for transmission (MOSI - Master Out Slave In) and reception (MISO - Master In Slave Out).

Multiple Slave Devices: SPI can support multiple slave devices on the same bus. Each slave device has a unique chip select (CS or SS) line that is activated by the master to select the specific slave with which it wants to communicate.

Data Frame Format: Data is typically sent in frames, with a frame consisting of a variable number of bits. The number of bits per frame is often configurable and can be 8, 16, or other values.

Configurable Clock Polarity and Phase: SPI supports configurable clock polarity (CPOL) and clock phase (CPHA), allowing flexibility in data sampling. These parameters determine whether data is sampled on the rising or falling edge of the clock signal.

Simple Protocol: SPI is relatively simple compared to other communication protocols, such as I2C or UART. It is suitable for high-speed communication in short-distance applications.

No Standardized Voltage Levels: SPI does not define specific voltage levels, making it flexible for use with various hardware configurations. However, it is essential for devices on the same SPI bus to operate at compatible voltage levels.

Working of SPI Protocol

Initialization: The master device initializes the SPI communication by configuring its SPI hardware, setting parameters such as clock polarity (CPOL), clock phase (CPHA), and data frame format.

Chip Select (CS) Activation: The master selects a specific slave device for communication by activating the Chip Select (CS) line corresponding to that slave. This informs the chosen slave that it should pay attention to the incoming data.

Clock Generation: The master generates a clock signal (SCLK) to synchronize data transfer. The clock signal has a frequency determined by the system requirements and is shared between the master and the selected slave.

Data Transmission (Master to Slave - MOSI): The master sends data to the slave on the Master Out Slave In (MOSI) line. Each bit of the data is shifted out on the rising or falling edge of the clock signal, depending on the configuration (CPHA).

Data Reception (Slave to Master - MISO): Simultaneously, the slave responds by sending data to the master on the Master In Slave Out (MISO) line. The master reads this data on the opposite edge of the clock signal. This allows for full-duplex communication.

Clock Polarity and Phase: The master and slave must agree on the clock polarity (CPOL) and clock phase (CPHA) settings. These settings determine the idle state of the clock signal and when data is sampled. The configuration is usually defined during initialization.

Frame Format: Data is typically sent in frames, with each frame containing a specified number of bits (e.g., 8 bits). The frame format may include additional bits for control purposes, such as a start bit, stop bit, or parity bit.

Data Transfer Completion: After the desired amount of data has been transferred, the master deactivates the Chip Select (CS) line, indicating the end of communication with the current slave.

Repeat for Multiple Slaves: If communication needs to occur with other slave devices on the bus, the master can repeat the process by activating the Chip Select (CS) line for the next slave.

Termination: Once all required communication is complete, the master device may terminate the SPI communication.

Advantages of SPI Protocol

High Speed: SPI supports high-speed communication, making it suitable for applications that require rapid data transfer. The clock frequency can often be adjusted to meet the speed requirements of the system.

Full-Duplex Communication: SPI supports full-duplex communication, allowing data to be transmitted and received simultaneously. This feature is advantageous in scenarios where real-time bidirectional communication is essential.

Simple Implementation: SPI has a relatively simple protocol compared to other communication interfaces like I2C or UART. This simplicity facilitates easier implementation, making it a preferred choice for many embedded systems.

Master-Slave Architecture: The master-slave architecture of SPI allows a single master device to communicate with multiple slave devices on the same bus. This makes it a versatile protocol for systems with multiple peripherals.

Multiple Slave Devices: SPI can support multiple slave devices on the same bus, each identified by its unique Chip Select (CS) line. This enables efficient communication with a variety of peripheral devices.

Configurable Clock Polarity and Phase: SPI allows the flexibility to configure clock polarity (CPOL) and clock phase (CPHA) to adapt to different device requirements. This configurability ensures compatibility with a wide range of devices.

No Acknowledgment Protocol: Unlike I2C, SPI does not require an acknowledgment protocol, simplifying the overall communication process. The absence of acknowledgment bits can reduce the complexity of the hardware and firmware.

Efficient for Short-Distance Communication: SPI is well-suited for short-distance communication within a circuit board or between closely located devices. Its simplicity and speed make it effective for these scenarios.

Suitable for Low-Power Devices: SPI can be power-efficient, especially when devices can enter low-power modes between communication transactions. The master can activate and deactivate individual slave devices as needed, conserving power.

Widely Supported: SPI is a widely supported protocol, and many microcontrollers, sensors, and other peripheral devices come with built-in SPI interfaces. This availability of SPI support contributes to its widespread adoption.

No Strict Voltage Levels: SPI does not define strict voltage levels, allowing it to be adaptable to different voltage requirements in various systems.

Disadvantages of SPI Protocol

Limited Distance: SPI is typically designed for short-distance communication within a PCB or between devices on the same board. It may not be suitable for long-distance communication due to signal degradation and noise susceptibility.

Point-to-Point Communication: SPI is generally a point-to-point communication protocol, which means that it is not well-suited for communication with multiple devices simultaneously unless a separate chip select line is used for each device. This can complicate the hardware design and increase the number of required pins.

Synchronous Communication: SPI relies on a synchronous communication mechanism, where data is transferred based on clock pulses. This can lead to timing issues, especially when interfacing with devices that have different clock speeds or require precise timing.

No Built-In Flow Control: SPI does not include built-in flow control mechanisms. This lack of flow control can result in potential data loss or corruption if there is a mismatch in the data rates between the communicating devices.

Higher Pin Count: Compared to other serial communication protocols like I2C, SPI typically requires more pins for communication. Each device in an SPI network may need dedicated lines for clock, data in, data out, and chip select, which can lead to increased pin count and more complex hardware designs.

Complex Protocol: SPI does not have a standardized protocol for addressing devices or managing transactions. It often requires a custom protocol implementation for specific applications, which can lead to more complex software development.

Applications of SPI Protocol

Microcontroller Communication: SPI is often used for communication between microcontrollers and peripheral devices such as sensors, displays, memory devices, and communication modules. Its simplicity makes it a preferred choice for such embedded systems.

Data Converters: SPI is frequently employed for interfacing with analog-to-digital converters (ADCs) and digital-to-analog converters (DACs). These components are crucial insystems where analog signals need to be converted to digital data or vice versa.

Flash Memory and EEPROMs: SPI is commonly used to interface with non-volatile memory devices like Flash memory and Electrically Erasable Programmable Read-Only Memory (EEPROM). It allows for high-speed data transfer for reading and writing data to memory.

Display Modules: SPI is utilized in various display technologies, including TFT (Thin-Film Transistor) and OLED (Organic Light-Emitting Diode) displays. It enables the microcontroller to send data quickly to update the screen.

Wireless Communication Modules: SPI is employed in communication modules such as RF (Radio Frequency) transceivers, Wi-Fi modules, and Bluetooth modules. These modules often use SPI for communication between the microcontroller and the wireless chip.

Sensors and Sensor Hubs: Many sensors, such as accelerometers, gyroscopes, and temperature sensors, use SPI for communication. Sensor hubs, which aggregate data from multiple sensors, may also use SPI to interface with the main microcontroller.

Motor Control: SPI can be used for communication between microcontrollers and motor control ICs. This is especially common in applications like robotics and industrial automation.

Audio Codecs: SPI is employed in audio systems for communication with audio codecs. It facilitates the transfer of audio data between microcontrollers and digital-to-analog converters or analog-to-digital converters.

FPGAs and CPLDs: SPI is often used to configure and communicate with Field-Programmable Gate Arrays (FPGAs) and Complex Programmable Logic Devices (CPLDs). It allows for rapid configuration and control of these programmable devices.

Automotive Electronics: SPI is used in various automotive applications, including communication with sensors, memory devices, and other control modules within the vehicle.

Industrial Control Systems: In industrial settings, SPI is employed for communication between microcontrollers and various control devices, sensors, or actuators used in automation and process control.

CAN Protocol

Networking: CAN is a multi-master, message-oriented protocol designed for networking in environments where noise and interference are common, such as in vehicles or industrial settings.

Message Format: CAN messages consist of an identifier, control bits, data, and a cyclic redundancy check (CRC). The identifier determines the priority of the message on the bus.

Two-Wire Communication: CAN uses a two-wire differential signaling scheme: CAN High (CAN_H) and CAN Low (CAN_L). This differential signaling helps in noise immunity.

Multi-Master Architecture: CAN allows multiple microcontrollers to communicate on the same bus without a central coordinator. It is a decentralized, multi-master network.

Collision Avoidance: CAN uses a non-destructive bitwise arbitration mechanism, where the message with the highest priority (lowest identifier) gets transmitted without collisions.

Error Detection and Handling: CAN provides built-in error detection and handling mechanisms. It uses CRC for error checking and can detect errors like bit errors, frame errors, and more. Additionally, it supports error frames to signal error conditions.

Flexible Data Rate (CAN FD): The original CAN protocol has a fixed data rate. However, the CAN FD extension allows for higher data rates and larger data payloads, making it suitable for applications with increased bandwidth requirements.

Real-Time Capability: CAN is designed for real-time communication, making it suitable for applications where timely and predictable data transmission is critical.

Common in Automotive Applications: CAN is widely used in the automotive industry for communication between various electronic control units (ECUs) within a vehicle. It facilitates communication between components like the engine control unit, transmission control unit, ABS (Anti-lock Braking System), airbags, and more.

Industrial Automation: CAN is also utilized in industrial automation for communication between sensors, actuators, and controllers in manufacturing environments.

Extended CAN (CANopen, J1939): There are several higher-layer protocols built on top of the CAN physical layer, such as CANopen and J1939, which define specific communication profiles and application layers for various industries.

Need of CAN Protocol

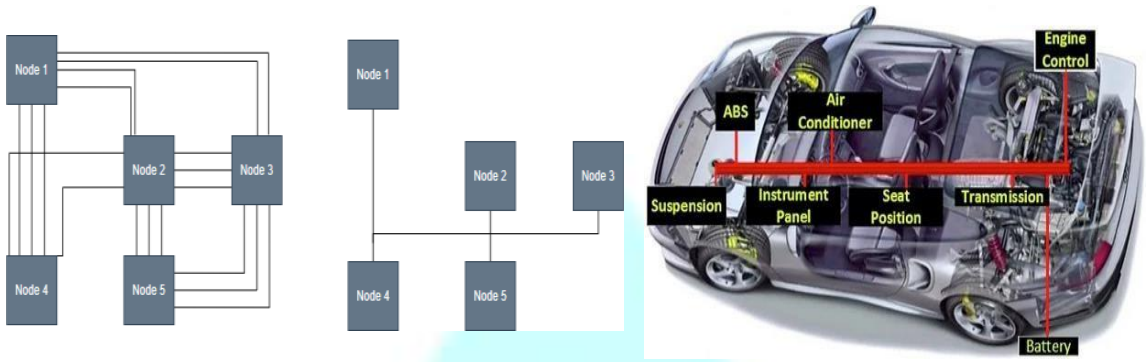


Fig.: ECUs In Automobiles

The need for a centralized standard communication protocol came because of the increase in the number of electronic devices. For example, there can be more than 7 TCU for various subsystems such as dashboard, transmission control, engine control unit, and many more in a modern vehicle. If all the nodes are connected one-to-one, then the speed of the communication would be very high, but the complexity and cost of the wires would be very high. In the above example, a single dashboard requires 8 connectors, so to overcome this issue, CAN was introduced as a centralized solution that requires two wires, i.e., CAN high and CAN low. The solution of using CAN protocol is quite efficient due to its message prioritization, and flexible as a node can be inserted or removed without affecting the network.

Onboard diagnostics (OBD) is your vehicle's diagnostic and reporting system that allows you or a technician to troubleshoot problems via diagnostic trouble codes (DTCs). When the “check engine” light comes on, a technician will often use a handheld device to read the engine codes off of the vehicle. At the lowest level, this data is transmitted via a signaling protocol, which in most cases is CAN

Reliability in Noisy Environments: CAN is designed to operate in noisy environments, such as those found in automotive and industrial settings. Its differential signaling and error-checking mechanisms make it resistant to electromagnetic interference, ensuring reliable communication.

Multi-Master Capability: CAN supports a multi-master architecture, allowing multiple microcontrollers to communicate on the same bus without requiring a central coordinator. This flexibility is crucial in distributed systems where different nodes need to communicate independently.

Deterministic and Real-Time Communication: CAN provides deterministic and real-time communication, making it suitable for applications where the timing of data transmission is critical. This is essential in systems where precise synchronization is required, such as automotive control systems.

Efficient Bandwidth Utilization: CAN's bitwise arbitration mechanism ensures efficient use of the available bandwidth. The protocol allows messages with higher priority to be transmitted first, reducing the risk of collisions and ensuring that critical messages are delivered promptly.

Error Detection and Handling: CAN includes robust error detection and handling mechanisms. It uses cyclic redundancy checks (CRC) to detect errors and supports error frames to signal fault conditions. This enhances the overall reliability of the communication.

Low Implementation Cost: CAN is cost-effective to implement, both in terms of hardware and software. The protocol is widely supported in microcontrollers, and the two-wire differential bus simplifies the physical layer, making it economical for mass production applications.

Scalability: CAN is scalable, and its design allows for easy integration of additional nodes into a network. New devices can be added without significant changes to the existing network infrastructure, making it suitable for systems that may evolve over time.

Wide Industry Adoption: The CAN protocol has been adopted as a standard in various industries, including automotive, industrial automation, medical devices, and more. This widespread adoption ensures compatibility and interoperability among different devices and systems.

Extended CAN Protocols: Higher-layer protocols built on top of the CAN physical layer, such as CANopen and J1939, provide standardized communication profiles for specific applications. These protocols enable interoperability and streamline the development of complex systems.

Automotive Applications: CAN is a de facto standard in the automotive industry, where it is used for communication between electronic control units (ECUs) in vehicles. It plays a crucial role in enabling functionalities such as engine control, transmission control, ABS, airbags, and more.

Applications of CAN Protocol

Automotive Systems: CAN is extensively used in the automotive industry for communication between Electronic Control Units (ECUs) in vehicles. It facilitates communication between components such as the engine control unit, transmission control unit, ABS (Anti-lock Braking System), airbags, dashboard instruments, and more.

Industrial Automation: In industrial settings, CAN is employed for communication between sensors, actuators, programmable logic controllers (PLCs), and other control devices. It is used in applications such as manufacturing automation, process control, and machinery control.

Medical Devices: CAN is utilized in medical devices and equipment for communication between different modules and sensors. It is commonly found in devices like infusion pumps, patient monitoring systems, and diagnostic equipment.

Agricultural Machinery: CAN is employed in agricultural machinery and equipment for communication between various electronic components. This includes tractors, harvesters, and other farm equipment that rely on electronic control systems.

Commercial Vehicles: CAN is used in communication systems for commercial vehicles, including trucks and buses. It enables communication between different systems such as engine control, transmission control, and safety features.

Aerospace and Defense: CAN is utilized in avionics and defense applications for communication between different systems and components. It is employed in aircraftsystems, unmanned aerial vehicles (UAVs), and military vehicles.

Railway Systems: CAN is applied in railway systems for communication between various control units, sensors, and other electronic components. It contributes to the control and monitoring of train systems.

Marine and Offshore Systems: In marine and offshore applications, CAN is used for communication between different electronic components, including navigation systems,engine control systems, and safety systems on ships and offshore platforms.

Home Automation: CAN finds applications in home automation systems, where it can beused for communication between smart devices, sensors, and controllers. It enables the integration and control of various home automation functionalities.

Telematics and Fleet Management: CAN is employed in telematics systems for communication between vehicles and fleet management systems. It helps in monitoring and managing vehicle performance, tracking, and diagnostics.

Renewable Energy Systems: CAN is used in renewable energy systems, such as wind turbines and solar power installations, for communication between different components ofthe control and monitoring systems.

Construction and Heavy Machinery: CAN is applied in construction equipment and heavy machinery for communication between electronic control units, sensors, and actuators. It enhances the control and efficiency of these machines.

CAN layered architecture

The Controller Area Network (CAN) protocol follows a layered architecture, similar to the OSI (Open Systems Interconnection) model. The CAN protocol stack is often simplified into two main layers: the Physical Layer and the Data Link Layer.

Physical Layer: The physical layer is the lowest layer of the CAN protocol stack and is responsible for the transmission and reception of bits over the physical medium. It defines the electrical characteristics, signaling, and connector specifications. The physical layer of CAN is differential, meaning it uses two wires: CAN High (CAN_H) and CAN Low (CAN_L). Common physical layer standards include ISO 11898-2 for high-speed CAN and ISO 11898-3 for low-speed CAN.

Data Link Layer: The Data Link Layer is the upper layer of the CAN protocol stack and is further divided into two sub-layers: the Logical Link Control (LLC) sub-layer and the Medium Access Control (MAC) sub-layer.

Logical Link Control (LLC) Sub-layer: Responsible for error checking and message framing. Includes the Message Authentication Code (MAC) and the Frame Check Sequence (FCS) for error detection. Ensures that the message being transmitted is error-free.

Medium Access Control (MAC) Sub-layer: Responsible for managing the access to the bus and controlling the flow of data. Includes the arbitration process for resolving conflicts when multiple nodes attempt to transmit simultaneously. Implements a Non-Destructive Bitwise Arbitration mechanism where lower identifier values have higher priority.

The CAN protocol uses a broadcast communication model, where all nodes on the bus receive transmitted messages. Each message has an identifier that determines its priority. The arbitration mechanism ensures that messages with lower identifiers have higher priority and get transmitted first. Additionally, the Data Link Layer of the CAN protocol supports two frame formats: the CAN 2.0A format (standard format) and the CAN 2.0B format (extended format). The standard format uses an 11-bit identifier, while the extended format uses a 29-bitidentifier, allowing for a larger address space.

It's important to note that the CAN protocol stack doesn't strictly adhere to the OSI model, as it combines aspects of both the physical and data link layers into a single layer. The simplicity and efficiency of CAN make it well-suited for real-time applications in automotive, industrial, and other embedded systems. Higher-layer protocols (like CANopen,J1939, etc.) can be implemented on top of the basic CAN protocol to provide additional functionality and application-specific features.

MODBUS Protocol

Communication Model: MODBUS follows a client-server or master-slave communication model. In this model, a master device (client) communicates with one or more slave devices (servers) on a network.

Serial and TCP/IP Versions: MODBUS has both serial and TCP/IP versions, making it adaptable to different communication media. The serial versions include MODBUS RTU (Remote Terminal Unit) and MODBUS ASCII, while the TCP/IP version is known as MODBUS TCP.

Serial Communication

MODBUS RTU (Remote Terminal Unit): RTU is a binary protocol that uses binary encoding for communication. It is transmitted in a continuous stream of characters, making it efficient for serial communication. RTU typically uses RS-485 or RS-232 for physical layer communication.

MODBUS ASCII: ASCII is a text-based protocol that represents data using ASCII characters. It includes a start-of-frame character (colon) and end-of-frame characters (carriage return and line feed). ASCII is less common than RTU and is generally slower due to its textual representation.

Ethernet Communication:

MODBUS TCP: MODBUS TCP is used for communication over Ethernet networks. It encapsulates MODBUS frames within TCP/IP packets. It allows for faster communication compared to serial versions.

Frame Format: A MODBUS frame consists of different fields, including the device address, function code, data, and error checking. The frame structure may vary slightly between RTU, ASCII, and TCP variants.

Function Codes: MODBUS uses function codes to define different operations or actions tobe performed by the slave devices. Common function codes include read/write holding registers, read/write input registers, read/write coils, and read/write discrete inputs.

Addressing: MODBUS uses addressing to identify specific data points in a device. Deviceson the network have unique addresses, and specific registers or coils within a device are accessed using these addresses.

Error Checking: MODBUS includes simple error-checking mechanisms, such as checksums or CRCs (Cyclic Redundancy Checks), to ensure the integrity of transmitted data.

Applications: MODBUS is widely used in various industries, including manufacturing,energy, and building automation. It is often used to control and monitor devices such asProgrammable Logic Controllers (PLCs), Remote Terminal Units (RTUs), sensors, andactuators.

Open Standard: MODBUS is an open standard, making it easy for different vendors to implement and support. This openness contributes to its widespread adoption.

Modbus Protocol in IoT applications

MODBUS protocol finds applications in IoT (Internet of Things) scenarios, particularly in industrial IoT (IIoT) and industrial automation. Here are some ways in which MODBUS isused in IoT applications:

Sensor and Actuator Communication: In IoT applications, sensors and actuators play a crucial role in collecting data and controlling physical devices. MODBUS is often used to facilitate communication between these devices and the central control system or gateway.This allows for the monitoring and control of various processes in real-time.

PLC (Programmable Logic Controller) Communication: Many industrial IoT systems rely on PLCs for control and automation. MODBUS is commonly used to enable communication between IoT platforms or gateways and PLCs. This allows for data exchange, control commands, and status updates between the central system and field devices.

Remote Monitoring and Control: MODBUS enables remote monitoring and control of devices in IoT applications. By using the protocol, IoT platforms can communicate with distributed sensors and devices, providing real-time data for monitoring and allowing remote control actions.

Data Acquisition Systems: IoT applications often involve data acquisition from various sources, such as sensors and meters. MODBUS facilitates the integration of data acquisition systems by enabling communication with these devices. This ensures that data is collected efficiently and transmitted to the central processing unit for analysis.

Building Automation: In smart building applications within the IoT ecosystem, MODBUSis used for communication between sensors, HVAC (Heating, Ventilation, and Air

Conditioning) systems, lighting controls, and other devices. This allows for centralizedcontrol and monitoring of building systems.

Energy Management: IoT applications in energy management leverage MODBUS for communication between smart meters, energy sensors, and control systems. This enables the monitoring of energy consumption, as well as the implementation of energy-efficient measures.

SCADA (Supervisory Control and Data Acquisition) Systems: SCADA systems, whichare integral to many IoT deployments, often use MODBUS for communication with field devices. This includes communication with RTUs (Remote Terminal Units) and other intelligent devices that gather data from the field and send it to a central SCADA system.

Industrial IoT Gateways: IoT gateways act as intermediaries between field devices and cloud-based or on-premises IoT platforms. MODBUS is used to establish communicationbetween these gateways and the diverse range of industrial devices, ensuring seamless integration and data exchange.

Predictive Maintenance: IoT applications focused on predictive maintenance utilize MODBUS to gather data from sensors and devices embedded in machinery. The collecteddata is then analyzed to predict potential failures and schedule maintenance activities, improving overall equipment efficiency.

Water and Wastewater Management: In IoT applications related to water and wastewater management, MODBUS is employed to connect and communicate with sensors and control devices for monitoring water quality, flow rates, and managing treatment processes.

MODBUS, with its simplicity and flexibility, continues to be a relevant protocol in IoT applications, especially in industrial settings where real-time communication and control are crucial. Its compatibility with a wide range of devices and systems makes it a valuable choice for integrating diverse components within an IoT ecosystem.

Application areas of modbus

- Modbus is a protocol developed by Modicon systems for transmitting data acrossserial lines between various electronic devices.
- There will be Modbus master and Modbus slaves, while master will be requesting the information; slaves will be supplying the information.
- Modbus protocol utilizes a simple message structure, which makes it easier todeploy.
- It is de-facto standard of the process control and automation industry.
- Since it is opensource protocol manufactures can build their applications without paying royalties.
- There are two variations of the Modbus protocol that go over serial associations.
    - One of these is Modbus RTU. This variety is more minimized, which uses binarydata communication.
    - The second one is Modbus ASCII. This rendition is more verbose, and it utilizes hexadecimal ASCII encryption of data that can be examined by human administrators.

Healthcare: For automated temperature monitoring Modbus can be used by hospital's IT department to monitor the temperature in single interface. Data from different floors can be directly taken via RS485 Modbus ADC devices.

Transportation: Traffic behavior detection The abnormal behavior of traffic can be detected by the cross referring with normal traffic patterns obtained through the Modbus TCP transactions.

Home automation: Easy transfer of data For transferring data from different sensors used in home automation devices can be done through Modbus protocol. Since the data can be transferred via single layer it will be much easier when we compare other protocols

Other Industries: Another main application of Modbus is while connecting industrial devices that need to communicate with other automation equipment. Other major industries include Gas and oil, Renewable energy sources like Wind, Solar, Geothermal and Hydel etc.

Application Layer

The application layer of IoT architecture is the topmost layer that interacts directly with the end-user. It is responsible for providing user-friendly interfaces and functionalities that enable users to access and control IoT devices. This layer includes various software and applications such as mobile apps, web portals, and other user interfaces that are designed to interact with the underlying IoT infrastructure. It also includes middleware services that allow different IoT devices and systems to communicate and share data seamlessly. The application layer also includes analytics and processing capabilities that allow data to be analysed and transformed into meaningful insights. This can include machine learning algorithms, data visualization tools, and other advanced analytics capabilities.

Four-Layer Architecture of IoT

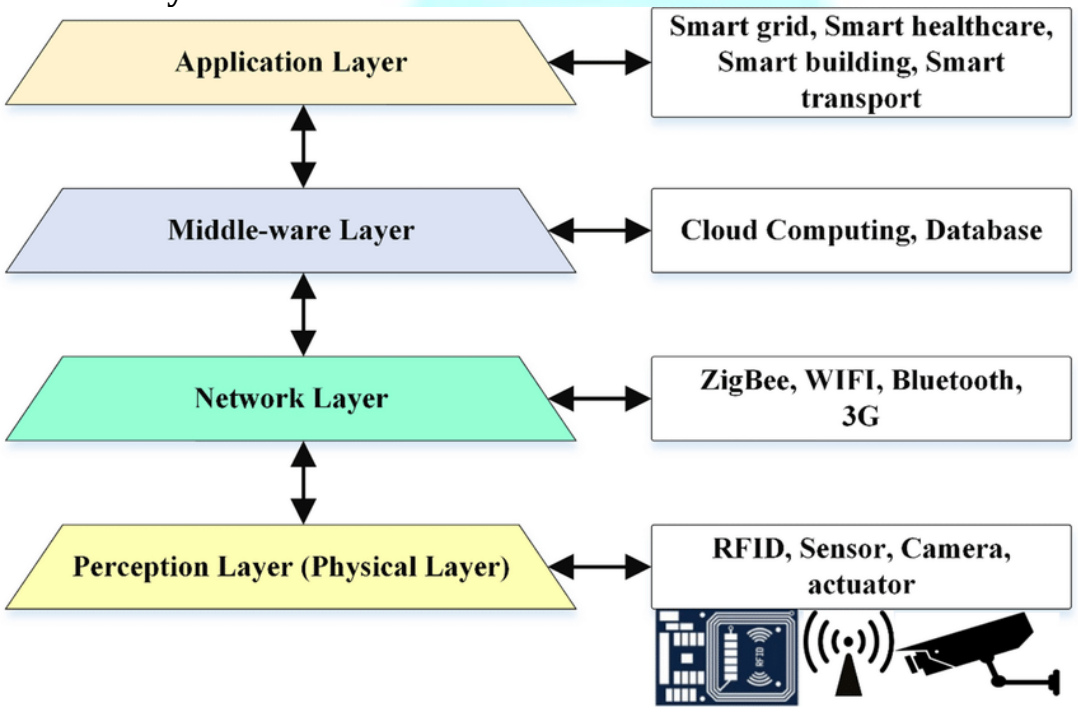


Fig.: 4-Layer Architecture

The four-layer architecture of the Internet of Things (IoT) provides a comprehensive framework for understanding how various components of IoT systems interact to deliver integrated and efficient solutions. This architecture is essential for designing IoT systems that are scalable, reliable, and secure. The four layers are:

1. **Perception Layer**
2. **Network Layer**
3. **Middleware Layer**
4. **Application Layer**

1. Perception Layer

The perception layer is the foundational layer of the IoT architecture. It is responsible for gathering data from the physical environment through various sensors and actuators.

Key Components:

- **Sensors:** Devices that detect and measure physical phenomena such as temperature, humidity, light, pressure, motion, and more. Examples include temperature sensors, motion detectors, and RFID tags.
- **Actuators:** Devices that can perform actions in response to commands, such as turning on a light or adjusting a thermostat.

Functions:

- **Data Collection:** Sensors capture raw data from the environment.
- **Data Conversion:** Analog signals from sensors are converted to digital data for processing.

The perception layer bridges the physical world and the digital world by collecting and digitizing environmental data.

2. Network Layer

The network layer facilitates the transmission of data collected by the perception layer to other parts of the IoT system. This layer ensures that data can move seamlessly from sensors and devices to the processing units and vice versa.

Key Components:

- **Communication Technologies:** Various wired and wireless communication protocols such as Wi-Fi, Bluetooth, Zigbee, LoRaWAN, and cellular networks.
- **Gateways and Routers:** Devices that aggregate data from multiple sensors and transmit it to the cloud or other processing systems.

Functions:

- **Data Transmission:** Ensures reliable communication between devices and network infrastructure.
- **Data Routing:** Directs data packets to their correct destination.
- **Security:** Implements encryption and authentication mechanisms to protect data during transmission.

The network layer is crucial for ensuring that data travels efficiently and securely from the edge devices to the processing and storage layers.

3. Middleware Layer

The middleware layer acts as an intermediary between the hardware-centric perception and network layers and the user-centric application layer. It provides a platform for data storage, processing, and management.

Key Components:

- **Data Management Systems:** Databases and data warehouses for storing large volumes of IoT data.
- **Processing Engines:** Systems for real-time and batch processing of data, such as cloud computing platforms and edge computing devices.
- **Middleware Services:** Software services that facilitate device management, data analytics, and system integration.

Functions:

- **Data Storage:** Stores large volumes of data generated by IoT devices.
- **Data Processing:** Analyzes data to extract useful insights, either in real-time (stream processing) or after data collection (batch processing).
- **Interoperability:** Ensures different devices and systems can work together by providing common communication protocols and data formats.

The middleware layer enables the efficient handling of vast amounts of data, making it possible to derive actionable insights and facilitate seamless integration of various IoT components.

4. Application Layer

The application layer is the topmost layer of the IoT architecture, directly interacting with end-users. It provides specific services and applications that utilize the data processed by the middleware layer.

Key Components:

- **User Interfaces:** Dashboards, mobile apps, and web applications that allow users to interact with the IoT system.
- **Application Services:** Domain-specific services such as smart home automation, industrial monitoring, healthcare management, and more.

Functions:

- **Service Delivery:** Provides end-users with the functionality and services derived from IoT data.
- **User Interaction:** Allows users to monitor and control IoT devices through various interfaces.
- **Visualization:** Displays data analytics results in a user-friendly manner, such as charts, graphs, and alerts.

The application layer is where the real-world impact of IoT is most visible, as it delivers the practical benefits and improvements enabled by IoT technology to end-users.

OSI Model

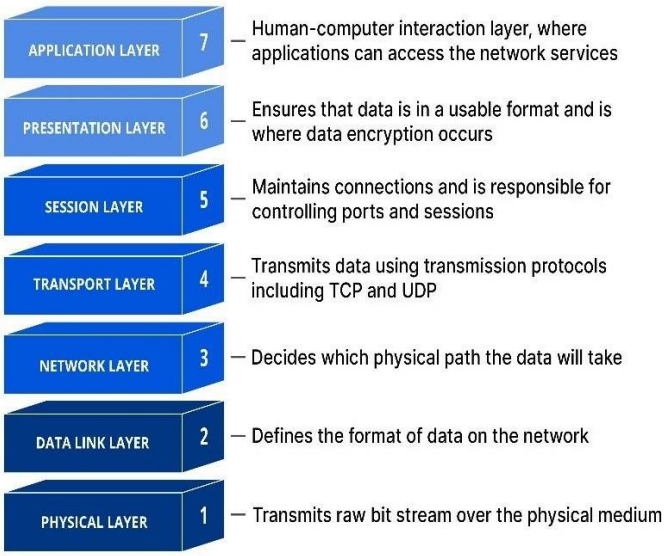


Fig.: OSI Model

The open systems interconnection (OSI) model is a conceptual model created by the International Organization for Standardization which enables diverse communication systems to communicate using standard protocols. In plain English, the OSI provides a standard for different computer systems to be able to communicate with each other.

The OSI Model can be seen as a universal language for computer networking. It is based on the concept of splitting up a communication system into seven abstract layers, each one stacked upon the last. Each layer of the OSI Model handles a specific job and communicates with the layers above and below itself.

7. The Application layer

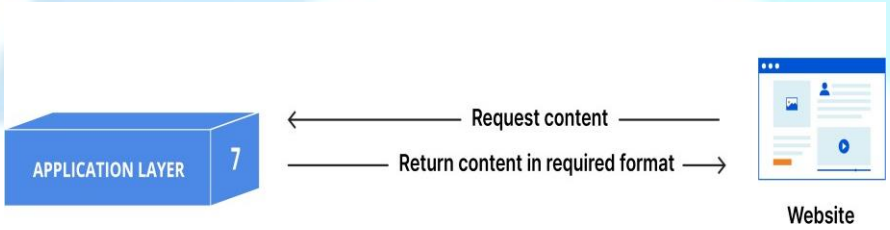


Fig.: Application Layer

This is the only layer that directly interacts with data from the user. Software applications like web browsers and email clients rely on the application layer to initiate communications. But it should be made clear that client software applications are not part of the application layer; rather the application layer is responsible for the protocols and data manipulation that the software relies on to present meaningful data to the user. Application layer protocols include HTTP as well as SMTP (Simple Mail Transfer Protocol is one of the protocols that enables email communications).

6. The Presentation layer

Fig.: Presentation Layer

This layer is primarily responsible for preparing data so that it can be used by the application layer; in other words, layer 6 makes the data presentable for applications to consume. The presentation layer is responsible for translation, encryption, and compression of data. Two communicating devices communicating may be using different encoding methods, so layer 6 is responsible for translating incoming data into a syntax that the application layer of the receiving device can understand. If the devices are communicating over an encrypted connection, layer 6 is responsible for adding the encryption on the sender's end as well as decoding the encryption on the receiver's end so that it can present the application layer with unencrypted, readable data. Finally the presentation layer is also responsible for compressing data it receives from the application layer before delivering it to layer 5. This helps improve the speed and efficiency of communication by minimizing the amount of data that will be transferred.

5. The Session layer

Fig.: Session Layer

This is the layer responsible for opening and closing communication between the two devices. The time between when the communication is opened and closed is known as the session. The session layer ensures that the session stays open long enough to transfer all the data being exchanged, and then promptly closes the session in order to avoid wasting resources. The session layer also synchronizes data transfer with checkpoints. For example, ifa 100 megabyte file is being transferred, the session layer could set a checkpoint every 5 megabytes. In the case of a disconnect or a crash after 52 megabytes have been transferred, the session could be resumed from the last checkpoint, meaning only 50 more megabytes of

data need to be transferred. Without the checkpoints, the entire transfer would have to begin again from scratch.

4. The Transport layer

Fig.: Transport Layer

Layer 4 is responsible for end-to-end communication between the two devices. This includes taking data from the session layer and breaking it up into chunks called segments before sending it to layer 3. The transport layer on the receiving device is responsible for reassembling the segments into data the session layer can consume. The transport layer is also responsible for flow control and error control. Flow control determines an optimal speedof transmission to ensure that a sender with a fast connection does not overwhelm a receiver with a slow connection. The transport layer performs error control on the receiving end by ensuring that the data received is complete, and requesting a retransmission if it isn't.
Transport layer protocols include the Transmission Control Protocol (TCP) and the User Datagram Protocol (UDP).

3. The Network layer

Fig.: Network Layer

The network layer is responsible for facilitating data transfer between two different networks. If the two devices communicating are on the same network, then the network layer is unnecessary. The network layer breaks up segments from the transport layer into smaller units, called packets, on the sender's device, and reassembling these packets on the receiving device. The network layer also finds the best physical path for the data to reach its destination; this is known as routing. Network layer protocols include IP, the Internet ControlMessage Protocol (ICMP), the Internet Group Message Protocol (IGMP), and the IPsec suite.

2. The Data Link layer

Fig.:Data link Layer

The data link layer is very similar to the network layer, except the data link layer facilitates data transfer between two devices on the same network. The data link layer takes packets from the network layer and breaks them into smaller pieces called frames. Like the network layer, the data link layer is also responsible for flow control and error control in intra-network communication (The transport layer only does flow control and error control for inter- network communications).

1. The Physical Layer

Fig.: Physical Layer

This layer includes the physical equipment involved in the data transfer, such as the cables and switches. This is also the layer where the data gets converted into a bit stream, which is astring of 1s and 0s. The physical layer of both devices must also agree on a signal conventionso that the 1s can be distinguished from the 0s on both devices.

How data flows through the OSI Model

In order for human-readable information to be transferred over a network from one device to another, the data must travel down the seven layers of the OSI Model on the sending device and then travel up the seven layers on the receiving end.

For example: Mr. Cooper wants to send Ms. Palmer an email. Mr. Cooper composes his message in an email application on his laptop and then hits ‘send’. His email application will pass his email message over to the application layer, which will pick a protocol (SMTP) and pass the data along to the presentation layer. The presentation layer will then compress the data and then it will hit the session layer, which will initialize the communication session.

The data will then hit the sender’s transportation layer where it will be segmented, then those segments will be broken up into packets at the network layer, which will be broken down even further into frames at the data link layer. The data link layer will then deliver those frames to the physical layer, which will convert the data into a bitstream of 1s and 0s and send it through a physical medium, such as a cable.

Once Ms. Palmer’s computer receives the bit stream through a physical medium (such as her wifi), the data will flow through the same series of layers on her device, but in the opposite order. First the physical layer will convert the bitstream from 1s and 0s into frames that get passed to the data link layer. The data link layer will then reassemble the frames into packets for the network layer. The network layer will then make segments out of the packets for the transport layer, which will reassemble the segments into one piece of data.

The data will then flow into the receiver's session layer, which will pass the data along to the presentation layer and then end the communication session. The presentation layer will then remove the compression and pass the raw data up to the application layer. The application layer will then feed the human-readable data along to Ms. Palmer’s email software, which will allow her to read Mr. Cooper’s email on her laptop screen.

IoT Gateway

An IoT Gateway is a solution for enabling IoT communication, usually device -to-device communications or device-to-cloud communications. The gateway is typically a hardware device housing application software that performs essential tasks. At its most basic level, the gateway facilitates the connections between different data sources and destinations.

A simple way to conceive of an IoT Gateway is to compare it to your home or office network router or gateway. Such a gateway facilitates communication between your devices, maintains security and provides an admin interface where you can perform basic functions.
An IoT Gateway does this and much more.

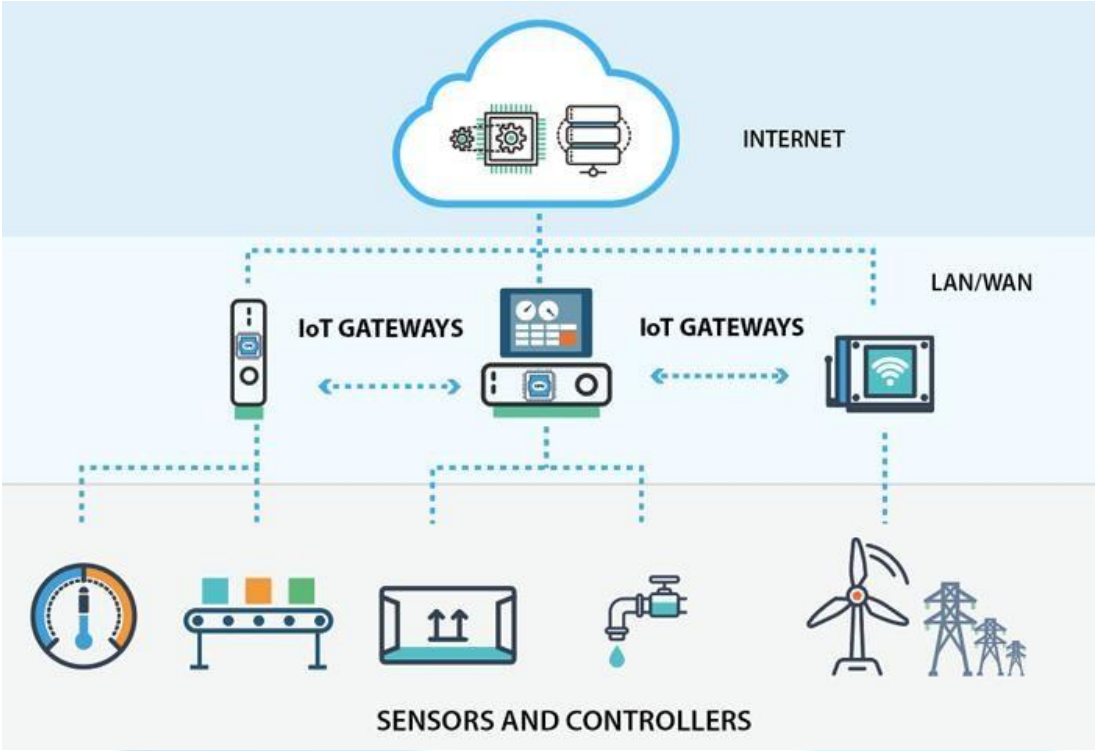


Fig.: IOT Gateway Working

Working of IoT Gateway

The working of an IoT gateway involves several key functions that enable communication and data flow between edge devices and the central/cloud-based system.

Device Connection: Edge devices, such as sensors and actuators, are connected to the IoT gateway. These devices could use various communication protocols (e.g., MQTT, CoAP, HTTP), and the gateway is equipped to handle these diverse protocols.

Protocol Translation: The IoT gateway translates the different communication protocols used by edge devices into a common format. This ensures that devices with different protocols can communicate seamlessly with each other and with the gateway.

Data Aggregation and Processing: The gateway aggregates data from multiple connected devices. It may perform initial processing tasks such as data filtering, normalization, or summarization. This processing can help in reducing the amount of data that needs to be transmitted to the cloud, optimizing bandwidth usage.

Local Storage and Edge Computing: Some IoT gateways have the capability to store and process data locally. This is particularly useful in scenarios where low latency or intermittent connectivity is a concern. The gateway may perform certain computations locally before transmitting relevant information to the central system.

Security Measures: The IoT gateway implements security measures to protect the data and communication. This can include encryption of data, authentication of devices, and authorization controls to ensure that only authorized devices can access and send/receive data.

Communication with the Cloud: Processed and aggregated data is transmitted from the IoT gateway to the central or cloud-based server. The communication may happen over various networking technologies such as Wi-Fi, Ethernet, cellular networks, or other wireless protocols.

Device Management: The IoT gateway is responsible for managing connected devices. This includes tasks such as monitoring the health of devices, applying software updates, and handling configuration changes. Device management ensures the overall reliability and efficiency of the IoT network.

Scalability and Connectivity Management: As the number of edge devices increases, the IoT gateway can efficiently manage the growing volume of data and connections. It handles issues related to network availability, optimizes data transmission over available communication channels, and adapts to changes in the network topology.

Architecture of an IoT Gateway

An IoT gateway is a central hub connecting IoT devices and sensors to cloud-based data processing and computing. Its architecture is simple yet unique. Here are 12 components that make up an IoT gateway.

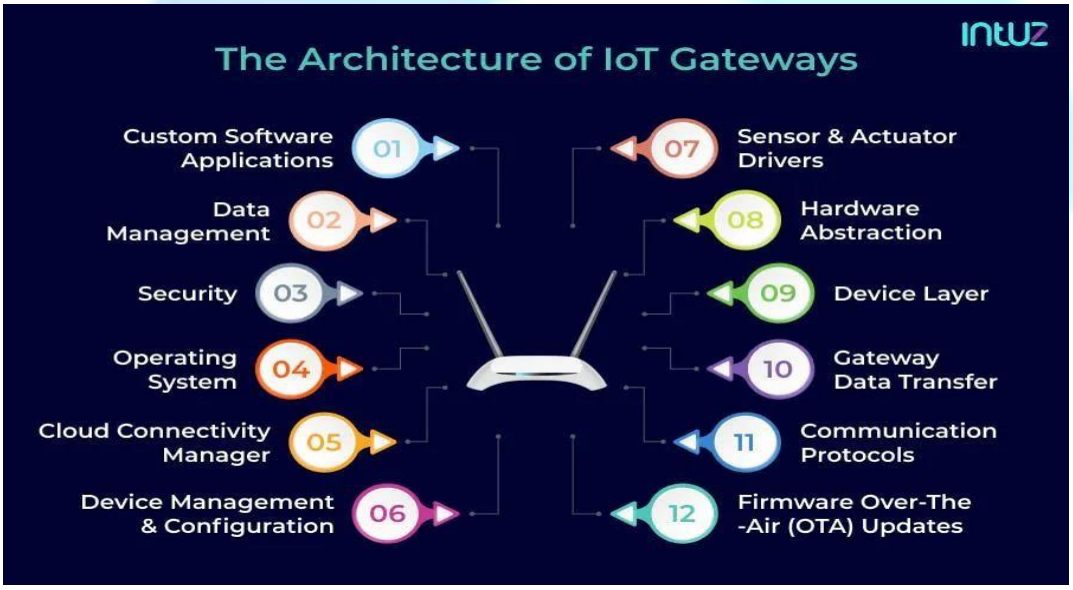


Fig.:IOT Gateway Architecture

1. Security: Security is one of the most critical factors in an IoT gateway architecture throughout the design phase. Using Crypto Authentication chips, device security and identity are implemented in the gateway hardware. Hardware tampering is done on the IoT gateway to boost overall device security. Additionally, a secure boot is used to prevent the gateway from functioning with illegal firmware. To guarantee data integrity and sensor node confidentiality, all messages between the gateway and the cloud, as well as messages between the gateway sensor nodes, are encrypted. To maintain network security, data is encrypted as it travels to and from each node in an app.

2. Device layer: IoT sensors, protective circuits, networking modules, and a processor or microcontroller make up the hardware of an IoT infrastructure. The IoT gateway's operating system determines the type of hardware (processor/microcontroller), processing speed, and memory space. The design of the IoT hardware also heavily depends on the end-user application.

3. Data management: Data streaming, filtering, and storage are all parts of data management (in case of loss of IoT device connectivity with the cloud). Data from the sensor nodes to the gateway and from the gateway to the cloud are both managed by IoT gateways. Here, the difficulty is in reducing the time while maintaining data quality.

4. Operating system: The IoT application is a crucial factor in the operating system choice. Real-Time Operating Systems (RTOS) are employed when designing a gateway for simple to medium-sized applications. However, Linux is preferable when the gateway needs to do complicated activities.

5. Hardware abstraction: The abstraction layer enables independent hardware-free software development and management. This facilitates software evolution and upgrades by enhancing the flexibility and agility of the IoT application design.

6. Gateway data transfer: This component regulates how the IoT gateway connects to the Internet through Ethernet, WiFi, a 5G/4G/3G/GPRS modem, an IoT module, or any combination of those. Additionally, to conserve processing power and data plan costs, the gateway analyses and determines which data must be transmitted to the cloud and which data must be cached for offline processing.

7. Communication protocols: IoT gateway protocols are chosen depending on the volume and frequency of data sent to the cloud. Although cellular modules (5G/4G/3G), Ethernet, and/or WiFi are typically required for gateway connections, the underlying communication protocol layer is the TCP/IP model.

8. Cloud connectivity manager: This component manages scenarios including reconnection, device status, heartbeat message, gateway device authentication with the cloud, and being in charge of flawless connectivity with the cloud.

9. Sensor and actuator drivers: Sensors and actuator drivers provide the IoT device's interface with sensors and modules. Depending on what the IoT application requires, specific stacks are merged.

10. Custom software applications: The IoT gateway application is specifically created to meet the organization's needs. To manage data between the sensor node and the IoT gateway and from the IoT gateway to the cloud in an effective, safe and timely manner, the gateway application interacts with the services and functions from all the other layers or modules.

11. Firmware Over-The-Air (OTA) updates: The key to maintaining IoT device integrity is to keep the firmware updated and enable security upgrades and fixes to protect against constantly changing threats. To protect device memory, power, and network traffic specifically, this component ensures that the Firmware Over-The-Air (OTA) updates are managed securely and effectively.

12. Device management and configuration: IoT gateways must keep track of all the connected objects and sensors with which they interact. This component keeps tabs on and controls sensor ecosystem-wide configurations, settings, properties, and IoT-connected devices.

Functionalities of IoT gateways

Modern IoT gateways often allow bi-direction data flow between the IoT devices and the cloud. This allows uploading IoT sensor data for processing and sending commands from cloud-based apps to IoT devices. Adding to that thought, here is a list of functions that a flexible IoT gateway could carry out

- System analysis
- Data aggregation
- Device configuration management
- M2M/device-to-device communications
- Caching, buffering, mixing streaming IoT data
- Networking capabilities and hosting live data
- Managing user access and network security features
- Pre-processing, cleanup, filtering, and optimization of data
- Facilitating connection with older or non-internet connected devices

IoT gateways: The ultimate bridge between IoT devices and servers

A genuine IoT gateway includes communication technologies that link the backend platforms for data, devices, subscriber administration, and end devices (sensors, actuators, or more complicated devices) to the gateway.

Fig.: IOT Gateway between devices and servers

An IoT gateway has a computing platform that enables pre-installed or user-defined programs to manage data, devices, security, communication, and other gateway-related functions for routing and computation on Edge. Gateways provide both short-term as well as long-term benefits, some of which are mentioned below.

1. Data filtration: Gateways enable IoT devices to communicate with one another, changing the way they do so by transforming data into valuable information. They operate on Edge, deliver the required filtered data to the cloud, and accelerate reaction and communication times. There is no requirement for a service provider to process the data because it can be handled and comprehended by the IoT gateway.

2. Risk mitigation: Security threats increase with the increase in the number of IoT devices. Thank goodness, a layer between the internet and IoT devices is provided by an IoT gateway, thereby preventing security issues.

3. Improved connectivity: Gateways function as universal remote controls that let you manage various IoT devices from one central location. It helps the establishments save a ton of time and effort. Facility managers interact with IoT systems through gateway internet devices using cloud-enabled applications to receive or send data to them.

4. Easy communication among devices: There are numerous devices and sensors in an IoT ecosystem. There is currently no common language used between these devices. Information is transmitted amongst them through gateways. It makes the entire communication process simpler.

Top 10 IoT gateways

IoT gateways are becoming increasingly important as the number of connected devices continues to grow. You cannot simply ignore what they offer in an IoT landscape if you want uninterrupted and safe data transmission and communication.

1. ReliaGATE 20-25

The ReliaGate 20-25 is LTE-ready, high-performance, and cloud-certified IoT edge gateway. It is a part of Eurotech IoT gateways. This one is best suited for challenging industrial environments and lightly rugged IoT applications because it was created using the exclusive Everyware Software Framework or ESF.

Functions:

- Excellent security
- Powerful remote access capabilities
- Amazing hardware diagnostics options
- A wide range of operating temperatures
- Interesting customization opportunities
- High-quality Bluetooth and WiFi connectivity
- Fast and smooth Everyware Cloud integration

2. Kontron KBox A-201

This KBox gateway functions best as a multi-featured industrial computing platform and has a fanless, battery-less, and soldered memory design. By the way, the design is dubbed "wartungsfrei." The gateway's cost-effectiveness is still another strength.

Functions:

- 15G shock resistance
- A wide range of hardware interfaces
- Powerful Intel Quark X1011 built-in processor
- Compatible with extreme operating temperatures

3. Advantech WISE-3310

This wireless IoT mesh network gateway from Advantech achieves high-performance levels and scores highly on the dependability scale since the Intelligent Gateway concept powers them. A sophisticated industrial IoT platform, the WISE-3310 gateway has built-in capability for 200 wireless nodes.

Functions:

- Power-saving mode
- Linux 3.10.17 embedded into the hardware
- IEEE 802.15.4e and the 6LoWPAN standards
- Processor: Freescale i.MX6 Dual Cortex 1 GHz
- Has the fanless underlying design for embedded IoT appsMultiple mounting (desktop mounting, cabinet mounting, and DIN RAIL)

4. Cisco 910 Industrial Router

One of the rapidly growing sub-sectors in IoT is the market for smart cities. The rugged designs of the Cisco 910 Industrial Router were mainly created to enable functioning high-end smart city IoT applications.

Functions:

- Glitch-free fog computing
- On-board memory capability
- Excellent horsepower support
- Different RF bands supported
- Cutting-edge modular slot design

5. Adlink MXE-5400i gateways

The Adlink IoT gateways are powered by the fourth-generation Intel Core (i3, i5, i7) processor and have official certification from Microsoft Azure. The gateways in the MXE-5400i series outperform in terms of CPU performance - while limiting overall power consumption.

Functions:

- Rugged design
- Vibration resistant
- Cable-free construction
- Access to the Wind River IDP and the robust QM87 chipset for high performance
- Intel vPro technology with Smart Embedded Management Agent (SEMA) by Adlink for greater security

6. B-Scada ethernet gateway

Unlike the bulk of the IoT gateways mentioned in this list, the B-Scada ethernet gateway offers end-to-end, real-time, wireless sensing capabilities. It enables the wireless IoT sensors to communicate with the cloud server and access sensor data from anywhere, anytime, on any web device.

Functions

- Pre-configured plug-and-play functionality
- A single gateway that can accommodate up to 50 sensors
- Three distinct frequency ranges: 433 MHz, 868 MHz, and 915 MHz
- Enables remote access to the retrieval of sensor data from any place

7. AR550E Series IoT Gateway

The AR550E Series IoT Gateway is a powerful IoT router specifically engineered for outdooroperations, manufacturing, transportation, and power utilities.

Functions

- Offers discrete advantages
- Network integration and sharing via virtualization
- Ethernet control and advanced Smart Grid operations
- Helps in retaining functionality under challenging conditions, such as temperature and humidity extremes and electromagnetic disturbances
- Multimedia and video services in different indoor and outdoor locations, including 'connected cars'

8. Dell Edge Gateway 500 Series

Dell Edge Gateway 500 is an ideal IoT gateway that excels at providing affordable solutions. It is simple to mount on walls and DIN rail and can be used in business and industrial contexts.

Functions:

- OS: Ubuntu Core 15.04
- Compatible with M.2 SATA
- DDR3L (2GB) memory space
- Processor: Intel Atom E3825
- 32GB solid state hard drive
- Operates even in extreme temperatures (compatible temperature range: -30℃ to 70℃)

9. HPE Edgeline Converged Edge Systems

Edgeline IoT gateways are now available in two different configurations from HP - the ten and the 20. This gateway is meant to work with the HP Moonshot architecture to boost overall performance, density, and power levels.

Functions

- Superior data management
- Offers strict data security
- Utilizes a Core i5 processor from Intel
- Operates in sync with the Microsoft Azure IoT
- Faster gathering, assembling, and analysis of crucial data/figures

10. FlexaGate FG400 IoT analytics gateway

When web connectivity is poor and/or intermittent, and storage space or cloud bandwidth usage is an issue, the FlexaGate FG400 IoT analytics gateway is especially helpful.

Functions

- High performance
- Distributed memory architecture
- Handy 'passive cooling' features
- Minimizes processing, memory, and bandwidth costs

Analyzing speed bolstered by the top-notch Ethernet input (RJ45 8×10 Gbps) and Ethernet output (RJ45 1×1 Gbps) capabilities

IoT Protocols

IoT protocols and standards are often overlooked when people think about the Internet of Things (IoT). More often than not, the industry has its attention firmly fixed upon communication. And while the interaction among devices, IoT sensors, gateways, servers, and user applications is essential to IoT, communication would fall down without the right IoT protocols.

Why are IoT protocols important?

IoT protocols are an integral part of the IoT technology stack. Without IoT protocols and standards, hardware would be useless. This is because IoT protocols are the things that enable that communication—that is, the exchange of data or sending commands—among all those various devices. And, out of these transferred pieces of data and commands, end users can extract useful information as well as interact with and control devices.

How many IoT protocols are there?

In short, a lot. The IoT is heterogeneous, meaning that there are all sorts of different smart devices, protocols, and applications involved in a typical IoT system. Different projects and use cases might require different kinds of devices and protocols.

For example, an IoT network meant to collect weather data over a wide area needs a bunch of different types of sensors, and lots of them. With that many sensors, the devices need to be lightweight and low-power, otherwise the energy required to transmit data would be enormous. In such an instance, low-power is the main priority, over, say, security or speed oftransmission.

If, however, an IoT system consists of medical sensors on ambulances, sensors that transmit patient data ahead to hospitals, time is clearly going to be of the essence. What's more, HIPAA requires special security protocols for health data. So a higher-power, faster, and more secure protocol would be necessary.

Since there are so many different types of IoT systems and so many different applications, experts have figured out a way to sort all the components of IoT architecture into different categories, called layers. These layers allow IT teams to home in on the different parts of a system that may need maintenance, as well as to promote interoperability. In other words, if every system adheres to, or can be defined by, a specific set of layers, the systems are more likely to be able to communicate with each other through those layers.

Application protocols can include:

- Extensible Messaging and Presence Protocol (XMPP)
- Message Queuing Telemetry Transport (MQTT)
- Constrained Application Protocol (CoAP)
- Simple Object Access Protocol (SOAP)
- Hypertext Transfer Protocol (HTTP)

Network layer protocols, which allow devices in an IoT network to communicate with each other, can include:

- Bluetooth
- Ethernet
- Wi-Fi
- Zigbee
- Thread
- Cellular networks (4G or 5G)

What protocols do IoT-qualified devices use?

In essence, IoT protocols and standards are broadly classified into two separate categories. These are:

1. IoT data protocols (Presentation / Application layers)
2. Network protocols for IoT (Datalink / Physical layers)

IoT Data Protocols

IoT data protocols are used to connect low-power IoT devices. They provide communication with hardware on the user side—without the need for any internet connection. The connectivity in IoT data protocols and standards is through a wired or cellular network. Someexamples of IoT data protocols are:

Extensible Messaging and Presence Protocol (XMPP)

XMPP is a fairly flexible data transfer protocol that is the basis for some instant messaging technologies, including Messenger and Google Hangouts. XMPP is an open protocol, freely available, and easy to use. That's why many use the protocol for machine-to machine (M2M) communication between IoT devices, or for communication between a device and the main server. XMPP gives devices an ID that's similar to an email address. Then the devices can communicate reliably and securely with each other. Experts can tailor XMPP to different use cases and for transferring both unstructured data or structured data like a fully formatted text message.

CoAP (Constrained Application Protocol)

A CoAP is an application layer protocol. It's designed to address the needs of HTTP-based IoT systems. HTTP is the foundation of data communication for the World Wide Web. Whilethe existing structure of the internet is freely available and usable by any IoT device, it's often too heavy and power-consuming for IoT applications. This has led many within the IoTcommunity to dismiss HTTP as a protocol not suitable for IoT. However, CoAP has addressed this limitation by translating the HTTP model into usage in restrictive devices and network environments. It has incredibly low overheads, is easy to employ, and has the ability to enable multicast support. Therefore, CoAP is ideal for use in devices with resource limitations, such as IoT microcontrollers or WSN nodes. It's traditionally used in applicationsinvolving smart energy and building automation.

AMQP (Advanced Message Queuing Protocol)

An AMQP is an open standard application layer protocol used for transactional messages between servers.

The main functions of this IoT protocol are as follows:

- Receiving and placing messages in queues
- Storing messages
- Setting up a relationship between these components

With its high level of security and reliability, AMQP is most commonly employed in settings that require server-based analytical environments, such as the banking industry. However, it's not widely used elsewhere. Due to its heaviness, AMQP is not suitable for IoT sensor devices with limited memory. As a result, its use is still quite limited within the world of the IoT.

DDS (Data Distribution Service)

DDS is another scalable IoT protocol that enables high-quality communication in IoT. Similar to the MQTT, DDS also works to a publisher-subscriber model. It can be deployed inmultiple settings, from the cloud to very small devices. This makes it perfect for real-time and embedded systems. Moreover, unlike MQTT, the DDS protocol allows for interoperable data exchange that is independent of the hardware and the software platform. In fact, DDS is considered the first open international middleware IoT standard.

HTTP (HyperText Transfer Protocol)

HTTP protocol is not preferred as an IoT standard because of its cost, battery life, huge power consumption, and weight issues. That being said, it is still used within some industries.For example, manufacturing and 3-D printing rely on the HTTP protocol due to the large amounts of data it can publish. It enables PC connection to 3-D printers in the network and printing of three-dimensional objects.

WebSocket

WebSocket was initially developed back in 2011 as part of the HTML5 initiative. WebSocket allows messages to be sent between the client and the server via a single TCP connection.
Like CoAp, WebSocket has a standard connectivity protocol that helps simplify many of the complexities and difficulties involved in the management of connections and bi-directional communication on the internet. WebSocket can be applied to an IoT network where data is communicated continuously across multiple devices. Therefore, you'll find it used most commonly in places that act as clients or servers. This includes runtime environments or libraries.

Network Protocols for IoT

IoT network protocols are used to connect devices over a network. These sets of protocols are typically used over the internet. Here are some examples of various IoT network protocols.

Lightweight M2M (LWM2M)

Many IoT systems rely on resource-constrained devices and sensors, devices that use very little power. The problem is that any communication between these devices or from the devices to a central server must also be extremely lightweight and require little energy.

Gathering meteorological data is one use case that requires hundreds or thousands of sensors over a wide area. Imagine the impact on the environment if all of those devices needed a lot of energy to function and transfer data. Instead, experts rely on lightweight communication protocols to reduce the energy consumption of such systems. One of the network protocols that greatly facilitates lightweight communication is the Lightweight M2M (LWM2M) protocol. LWM2M provides remote, long-distance connectivity between devices in an IoT system. The protocol primarily transfers data in small, efficient packages.

Cellular

Cellular networks are another method for connecting IoT devices. Most IoT systems rely on the 4G cellular protocol, though 5G has some applications as well. Cellular networks generally take more power than most of the protocols mentioned above, but perhaps more importantly, a cellular connection requires you to pay for a sim card. When you need a lot of devices over a wide area, the cost can quickly become prohibitive. Still, cellular networks arevery low-latency and can support high speeds for data transfer. 4G in particular was a huge step above 3G, since 4G is ten times faster. When you use your smartphone's data to control or interact with some IoT devices remotely, you're typically using 4G or 5G. An example would be when you use your cell data to view the feed from your smart security cameras when you're away from home.

Wi-Fi

Wi-Fi is the most well-known IoT protocol on this list. However, it's still worth explaining how the most popular IoT protocol works. In order to create a Wi-Fi network, you need a device that can send wireless signals. These include:

- Telephones
- Computers
- Routers

Wi-Fi provides an internet connection to nearby devices within a specific range. Another way to use Wi-Fi is to create a Wi-Fi hotspot. Mobile phones or computers may share a wireless or wired internet connection with other devices by broadcasting a signal. Wi-Fi uses radio waves that broadcast information on specific frequencies, such as 2.4 GHz or 5GHz channels. Furthermore, both of these frequency ranges have a number of channels through which different wireless devices can work. This prevents the overflowing of wireless networks. A range of 100 meters is common for a Wi-Fi connection. That being said, the most common is limited to 10 to 35 meters. The main impacts on the range and speed of a Wi-Fi connection are the environment and whether it provides internal or external coverage.

Bluetooth

Compared to other IoT network protocols listed here, Bluetooth tends to frequency hop and has a generally shorter range. However, it's gained a huge user base due to its integration into modern mobile devices—smartphones and tablets, to name a couple—as well as wearable technology, such as wireless headphones. Since many different devices and systems can use bluetooth, it's great for promoting interoperability, enabling communication between vastly different systems and applications. Standard Bluetooth technology uses radio waves in the 2.4 GHz ISM frequency band and is sent in the form of packets to one of 79 channels. However, the latest Bluetooth 4.0 standard has 40 channels and a bandwidth of 2Mhz. This guarantees a maximum data transfer of up to 3 Mb/s. This new technology is known as Bluetooth Low Energy (BLE) and can be the foundation for IoT applications that require significant flexibility, scalability, and low power consumption. BLE wasn't originally a Bluetooth technology. Nokia developed the concept, which soon gained the notice of the Bluetooth Special Interest Group. Since then, Bluetooth fully adopted BLE as an important communication protocol. But because of the low range of Bluetooth, it's not ideal for all use cases. Additionally, since any Bluetooth-enabled device is discoverable to any other bluetooth enabled device for pairing, BLE doesn't have a built-in security feature to prevent unauthorized connections. You would need to add security measures at the application layer. So while BLE makes a great option for smart homes and offices, where there are a reasonablenumber of IoT devices in proximity that make up a single IoT system, BLE is not so great fortransferring sensitive data over long distances or remotely accessing an IoT system from a long distance.

ZigBee

ZigBee-based networks are similar to Bluetooth networks in the sense that ZigBee already has a significant user base in the world of IoT. However, its specifications slightly eclipse themore universally used Bluetooth. ZigBee has lower power consumption, low data-range, high security, and has a longer range of communication. (ZigBee can reach 200 meters, while Bluetooth maxes out at 100 meters.) Zigbee is a relatively simple protocol, and is often implemented in devices with small requirements, such as microcontrollers and sensors. Furthermore, it easily scales to thousands of nodes. No surprise that many suppliers are offering devices that support ZigBee's open standard self-assembly and self-healing grid topology model.

Thread

Thread is a new IoT protocol based on Zigbee and is a method for providing radio-based internet access to low-powered devices within a relatively small area. Thread is similar very to Zigbee or Wi-Fi, but it is highly power efficient. Also, a Thread network is self-healing, meaning certain devices in the network can act as routers to take the place of a broken or failing router. Thread can connect up to 250 devices, with up to 32 of those devices being active routers in the network. The newly developed Matter protocol, which operates at the application level to support interoperability (compatibility) between different IoT devices andprotocols, often runs on top of Thread.

Z-Wave

Z-Wave is an increasingly-popular IoT protocol. It's a wireless, radio frequency cased communication technology that's primarily used for IoT home applications. It operates on the 800 to 900MHz radio frequency, while Zigbee operates on 2.4GHz, which is also a major frequency for Wi-Fi. By operating in its own range, Z-Wave rarely suffers from any significant interference problems. However, the frequency that Z-Wave devices operate on is location-dependent, so make sure you buy the right one for your country. Z-Wave is an impressive IoT protocol. However, like ZigBee, it's best used within the home and not withinthe business world.

LoRaWAN (Long Range WAN)

LoRaWAN is a media access control (MAC) IoT protocol. LoRaWAN allows low-powered devices to communicate directly with internet-connected applications over a long-range wireless connection. Moreover, it has the capability to be mapped to both the 2nd and 3rd layer of the OSI model. LoRaWAN is implemented on top of LoRa or FSK modulation for industrial, scientific, and medical (ISM) radio bands.

Communication Models in IoT

IoT devices are found everywhere and will enable circulatory intelligence in the future. For operational perception, it is important and useful to understand how various IoT devices communicate with each other. Communication models used in IoT have great value. The IoTs allow people and things to be connected any time, any space, with anything and anyone,using any network and any service.

1. Request & Response Model

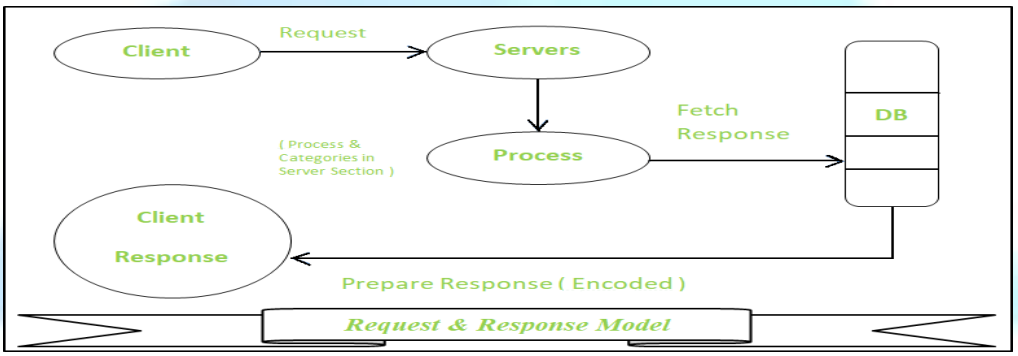


Fig.: Request and Response Model

- The client, when required, requests the information from the server. This request is usually in the encoded format.
- This model is stateless since the data between the requests is not retained and each request is independently handled.
- The server Categories the request, and fetches the data from the database and its resource representation. This data is converted to response and is transferred in an encoded format to the client. The client, in turn, receives the response.
- On the other hand — In Request-Response communication model client sends a request to the server and the server responds to the request. When the server receives the request it decides how to respond, fetches the data retrieves resources, and prepares the response, and sends it to the client.

2. Publisher-Subscriber Model

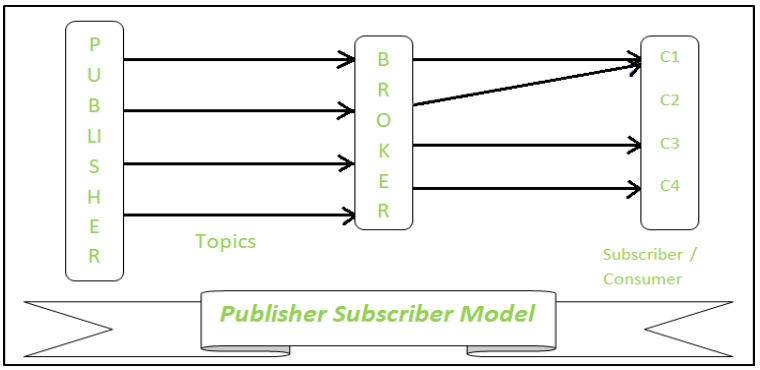


Fig.: Publisher-Subscriber Model

- Publishers are the source of data. It sends the data to the topic which are managed by the broker. They are not aware of consumers.
- Consumers subscribe to the topics which are managed by the broker.
- Hence, Brokers responsibility is to accept data from publishers and send it to the appropriate consumers. The broker only has the information regarding the consumer to which a particular topic belongs to which the publisher is unaware of.

3. Push-Pull Model

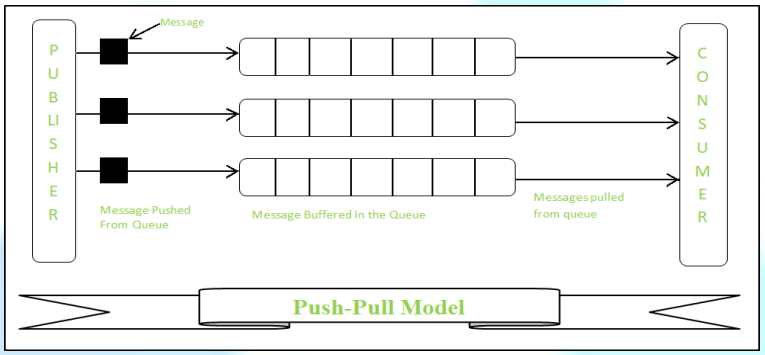


Fig.: Push-Pull Model

- Publishers and Consumers are not aware of each other.
- Publishers publish the message/data and push it into the queue. The consumers, present on the other side, pull the data out of the queue.
- Thus, the queue acts as the buffer for the message when the difference occurs in the rate of push or pull of data on the side of a publisher and consumer.
- Queues help in decoupling the messaging between the producer and consumer.
- Queues also act as a buffer which helps in situations where there is a mismatch between the rate at which the producers push the data and consumers pull the data.

4. Exclusive Pair

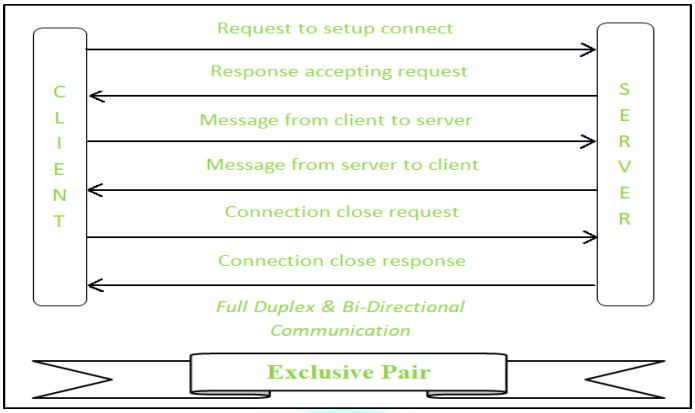


Fig.: Exclusive Pair

- Exclusive Pair is the bi-directional model, including full-duplex communication among client and server. The connection is constant and remains open till the client sends a request to close the connection.
- The Server has the record of all the connections which has been opened.
- This is a state-full connection model and the server is aware of all open connections.
- WebSocket based communication API is fully based on this model.

Application Programming Interface

An application programming interface (API) is a 'software mediator' that enables applications to communicate with one another. APIs give internal and external developers access to a secure, documented interface that they can use to leverage the functionality and data of enterprise applications.

In the last ten years, numerous business models that rely heavily on APIs have been established. This has led to the creation of an 'API economy,' a term that refers to businesses using APIs and microservices to make their services and data more scalable, accessible, and ultimately profitable. As a result, many software applications that are a mainstay in our lives today rely on APIs to function.

APIs simplify the development of new tools and the management of existing ones. They also enhance collaboration by allowing enterprises to connect the thousands of otherwise disjointed applications they use daily. Organizations also use API integration for workflow automation– a reliable way to boost productivity.

However, that's not all. In today's fast-paced market, enterprises must keep innovating to stay relevant and profitable. APIs simplify the innovation process by giving organizations the flexibility to enhance their products with solutions created by external development teams. This lets them bring new offerings to their clientele and also allows them to access new markets.

Another benefit of using APIs is the added layer of data security between servers and enterprise data. One can further fortify this layer through signatures, tokens, and transport layer security (TLS) encryption. Developers can also implement API gateways for traffic authentication and management.

How Does an API Work?

Just like a user interface (usually a graphical user interface or GUI) is built for humans to interact with applications, APIs are built for applications to interact with each other.

The application programming interface layer is responsible for data transfer and processing between a server and an application. APIs collate data from one application, format it for export, and transmit it to the destination application without compromising accuracy or security.This process is unaffected even in cases where the feature sets of the requesting app have been updated. Additionally, developers without knowledge of the backend workings of a specific API can still integrate it with their software.

Step 1: An API call (also known as a request) is placed by a client application. This call consists of headers, a request verb, and occasionally a request body.

Step 2: The API's uniform resource identifier (URI) is used to process this request for data retrieval from an application to the web server.

Step 3: Once the API receives a valid request, it places a call to the external server or application.

Step 4: The external server or application transmits the requested information back to the API. This is known as a response.

Step 5: The API sends the information to the client application that originally requested it.

Simply put, APIs delink the requesting application from the architecture that provides the requested service. While using different web services can lead to changes in the data transfer process, the entire request and response system relies on APIs for completion.

Types of APIs

Most application programming interfaces today are web-based. These APIs allow external stakeholders to access the functionality and data of an application over the internet.

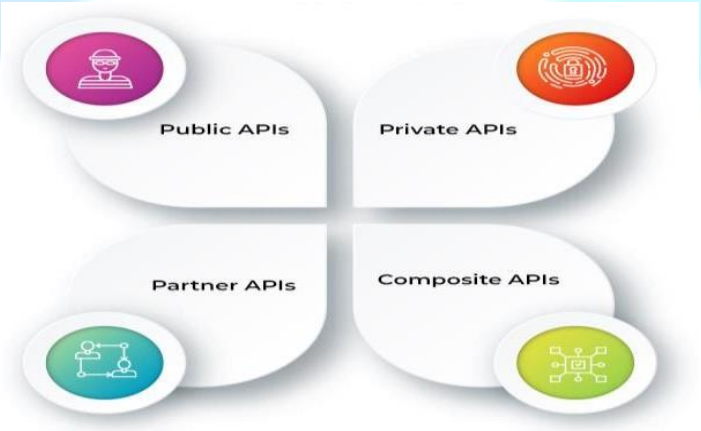


Fig.: Types of APIs

1. Public APIs

Public APIs are open source and disseminated for general use. This is why they are also referred to as open APIs. These application programming interfaces have specific API endpoints and formats for calls and responses, and they can be accessed using the HTTP protocol. Open APIs allow users to request information from any enterprise that provides the interface. This type of API is a key component of smartphone applications. It is also used to integrate popular services with websites easily. Google Maps API is an example of a popular public API.

2. Private APIs

Unlike open APIs that are accessible by the public at large, private APIs exist within a software vendor'sOpens a new window system framework. They are also known as closed orinternal APIs and are often proprietary. These interfaces aim to bolster communication and boost productivity. Enterprises leverage closed APIs to privately transmit data among internal business applications such as enterprise resource planning (ERP), financial systems, or customer relationship management (CRM). Private APIs are normally not revealed to external users.

3. Partner APIs

As the name suggests, partner APIs allow two different companies to enter into an exclusive data-sharing agreement. Using this type of application programming interface, vendors gain access to the data streams of partner companies. In return, the company granting access to its data receives added services or system features. Developers can normally access these partner interfaces in self-service mode using an open API dev portal. However, they would still be required to go through an onboarding process and enter login credentials to gain access to partner APIs. This type of API is a critical component of strategic business partnerships in the API economy.

4. Composite APIs

Composite APIs combine different service or data APIs. This variant of the application programming interface enables dev teams to access multiple endpoints by raising a single call. Composite APIs are often seen in microservices architectures, where data from more than one source is frequently needed to complete a given task.

Composite interfaces compile multiple calls sequentially and create a single API request. This request is transmitted to the server, which, in turn, sends back one response. The distinction between composite APIs and batch APIs is the lack of a sequence in the latter. Forinstance, an ecommerce platform might use a composite API to create an order by a new customer. By doing so, only a single request would need to be raised to create a new customer profile, generate an order for the new customer profile, add an item to the new order, and revise the order status.

Types of API Protocols

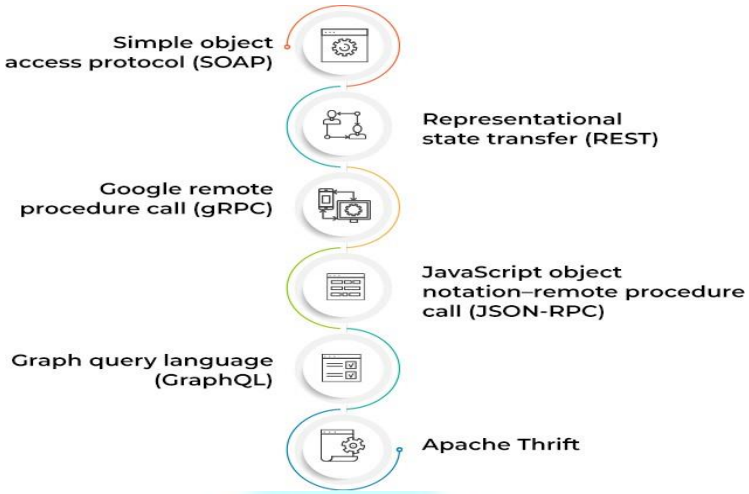


Fig.: Types of API Protocols

1. Simple object access protocol (SOAP)

The SOAP API protocol uses extensible markup language (XML) to power API communications. With its first appearance in June 1998, SOAP is the oldest protocol still in use today. This protocol primarily uses XML files to transmit data over an HTTP or HTTPS connection. However, it is also flexible enough to transmit data over other protocols such as simple mail transport protocol (SMTP), transmission control protocol (TCP), and user data protocol (UDP).

XML-encoded SOAP messages use the format defined below:

Envelope: The core element of the message. It 'envelopes' the message by placing tags at the start and the end.

Header (optional): It defines specific additional message requirements, such as authentication.

Body: The request or response is included here.

Fault (optional): Information about errors that might arise during the execution of the API call or response is highlighted here, along with information on how one can address these errors.

SOAP is popular for the flexibility of its transmission channel. However, it relies heavily on XML, thus requiring rigid formatting rules. Also, XML is difficult to debug, making SOAP less popular for modern-day API requirements.

2. Representational state transfer (REST)

REST protocols are more flexible than SOAP. This is primarily achieved by removing the dependency on XML; REST prominently transmits data in JSON. However, it can transfer data in multiple other formats, including Python, HTML, and even media files and plain text. An API that uses the REST protocol is known as a RESTful API. These APIs use a client-server structure. Unlike SOAP, REST can only use HTTP and HTTPS to communicate, making it less flexible on this front.

REST requests typically include these key components:

HTTP method: This component outlines the four basic processes that a resource can be subjected to–POST (create a resource), GET (retrieve a resource), PUT (update a resource), and DELETE (delete a resource).

Endpoint: The uniform resource identifier that locates the resource on the internet is a part of this component. URLs are the most common type of URI.

Header: Data related to the server and the client is stored in this component. Like in SOAP, one can also use REST headers to store authentication measures such as API keys, server IP addresses, and the response format.

Body: This component contains additional information for the server, such as data that needs to be added or replaced.

Compared to SOAP, REST implementation is flexible, scalable, and lightweight. A key feature of RESTful APIs is stateless communication. This forbids the storage of client data between GET requests. Caching is another key feature of REST. This enables web browsers to store the request response received locally and access it periodically to enhance efficiency.Finally, GET requests are required to be disconnected and distinct. With REST, a unique URL is assigned to every operation. This allows the server to follow preset instructions for executing a received request.

3. Google remote procedure call (gRPC)

Developed by Google and released for public use in 2015, gRPC is an open-source remote procedure call (RPC) architecture that can operate in numerous environments. The gRPC transport layer primarily relies on HTTP. The ability for developers to specify custom functions that allow for flexible inter-service communication is a significant feature of gRPC. This API protocol also offers extra features such as timeouts, authentication, and flow control. In the gRPC protocol, data is transmitted in protocol buffers, a platform and language-agnostic mechanism that allows for data to be structured intuitively. This

mechanism defines the service and then the data structures that the service will use. Compiling is taken care of by protoc, the protocol buffer compiler. The output of this process is a comprehensive class containing the user's defined data types and basic set methods in the chosen development language. Users can implement in-depth API operations using this class.

4. JavaScript object notation–remote procedure call (JSON-RPC)

JSON-RPC is a stateless and lightweight API protocol that communicates between web services using request objects and response objects. Introduced shortly after the turn of the millennium, JSON-RPC leverages JavaScript Object Notation (JSON) to allow API communications' simple, albeit limited, execution. This protocol defines requests that can take care of all functionalities within its narrow scope. JSON-RPC has the potential to outperform REST in cases where one can apply it.

5. Graph query language (GraphQL)

Released in 2015, GraphQL is a database query language, and a server-side runtime for APIs developed at Facebook. By design, this protocol prioritizes giving users the exact data requested–no less, no more. GraphQL is developer-friendly and supports the creation of fast and flexible APIs, including composite APIs. GraphQL can be used as an alternative for REST.

6. Apache Thrift

Also developed at Facebook, Thrift is a lightweight, language-agnostic software stack. This API protocol supports HTTP transmission, along with binary transport formats. Thrift is capable of clean abstractions and implementations for data serialization and transport and application-level processing. Its primary objective is point-to-point RPC implementation. Apache can support 28 programming languages, allowing programs written in any of these languages to communicate with each other and request remote services using APIs.

REST API

Representational State Transfer (REST) is an architectural style that defines a set of constraints to be used for creating web services. REST API is a way of accessing web services in a simple and flexible way without having any processing. REST technology is generally preferred to the more robust Simple Object Access Protocol (SOAP) technology because REST uses less bandwidth, simple and flexible making it more suitable for internet usage. It's used to fetch or give some information from a web service. All communication done via REST API uses only HTTP request.

Working: A request is sent from client to server in the form of a web URL as HTTP GET or POST or PUT or DELETE request. After that, a response comes back from the server in the form of a resource which can be anything like HTML, XML, Image, or JSON. But now JSON is the most popular format being used in Web Services.

Fig.: REST API Working

In HTTP there are five methods that are commonly used in a REST-based Architecture i.e., POST, GET, PUT, PATCH, and DELETE. These correspond to create, read, update, and delete (or CRUD) operations respectively. There are other methods which are less frequently used like OPTIONS and HEAD.

GET: The HTTP GET method is used to read (or retrieve) a representation of a resource. In the safe path, GET returns a representation in XML or JSON and an HTTP response code of 200 (OK). In an error case, it most often returns a 404 (NOT FOUND) or 400 (BAD REQUEST).

POST: The POST verb is most often utilized to create new resources. In particular, it's used to create subordinate resources. That is, subordinate to some other (e.g. parent) resource. On successful creation, return HTTP status 201, returning a PUT: It is used for updating the capabilities. However, PUT can also be used to create a resource in the case where the resource ID is chosen by the client instead of by the server. In other words, if the PUT is to aURI that contains the value of a non-existent resource ID. On successful update, return 200 (or 204 if not returning any content in the body) from a PUT. If using PUT for create, returnHTTP status 201 on successful creation. PUT is not safe operation but it's idempotent.

PATCH: It is used to modify capabilities. The PATCH request only needs to contain the changes to the resource, not the complete resource. This resembles PUT, but the body contains a set of instructions describing how a resource currently residing on the server should be modified to produce a new version. This means that the PATCH body should not just be a modified part of the resource, but in some kind of patch language like JSON Patch or XML Patch. PATCH is neither safe nor idempotent.

DELETE: It is used to delete a resource identified by a URI. On successful deletion, return HTTP status 200 (OK) along with a response body. Location header with a link to the newly-created resource with the 201 HTTP status.

IoT enabling Technology

1. Wireless Sensor Network
2. Cloud Computing
3. Big Data Analytics
4. Communications Protocols
5. Embedded System

1. Wireless Sensor Network (WSN):

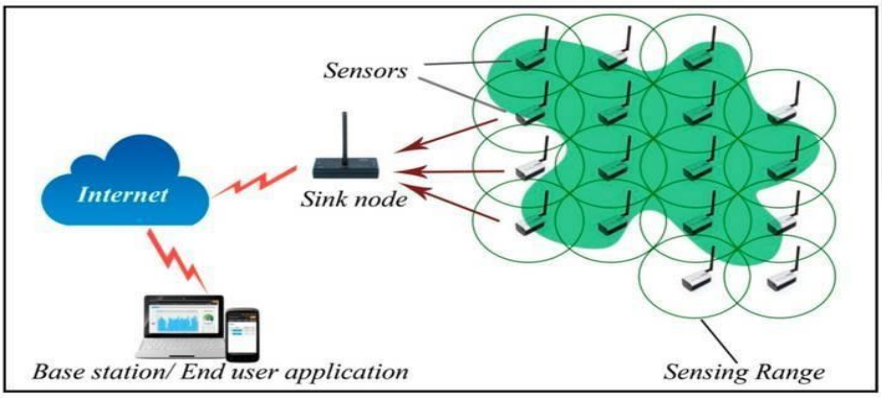


Fig.: WSN Working

A WSN comprises distributed devices with sensors which are used to monitor the environmental and physical conditions. A wireless sensor network consists of end nodes, routers and coordinators. End nodes have several sensors attached to them where the data is passed to a coordinator with the help of routers. The coordinator also acts as the gateway that connects WSN to the internet.

Example –

- Weather monitoring system
- Indoor air quality monitoring system
- Soil moisture monitoring system
- Surveillance system
- Health monitoring system

2. Cloud Computing:

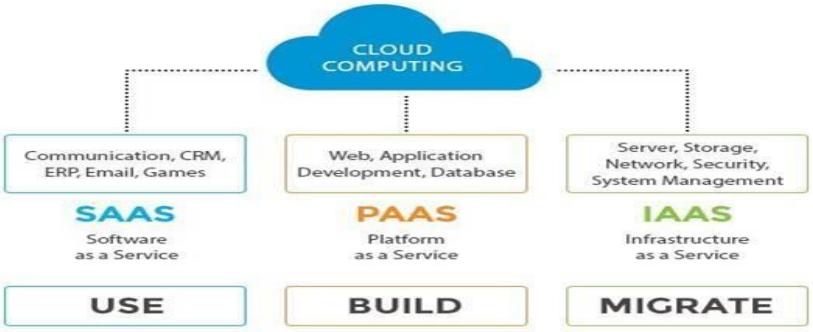


Fig.: Cloud Computing

It provides us the means by which we can access applications as utilities over the internet. Cloud means something which is present in remote locations.

With Cloud computing, users can access any resources from anywhere like databases, webservers, storage, any device, and any software over the internet.

Characteristics –

- Broad network access
- On demand self-services
- Rapid scalability
- Measured service
- Pay-per-use

Provides different services, such as –

IaaS (Infrastructure as a service)

Infrastructure as a service provides online services such as physical machines, virtual machines, servers, networking, storage and data center space on a pay per use basis. Major IaaS providers are Google Compute Engine, Amazon Web Services and Microsoft Azure etc.

Ex : Web Hosting, Virtual Machine etc.

PaaS (Platform as a service)

Provides a cloud-based environment with a very thing required to support the complete life cycle of building and delivering West web based (cloud) applications – without the cost and complexity of buying and managing underlying hardware, software provisioning and hosting. Computing platforms such as hardware, operating systems and libraries etc. Basically, it provides a platform to develop applications.

Ex : App Cloud, Google app engine

SaaS (Software as a service)

It is a way of delivering applications over the internet as a service. Instead of installing and maintaining software, you simply access it via the internet, freeing yourself from complex software and hardware management. SaaS Applications are sometimes called web-based software on demand software or hosted software. SaaS applications run on a SaaS provider's service and they manage security availability and performance.

Ex : Google Docs, Gmail, office etc.

3. Big Data Analytics:

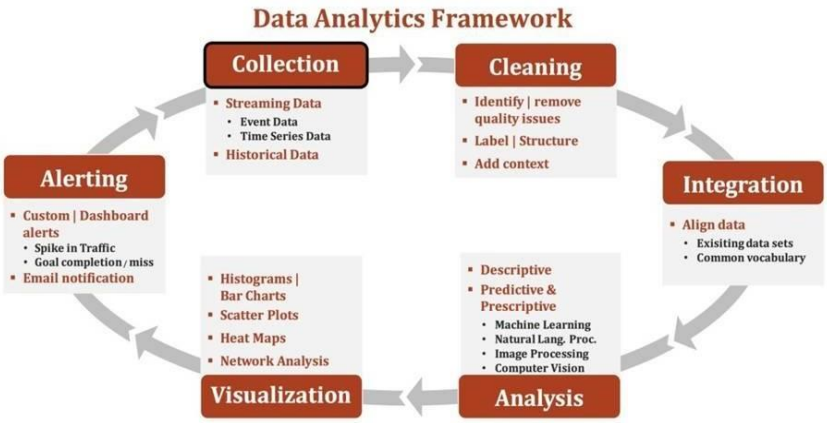


Fig.: Data Analytics Framework

It refers to the method of studying massive volumes of data or big data. Collection of data whose volume, velocity or variety is simply too massive and tough to store, control, process and examine the data using traditional databases.

Big data is gathered from a variety of sources including social network videos, digital images, sensors and sales transaction records.

Several steps involved in analysing big data –

- Data cleaning
- Munging
- Processing
- Visualization

Examples –

- Bank transactions
- Data generated by IoT systems for location and tracking of vehicles
- E-commerce and in Big-Basket
- Health and fitness data generated by IoT system such as a fitness band

4. Communications Protocols:

They are the backbone of IoT systems and enable network connectivity and linking to applications. Communication protocols allow devices to exchange data over the network. Multiple protocols often describe different aspects of a single communication. A group of protocols designed to work together is known as a protocol suite; when implemented in software they are a protocol stack.

They are used in -

- Data encoding
- Addressing schemes

5. Embedded Systems:

It is a combination of hardware and software used to perform special tasks. It includes microcontroller and microprocessor memory, networking units (Ethernet Wi-Fi adapters), input output units (display keyword etc.) and storage devices (flash memory). It collects the data and sends it to the internet.

Examples –

- Digital camera
- DVD player, music player
- Industrial robots
- Wireless Routers etc.

User Interface

User Interface (UI) defines the way humans interact with the information systems. In Layman's term, User Interface (UI) is a series of pages, screens, buttons, forms and other visual elements that are used to interact with the device. Every app and every website has a user interface. User Interface (UI) Design is the creation of graphics, illustrations, and use of photographic artwork and typography to enhance the display and layout of a digital product within its various device views. Interface elements consist of input controls (buttons, drop-down menus, data fields), navigational components (search fields, slider, icons, tags), informational components (progress bars, notifications, message boxes).

User Interface Design for Mobile Apps

The mobile user interface or mobile app interface design is what you see when you use an application. Mobile app UI consists of the visual representation, interface, navigation, and processes that work behind the functional and structural actions in the mobile app. The importance of mobile app interface design and user experience (UX) go hand in hand. UX is the holistic appearance and function of the app, while UI is all of the interaction in the application. In short, an app that is appealing and most feasible to *users* is an outcome of efficient user experience and user interface design techniques involved in building an app.

User interface design for web app

Like a website, a web app is accessed by going to a certain url
(e.g. https://examplewebapp.com). However, while websites are largely informational (like Wikipedia), web apps are built to have certain functionalities (like controlling a device remotely). The advantage of web apps is that they can work on both iOS and Android because you're just using a web browser instead of actually downloading something. Also, because you don't need to download anything, it can make it easier to get into the hands of users (just send them the url link). The disadvantage is that you have somewhat limited access to the phone's full capabilities (like the inability to send push notifications) and less control over the overall user experience.

User interface design for hybrid app

As the name implies, hybrid apps are between Mobile apps and web apps. You still download something, like a web app, but when you open the app it is essentially opening a web page meaning that it can act like a web app. This can be a good option if you know you'll be creating Mobile apps eventually, but you want to get a minimum viable product into the hands of users early, and can therefore benefit from the speedier development offered by webapps.

User interface design for desktop app

A desktop application is a software program that can be run on a standalone computer to perform a specific task by an end-user. Some desktop applications such as word editor, photo editing app and media player allow you to perform different tasks while other such as gaming apps are developed purely for entertainment. When you purchase a computer or a laptop, there is a set of apps that are already installed on your desktop. You can also download and install different desktop applications directly from the Internet or purchased from software vendors. The examples of some of the popular desktop applications are, word processing applications such as Microsoft Word and WPS Office which are designed to edit the textual content, gaming applications such as Minesweeper and Solitaire which are used for entertainment, web browsers such as Internet Explorer, Chrome and Firefox which help you to connect to the Internet from your computer, media player applications such as iTunes, Windows Media Player, and VLC media player which let you listen to music, watch videos and movies and create collections of media content.

IoT System Architecture and Operation with REST API

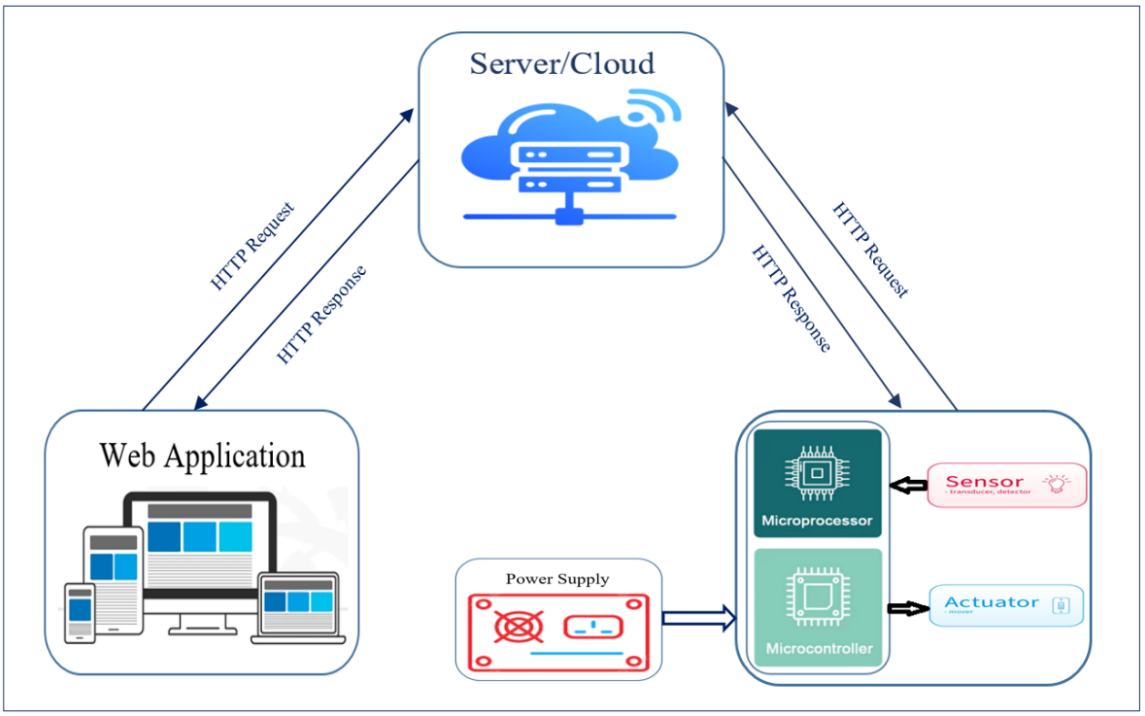


Fig.: IoT System Architecture and Operation with REST API

This diagram illustrates a basic architecture for an Internet of Things (IoT) system, enabling seamless integration of physical devices with cloud services and web applications for real-time monitoring and control from anywhere with internet access. REST (Representational State Transfer) API methods are used to facilitate communication between the different components of the system. These methods typically include HTTP requests such as GET, POST, PUT, and DELETE to perform various operations like retrieving data, sending data, updating resources, and deleting resources. The architecture, facilitated by REST API methods, supports efficient data collection, processing, and interaction, allowing users to control and monitor their IoT devices seamlessly from anywhere in the world.

It consists of following components

1. **Sensor:** Devices that detect and measure physical properties (e.g., temperature, light, motion) and convert them into data that can be read by a microcontroller or microprocessor.
2. **Actuator:** Devices that receive signals from the microcontroller or microprocessor to perform a physical action (e.g., turning on a light, opening a valve).
3. **Microcontroller:** A compact integrated circuit designed to govern a specific operation in an embedded system.
4. **Microprocessor:** A more complex device capable of handling multiple tasks, typically found in general computing systems.
5. **Power Supply:** Provides the necessary electrical power to the sensors, actuators, microcontrollers, and microprocessors.
6. **Server/Cloud:** A centralized system that processes, stores, and manages data from various devices. It also facilitates communication between the web application and the devices.
7. **Web Application:** A user interface accessible via various devices (e.g., smartphones, tablets, computers) that allows users to monitor and control the IoT system.

Data Flow

1. **Sensors gather data** from the environment (e.g., temperature, humidity) and send this data to the microcontroller/microprocessor.
2. **Microcontroller/Microprocessor processes the data** received from the sensors. This might involve filtering, aggregating, or analyzing the data.
3. **Data is sent to the Server/Cloud** from the microcontroller/microprocessor via HTTP requests. This data transfer might include sensor readings, status updates, or other relevant information.
4. **Server/Cloud processes and stores the data.** It might perform additional analysis, log the data for historical reference, or make decisions based on pre-defined rules.
5. **Web Application makes HTTP requests** to the Server/Cloud to retrieve data or send commands. Users interact with the web application to monitor system status, view sensor data, and send control commands to actuators.
6. **Server/Cloud sends HTTP responses** back to the web application, providing the requested data or acknowledgment of received commands.
7. **Web Application displays data** and provides controls to the user, allowing for real-time monitoring and interaction with the IoT system.
8. **Server/Cloud sends HTTP responses** back to the microcontroller/microprocessor, potentially with new instructions or configurations.
9. **Microcontroller/Microprocessor sends commands** to the actuators based on the received instructions, thereby affecting the physical environment (e.g., turning on a light, adjusting a thermostat).

Frontend Programming

Frontend programming involves creating the user interface and user experience elements of a website or web application. This includes everything that users interact with directly in their web browser, such as layout, design, and interactivity. The primary technologies used in frontend development are HTML, CSS, and JavaScript, with frameworks and libraries like Bootstrap enhancing these core technologies.

In the context of embedded full-stack IoT (Internet of Things) development, frontend programming plays a crucial role in creating user interfaces that allow users to interact with IoT devices. These interfaces can range from simple web dashboards to complex web applications that monitor and control IoT devices. Key technologies used include HTML, CSS, JavaScript, and various frameworks and libraries that facilitate the development of responsive, user-friendly interfaces.

In the realm of embedded full-stack IoT applications, creating an effective and engaging user interface is crucial for monitoring and controlling IoT devices. The frontend development process leverages a combination of HTML, CSS, JavaScript, and Bootstrap to achieve this goal.

- **HTML (HyperText Markup Language)** provides the foundational structure of the web page, defining the layout and elements such as headings, paragraphs, buttons, forms, and data displays that users interact with.
- **CSS (Cascading Style Sheets)** is used to style these HTML elements, ensuring that the web page is visually appealing and user-friendly. CSS also handles responsive design, making sure the interface works seamlessly across different devices, from desktops to mobile phones.
- **JavaScript** adds interactivity and dynamic functionality to the web page. It allows for real-time updates, user input handling, and communication with backend systems through APIs or WebSockets, essential for the dynamic control and monitoring of IoT devices.
- **Bootstrap** is a popular front-end framework that offers pre-designed components and a robust grid system. It streamlines the development process, enabling developers to create modern, responsive web pages quickly and efficiently.

By combining these technologies, developers can create sophisticated, interactive, and responsive user interfaces that enhance the usability and effectiveness of IoT applications. This integration ensures that the web pages not only look good but also provide seamless communication with backend systems, enabling efficient management and control of IoT devices.

Backend Programming

Backend programming is a critical aspect of web and application development, focusing on the server-side components that power the front-end interface. This involves creating and managing the server, database, and application logic that ensure smooth and efficient operation of web applications. Backend programming handles data storage, retrieval, and manipulation, authentication, authorization, and business logic implementation. Key technologies used in backend development include server-side languages, databases, and various frameworks and libraries.In the context of embedded full-stack IoT (Internet of Things) development, backend programming is essential for managing and processing data from IoT devices, ensuring secure and reliable communication, and providing the necessary logic for IoT applications to function correctly. Backend developers work with languages such as Python, Node.js, Java, and frameworks like Django, Express, and Spring Boot, as well as databases like MySQL, MongoDB, and PostgreSQL.

Key Components of Backend Programming

1. **Server-Side Languages**
   - **Python:** Known for its simplicity and readability, Python is widely used in backend development, especially with frameworks like Django and Flask, which facilitate rapid development and clean, pragmatic design.
   - **Node.js:** A JavaScript runtime built on Chrome's V8 engine, Node.js is popular for building scalable network applications. It is event-driven and non-blocking, making it ideal for real-time applications.
   - **Java:** A robust and versatile language, Java is used with frameworks such as Spring Boot to create high-performance, secure, and scalable backend services.
2. **Databases**
   - **MySQL:** A relational database management system (RDBMS) that uses SQL for data manipulation. It is known for its reliability and ease of use.
   - **MongoDB:** A NoSQL database that stores data in flexible, JSON-like documents. It is designed for handling large volumes of unstructured data.
   - **PostgreSQL:** An advanced open-source RDBMS known for its extensibility and standards compliance, offering robust data integrity and powerful SQL support.
3. **Frameworks and Libraries**
   - **Django (Python):** A high-level Python web framework that encourages rapid development and clean, pragmatic design. It comes with a built-in admin panel, ORM, and authentication system.
   - **Express (Node.js):** A minimal and flexible Node.js web application framework that provides a robust set of features for building single and multi-page web applications.
   - **Spring Boot (Java):** A framework that simplifies the development of new Spring applications. It provides a comprehensive infrastructure for developing Java applications with a microservices architecture.

Functions of Backend Programming in IoT

1. **Data Management:**
    - **Storage:** Efficiently storing large volumes of data generated by IoT devices in databases.
    - **Retrieval:** Providing fast and reliable data retrieval mechanisms to serve the frontend and other services.
    - **Processing:** Performing complex data processing and analysis to generate actionable insights.
2. **Authentication and Authorization:**
    - **User Management:** Implementing secure user authentication and role-based access control to ensure that only authorized users can access and control IoT devices.
    - **Security Protocols:** Using industry-standard security protocols to protect data integrity and privacy during transmission and storage.
3. **API Management:**
    - **RESTful APIs:** Designing and implementing RESTful APIs that allow the frontend to communicate with the backend. These APIs handle requests such as data retrieval, updates, and device control.
    - **WebSockets:** Enabling real-time communication between the frontend and backend through WebSockets, essential for real-time monitoring and control of IoT devices.
4. **Business Logic:**
    - **Rule Engine:** Implementing business rules and logic that determine how data is processed and actions are taken based on the data from IoT devices.
    - **Event Handling:** Managing events and triggers based on sensor data to automate actions and alerts in the IoT ecosystem.

HTML

HTML (HyperText Markup Language) is the most widely used language for creating webpages and serves as the fundamental building block of the Web. It used to structure and organize the content on a web page. It uses various tags to define the different elements on a page, such as headings, paragraphs, and links.

Introduction to HTML

HTML stands for HyperText Markup Language. It is a markup language, not a programming language, which means it is used to "mark up" a text document with tags that tell a web browser how to structure and display the content.

Key Concepts of HTML

HyperText : HyperText refers to the way web pages (HTML documents) are linked together. The links available on a webpage are called Hypertext. These links allow users to navigate to different parts of the same web page or to different web pages altogether.

Markup Language : A markup language is a computer language used to add structure and formatting to a text document. Markup languages use a system of tags to define the structure and content of a document. These tags are interpreted by web browsers to display the document in a specific way.

HTML Features

HTML is a text-based language with several powerful features that make it essential for creating web pages:

- **Standard Language:** HTML is the standard language used for creating and structuring web pages. It organizes content using elements such as headings, paragraphs, lists, and tables.
- **Media Support:** HTML supports a wide range of media types, including text, images, audio, and video, making web pages more engaging and interactive.
- **Flexibility:** HTML can be used with other technologies, such as CSS (Cascading Style Sheets) and JavaScript, to add additional features and functionality to a web page.
- **Cross-Browser Compatibility:** HTML is compatible with all web browsers, ensuring that web pages are displayed consistently across a variety of platforms and devices.
- **Open and Standardized:** HTML is an open and standardized language, continuously updated and improved by a community of developers and experts.

HTML provides the essential framework for building web pages, offering a structured way to present content and integrate multimedia. By understanding and utilizing HTML, developers can create robust and accessible web pages that are the cornerstone of modern web development.

HTML History

HTML has evolved significantly since the early days of the World Wide Web. Here are key milestones in its development:

Year	Version
1989	Tim Berners-Lee invented WWW
1991	Tim Berners-Lee invented HTML
1993	Dave Raggett drafted HTML+
1995	HTML Working Group defined HTML 2.0
1997	W3C Recommendation: HTML 3.2
1999	W3C Recommendation: HTML 4.01
2000	W3C Recommendation: XHTML 1.0
2008	WHATWG HTML5 First Public Draft
2012	WHATWG HTML5 Living Standard
2014	W3C Recommendation: HTML5
2016	W3C Candidate Recommendation: HTML 5.1
2017	W3C Recommendation: HTML5.1 2nd Edition
2017	W3C Recommendation: HTML5.2

Fig.: Evolution of HTML

Basic Structure of HTML Document

```
<!DOCTYPE html>
<html>
<head>
<title>This is document title</title>
</head>
<body>
<h1>This is a heading</h1>
<p>Document content goes here.....</p>
</body>
</html>
```

To check the result of this HTML code, or let's save it in an HTML file test.htm using your favourite text editor. Finally open it using a web browser like Internet Explorer or Google Chrome, or Firefox etc. It must show the following output:

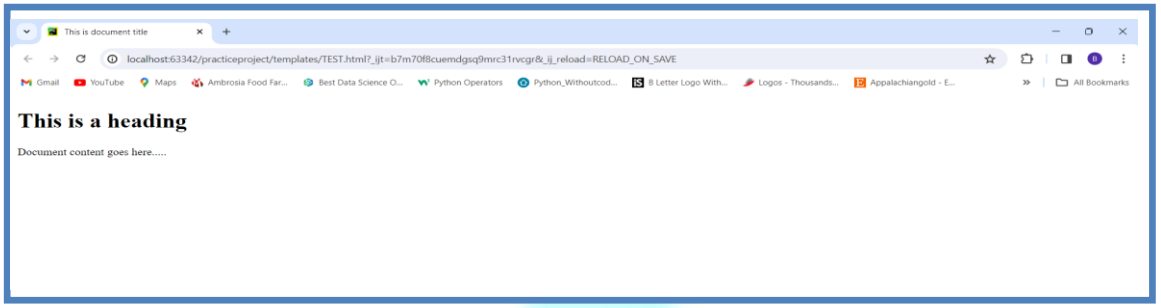


Fig.: Browser Output

HTML elements are hierarchical, which means that they can be nested inside each other to create a tree-like structure of the content on the web page.

This hierarchical structure is called the DOM (Document Object Model), and it is used by the web browser to render the web page.

In this example, the html element is the root element of the hierarchy and contains two child elements: head and body. The head element, in turn, contains a child element called the title, and the body element contains child elements: h1 and p.

HTML Tags

HTML is a markup language and makes use of various tags to format the content. These tags are enclosed within angle braces **<Tag Name>**. Except few tags, most of the tags have their corresponding closing tags. For example, **<html>** has its closing tag**</html>** and **<body>** tag has its closing tag **</body>** tag etc.

In above example of HTML document uses the following tags

- **<!DOCTYPE html>:** This declaration defines the document type and version of HTML being used. In this case, it specifies that the document is an HTML5 document.
- **<html>:** This is the root element of an HTML document. All other HTML elements are contained within this tag.
- **<head>:** The head element contains meta-information about the HTML document, such as its title, character set, styles, scripts, and other meta information.
- **<title>This is document title</title>:** The title element sets the title of the HTML document, which is displayed in the browser's title bar or tab. In this case, the title of the document is "This is document title".
- **<body>:** The body element contains the content of the HTML document that is visible to users. All visible HTML elements such as text, images, links, tables, etc., are placed inside the body element.

- **<h1>This is a heading</h1>:** The h1 element defines a top-level heading. Headings are used to define sections of content. In this case, the heading text is "This is a heading".
- ** Document content goes here..... :** The p element defines a paragraph. Paragraphs are used to group blocks of text. In this case, the paragraph text is "Document content goes here.....".

HTML Document Structure

```
Document declaration tag
<html>
<head>
Document header related tags
</head>
<body>
Document body related tags
</body>
</html>
```

The <!DOCTYPE> Declaration

The <!DOCTYPE> declaration tag is used by the web browser to understand the version of the HTML used in the document. Current version of HTML is 5 and it makes use of the following declaration:

```
<! DOCTYPE html>
```

There are many other declaration types which can be used in HTML document depending on what version of HTML is being used. We will see more details on this while discussing <!DOCTYPE...> tag along with other HTML tags.

HTML Elements

HTML elements consist of several parts, including the opening and closing tags, the content, and the attributes.

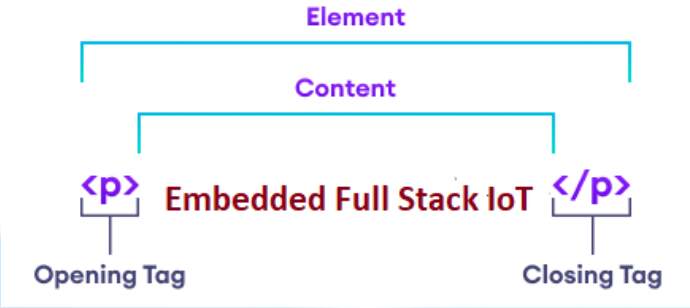


Fig.:HTML elements

Here,

- **The opening tag:** This consists of the element name, wrapped in angle brackets. It indicates the start of the element and the point at which the element's effects begin.
- **The closing tag:** This is the same as the opening tag, but with a forward slash before the element name. It indicates the end of the element and the point at which the element's effects stop.
- **The content:** This is the content of the element, which can be text, other elements, or a combination of both.
- **The element:** The opening tag, the closing tag, and the content together make up the element.

HTML (HyperText Markup Language) provides a variety of elements to structure and present content on the web. Here's a list of some common HTML elements:

1. **Text Formatting:**
    - <h1> to <h6>: Headings (from highest importance to lowest).
    - <p>: Paragraph.
    - : Strong emphasis.
    - : Emphasis.
    - : Generic inline container.
    -
: Line break.
    - <hr>: Horizontal rule.
2. **Lists:**
    - : Unordered list.
    - : Ordered list.
    - : List item.
    - <dl>: Description list.
    - <dt>: Description term.
    - <dd>: Description details.
3. **Links and Anchors:**
    - <a>: Anchor.
    - <link>: External resource link (typically used for stylesheets).
    - <nav>: Navigation links container.
4. **Images and Multimedia:**
    - <img>: Image.
    - <audio>: Audio player.
    - <video>: Video player.
5. **Tables:**
    - <table>: Table container.
    - <tr>: Table row.
    - <th>: Table header cell.
    - <td>: Table data cell.
    - <caption>: Table caption.

6. **Forms:**

    - <form>: Form container.
    - <input>: Input control.
    - <textarea>: Text input area.
    - <select>: Dropdown selection.
    - <button>: Button.
    - <label>: Form field label.
    - <fieldset>: Group of form controls.
    - <legend>: Title for <fieldset>.

7. **Sections and Grouping:**

    - : Generic container.
    - <section>: Section of a document.
    - <header>: Header section.
    - <footer>: Footer section.
    - <main>: Main content area.
    - <article>: Article or independent content.
    - <aside>: Sidebar content.
    - <nav>: Navigation links.

8. **Semantic Elements:**

    - <header>: Defines a header for a document or a section.
    - <footer>: Defines a footer for a document or a section.
    - <main>: Defines the main content of a document.
    - <article>: Defines an article.
    - <section>: Defines a section in a document.
    - <aside>: Defines content aside from the content (like a sidebar).
    - <details>: Defines additional details that the user can view or hide.
    - <summary>: Defines a heading for the <details> element.

9. **Meta Information:**

    - <meta>: Metadata that provides information about the HTML document.
    - <title>: Title of the document (displayed in the browser's title bar).

Self Closing Tags

Self-closing tags, also known as void elements or empty elements, are HTML elements that do not require closing tags because they don't contain any content. Instead, they self-terminate with a trailing slash (/) immediately before the closing angle bracket (>).

Here's an explanation of self-closing tags with some examples:

1. **Image Tag (<img>):**
    - The <img> tag is used to embed images in an HTML document.
    - It doesn't contain any content, so it's self-closing.
    - Example:
2. **Line Break Tag (
):**
    - The
 tag inserts a single line break.
    - It doesn't require a closing tag since it doesn't contain any content.
    - Example: First Line
Second Line
3. **Horizontal Rule Tag (<hr>):**
    - The <hr> tag inserts a horizontal rule to separate content.
    - Like
, it doesn't contain content and is self-closing.
    - Example: <hr>
4. **Input Tag (<input>):**
    - The <input> tag creates form controls like text fields, checkboxes, radio buttons, etc.
    - It doesn't contain content and is self-closing.
    - Example: <input type="text" name="username" />
5. **Meta Tag (<meta>):**
    - The <meta> tag provides metadata about the HTML document.
    - It's self-closing and commonly used to specify character encoding, viewport settings, etc.
    - Example: <meta charset="UTF-8" />
6. **Embedded Content Tags (<audio>, <video>, <source>):**
    - Tags like <audio>, <video>, and <source> for media embedding are also self-closing.
    - Example: <audio controls><source src="audio.mp3" /></audio>

Self-closing tags make HTML cleaner and more concise, especially for elements that don't contain content. They're essential for adhering to HTML standards and ensuring compatibility across different browsers and platforms.

Classifications of HTML Elements

In HTML, elements are categorized as either inline or block elements. Understanding the difference between these types is essential for effective web design and layout.

Block Elements

Block elements are typically used to create the structure of a webpage. They start on a new line and take up the full width available (stretching out to the left and right as far as they can).

Characteristics of Block Elements:

- Always start on a new line.
- Take up the full width available by default.
- Can contain other block elements and inline elements.
- Examples include: , <p>, <h1> through <h6>, <ul>, <ol>, <li>, <header>, <footer>, <section>, <article>, and more.

Inline Elements

Inline elements do not start on a new line and only take up as much width as necessary. They are typically used for smaller pieces of content within block elements.

Characteristics of Inline Elements:

- Do not start on a new line.
- Only take up as much width as necessary.
- Cannot contain block elements but can contain other inline elements and text.
- Examples include: , <a>, <img>, , ,
, <i>, <b>, and more.

Differences Between Block and Inline Elements

- **Block Elements**: Take up the full width available and start on a new line. They can contain both block and inline elements.
- **Inline Elements**: Take up only as much width as necessary and do not start on a new line. They can contain other inline elements and text but not block elements.

HTML Attributes

HTML elements can have attributes, which provide additional information about the element. They are specified in the opening tag of the element and take the form of name-value pairs.

Example:

```
<a href="http://example.com"> Example </a>
```

Here,

The href is an attribute. It provides the link information about the <a> tag. In the above example,

- href - the name of attribute
- https://www.programiz.com - the value of attribute

HTML attributes are mostly optional.

The various HTML attributes along with their descriptions

Attribute	Description	Example
Global Attributes		
id	Specifies a unique id for an HTML element.	Header Section
class	Specifies one or more class names for an element (used for CSS and JavaScript).	This is an introductory paragraph.
style	Specifies inline CSS styles for an element.	Blue Heading
title	Provides extra information about an element (displayed as a tooltip).	<abbr title="HyperText Markup Language">HTML</abbr>
data-*	Custom data attributes used to store private data.	User Data
Event Attributes		
onclick	Script to be run when an element is clicked.	<button onclick="alert('Button clicked!')">Click Me</button>
onmouseover	Script to be run when the mouse pointer is moved over an element.	Hover over this text.
onmouseout	Script to be run when the mouse pointer is moved out of an element.	Move the mouse away from this text.

Attribute	Description	Example
Anchor (a) Attributes		
href	Specifies the URL of the page the link goes to.	Visit Example
target	Specifies where to open the linked document.	Open in New Tab
Image (img) Attributes		
src	Specifies the path to the image.	<img src="image.jpg" alt="Sample Image">
alt	Provides alternative text for an image if it cannot be displayed.	<img src="image.jpg" alt="A beautiful scenery">
width	Specifies the width of the image.	<img src="image.jpg" alt="Sample Image" width="500">
height	Specifies the height of the image.	<img src="image.jpg" alt="Sample Image" height="600">
Input (input) Attributes		
type	Specifies the type of input element.	<input type="text" placeholder="Enter your name">
placeholder	Provides a short hint that describes the expected value of the input field.	<input type="text" placeholder="Enter your name">
value	Specifies the initial value of the input field.	<input type="text" value="John Doe">
name	Specifies the name of the input element.	<input type="text" name="username">
required	Specifies that an input field must be filled out before submitting the form.	<input type="email" required>
Form (form) Attributes		
action	Specifies where to send the form-data when a form is submitted.	<form action="/submit-form" method="post">
method	Specifies the HTTP method to be used when sending form-data.	<form action="/submit-form" method="post">

Attribute	Description	Example
Table (table) Attributes		
border	Specifies whether the table cells should have borders.	<table border="1"> <tr> <th>Header</th> <th>Header</th> </tr> <tr> <td>Data</td> <td>Data</td> </tr> </table>

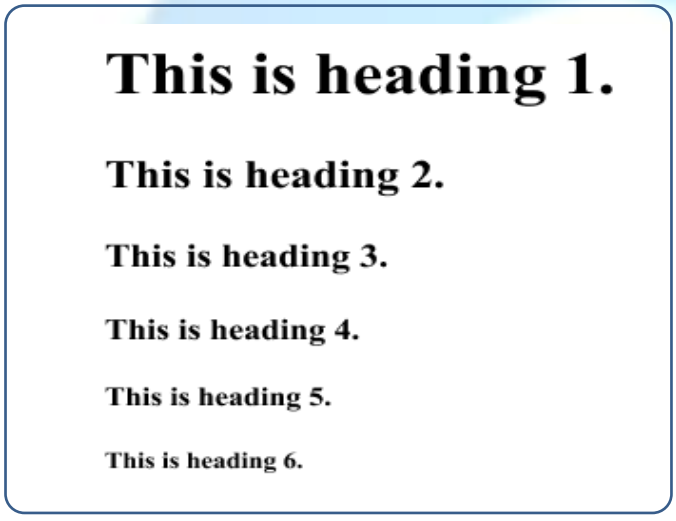
HTML Headings

The HTML heading tags (<h1> to <h6>) are used to add headings to a webpage.

For example,

```
<h1>This is heading 1.</h1>

<h2>This is heading 2.</h2>

<h3>This is heading 3.</h3>

<h4>This is heading 4.</h4>

<h5>This is heading 5.</h5>

<h6>This is heading 6.</h6>
```

The browser output is



In the example, we have used tags <h1> to <h6> to create headings of varying sizes and importance.

The <h1> tag denotes the most important heading on a webpage. Similarly, <h6> denotes the least important heading.

The difference in sizes of heading tags comes from the browser's default styling. And, you can always change the styling of heading tags, including font size, using CSS.

Note: Do not use heading tags to create large texts. It's because search engines like Google use heading tags to understand what a page is about.

h1 Tag is Important

The HTML <h1> tag defines the most important heading in a webpage.

Although it's possible to include multiple <h1> tags in a webpage, the general practice is to use a single <h1> tag (usually at the beginning of the page).

The <h1> tag is also important for SEO. Search engines (such as Google) treat content inside <h1> with greater importance compared to other tags.

Headings are block-level elements

The Headings <h1> to <h6> tags are block-level elements. They start on a new line and take up the full width of their parent element.

For example,

```
<h1>Headings</h1> <h2>are</h2> <h3>fun!</h3>
```

The browser output

HTML Paragraphs

The HTML tag is used to create paragraphs.

For example,

```
<p>HTML is fun to learn.</p>
```

Browser Output

HTML is fun to learn.

A paragraph starts with the and ends with the tag. HTML paragraphs always start on a new line.

```
<p>Paragraph 1: Short Paragraph</p>

<p>Paragraph 2: Long Paragraph, this is a long paragraph with more text to fill an entire line in the website.</p>
```

Browser Output

Paragraph 1: Short Paragraph
Paragraph 2: Long Paragraph, this is a long paragraph with more text to fill an entire line in the website.

HTML Paragraphs and Spaces

Paragraphs automatically remove extra spaces and lines from our text.

For example,

```
<p>
The paragraph tag removes      all extra spaces.


The paragraph tag also removes all extra lines.
</p>
```

Browser Output

The paragraph tag removes all extra spaces. The paragraph tag also removes all extra lines.

Here, the output

- remove all the extra spaces between words
- remove extra lines between sentences

Adding Line Breaks in Paragraphs

We can use the HTML line break tag,
, to add line breaks within our paragraphs.

For example,

```
<p>We can use the <br> HTML br tag <br> to add a line break.</p>
```

Browser Output

We can use the
HTML br tag
to add a line break without creating a new paragraph

Note: The
 tag does not need a closing tag like the <p> tag.

Pre-formatted Text in HTML

The paragraphs cannot preserve extra lines and space. If we need to create content that uses multiple spaces and lines, we can use the HTML <pre> tag.

The <pre> tag creates preformatted text. Preformatted texts are displayed as written in the HTML file. For example,

```
<pre>
 This text
  will be
   displayed


   in the same manner
    as it is written
</pre>
```

Browser Output

```
This text
  will be
    displayed

  in the same manner
    as it is written
```

Other Elements Inside Paragraphs

It's possible to include one HTML element inside another. This is also true for <p> tags. For example,

```
<p>
We can use other tags like <strong>the strong tag to emphasize text</strong>
</p>
```

Browser Output

We can use other tags like **the strong tag to emphasize text**

In the above example, we have used the tag inside the <p> tag. Browsers automatically make the contents inside tags bold.

Paragraph is Block-level

The <p> tag is a block-level element. It starts on a new line and takes up the full width (of its parent element).

Add Extra Space Inside Paragraphs

As discussed earlier, we cannot normally add extra empty spaces inside paragraphs. However, we can use a certain HTML entity called non-breaking space to add extra spaces.

For example,

```
<p>Extra space    inside a paragraph</p>
```

Browser Output

Extra space inside a paragraph

Here,   is an HTML entity, which is interpreted as a space by browsers. This allows us to create multiple spaces inside paragraphs and other HTML tags.