



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.

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	8
Microcontrollers	13
Microprocessor	15
Difference Between Microprocessor and Microcontroller	18
Application of Microcontrollers	19
Application of Microprocessors	19
Sensors	20
Actuator	24
Introduction to IoT	26
Four-Layer Architecture of IoT	45
OSI Model	47
IoT Gateway	52
IoT Protocols	60
Communication Models in IoT	65
Application Programming Interface	67
REST API	72
IoT enabling Technology	73
User Interface	76
Frontend Programming	79
Backend Programming	80
HTML	83
Introduction to HTML	83
Basic Structure of HTML Document	84
HTML Tags	85
HTML Elements	87
HTML Attributes	92
HTML Headings	94
HTML Paragraphs	96
HTML Formatting Elements	100
HTML Quotations	104
HTML Colors	107
HTML Images	110
HTML Link- Hyperlinks	112
HTML Tables	118
HTML Lists	122
Inline and Block Elements in HTML	133

Class and ID in HTML	134
HTML Iframes	137
CSS	145
Introduction to CSS	145
CSS Syntax.....	146
CSS Selectors	147
Inline CSS	163
Internal CSS	163
External CSS	165
CSS Background	168
CSS Borders	178
CSS Margin	185
CSS Padding.....	190
CSS Text	194
CSS Fonts	198
CSS Icons	205
CSS Colors	212
CSS Opacity	213
Bootstrap-5 Framework.....	216
Introduction to Bootstrap.....	216
Advantages of Using Bootstrap.....	218
Fig.:Advantages of Using Bootstrap	218
Responsive containers	221
Bootstrap Grid System	223
Fig.:Bootstrap Grid System.....	223
Typography	229
Bootstrap Jumbotron	235
Bootstrap Colors.....	239
Bootstrap Buttons.....	249
Bootstrap Framework Inputs	254
Bootstrap Forms	255
Bootstrap Tables.....	264
Bootstrap Navigation Bar.....	266
PYTHON PROGRAMMING.....	276
Introduction to Python.....	276
Python Syntax.....	279
Python Variables	282
Python - Data Types	287
Python String Data Type	289
Python List Data Type.....	290

Python Dictionaries Data Type	301
Python – Operators.....	303
Python Condition Statements	308
Python-Loops	315
Python Functions.....	321
File Handling in Python	352
Python3 and PyCharm IDE Installation	353
PIP and Python Libraries Installation.....	354
SQL Truncate Command.....	403
EMBEDDED C PROGRAMMING	407
Introduction	407
Embedded C Syntax	409
Preprocessor Directives	411
Global Variables.....	411
Infinite Loops	412
Function In C.....	422
Function call	423
Function Definitions.....	425
Local Variables.....	425
Arduino IDE.....	428
Introduction	428
Project Creation.....	429
Board and Library Installation	430
Compiling and Uploading	430
Arduino Uno: Pin Out, Pin Configuration, and Specifications	430
ESP8266: Pin Out, Pin Configuration, and Specifications.....	432
Arduino Programming.....	433
Installation	457
HARDWARE PROGRAMMING (ESP8266)	457
ESP8266 INTERFACE WITH IR SENSOR	457
ESP8266 INTERFACE WITH ULTRASONIC SENSOR (HC-SR04).....	459
ESP8266 INTERFACE WITH PHOTO SENSOR	461
ESP8266 INTERFACE WITH MAGNETIC SENSORS (FLOATING & DOOR)	462
ESP8266 INTERFACE WITH RELAY MODULE	464
ESP8266 INTERFACE WITH TEMPERATURE & HUMIDITY SENSOR (DHT11)	466
ESP8266 INTERFACE WITH LCD & I2C MODULE.....	469
ESP8266 INTERFACE WITH BLUETOOTH MODULE & ANDROID APP	471
ESP8266 INTERFACE WITH GSM MODULE	473
ESP8266 INTERFACE WITH RFID (MFRC522).....	476

ARM Controller and Raspberry Pi.....	481
Introduction to ARM Controller.....	481
Introduction to Raspberry Pi	481
Block Diagram and Specifications of Raspberry Pi	482
Raspbian Operating System Installation and Configuration	483
Terminal Commands	484
Introduction to Thonny Python IDE.....	484
Project Creation and Python Library Installation	485
Programming with Python on Raspberry Pi.....	485
Raspberry Pi	486
Architecture of Raspberry Pi.....	488
HARDWARE PROGRAMMING (RASPBERRY PI).....	511
RASPBERRY PI INTERFACE WITH ULTRASONIC SENSOR (HC-SR04).....	511
RASPBERRY PI INTERFACE WITH DHT11.....	513
RASPBERRY PI INTERFACE WITH I2C & LCD MODULE.....	515
RASPBERRY PI INTERFACE WITH RELAY MODULE	518
RASPBERRY PI INTERFACE WITH OLED DISPLAY	520
RASPBERRY PI INTERFACE WITH ACCELEROMETER	522
RASPBERRY PI INTERFACE WITH BME280	524
RASPBERRY PI INTERFACE WITH USB WEBCAM	526
Introduction to IoT and LPWAN.....	529
Introduction to IoT	529
Low Power Wide Area Networks (LPWAN).....	531
Overview of IoT	532
Types of IoT Networks.....	534
Understanding LoRa Technology.....	536
History and Evolution of LoRa	537
LoRaWAN Architecture Overview	543
Key Components of LoRaWAN	543
LoRa vs. Other Wireless Technologies	544
Gateways and End Devices	544
LoRa Network Server (LNS)	547
Application Servers and Backend Integration	549
Types of LoRa Devices	552
Connecting sensors to LoRa modules	554
Data Transmission and Payload Formats	556
Security in IoT	558
LoRaWAN Security	559
Encryption: Protecting Data from Prying Eyes	566
Authentication: Verifying Identities and Ensuring Trust	567

Best Practices for Securing LoRa Networks	568
LoRa Network Deployment	569
Steps to Setting Up a LoRa Network	569
Programming LoRa Modules	572
HARDWARE PROGRAMS(LoRA)	576

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. There 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 may involve 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

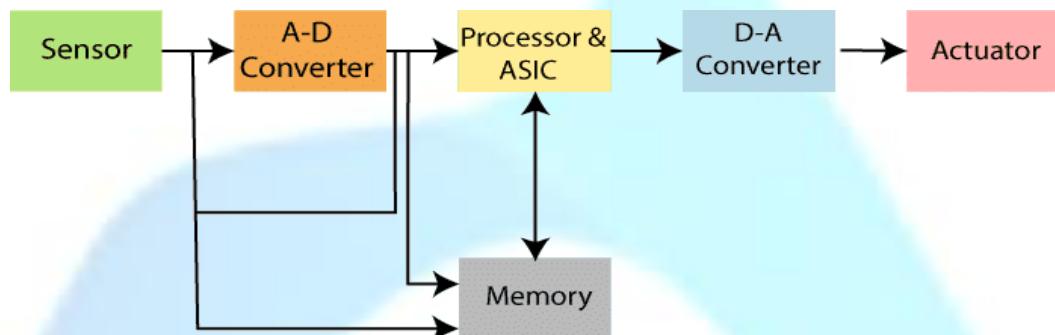


Fig.: Structure of Embedded System

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 as UART, 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, image processing, 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.

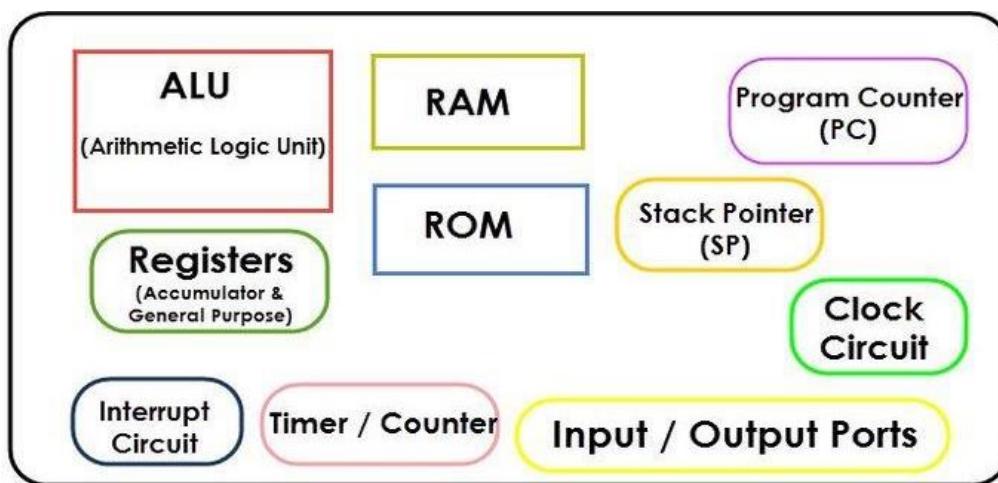


Fig.: Block Diagram of Microcontroller

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.

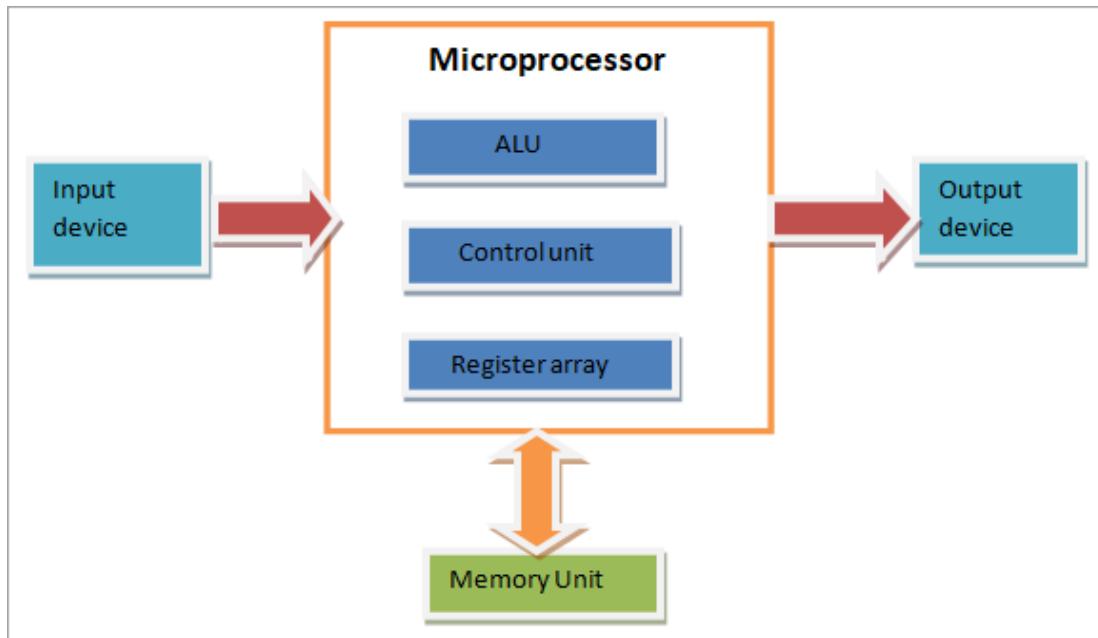


Fig.: Block Diagram Of Microprocessor

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 ALU operates 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 dedicated functions such as controlling a device or a system.	Primarily designed for general-purpose computing tasks. It serves as the central processing unit (CPU) in a computer system and is often used 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 specific applications.	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, and IoT 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 to its higher processing capabilities.
Instruction Set	Features a more specialized instruction set tailored to the specific applications it is designed for, which can contribute to better performance in those tasks.	Often has a more extensive and general-purpose instruction set to handle a variety of tasks.
Complexity	Simplified architecture with the essential 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.

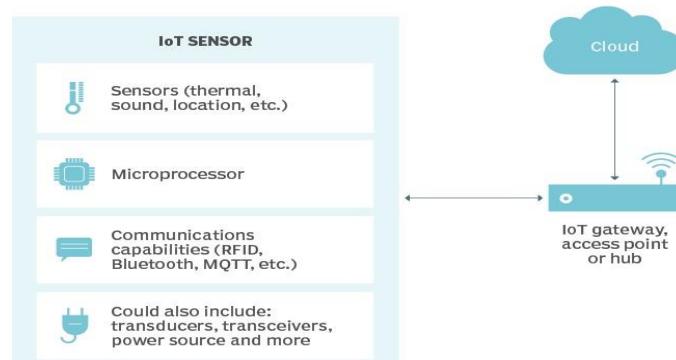


Fig.: IOT in action

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

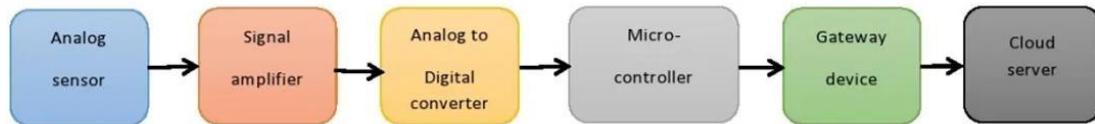


Fig.: IOT Sensor Workflow

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

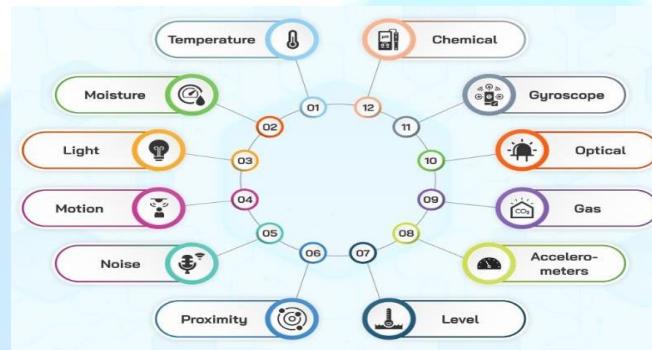


Fig.: Types of Sensors

- **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.

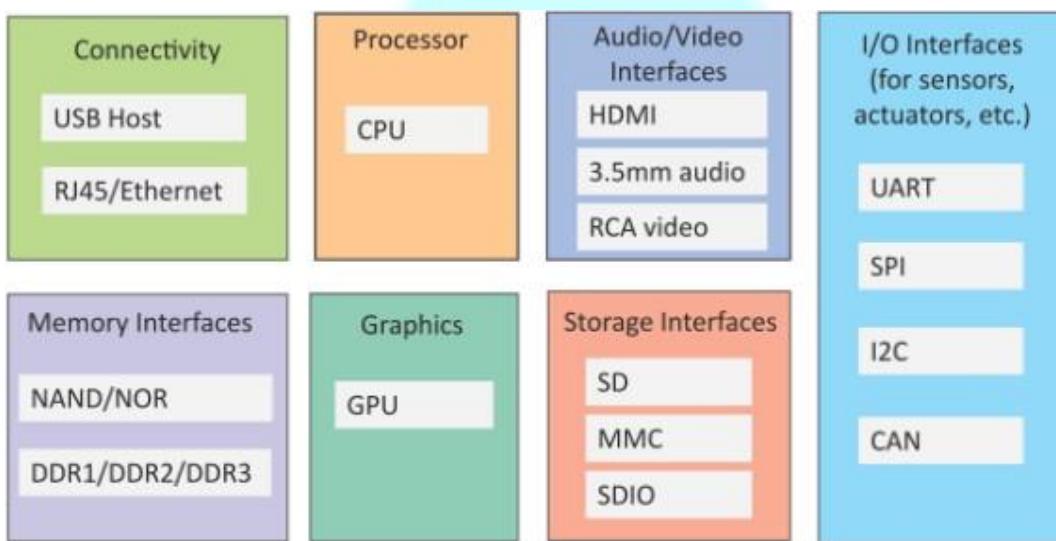


Fig.:Block Diagram of IOT

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
- MODBUS Protocol

I2C Protocol

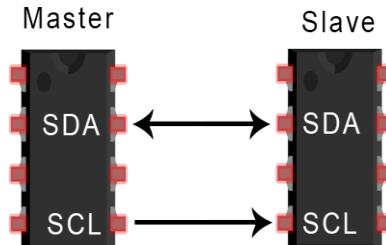


Fig.: I2C Protocol

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

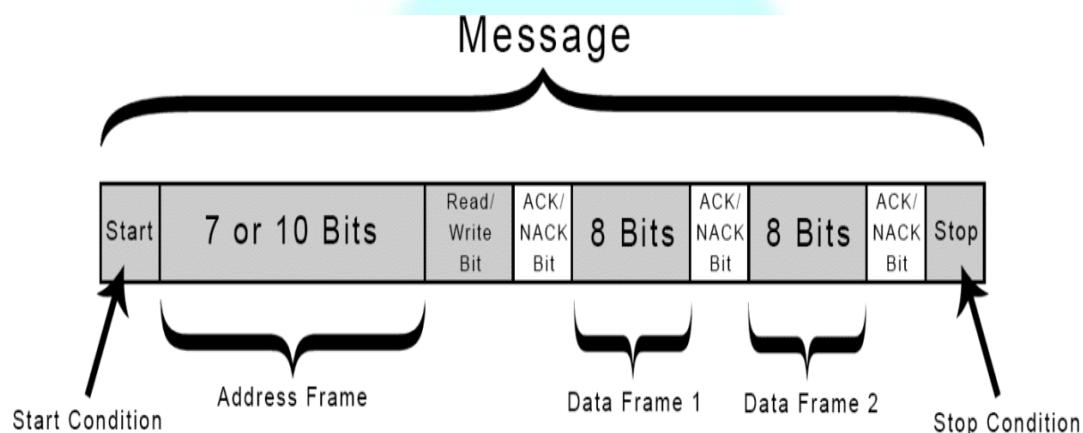
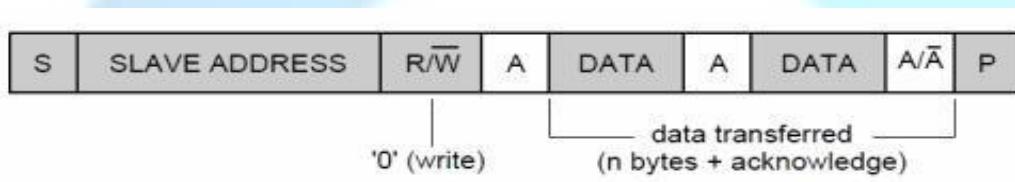


Fig.: I2C Frame Format



 from master to slave

A = acknowledge (SDA LOW)

 from slave to master

\bar{A} = not acknowledge (SDA HIGH)

S = START condition

P = STOP condition

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

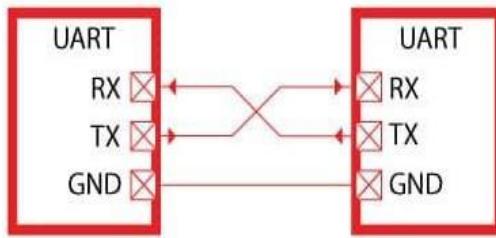


Fig.: UART Protocol

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

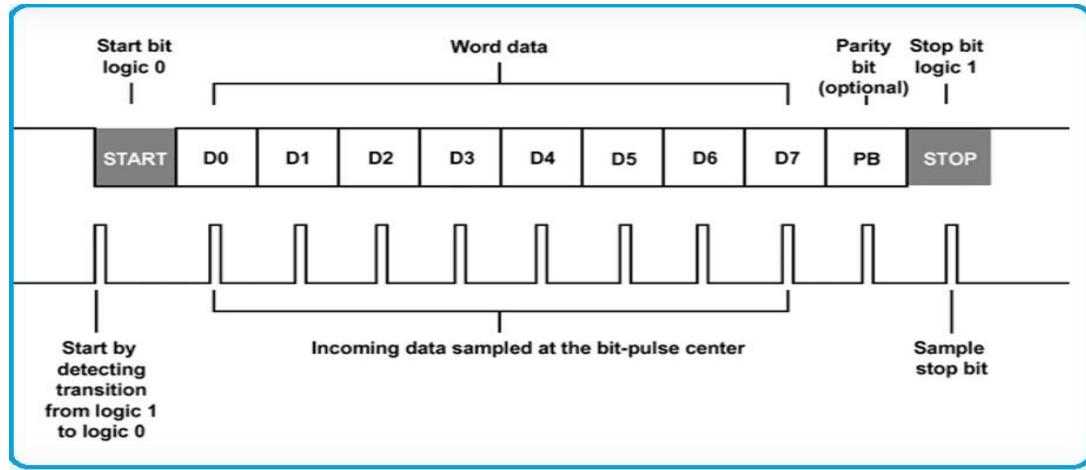


Fig.: UART Protocol Working

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.

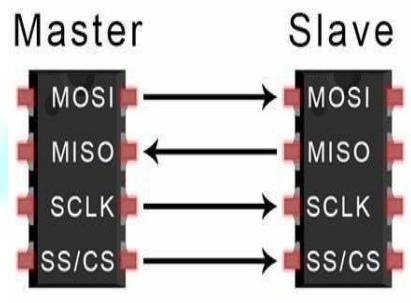


Fig.: SPI Protocol

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 in systems 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

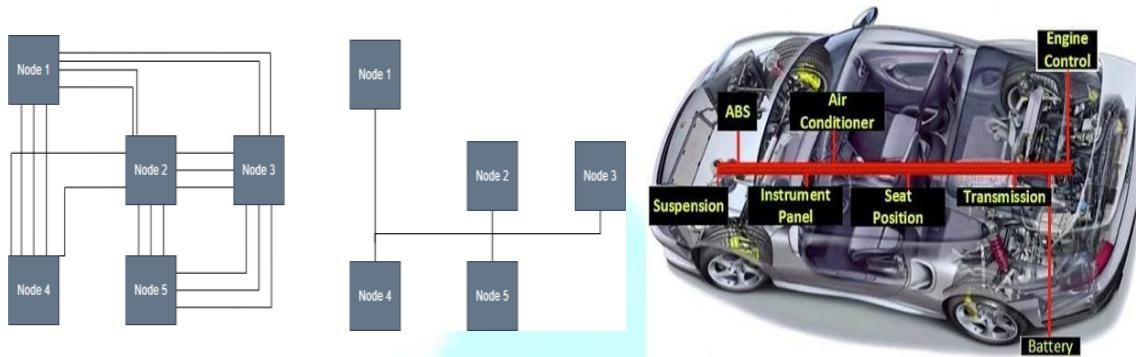


Fig.: ECUs In Automobiles

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 be used 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-bit identifier, 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 to be 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. Devices on 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 as Programmable Logic Controllers (PLCs), Remote Terminal Units (RTUs), sensors, and actuators.

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 is used 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, MODBUS is used for communication between sensors, HVAC (Heating, Ventilation, and Air Conditioning) systems, lighting controls, and other devices. This allows for centralized control 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, which are 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 communication between 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 collected data 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 across serial lines between various electronic devices.
- There will be Modbus master and Modbus slaves, while the master will be requesting the information; slaves will be supplying the information.
- Modbus protocol utilizes a simple message structure, which makes it easier to deploy.
- It is de-facto standard of the process control and automation industry.
- Since it is open-source protocol manufacturers 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 binary data 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

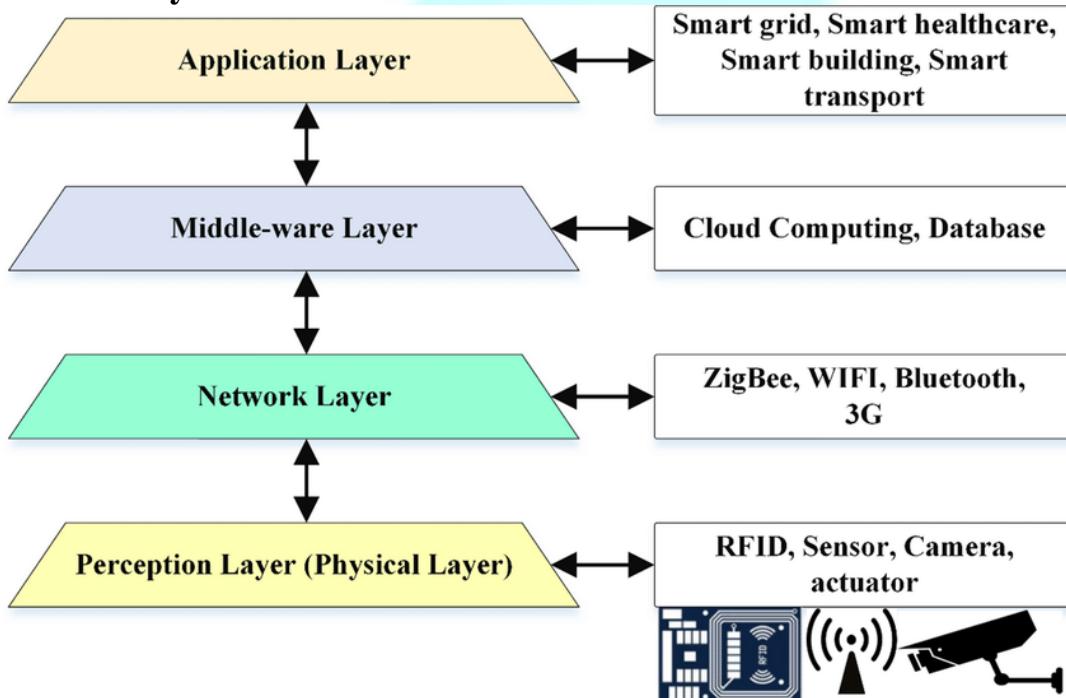


Fig.: 4-Layer Architecture

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

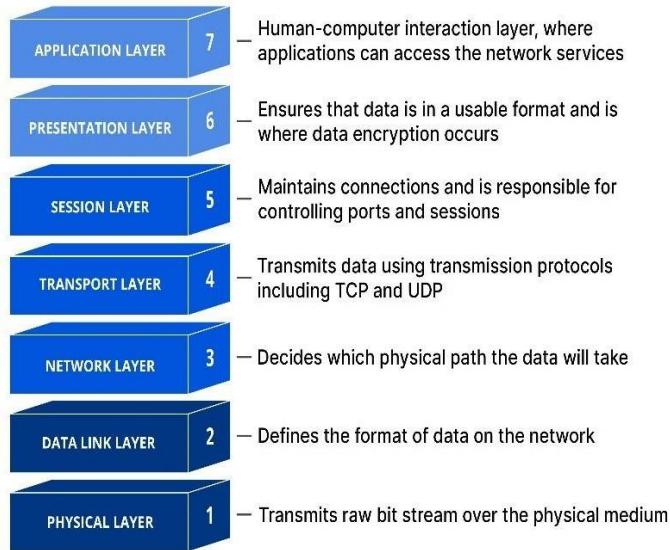


Fig.: OSI Model

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

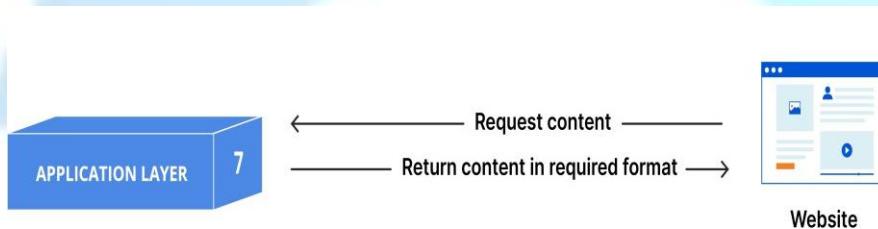


Fig.: Application Layer

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



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, if a 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 speed of 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 Control Message Protocol (ICMP), the Internet Group Message Protocol (IGMP), and the IPsec suite.

2. The Data Link layer

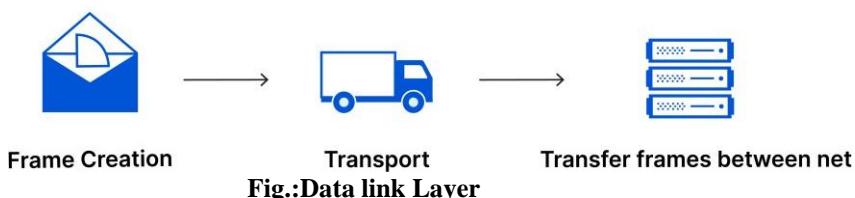


Fig.:Data link Layer

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 a string of 1s and 0s. The physical layer of both devices must also agree on a signal convention so 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.

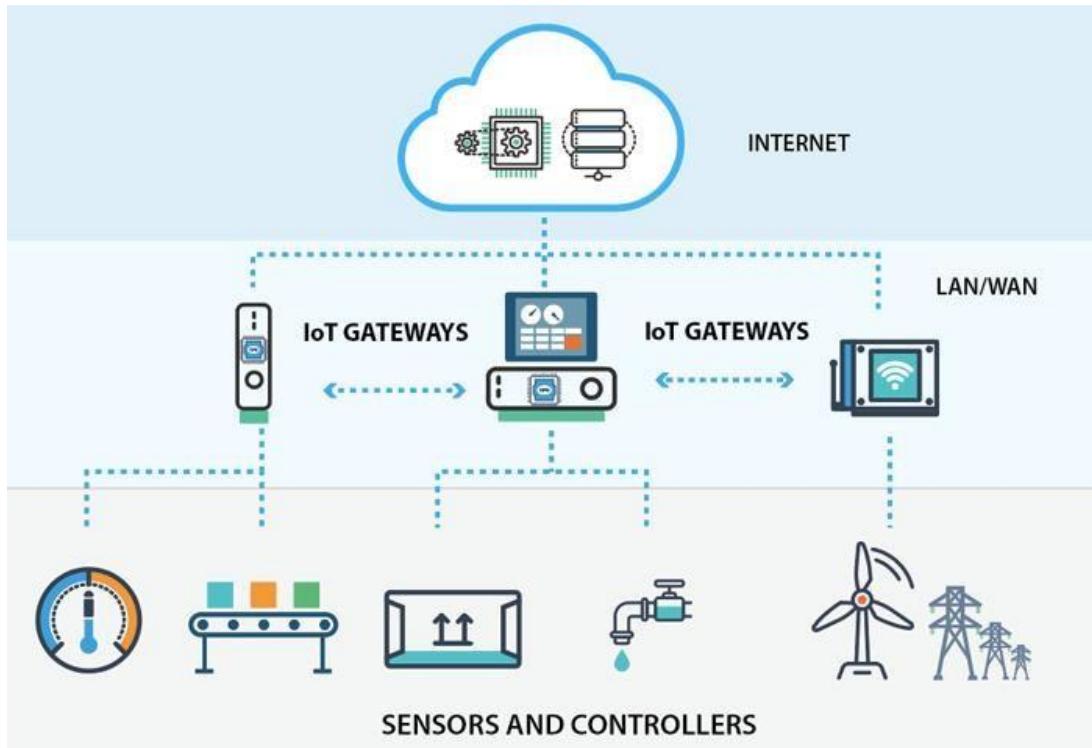


Fig.: IOT Gateway Working

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.

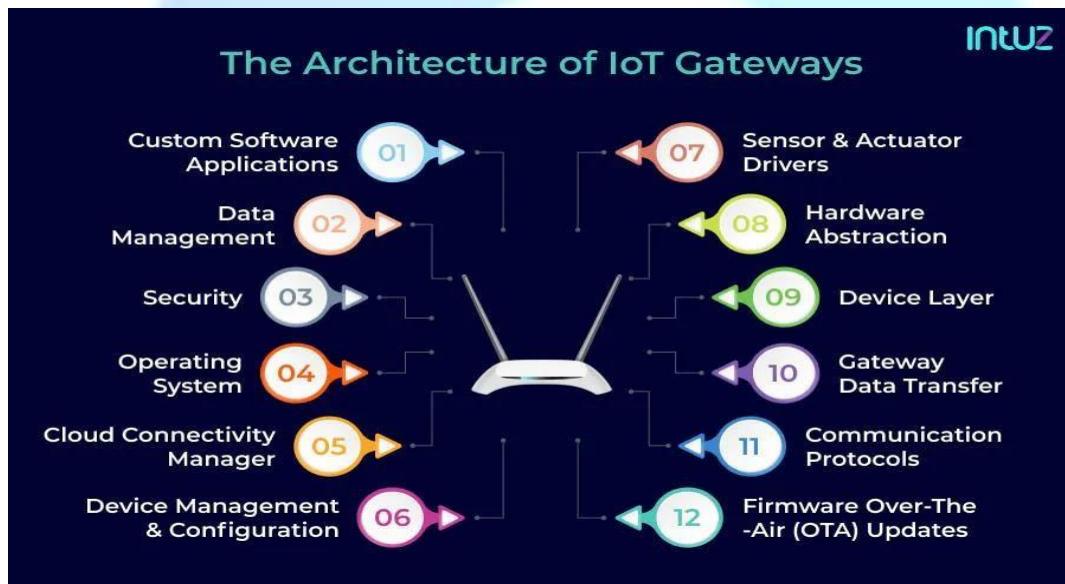


Fig.:IOT Gateway Architecture

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 outdoor operations, 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°C to 70°C)

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 of transmission.

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. Some examples 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. While the 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 IoT community 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 applications involving 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 in multiple 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 are very 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 reasonable number of IoT devices in proximity that make up a single IoT system, BLE is not so great for transferring 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 the more 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 and protocols, 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 within the 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

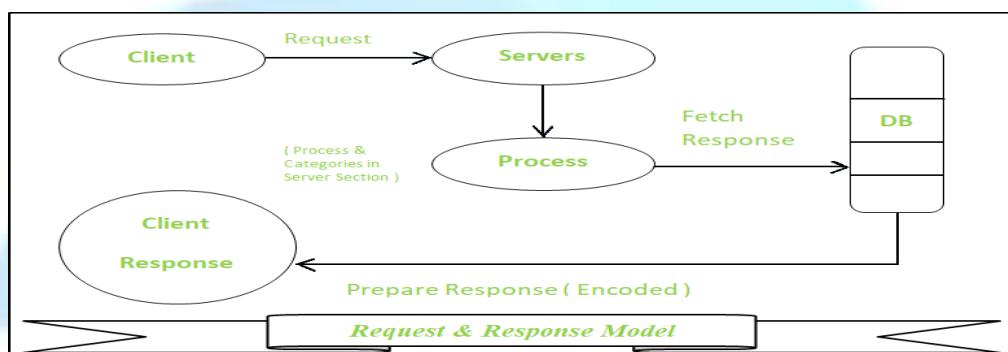


Fig.: Request and Response Model

- 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

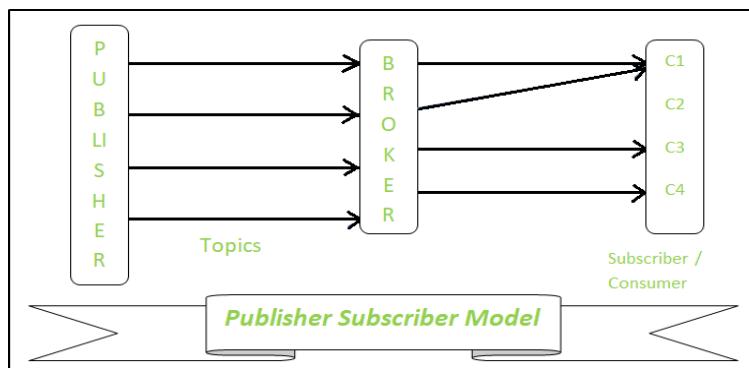


Fig.: Publisher-Subscriber Model

- 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

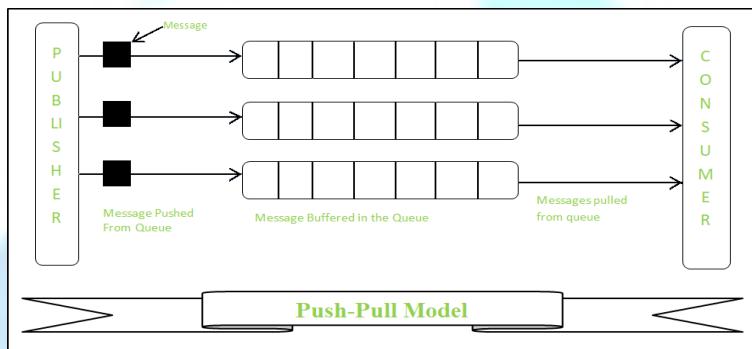


Fig.: Push-Pull Model

- 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

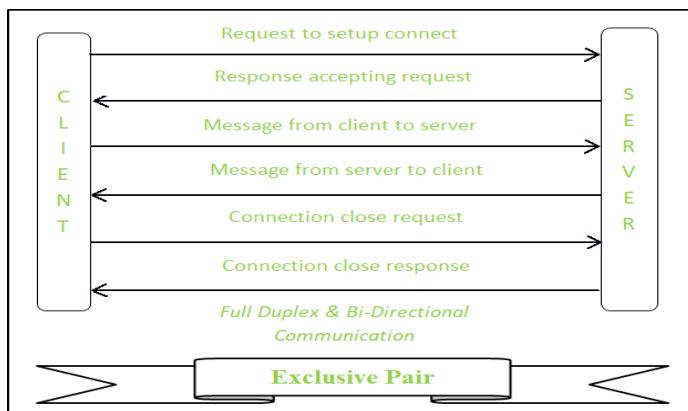


Fig.: Exclusive Pair

- 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.

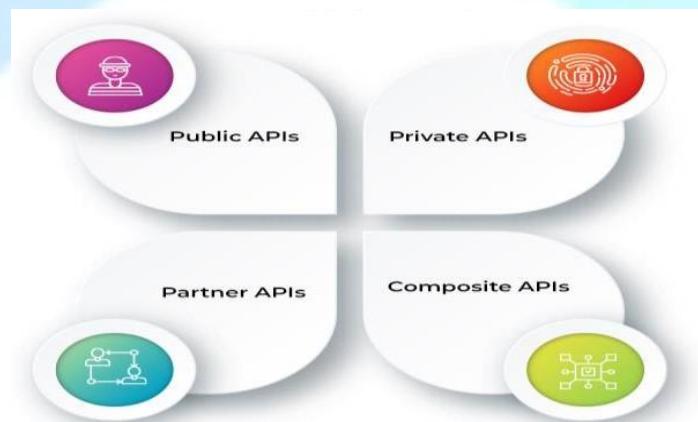


Fig.: Types of APIs

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's system framework. They are also known as closed or internal 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. For instance, 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

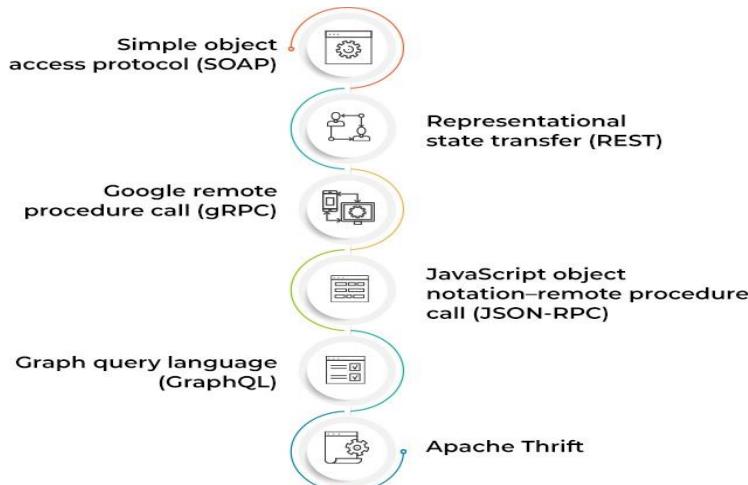


Fig.: Types of API Protocols

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.



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 a URI 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, return HTTP 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):

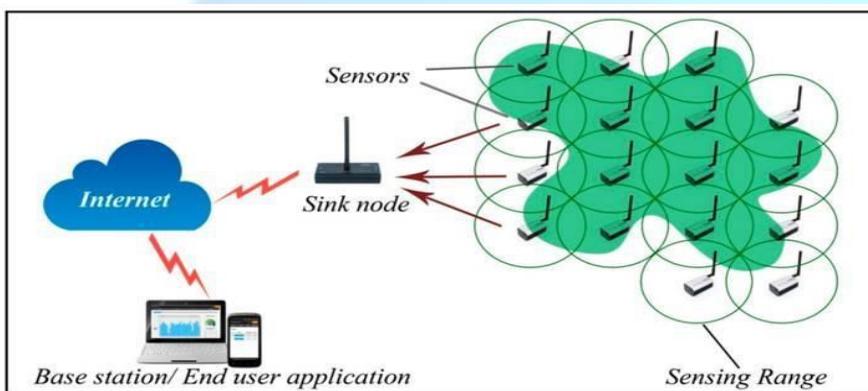


Fig.: WSN Working

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:

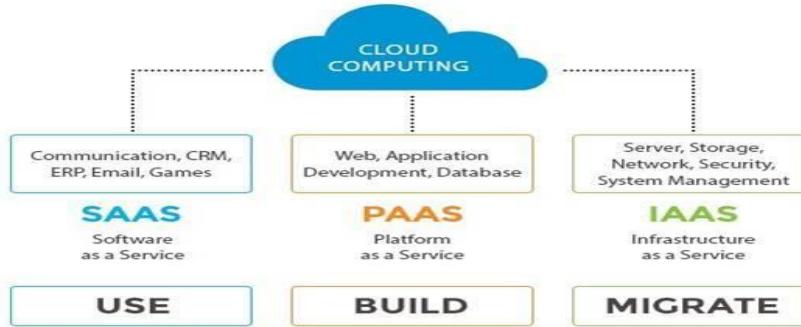


Fig.: Cloud Computing

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, web servers, 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:

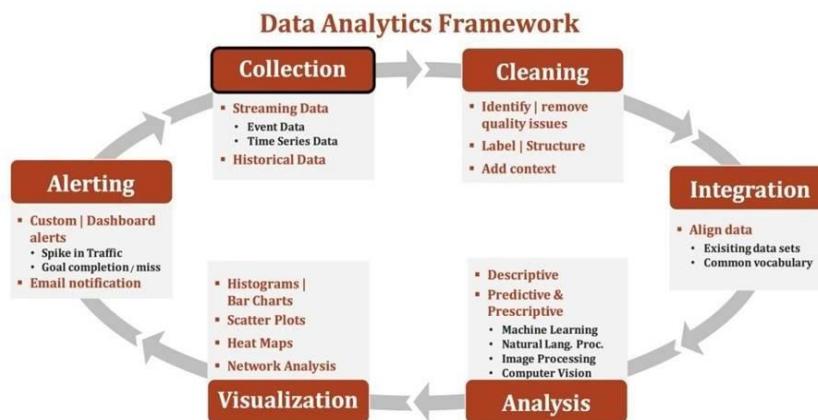


Fig.: Data Analytics Framework

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

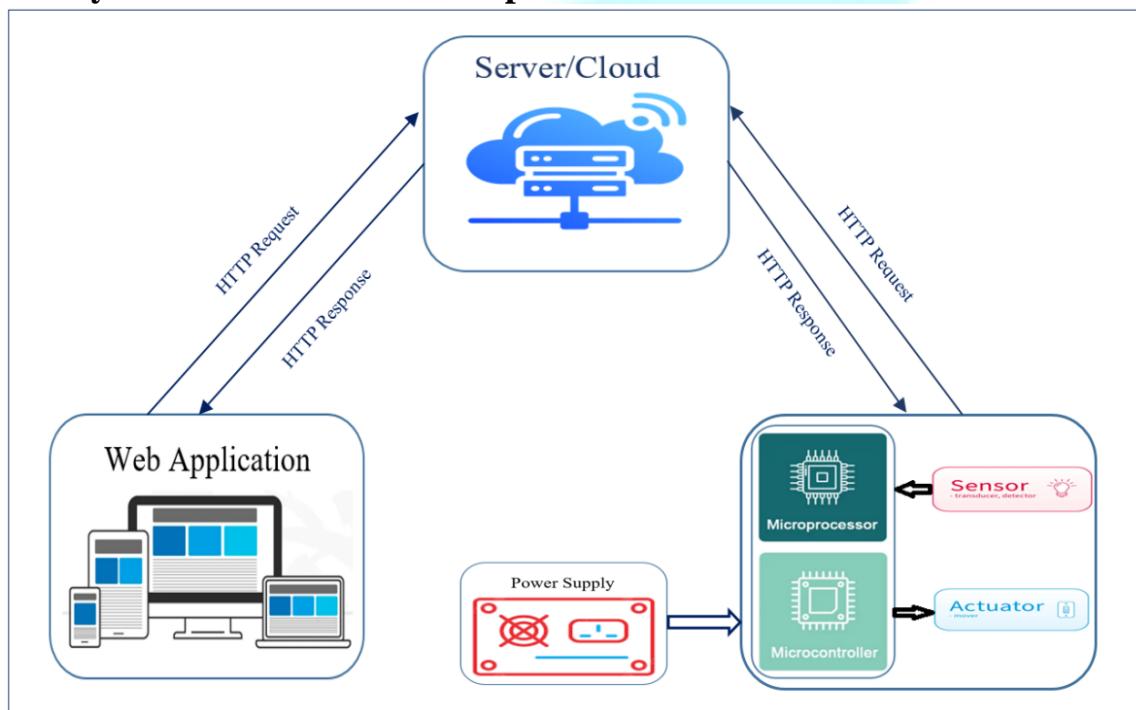


Fig.: IoT System Architecture and Operation with REST API

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 is 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:

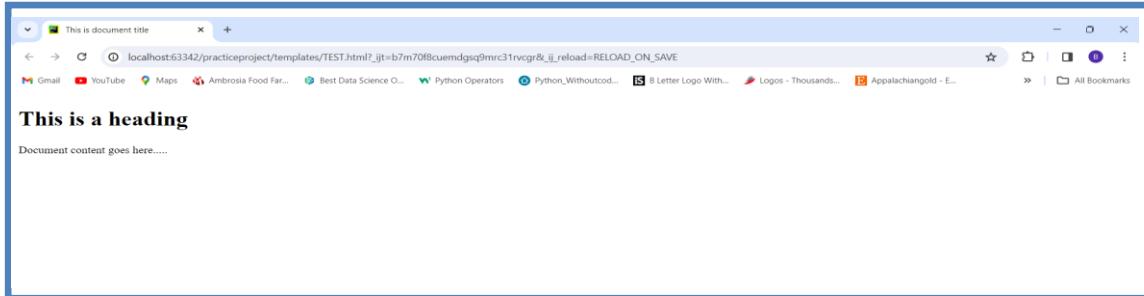


Fig.: Browser Output

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 `title`, and the `body` element contains child elements: `h1` and `p`.

HTML Tags

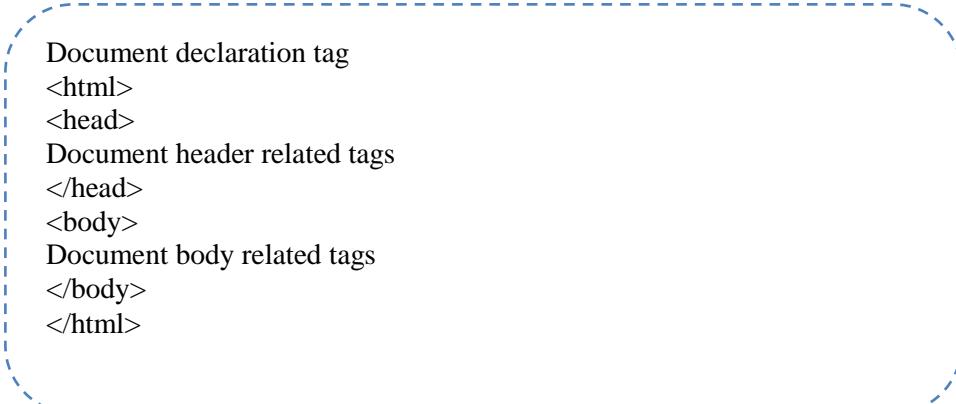
HTML is a markup language and makes use of various tags to format the content. These tags are enclosed within angle braces **<Tag Name>**. Except few tags, most of the tags have their corresponding closing tags. For example, **<html>** has its closing tag **</html>** and **<body>** tag has its closing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

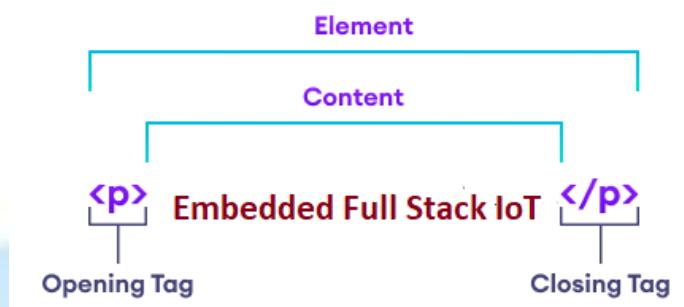


Fig.:HTML elements

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:

- : 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:

- <div>: 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 ():

- The 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: <p>First Line
Second Line</p>

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: `<div>`, `<p>`, `<h1>` through `<h6>`, ``, ``, ``, `<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>`, ``, ``, ``, `
`, `<i>`, ``, 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.	<code><div id="header">Header Section</div></code>
class	Specifies one or more class names for an element (used for CSS and JavaScript).	<code><p class="intro">This is an introductory paragraph.</p></code>
style	Specifies inline CSS styles for an element.	<code><h1 style="color: blue;">Blue Heading</h1></code>
title	Provides extra information about an element (displayed as a tooltip).	<code><abbr title="HyperText Markup Language">HTML</abbr></code>
data-*	Custom data attributes used to store private data.	<code><div data-user-id="12345">User Data</div></code>
Event Attributes		
onclick	Script to be run when an element is clicked.	<code><button onclick="alert('Button clicked!')">Click Me</button></code>
onmouseover	Script to be run when the mouse pointer is moved over an element.	<code><p onmouseover="this.style.color='red'">Hover over this text.</p></code>
onmouseout	Script to be run when the mouse pointer is moved out of an element.	<code><p onmouseout="this.style.color='black'">Move the mouse away from this text.</p></code>

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.	
alt	Provides alternative text for an image if it cannot be displayed.	
width	Specifies the width of the image.	
height	Specifies the height of the image.	
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

This is heading 1.

This is heading 2.

This is heading 3.

This is heading 4.

This is heading 5.

This is heading 6.

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

Headings

are

fun!

HTML Paragraphs

The HTML `<p>` 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 `<p>` and ends with the `</p>` 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 `the strong tag to emphasize text`

```
</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 &nbsp;&nbsp; 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.

HTML Formatting Elements

Formatting elements were designed to display special types of text:

- ** - Bold text :**

The HTML `` element defines bold text, without any extra importance.

For example:

```
<p>This is <b>bold</b> text.</p>
```

- ** - Important text :**

The HTML `` element defines text with strong importance. The content inside is typically displayed in bold.

For example:

```
<p>This is <strong>important</strong> text.</p>
```

- **<i> - Italic text :**

The HTML `<i>` element defines a part of text in an alternate voice or mood. The content inside is typically displayed in italic.

For example:

```
<p>This is <i>italic</i> text.</p>
```

- ** - Emphasized text :**

The HTML `` element defines emphasized text. The content inside is typically displayed in italic.

For example:

```
<p>This is <em>emphasized</em> text.</p>
```

- **<mark> - Marked text :**

The HTML `<mark>` element defines text that should be marked or highlighted

For example:

```
<p>This is <mark>highlighted</mark> text.</p>
```

- **<small> - Smaller text :**

The HTML `<small>` element defines smaller text

For example:

```
<p>This is <small>smaller</small> text.</p>
```

- ** - Deleted text :**

The HTML `` element defines text that has been deleted from a document.

Browsers will usually strike a line through deleted text

For example:

```
<p>This is <del>deleted</del> text.</p>
```

- **<ins> - Inserted text :**

The HTML `<ins>` element defines a text that has been inserted into a document. Browsers will usually underline inserted text

For example:

```
<p>This is <ins>inserted</ins> text.</p>
```

- **<sub> - Subscript text :**

The HTML `<sub>` element defines subscript text. Subscript text appears half a character below the normal line, and is sometimes rendered in a smaller font. Subscript text can be used for chemical formulas, like H₂O

For example:

```
<p>This is <sub>subscript</sub> text.</p>
```

- **<sup> - Superscript text :**

The HTML `<sup>` element defines superscript text. Superscript text appears half a character above the normal line, and is sometimes rendered in a smaller font. Superscript text can be used for footnotes, like WWW[1]

For example:

```
<p>This is <sup>superscript</sup> text.</p>
```

Examples in Context of all of these formatting elements

```
<!DOCTYPE html>

<html lang="en">

<head>

    <meta charset="UTF-8">

    <meta name="viewport" content="width=device-width, initial-scale=1.0">

    <title>HTML Formatting Elements</title>

</head>

<body>

    <h1>HTML Formatting Elements</h1>

    <p>This is <b>bold</b> text.</p>

    <p>This is <strong>important</strong> text.</p>

    <p>This is <i>italic</i> text.</p>

    <p>This is <em>emphasized</em> text.</p>

    <p>This is <mark>highlighted</mark> text.</p>

    <p>This is <small>smaller</small> text.</p>

    <p>This is <del>deleted</del> text.</p>

    <p>This is <ins>inserted</ins> text.</p>

    <p>This is <sub>subscript</sub> text, and this is <sup>superscript</sup> text.</p>

</body>

</html>
```

The browser output:

HTML Formatting Elements

This is **bold** text.

This is **important** text.

This is *italic* text.

This is *emphasized* text.

This is **highlighted** text.

This is smaller text.

This is ~~deleted~~ text.

This is inserted text.

This is _{subscript} text, and this is ^{superscript} text.

HTML Comments

HTML comments are used to add notes or explanations within the HTML code that are not displayed in the web browser. Comments are useful for documentation, reminders, and for temporarily disabling parts of the code during development and debugging.

Here is how to create and use HTML comments

The syntax for a single line HTML comment is:

```
<!-- This is a comment -->
```

The syntax for an multi line HTML comment is:

```
<!--  
This is a multi-line comment.  
It can span multiple lines.  
-->
```

HTML Quotations

The HTML Quotation elements are used to insert quoted texts in a web page, that is the portion of texts different from the normal texts in the web page. Below are some of the most used quotation elements of the HTML.

Quotation elements of the HTML

Tag	Description
<abbr>	Defines abbreviation or acronym.
<address>	Defines contact info for the author/owner of a document.
<bdo>	Defines text direction, left-to-right or right-to-left.
<blockquote>	Defines a section quoted from another source.
<cite>	Defines the title of a work, book, article, or publication.
<q>	Defines short inline quotation, enclosed in quotation marks.

```
<!DOCTYPE html>

<html>

    <head>
        <title>HTML Quotations</title>
    </head>

    <body>
        <!--Normal text-->
        <p>Internet of Things</p>
        <!--Inside <bdo> tag-->
        <p> <bdo dir="rtl">Internet of Things</bdo> </p>
        <p> Embedded Full Stack <abbr title="Internet of Things">IoT</abbr></p>
        <address>
            <p> Address:<br />
            MSK Solutions,<br/>
            #475,Sangolli Rayanna Nagar, Dharwad </p>
        </address>
        <p>As Mahatma Gandhi once said, <q
        cite="https://www.example.com/quote">Be the change you wish to see in the
        world.</q></p>
        <blockquote cite="https://www.example.com"> This is a long quotation from a
        source. </blockquote>
    </body>
</html>
```

Browser output:

Internet of Things

sgnihT fo tenretnI

Embedded Full Stack IoT

Address:

*MSK Solutions,
#475, Sangolli Rayanna Nagar, Dharwad*

As Mahatma Gandhi once said, “Be the change you wish to see in the world.”

This is a long quotation from a source.



HTML Colors

HTML colors are used in web design to enhance the visual appeal of web pages. Colors in HTML can be specified in various formats: by name, by hexadecimal value, by RGB value, by RGBA value, by HSL value, and by HSLA value.

Here's an overview of these formats:

1. Named Colors

HTML supports 140 named colors, like red, blue, green, black, white, gray, aqua, fuchsia, etc.

```
<p style="color: red;">This is red text.</p>
<p style="background-color: blue;">This is a blue background.</p>
```

2. Hexadecimal Colors

Colors can be specified using a hex code, which is a combination of three pairs of hexadecimal digits, representing the red, green, and blue components.

```
<p style="color: #FF0000;">This is red text.</p>
<p style="background-color: #0000FF;">This is a blue background.</p>
```

3. RGB Colors

The RGB color model represents colors using their red, green, and blue components. Each component can have a value between 0 and 255.

```
<p style="color: rgb(255, 0, 0);">This is red text.</p>
<p style="background-color: rgb(0, 0, 255);">This is a blue background.</p>
```

4. RGBA Colors

RGBA is similar to RGB but includes an alpha channel, which represents the opacity. The alpha value ranges from 0 (completely transparent) to 1 (completely opaque).

```
<p style="color: rgba(255, 0, 0, 0.5);">This is semi-transparent red text.</p>  
<p style="background-color: rgba(0, 0, 255, 0.5);">This is a semi-transparent blue background.</p>
```

5. HSL Colors

HSL stands for Hue, Saturation, and Lightness. Hue is a degree on the color wheel (0-360), Saturation is a percentage (0%-100%), and Lightness is a percentage (0%-100%).

```
<p style="color: hsl(0, 100%, 50%);>This is red text.</p>  
<p style="background-color: hsl(240, 100%, 50%);>This is a blue background.</p>
```

6. HSLA Colors

HSLA is similar to HSL but includes an alpha channel for opacity.

```
<p style="color: hsla(0, 100%, 50%, 0.5);>This is semi-transparent red text.</p>  
<p style="background-color: hsla(240, 100%, 50%, 0.5);>This is a semi-transparent blue background.</p>
```

```
<!DOCTYPE html>

<html lang="en">

<head>

    <meta charset="UTF-8">

    <meta name="viewport" content="width=device-width, initial-scale=1.0">

    <title>HTML Colors Example</title>

</head>

<body>

    <p style="color: red;">This is red text (Named Color).</p>

    <p style="color: #FF0000;">This is red text (Hexadecimal Color).</p>

    <p style="color: rgb(255, 0, 0);>This is red text (RGB Color).</p>

    <p style="color: rgba(255, 0, 0, 0.5);>This is semi-transparent red text (RGBA Color).</p>

    <p style="color: hsl(0, 100%, 50%);>This is red text (HSL Color).</p>

    <p style="color: hsla(0, 100%, 50%, 0.5);>This is semi-transparent red text (HSLA Color).</p>

</body>

</html>
```

Browser Output:

This is red text (Named Color).
This is red text (Hexadecimal Color).
This is red text (RGB Color).
This is semi-transparent red text (RGBA Color).
This is red text (HSL Color).
This is semi-transparent red text (HSLA Color).

HTML Images

The HTML `` tag is used to embed an image in web pages by linking them. It creates a placeholder for the image, defined by attributes like `src`, `width`, `height`, and `alt`, and does not require a closing tag.

There are 2 ways to insert the images into a webpage:

- By providing a full path or address (URL) to access, an internet file.
- By providing the file path relative to the location of the current web page file.

Syntax :

```
<img src = "url" alt = "some_text" width="width_value" height="height_value">
```

```

```

- **src:** The source attribute specifies the path to the image. This can be a relative path (to an image file in your project) or an absolute URL (to an image hosted online).
- **alt:** The alternative text attribute provides a textual description of the image. This text is displayed if the image cannot be loaded and is also used for accessibility purposes.
- **width:** Specifies the width of the image in pixels or percentage.
- **height:** Specifies the height of the image in pixels or percentage.

Common Image Format:

Here is the commonly used image file format that is supported by all the browsers.

S.No.	Abbreviation	File Type	Extension
1.	PNG	Portable Network Graphics.	.png
2.	JPEG.	Joint Photographic Expert Group image.	.jpg, .jpeg, .jfif, .pjpeg, .pjp
3.	SVG	Scalable Vector Graphics.	.svg.
4.	GIF	Graphics Interchange Format.	.gif
5.	ICO	Microsoft Icon.	.ico, .cur
6.	APNG	Animated Portable Network Graphics.	.apng

```

<!DOCTYPE html>
<html>
<body>
<h2>Image with size attributes </h2>

</body>
</html>

```

Browser output:

Image with size attributes



HTML Link- Hyperlinks

HTML links, or hyperlinks, connect web pages. They're created using the `<a>` tag , which stands for "anchor." with the `href` attribute, which specifies the destination URL. Users can click on links to navigate between different pages or resources.

Syntax:

```
<a href="url">link text</a>
```

Note: A hyperlink can be represented by an image or any other HTML element, not just text.

By default, links will appear as follows in all browsers:

- An unvisited link is underlined and blue
- A visited link is underlined and purple
- An active link is underlined and red

Target Attribute:

Attribute	Description
_blank	Opens the linked document in a new window or tab.
_self	Opens the linked document in the same frame or window as the link. (Default behavior)
_parent	Opens the linked document in the parent frame.
_top	Opens the linked document in the full body of the window.
framename	Opens the linked document in a specified frame. The frame's name is specified in the attribute.

Link to an External Website

```
<a href="https://www.google.com">Visit Google</a>
```

Link to an Internal Page

Assuming your file structure is:

```
/project
```

```
  /about.html
```

```
  /index.html
```

```
<a href="about.html">About Us</a>
```

Link to a Section Within the Same Page

First, create an anchor in the page:

```
<h2 id="section1">Section 1</h2>
```

Then link to this section:

```
<a href="#section1">Go to Section 1</a>
```

Link with an Image

```
<a href="https://www.example.com">  
    
</a>
```

Opening Links in a New Tab

Use the target attribute with the value _blank:

```
<a href="https://www.example.com" target="_blank">Visit Example.com</a>
```

Adding a Tooltip to a Link

Use the title attribute:

```
<a href="https://www.example.com" title="Go to Example.com">Visit Example.com</a>
```

Link with Email

Use the mailto: protocol:

```
<a href="mailto:someone@example.com">Send Email</a>
```

Example of a Basic Link

```
<!DOCTYPE html>  
  
<html lang="en">  
  
<head>  
  
    <meta charset="UTF-8">  
  
    <meta name="viewport" content="width=device-width, initial-scale=1.0">  
  
    <title>HTML Links Example</title>  
  
</head>  
  
<body>  
  
    <h1>Example of a Link in HTML</h1>  
  
    <a href="https://www.example.com">Visit Example.com</a>  
  
</body>  
  
</html>
```

The Browser output:

Example of a Link in HTML

[Visit Example.com](https://www.example.com)

```

<!DOCTYPE html>
<html lang="en">
<head><meta charset="UTF-8">
<meta name="viewport" content="width=device-width, initial-scale=1.0">
<title>HTML Links Example</title></head>
<body>
<h1>Example of Links in HTML</h1>
<!-- External Link -->
<p><a href="https://www.google.com" target="_blank" title="Visit Google">Visit Google</a></p>
<!-- Internal Link -->
<p><a href="about.html">About Us</a></p>
<!-- Link to a Section -->
<p><a href="#section1">Go to Section 1</a></p>
<!-- Section for Internal Link -->
<h2 id="section1">Section 1</h2>
<p>This is section 1 of the page.</p>
<!-- Link with an Image -->
<p><a href="https://www.example.com">

</a></p>
<!-- Email Link -->
<p><a href="mailto:someone@example.com">Send Email</a></p>
</body>
</html>

```

The Browser output:

Example of Links in HTML

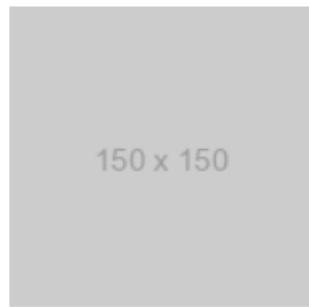
[Visit Google](#)

[About Us](#)

[Go to Section 1](#)

Section 1

This is section 1 of the page.



[Send Email](#)

HTML Tables

An HTML Table is an arrangement of data in rows and columns in tabular format. Tables are useful for various tasks, such as presenting text information and numerical data. A table is a useful tool for quickly and easily finding connections between different types of data. Tables are also used to create databases.

An HTML table is defined with the `<table>` tag. Each table row is defined with the `<tr>` tag. A table header is defined with the `<th>` tag. By default, table headings are bold and centered. A table data/cell is defined with the `<td>` tag.

- **Table Cells :** Table Cell are the building blocks for defining the Table. It is denoted with `<td>` as a start tag & `</td>` as a end tag.

Syntax : `</td> Content...</td>`

- **Table Rows :** The rows can be formed with the help of combination of Table Cells. It is denoted by `<tr>` and `</tr>` tag as a start & end tags.

Syntax : `</tr> Content...</tr>`

- **Table Headers :** The Headers are generally use to provide the Heading. The Table Headers can also be used to add the heading to the Table. This contains the `<th>` & `</th>` tags.

Syntax : `</th> Content...</th>`

- **Adding a border to an HTML Table :** A border is set using the CSS border property. If you do not specify a border for the table, it will be displayed without borders.

Syntax : `table, th, td { border: 1px solid black; }`

- **Adding Collapsed Borders in an HTML Table:** For borders to collapse into one border, add the CSS border-collapse property.

Syntax :

```
table, th, td {
    border: 1px solid black;
    border-collapse: collapse;
}
```

- **Adding Cell Padding in an HTML Table :** Cell padding specifies the space between the cell content and its borders. If we do not specify a padding, the table cells will be displayed without padding.

Syntax :

```
th, td {  
    padding: 20px;  
}
```

- **Adding Border Spacing in an HTML Table :** Border spacing specifies the space between the cells. To set the border-spacing for a table, we must use the CSS border-spacing property.

Syntax

```
table {  
    border-spacing: 5px;  
}
```

Tags used in HTML Tables

HTML Tags	Descriptions
<u><table></u>	Defines the structure for organizing data in rows and columns within a web page.
<u><tr></u>	Represents a row within an HTML table, containing individual cells.
<u><th></u>	Shows a table header cell that typically holds titles or headings.
<u><td></u>	Represents a standard data cell, holding content or data.
<u><caption></u>	Provides a title or description for the entire table.
<u><thead></u>	Defines the header section of a table, often containing column labels.
<u><tbody></u>	Represents the main content area of a table, separating it from the header or footer.
<u><tfoot></u>	Specifies the footer section of a table, typically holding summaries or totals.
<u><col></u>	Defines attributes for table columns that can be applied to multiple columns at once.
<u><colgroup></u>	Groups together a set of columns in a table to which you can apply formatting or properties collectively.

Example for HTML Table

```
<!DOCTYPE html>
<html>
<head>
    <style> table, th , td { border: 1px solid black; } </style>
</head>
<body>
    <table style="width:100%; text-align :center;">
        <tr>
            <th>First Name</th>
            <th>Last Name</th>
            <th>Age</th>
        </tr>
        <tr>
            <td>Priya</td>
            <td>Sharma</td>
            <td>24</td>
        </tr>
        <tr>
            <td>Arun</td>
            <td>Singh</td>
            <td>32</td>
        </tr>
        <tr>
            <td>Sam</td>
            <td>Watson</td>
            <td>41</td>
        </tr>
    </table>
</body>
</html>
```

Browser Output:

Firstname	Lastname	Age
Priya	Sharma	24
Arun	Singh	32
Sam	Watson	41

HTML Lists

HTML lists are used to display related information in an easy-to-read and concise way as lists.

We can use three types of lists to represent different types of data in HTML:

1. Unordered List
2. Ordered List
3. Description List <dl>

Unordered List

The unordered list is used to represent data in a list for which the order of items does not matter.

In HTML, we use the tag to create unordered lists. Each item of the list must be a tag which represents list items. For example,

```
<ul>
    <li>Apple</li>
    <li>Orange</li>
    <li>Mango</li>
</ul>
```

Here, Apple, Orange, and Mango are the list item

Browser Output

- Apple
- Orange
- Mango

Unordered HTML List - Choose List Item Marker

The CSS list-style-type property is used to define the style of the list item marker. It can have one of the following values:

Value	Description
disc	Sets the list item marker to a bullet (default)
circle	Sets the list item marker to a circle
square	Sets the list item marker to a square
none	The list items will not be marked

Examples for all the marker types

Dot	Circle	Square	None
● Apple	○ Apple	■ Apple	Apple
● Mango	○ Mango	■ Mango	Mango
● Orange	○ Orange	■ Orange	Orange
● Banana	○ Banana	■ Banana	Banana

Nesting Lists

In HTML, we can create a nested list by adding one list inside another.

Example for nested unordered list:

```
<ul>
  <li>
    Coffee
    <ul>
      <li>Cappuccino</li>
      <li>Americano</li>
      <li>Espresso</li>
    </ul>
  </li>
  <li>
    Tea
    <ul>
      <li>Milk Tea</li>
      <li>Black Tea</li>
    </ul>
  </li>
  <li>Milk</li>
</ul>
```

Browser Output:

- Coffee
 - Cappuccino
 - Americano
 - Espresso
- Tea
 - Milk Tea
 - Black Tea
- Milk

Ordered List

The ordered list is used to represent data in a list for which the order of items has significance.

The `` tag is used to create ordered lists. Similar to unordered lists, each item in the ordered list must be a `` tag. For example,

```
<ol>
    <li>Ready</li>
    <li>Set</li>
    <li>Go</li>
</ol>
```

Browser Output

1. Ready
2. Set
3. Go

Ordered HTML List - The Type Attribute

The type attribute of the `` tag, defines the type of the list item marker:

Type	Description
<code>type="1"</code>	The list items will be numbered with numbers (default)
<code>type="A"</code>	The list items will be numbered with uppercase letters
<code>type="a"</code>	The list items will be numbered with lowercase letters
<code>type="I"</code>	The list items will be numbered with uppercase roman numbers
<code>type="i"</code>	The list items will be numbered with lowercase roman numbers

Examples of all ordered number type lists:

type = "1"	type = "a"	type = "A"	type = "i"	type = "I"
1. Name	a. Name	A. Name	i. Name	I. Name
2. Address	b. Address	B. Address	ii. Address	II. Address
3. Phone Number	c. Phone Number	C. Phone Number	iii. Phone Number	III. Phone Number

start Attribute

We use the start attribute to change the starting point for the numbering of the list. For example,

```
<ol start='5'>
    <li>Harry</li>
    <li>Ron</li>
    <li>Sam</li>
</ol>
```

Browser Output

```
5. Harry
6. Ron
7. Sam
```

Here, we change the starting value of the list to 5.

This attribute also works with other types. For example,

```
<ol type="a" start='5'>  
    <li>Harry</li>  
    <li>Ron</li>  
    <li>Sam</li>  
</ol>
```

Browser Output

```
e. Harry  
f. Ron  
g. Sam
```

Similarly, we can use the start attribute along with all other types.

reversed Attribute

We can use the reversed attribute on the ordered list to reverse the numbering on the list. For example,

```
<ol reversed>  
    <li>Cat</li>  
    <li>Dog</li>  
    <li>Elephant</li>  
    <li>Fish</li>  
</ol>
```

Browser Output

- 4. Cat
- 3. Dog
- 2. Elephant
- 1. Fish

Here, we can see the order of the list is reversed, the first list item is numbered 4 and the last is numbered 1.

Similarly, the reversed attribute can also be used with other types and in conjunction with the start attribute. For example,

```
<ol reversed type="I" start="10">  
    <li>Cat</li>  
    <li>Dog</li>  
    <li>Elephant</li>  
    <li>Fish</li>  
</ol>
```

Browser Output

- X. Cat
- IX. Dog
- VIII. Elephant
- VII. Fish

In the above example, we use the upper-case roman numeral type and start at 10 and reverse the order of the numbers.

Description List

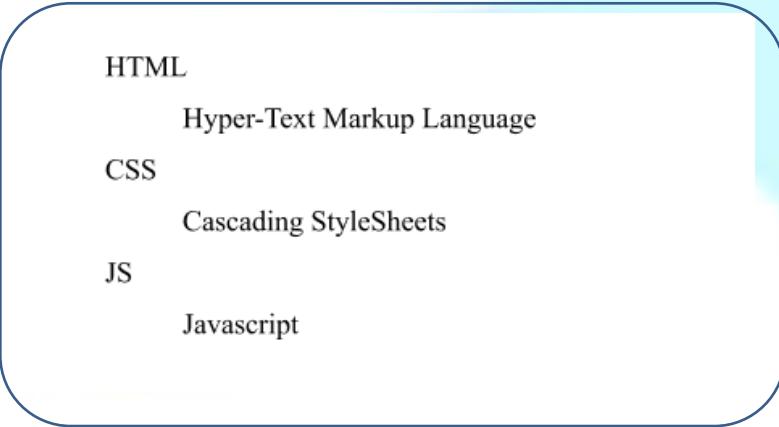
The HTML description list is used to represent data in the name-value form. We use the `<dl>` tag to create a definition list and each item of the description list has two elements:

- term/title - represented by the `<dt>` tag
- description of the term - represented by the `<dd>` tag

Let's see an example,

```
<dl>
    <dt>HTML</dt>
    <dd>Hyper-Text Markup Language</dd>
    <dt>CSS</dt>
    <dd>Cascading StyleSheets</dd>
    <dt>JS</dt>
    <dd>Javascript</dd>
</dl>
```

Browser Output



```
HTML
Hyper-Text Markup Language
CSS
Cascading StyleSheets
JS
Javascript
```

Here, it consists of two different types of list items:

- `<dt>` - defines the terms/name
- `<dd>` - defines the description/value of the term/name

The description list includes two related values, hence it can also be used to store items in key/value pairs.

Since the description list includes the definition of a term, it is also known as the definition list.

Multiple Terms and Multiple Definitions

The definition list is used to display data in a key/value format, where the `<dt>` tag indicates the key elements and the `<dd>` tag element indicates the value (definition) of the key.

However, while creating a description list, it's not necessary that a single `<dt>` tag (key) should have a single `<dd>` tag (value). We can have any combination of the `<dt>` and `<dd>` elements.

Possible Combinations of terms and term descriptions:

- Single term (`<dt>`) with a single definition (`<dd>`).
- Multiple terms (`<dt>`) with a single definition (`<dd>`).
- Multiple definitions (`<dd>`) with a single term (`<dt>`).

Let's take a look at a few examples,

1. Single Term and Description

```
<dl>
  <dt>HTML</dt>
  <dd>HyperText Markup Language</dd>
  <dt>CSS</dt>
  <dd>Cascading Style Sheets</dd>
</dl>
```

Browser Output

```
HTML
  HyperText Markup Language
CSS
  Cascading Style Sheets
```

Here, we can see a single term is followed by a single description.

2. Single Term and Multiple Description

Sometimes, we can come across data where there are multiple descriptors that fit the same term, like a product and its features list.

In such a case, rather than listing the product name multiple times before each feature, we can follow the single term `<dt>` with multiple feature/description `<dd>` to present it better. For example,

```
<dl>  
  <dt>HTML</dt>  
  <dd>HyperText Markup Language.</dd>  
  <dd>Used to create Websites.</dd>  
  <dd>Released in 1993.</dd>  
</dl>
```

Browser Output

```
HTML  
  HyperText Markup Language.  
  Used to create Websites.  
  Released in 1993.
```

Here, we can see that the single term `<dt>HTML</dt>` is described by multiple definitions `<dd>`

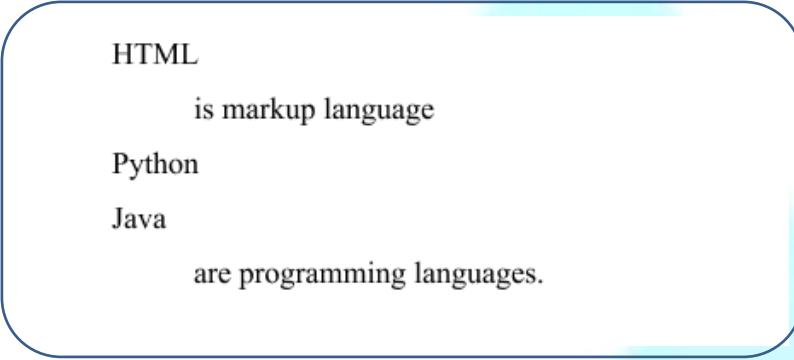
3. Multiple Term and Single Description

Sometimes, we come across data where multiple keys may have similar values. Like multiple programming languages can feature the same feature sets.

In such cases, rather than repeating the key/value pair, we can group several keys `<dt>` followed by a single description `<dd>` such that the single description describes many terms,

```
<dl>  
    <dt>HTML</dt>  
    <dd>is markup language</dd>  
    <dt>Python</dt>  
    <dt>Java</dt>  
    <dd>are programming languages.</dd>  
</dl>
```

Browser Output



```
HTML  
is markup language  
Python  
Java  
are programming languages.
```

Here, we can see that multiple terms `<dt>Python</dt>` and `<dt>Java</dt>` share the same description `<dd>are programming languages.</dd>`.

A description list is the best choice when we want to express the semantically correct format of title-description elements in our HTML. We can also denote the relation between pairs using Description lists.

Inline and Block Elements in HTML

In HTML, elements are categorized into two main types based on their display behavior: inline elements and block elements. Understanding the difference between these two types is essential for creating well-structured and visually appealing web pages.

Inline Elements

Inline elements are those that do not start on a new line and only take up as much width as necessary. They are typically used for formatting small parts of a document, such as text within a paragraph. Inline elements can be nested within block elements, but they cannot contain block elements.

Common Inline Elements:

- **<a> (Anchor):** Used to create hyperlinks. Example: `Link`
- **:** A generic container for inline content. Example: `Red text`
- ** (Image):** Embeds an image in the document. Example: ``
- ** and :** Used for emphasizing text. Example: `Bold text and Italic text`
- **<label>:** Associates a text label with a form control. Example: `<label for="inputId">Label text</label>`

Characteristics of Inline Elements:

- **No Line Breaks:** Inline elements do not start on a new line. Multiple inline elements will appear on the same line, flowing one after another.
- **Width and Height:** The width and height of inline elements are determined by their content. They do not respect the width or height CSS properties.
- **Margin and Padding:** Inline elements only respect horizontal padding and margins, not vertical. This means you can control spacing to the left and right but not above and below.

Usage: Inline elements are ideal for styling and structuring content within a block element. For instance, applying specific styles to a part of a text within a paragraph without disrupting the flow of text.

Block Elements

Block elements, on the other hand, always start on a new line and take up the full width available. They are used to define larger sections of content, structuring the overall layout of the document.

Common Block Elements:

- **<div> (Division):** A generic container for block content. Example: `<div class="container">Content here</div>`
- **<p> (Paragraph):** Defines a paragraph of text. Example: `<p>This is a paragraph.</p>`
- **<h1> to <h6> (Headings):** Define headings, with `<h1>` being the highest level and `<h6>` the lowest. Example: `<h1>Main Heading</h1>`

- **, , and (Lists):** Define unordered and ordered lists, with for list items.
Example: Item 1Item 2
- **<form>:** Represents a form for user input. Example: <form action="/submit" method="post">Form elements here</form>

Characteristics of Block Elements:

- **Line Breaks:** Block elements always start on a new line and occupy the full width of their parent container, forcing any following content to move to the next line.
- **Width and Height:** Block elements can have their width and height explicitly set using CSS properties.
- **Margin and Padding:** Block elements respect all margin and padding settings, allowing for full control over spacing around and within the element.

Usage: Block elements are essential for creating the overall structure and layout of a web page. They help group related content, making it easier to manage and style. For example, using <div> to create sections of a webpage, <p> for paragraphs, and <header> for the top section of a page.

Combining Inline and Block Elements

Inline and block elements can be combined to create complex and well-organized web pages. Typically, block elements are used to create the main structure, while inline elements are used to format content within these blocks. For example:

```
html
<div class="article">
  <h1>Article Title</h1>
  <p>This is a <span style="font-weight: bold;">sample</span> paragraph with an <a href="#">inline link</a>.</p>
  
</div>
```

In this example, <div> and <h1> are block elements that structure the page, while and <a> are inline elements used for styling and linking text within a paragraph.

Class and ID in HTML

In HTML (HyperText Markup Language), classes and IDs are attributes used to identify and style HTML elements. They are crucial for applying CSS (Cascading Style Sheets) and JavaScript functionalities, enabling developers to create well-structured, visually appealing, and interactive web pages. Understanding the differences and appropriate usage of classes and IDs is essential for effective web development.

Class Attribute

The class attribute is used to define a group of HTML elements that share the same styling or behavior. It is a flexible way to apply CSS styles or JavaScript functions to multiple elements simultaneously.

Syntax:

```
html
<tag class="className">Content</tag>
```

Key Points:

- **Reusability:** Classes can be reused across multiple elements on the same web page or across different pages. This makes it easy to apply the same styles or scripts to a large number of elements without repeating code.
- **CSS Usage:** In CSS, classes are referenced with a dot (.) prefix. For example, to style all elements with the class example, you would write:

```
css
.example {
    color: blue;
    font-size: 14px;
}
```

- **JavaScript Usage:** In JavaScript, classes can be accessed using methods like `document.getElementsByClassName("example")` or `document.querySelectorAll(".example")`. For example:

```
javascript
let elements = document.getElementsByClassName("example");
for (let i = 0; i < elements.length; i++) {
    elements[i].style.color = "blue";
}
```

Example:

```
html
<p class="intro">Welcome to our website!</p>
<p class="intro">We hope you enjoy your stay.</p>
<div class="intro">Check out our latest features.</div>
```

In this example, the class intro is applied to a paragraph and a div, allowing both elements to share the same styles or behaviors.

ID Attribute

The id attribute is used to identify a single, unique element on a web page. Each ID must be unique within the HTML document, meaning no two elements should have the same ID.

Syntax:

```
html
<tag id="uniqueID">Content</tag>
```

Key Points:

- **Uniqueness:** IDs are unique to each element, ensuring that each ID refers to one specific element. This is useful for applying styles or behaviors to a single element.

- **CSS Usage:** In CSS, IDs are referenced with a hash (#) prefix. For example, to style an element with the ID main-heading, you would write:

```
css
#main-heading {
    color: red;
    font-size: 24px;
}
```

- **JavaScript Usage:** In JavaScript, IDs can be accessed using methods like document.getElementById(). For example:

```
javascript
let element = document.getElementById("main-heading");
element.style.color = "red";
```

Example:

```
html
<h1 id="main-heading">Welcome to our website!</h1>
<p>Enjoy your visit.</p>
```

In this example, the ID main-heading is applied to an h1 element, making it unique and easily identifiable for styling or scripting purposes.

Differences Between Class and ID

1. Reusability vs. Uniqueness:

- **Class:** Can be used multiple times on different elements within a single HTML document.
- **ID:** Must be unique and used only once per HTML document.

2. CSS and JavaScript Selectors:

- **Class:** Selected with a dot (.) prefix in CSS and can be accessed via getElementsByClassName or querySelectorAll in JavaScript.
- **ID:** Selected with a hash (#) prefix in CSS and can be accessed via getElementById in JavaScript.

3. Specificity:

- **Class:** Lower specificity compared to IDs, making it less powerful in the CSS hierarchy.
- **ID:** Higher specificity, meaning styles applied using IDs can override styles applied using classes if both are used on the same element.

Combined Example:

```

html
Copy code
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Class and ID Example</title>
    <style>
        .highlight {
            background-color: yellow;
        }
        #special-heading {
            color: green;
            font-size: 20px;
        }
    </style>
</head>
<body>
    <h1 id="special-heading" class="highlight">Unique Heading</h1>
    <p class="highlight">This paragraph shares the same highlight class.</p>
    <p>Another paragraph without the class.</p>
</body>
</html>

```

In this example, the `highlight` class is used for multiple elements, while the `special-heading` ID is unique to a single element, demonstrating the appropriate use of both attributes.

HTML Iframes

An HTML iframe is used to display a web page within a web page. HTML iframes offer a powerful way to embed external content, such as videos, maps, or other webpages, directly into your own webpage. An iframe is an HTML document embedded inside another HTML document. The `<iframe>` tag specifies the URL of the embedded content, **allowing for seamless integration of external resources**.

Syntax:

```
<iframe src="url" width="width_value" height="height_value" title="description"></iframe>
```

- `src`: Specifies the URL of the page to embed.
- `width`: Specifies the width of the iframe.
- `height`: Specifies the height of the iframe.
- `title`: Provides an accessible name for the iframe (important for accessibility).

Attributes Value:

It contains a single value URL that specifies the URL of the document that is embedded in the iframe. There are two types of URL links which are listed below:

URL	Descriptions
Absolute URL	It points to another webpage.
Relative URL	It points to other files of the same web page.

Example of iframe

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>HTML Iframe Example</title>
  </head>
  <body>
    <h1>Embedding a Website with an Iframe</h1>
    <iframe      src="https://www.example.com"      width="600"      height="400"
      title="Example Website"></iframe>
  </body>
</html>
```

Browser output:

Embedding a Website with an Iframe

Example Domain

This domain is for use in illustrative examples in documents. You may use this domain in literature without prior coordination or asking for permission.

[More information...](#)

HTML Forms

An HTML form is a section of a document designed to collect user inputs through various interactive controls. These controls include text fields, password fields, checkboxes, radio buttons, submit buttons, and menus, among others. HTML forms are essential for gathering data from site visitors, such as names, email addresses, passwords, and phone numbers, which are then sent to the server for processing.

Why Use HTML Forms?

HTML forms are crucial when there is a need to collect data from users. For example, if a user wants to purchase items online, they must fill out a form with their shipping address and credit/debit card details to ensure the item is sent to the correct address. By utilizing the `<form>` element, HTML forms act as a powerful tool for collecting user input through a variety of interactive controls. These controls range from text fields, numeric inputs, email fields, password fields, to checkboxes, radio buttons, and submit buttons.

In essence, an HTML form serves as a versatile container for numerous input elements, thereby enhancing user interaction. Whether it's for signing up for a newsletter, making a purchase, or providing feedback, HTML forms play a vital role in the user experience on a website.

HTML Form Syntax

```
<form action="server url" method="get|post">  
  //input controls e.g. textfield, textarea, radiobutton, button  
</form>
```

HTML Form Elements

The HTML <form> comprises several elements, each serving a unique purpose. For instance, the <label> element is used to define labels for other <form> elements. The <input> element, on the other hand, is versatile and can be used to capture various types of input data such as text, password, email, and more, simply by altering its type attribute.

List of HTML form elements

Elements	Descriptions
<label>	It defines labels for <form> elements.
<input>	It is used to get input data from the form in various types such as text, password, email, etc by changing its type.
<button>	It defines a clickable button to control other elements or execute a functionality.
<select>	It is used to create a drop-down list.
<textarea>	It is used to get input long text content.
<fieldset>	It is used to draw a box around other form elements and group the related data.
<legend>	It defines a caption for fieldset elements
<datalist>	It is used to specify pre-defined list options for input controls.
<output>	It displays the output of performed calculations.
<option>	It is used to define options in a drop-down list.
<optgroup>	It is used to define group-related options in a drop-down list.

Commonly Used Input Types in HTML Forms

In HTML forms, various input types are used to collect different types of data from users. Here are some commonly used input types:

Input Type	Description
<input type="text">	Defines a one-line text input field
<input type="password">	Defines a password field
<input type="submit">	Defines a submit button
<input type="reset">	Defines a reset button
<input type="radio">	Defines a radio button
<input type="email">	Validates that the input is a valid email address.
<input type="number">	Allows the user to enter a number. You can specify min, max, and step attributes for range.
<input type="checkbox">	Used for checkboxes where the user can select multiple options.
<input type="date">	Allows the user to select a date from a calendar.
<input type="time">	Allows the user to select a time.
<input type="file">	Allows the user to select a file to upload.

Example for HTML Form

```
<!DOCTYPE html>

<html>
<head>
<title>Form in HTML</title>
</head>
<body>
<h2>Registration form</h2>
<form>
<fieldset>
<legend>User personal information</legend>
<label>Enter your full name</label><br>
<input type="text" name="name"><br>
<label>Enter your email</label><br>
<input type="email" name="email"><br>
<label>Enter your password</label><br>
<input type="password" name="pass"><br>
<label>confirm your password</label><br>
<input type="password" name="pass"><br>
<br><label>Enter your gender</label><br>
<input type="radio" id="gender" name="gender" value="male"/>Male <br>
<input type="radio" id="gender" name="gender" value="female"/>Female <br/>
<input type="radio" id="gender" name="gender" value="others"/>others <br/>
<br>Enter your Address:<br>
<textarea></textarea><br>
<input type="submit" value="sign-up">
</fieldset>
</form>
</body>
</html>
```

Browser Output:

Registration form

User personal information

Enter your full name

Enter your email

Enter your password

confirm your password

Enter your gender

Male

Female

others

Enter your Address:

HTML Form Attributes

Attribute	Description
accept-charset	Specifies the character encodings used for form submission
action	Specifies where to send the form-data when a form is submitted
autocomplete	Specifies whether a form should have autocomplete on or off
enctype	Specifies how the form-data should be encoded when submitting it to the server (only for method="post")
method	Specifies the HTTP method to use when sending form-data
name	Specifies the name of the form
novalidate	Specifies that the form should not be validated when submitted
rel	Specifies the relationship between a linked resource and the current document
target	Specifies where to display the response that is received after submitting the form



CSS

Introduction to CSS

Cascading Style Sheets (CSS) is a styling language used to define the presentation and layout of web pages. It allows web developers to control the appearance of web documents (HTML, XML) by specifying styles for various elements on a page. CSS can change the color, size, and style of text, backgrounds, and links. It is indispensable for creating interesting layouts and enhancing user experience.

Key Features of CSS:

1. **Separation of Content and Style:** CSS separates the visual presentation of a webpage from its content. This separation makes it easier to maintain and update web pages, as changes to the style can be made in a single CSS file rather than in multiple HTML files.
2. **Styling Elements:** CSS can be used to style a wide range of HTML elements. For instance, it can modify text attributes (such as font, size, and color), change background colors or images, and style links. This capability helps create visually appealing web pages that are more engaging for users.
3. **Layout Control:** CSS is essential for defining the layout of web pages. It allows developers to create complex layouts using features like flexbox and grid, making it easier to design responsive websites that look good on various screen sizes and devices.
4. **Animations and Transitions:** CSS also supports animations and transitions, which can be used to create dynamic effects on web pages. This includes animating elements, creating hover effects, and adding smooth transitions between states.
5. **Selectors and Specificity:** CSS uses selectors to target HTML elements for styling. Understanding selectors and their specificity is crucial for applying styles correctly and avoiding conflicts between different styles.
6. **Media Queries:** CSS supports media queries, which enable the creation of responsive designs. Media queries allow developers to apply different styles based on the characteristics of the device, such as its width, height, and orientation.

Example Use Cases:

- **Text Styling:** Change the font, size, and color of text to improve readability and match the design of the website.
- **Backgrounds:** Add background colors or images to enhance the visual appeal of sections or the entire page.
- **Navigation:** Create styled navigation bars and sidebars to improve user experience and ease of navigation.
- **Layouts:** Design complex layouts with columns, grids, and flexible boxes to present content in an organized manner.

CSS is an essential component of web development, alongside HTML and JavaScript. HTML provides the structure and content of a webpage, while CSS enhances the visual presentation of that content through various styles. By mastering CSS, developers can create visually stunning and user-friendly websites that provide a better experience for visitors.

CSS Syntax

A CSS rule consists of a selector and a declaration block.

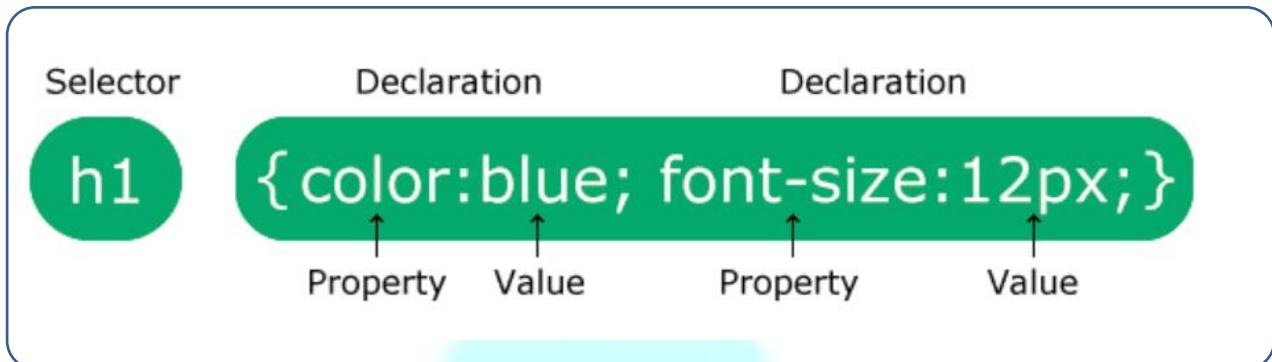


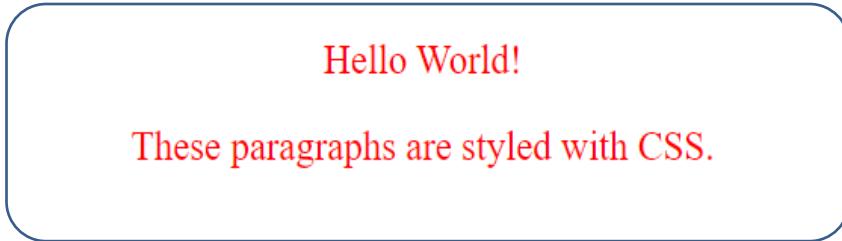
Fig.: Syntax of CSS

- The selector points to the HTML element you want to style.
- The declaration block contains one or more declarations separated by semicolons.
- Each declaration includes a CSS property name and a value, separated by a colon.
- Multiple CSS declarations are separated with semicolons, and declaration blocks are surrounded by curly braces.

Example :

```
<!DOCTYPE html>
<html>
  <head>
    <style>
      p {
        color: red;
        text-align: center;
      }
    </style>
  </head>
  <body>
    <p>Hello World!</p>
    <p>These paragraphs are styled with CSS.</p>
  </body>
</html>
```

Browser Output



CSS Selectors

CSS selectors are used to select the content you want to style. Selectors are the part of CSS rule set. CSS selectors select HTML elements according to its id, class, type, attribute etc.

There are several different types of selectors in CSS.

1. CSS Element Selector
2. CSS Id Selector
3. CSS Class Selector
4. CSS Universal Selector
5. CSS Group Selector
6. CSS Child Selector
7. CSS Descendant selectors
8. CSS Adjacent sibling selectors
9. CSS General sibling selectors
10. CSS Attribute selector

CSS Element Selector:

The element selector selects the HTML element by name.

```
<!DOCTYPE html>
<html>
    <head>
        <style> p{ text-align: center; color: blue; } </style>
    </head>
    <body>
        <p>This style will be applied on every paragraph.</p>
        <p> Me too! </p>
        <p>And me! </p>
    </body>
</html>
```

Browser output

This style will be applied on every paragraph.

Me too!

And me!

CSS Id Selector:

The id selector selects the id attribute of an HTML element to select a specific element. An id is always unique within the page so it is chosen to select a single, unique element.

It is written with the hash character (#), followed by the id of the element.

Lets take an example with the id "para1".

```
<!DOCTYPE html>
<html>
  <head>
    <style> #para1 { text-align: center; color: blue; } </style>
  </head>
  <body>
    <p id="para1">Hello Everyone</p>
    <p>This paragraph will not be affected.</p>
  </body>
</html>
```

Browser Output

Hello Everyone

This paragraph will not be affected.

CSS Class Selector

The class selector selects HTML elements with a specific class attribute. It is used with a period character . (full stop symbol) followed by the class name. A class name should not be started with a number.

Example with a class "center".

```
<!DOCTYPE html>
<html>
<head>
<style>
    .center {
        text-align: center;
        color: blue;
    }
</style>
</head>
<body>
<h1 class="center">This heading is blue and center-aligned.</h1>
<p class="center">This paragraph is blue and center-aligned.</p>
</body>
</html>
```

Browser output:

This heading is blue and center-aligned.

This paragraph is blue and center-aligned.

CSS Class Selector for specific element

If you want to specify that only one specific HTML element should be affected then you should use the element name with class selector. It is also called element class selector.

Let's see an example.

```
<!DOCTYPE html>
<html>
<head>
<style>
    p.center {
        text-align: center;
        color: blue;
    }
</style>
</head>
<body>
<h1 class="center">This heading is not affected</h1>
<p class="center">This paragraph is blue and center-aligned.</p>
</body>
</html>
```

Browser output:

This heading is not affected

This paragraph is blue and center-aligned.

CSS Universal Selector

The universal selector is used as a wildcard character. It selects all the elements on the pages.

```
<!DOCTYPE html>
<html>
<head>
<style>
  * {
    color: green;
    font-size: 20px;
  }
</style>
</head>
<body>
<h2>This is heading</h2>
<p>This style will be applied on every paragraph.</p>
<p>Me too!</p>
<h3>And me!</h3>
</body>
</html>
```

Browser Output:

This is heading

This style will be applied on paragraph also.

Me too!

And me!

CSS Group Selector

The grouping selector is used to select all the elements with the same style definitions.

Grouping selector is used to minimize the code. Commas are used to separate each selector in grouping.

Let's see the CSS code without group selector.

```
h1 {  
    text-align: center;  
    color: blue;  
}  
  
h2 {  
    text-align: center;  
    color: blue;  
}  
  
p {  
    text-align: center;  
    color: blue;  
}
```

As you can see, you need to define CSS properties for all the elements. It can be grouped in following ways:

```
h1,h2,p {  
    text-align: center;  
    color: blue;  
}
```

Let's see the full example of CSS group selector.

```
<!DOCTYPE html>
<html>
<head>
<style>
    h1, h2, p {
        text-align: center;
        color: blue;
    }
</style>
</head>
<body>
<h1> Embedded Full Stack IoT </h1>
<h2> Embedded Full Stack IoT (In smaller font)</h2>
<p>This is a paragraph.</p>
</body>
</html>
```

Browser Output:

Embedded Full Stack IoT

Embedded Full Stack IoT (In smaller font)

This is a paragraph.

CSS Child Selector:

The child selector selects all elements that are the children of a specified element. It gives the relation between two elements. The element > element selector selects those elements which are the children of the specific parent. The operand on the left side of > is the parent and the operand on the right is the children element.

```
<!DOCTYPE html>
<html>
<head>
<style>
    div > p {
        background-color: yellow;
    }
</style>
</head>
<body>
<h2>Child Selector</h2>
<p>The child selector (>) selects all elements that are the children of a specified element.</p>
<div>
    <p>Paragraph 1 in the div.</p>
    <p>Paragraph 2 in the div.</p>
    <section>
        <!-- not Child but Descendant -->
        <p>Paragraph 3 in the div (inside a section element).</p>
    </section>
    <p>Paragraph 4 in the div.</p>
</div>
<p>Paragraph 5. Not in a div.</p>
<p>Paragraph 6. Not in a div.</p>
</body>
</html>
```

Browser output:

Child Selector

The child selector (>) selects all elements that are the children of a specified element.

Paragraph 1 in the div.

Paragraph 2 in the div.

Paragraph 3 in the div (inside a section element).

Paragraph 4 in the div.

Paragraph 5. Not in a div.

Paragraph 6. Not in a div.

CSS Adjacent sibling selectors

The CSS adjacent sibling selector is used to select the adjacent sibling of an element. Sibling elements must have the same parent element, and "adjacent" means "immediately following". It is used to select only those elements which immediately follow the first selector. The + sign is used as a separator. For example, the direct next element is selected here with the adjacent sibling selector concept

```
<!DOCTYPE html>
<html>
<head>
<style>
    div + p {
        background-color: yellow;
    }
</style>
</head>
<body>
<h2>Adjacent Sibling Selector</h2>
<p>The + selector is used to select an element that is directly after another specific element.</p>
<p>The following example selects the first p element that are placed immediately after div elements:</p>
<div>
    <p>Paragraph 1 in the div.</p>
    <p>Paragraph 2 in the div.</p>
</div>
<p>Paragraph 3. After a div.</p>
<p>Paragraph 4. After a div.</p>
<div>
    <p>Paragraph 5 in the div.</p>
    <p>Paragraph 6 in the div.</p>
</div>
<p>Paragraph 7. After a div.</p>
<p>Paragraph 8. After a div.</p>
</body>
</html>
```

Browser output

Adjacent Sibling Selector

The + selector is used to select an element that is directly after another specific element.

The following example selects the first p element that are placed immediately after div elements:

Paragraph 1 in the div.

Paragraph 2 in the div.

Paragraph 3. After a div.

Paragraph 4. After a div.

Paragraph 5 in the div.

Paragraph 6 in the div.

Paragraph 7. After a div.

Paragraph 8. After a div.

Descendant Selector

The descendant selector matches all elements that are descendants of a specified element. Descendant selector is used to select all the elements which are child of the element (not a specific element). It selects the elements inside the elements i.e it combines two selectors such that elements matched by the second selector are selected if they have an ancestor element matching the first selector.

```
<!DOCTYPE html>

<html>
<head>
<style>
    div p {
        background-color: yellow;
    }
</style>
</head>
<body>
<h2>Descendant Selector</h2>
<p>The descendant selector matches all elements that are descendants of a specified element.</p>
    <div>
        <p>Paragraph 1 in the div.</p>
        <p>Paragraph 2 in the div.</p>
        <section> <p>Paragraph 3 in the div.</p> </section>
    </div>
<p>Paragraph 4. Not in a div.</p>
<p>Paragraph 5. Not in a div.</p>
</body>
</html>
```

Browser Output:

Descendant Selector

The descendant selector matches all elements that are descendants of a specified element.

Paragraph 1 in the div.

Paragraph 2 in the div.

Paragraph 3 in the div.

Paragraph 4. Not in a div.

Paragraph 5. Not in a div.

CSS Attribute Selectors

These are used to style HTML elements with specific attributes. It is possible to style HTML elements that have specific attributes or attribute values. The [attribute] selector is used to select elements with a specified attribute.

```
<!DOCTYPE html>
<html>
<head>
<style>
    a[target] {
        background-color: yellow;
    }
</style>
</head>
<body>
<h2>CSS [attribute] Selector</h2>
<p>The links with a target attribute gets a yellow background:</p>
<a href="https://msksolutions.in/">msksolutions.in</a>
<a href="http://www.disney.com" target="_blank">disney.com</a>
<a href="http://www.wikipedia.org" target="_top">wikipedia.org</a>
</body>
</html>
```

Browser output:

CSS [attribute] Selector

The links with a target attribute gets a yellow background:

msksolutions.in [disney.com](http://www.disney.com) [wikipedia.org](http://www.wikipedia.org)

CSS General sibling selectors

The general sibling selector selects all elements that are next siblings of a specified element. The general sibling selector is used to select the element that follows the first selector element and also shares the same parent as the first selector element. This can be used to select a group of elements that share the same parent element.

```
<!DOCTYPE html>
<html>
<head>
<style>
    div ~ p {
        background-color: yellow;
    }
</style>
</head>
<body>
<h2>General Sibling Selector</h2>
<p>The general sibling selector (~) selects all elements that are next siblings of a specified element.</p>
<p>Paragraph 1.</p>
<div>
    <p>Paragraph 2.</p>
</div>
<p>Paragraph 3.</p>
<code>Some code.</code>
<p>Paragraph 4.</p>
</body>
</html>
```

Browser Output:

General Sibling Selector

The general sibling selector (~) selects all elements that are next siblings of a specified element.

Paragraph 1.

Paragraph 2.

Paragraph 3.

Some code.

Paragraph 4.

Types of CSS (Cascading Style Sheet)

Cascading Style Sheets (CSS) is a language used to style web pages that contain HTML elements, defining how elements are displayed on webpages, including layout, colors, fonts, and other properties of the elements on a web page. CSS works by targeting HTML elements and applying style rules to define how they should be displayed, including properties like color, size, layout, and positioning.

There are three types of CSS which are given below:

1. Inline CSS
2. Internal or Embedded CSS
3. External CSS

Inline CSS

Inline CSS is a method of applying styling directly to individual HTML elements using the “style” attribute within the HTML tag, allowing for specific styling of individual elements within the HTML document, overriding any external or internal styles.

Example for Inline CSS

```
<!DOCTYPE html>  
  
<html>  
  
<body>  
  
    <h1 style="color:blue; text-align:center;">This is a heading</h1>  
  
    <p style="color:red;">This is a paragraph.</p>  
  
</body>  
  
</html>
```

Browser Output:

This is a heading

This is a paragraph.

Internal CSS

Internal or Embedded CSS:

Internal or Embedded CSS is defined within the HTML document’s `<style>` element. It applies styles to specified HTML elements. The CSS rule set should be within the HTML file in the head section i.e. the CSS is embedded within the `<style>` tag inside the head section of the HTML file.

```
<!DOCTYPE html>

<html>
<head>
<style>

    body {
        background-color: linen;
    }

    h1 {
        color: maroon;
        margin-left: 40px;
    }

</style>
</head>
<body>
<h1>This is a heading</h1>
<p>This is a paragraph.</p>
</body>
</html>
```

Browser Output:

This is a heading

This is a paragraph.

External CSS

External CSS contains separate CSS files that contain only style properties with the help of tag attributes (For example class, id, heading, ... etc). CSS property is written in a separate file with a .css extension and should be linked to the HTML document using a link tag. It means that, for each element, style can be set only once and will be applied across web pages.

- An external style sheet can be written in any text editor, and must be saved with a .css extension.
- The external .css file should not contain any HTML tags.

"mystyle.css"

```

body {
    background-color: lightblue;
}

h1 {
    color: navy;
    margin-left: 20px;
}

```

Example for external CSS

```

<!DOCTYPE html>

<html>
    <head>
        <link rel="stylesheet" href="mystyle.css">
    </head>
    <body>
        <h1>This is a heading</h1>
        <p>This is a paragraph.</p>
    </body>
</html>

```

Browser output

This is a heading

This is a paragraph.

External style sheets have the following advantages over internal and inline styles:

- one change to the style sheet will change all linked pages
- you can create classes of styles that can then be used on many different HTML elements
- consistent look and feel across multiple web pages
- improved load times because the css file is downloaded once and applied to each relevant page as needed

Differences between Inline, Internal, and External CSS:

Feature	Inline CSS	Internal CSS	External CSS
Location	It is used within HTML tag using the style attribute.	It is used within <head> section of HTML document.	It is used in a separate .css file.
Selector Scope	Affects a single element or a group of elements.	Affects multiple elements within the same HTML element.	Affects multiple HTML documents or an entire website.
Reusability	Not reusable. Styles need to be repeated for each element.	Can be reused on multiple elements within the same HTML document.	Can be reused on multiple HTML documents or an entire website.
Priority	Highest priority. Overrides internal and external styles.	Medium priority. Overrides external styles but can be overridden by inline styles.	Lowest priority. Can be overridden by both inline and internal styles.
File Size	Inline styles increase the HTML file size, which can affect the page load time.	Internal styles are part of the HTML file, which increases the file size.	External styles are in a separate file, which reduces the HTML file size and can be cached for faster page loads.
Maintainability	Not easy to maintain. Changes need to be made manually to each element.	Relatively easy to maintain. Changes need to be made in one place in the <head> section.	Easiest to maintain. Changes need to be made in one place in the external .css file

CSS Background

CSS background property is used to define the background effects on element. There are 8 CSS background properties that affects the HTML elements:

1. background-color
2. background-image
3. background-position
4. background-size
5. background-repeat
6. background-origin
7. background-clip
8. background-attachment

CSS background-color:

The background-color property sets the background color of an element. The background of an element is the total size of the element, including padding and border (but not the margin).

Example for CSS background-color:

```
<!DOCTYPE html>
<html>
<head>
<style>
body {
    background-color: coral;
}
</style>
</head>
<body>
<h1>The background-color Property</h1>
<p>Here the background color can be specified with a color name coral.</p>
</body>
</html>
```

Browser output:

The background-color Property

Here the background color can be specified with a color name coral.

CSS background-image

The background-image property is used to set an image as a background of an element. The background-image property sets one or more background images for an element. By default, a background-image is placed at the top-left corner of an element, and repeated both vertically and horizontally. **The background of an element is the total size of the element, including padding and border (but not the margin).**

```
<!DOCTYPE html>
<html>
<head>
<style>
    body {
        background-image: url("paper.gif");
        background-color: #cccccc;
    }
</style>
</head>
<body>
    <h1>The background-image Property</h1>
    <p>Hello World!</p>
</body>
</html>
```

Browser output:

The background-image Property

Hello World!

CSS background-repeat

The background-repeat property sets if/how a background image will be repeated. By default, a background-image is repeated both vertically and horizontally. **The background image is placed according to the background-position property. If no background-position is specified, the image is always placed at the element's top left corner.**

Example for background-repeat: repeat-y;

```
<!DOCTYPE html>
<html>
<head>
<style>
    body {
        background-image: url("paper.gif");
        background-repeat: repeat-y;
    }
</style>
</head>
<body>
<h1>The background-repeat Property</h1>
<p>Here, the background image is repeated only vertically.</p>
</body>
</html>
```

Browser Output:

The background-repeat Property

Here, the background image is repeated only vertically.

Example for background-repeat: repeat-x;

```
<!DOCTYPE html>
<html>
<head>
<style>
    body {
        background-image: url("paper.gif");
        background-repeat: repeat-x;
    }
</style>
</head>
<body>
<h1>The background-repeat Property</h1>
<p>Here, the background image is repeated only horizontally.</p>
</body>
</html>
```

Browser output:

The background-repeat Property

Here, the background image is repeated only horizontally.

CSS background-attachment

The background-attachment property is used to specify if the background image is fixed or scroll with the rest of the page in browser window. If you set fixed the background image then the image will not move during scrolling in the browser.

Example with fixed background image.

```
body {  
    background-image: url("img_tree.gif");  
    background-repeat: no-repeat;  
    background-attachment: fixed;  
}
```

Example with fixed background image.

```
body {  
    background-image: url("img_tree.gif");  
    background-repeat: no-repeat;  
    background-attachment: fixed;  
}
```

CSS background-position

The background-position property sets the starting position of a background image. By default, the background image is placed on the top-left of the webpage.

You can set the following positions:

1. center
2. top
3. bottom
4. left
5. right

Example for background-position: center;

```
<!DOCTYPE html>
<html>
<head>
<style>
body {
    background-image: url('w3css.gif');
    background-repeat: no-repeat;
    background-attachment: fixed;
    background-position: center;
}
</style>
</head>
<body>
<h1>The background-position Property</h1>
<p>Here, the background image will be positioned in the center of the element (in this case, the body element).</p>
</body>
</html>
```

Browser Output:

The background-position Property

Here, the background image will be positioned in the center of the element (in this case, the body element).



CSS background-size

The background-size property specifies the size of the background images.

```
#example1 {  
    border: 2px solid black;  
    padding: 25px;  
    background: url(mountain.jpg);  
    background-repeat: no-repeat;  
    background-size: auto;  
}
```

```
#example2 {  
    border: 2px solid black;  
    padding: 25px;  
    background: url(mountain.jpg);  
    background-repeat: no-repeat;  
    background-size: 300px 100px;  
}
```

CSS background-origin

The background-origin property specifies the origin position (the background positioning area) of a background image. This property has no effect if background-attachment is "fixed".

```
#example1 {  
    border: 10px dashed black;  
    padding: 25px;  
    background: url(paper.gif);  
    background-repeat: no-repeat;  
background-origin: padding-box;  
}  
  
#example2 {  
    border: 10px dashed black;  
    padding: 25px;  
    background: url(paper.gif);  
    background-repeat: no-repeat;  
background-origin: border-box;  
}  
  
#example3 {  
    border: 10px dashed black;  
    padding: 25px;  
    background: url(paper.gif);  
    background-repeat: no-repeat;  
background-origin: content-box;  
}
```

CSS background-clip

The background-clip property defines how far the background (color or image) should extend within an element.

```
#example1 {  
    border: 10px dotted black;  
    padding: 15px;  
    background: lightblue;  
    background-clip: border-box;  
}
```

```
#example2 {  
    border: 10px dotted black;  
    padding: 15px;  
    background: lightblue;  
    background-clip: padding-box;  
}
```

```
#example3 {  
    border: 10px dotted black;  
    padding: 15px;  
    background: lightblue;  
    background-clip: content-box;  
}
```

Example of CSS background Properties

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>CSS Background Property Example</title>
    <style>
        .background-example {
            /* Background shorthand property */
            background:
                #ffffff          /* background-color */ 
                url('example.jpg')      /* background-image */ 
                no-repeat          /* background-repeat */ 
                fixed              /* background-attachment */ 
                center center / cover /* background-position / background-size */ 
                border-box          /* background-origin */ 
                content-box          /* background-clip */ 
                rgba(255, 255, 255, 0.8); /* background-blend-mode */ 
                width: 300px;
                height: 300px;
                border: 1px solid #000;
        }
    </style>
</head>
<body>
    <div class="background-example"></div>
</body>
</html>
```

CSS Borders

CSS borders are essential elements in websites, representing the edges of various components and elements. CSS Borders refer to the lines that surround elements, defining their edges. Borders can be styled, colored, and sized using CSS properties such as border style, border color, border width, and border radius. borders can be styled with the top border, the right border, the bottom border, and the left border.

CSS provides several properties to customize borders:

1. **border-style:** Determines the type of border (e.g., solid, dashed, dotted).
2. **border-width:** Sets the width of the border (in pixels, points, or other units).
3. **border-color:** Specifies the border color.
4. **border-radius:** Creates rounded corners for elements

Ways to Style Border in CSS

The CSS border property enables the styling of an element's border by setting its width, style, and color, allowing for customizable visual boundaries in web design.

1. Border Style

- CSS border-top style Property
- border-right-style Property
- border-bottom-style Property
- border-left-style Property

2. Border Width

1. border-top-width Property
2. border-right-width Property
3. border-bottom-width Property
4. border-left-width Property

3. Border Color

- border-top-color Property
- border-right-color Property
- border-bottom-color Property
- border-left-color Property

4. Border individual sides

5. Border radius property

Common Border Styles

The border-style property specifies the type of border. None of the other border properties will work without setting the border style.

Following are the types of borders:

- **Dotted**: Creates a series of dots.
- **Dashed**: Forms a dashed line.
- **Solid**: Produces a continuous line.
- **Double**: Renders two parallel lines.
- **Groove** and **Ridge**: Create 3D grooved and ridged effects.
- **Inset** and **Outset**: Add 3D inset and outset borders.
- **None**: Removes the border.
- **Hidden**: Hides the border.

Examples of CSS border Style

```

<!DOCTYPE html>
<html>
<head>
<style>
  p.dotted {border-style: dotted; }

  p.dashed {border-style: dashed; }

  p.solid { border-style: solid; }

  p.double {border-style: double; }

</style>
</head>

<body>
  <h2>The border-style Property</h2>

  <p class="dotted">A dotted border.</p>

  <p class="dashed">A dashed border.</p>

  <p class="solid">A solid border.</p>

  <p class="double">A double border.</p>

</body>
</html>

```

Browser Output:

The border-style Property

A dotted border.

A dashed border.

A solid border.

A double border.

CSS Border Width

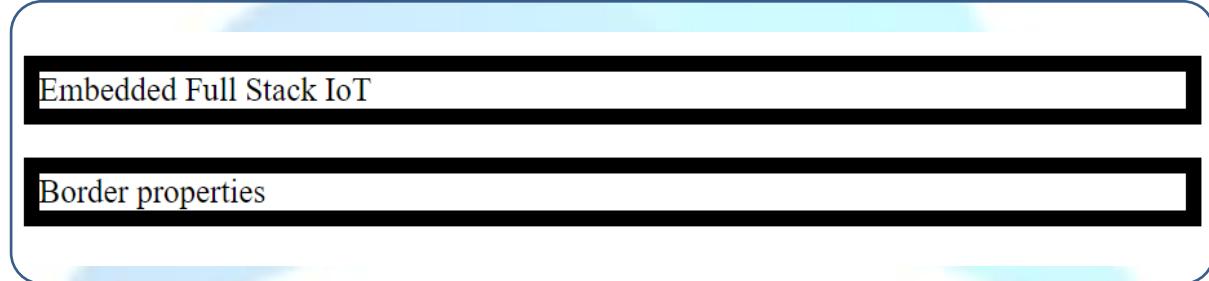
Border width sets the width of the border. The width of the border can be in px, pt, cm or thin, medium, and thick.

Example of Border Width

```
<!DOCTYPE html>
<html>
<head>
<style>
p {
    border-style: solid;
    border-width: 8px;
}
</style>
</head>

<body>
<p> Embedded Full Stack IoT </p>
<p> Border properties </p>
</body>
</html>
```

Browser Output:



Embedded Full Stack IoT

Border properties

CSS Border Color

This property is used to set the color of the border. Color can be set using the color name, hex value, or RGB value. If the color is not specified border inherits the color of the element itself.

Example of CSS Border Color

```
<!DOCTYPE html>
<html>

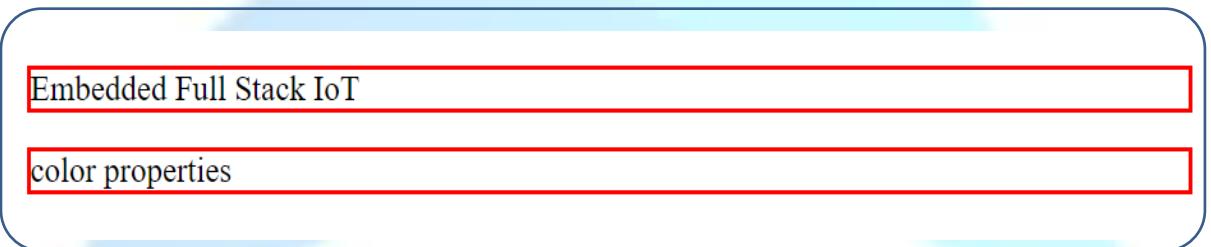
<head>
  <style>
    p {
      border-style: solid;
      border-color: red
    }
  </style>
</head>

<body>
  <p> Embedded Full Stack IoT </p>
  <p> color properties </p>

</body>

</html>
```

Browser Output:



Embedded Full Stack IoT

color properties

CSS Border individual sides:

Using border property, we can provide width, style, and color to all the borders separately for that we have to give some values to all sides of the border.

Example for CSS Border individual sides

```
<!DOCTYPE html>

<html>
  <head>
    <style>
      p {
        border-top-style: dotted;
        border-right-style: solid;
        border-bottom-style: dotted;
        border-left-style: solid;
      }
    </style>
  </head>
  <body>
    <h2>Individual Border Sides</h2>
    <p>2 different border styles.</p>
  </body>
</html>
```

Browser Output:

Individual Border Sides

[2 different border styles.]

Border radius property

The CSS border-radius property rounds the corners of an element's border, creating smoother edges, with values specifying the curvature radius.

Example of Border radius property

```
<!DOCTYPE html>
<html>
<head>
<style>
    h1 {
        border-style: solid;
        text-align: center;
        background: green;
        border-radius: 20px;
    }
</style>
</head>
<body>
    <h1>Embedded Full Stack IoT</h1>
</body>
</html>
```

Browser Output:



Embedded Full Stack IoT

CSS Margin

Margins, as defined by the CSS margin property, are the spaces created around an element, setting it apart from its neighboring elements. These margins can be individually set for each side – top, right, bottom, and left. The values for these margins can be specified in lengths (e.g., px, rem, em, ex, vh, vw, etc.), percentages (relative to the element's width), or auto (calculated by the browser). Interestingly, margins also allow negative values.

Margin Values

- Length (e.g., px, rem, em, ex, vh, vw, etc)
- Percentage (relative to the element's width)
- auto (calculated by the browser)
- margin allows negative values.

Margin Properties

- **margin-top:** Sets the top margin of an element.
- **margin-right:** Sets the right margin of an element.
- **margin-bottom:** Specifies the margin at the bottom of an element.
- **margin-left:** Determines the width of the margin on the left side of an element.

Syntax:

```
body {  
    margin: value;  
}
```

Example of margin property with 4 values:

margin: 40px 100px 120px 80px;

- top margin = 40px
- right margin = 100px
- bottom margin = 120px
- left margin = 80px

```
<!DOCTYPE html>  
  
<html>  
  
<head>  
  
    <style>  
  
        p {  
  
            margin: 80px 100px 50px 80px;  
  
        }  
  
    </style>  
  
</head>  
  
<body>  
  
    <h1> Embedded Full Stack IoT </h1>  
  
    <p> Margin properties </p>  
  
</body>  
  
</html>
```

Browser Output:

Embedded Full Stack IoT

Margin properties

Example of margin property with 3 values:

margin: 40px 100px 120px;

- top = 40px
- right and left = 100px
- bottom = 120px

```
<!DOCTYPE html>

<html>
  <head>
    <style>
      p { margin: 80px 50px 100px; }
    </style>
  </head>
  <body>
    <h1> Embedded Full Stack IoT </h1>
    <p> Margin properties </p>
  </body>
</html>
```

Browser output:

Embedded Full Stack IoT

Margin properties

Example of margin property with 2 values:

margin: 40px 100px;

- top and bottom = 40px;
- left and right = 100px;

```
<!DOCTYPE html>

<html>
<head>
<style>
    P{ margin: 100px 150px; }
</style>
</head>
<body>
    <h1> Embedded Full Stack IoT </h1>
    <p> Margin properties </p>
</body>
</html>
```

Browser Output:

Embedded Full Stack IoT

Margin properties

Example of margin property with 1 value:

margin: 40px;

- top, right, bottom and left = 40px

```
<!DOCTYPE html>  
  
<html>  
  
<head>  
  
<style>  
  
    p { margin: 40px; }  
  
</style>  
  
</head>  
  
<body>  
  
    <h1> Embedded Full Stack IoT </h1>  
  
    <p> Margin properties </p>  
  
</body>  
  
</html>
```

Browser Output:

Embedded Full Stack IoT

Margin properties

CSS Padding

CSS Padding property is used to create space between the element's content and the element's border. It only affects the content inside the element.

CSS padding is different from CSS margin as the margin is the space between adjacent element borders and padding is the space between content and element's border.

We can independently change the top, bottom, left, and right padding using padding properties.
CSS Padding Properties

CSS provides properties to specify padding for individual sides of an element which are defined as follows:

- **padding-top:** Sets the padding for the top side of the element.
- **padding-right:** Sets the padding for the right side of the element.
- **padding-bottom:** Sets the padding for the bottom side of the element.
- **padding-left:** Sets the padding for the left side of the element.

Padding properties can have the following padding values:

- Length- in cm, px, pt, etc.
- Width- % width of the element.
- inherit- inherit padding from the parent element

Syntax:

```
.myDiv {
    padding-top: 80px;
    padding-right: 100px;
    padding-bottom: 50px;
    padding-left: 80px;
}
```

Shorthand Property for Padding in CSS

The Shorthand Padding Property in CSS allows you to set the padding on all sides (top, right, bottom, left) of an element in a single line with some combinations, so we can easily apply padding to our targeted element.

There are four cases while using shorthand property:

1. If the padding property has one value.
2. If the padding property contains two values.
3. If the padding property contains three values.
4. If the padding property contains four values.

CSS Shorthand Padding Property for One Value:

If the padding property has one value, then it applies padding to all sides of an element.

For example padding: 20px applies 20 pixels of padding to all sides equally.

Syntax:

```
.element {  
    /* Applies 20px padding to all sides */  
    padding: 20px;  
}
```

CSS Shorthand Padding Property for Two Values:

If the padding property contains two values, then the first value applies to the top and bottom padding, and the second value applies to the right and left padding.

For Example – padding: 10px 20px i.e. top and bottom padding are 10px while right and left padding is 20px.

Syntax:

```
.element {  
    /* Applies 10px padding to top and bottom,  
    20px padding to right and left */  
    padding: 10px 20px;  
}
```

CSS Shorthand Padding Property for Three Values

If the padding property contains three values, then the first value sets the top padding, the second value sets the right and left padding, and the third value sets the bottom padding.

For Example – padding: 10px 20px 30px;

- top padding is 10px
- right and left padding is 20px
- bottom padding is 30px

Syntax:

```
.element {  
    /* Applies 10px padding to top,  
    20px padding to right and left,  
    30px padding to bottom */  
    padding: 10px 20px 30px;  
}
```

CSS Shorthand Padding Property Having Four Values

If the padding property contains four values, then the first value sets the top padding, the second value sets the right padding, the third value sets the bottom padding, and the fourth value sets the left padding.:.

For Example – padding: 10px 20px 15px 25px;

- top padding is 10px
- right padding is 5px
- bottom padding is 15px
- left padding is 20px

Syntax:

```
.element {  
    /* Applies 10px padding to top,  
     20px padding to right,  
     15px padding to bottom,  
     and 25px padding to left */  
    padding: 10px 20px 15px 25px;  
}
```

Example for the padding shorthand property with four values:

```
<!DOCTYPE html>

<html>
<head>
<style>
    div {
        border: 1px solid black;
        padding: 25px 50px 75px 100px;
        background-color: lightblue;
    }
</style>
</head>
<body>
<h2>The padding shorthand property - 4 values</h2>
<div>This div element has a top padding of 25px, a right padding of 50px, a bottom padding of 75px, and a left padding of 100px.</div>
</body>
</html>
```

Browser Output:

The padding shorthand property - 4 values

This div element has a top padding of 25px, a right padding of 50px, a bottom padding of 75px, and a left padding of 100px.

CSS Text

CSS Text Formatting refers to applying styles to text elements to control appearance and layout. This includes properties for color, alignment, decoration, indentation, justification, shadows, spacing, and direction. These properties enhance readability and aesthetics, improving the presentation of textual content on web pages.

Syntax:

The Syntax to write this property:

```
Selector {  
    property-name : /*value*/  
}
```

CSS Text Formatting Properties:

These are the following text formatting properties.

Property	Description
text-color	Sets the color of the text using color name, hex value, or RGB value.
text-align	Specifies horizontal alignment of text in a block or table-cell element.
text-align-last	Sets alignment of last lines occurring in an element.
text-decoration	Decorates text content.
text-decoration-color	Sets color of text decorations like overlines, underlines, and line-throughs.
text-decoration-line	Sets various text decorations like underline, overline, line-through.
text-decoration-style	Combines text-decoration-line and text-decoration-color properties.
text-indent	Indents first line of paragraph.
text-justify	Justifies text by spreading words into complete lines.

Property	Description
text-overflow	Specifies hidden overflow text.
text-transform	Controls capitalization of text.
text-shadow	Adds shadow to text.
letter-spacing	Specifies space between characters of text.
line-height	Sets space between lines.
direction	Sets text direction.
word-spacing	Specifies space between words of line.

Example for CSS Text Formatting

In this example we demonstrates basic text formatting using CSS. It includes paragraphs with different styles: changing color, aligning center, adding underline, setting indentation, and adjusting letter spacing for improved readability and aesthetics.

```
<!DOCTYPE html>

<html lang="en">

<head>

    <meta charset="UTF-8">

    <meta name="viewport" content="width=device-width, initial-scale=1.0">

    <title>Basic Text Formatting</title>

    <style>

        .text-color {color: blue;}

        .text-align-center {text-align: center;}

        .text-decoration {text-decoration: underline;}

        .text-indent {text-indent: 20px;}

        .letter-spacing {letter-spacing: 2px;}

    </style>

</head>

<body>

    <p class="text-color">Changing Text Color</p>

    <p class="text-align-center">Aligning Text</p>

    <p class="text-decoration">Adding Text Decoration</p>

    <p class="text-indent">Setting Text Indentation</p>

    <p class="letter-spacing">Adjusting Letter Spacing</p>

</body>

</html>
```

Browser Output:

[Changing Text Color](#)

[Aligning Text](#)

[Adding Text Decoration](#)

[Setting Text Indentation](#)

[Adjusting Letter Spacing](#)

Example for CSS text advanced properties

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Advanced Text Formatting</title>
  <style>
    .line-height {line-height: 1.5;}
    .text-shadow {text-shadow: 2px 2px 4px rgba(0, 0, 0, 0.5);}
    .text-transform {text-transform: uppercase;}
    .word-spacing {word-spacing: 5px;}
    .text-direction {direction: rtl;}
  </style>
</head>
<body>
  <p class="line-height">Changing Line Height</p>
  <p class="text-shadow">Applying Text Shadow</p>
  <p class="text-transform">Controlling Text Transformation</p>
  <p class="word-spacing">Setting Word Spacing</p>
  <p class="text-direction">Specifying Text Direction</p>
</body> </html>
```

Browser output:

Changing Line Height

Applying Text Shadow

CONTROLLING TEXT TRANSFORMATION

Setting Word Spacing

Specifying Text Direction

CSS Fonts

CSS fonts offer a range of options to style the text content within HTML elements. It gives you the ability to control different aspects of fonts, including font family, font size, font weight, font style, and more.

There are many font properties in CSS which are mentioned and briefly discussed below:

- **CSS font-family Property:** The font-family property specifies the font of an element.
- **CSS font-style Property:** If we want to give design to any type of text then we can make use of CSS font-style property.
- **CSS font-weight Property:** The font-weight property of the CSS is used to set the weight or thickness of the font being used with the HTML Text.
- **CSS font-variant Property:** The font-variant property is used to convert all lowercase letters into uppercase letters.
- **CSS font-size Property:** The font-size property in CSS is used to set the font size of the text in an HTML document.
- **CSS font-stretch Property:** The font-stretch property in CSS is used to set the text wider or narrower.
- **CSS font-kerning Property:** This property is used to control the usage of the Kerning Information that has been stored in the Font.

Example for few CSS Font properties.

```
<!DOCTYPE html>
<html>
<head>
<title>CSS Font</title>
<style>
.iot {
    font-family: "Arial, Helvetica, sans-serif";
    font-size: 60px;
    color: #090;
    text-align: center;
}
.eiot {
    font-family: "Comic Sans MS", cursive, sans-serif;
    font-variant:small-caps;
    text-align: center;
}
</style>
</head>
<body>
<div class="iot">Internet of Things</div>
<div class="eiot">Embedded Full Stack IoT</div>
</body>
</html>
```

Browser Output:



Internet of Things
EMBEDDED FULL STACK IoT

Examples for CSS Font Collection :

font-family: It is used to set the font type of an HTML element. It holds several font names as a fallback system.

Syntax: `font-family: "font family name";`

```
<!DOCTYPE html>
<html>
<head>
    <title>CSS Font</title>
    <style>
        .iot {
            font-family: "Times New Roman";
            font-weight: bold;
            font-size: 40px;
            color: #090;
            text-align: center;
        }
        .eiot {
            font-family: "Comic Sans MS", cursive, sans-serif;
            text-align: center;
        }
    </style>
</head>
<body>
    <div class="iot">Internet of Things</div>
    <div class="eiot">Embedded Full Stack IoT</div>
</body>
</html>
```

Browser Output:

Internet of Things

Embedded Full Stack IoT

Examples for CSS Font Style Collection:

font-style: It is used to specify the font style of an HTML element. It can be “normal, italic or oblique”.

Syntax: `font-style: style name;`

```
<!DOCTYPE html>
<html>
<head>
<title>CSS Font</title>
<style>
    .iot {
        font-style: normal;
        font-family: "Times New Roman";
        font-weight: bold;
        font-size: 40px;
        color: #00f;
        text-align: center; }

    .eiot {
        font-style: italic;
        text-align: center; }

</style> </head>
<body>
    <div class="iot">Internet of Things</div>
    <div class="eiot">Embedded Full Stack IoT</div>
</body></html>
```

Browser Output:

Internet of Things

Embedded Full Stack IoT

Examples for CSS Font Weight Collection:

font-weight: It is used to set the boldness of the font. Its value can be “normal, bold, lighter, bolder”.

Syntax: **font-weight: font weight value;**

```
<!DOCTYPE html>
<html>
<head>
    <title>CSS Font</title>
    <style>
        .iot { font-weight: bold;
            font-style: normal;
            font-family: "Times New Roman";
            font-size: 40px;
            color: #f00;
            text-align: center; }

        .eiot { font-weight: normal;
            text-align: center; }

    </style>
</head>
<body>
    <div class="iot">Internet of Things</div>
    <div class="eiot">Embedded Full Stack IoT</div>
</body>
</html>
```

Browser Output:

Internet of Things

Embedded Full Stack IoT

Examples for CSS Font Variant Collection:

font-variant: It is used to create the small-caps effect. It can be “normal or small-caps”.

Syntax: `font-variant: font variant value;`

```
<!DOCTYPE html>
<html>
<head>
<title>CSS Font</title>
<style>
    .iot {
        font-variant: small-caps;
        font-weight: bold;
        font-family: "Times New Roman";
        font-size: 40px;
        color: #800080;
        text-align: center; }

    .eiot {
        font-variant: normal;
        text-align: center; }

</style>
</head>
<body>
    <div class="iot">Internet of Things</div>
    <div class="eiot">Embedded Full Stack IoT</div>
</body>
</html>
```

Browser Output:

INTERNET OF THINGS

Embedded Full Stack IoT

Examples for CSS Font Size Collection:

font-size: It is used to set the font size of an HTML element. The font-size can be set in different ways like in “pixels, percentage, em or we can set values like small, large” etc.

Syntax: `font-size: font size value;`

```
<!DOCTYPE html>
<html>
<head>
<title>CSS Font</title>
<style>
    .iot{
        font-size: 40px;
        font-weight: bold;
        font-family: "Times New Roman";
        color: #800000;
        text-align: center; }

    .eiot {
        font-size: 1.2em;
        text-align: center; }

</style> </head>
<body>
    <div class="iot">Internet of Things</div>
    <div class="eiot">Embedded Full Stack IoT</div>
</body>
</html>
```

Browser Output:

Internet of Things

Embedded Full Stack IoT

CSS Icons

CSS Icons from various libraries can be effortlessly styled and customized with CSS, allowing for alterations in size, color, shadow, and more. These icons serve as intuitive graphical elements, enhancing navigation and conveying specific meanings.

There are 3 types of icon libraries available, namely

1. Font Awesome Icons
2. Google Icons
3. Bootstrap Icons

We will include the required CDN link from the available icon library, which will help us to use the pre-defined icon classes or we can customize it using the CSS.

Font Awesome Icons

To use the Font Awesome icons, go to fontawesome.com, sign in, and get a code to add in the <head> section of your HTML page:

```
<script src="https://kit.fontawesome.com/yourcode.js" crossorigin="anonymous"></script>
```

```

<!DOCTYPE html>
<html>
<head>
<title>Font Awesome Icons</title>
<meta name="viewport" content="width=device-width, initial-scale=1">
<script src="https://kit.fontawesome.com/a076d05399.js"
crossorigin="anonymous"></script>
<!--Get your own code at fontawesome.com-->
</head>
<body>
<h1>Font Awesome icon library</h1>

<p>Some Font Awesome icons:</p>
<i class="fas fa-cloud"></i>
<i class="fas fa-heart"></i>
<i class="fas fa-car"></i>
<i class="fas fa-file"></i>
<i class="fas fa-bars"></i>

<p>Styled Font Awesome icons (size and color):</p>
<i class="fas fa-cloud" style="font-size:24px;"></i>
<i class="fas fa-cloud" style="font-size:36px;"></i>
<i class="fas fa-cloud" style="font-size:48px;color:red;"></i>
<i class="fas fa-cloud" style="font-size:60px;color:lightblue;"></i>
</body>
</html>

```

Browser Output:

Font Awesome icon library

Some Font Awesome icons:



Styled Font Awesome icons (size and color):



Bootstrap Icons

To use the Bootstrap glyphicons, add the following line inside the <head> section of your HTML page:

```
<link rel="stylesheet"  
      href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/css/bootstrap.min.css">
```

Note: No downloading or installation is required!

```
<!DOCTYPE html>

<html>

<head>
<title>Bootstrap Icons</title>
<meta name="viewport" content="width=device-width, initial-scale=1">
<link rel="stylesheet"
href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/css/bootstrap.min.css">
</head>

<body class="container">
<h1>Bootstrap icon library</h1>
<p>Some Bootstrap icons:</p>
<i class="glyphicon glyphicon-cloud"></i>
<i class="glyphicon glyphicon-remove"></i>
<i class="glyphicon glyphicon-user"></i>
<i class="glyphicon glyphicon-envelope"></i>
<i class="glyphicon glyphicon-thumbs-up"></i>
<br><br>
<p>Styled Bootstrap icons (size and color):</p>
<i class="glyphicon glyphicon-cloud" style="font-size:24px;"></i>
<i class="glyphicon glyphicon-cloud" style="font-size:36px;"></i>
<i class="glyphicon glyphicon-cloud" style="font-size:48px;color:red;"></i>
<i class="glyphicon glyphicon-cloud" style="font-size:60px;color:lightblue;"></i>
</body>
</html>
```

Browser output:

Bootstrap icon library

Some Bootstrap icons:



Styled Bootstrap icons (size and color):



Google Icons

To use the Google icons, add the following line inside the <head> section of your HTML page:

```
<link rel="stylesheet" href="https://fonts.googleapis.com/icon?family=Material+Icons">
```

Note: No downloading or installation is required!

```
<!DOCTYPE html>

<html>
  <head>
    <title>Google Icons</title>
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <link rel="stylesheet"
      href="https://fonts.googleapis.com/icon?family=Material+Icons">
  </head>
  <body>
    <h1>Google icon library</h1>
    <p>Some Google icons:</p>
    <i class="material-icons">cloud</i>
    <i class="material-icons">favorite</i>
    <i class="material-icons">attachment</i>
    <i class="material-icons">computer</i>
    <i class="material-icons">traffic</i>
    <br><br>
    <p>Styled Google icons (size and color):</p>
    <i class="material-icons" style="font-size:24px;">cloud</i>
    <i class="material-icons" style="font-size:36px;">cloud</i>
    <i class="material-icons" style="font-size:48px;color:red;">cloud</i>
    <i class="material-icons" style="font-size:60px;color:lightblue;">cloud</i>
  </body>
</html>
```

Browser Output:

Google icon library

Some Google icons:



Styled Google icons (size and color):



CSS Colors

CSS Colors are an essential part of web design, providing the ability to bring your HTML elements to life. This feature allows developers to set the color of various HTML elements, including font color, background color, and more.

Color Format	Description
Color Name	<p>These are a set of predefined colors which are used by their names. For example: red, blue, green etc.</p> <p>Syntax: <code>h1 {color: color-name;}</code></p>
RGB Format	<p>The RGB (Red, Green, Blue) format is used to define the color of an HTML element by specifying the R, G, and B values range between 0 to 255.</p> <p>Syntax: <code>h1 {color: rgb(R, G, B);}</code></p>
RGBA Format	<p>The RGBA format is similar to the RGB, but it includes an Alpha component that specifies the transparency of elements.</p> <p>Syntax: <code>h1 { color: rgba(R, G, B, A); }</code></p>
Hexadecimal Notation	<p>The hexadecimal notation begins with a # symbol followed by 6 characters each ranging from 0 to F.</p> <p>Syntax: <code>h1 { color: #(0-F)(0-F)(0-F)(0-F)(0-F)(0-F); }</code></p>
HSL	<p>HSL stands for Hue, Saturation, and Lightness respectively. This format uses the cylindrical coordinate system.</p> <p>Syntax: <code>h1 { color: hsl(H, S, L); }</code></p>
HSLA	<p>The HSLA color property is similar to the HSL property, but it includes an Alpha component that specifies the transparency of elements.</p> <p>Syntax: <code>h1 { color: hsla(H, S, L, A); }</code></p>

CSS Opacity

The CSS opacity property is used to specify the transparency of an element. In simple word, you can say that it specifies the clarity of the image.

In technical terms, Opacity is defined as degree in which light is allowed to travel through an object.

Opacity setting is applied uniformly across the entire object and the opacity value is defined in term of digital value less than 1. The lesser opacity value displays the greater opacity. Opacity is not inherited.

Example for CSS Opacity

```
<!DOCTYPE html>
<html>
<head>
<style>
    .op1{
        opacity: 0.5;
    }
</style>
</head>
<body>
    <h1>Image Transparency</h1>
    <p>The opacity property specifies the transparency of an element. The lower the value, the more transparent:</p>
    <p>Image without opacity:</p>
    
    <p>Image with 50% opacity:</p>
    
</body>
</html>
```

Browser Output:

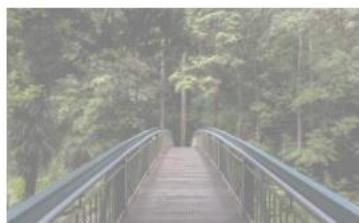
Image Transparency

The opacity property specifies the transparency of an element. The lower the value, the more transparent:

Image without opacity:



Image with 50% opacity:





Bootstrap-5 Framework

Introduction to Bootstrap

Bootstrap is a free and open-source collection of CSS and JavaScript/jQuery code used for creating dynamic website layouts and web applications. As one of the most popular front-end frameworks, Bootstrap offers a comprehensive set of predefined CSS classes and components that simplify the development of responsive, mobile-first websites.

Bootstrap's versatility makes it suitable for mobile-friendly web development. The framework ensures that web pages are adaptable to various screen sizes, providing a consistent experience across different devices. This responsiveness is achieved through a variety of classes designed to handle different layouts and elements dynamically.

The framework is available for free and can be integrated into projects in two main ways: by downloading the zip files and including Bootstrap's libraries/modules locally, or by directly linking to the online version through a CDN (Content Delivery Network).

Bootstrap 5, the latest stable version, was officially released on June 16, 2020, after several months of feature refinement. It continues the tradition of being a robust, mobile-first framework, compatible with the latest stable releases of all major browsers and operating systems. Bootstrap 5 emphasizes responsiveness by default, ensuring that websites look great on all devices, from small smartphones to large desktop monitors.

Bootstrap is an essential toolkit for modern web development, providing an array of CSS and JavaScript components that streamline the process of creating responsive, mobile-first websites and applications. Its widespread use and continuous updates make it a reliable choice for developers looking to build dynamic and accessible web projects.

Bootstrap Versions

Bootstrap 5 (released 2021) is the newest version of Bootstrap (released 2013); with new components, faster stylesheet and more responsiveness.

Bootstrap 5 supports the latest, stable releases of all major browsers and platforms. However, Internet Explorer 11 and down is not supported.

The main differences between Bootstrap 5 and Bootstrap 3 & 4, is that Bootstrap 5 has switched to vanilla JavaScript instead of jQuery.

Key Features of Bootstrap 5

- **Responsive Grid System:** Bootstrap 5 continues to use its powerful grid system based on flexbox. This system allows developers to create flexible and responsive layouts that automatically adapt to different screen sizes.
- **Predefined CSS Classes:** The framework offers a vast collection of predefined CSS classes that cover typography, forms, buttons, navigation, and other essential design elements. These classes facilitate rapid development and ensure consistent styling across different components.
- **JavaScript Components:** Bootstrap 5 includes a variety of JavaScript components such as modals, carousels, tooltips, and popovers. These components add interactivity to web pages, enhancing the user experience without requiring extensive JavaScript coding.
- **Customization and Theming:** Developers can easily customize Bootstrap 5 to match their project's branding by modifying Sass variables and using mixins. The theming system allows for the creation of custom themes, providing flexibility in design.
- **Mobile-First Approach:** Designed with a mobile-first philosophy, Bootstrap 5 ensures that web pages are optimized for mobile devices from the start. This approach helps create a seamless and user-friendly experience on smartphones, tablets, and desktops.
- **Improved Forms:** Bootstrap 5 introduces improved form controls, with enhanced styling and layout options. The new form elements are more customizable and accessible, making it easier to create complex forms.
- **Utility Classes:** A broad range of utility classes in Bootstrap 5 enables quick adjustments to layout, spacing, alignment, and other styling aspects without writing additional CSS. These utility classes simplify the process of fine-tuning designs.
- **No More jQuery Dependency:** One of the significant changes in Bootstrap 5 is the removal of jQuery as a dependency. This shift to vanilla JavaScript reduces the overall size of the framework and improves performance.
- **Icons and SVGs:** Bootstrap 5 supports a wide range of icons through its integration with Bootstrap Icons, a separate library of high-quality SVG icons that can be easily customized and used within projects.

- **Enhanced Documentation:** The documentation for Bootstrap 5 has been significantly improved, providing detailed examples, comprehensive explanations, and robust search functionality. This makes it easier for developers to learn and implement the framework's features.

Advantages of Using Bootstrap

- Ease of Use: Bootstrap 5 simplifies the web development process with its extensive set of components and utilities, making it accessible even for developers with limited design skills.
- Consistency: By using predefined classes and components, Bootstrap 5 ensures a consistent look and feel across the entire website.
- Community and Support: As an open-source project with a large and active community, Bootstrap 5 benefits from continuous improvements, community support, and a wealth of shared resources.
- Cross-Browser Compatibility: The framework is tested and compatible with the latest stable releases of all major browsers, ensuring reliable performance across different platforms.



Fig.:Advantages of Using Bootstrap

Bootstrap Containers

In Bootstrap, container is used to set the content's margins dealing with the responsive behaviors of your layout. It contains the row elements and the row elements are the container of columns (known as grid system).

Containers are the most basic layout element in Bootstrap and are required when using our default grid system. Containers are used to contain, pad, and (sometimes) center the content within them. While containers can be nested, most layouts do not require a nested container.

Bootstrap comes with three different containers:

- **.container**, which sets a max-width at each responsive breakpoint
- **.container-fluid**, which is width: 100% at all breakpoints
- **.container-{breakpoint}**, which is width: 100% until the specified breakpoint

The table below illustrates how each container's max-width compares to the original .container and .container-fluid across each breakpoint.

	Extra small <code><576px</code>	Small <code>≥576px</code>	Medium <code>≥768px</code>	Large <code>≥992px</code>	X-Large <code>≥1200px</code>	XX-Large <code>≥1400px</code>
.container	100%	540px	720px	960px	1140px	1320px
.container-sm	100%	540px	720px	960px	1140px	1320px
.container-md	100%	100%	720px	960px	1140px	1320px
.container-lg	100%	100%	100%	960px	1140px	1320px
.container-xl	100%	100%	100%	100%	1140px	1320px
.container-xxl	100%	100%	100%	100%	100%	1320px
.container-fluid	100%	100%	100%	100%	100%	100%

Default container/ Fixed Container

.container class is a responsive, fixed-width container, meaning its **max-width** changes at each breakpoint.

```
<!DOCTYPE html>
<html lang="en">
<head>
<title>Bootstrap Example</title>
<meta charset="utf-8">
<meta name="viewport" content="width=device-width, initial-scale=1">
<link href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.3/dist/css/bootstrap.min.css" rel="stylesheet">
<script src="https://cdn.jsdelivr.net/npm/bootstrap@5.3.3/dist/js/bootstrap.bundle.min.js"></script>
</head>
<body>
<div class="container">
<h1>My First Bootstrap Page</h1>
<p>This part is inside a .container class.</p>
<p>The .container class provides a responsive fixed width container.</p>
<p>Resize the browser window to see that the container width will change at different breakpoints.</p>
</div>
</body>
</html>
```

Browser Output:

My First Bootstrap Page

This part is inside a .container class.

The .container class provides a responsive fixed width container.

Resize the browser window to see that the container width will change at different breakpoints.

Responsive containers

Responsive containers allow you to specify a class that is 100% wide until the specified breakpoint is reached, after which we apply max-widths for each of the higher breakpoints. For example, `.container-sm` is 100% wide to start until the `sm` breakpoint is reached, where it will scale up with `md`, `lg`, `xl`, and `xxl`.

```
<!DOCTYPE html>
<html lang="en">
<head>
  <title>Bootstrap Example</title>
  <meta charset="utf-8">
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.3/dist/css/bootstrap.min.css" rel="stylesheet">
  <script src="https://cdn.jsdelivr.net/npm/bootstrap@5.3.3/dist/js/bootstrap.bundle.min.js"></script>
</head>
<body>
  <div class="container-sm bg-secondary">100% wide until small breakpoint</div>
  <div class="container-md bg-secondary">100% wide until medium breakpoint</div>
  <div class="container-lg bg-secondary">100% wide until large breakpoint</div>
  <div class="container-xl bg-secondary">100% wide until extra large breakpoint</div>
  <div class="container-xxl bg-secondary">100% wide until extra extra large breakpoint</div>
</body>
</html>
```

Browser output:



Fluid containers

Use **.container-fluid** for a full width container, spanning the entire width of the viewport.

```
<!DOCTYPE html>

<html lang="en">
  <head>
    <title>Bootstrap Example</title>
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.3/dist/css/bootstrap.min.css" rel="stylesheet">
    <script src="https://cdn.jsdelivr.net/npm/bootstrap@5.3.3/dist/js/bootstrap.bundle.min.js"></script>
  </head>
  <body>
    <div class="container-fluid">
      <h1>My First Bootstrap Page</h1>
      <p>This part is inside a .container-fluid class.</p>
      <p>The .container-fluid class provides a full width container, spanning the entire width of the viewport.</p>
    </div>
  </body>
</html>
```

Browser Output

My First Bootstrap Page

This part is inside a .container-fluid class.

The .container-fluid class provides a full width container, spanning the entire width of the viewport.

Bootstrap Grid System

Bootstrap Grid System allows up to 12 columns across the page. You can use each of them individually or merge them together for wider columns. You can use all combinations of values summing up to 12. You can use 12 columns each of width 1, or use 4 columns each of width 3 or any other combination.

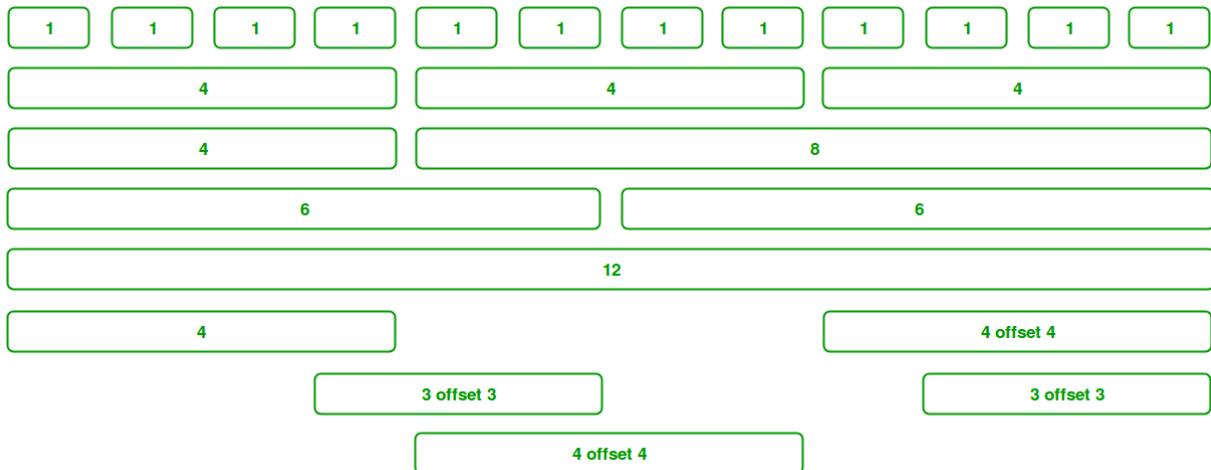


Fig.:Bootstrap Grid System

Bootstrap's grid system is responsive, and the columns will re-arrange depending on the screen size: On a big screen it might look better with the content organized in three columns, but on a small screen it would be better if the content items were stacked on top of each other.

Grid Classes

The Bootstrap grid system has four classes:

- xs (for phones - screens less than 768px wide)
- sm (for tablets - screens equal to or greater than 768px wide)
- md (for small laptops - screens equal to or greater than 992px wide)
- lg (for laptops and desktops - screens equal to or greater than 1200px wide)

The classes above can be combined to create more dynamic and flexible layouts.

Each class scales up, so if you wish to set the same widths for xs and sm, you only need to specify xs.

Grid System Rules

Some Bootstrap grid system rules:

- Rows must be placed within a `.container` (fixed-width) or `.container-fluid` (full-width) for proper alignment and padding
- Use rows to create horizontal groups of columns
- Content should be placed within columns, and only columns may be immediate children of rows
- Predefined classes like `.row` and `.col-sm-4` are available for quickly making grid layouts
- Columns create gutters (gaps between column content) via padding. That padding is offset in rows for the first and last column via negative margin on `.rows`
- Grid columns are created by specifying the number of 12 available columns you wish to span. For example, three equal columns would use three `.col-sm-4`
- Column widths are in percentage, so they are always fluid and sized relative to their parent element

Basic Structure of a Bootstrap Grid

The following is a basic structure of a Bootstrap grid:

```
<div class="container">
  <div class="row">
    <div class="col-*-*"></div>
    <div class="col-*-*"></div>
  </div>
  <div class="row">
    <div class="col-*-*"></div>
    <div class="col-*-*"></div>
    <div class="col-*-*"></div>
  </div>
  <div class="row">
    ...
  </div>
</div>
```

So, to create the layout you want, create a container (`<div class="container">`). Next, create a row (`<div class="row">`). Then, add the desired number of columns (tags with appropriate `.col-*-*` classes). Note that numbers in `.col-*-*` should always add up to 12 for each row.

Grid Options

While Bootstrap uses **ems** or **rems** for defining most sizes, **pxs** are used for grid breakpoints and container widths. This is because the viewport width is in pixels and does not change with the font size.

See how aspects of the Bootstrap grid system work across multiple devices with a handy table.

	Extra small ≤576px	Small ≥576px	Medium ≥768px	Large ≥992px	Extra large ≥1200px
Max container width	None (auto)	540px	720px	960px	1140px
Class prefix	.col-	.col-sm-	.col-md-	.col-lg-	.col-xl-
# of columns	12				
Gutter width	30px (15px on each side of a column)				
Nestable	Yes				
Column ordering	Yes				

Fig.:Working of Bootstrap grid system across multiple devices



Bootstrap Example for Three Equal Columns

```

<!DOCTYPE html>

<html lang="en">

<head>
    <title>Bootstrap Example</title>
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <link href="https://maxcdn.bootstrapcdn.com/bootstrap/3.4.1/css/bootstrap.min.css" rel="stylesheet">
    <script src="https://ajax.googleapis.com/ajax/libs/jquery/3.7.1/jquery.min.js"></script>
    <script src="https://maxcdn.bootstrapcdn.com/bootstrap/3.4.1/js/bootstrap.min.js"></script>
</head>

<body>
    <div class="container-fluid">
        <h1>Hello World!</h1>
        <p>Resize the browser window to see the effect.</p>
        <p>The columns will automatically stack on top of each other when the screen is less than 768px wide.</p>
        <div class="row">
            <div class="col-sm-4" style="background-color:lavender;">.col-sm-4</div>
            <div class="col-sm-4" style="background-color:lavenderblush;">.col-sm-4</div>
            <div class="col-sm-4" style="background-color:lavender;">.col-sm-4</div>
        </div>
    </div>
</body></html>

```

Browser Output:

Hello World!

Resize the browser window to see the effect.

The columns will automatically stack on top of each other when the screen is less than 768px wide.

.col-sm-4	.col-sm-4	.col-sm-4
-----------	-----------	-----------

Three Unequal Columns

Syntax:

```
<div class="container-fluid">
    <div class="row">
        <div class="col-sm-3">.col-sm-3</div>
        <div class="col-sm-6">.col-sm-6</div>
        <div class="col-sm-3">.col-sm-3</div>
    </div>
</div>
```

Browser Output:

.col-sm-3	.col-sm-6	.col-sm-3
-----------	-----------	-----------

Two Unequal Columns

Syntax:

```
<div class="container">
    <div class="row">
        <div class="col-sm-4">.col-sm-4</div>
        <div class="col-sm-8">.col-sm-8</div>
    </div>
</div>
```

Browser Output:

.col-sm-4	.col-sm-8
-----------	-----------

No gutters

Use the `.row-no-gutters` class to remove the gutters from a row and its columns:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <title>Bootstrap Example</title>
  <meta charset="utf-8">
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/3.4.1/css/bootstrap.min.css">
  <script src="https://ajax.googleapis.com/ajax/libs/jquery/3.7.1/jquery.min.js"></script>
  <script src="https://maxcdn.bootstrapcdn.com/bootstrap/3.4.1/js/bootstrap.min.js"></script>
</head>
<body>
<div class="container-fluid">
  <h1>No gutters</h1>
  <p>Use the .row-no-gutters class to remove the gutters from a row and its columns:</p>
  <div class="row row-no-gutters">
    <div class="col-sm-4" style="background-color:red;">.col-sm-4</div>
    <div class="col-sm-8" style="background-color:green;">.col-sm-8</div>
  </div>
  <br>
  <p>And here's a row with gutters to demonstrate the differences:</p>
  <div class="row">
    <div class="col-sm-4" style="background-color:red;">.col-sm-4</div>
    <div class="col-sm-8" style="background-color:green;">.col-sm-8</div>
  </div>
</div> </body> </html>
```

Browser Output:

No gutters

Use the `.row-no-gutters` class to remove the gutters from a row and its columns:

```
.col-sm-4  
.col-sm-8
```

And here's a row with gutters to demonstrate the differences:

```
.col-sm-4  
.col-sm-8
```

Typography

Bootstrap Text/Typography

Bootstrap's Default Settings

- Bootstrap's global default font-size is **14px**, with a line-height of **1.428**.
- This is applied to the `<body>` element and all paragraphs (`<p>`).
- In addition, all `<p>` elements have a bottom margin that equals half their computed line-height (10px by default).

Bootstrap Typography provides a standardized and flexible approach to text styling, offering various classes for headings, paragraphs, and inline text elements. It ensures consistent typography across different devices and screen sizes, enhancing readability and aesthetics.

Typography can be used to create:

- Headings
- Subheadings
- Text and Paragraph font color, font type, and alignment
- Lists
- Other inline elements

Headings

All HTML headings, `<h1>` through `<h6>`, are available. `.h1` through `.h6` classes are also available, for when you want to match the font styling of a heading but cannot use the associated HTML element.

Display headings

Traditional heading elements are designed to work best in the meat of your page content. When you need a heading to stand out, consider using a **display heading**—a larger, slightly more opinionated heading style. `.display-1` through `.display-6` classes are available for display headings

Abbreviations

Stylized implementation of HTML's `<abbr>` element for abbreviations and acronyms to show the expanded version on hover. Abbreviations have a default underline and gain a help cursor to provide additional context on hover and to users of assistive technologies.

Lead

Make a paragraph stand out by adding `.lead`.

Some classes and Tags to implement the typography feature of bootstrap:

Inline Element	Description
<code>h1 – h6</code>	Font styling resembling a heading, without using the associated HTML element
<code>text-muted</code>	Fades the text, making it appear greyed out
<code>display</code>	Enhances headings for better visual presentation
<code>lead</code>	Makes a paragraph stand out visually
<code>mark</code>	Highlights the text
<code>small</code>	Creates secondary subheadings
<code>initialism</code>	Renders abbreviations in slightly smaller text size
<code>blockquote</code>	Indicates quoted content
<code>blockquote-footer</code>	Provides footer details for identifying the source of the quote
<code>text-center</code>	Aligns text to the center

Inline Element	Description
list-inline	Makes list items appear inline
text-truncate	Shortens longer text by truncating with an ellipsis
text-uppercase	Transforms text to uppercase
text-lowercase	Transforms text to lowercase
text-capitalize	Transforms text to capitalize the first letter of each word, leaving other letters in lowercase
pre-scrollable	Makes a preformatted element scrollable
dl-horizontal	Aligns term and description elements side-by-side
list-unstyled	Removes default list-style and left margin from list items
text-right	Aligns text to the right
text-left	Aligns text to the left

Example 1:

In this example it includes headings from h1 to h3 with corresponding classes, along with display classes (display-3, display-4, and display-5) for different styles and the <small> element is used for secondary text.

```
<!DOCTYPE html>
<html>
  <head>
    <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.0.2/dist/css/bootstrap.min.css" rel="stylesheet" integrity="sha384-EVSTQN3/azprG1Anm3QDgpJLIm9Nao0Yz1ztcQTWfspd3yD65VohhpuuC0mLASjC" crossorigin="anonymous">
    <title>Text-muted</title>
  </head>
  <body>
    <p class="h1">h1. Bootstrap heading</p>
    <p class="h2">h2. Bootstrap heading</p>
    <p class="h3">h3. Bootstrap heading</p>
    <h3 class="display-3">Display-3 property</h3>
    <h3 class="display-4"> Display-4 property</h3>
    <h3 class="display-5"> Display-5 property </h3>
    <h1>Lighter, Secondary Text</h1>
    <p>The small element is used to create a lighter, secondary text in any heading:</p>
    <h1>h1 heading <small> secondary text </small> </h1>
    <h2>h2 heading <small> secondary text </small> </h2>
    <h3>h3 heading <small> secondary text </small> </h3>
  </body>
</html>
```

Browser Output

h1. Bootstrap heading

h2. Bootstrap heading

h3. Bootstrap heading

Display-3 property

Display-4 property

Display-5 property

Lighter, Secondary Text

The small element is used to create a lighter, secondary text in any heading:

h1 heading secondary text

h2 heading secondary text

h3 heading secondary text

Example 2:

In this example includes muted text, lead paragraph, highlighted text using the mark element, blockquote with footer, and a n abbreviation with initialism class.

```
<!DOCTYPE html>
<html>
  <head>
    <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.0.2/dist/css/bootstrap.min.css" rel="stylesheet" integrity="sha384-EVSTQN3/azprG1Anm3QDgpJLIm9Nao0Yz1ztcQTwFspd3yD65VohhpuuCOMLASjC" crossorigin="anonymous">
    <title>Bootstrap Typography</title>
  </head>
  <body>
    <!-- Muted text-->
    <p class="text-muted">Internet of Things //A Muted Text.</p>
    <p>Internet of Things - Normal Paragraph</p>
    <!-- using lead class-->
    <p class="lead">Internet of Things - Lead Paragraph </p>
    <!-- using mark-->
    <mark>Internet of Things - Highlighted</mark> <br><br>
    <!-- using blockquote tag and blockquote-footer class.-->
    <blockquote>Internet of Things</blockquote>
      <blockquote class="blockquote-footer">is one of the technologies that are driving digital transformation. </blockquote>
    <!-- using initialism class-->
    <abbr title="Internet of Things" class="text-success initialism" >IoT</abbr> is a network of interrelated devices that connect and exchange data with other IoT devices and the cloud.
  </body>
</html>
```

Browser Output

Internet of Things //A Muted Text.

Internet of Things - Normal Paragraph

Internet of Things - Lead Paragraph

Internet of Things - Highlighted

Internet of Things

— is one of the technologies that are driving digital transformation.

IOT is a network of interrelated devices that connect and exchange data with other IoT devices and the cloud.

Bootstrap Jumbotron

A jumbotron was introduced in Bootstrap 3 as a big padded box for calling extra attention to some special content or information.

Bootstrap 4 Jumbotron

A jumbotron indicates a big grey box for calling extra attention to some special content or information. Inside a jumbotron you can put nearly any valid HTML, including other Bootstrap elements/classes.

Example for Bootstrap 4

```
<!DOCTYPE html>

<html lang="en">

<head>

<title>Bootstrap Jumbotron Example</title>

<meta charset="utf-8">

<meta name="viewport" content="width=device-width, initial-scale=1">

<link rel="stylesheet" href="https://cdn.jsdelivr.net/npm/bootstrap@4.6.2/dist/css/bootstrap.min.css">

</head>

<body>

<div class="container">

<div class="jumbotron">

<h1>Internet of Things</h1>

<p> Internet of Things is one of the technologies that are driving digital transformation.</p>

</div>

</div>

</body>

</html>
```

Browser Output:

Internet of Things

Internet of Things is one of the technologies that are driving digital transformation.

Full-width Jumbotron

If you want a full-width jumbotron without rounded borders, add the `.jumbotron-fluid` class and a `.container` or `.container-fluid` inside of it:

Example for Full width Jumbotron

```
<!DOCTYPE html>
<html lang="en">
<head>
  <title>Bootstrap Example</title>
  <meta charset="utf-8">
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <link rel="stylesheet" href="https://cdn.jsdelivr.net/npm/bootstrap@4.6.2/dist/css/bootstrap.min.css">
</head>
<body>
  <div class="jumbotron jumbotron-fluid">
    <div class="container">
      <h1>Internet of Things</h1>
      <p> Internet of Things is one of the technologies that are driving digital transformation.</p>
    </div>
  </div>
</body> </html>
```

Browser Output

Internet of Things

Internet of Things is one of the technologies that are driving digital transformation.

Bootstrap 5 Jumbotron

Jumbotrons are no longer supported in Bootstrap 5. However, you can use a `<div>` element and add special helper classes together with a color class to achieve the same effect:

Example for Bootstrap 5

```
<!DOCTYPE html>
<html>
<head>
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.3/dist/css/bootstrap.min.css" rel="stylesheet">
</head>
<body>
<div class="container mt-3">
  <h2>Example of Jumbotron</h2>
  <div class="mt-4 p-5 bg-primary text-white rounded">
    <h1>Internet of Things</h1>
    <p> Internet of Things is one of the technologies that are driving digital transformation. </p>
  </div>
</div>
</body>
</html>
```

Browser Output:

Example of Bootstrap Jumbotron 5

Internet of Things

Internet of Things is one of the technologies that are driving digital transformation.

Bootstrap Colors

Text Colors

Bootstrap 5 has some contextual classes that can be used to provide "meaning through colors".

The classes for text colors are: `.text-muted`, `.text-primary`, `.text-success`, `.text-info`, `.text-warning`, `.text-danger`, `.text-secondary`, `.text-white`, `.text-dark`, `.text-body` (default body color/often black) and `.text-light`. You can also add 50% opacity for black or white text with the `.text-black-50` or `.text-white-50` classes:

Background Colors

The classes for background colors are: `.bg-primary`, `.bg-success`, `.bg-info`, `.bg-warning`, `.bg-danger`, `.bg-secondary`, `.bg-dark` and `.bg-light`.

Table Colors

The classes for table colors are:

Class	Description
<code>.table-primary</code>	Blue: Indicates an important action
<code>.table-success</code>	Green: Indicates a successful or positive action
<code>.table-danger</code>	Red: Indicates a dangerous or potentially negative action
<code>.table-info</code>	Light blue: Indicates a neutral informative change or action
<code>.table-warning</code>	Orange: Indicates a warning that might need attention
<code>.table-active</code>	Grey: Applies the hover color to the table row or table cell
<code>.table-secondary</code>	Grey: Indicates a slightly less important action
<code>.table-light</code>	Light grey table or table row background
<code>.table-dark</code>	Dark grey table or table row background

Example for Bootstrap text colors

```
<!DOCTYPE html>

<html lang="en">

<head>
  <title>Bootstrap text colors</title>
  <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.3/dist/css/bootstrap.min.css" rel="stylesheet">
</head>

<body>
  <div class="container mt-3">
    <h2>Contextual Colors</h2>
    <p>Use the contextual classes to provide "meaning through colors":</p>
    <p class="text-muted">This text is muted.</p>
    <p class="text-primary">This text is important.</p>
    <p class="text-success">This text indicates success.</p>
    <p class="text-info">This text represents some information.</p>
    <p class="text-warning">This text represents a warning.</p>
    <p class="text-danger">This text represents danger.</p>
    <p class="text-secondary">Secondary text.</p>
    <p class="text-dark">This text is dark grey.</p>
    <p class="text-body">Default body color (often black).</p>
    <p class="text-light">This text is light grey (on white background).</p>
    <p class="text-white">This text is white (on white background).</p>
  </div>
</body> </html>
```

Browser Output:

Contextual Colors

Use the contextual classes to provide "meaning through colors":

This text is muted.

This text is important.

This text indicates success.

This text represents some information.

This text represents a warning.

This text represents danger.

Secondary text.

This text is dark grey.

Default body color (often black).

This text is light grey (on white background)

Example for Bootstrap Opacity Text Colors

```
<!DOCTYPE html>
<html lang="en">
<head>
<title>Example For Opacity Text Colors </title>
<link href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.3/dist/css/bootstrap.min.css" rel="stylesheet">
</head>
<body>
<div class="container mt-3">
<h2>Opacity Text Colors</h2>
<p>Add 50% opacity for black or white text with the .text-black-50 or .text-white-50 classes:</p>
<p class="text-black-50">Black text with 50% opacity on white background</p>
<p class="text-white-50 bg-dark">White text with 50% opacity on black background</p>
</div>
</body> </html>
```

Browser Output:

Opacity Text Colors

Add 50% opacity for black or white text with the .text-black-50 or .text-white-50 classes:

Black text with 50% opacity on white background

White text with 50% opacity on black background

Example for Bootstrap background colors

```
<!DOCTYPE html>

<html lang="en">

<head>
    <title>Bootstrap Example for Background Colors</title>
    <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.3/dist/css/bootstrap.min.css" rel="stylesheet">
</head>

<body>
    <div class="container mt-3">
        <h2>Contextual Backgrounds</h2>
        <p>Use the contextual background classes to provide "meaning through colors".</p>
        <div class="bg-primary p-3">Embedded Full Stack IoT </div>
        <div class="bg-success p-3">Embedded Full Stack IoT</div>
        <div class="bg-info p-3">Embedded Full Stack IoT</div>
        <div class="bg-warning p-3">Embedded Full Stack IoT</div>
        <div class="bg-danger p-3">Embedded Full Stack IoT</div>
        <div class="bg-secondary p-3">Embedded Full Stack IoT</div>
        <div class="bg-dark p-3 text-white">Embedded Full Stack IoT</div>
        <div class="bg-light p-3">Embedded Full Stack IoT</div>
    </div>
</body>
</html>
```

Browser Output:

Contextual Backgrounds

Use the contextual background classes to provide "meaning through colors".

Embedded Full Stack IoT



Bootstrap 5 Images

Bootstrap 5 Responsive images are used to resize the images according to their parent element and screen sizes. It means, the size of the image should not overflow its parent element and will grow and shrink according to the change in the size of its parent without losing its aspect ratio.

Image Shapes

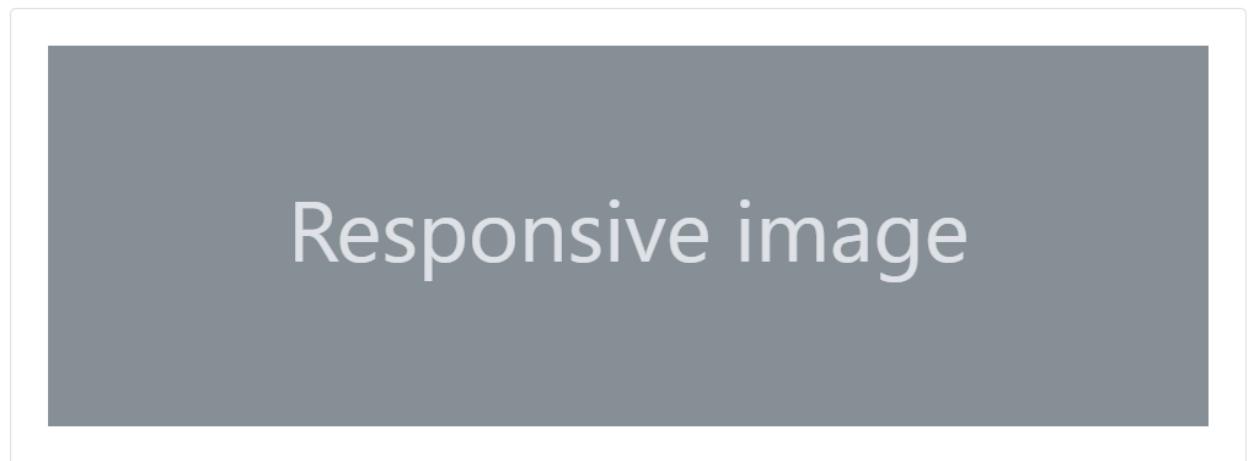
- **Rounded Corners:** The .rounded class adds rounded corners to an image:
``
- **Circle:** The .rounded-circle class shapes the image to a circle:
``
- **Thumbnail:** The .img-thumbnail class shapes the image to a thumbnail (bordered):
``



Fig.:Types of image shapes

Responsive images

Images in Bootstrap are made responsive with `.img-fluid`. This applies `max-width:100%;` and `height: auto;` to the image so that it scales with the parent element.

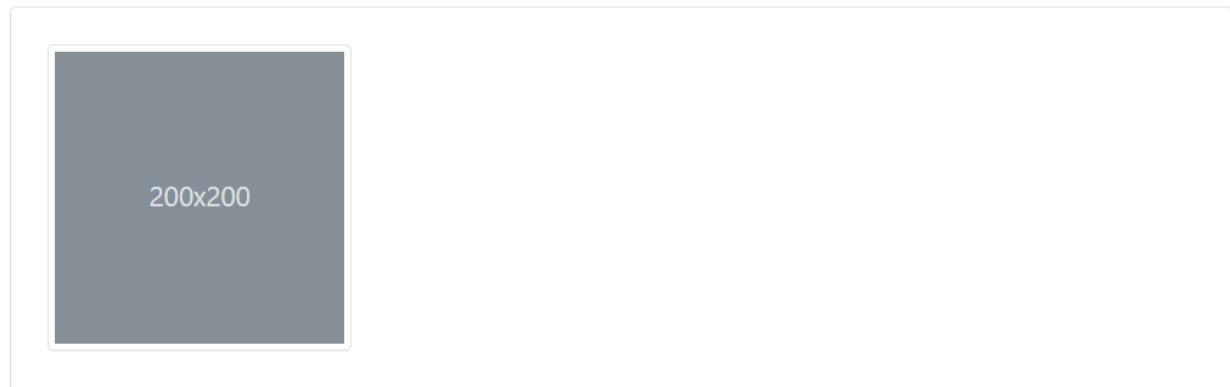


```
>
```

Image thumbnails

In addition to our border-radius utilities, you can use `.img-thumbnail` to give an image a rounded 1px border appearance.

```
>
```

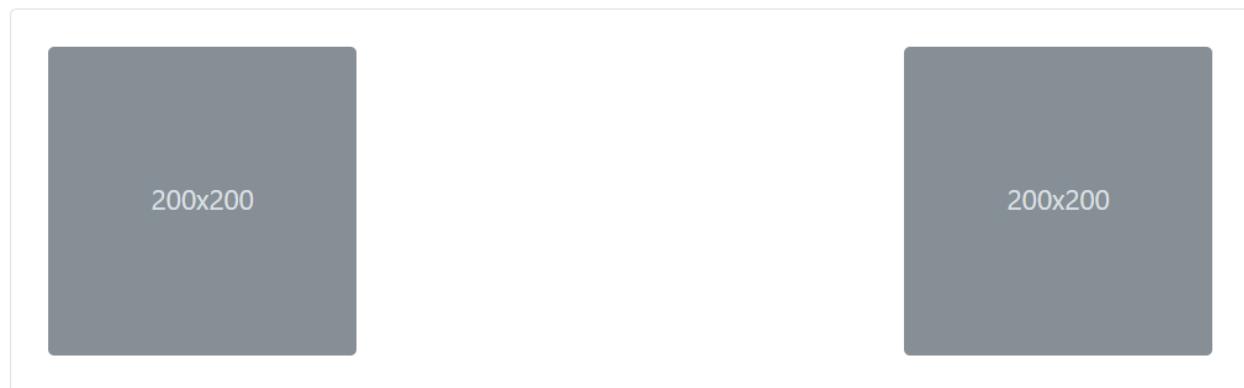


Aligning images

Align images with the helper float classes or text alignment classes. block-level images can be centered using the `.mx-auto` margin utility class.

```
>
```

```
>
```



```
>
```



```
<div class="text-center">  
  >  
</div>
```

Bootstrap Cards

A card is a flexible and extensible content container. It includes options for headers and footers, a wide variety of content, contextual background colors, and powerful display options.

It replaces the use of panels, wells and thumbnails. All of it can be used in a single container called card.

Example for bootstrap card

```
<!DOCTYPE html>

<html lang="en">
  <head>
    <title>Bootstrap Card Example</title>
    <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.3/dist/css/bootstrap.min.css" rel="stylesheet">
  </head>
  <body>
    <div class="container mt-3">
      <h2> Bootstrap Card With Header and Footer</h2>
      <div class="card">
        <div class="card-header">Header</div>
        <div class="card-body">Content</div>
        <div class="card-footer">Footer</div>
      </div>
    </div>
  </body>
</html>
```

Browser Output:

Bootstrap Card With Header and Footer

Header

Content

Footer

Bootstrap Buttons

Bootstrap Buttons are pre-styled components in the Bootstrap framework, offering consistent design and functionality. They streamline development by providing ready-to-use button styles, sizes, and behaviors, ensuring a cohesive and responsive user interface across web applications with minimal CSS customization. The `.btn` classes are designed to be used with the `<button>` element.

Types: Following are the nine types of buttons available in Bootstrap 5:

- btn-primary
- btn-secondary
- btn-success
- btn-danger
- btn-warning
- btn-info
- btn-light
- btn-dark
- btn-link

The button classes can be used on `<a>`, `<button>`, or `<input>` elements:

```

<!DOCTYPE html>

<html lang="en">

<head>

<title>Example For Bootstrap Buttons </title>

<meta name="viewport" content="width=device-width, initial-scale=1">

<link href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.3/dist/css/bootstrap.min.css" rel="stylesheet">

<script src="https://cdn.jsdelivr.net/npm/bootstrap@5.3.3/dist/js/bootstrap.bundle.min.js"></script>

</head>

<body>

<div class="container mt-3">

<h2>Button Styles</h2>

<button type="button" class="btn">Basic</button>

<button type="button" class="btn btn-primary">Primary</button>

<button type="button" class="btn btn-secondary">Secondary</button>

<button type="button" class="btn btn-success">Success</button>

<button type="button" class="btn btn-info">Info</button>

<button type="button" class="btn btn-warning">Warning</button>

<button type="button" class="btn btn-danger">Danger</button>

<button type="button" class="btn btn-dark">Dark</button>

<button type="button" class="btn btn-light">Light</button>

<button type="button" class="btn btn-link">Link</button>

</div>

</body> </html>

```

Browser Output:

Button Styles

Basic **Primary** Secondary Success Info Warning Danger Dark Light [Link](#)

Outline buttons

In need of a button, but not the hefty background colors they bring? Replace the default modifier classes with the `.btn-outline-*` ones to remove all background images and colors on any button.

Example:

```
<button type="button" class="btn btn-outline-primary">Primary</button>  
  
<button type="button" class="btn btn-outline-secondary">Secondary</button>  
  
<button type="button" class="btn btn-outline-success">Success</button>  
  
<button type="button" class="btn btn-outline-danger">Danger</button>  
  
<button type="button" class="btn btn-outline-warning">Warning</button>  
  
<button type="button" class="btn btn-outline-info">Info</button>  
  
<button type="button" class="btn btn-outline-light">Light</button>  
  
<button type="button" class="btn btn-outline-dark">Dark</button>
```

Browser Output:

Primary Secondary Success Danger Warning Info Light Dark

Button Sizes

Bootstrap provides four button sizes:

The classes that define the different sizes are:

- `.btn-lg`
- `.btn-sm`
- `.btn-xs`

Example:

```
<button type="button" class="btn          btn-primary          btn-lg">Large</button>
<button type="button" class="btn          btn-primary          btn-primary">Normal</button>
<button type="button" class="btn          btn-primary          btn-sm">Small</button>
<button type="button" class="btn btn-primary btn-xs">XSmall</button>
```

Browser Output:



Large Normal Small XSmall

Block Level Buttons

A block level button spans the entire width of the parent element. Add class `.btn-block` to create a block level button:

Example:

```
<button type="button" class="btn btn-primary btn-block">Button 1</button>
<button type="button" class="btn btn-default btn-block">Button 2</button>
```

Browser Output:



Button 1

Button 2

Active/Disabled Buttons

A button can be set to an active (appear pressed) or a disabled (unclickable) state:

The class `.active` makes a button appear pressed, and the class `.disabled` makes a button unclickable:

Example:

```
<button type="button" class="btn btn-primary active">Active Primary</button>
<button type="button" class="btn btn-primary disabled">Disabled Primary</button>
```

Browser Output:



Active Primary Disabled Primary

Bootstrap Framework Inputs

Basic Form Inputs

1. Text Inputs:

- **Basic Text Field:** <input type="text" class="form-control" placeholder="Enter text">
 - A standard single-line text input field.
- **Password Field:** <input type="password" class="form-control" placeholder="Enter password">
 - Used for entering passwords, with masked input characters.

2. Email and URL Inputs:

- **Email Input:** <input type="email" class="form-control" placeholder="Enter email">
 - Validates the input to ensure it follows the format of an email address.
- **URL Input:** <input type="url" class="form-control" placeholder="Enter website URL">
 - Ensures the input is a valid URL.

3. Number and Range Inputs:

- **Number Input:** <input type="number" class="form-control" placeholder="Enter a number">
 - Allows users to enter numeric values with optional constraints (min, max, step).
- **Range Input:** <input type="range" class="form-range">
 - Provides a slider control for selecting a value within a specified range.

4. Date and Time Inputs:

- **Date Picker:** <input type="date" class="form-control">
 - Allows users to select a date from a calendar interface.
- **Time Picker:** <input type="time" class="form-control">
 - Enables users to select a time.

Advanced Form Controls

1. Select Boxes:

- **Basic Select:** <select class="form-select"> <option>Option 1</option> <option>Option 2</option> </select>
 - A dropdown list for selecting one of several options.
- **Multiple Select:** <select multiple class="form-select"> <option>Option 1</option> <option>Option 2</option> </select>
 - Allows users to select multiple options.

2. Checkboxes and Radio Buttons:

- **Checkboxes:** <input type="checkbox" id="exampleCheck1" class="form-check-input" for="exampleCheck1"> <label class="form-check-label" for="exampleCheck1">Check me out</label>
 - Used for multiple selections.
- **Radio Buttons:** <input type="radio" class="form-check-input" name="options" id="option1" checked="" for="option1"> <label class="form-check-label" for="option1">Option 1</label>
 - Allows users to select one option from a group.

3. File Inputs:

- **File Upload:** <input type="file" class="form-control">
 - Enables users to upload files from their device.

Bootstrap Forms

Bootstrap Forms make creating user input areas easy with pre-styled components. They simplify form design from basic text fields to complex layouts, ensuring consistency and mobile-friendly responsiveness for capturing user data effectively.

Bootstrap's Default Settings

Form controls automatically receive some global styling with Bootstrap:

All textual <input>, <textarea>, and <select> elements with class **.form-control** have a width of 100%.

Bootstrap Form Layouts

Bootstrap provides three types of form layouts:

- Vertical form (this is default)
- Horizontal form
- Inline form

Standard rules for all three form layouts:

- Wrap labels and form controls in <div class="form-group"> (needed for optimum spacing)
- Add class .form-control to all textual <input>, <textarea>, and <select> elements

Example for Bootstrap Vertical Form

```

<!DOCTYPE html>
<html lang="en">
<head>
<title>Bootstrap Example</title>
<meta charset="utf-8">
<meta name="viewport" content="width=device-width, initial-scale=1">
<link rel="stylesheet"
      href="https://maxcdn.bootstrapcdn.com/bootstrap/3.4.1/css/bootstrap.min.css">
<script src="https://ajax.googleapis.com/ajax/libs/jquery/3.7.1/jquery.min.js"></script>
<script src="https://maxcdn.bootstrapcdn.com/bootstrap/3.4.1/js/bootstrap.min.js"></script>
</head>
<body>
<div class="container">
<h2>Vertical (basic) form</h2>
<form action="/action_page.php">
<div class="form-group">
<label for="email">Email:</label>
<input type="email" class="form-control" id="email" placeholder="Enter email"
       name="email">
</div>
<div class="form-group">
<label for="pwd">Password:</label>
<input type="password" class="form-control" id="pwd" placeholder="Enter password"
       name="pwd">
</div>
<div class="checkbox">
<label><input type="checkbox" name="remember"> Remember me</label>
</div>
<button type="submit" class="btn btn-default">Submit</button>
</form>
</div>
</body>
</html>

```

Browser Output:

Vertical (basic) form

Email:

Password:

Remember me

Bootstrap Inline Form

In an inline form, all of the elements are inline, left-aligned, and the labels are alongside.

Note: This only applies to forms within viewports that are at least 768px wide!

Additional rule for an inline form:

Add class `.form-inline` to the `<form>` element

Example for Bootstrap Inline Form

```

<!DOCTYPE html>
<html lang="en">
<head>
<title>Bootstrap Example</title>
<link rel="stylesheet"
      href="https://maxcdn.bootstrapcdn.com/bootstrap/3.4.1/css/bootstrap.min.css">
<script src="https://ajax.googleapis.com/ajax/libs/jquery/3.7.1/jquery.min.js"></script>
<script src="https://maxcdn.bootstrapcdn.com/bootstrap/3.4.1/js/bootstrap.min.js"></script>
</head>
<body>
<div class="container">
<h2>Inline form</h2>
<p>Make the viewport larger than 768px wide to see that all of the form elements are inline, left aligned, and the labels are alongside.</p>
<form class="form-inline" action="/action_page.php">
<div class="form-group">
<label for="email">Email:</label>
<input type="email" class="form-control" id="email" placeholder="Enter email" name="email">
</div>
<div class="form-group">
<label for="pwd">Password:</label>
<input type="password" class="form-control" id="pwd" placeholder="Enter password" name="pwd">
</div>
<div class="checkbox">
<label><input type="checkbox" name="remember"> Remember me</label>
</div>
<button type="submit" class="btn btn-default">Submit</button>
</form>
</div> </body> </html>

```

Browser Output:

Inline form

Make the viewport larger than 768px wide to see that all of the form elements are inline, left aligned, and the labels are alongside.

Email: Password: Remember me

Bootstrap Horizontal Form

A horizontal form means that the labels are aligned next to the input field (horizontal) on large and medium screens. On small screens (767px and below), it will transform to a vertical form (labels are placed on top of each input).

Additional rules for a horizontal form:

- Add class `.form-horizontal` to the `<form>` element
- Add class `.control-label` to all `<label>` elements

Use Bootstrap's predefined grid classes to align labels and groups of form controls in a horizontal layout.

Example for bootstrap horizontal form

```

<!DOCTYPE html>
<html lang="en">
<head>
    <title> Example for bootstrap horizontal form </title>
    <link rel="stylesheet"
        href="https://maxcdn.bootstrapcdn.com/bootstrap/3.4.1/css/bootstrap.min.css">
    <script src="https://ajax.googleapis.com/ajax/libs/jquery/3.7.1/jquery.min.js"></script>
    <script src="https://maxcdn.bootstrapcdn.com/bootstrap/3.4.1/js/bootstrap.min.js"></script>
</head>
<body>
<div class="container">
    <h2>Horizontal form</h2>
    <form class="form-horizontal" action="/action_page.php">
        <div class="form-group">
            <label class="control-label col-sm-2" for="email">Email:</label>
            <div class="col-sm-10">
                <input type="email" class="form-control" id="email" placeholder="Enter email"
                    name="email">
            </div>
        </div>
        <div class="form-group">
            <label class="control-label col-sm-2" for="pwd">Password:</label>
            <div class="col-sm-10">
                <input type="password" class="form-control" id="pwd" placeholder="Enter password"
                    name="pwd">
            </div>
        </div>
        <div class="form-group">
            <div class="col-sm-offset-2 col-sm-10">
                <div class="checkbox">
                    <label><input type="checkbox" name="remember"> Remember me</label>
                </div>
            </div>
        </div>
    </form>
</div>

```

```
</div>

<div class="form-group">
  <div class="col-sm-offset-2 col-sm-10">
    <button type="submit" class="btn btn-default">Submit</button>
  </div>
</div>
</form>
</div>
</body>
</html>
```

Browser Output:

Horizontal form

Email:

Password:

Remember me

Bootstrap 5 Forms

Stacked Form

All textual `<input>` and `<textarea>` elements with class `.form-control` get proper form styling:

```

<!DOCTYPE html>
<html lang="en">
<head>
<title>Bootstrap Example</title>
<meta charset="utf-8">
<meta name="viewport" content="width=device-width, initial-scale=1">
<link href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.3/dist/css/bootstrap.min.css" rel="stylesheet">
<script
src="https://cdn.jsdelivr.net/npm/bootstrap@5.3.3/dist/js/bootstrap.bundle.min.js"></script>
</head>
<body>
<div class="container mt-3">
<h2>Stacked form</h2>
<form action="/action_page.php">
<div class="mb-3 mt-3">
<label for="email">Email:</label>
<input type="email" class="form-control" id="email" placeholder="Enter email" name="email">
</div>
<div class="mb-3">
<label for="pwd">Password:</label>
<input type="password" class="form-control" id="pwd" placeholder="Enter password" name="pswd">
</div>
<div class="form-check mb-3">
<label class="form-check-label">
<input class="form-check-input" type="checkbox" name="remember"> Remember me
</label>
</div>
<button type="submit" class="btn btn-primary">Submit</button>
</form>
</div>

```

```
</body>
```

```
</html>
```

Browser Output:

Stacked form

Email:

Password:

Remember me

Submit

Bootstrap Tables

Example for Bootstrap Table

```
<!DOCTYPE html>

<html lang="en">

<head>

<title>Example For Bootstrap Table</title>

<link href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.3/dist/css/bootstrap.min.css" rel="stylesheet">

</head>

<body>

<div class="container mt-3">

<h2> Bootstrap table for dark background with border </h2>

<table class="table table-bordered table-dark">

<thead>

<tr>
    <th>Firstname</th>
    <th>Lastname</th>
    <th>Email</th>
</tr>

</thead>

<tbody>

<tr>
    <td>John</td>
    <td>Doe</td>
    <td>john@example.com</td>
</tr>

<tr>
    <td>Mary</td>
</tr>

</tbody>
</table>
</div>
</body>

```

```

<td>Moe</td>
<td>mary@example.com</td>
</tr>
<tr>
<td>July</td>
<td>Dooley</td>
<td>july@example.com</td>
</tr>
</tbody>
</table>
</div>
</body>
</html>

```

Browser Output:

Bootstrap table for dark background with border

Firstname	Lastname	Email
John	Doe	john@example.com
Mary	Moe	mary@example.com
July	Dooley	july@example.com

Bootstrap Navigation Bar

Bootstrap Navigation Bar provides a responsive, customizable, and pre-styled navigation component for web applications. It incorporates features like branding, navigation links, dropdowns, and responsiveness, enabling developers to create effective and visually appealing navigation menus effortlessly.

Navbar: Bootstrap provides various types of navigation bars:

Navbar Type	Description
Basic Navbar	Standard navigation bar layout.
Inverted Navbar	Navbar with inverted colors for a different look.
Coloured Navigation Bar	Navbar with customized colors for visual appeal.
Right-Aligned Navbar	The navbar is aligned to the right side of the page.
Fixed Navigation Bar	Navbar that remains fixed at the top of the page.
Drop-Down menu Navbar	Navbar with drop-down menu options for navigation.
Collapsible Navbar	Navbar with collapsible menu for space efficiency.

A navigation bar is a navigation header that is placed at the top of the page:

Basic Navbar

With Bootstrap, a navigation bar can extend or collapse, depending on the screen size.

A standard navigation bar is created with the `.navbar` class, followed by a responsive collapsing class: `.navbar-expand-xxl|xl|lg|md|sm` (stacks the navbar vertically on xxlarge, extra large, large, medium or small screens).

To add links inside the navbar, use either an `` element (or a `<div>`) with `class="navbar-nav"`. Then add `` elements with a `.nav-item` class followed by an `<a>` element with a `.nav-link` class:

```
<nav class="navbar navbar-expand-sm bg-light">
<div class="container-fluid">
<ul class="navbar-nav">
<li class="nav-item">
<a class="nav-link" href="#">Link 1</a>
</li>
<li class="nav-item">
<a class="nav-link" href="#">Link 2</a>
</li>
<li class="nav-item">
<a class="nav-link" href="#">Link 3</a>
</li>
</ul>
</div>
</nav>
```

Browser Output:



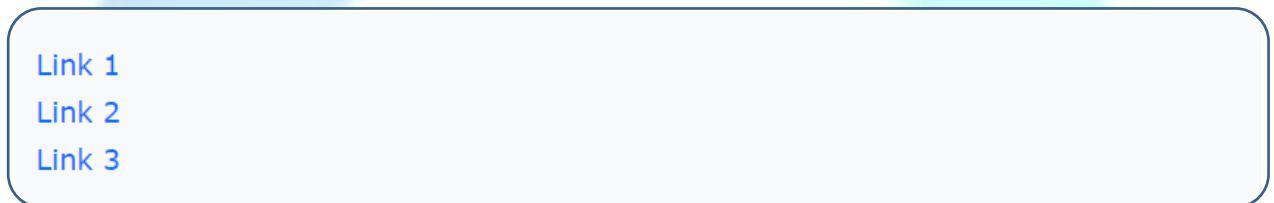
Link 1 Link 2 Link 3

Vertical Navbar

Remove the `.navbar-expand-*` class to create a navigation bar that will always be vertical:

```
<nav class="navbar bg-light">  
  <div class="container-fluid">  
    <ul class="navbar-nav">  
      <li class="nav-item">  
        <a class="nav-link" href="#">Link 1</a>  
      </li>  
      <li class="nav-item">  
        <a class="nav-link" href="#">Link 2</a>  
      </li>  
      <li class="nav-item">  
        <a class="nav-link" href="#">Link 3</a>  
      </li>  
    </ul>  
  </div>  
</nav>
```

Browser Output:



Link 1
Link 2
Link 3

Centered Navbar

Add the `.justify-content-center` class to center the navigation bar:

```
<nav class="navbar navbar-expand-sm bg-light justify-content-center">  
  <ul class="navbar-nav">  
    <li class="nav-item">  
      <a class="nav-link" href="#">Link 1</a>  
    </li>  
    <li class="nav-item">  
      <a class="nav-link" href="#">Link 2</a>  
    </li>  
    <li class="nav-item">  
      <a class="nav-link" href="#">Link 3</a>  
    </li>  
  </ul>  
</nav>
```

Browser Output:



Link 1 Link 2 Link 3

Colored Navbar

Use any of the `.bg-color` classes to change the background color of the navbar (`.bg-primary`, `.bg-success`, `.bg-info`, `.bg-warning`, `.bg-danger`, `.bg-secondary`, `.bg-dark` and `.bg-light`)

Add a white text color to all links in the navbar with the `.navbar-dark` class, or use the `.navbar-light` class to add a black text color.

```

<nav class="navbar navbar-expand-sm bg-dark navbar-dark">

<div class="container-fluid">

<ul class="navbar-nav">

<li class="nav-item"><a class="nav-link active" href="#">Active</a> </li>

</ul>

</div>

</nav>

<nav class="navbar navbar-expand-sm bg-primary navbar-dark">

<div class="container-fluid">

<ul class="navbar-nav">

<li class="nav-item"><a class="nav-link active" href="#">Active</a> </li>

</ul>

</div>

</nav>

```

Browser Output:



Fixed Navigation Bar

The navigation bar can also be fixed at the top or at the bottom of the page.

A fixed navigation bar stays visible in a fixed position (top or bottom) independent of the page scroll.

- The `.fixed-top` class makes the navigation bar fixed at the top:
- The `.fixed-bottom` class to make the navbar stay at the bottom of the page:

Example for Fixed Top

```
<nav class="navbar navbar-expand-sm bg-dark navbar-dark fixed-top">
<div class="container-fluid">
<a class="navbar-brand" href="#">Fixed top</a>
</div>
</nav>
```

Example for Fixed Bottom

```
<nav class="navbar navbar-expand-sm bg-dark navbar-dark fixed-bottom">
<div class="container-fluid">
<a class="navbar-brand" href="#">Fixed bottom</a>
</div>
</nav>
```

Sticky Navbar

The `.sticky-top` class to make the navbar fixed/stay at the top of the page when you scroll past it.

```
<nav class="navbar navbar-expand-sm bg-dark navbar-dark sticky-top">
<div class="container-fluid">
<a class="navbar-brand" href="#">Sticky top</a>
</div>
</nav>
```

Example for bootstrap navigation bar

```

<!DOCTYPE html>
<html lang="en">
<head>
<title>Example For Bootstrap Navbar</title>
<meta charset="utf-8">
<meta name="viewport" content="width=device-width, initial-scale=1">
<link href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.3/dist/css/bootstrap.min.css" rel="stylesheet">
<script src="https://cdn.jsdelivr.net/npm/bootstrap@5.3.3/dist/js/bootstrap.bundle.min.js"></script>
</head>
<body>
<nav class="navbar navbar-expand-sm bg-dark navbar-dark">
<div class="container-fluid">
<a class="navbar-brand" href="#">Logo</a>
<button class="navbar-toggler" type="button" data-bs-toggle="collapse" data-bs-target="#collapsibleNavbar">
<span class="navbar-toggler-icon"></span>
</button>
<div class="collapse navbar-collapse" id="collapsibleNavbar">
<ul class="navbar-nav">
<li class="nav-item">
<a class="nav-link" href="#">Link</a>
</li>
<li class="nav-item">
<a class="nav-link" href="#">Link</a>
</li>
<li class="nav-item">
<a class="nav-link" href="#">Link</a>
</li>
<li class="nav-item dropdown">

```

```

<a      class="nav-link      dropdown-toggle"      href="#"      role="button"      data-bs-
toggle="dropdown">Dropdown</a>
    <ul class="dropdown-menu">
        <li><a class="dropdown-item" href="#">Link</a></li>
        <li><a class="dropdown-item" href="#">Another link</a></li>
        <li><a class="dropdown-item" href="#">A third link</a></li>
    </ul>
</li>
</ul>
</div>
</div>
</nav>
<div class="container-fluid mt-3">
    <h3>Navbar With Dropdown</h3>
    <p>This example adds a dropdown menu in the navbar.</p>
</div>
</body>
</html>

```

Browser Output:

Logo
Link
Link
Link
Dropdown ▾

Navbar With Dropdown

This example adds a dropdown menu in the navbar.

Reference for Frontend Programming

1. [MDN Web Docs](#)
 - Comprehensive resource for HTML, CSS, and JavaScript documentation and tutorials.
 - Great for learning web standards and best practices.
2. [W3Schools](#)
 - Offers tutorials and references on web development languages.
 - Interactive coding examples and exercises.
3. [CSS-Tricks](#)
 - Articles, tutorials, and tips about CSS.
 - Includes guides on modern web development practices and techniques.
4. Bootstrap Official Documentation
 - Comprehensive guide to using Bootstrap.
 - Covers all components, utilities, and layout options.
 - Includes examples and customization options.
5. Bootstrap on W3Schools
 - Tutorials on using Bootstrap with examples.
 - Covers the basics to advanced features of Bootstrap.
6. [MDN Web Docs: Bootstrap](#)
 - Introduction to Bootstrap within the MDN learning area.
 - Explains how to integrate and use Bootstrap in projects.
7. <https://www.javatpoint.com/html-tutorial>
8. <https://www.javatpoint.com/css-tutorial>
9. <https://www.javatpoint.com/bootstrap-tutorial>
10. <https://www.geeksforgeeks.org/html-tutorial/>
11. <https://www.geeksforgeeks.org/css-tutorial/>



PYTHON PROGRAMMING

Introduction to Python

Python is a high-level, versatile programming language that has gained widespread popularity due to its readability, simplicity, and flexibility. Created by Guido van Rossum and first released in 1991, Python has since become one of the most popular programming languages for various applications, including web development, data science, desktop applications, artificial intelligence, automation, and more. Fortunately, for beginners, Python has a simple, easy-to-use syntax. This makes Python a great language to learn for beginners.

Features of Python programming language:

1. Free and Open Source:

Python language is freely available at the official website. Since it is open-source, this means that source code is also available to the public. So you can download it, use it as well as share it.

2. Easy to code:

Python is a high-level programming language. Python is very easy to learn the language as compared to other languages like C, C#, Javascript, Java, etc. It is very easy to code in the Python language and anybody can learn Python basics in a few hours or days. It is also a developer-friendly language.

3. Easy to Read:

As you will see, learning Python is quite simple. As was already established, Python's syntax is really straightforward. The code block is defined by the indentations rather than by semicolons or brackets.

4. Object-Oriented Language:

One of the key features of Python is Object-Oriented programming. Python supports object-oriented language and concepts of classes, object encapsulation, etc.

5. GUI Programming Support:

Graphical User interfaces can be made using a module such as PyQt5, PyQt4, wxPython, or Tk in Python. PyQt5 is the most popular option for creating graphical apps with Python.

6. High-Level Language:

Python is a high-level language. When we write programs in Python, we do not need to remember the system architecture, nor do we need to manage the memory.

7. Large Community Support:

Python has gained popularity over the years. Our questions are constantly answered by the enormous StackOverflow community. These websites have already provided answers to many questions about Python, so Python users can consult them as needed.

8. Easy to Debug:

Excellent information for mistake tracing. You will be able to quickly identify and correct the majority of your program's issues once you understand how to interpret Python's error traces. Simply by glancing at the code, you can determine what it is designed to perform.

9. Python is a Portable language:

Python language is also a portable language. For example, if we have Python code for Windows and if we want to run this code on other platforms such as Linux, Unix, and Mac then we do not need to change it, we can run this code on any platform.

10. Python is an Integrated language:

Python is also an Integrated language because we can easily integrate Python with other languages like C, C++, etc.

11. Interpreted Language:

Python is an Interpreted Language because Python code is executed line by line at a time. like other languages C, C++, Java, etc. there is no need to compile Python code this makes it easier to debug our code. The source code of Python is converted into an immediate form called bytecode.

12. Large Standard Library:

Python has a large standard library that provides a rich set of modules and functions so you do not have to write your own code for every single thing. There are many libraries present in Python such as regular expressions, unit-testing, web browsers, etc.

13. Dynamically Typed Language:

Python is a dynamically-typed language. That means the type (for example- int, double, long, etc.) for a variable is decided at run time not in advance because of this feature we don't need to specify the type of variable.

14. Frontend and backend development:

With a new project py script, you can run and write Python codes in HTML with the help of some simple tags <py-script>, <py-env>, etc. This will help you do frontend development work in Python like javascript. Backend is the strong forte of Python it's extensively used for this work cause of its frameworks like Django and Flask.

15. Allocating Memory Dynamically

In Python, the variable data type does not need to be specified. The memory is automatically allocated to a variable at runtime when it is given a value. Developers do not need to write int y = 18 if the integer value 15 is set to y. You may just type y=18.

Python Applications

We are living in a digital world that is completely driven by chunks of code. Every industry depends on software for its proper functioning be it healthcare, military, banking, research, and the list goes on. We have a huge list of programming languages that facilitate the software development process. As per a survey it is observed that python is the main coding language for more than 80% of developers. The main reason behind this is its extensive libraries and frameworks that fuel up the process.

Here are some notable areas where Python is commonly used:

1. Web Development:

- Frameworks like Django and Flask make it easy to build robust and scalable web applications.
- Python is used for server-side scripting and back-end development.

2. Data Science and Machine Learning:

- Libraries such as NumPy, pandas, and scikit-learn are extensively used for data manipulation, analysis, and machine learning.
- TensorFlow and PyTorch are popular frameworks for building and training machine learning models.

3. Artificial Intelligence:

- Python is widely adopted for AI research and development due to its simplicity and extensive libraries.
- Natural Language Processing (NLP) libraries like NLTK and spaCy are commonly used.

4. Automation and Scripting:

- Python is an excellent language for automating repetitive tasks and writing scripts.
- It is used for system administration, file manipulation, and process automation.

5. Game Development:

- Pygame is a popular library for game development in Python.
- Python is used for scripting and building game logic in various game engines.

6. Desktop GUI Applications:

- Libraries like Tkinter, PyQt, and Kivy enable the creation of desktop GUI applications.
- Python applications with graphical interfaces are used for tools, utilities, and desktop applications.

7. Networking and Cybersecurity:

- Python is widely used for network programming and scripting.
- Security professionals use Python for tasks like penetration testing and vulnerability scanning.

8. Databases:

- Python has libraries and frameworks for interacting with databases, such as SQLite, MySQL, and PostgreSQL.
- ORM (Object-Relational Mapping) tools like SQLAlchemy simplify database interactions.

9. Scientific Computing:

- Python is used in scientific research and computation with libraries like SciPy and Matplotlib.
- Jupyter Notebooks provide an interactive environment for scientific computing and data visualization.

10. IoT (Internet of Things):

- Python is used for IoT development, including controlling and collecting data from devices.
- MicroPython is a version of Python designed for microcontrollers and embedded systems.

11. Educational Purposes:

- Python's readability and simplicity make it an ideal language for teaching programming to beginners.
- Educational institutions use Python to introduce students to coding and computer science concepts.

12. Finance and Quantitative Analysis:

- Python is employed in finance for algorithmic trading, risk management, and financial analysis.
- Libraries like Pandas are used for handling financial data efficiently.

Python Syntax

- The Python syntax defines a set of rules that are used to create Python statements while writing a Python Program. Python was designed for readability, and has some similarities to the English language with influence from mathematics.
- Python uses new lines to complete a command, as opposed to other programming languages which often use semicolons or parentheses.
- Python relies on indentation, using whitespace, to define scope; such as the scope of loops, functions and classes. Other programming languages often use curly-brackets for this purpose.

Python Keywords or Reserved Words

Keywords are predefined, reserved words used in Python programming that have special meanings to the compiler. We cannot use a keyword as a variable name, function name, or any other identifier. They are used to define the syntax and structure of the Python language. All the keywords except True, False and None are in lowercase and they must be written as they are. Python contains thirty-five keywords in the most recent version, i.e., Python 3.8.

Python Keywords

False	Await	else	import	pass
None	break	except	in	raise
True	class	finally	is	return
and	continue	for	lambda	try
as	def	from	nonlocal	while
assert	def	global	not	with
async	elif	if	or	yield

Python Identifiers

Identifiers are the name given to variables, classes, methods, etc

Rules for Naming an Identifier

- Identifiers cannot be a keyword.
- Identifiers are case-sensitive.
- It can have a sequence of letters and digits. However, it must begin with a letter or `_`. The first letter of an identifier cannot be a digit.
- It's a convention to start an identifier with a letter rather `_`.
- Whitespaces are not allowed.
- We cannot use special symbols like `!`, `@`, `#`, `$`, and so on.

Some naming conventions for Python identifiers –

- Python Class names start with an uppercase letter. All other identifiers start with a lowercase letter.
- Starting an identifier with a single leading underscore indicates that the identifier is private identifier.
- Starting an identifier with two leading underscores indicates a strongly private identifier.
- If the identifier also ends with two trailing underscores, the identifier is a language-defined special name.

Python Comments

Comments can be used to explain Python code. Comments can be used to make the code more readable. Comments can be used to prevent execution when testing code There are three types of comments available in Python

1. Single line Comments
2. Multiline Comments
3. Docstring Comments

1. Single line Comments

A single-line comment of Python is the one that has a hashtag # at the beginning of it and continues until the finish of the line. If the comment continues to the next line, add a hashtag to the subsequent line and resume the conversation.

EX: `# This is a single line comment in python`

```
print ("Hello, World!")
```

2. Multiline Comments

Python does not provide a direct way to comment multiple line. You can comment multiple lines as follows

EX: `#This is comment`

```
#This is comment, too
```

```
#This is comment, too
```

Following triple-quoted string is also ignored by Python interpreter and can be used as a multiline comments: Another way of doing this is to use triple quotes, either `'''` or `"""`.

These triple quotes are generally used for multi-line strings. But if we do not assign it to any variable or function, we can use it as a comment.

EX:

```
''' This is
    a multiline
    comment. '''
```

3. Docstring Comments

Python docstrings provide a convenient way to provide a help documentation with Python modules, functions, classes, and methods. The **docstring** is then made available via the `__doc__` attribute.

EX: `def add(a, b):`

```
    """Function to add the value of a and b"""

    return a+b

print(add.__doc__)
```

The result is: Function to add the value of a and b

Use of Python Comment

1. Make Code Easier to Understand:
2. If we write comments in our code, it will be easier for future reference. Also, it will be easier for other developers to understand the code.
3. Using Comments for Debugging:
4. If we get an error while running the program, we can comment the line of code that causes the error instead of removing it.

Python Variables

Python variables are the reserved memory locations used to store values with in a Python Program. This means that when you create a variable you reserve some space in the memory. Based on the data type of a variable, Python interpreter allocates memory and decides what can be stored in the reserved memory. Therefore, by assigning different data types to Python variables, you can store integers, decimals or characters in these variables. Variables are containers for storing data values.

EX: `x = 50`

`y = 59.7`

`z = "GTTC"`

Every Python variable should have a unique name like a, b, c. A variable name can be meaningful like color, age, name etc. There are certain rules which should be taken care while naming a Python variable:

1. A variable name must start with a letter or the underscore character
2. A variable name cannot start with a number or any special character like \$, (, * % etc.
3. A variable name can only contain alpha-numeric characters and underscores (A-z, 0-9, and _)
4. Python variable names are case-sensitive which means Name and NAME are two different variables in Python.
5. Python reserved keywords cannot be used naming the variable.

Example:

```
counter = 1000
_count = 100
name1 = "GTTC"
name2 = "IoT"
Age = 20
net_salary = 100000
```

print(counter)	#Output is 1000
print(_count)	#Output is 100
print(name1)	#Output is GTTC
print(name2)	#Output is IoT
print(Age)	#Output is 20
print(net_salary)	#Output is 100000

Multi Words Variable Names

Variable names with more than one word can be difficult to read. There are several techniques you can use to make them more readable:

- 1. Camel Case:** Each word, except the first, starts with a capital letter:

```
EX: internetOfThings, kmPerHour, pricePerLitre
```

2. Pascal Case: Each word starts with a capital letter:

EX: InternetOfThings, KmPerHour, PricePerLitre

3. Snake Case: Each word is separated by an underscore character:

EX: internet_of_things, km_per_hour, price_per_litre

Python Basic Input and Output

Python Input

While programming, we might want to take the input from the user. In Python, we can use the `input()` function.

Syntax of `input()`

`input(prompt)`

Here, `prompt` is the string we wish to display on the screen. It is optional.

Example: Python User Input

```
# using input() to take user input
num = input('Enter a number: ')
print('You Entered:', num)
print('Data type of num:', type(num))
```

Output

```
Enter a number: 10
You Entered: 10
Data type of num: <class 'str'>
```

In the above example, we have used the `input()` function to take input from the user and stored the user input in the `num` variable.

It is important to note that the entered value 10 is a string, not a number. So, `type(num)` returns `<class 'str'>`.

To convert user input into a number we can use `int()` or `float()` functions as:

`num = int(input('Enter a number: '))`

Here, the data type of the user input is converted from string to integer.

Python Output

In Python, we can simply use the `print()` function to print output

Syntax of `print()`:

The python print function accepts 5 parameters

```
print(object= separator= end= file= flush=)
```

Here,

- object - value(s) to be printed
- sep (optional) - allows us to separate multiple objects inside `print()`.
- end (optional) - allows us to add specific values like new line "\n", tab "\t"
- file (optional) - where the values are printed. It's default value is `sys.stdout` (screen)
- flush (optional) - boolean specifying if the output is flushed or buffered. Default: False

Example 1: Python Print Statement

```
print('Good Morning!')
print('It is rainy today')
```

Output

```
Good Morning!
It is rainy today
```

In the above example, the `print()` statement only includes the object to be printed. Here, the value for `end` is not used. Hence, it takes the default value '\n'. So we get the output in two different lines.

Example 2: Python `print()` with `end` Parameter

```
# print with end whitespace
print('Good Morning!', end=' ')
print('It is rainy today')
```

Output

```
Good Morning! It is rainy today
```

Here we have included the `end= ''` after the end of the first `print()` statement. Hence, we get the output in a single line separated by space.

Example 3: Python print() with sep parameter

```
print('New Year', 2023, 'See you soon!', sep= ' . ')
```

Output

```
New Year. 2023. See you soon!
```

In the above example, the print() statement includes multiple items separated by a comma. Notice that we have used the optional parameter sep= ". " inside the print() statement. Hence, the output includes items separated by . not comma.

Example 4: Print Python Variables and Literals

For example,

```
number = -10.6  
  
name = “ Embedded Full Stack IoT”  
  
# print literals  
  
print(5)  
  
# print variables  
  
print(number)  
  
print(name)
```

Output

```
5  
  
-10.6  
  
Embedded Full Stack IoT
```

Example 5: Print Concatenated Strings

We can also join two strings together inside the print() statement.

For example,

```
print('Internet' + 'of' + 'Things.' )
```

Output

Internet of Things.

Here, the + operator joins two strings ‘Internet’ , ‘of’ ,and ‘Things.’ the print() function prints the joined string

Output formatting

Sometimes we would like to format our output to make it look attractive. This can be done by using the **str.format()** method. For example,

x = 5

y = 10

```
print("The value of x is {} and y is {}".format(x,y))
```

Here, the curly braces {} are used as placeholders. We can specify the order in which they are printed by using numbers (tuple index).

Python - Data Types

In computer programming, data types specify the type of data that can be stored inside a variable. Python Data Types are used to define the type of a variable. It defines what type of data we are going to store in a variable. The data stored in memory can be of many types. For example, a person's age is stored as a numeric value and his or her address is stored as alphanumeric characters.

Python has various built-in data types, They are

Data Types	Classes	Description
Numeric	int, float, complex	holds numeric values
String	str	holds sequence of characters
Sequence	list, tuple, range	holds collection of items
Mapping	dict	holds data in key-value pair form
Boolean	bool	holds either, True or False
Set	set, frozenset	hold collection of unique items

Since everything is an object in Python programming, data types are actually classes and variables are instances(object) of these classes.

Python Numeric Data Type

Python numeric data types store numeric values. Number objects are created when you assign a value to them.

Example: `a = 55`

`b = 39.5`

`c = 1 + 5j`

Python supports four different numerical types –

- `int` (signed integers)
- `long` (long integers, they can also be represented in octal and hexadecimal)
- `float` (floating point real values)
- `complex` (complex numbers)

Python String Data Type

- Python Strings are identified as a contiguous set of characters represented in the quotation marks.
- Python allows for either pairs of single or double quotes.
- Subsets of strings can be taken using the slice operator ([] and [:]) with indexes starting at 0 in the beginning of the string and working their way from -1 at the end.
- The plus (+) sign is the string concatenation operator and the asterisk (*) is the repetition operator in Python.

Example:

```
str = 'Hello World!'

print (str) # Prints complete string

print (str[0]) # Prints first character of the string

print (str[2:5]) # Prints characters starting from 3rd to 5th

print (str[2:]) # Prints string starting from 3rd character

print (str * 2) # Prints string two times

print (str + "TEST") # Prints concatenated string
```

RESULT:

```
Hello World!
H
llo
llo World!
Hello World!Hello World!
Hello World!TEST
```

Python String Methods

Data Types	Classes	Description
Numeric	int, float, complex	holds numeric values
String	str	holds sequence of characters
Sequence	list, tuple, range	holds collection of items
Mapping	dict	holds data in key-value pair form
Boolean	bool	holds either, True or False
Set	set, frozenset	hold collection of unique items

Python List Data Type

- Python Lists are the most versatile compound data types.
- Lists are used to store multiple items in a single variable.
- List items are ordered, changeable, and allow duplicate values
- List items are indexed, the first item has index [0], the second item has index [1], etc
- A Python list contains items separated by commas and enclosed within square brackets ([]).
- Python list can be of different data type.
- The values stored in a Python list can be accessed using the slice operator ([] and [:]) with indexes starting at 0 in the beginning of the list and working their way to end -1.
- The plus (+) sign is the list concatenation operator, and the asterisk (*) is the repetition operator.

Example:

```
list = [ 'abcd', 786 , 2.23, 'john', 70.2 ]  
  
tinylist = [123, 'john']  
  
print (list)  # Prints complete list  
  
print (list[0])  # Prints first element of the list  
  
print (list[1:3])  # Prints elements starting from 2nd till 3rd  
  
print (list[2:])  # Prints elements starting from 3rd element  
  
print (tinylist * 2)  # Prints list two times  
  
print (list + tinylist) # Prints concatenated lists
```

RESULT:

```
[‘abcd’, 786, 2.23, ‘john’ , 70.2]  
‘abcd’  
[786, 2.23]  
[ 2.23, ‘john’, 70.2]  
[123, ‘john’, 123, ‘john’]  
[‘abcd’, 786, 2.23, ‘john’ , 70.2, 123, ‘john’]
```

List Methods

Method	Description
append()	add an item to the end of the list
extend()	add all the items of an iterable to the end of the list
insert()	inserts an item at the specified index
remove()	removes item present at the given index
pop()	returns and removes item present at the given index
clear()	removes all items from the list
index()	returns the index of the first matched item
count()	returns the count of the specified item in the list
sort()	sort the list in ascending/descending order
reverse()	reverses the item of the list
copy()	returns the shallow copy of the list

Advantages of Lists:

- **Dynamic Size:** Lists in Python are dynamic, meaning they can grow or shrink in size during runtime. Elements can be added or removed without specifying the size beforehand. This makes lists very flexible for handling varying amounts of data.
- **Mutable:** Lists are mutable, which means you can modify their elements after creation. You can change, add, or remove items in a list.
- **Versatility:** Lists can hold elements of different data types. You can have integers, strings, objects, or even other lists within a single list.
- **Rich in Built-in Functions:** Lists come with a variety of built-in functions and methods for manipulation, such as append(), extend(), remove(), pop(), and more. These functions make it easy to work with lists.
- **Ease of Use:** Lists are easy to create and manipulate in Python, making them a popular choice for many programming tasks.

Disadvantages of Lists:

1. **Slower Operations** : Compared to arrays, lists in Python can be slower for certain operations. This is because lists are implemented as dynamic arrays, which sometimes require resizing and copying elements to accommodate new additions. This resizing operation can be time-consuming.
2. **Higher Memory Consumption** : Lists can consume more memory than arrays or tuples because of their dynamic and versatile nature. They need extra space to accommodate potential resizing and storing different types of objects.
3. **Less Efficient for Numerical Computations** : If you're working with numerical computations, arrays provided by libraries like NumPy are more efficient than lists. NumPy arrays are implemented in C and provide a significant performance boost for numerical operations.

Python Tuple Data Type

- Python tuple is another sequence data type that is similar to a list.
- Tuples are used to store multiple items in a single variable.
- A tuple is a collection which is ordered and unchangeable.
- Tuple items are ordered, unchangeable, and allow duplicate values.
- Unchangeable: Tuples are unchangeable, meaning that we cannot change, add or remove items after the tuple has been created.
- A Python tuple consists of a number of values separated by commas. Unlike lists, however, tuples are enclosed within parentheses.
- The main differences between lists and tuples are: Lists are enclosed in brackets ([]) and their elements and size can be changed, while tuples are enclosed in parentheses (()) and cannot be updated.
- Tuples can be thought of as read-only lists.

Examples:

```
tuple = ( 'abcd', 786 , 2.23, 'john', 70.2 )  
  
tinytuple = (123, 'john')  
  
print (tuple) # Prints the complete tuple  
  
print (tuple[0]) # Prints first element of the tuple  
  
print (tuple[1:3]) # Prints elements of the tuple starting from 2nd till 3rd  
  
print (tuple[2:]) # Prints elements of the tuple starting from 3rd element  
  
print (tinytuple * 2) # Prints the contents of the tuple twice  
  
print (tuple + tinytuple) # Prints concatenated tuples
```

RESULT:

```
(‘abcd’, 786, 2.23, ’john’ , 70.2)  
‘abcd’  
(786, 2.23)  
(2.23, ‘john’, 70.2)  
(123, ‘john’, 123, ‘john’)  
(‘abcd’, 786, 2.23, ’john’ , 70.2, 123, ‘john’)
```

Python Tuple Methods

There are only two tuple methods `count()` and `index()` that a tuple object can call.

1. Python Tuple `count()`: returns count of the element in the tuple
2. Python Tuple `index()`: returns the index of the element in the tuple

Count() Method

The count() method of Tuple returns the number of times the given element appears in the tuple.

Syntax: `tuple.count(element)`

Where the element is the element that is to be counted.

```
# Creating tuples
Tuple1 = (0, 1, 2, 3, 2, 3, 1, 3, 2)
Tuple2 = ('python', 'plc', 'python', 'cad', 'java', 'python', 'IoT')
# Count the appearance of 3
res = Tuple1.count(3)
print('Count of 3 in Tuple1 is:', res)
```

OUTPUT:

```
Count of 3 in Tuple1 is: 3
```

```
# count the appearance of python
res1 = Tuple2.count('python')
print('Count of Python in Tuple2 is:', res1)
```

OUTPUT:

```
Count of Python in Tuple2 is: 3
```

Index() Method

The Index() method returns the first occurrence of the given element from the tuple.

Syntax: `tuple.index(element, start, end)`

Parameters:

- element: The element to be searched.
- start (Optional): The starting index from where the searching is started
- end (Optional): The ending index till where the searching is done

Note: This method raises a Value Error if the element is not found in the tuple.

Example:

```
# Creating tuples  
  
Tuple = (0, 1, 2, 3, 2, 3, 1, 3, 2)  
  
# getting the index of 3  
  
res = Tuple.index(3)  
  
print('First occurrence of 3 is', res)
```

OUTPUT:

```
First occurrence of 3 is 3
```

```
# getting the index of 3 after 4th  
  
# index  
  
res = Tuple.index(3, 4)  
  
print('First occurrence of 3 after 4th index is:', res)
```

OUTPUT:

```
First occurrence of 3 after 4th index is: 5
```

Advantages of Tuple over List in Python

- 1. Immutable:** Tuples are immutable, meaning that their contents cannot be modified once they are created. This can be useful in situations where you want to prevent accidental changes to the data, or when you need to ensure that the data remains constant throughout the lifetime of the program.
- 2. Faster:** Tuples are generally faster than lists for accessing and iterating over elements. This is because tuples are implemented as a contiguous block of memory, whereas lists are implemented as an array of pointers to objects.
- 3. Memory-efficient:** Tuples are more memory-efficient than lists, especially when the size of the collection is small. This is because tuples do not require additional space to store their length or other metadata, unlike lists.
- 4. Good for data integrity:** Tuples can be used to enforce data integrity, meaning that you can ensure that certain data is always present and in a specific format. For example, you might use a tuple to represent a coordinate in a two-dimensional space, where the first element is the x-coordinate and the second element is the y-coordinate.
- 5. Suitable for use as dictionary keys:** Tuples can be used as dictionary keys, whereas lists cannot. This is because tuples are immutable, and therefore their contents cannot change, which makes them suitable for use as a key in a hash table.

Python Set Data Type

- Sets are used to store multiple items in a single variable
- A set is a collection which is unordered (Unordered means that the items in a set do not have a defined order.), unchangeable, and unindexed.
- Set items can appear in a different order every time you use them, and cannot be referred to by index or key.
- Sets cannot have two items with the same value
- Sets are written with curly brackets.

Create a Set in Python

A set can have any number of items and they may be of different types

Examples:

```
student_id = {112, 114, 116, 118, 115} # create a set of integer type
print('Student ID:', student_id)
```

Output:

```
Student ID: {112, 114, 115, 116, 118}
```

```
vowel_letters = {'a', 'e', 'i', 'o', 'u'} # create a set of string type
print('Vowel Letters:', vowel_letters)
```

Output:

```
Vowel Letters: {'u', 'a', 'e', 'i', 'o'}
```

```
mixed_set = {'Hello', 101, -2, 'Bye'} # create a set of mixed data types  
print('Set of mixed data types:', mixed_set)
```

Output:

```
Set of mixed data types: {'Hello', 'Bye', 101, -2}
```

Python Set Methods

Functions Name	Description
add()	Adds a given element to a set
clear()	Removes all elements from the set
copy()	Returns a shallow copy of the set
difference()	Returns a set that is the difference between two sets
difference_update()	Updates the existing caller set with the difference between two sets
discard()	Removes the element from the set
frozenset()	Return an immutable frozenset object
intersection()	Returns a set that has the intersection of all sets
intersection_update()	Updates the existing caller set with the intersection of sets
isdisjoint()	Checks whether the sets are disjoint or not
issubset()	Returns True if all elements of a set A are present in another set B
issuperset()	Returns True if all elements of a set A occupies set B
pop()	Returns and removes a random element from the set
remove()	Removes the element from the set
symmetric_difference()	Returns a set which is the symmetric difference between the two sets
symmetric_difference_update()	Updates the existing caller set with the symmetric difference of sets
union()	Returns a set that has the union of all sets
update()	Adds elements to the set

Built-in Functions with Python Set

Function	Description
all()	Returns True if all elements of the set are true (or if the set is empty).
any()	Returns True if any element of the set is true. If the set is empty, returns False.
enumerate()	Returns an enumerate object. It contains the index and value for all the items of the set as a pair.
len()	Returns the length (the number of items) in the set.
max()	Returns the largest item in the set.
min()	Returns the smallest item in the set.
sorted()	Returns a new sorted list from elements in the set (does not sort the set itself).
sum()	Returns the sum of all elements in the set.

Python Dictionaries Data Type

- Dictionaries are used to store data values in key : value pairs.
- A dictionary is a collection which is ordered*, changeable and do not allow duplicates.
- Dictionaries are written with curly brackets, and have keys and values:

Create a Dictionary

```
# creating a dictionary
country_capitals = {
    "United States": "Washington D.C.",
    "Italy": "Rome",
    "England": "London" }
print(country_capitals) # printing the dictionary
```

Output:

```
{'United States': 'Washington D.C.', 'Italy': 'Rome', 'England': 'London'}
```

Python Dictionary Methods

Functions Name	Descriptions
clear()	Removes all items from the dictionary
copy()	Returns a shallow copy of the dictionary
fromkeys()	Creates a dictionary from the given sequence
get()	Returns the value for the given key
items()	Return the list with all dictionary keys with values
keys()	Returns a view object that displays a list of all the keys in the dictionary in order of insertion
pop()	Returns and removes the element with the given key
popitem()	Returns and removes the key-value pair from the dictionary
setdefault()	Returns the value of a key if the key is in the dictionary else inserts the key with a value to the dictionary

values()	Returns a view object containing all dictionary values, which can be accessed and iterated through efficiently
update()	Updates the dictionary with the elements from another dictionary or an iterable of key-value pairs. With this method, you can include new data or merge it with existing dictionary entries

Python Boolean Data Type

- Python boolean type is one of the built-in data types provided by Python, which represents one of the two values i.e. True or False.
- Python bool() function allows you to evaluate the value of any expression and returns either True or False based on the expression.

```
# Declaring variables
a = 10
b = 20
# Comparing variables
print(a == b)
```

Output:

```
False
```

Python – Operators

Operators are used to perform operations on variables and values. Python operators are the constructs which can manipulate the value of operands. These are symbols used for the purpose of logical, arithmetic and various other operations.

Types of Python Operators

Python language supports the following types of operators.

1. Arithmetic Operators
2. Comparison (Relational) Operators
3. Assignment Operators
4. Logical Operators
5. Bitwise Operators
6. Membership Operators
7. Identity Operators

1. Python Arithmetic Operators

Python arithmetic operators are used to perform mathematical operations on numerical values. These operations are Addition, Subtraction, Multiplication, Division, Modulus, Exponents and Floor Division.

Operator	Name	Example
+	Addition	$10+20= 30$
-	Subtraction	$20 - 10 = 10$
*	Multiplication	$10 * 20 = 200$
/	Division	$20/ 10 = 2$
%	Modulus	$22 \% 10 = 2$
**	Exponent	$4 ** 2 = 16$
#	Floor Division	$9//2 = 4$

2. Python Comparison Operators

Python comparison operators compare the values on either sides of them and decide the relation among them. They are also called relational operators. These operators are equal, not equal, greater than, less than, greater than or equal to and less than or equal to.

Operator	Name	Examples
==	Equal	4==5 is not true
!=	Not Equal	4!=5 is true
>	Greater Than	4>5 is not true
<	Less Than	4<5 is true
>=	Greater than or Equal to	4>= is not true
<=	Less than or Equal to	4<=5 is true

3. Python Assignment Operators

Python assignment operators are used to assign values to variables. These operators include simple assignment operator, addition assign, subtraction assign, multiplication assign, division and assign operators etc.

Operator	Name	Example
=	Assignment Operator	a = 10
+=	Addition Assignment	a += 5 (Same as a = a + 5)
-=	Subtraction Assignment	a -= 5 (Same as a = a - 5)
*=	Multiplecaion Assignment	a *= 5 (Same as a = a * 5)
/=	Division Assignment	a /= 5 (Same as a = a/5)
%=	Remainder Assignment	a %= 5 (Same as a = a % 5)
**=	Exponent Assignment	a **= 2 (Same as a = a ** 2)
//=	Floor Division Assignment	a //= 3 (Same as a = a // 3)

4. Python Bitwise Operators

Bitwise operator works on bits and performs bit by bit operation. There are following Bitwise operators supported by Python language

Operator	Name	Example
&	Binary AND	Sets each bit to 1 if both bits are 1
	Binary OR	Sets each bit to 1 if one of two bits is 1
^	Binary XOR	Sets each bit to 1 if only one of two bits is 1
~	Binary Ones Complement	Inverts all the bits
<<	Binary Left Shift	Shift left by pushing zeros in from the right and let the left most bits fall off
>>	Binary RightShift	Shift right by pushing copies of the leftmost bit in from the left, and let the rightmost bits fall off

5. Python Logical Operators

There are following logical operators supported by Python language.

Operator	Name	Example
and Logical AND	If both the operands are true then condition becomes true.	(a and b) is true.
or Logical OR	If any of the two operands are non-zero then condition becomes true.	(a or b) is true.
not Logical NOT	Used to reverse the logical state of its operand.	Not(a and b) is false.

6. Python Membership Operators

Python's membership operators test for membership in a sequence, such as strings, lists, or tuples. There are two membership operators

Operator	Description	Example
in	Evaluates to true if it finds a variable in the specified sequence and false otherwise.	x in y, here in results in a 1 if x is a member of sequence y.
not in	Evaluates to true if it does not find a variable in the specified sequence and false otherwise.	x not in y, here not in results in a 1 if x is not a member of sequence y.

7. Python Identity Operators

Identity operators compare the memory locations of two objects. There are two Identity operators

Operator	Description	Example
is	Evaluates to true if the variables on either side of the operator point to the same object and false otherwise.	x is y, here is results in 1 if id(x) equals id(y).
is not	Evaluates to false if the variables on either side of the operator point to the same object and true otherwise.	x is not y, here is not results in 1 if id(x) is not equal to id(y).

Python Operators Precedence

The following table lists all operators from highest precedence to lowest.

Operators	Meaning
()	Parentheses
**	Exponent
+x, -x, ~x	Unary plus, Unary minus, Bitwise NOT
*, /, //, %	Multiplication, Division, Floor division, Modulus
+, -	Addition, Subtraction
<<, >>	Bitwise shift operators
&	Bitwise AND
^	Bitwise XOR
	Bitwise OR
==, !=, >, >=, <, <=, is, is not, in, not in	Comparisons, Identity, Membership operators
not	Logical NOT
and	Logical AND
or	Logical OR
Operator	Description
**	Exponentiation (raise to the power)
~+-	Complement, unary plus and minus (method names for the last two are +@ and -@)
* / % //	Multiply, divide, modulo and floor division
+-	Addition and subtraction
>> <<	Right and left bitwise shift
&	Bitwise 'AND'td>
^	Bitwise exclusive 'OR' and regular 'OR'
<= < >>=	Comparison operators
<> ==!=	Equality operators
= %= /= //= -= += *= **=	Assignment operators
is is not	Identity operators
in not in	Membership operators
not or and	Logical operators

Python Condition Statements

Python - Decision Making / Python Condition Statements

Decision making is anticipation of conditions occurring while execution of the program and specifying actions taken according to the conditions. Decision structures evaluate multiple expressions which produce TRUE or FALSE as outcome.

To determine which action to take and which statements to execute if outcome is TRUE or FALSE otherwise.

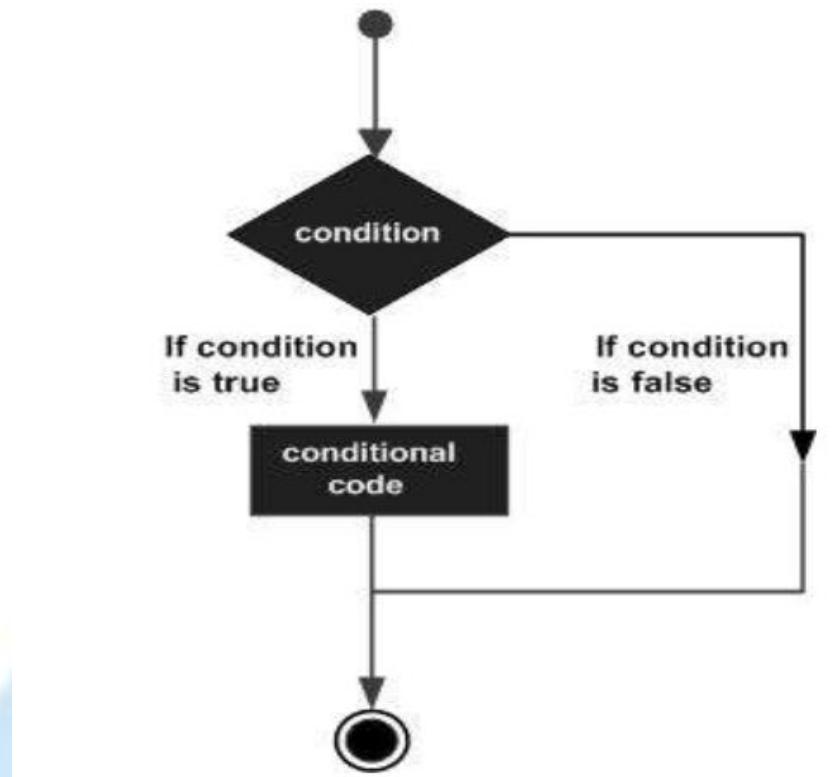


Fig.:Python condition statement flow diagram

Python programming language assumes any non-zero and non-null values as TRUE, and if it is either zero or null, then it is assumed as FALSE value. Decision-making statements in programming languages decide the direction of the flow of program execution. In Python, if-else elif statement is used for decision making.

if statement

if statement is the most simple decision-making statement. It is used to decide whether a certain statement or block of statements will be executed or not i.e if a certain condition is true then a block of statement is executed otherwise not.

Syntax:

if condition:

statement 1

statement 2

Python uses indentation to identify a block. So the block under an if statement will be identified

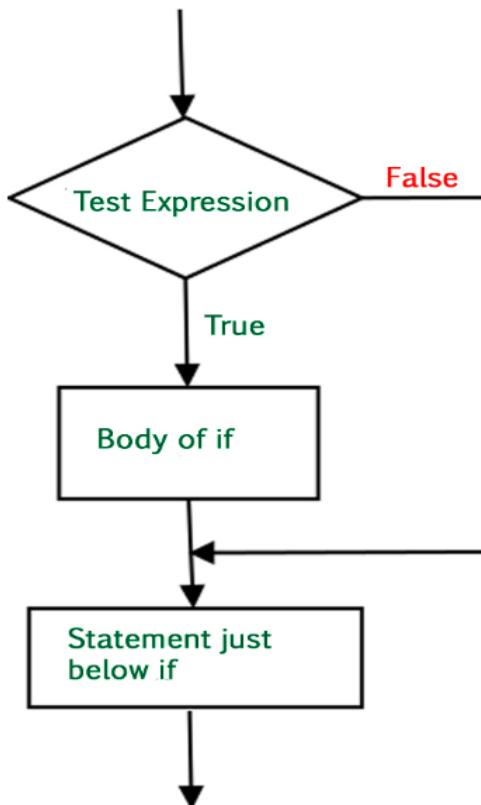


Fig.:Flowchart of Python if statement

- Here, the condition after evaluation will be either true or false.
- if the statement accepts Boolean values – if the value is true then it will execute the block of statements below it otherwise not.

Example:

```
a = 33  
  
b = 200  
  
if b > a:  
  
    print("b is greater than a")
```

Output:

```
b is greater than a
```

if-else

The if statement alone tells us that if a condition is true it will execute a block of statements and if the condition is false it won't. But what if we want to do something else if the condition is false. Here comes the else statement. We can use the else statement with if statement to execute a block of code when the condition is false.

Syntax:

```
if (condition):  
    # Executes this block if  
    # condition is true  
  
else:  
    # Executes this block if  
    # condition is false
```

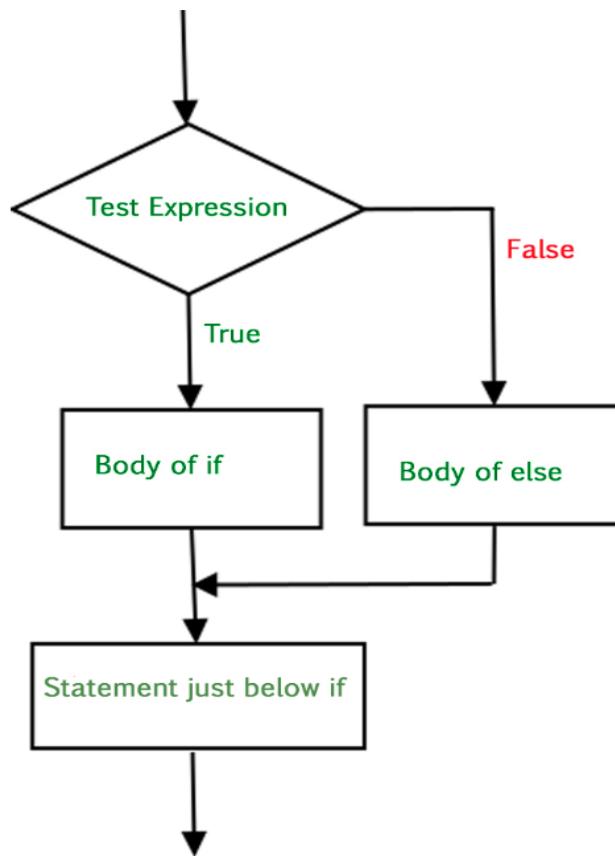


Fig.:Flow Chart of Python If-else statement

Example:

```

a = 200
b = 33
if b > a:
    print("b is greater than a")
else:
    print("b is not greater than a")
  
```

Output:

b is not greater than a

nested-if

A nested if is an if statement that is the target of another if statement. Nested if statements mean an if statement inside another if statement. Python allows us to nest if statements within if statements. i.e, we can place an if statement inside another if statement.

Syntax:

```
if (condition1):
    # Executes when condition1 is true
    if (condition2):
        # Executes when condition2 is true
        # if Block is end here
    # if Block is end here
```

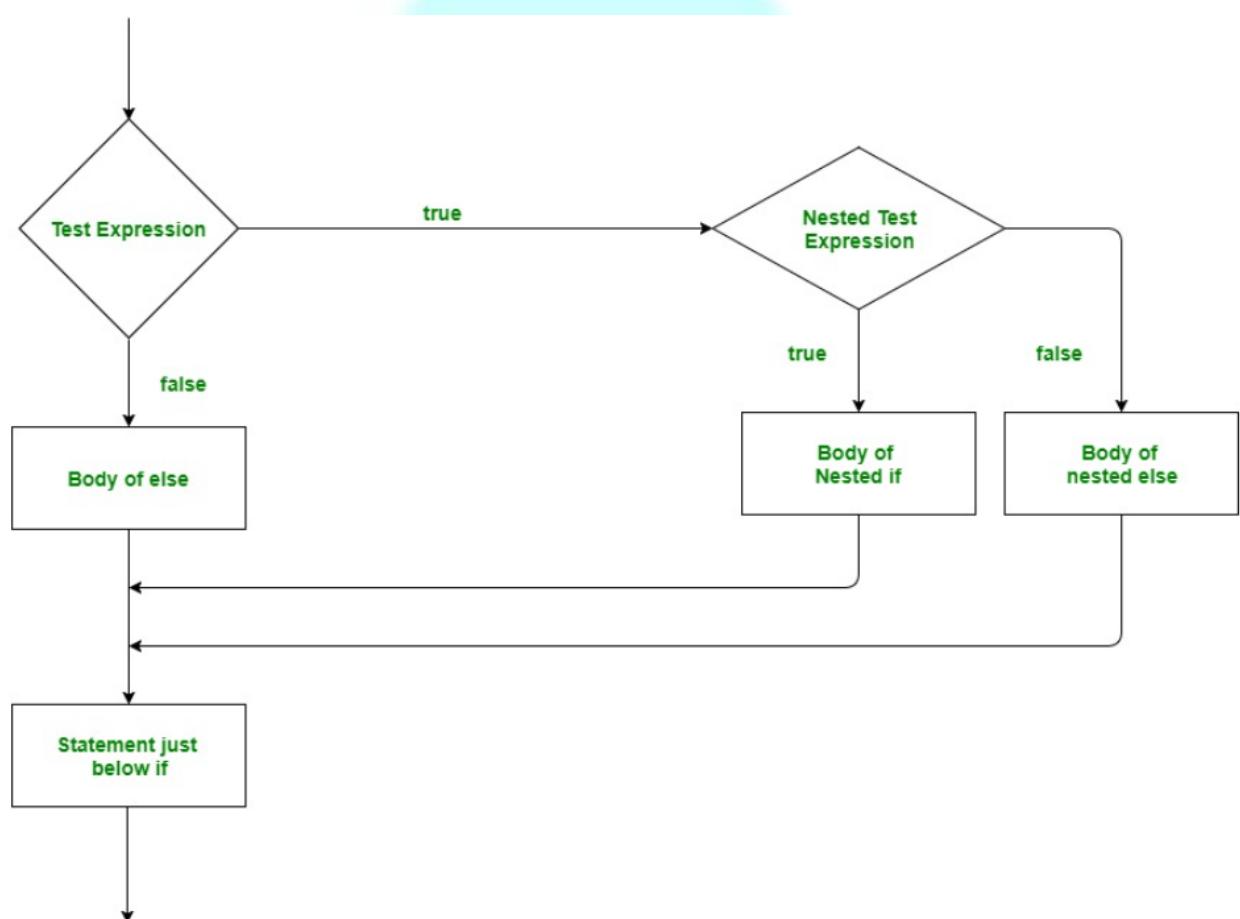


Fig.:Flowchart of Python Nested if Statement

Example:

```
x = 41

if x > 10:
    print("Above ten,")
    if x > 20:
        print("and also above 20!")
    else:
        print("but not above 20.")
```

Output:

```
Above ten
and also above 20!
```

if-elif-else ladder

Here, a user can decide among multiple options. The if statements are executed from the top down. As soon as one of the conditions controlling the if is true, the statement associated with that if is executed, and the rest of the ladder is bypassed. If none of the conditions is true, then the final else statement will be executed.

Syntax:

```
if (condition):
    statement
elif (condition):
    statement
.
.
else:
    statement
```

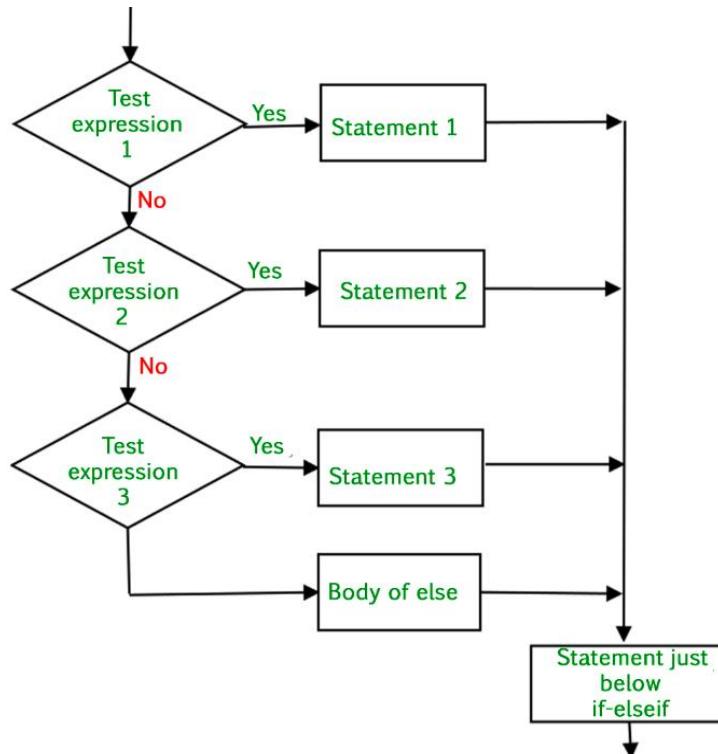


Fig.:Flow Chart of Python if elif else statements

Example:

```

z = 3
if z % 2 == 0:
    print("z is divisible by 2")
elif z % 3 == 0:
    print("z is divisible by 3")
else:
    print("z is neither divisible by 2 nor by 3")
  
```

Output:

z is divisible by 3

Python-Loops

A loop statement allows us to execute a statement or group of statements multiple times. The following diagram illustrates a loop statement

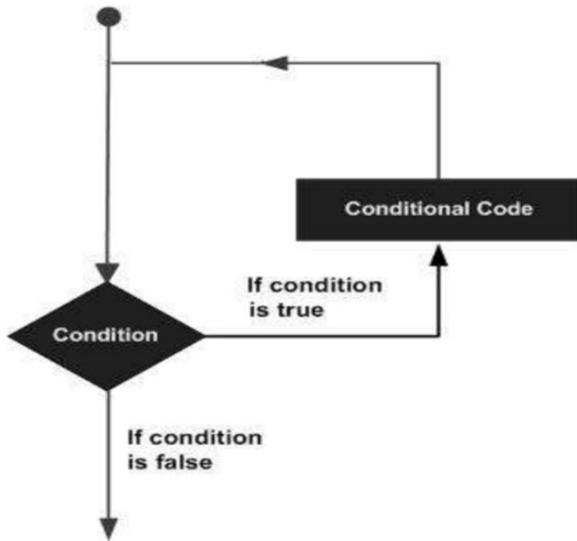


Fig.:Illustration of loop statement

Python programming language provides following types of loops to handle looping requirements.

- **While loop:** Repeats a statement or group of statements while a given condition is TRUE. It tests the condition before executing the loop body.
- **For loop:** Executes a sequence of statements multiple times and abbreviates the code that manages the loop variable.
- **Nested loop:** You can use one or more loop inside any another while, for or do while loop.

Python While Loop

A while loop statement in Python programming language repeatedly executes a target statement as long as a given condition is true.

Syntax:

While expression:

Statement(s)

- Here, **statement(s)** may be a single statement or a block of statements.
- The **condition** may be any expression, and true is any non-zero value. The loop iterates while the condition is true.
- When the condition becomes false, program control passes to the line immediately following the loop.
- Python uses indentation as its method of grouping statements.

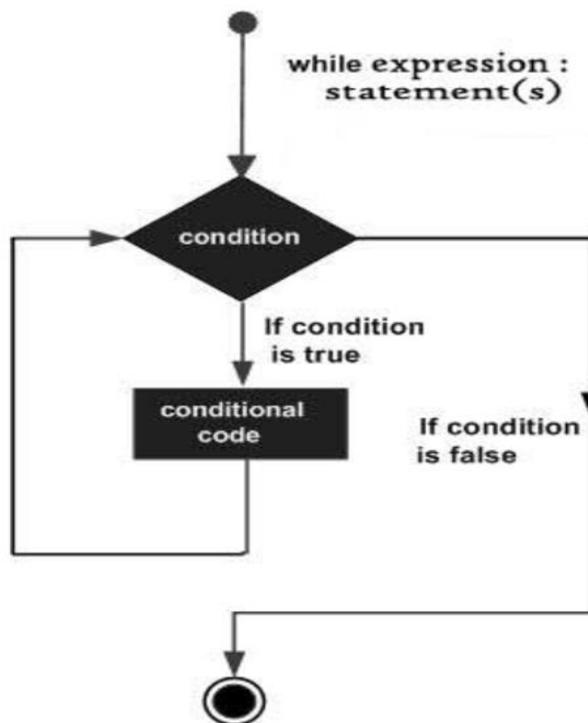


Fig.:While Loop Flowchart

In while loop is that the loop might not ever run. When the condition is tested and the result is false, the loop body will be skipped and the first statement after the while loop will be executed.

Python for Loop

It has the ability to iterate over the items of any sequence, such as a list or a string.

Syntax:

```

for iterating_var in sequence
    statements(s)
  
```

If a sequence contains an expression list, it is evaluated first. Then, the first item in the sequence is assigned to the iterating variable `iterating_var`. Next, the `statements` block is executed. Each item in the list is assigned to `iterating_var`, and the `statement(s)` block is executed until the entire sequence is exhausted.

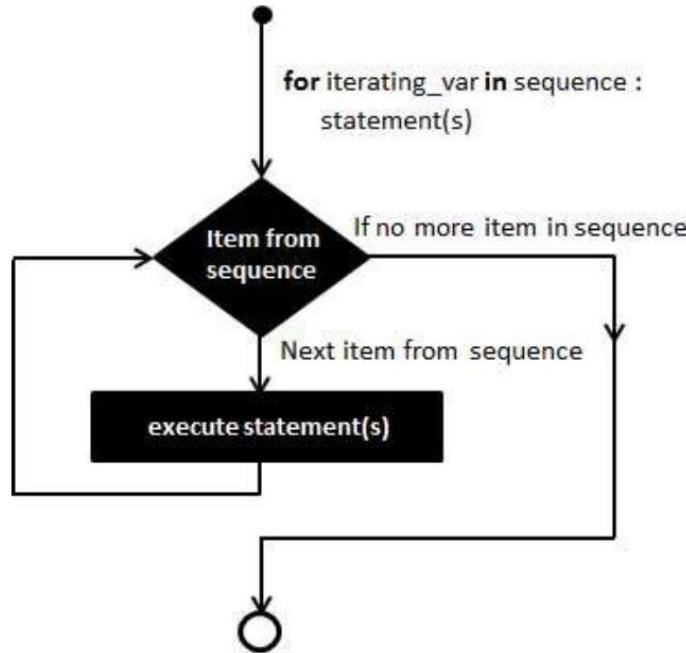


Fig.:For Loop Flow Chart

Python nested loops

Python programming language allows to use one loop inside another loop.

The syntax for nested for loop:

```

for iterating_var in sequence:
    for iterating_var in sequence:
        statements(s)
        statements(s)
  
```

The syntax for nested while loop:

```

while expression:
    while expression:
        statement(s)
        statement(s)
  
```

In loop nesting is that you can put any type of loop inside of any other type of loop. For example a for loop can be inside a while loop or vice versa.

Loop Control Statements

Loop control statements change execution from its normal sequence. When execution leaves a scope, all automatic objects that were created in that scope are destroyed. Python supports the following control statements.

1. **Break statement** : Terminates the loop statement and transfers execution to the statement immediately following the loop
2. **Continue statement** : Causes the loop to skip the remainder of its body and immediately retest its condition prior to reiterating.
3. **Pass statement** : The pass statement in Python is used when a statement is required syntactically but you do not want any command or code to execute.

Python break statement

It terminates the current loop and resumes execution at the next statement. The most common use for break is when some external condition is triggered requiring a hasty exit from a loop. The **break** statement can be used in both *while* and *for* loops.

The syntax for a **break** statement in Python is: **break**

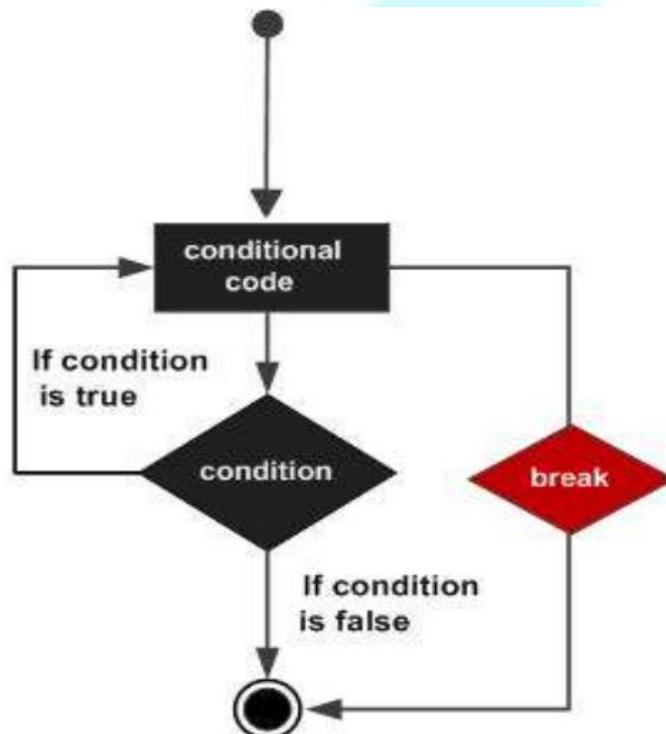


Fig.:Flow Chart of Python break statement

Python continue statement

It returns the control to the beginning of the while loop. The **continue** statement rejects all the remaining statements in the current iteration of the loop and moves the control back to the top of the loop. The **continue** statement can be used in both *while* and *for* loops.

Syntax: **continue**

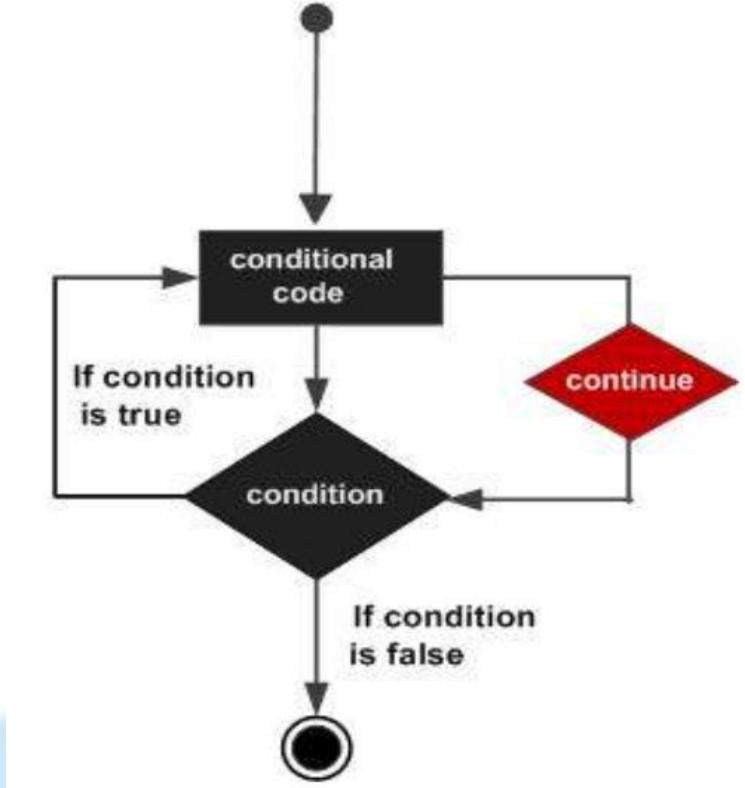


Fig.:Flow Chart of Python continue statement

Python pass Statement

It is used when a statement is required syntactically but you do not want any command or code to execute. The pass statement is a null operation; nothing happens when it executes. The difference between pass and comment is that comment is ignored by the interpreter whereas pass is not ignored. The pass is also useful in places where your code will eventually go, but has not been written yet

Syntax: **pass**

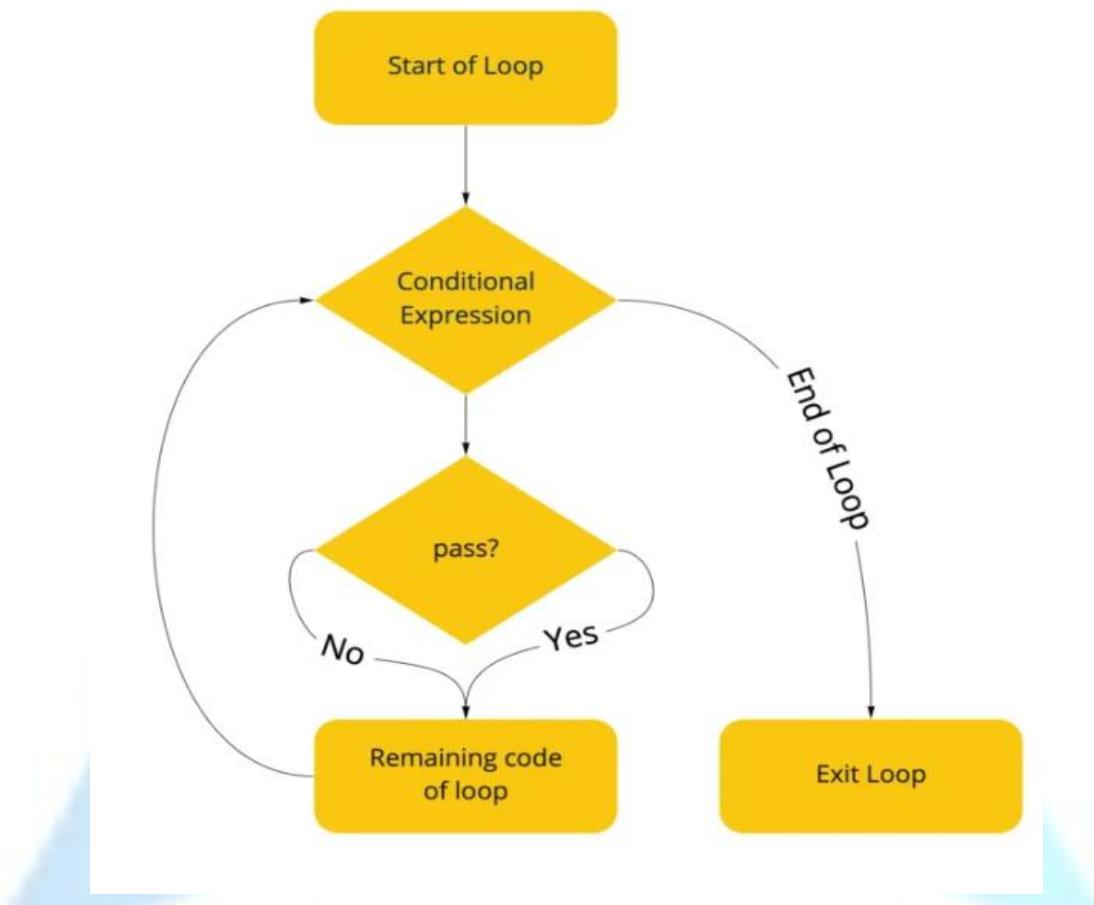


Fig.:Flow Chart of Python pass statement

Python Functions

A function is a block of code that performs a specific task. Function helps Dividing a complex problem into smaller chunks makes our program easy to understand and reuse.

Benefits of Using Functions

1. **Code Reusable** - We can use the same function multiple times in our program which makes our code reusable.
2. **Code Readability** - Functions help us break our code into chunks to make our program readable and easy to understand.

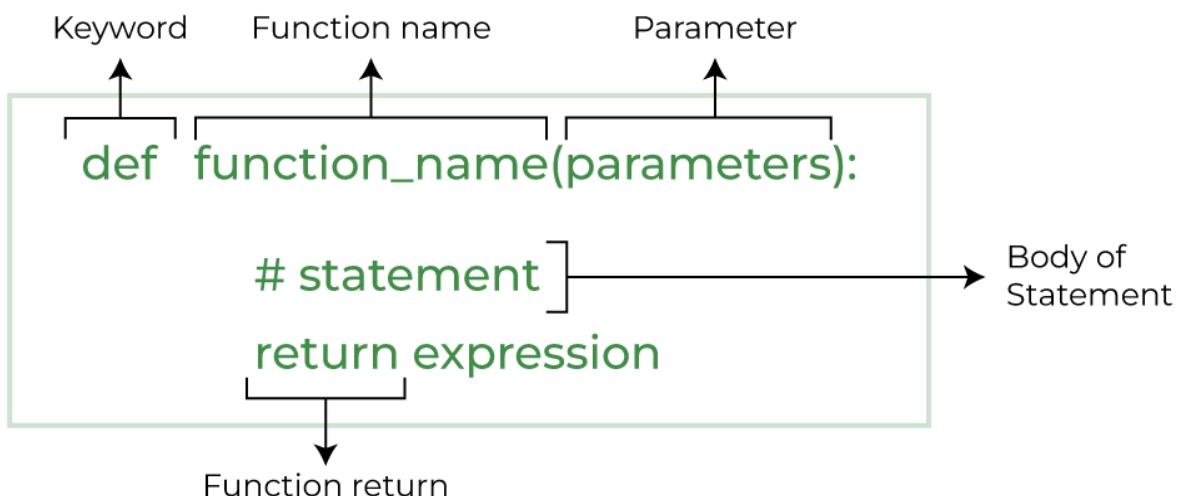


Fig.:syntax to declare a function

Here,

- **def** - keyword used to declare a function
- **function_name** - any name given to the function
- **parameters** -any value passed to function
- **return - (optional)** - returns value from a function

There are two types of function in Python programming:

1. **Standard library functions** - These are built-in functions in Python that are available to use.

Examples: print(), abs(), filter(), set(), map(), len(), pow() etc.

2. **User-defined functions** - We can create our own functions based on our requirements.

Creating a Function in Python

We can create a user-defined function in Python, using the **def** keyword. We can add any type of functionalities and properties to it as we require.

```
# A simple python function  
  
def fun( ):  
  
    print("Well Come To GTTC")  
  
# Driver code to call a function  
  
fun()
```

OUTPUT:

```
Well Come To GTTC
```

After creating a function in Python we can call it by using the name of the function followed by parenthesis containing parameters of that particular function.

When the function is called, the control of the program goes to the function definition. All codes inside the function are executed. The control of the program jumps to the next statement after the function call.

Python Function Arguments

A function can also have arguments. An argument is a value that is accepted by a function.

In this example the function `isadd_numbers` with arguments are `num1`, `num2`

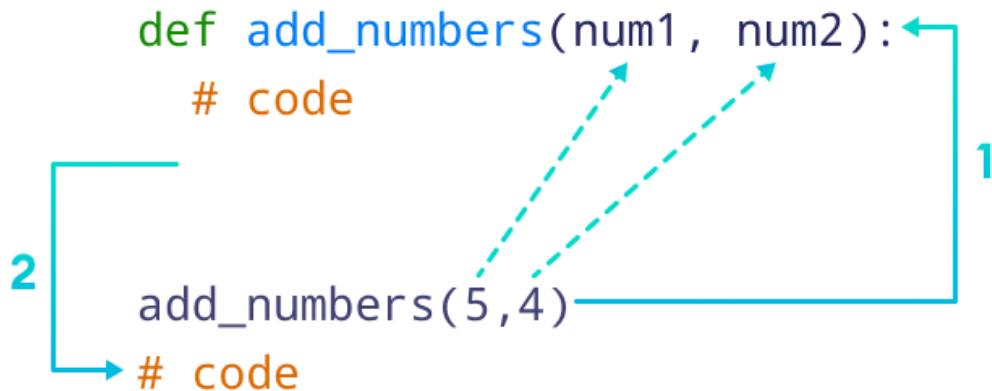


Fig.:syntax to declare function arguments

Python supports various types of arguments that can be passed at the time of the function call.
In Python, we have the following 4 types of function arguments.

1. **Default argument:** A default argument is a parameter that assumes a default value if a value is not provided in the function call for that argument.
2. **Keyword arguments (named arguments):** The idea is to allow the caller to specify the argument name with values so that the caller does not need to remember the order of parameters.
3. **Positional arguments**
4. **Arbitrary arguments** (variable-length arguments `*args` and `**kwargs`)

Python Function Arguments Example

```
# function with two arguments

def add_numbers(num1, num2):

    sum = num1 + num2

    print("Sum: ",sum)

# function call with two values

add_numbers(5, 4)

# Output: Sum: 9
```

The return Statement in Python

A Python function may or may not return a value. If we want our function to return some value to a function call, we use the return statement.

```
# function with return type

def find_square(num):

    result = num * num

    return result

# function call

square = find_square(3)

print(' Square: ' , square)

# Output: Square: 9
```

Function Argument with Default Values

```
def add_numbers( a = 7, b = 8):  
    sum = a + b  
    print('Sum:', sum)  
  
# 1. function call with two arguments  
add_numbers(2, 3)  
  
# 2. function call with one argument  
add_numbers(a = 2)  
  
# 3. function call with no arguments  
add_numbers()
```

1. add_number(2, 3)

Both values are passed during the function call. Hence, these values are used instead of the default values.

Output: Sum: 5

2. add_number(2)

Only one value is passed during the function call. So, according to the positional argument 2 is assigned to argument a, and the default value is used for parameter b

Output: Sum: 10

3. add_number()

No value is passed during the function call. Hence, default value is used for both parameters a and b

Output:

Sum: 15

Python Keyword Argument

In keyword arguments, arguments are assigned based on the name of arguments. In keyword arguments the position of arguments doesn't matter

```
def display_info(first_name, last_name):
    print('First Name:', first_name)
    print('Last Name:', last_name)
display_info(last_name = 'Hubli', first_name = 'GTTC')
```

Output:

First Name: GTTC

Last Name: Hubli

Python Function With Arbitrary Arguments

Sometimes, we do not know in advance the number of arguments that will be passed into a function. To handle this kind of situation, we can use arbitrary arguments in Python. Arbitrary arguments allow us to pass a varying number of values during a function call. An asterisk (*) before the parameter name to denote this kind of argument.

```
# Program to find sum of multiple numbers
def find_sum(*numbers):
    result = 0
    for num in numbers:
        result = result + num
    print("Sum = ", result)
# function call with 3 arguments
find_sum(1, 2, 3)
# function call with 2 arguments
find_sum(4, 9)
```

In the example, we have created the function `find_sum()` that accepts arbitrary arguments. Here, we are able to call the same function with different arguments. After getting multiple values, numbers behave as an array so we are able to use the for loop to access each value.

```
find_sum(1, 2, 3)
```

Output: Sum = 5

```
find_sum(4, 9)
```

Output: Sum = 13

Python Recursion

Recursion is the process of defining something in terms of itself. In Python, the function can call other functions. It is even possible for the function to call itself. These types of constructs are termed as recursive functions.

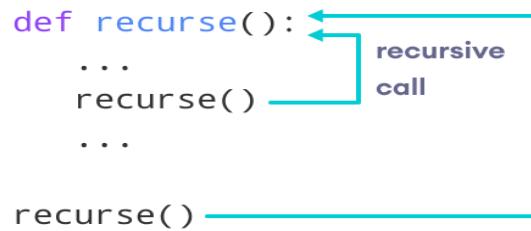


Fig.:Syntax for python recursion

Example of a recursive function

```
# This is a recursive function to find the factorial of an integer

def factorial(x):
    if x == 1:
        return 1
    else:
        return (x * factorial(x-1))

num = 3
print("The factorial of", num, "is", factorial(num))
```

Output: The factorial of 3 is 6

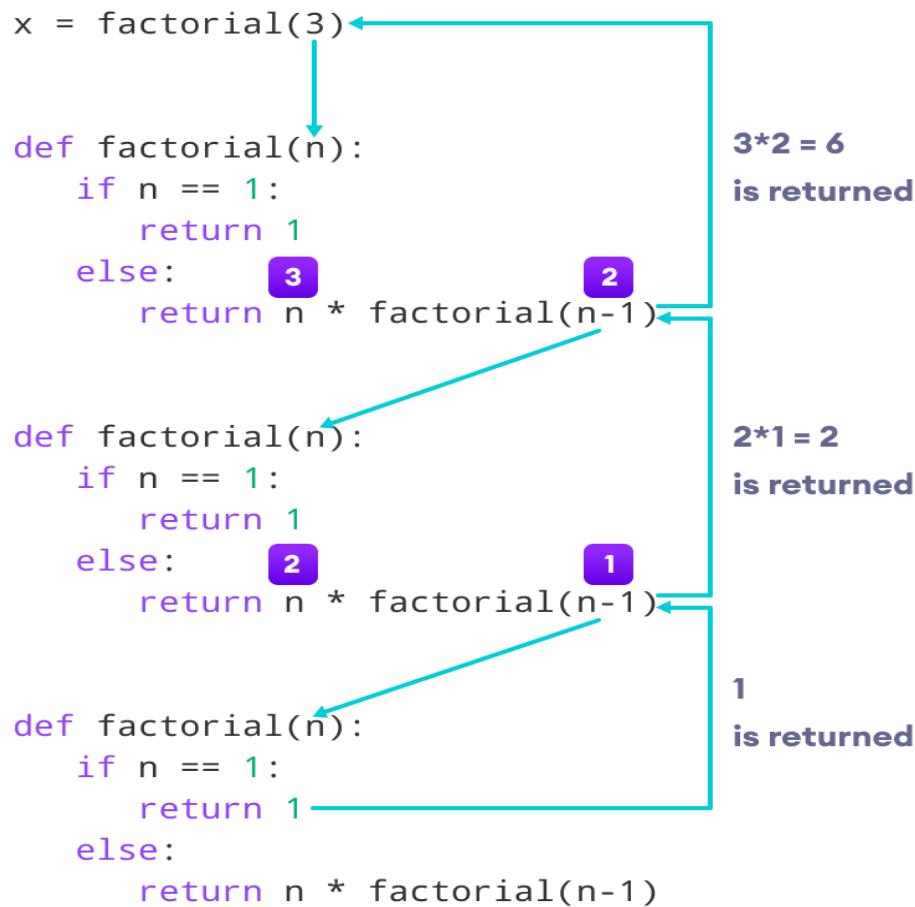


Fig.:Example of a recursive function

Advantages of Recursion

- Recursive functions make the code look clean and elegant.
- A complex task can be broken down into simpler sub-problems using recursion.
- Sequence generation is easier with recursion than using some nested iteration.

Disadvantages of Recursion

- Sometimes the logic behind recursion is hard to follow through.
- Recursive calls are expensive (inefficient) as they take up a lot of memory and time.
- Recursive functions are hard to debug.

Python Lambda/Anonymous Function

In Python, a lambda function is a special type of function without the function name. The `lambda` keyword is used instead of `def` to create a lambda function.

The syntax to declare the lambda function is

```
lambda argument(s) : expression
```

Here,

argument(s) - any value passed to the lambda function

expression - expression is executed and returned

```
# declare a lambda function
course = lambda : print('Internet of Things')

# call lambda function
course()

# Output: Internet of Things
```

In this example, we have defined a lambda function and assigned it to the `course` variable. When we call the lambda function, the `print()` statement inside the lambda function is executed.

Python lambda Function with an Argument

Similar to normal functions, the lambda function can also accept arguments.

```
# lambda that accepts one argument
course_name = lambda name : print('The course is,', name)

# lambda call
course_name('Internet of Things')
course_name('Programable Logic Controller')
```

Output: The course is, Internet of Things

Output: The course is, Programable Logic Controller

Python lambda function with filter()

The filter() function in Python takes in a function and an inerrable (lists, tuples, and strings) as arguments. The function is called with all the items in the list and a new list is returned which contains items for which the function evaluates to True.

```
# Program to filter out only the even items from a list
```

```
my_list = [1, 5, 4, 6, 8, 11, 3, 12]
```

```
new_list = list(filter(lambda x: (x%2 == 0) , my_list))
```

```
print(new_list)
```

Output: [4, 6, 8, 12]

Here, the filter() function returns only even numbers from a list

Python lambda function with map()

The map() function in Python takes in a function and an inerrable (lists, tuples, and strings) as arguments. The function is called with all the items in the list and a new list is returned which contains items returned by that function for each item.

```
# Program to double each item in a list using map()
```

```
my_list = [1, 5, 4, 6, 8, 11, 3, 12]
```

```
new_list = list(map(lambda x: x * 2 , my_list))
```

```
print(new_list)
```

Output: [2, 10, 8, 12, 16, 22, 6, 24]

Here, the map() function doubles all the items in a list.

Python Arrays

The Array is a process of memory allocation. It is performed as a dynamic memory allocation. It is a container that can hold a fixed number of items, and these items should be the same type. The Array is an idea of storing multiple items of the same type together, making it easier to calculate the position of each element by simply adding an offset to the base value. A combination of the arrays could save a lot of time by reducing the overall size of the code. It is used to store multiple values in a single variable.

Array Representation

Arrays can be declared in various ways in different languages

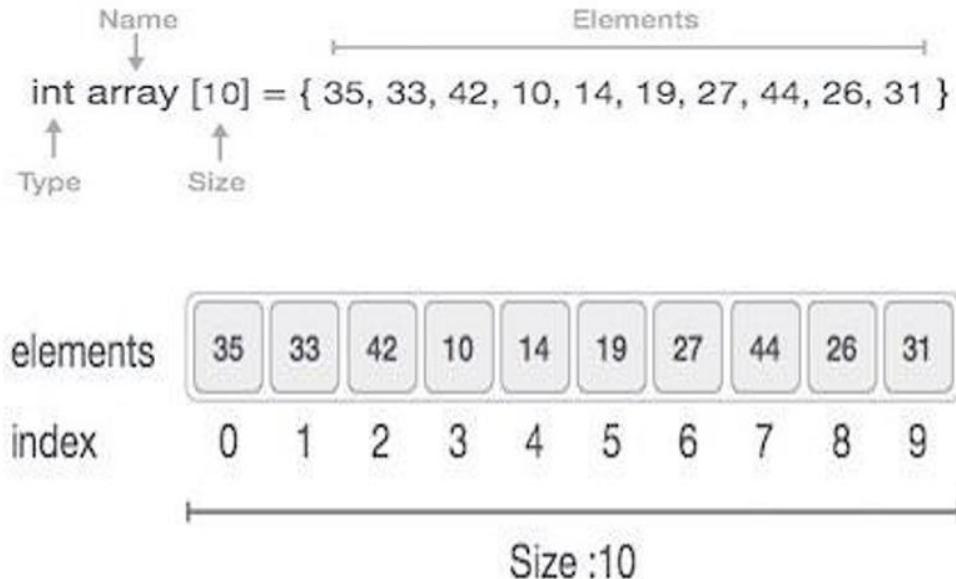


Fig.:Array Representation

- As per the diagram Index starts with 0.
- Array length is 10, which means it can store 10 elements.
- Each element can be accessed via its index. For example, we can fetch an element at index 6 as 27.

Basic Operations Of Array

The basic operations supported by an array are

- Traverse – print all the array elements one by one.
- Insertion – Adds an element at the given index.
- Deletion – Deletes an element at the given index.
- Search – Searches an element using the given index or by the value.
- Update – Updates an element at the given index.

Creating an Array

- The array can be handled in Python by a module named array.
- Array in Python can be created by importing an array module.

```
from array import *
arrayName = array(typecode, [Initializers])
```

Here, **typecode** are the codes that are used to define the type of value the array will hold

Typecode	Value
B	Represents signed integer of size 1 byte
B	Represents unsigned integer of size 1 byte
C	Represents character of size 1 byte
i	Represents signed integer of size 2 bytes
I	Represents unsigned integer of size 2 bytes
f	Represents floating point of size 4 bytes
d	Represents floating point of size 8 bytes

Typecodes

Example:

```
from array import *
array1 = array('i', [10,20,30,40,50])
for x in array1
    print(x)
```

Here **array1** is the name of the array

Output:

```
10  
20  
30  
40  
50
```

Accessing Array Element

We can access each element of an array using the index of the element.

Example:

```
from array import *  
  
array1 = array('i', [10,20,30,40,50])  
  
print (array1[0])  
  
print (array1[2])
```

Output:

```
10  
30
```

Insertion Operation

Insert operation is to insert one or more data elements into an array. Based on the requirement, a new element can be added at the beginning, end, or any given index of array.

Example:

```
from array import *  
  
array1 = array('i', [10,20,30,40,50])  
  
array1.insert(1,60) for x in array1:  
  
    print(x)
```

It produces the following result which shows the element is inserted at index position 1.

Output:

```
10  
60  
20  
30  
40  
50
```

Deletion Operation:

Deletion refers to removing an existing element from the array and re-organizing all elements of an array.

Example

Here, we remove a data element at the middle of the array using the python in-built remove() method.

```
from array import *  
  
array1 = array('i', [10,20,30,40,50])  
  
array1.remove(40)  
  
for x in array1:  
  
    print(x)
```

Output:

It produces the following result which shows the element is removed from the array.

```
10  
20  
30  
40  
50
```

Search Operation

Search for an array element based on its value or its index.

Example

Here, we search a data element using the python in-built index() method.

```
from array import *  
  
array1 = array('i', [10,20,30,40,50])  
  
print (array1.index(40))
```

Output:

It produces the following result which shows the index of the element. If the value is not present in the array then the program returns an error.- “3”.

Update Operation

- Update operation refers to updating an existing element from the array at a given index.
Example
- Here, we simply reassign a new value to the desired index we want to update.
- From array import * array1 = array('i', [10,20,30,40,50]) array1[2] = 80 for x in array1:
print(x)
- When we compile and execute the above program, it produces the following result which shows the new value at the index position 2.

Output:

10

20

80

40

50

Python Classes and Objects

- A class is a user-defined blueprint or prototype from which objects are created.
- Classes provide a means of bundling data and functionality together.
- Creating a new class creates a new type of object, allowing new instances of that type to be made. Each class instance can have attributes attached to it for maintaining its state.
- Class instances can also have methods (defined by their class) for modifying their state.
- The class creates a user-defined data structure, which holds its own data members and member functions, which can be accessed and used by creating an instance of that class.

Python Classes

The class creates a user-defined data structure, which holds its own data members and member functions, which can be accessed and used by creating an instance of that class.

A class is like a blueprint for an object.

Some points on Python class:

- Classes are created by keyword class.
- Attributes are the variables that belong to a class.
- Attributes are always public and can be accessed using the dot (.) operator.

Eg.: My class.Myattribute

Syntax: Class Definition:

```
class ClassName:  
    # Statement
```

Creating a Python Class:

- Here, the class keyword indicates that you are creating a class followed by the name of the class

Example:

```
class Dog:  
    sound = "bark"  
  
class Person:  
    def __init__(self, name, age):  
        # This is the constructor method that is called when creating a new Person object  
        # It takes two parameters, name and age, and initializes them as attributes of the object  
        self.name = name  
        self.age = age  
  
    def greet(self):  
        # This is a method of the Person class that prints a greeting message  
        print("Hello, myname is " + self.name)
```

Name and age are the two properties of the Person class. Additionally, it has a function called greet that prints a greeting.

Python Objects

- An Object is an instance of a Class.
- A class is like a blueprint while an instance is a copy of the class with actual values
- An object consists of:
- State: It is represented by the attributes of an object. It also reflects the properties of an object.
- Behavior: It is represented by the methods of an object. It also reflects the response of an object to other objects.
- Identity: It gives a unique name to an object and enables one object to interact with other objects.

Example:



Declaring Class Objects

- When an object of a class is created, the class is said to be instantiated.
- All the instances share the attributes and the behavior of the class. But the values of those attributes, i.e. the state are unique for each object.
- A single class may have any number of instances
- An object is a particular instance of a class with unique characteristics and functions.
- After a class has been established, you may make objects based on it. By using the class constructor, you may create an object of a class in Python.
- The object's attributes are initialised in the constructor, which is a special procedure with the name `__init__`.

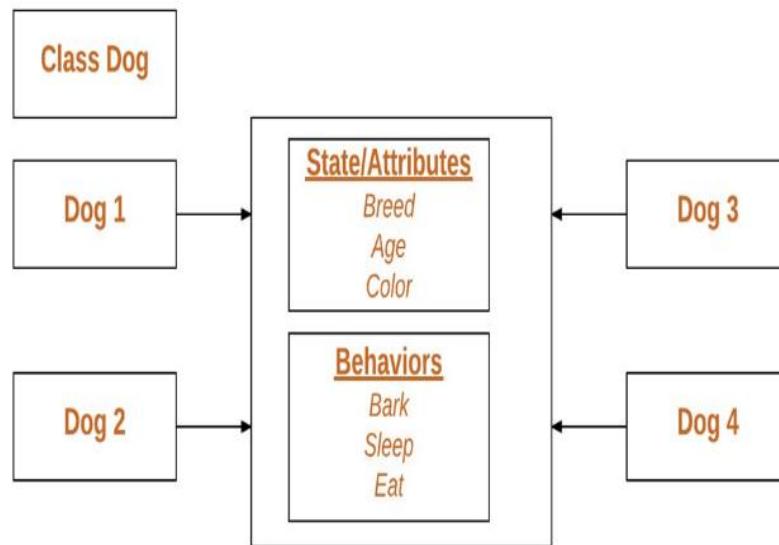


Fig.:Declaring Class Objects

Syntax:

```

# Declare an object of a class

object_name = Class_Name(arguments)

class Person:

    def __init__(self, name, age):
        self.name = name
        self.age = age

    def greet(self):
        print("Hello, my name is " + self.name)

# Create a new instance of the Person class and assign it to the variable person1

person1 = Person("Ayan", 25)

person1.greet()

```

Output:

"Hello, my name is Ayan"

Python Inheritance

- Inheritance is an important aspect of the object-oriented paradigm. Inheritance provides code reusability to the program because we can use an existing class to create a new class instead of creating it from scratch.
- In inheritance, the child class acquires the properties and can access all the data members and functions defined in the parent class. A child class can also provide its specific implementation to the functions of the parent class.
- In python, a derived class can inherit base class by just mentioning the base in the bracket after the derived class name.
- Inheritance allows us to create a new class from an existing class.
- The new class that is created is known as subclass (child or derived class) and the existing class from which the child class is derived is known as superclass (parent or base class).

Python Inheritance Syntax:

```
# define a superclass  
  
class super_class:  
  
    # attributes and method definition  
  
    # inheritance  
  
    class sub_class(super_class):  
  
        # attributes and method of super_class  
  
        # attributes and method of sub_class
```

Here, we are inheriting the sub_class class from the super_class class.

Example:

```
class Animal:  
  
    def speak(self):  
  
        print("Animal Speaking")  
  
#child class Dog inherits the base class Animal  
  
class Dog(Animal):  
  
    def bark(self):  
  
        print("dog barking")  
  
d = Dog()  
  
d.bark()  
  
d.speak()
```

Output:

```
dog barking  
Animal Speaking
```

Python Multi-Level inheritance

- Multi-Level inheritance is possible in python like other object-oriented languages.
- Multi-level inheritance is archived when a derived class inherits another derived class.
- There is no limit on the number of levels up to which, the multi-level inheritance is archived in python.

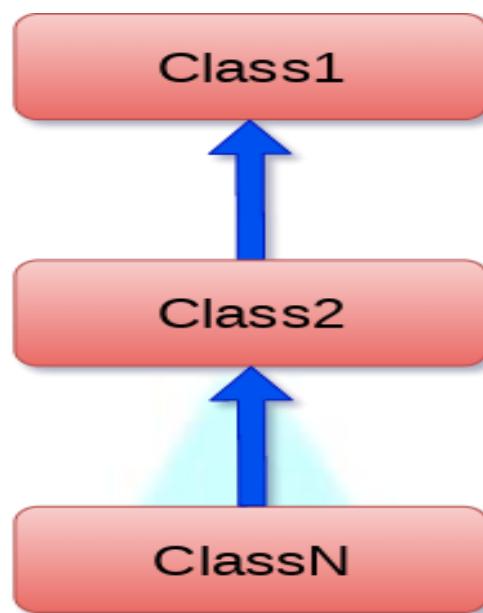


Fig.: syntax of multi-level inheritance

```
class class1:  
<class-suite>  
  
class class2:(class1):  
<class suite>  
  
class class3(class2):  
<class suite>
```

Example

```

class Animal:

    def speak(self):
        print("Animal Speaking")

#The child class Dog inherits the base class Animal

class Dog(Animal):

    def bark(self):
        print("dog barking")

#The child class Dogchild inherits another child class Dog

class DogChild(Dog):

    def eat(self):
        print("Eating bread...")

d = DogChild()

d.bark()
dog barking

d.speak()

d.eat()

```

Python Multiple inheritance

Python provides us the flexibility to inherit multiple base classes in the child class

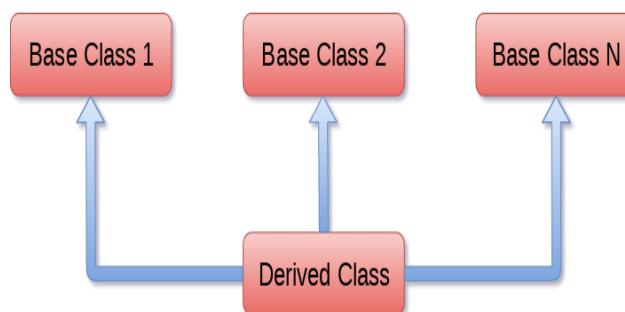


Fig.:Syntax of python multiple inheritance

The syntax to perform multiple inheritance is

```

class Base1:
    <class-suite>

class Base2: <class-suite>

.
.
.

class BaseN:
    <class-suite>

class Derived(Base1, Base2, ..... BaseN):
    <class-suite>

```

Example:

```

class Calculation1:

def Summation(self,a,b):

    return a+b;

class Calculation2:

def Multiplication(self,a,b):

    return a*b;

class Derived(Calculation1,Calculation2):

def Divide(self,a,b):

    return a/b;

d = Derived()

print(d.Summation(10,20))

print(d.Multiplication(10,20))

print(d.Divide(10,20))

```

Output:

30

200

0.5

Polymorphism in Python

Polymorphism is a very important concept in programming. It refers to the use of a single type entity (method, operator or object) to represent different types in different scenarios.

Function Polymorphism in Python

- There are some functions in Python which are compatible to run with multiple data types.
- One such function is the `len()` function. It can run with many data types in Python. Let's
- look at some example use cases of the function.

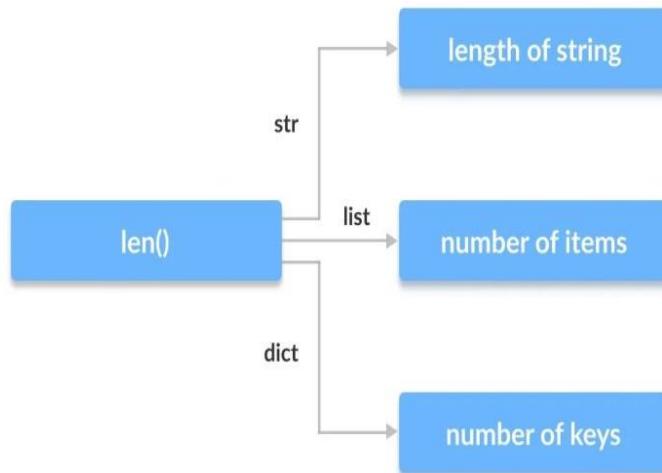
Example:

Polymorphic `len()` function:

```
print(len("GTTC"))
print(len(["PLC", "IoT", "Automation"]))
print(len({"Name": "Asha", "Address": "Hubli"}))
```

Output:

- Here, we can see that many data types such as string, list, tuple, set, and dictionary can work with the `len()` function.
- However, we can see that it returns specific information about specific data types.



Python Exceptions

- When a Python program meets an error, it stops the execution of the rest of the program. An error in Python might be either an error in the syntax of an expression or a Python exception.
- An exception in Python is an incident that happens while executing a program that causes the regular course of the program's commands to be disrupted. When a Python code comes across a condition it can't handle, it raises an exception. An object in Python that describes an error is called an exception.
- When a Python code throws an exception, it has two options: handle the exception immediately or stop and quit.

Python Logical Errors (Exceptions)

Errors that occur at runtime (after passing the syntax test) are called exceptions or logical errors.

For instance, they occur when we

- try to open a file(for reading) that does not exist (FileNotFoundException)
- try to divide a number by zero (ZeroDivisionError)
- try to import a module that does not exist (ImportError) and so on.

Whenever these types of runtime errors occur, Python creates an exception object.

If not handled properly, it prints a traceback to that error along with some details about why that error occurred.

Some of the common built-in exceptions in Python programming along with the error that cause them are

Exception	Cause of Error
AssertionError	Raised when an assert statement fails.
AttributeError	Raised when attribute assignment or reference fails.
EOFError	Raised when the input() function hits end-of-file condition.
FloatingPointError	Raised when a floating point operation fails.
GeneratorExit	Raise when a generator's close() method is called.
ImportError	Raised when the imported module is not found.
IndexError	Raised when the index of a sequence is out of range.

Some of the common built-in exceptions in Python programming along with the error that cause them are

Exception	Cause of Error
KeyError	Raised when a key is not found in a dictionary.
KeyboardInterrupt	Raised when the user hits the interrupt key (Ctrl+C or Delete).
MemoryError	Raised when an operation runs out of memory.
NameError	Raised when a variable is not found in local or global scope.
NotImplementedError	Raised by abstract methods.
OSError	Raised when system operation causes system related error.
OverflowError	Raised when the result of an arithmetic operation is too large to be represented.
ReferenceError	Raised when a weak reference proxy is used to access garbage collected referent.
RuntimeError	Raised when an error does not fall under any other category.
StopIteration	Raised by next() function to indicate that there is no further item to be returned by iterator.

Some of the common built-in exceptions in Python programming along with the error that cause them are

Exception	Cause of Error
SyntaxError	Raised by parser when syntax error is encountered.
IndentationError	Raised when there is incorrect indentation.
TabError	Raised when indentation consists of inconsistent tabs and spaces.
SystemError	Raised when interpreter detects internal error
SystemExit	Raised by sys.exit() function
TypeError	Raised when a function or operation is applied to an object of incorrect type.
UnicodeError	Raised when a Unicode-related encoding or decoding error occurs.
UnicodeEncodeError	Raised when a Unicode-related error occurs during encoding.
UnicodeDecodeError	Raised when a Unicode-related error occurs during decoding.
UnicodeTranslateError	Raised when a Unicode-related error occurs during translating.
ValueError	Raised when a function gets an argument of correct type but improper value.
ZeroDivisionError	Raised when the second operand of division or modulo operation is zero.
UnboundLocalError	Raised when a reference is made to a local variable in a function or method, but no value has been bound to that variable.

Python Error and Exception

- Errors represent conditions such as compilation error, syntax error, error in the logical part of the code, library incompatibility, infinite recursion, etc.
- Errors are usually beyond the control of the programmer and we should not try to handle errors.
- Exceptions can be caught and handled by the program.

Python Exception Handling

The exceptions abnormally terminate the execution of a program. In Python, we use the **try...except** block to handle exceptions.

The syntax of try...except block:

```
try:  
    # code that may cause exception  
  
    except:  
        # code to run when exception occurs
```

- Here, we have placed the code that might generate an exception inside the try block. Every try block is followed by an except block.
- When an exception occurs, it is caught by the except block. The except block cannot be used without the try block.

Example: Exception Handling Using try...except

```
try:  
    numerator = 10  
  
    denominator = 0  
  
    result = numerator/denominator  
  
    print(result)  
  
except:  
    print("Error: Denominator cannot be 0.")
```

Output:

Error: Denominator cannot be 0.

- In the example, we are trying to divide a number by 0. Here, this code generates an exception.
- To handle the exception, we have put the code,

```
result = numerator/denominator
```

- Inside the try block. Now when an exception occurs, the rest of the code inside the try block is skipped.
- The except block catches the exception and statements inside the except block are executed.
- If none of the statements in the try block generates an exception, the except block is skipped

Reference

- <https://www.w3schools.com/python/>
- <https://www.geeksforgeeks.org/python-programming-language/>
- <https://www.tutorialspoint.com/python/index.htm>
- <https://www.programiz.com/python-programming/online-compiler/>
- <https://www.codecademy.com/learn/learn-python>

File Handling in Python

File handling is a crucial aspect of programming that allows for the manipulation of files on a system. Python provides a variety of built-in functions and methods to create, read, update, and delete files. This ability is essential for many applications, including data storage, configuration management, and logging.

Opening a File

Python uses the `open()` function to open files. This function requires at least one argument: the name of the file. It can also take a second argument that specifies the mode in which the file should be opened.

```
file = open("example.txt", "r") # Open file for reading
```

The common modes are:

- '`r
- 'w
- 'a
- 'brb', 'wb')
- 'x`

Reading from a File

Once a file is opened, you can read its contents using various methods.

```
# Read the entire file
with open("example.txt", "r") as file:
    content = file.read()
    print(content)

# Read a single line
with open("example.txt", "r") as file:
    line = file.readline()
    print(line)

# Read all lines into a list
with open("example.txt", "r") as file:
    lines = file.readlines()
    print(lines)
```

Writing to a File

Writing to a file can be done using the `write()` or `writelines()` methods.

```
# Write a single string to a file
with open("example.txt", "w") as file:
    file.write("Hello, World!\n")

# Write multiple lines to a file
with open("example.txt", "w") as file:
    lines = ["First line\n", "Second line\n", "Third line\n"]
    file.writelines(lines)
```

Appending to a File

Appending data to a file adds content to the end without altering existing data.

```
with open("example.txt", "a") as file:  
    file.write("This is an appended line.\n")
```

Closing a File

It's important to close a file after completing operations to free up system resources. This is done using the `close()` method or by using a `with` statement, which ensures that the file is properly closed.

```
file = open("example.txt", "r")  
# Perform operations  
file.close()  
  
# Using with statement (recommended)  
with open("example.txt", "r") as file:  
    content = file.read()  
    print(content)
```

File Handling Exceptions

Errors can occur during file operations, such as attempting to open a non-existent file. Python provides exception handling to manage these errors gracefully.

```
try:  
    file = open("non_existent_file.txt", "r")  
except FileNotFoundError:  
    print("File not found.")  
except IOError:  
    print("An IOError occurred.")
```

Working with Binary Files

Binary files are used to store data in binary format. Reading and writing binary files is similar to text files but requires the '`b`' mode.

```
# Writing to a binary file  
with open("example.bin", "wb") as file:  
    file.write(b'\x00\x01\x02\x03')  
  
# Reading from a binary file  
with open("example.bin", "rb") as file:  
    data = file.read()  
    print(data)
```

Python3 and PyCharm IDE Installation

To start with Python development, you need to install Python3 and an Integrated Development Environment (IDE) like PyCharm.

Installing Python3

1. Windows:

- Download the Python installer from the [official website](#).
- Run the installer and ensure you check "Add Python to PATH".
- Follow the installation steps.

2. macOS:

- Download the macOS installer from the [official website](#).
- Open the downloaded package and follow the instructions.

3. Linux:

- Most distributions come with Python pre-installed.
- To install or update, use the package manager, e.g., sudo apt-get install python3.

Verifying the Installation

To verify the installation, open a terminal or command prompt and run:

```
python3 --version
```

Installing PyCharm IDE

1. Download PyCharm:

- Go to the [PyCharm website](#) and download the Community edition (free).

2. Installation:

- **Windows:** Run the downloaded .exe file and follow the installation wizard.
- **macOS:** Open the .dmg file and drag PyCharm to the Applications folder.
- **Linux:** Unpack the tar.gz file and run ./pycharm.sh from the bin subdirectory.

3. Launching PyCharm:

- Open PyCharm and configure it according to your preferences.

PIP and Python Libraries Installation

PIP is the package installer for Python, allowing you to install and manage additional libraries that are not part of the standard library.

Installing PIP

PIP usually comes pre-installed with Python3. To check if PIP is installed, run:

```
pip --version
```

If not installed, you can install it using:

```
curl https://bootstrap.pypa.io/get-pip.py -o get-pip.py
python3 get-pip.py
```

Installing Python Libraries

With PIP, you can install libraries using the following command:

```
pip install <library_name>
```

For example, to install NumPy and Pandas:

```
pip install numpy pandas
```

Listing Installed Libraries

To list all installed libraries, use:

```
pip list
```

Upgrading a Library

To upgrade a library to the latest version:

```
pip install --upgrade <library_name>
```

Uninstalling a Library

To uninstall a library:

```
pip uninstall <library_name>
```

Flask

Flask is an open-source microframework written in the Python programming language. Flask allows users to quickly and easily build web applications without worrying about low-level tasks such as thread management and protocol. Due to it being open-source, it's regularly updated with new tools and features by software engineers and developers.

Even though Flask is a microframework, it is still rich in features. It uses the Werkzeug WSI toolkit and the Jinja2 template engine for its backend development. It also provides a development server and debugger, and it improves functionality via extensions.

It was developed by Armin Ronacher in 2004 and it is still an incredibly popular framework. Today, it ranks in the most used frameworks among developers. This beginner-friendly code is ideal if you want to get started with fully functional website development.

Flask is based on WSGI(Web Server Gateway Interface) toolkit and Jinja2 template engine.

- **WSGI:** It is an acronym for web server gateway interface which is a standard for python web application development. It is considered as the specification for the universal interface between the web server and web application.
- **Jinja:** Jinja2 is a web template engine which combines a template with a certain data source to render the dynamic web pages.

Advantages of Flask

- **Open Source.** Flask has an active developer community that continuously contributes new tools and APIs to improve the framework.
- **Lightweight.** Flask is a microframework that has built-in modules to keep it lightweight and minimal.
- **Powerful.** Flask has almost everything you need to start a web application. This makes it ideal for large companies and high-end production engineers.
- **Speed.** Flask lets you build applications quickly and easily. This increases productivity in software development and mobile development.
- **Easy to Use.** Flask has an easy-to-understand interface that's perfect for beginners. Advanced users can add extensions to Flask if they think it is lacking features.

Flask Environment

Install virtualenv for development environment **virtualenv** is a virtual Python environment builder. It helps a user to create multiple Python environments side-by-side. Thereby, it can avoid compatibility issues between the different versions of the libraries.

The following command installs **virtualenv**

pip install virtualenv

This command needs administrator privileges. Add **sudo** before **pip** on Linux/Mac OS. If you are on Windows, log in as Administrator. On Ubuntu **virtualenv** may be installed using its package manager.

Sudo apt-get install virtualenv

Once installed, new virtual environment is created in a folder.

```
mkdir newproj  
cd newproj  
virtualenv venv
```

To activate corresponding environment, on **Linux/OS X**, use the following –

venv/bin/activate

On **Windows**, following can be used

venv\scripts\activate

We are now ready to install Flask in this environment.

pip install Flask

The above command can be run directly, without virtual environment for system-wide installation.

Flask – Application

```
# The file named as Flask.py.

from flask import Flask
app = Flask(__name__) # creating the Flask class object

@app.route('/')
def home():
    return "Internet of Things";

if __name__ == '__main__':
    app.run(debug=True)
```

The result in command line .



```
Flask ×
E:\projectnew11\venv\Scripts\python.exe E:/projectnew11/Flask.py
* Serving Flask app 'Flask' (lazy loading)
* Environment: production
  WARNING: This is a development server. Do not use it in a production deployment.
  Use a production WSGI server instead.
* Debug mode: on
* Running on http://127.0.0.1:5000 (Press CTRL+C to quit)
* Restarting with stat
* Debugger is active!
* Debugger PIN: 120-807-968
127.0.0.1 - - [31/Oct/2023 05:16:51] "GET / HTTP/1.1" 200 -
127.0.0.1 - - [31/Oct/2023 05:16:55] "GET /favicon.ico HTTP/1.1" 404 -
```

A web application, run on the browser at <http://localhost:5000>.



To build the python web application, we need to import the Flask module. An object of the Flask class is considered as the WSGI application. We need to pass the name of the current module, i.e. `__name__` as the argument into the Flask constructor.

The route () function of the Flask class defines the URL mapping of the associated function. The syntax is given below.

app.route(rule, options)

It accepts the following parameters.

- **rule:** It represents the URL binding with the function.
- **options:** It represents the list of parameters to be associated with the rule object

As we can see here, the / URL is bound to the main function which is responsible for returning the server response. It can return a string to be printed on the browser's window or we can use the HTML template to return the HTML file as a response from the server.

Finally, the run method of the Flask class is used to run the flask application on the local development server.

The syntax is given below.

app.run(host, port, debug, options)

SN	Option	Description
1	host	The default hostname is 127.0.0.1, i.e. localhost.
2	port	The port number to which the server is listening to. The default port number is 5000.
3	debug	The default is false. It provides debug information if it is set to true.
4	options	It contains the information to be forwarded to the server.

Debug mode

A Flask application is started by calling the run() method. However, while the application is under development, it should be restarted manually for each change in the code. To avoid this inconvenience, enable debug support. The server will then reload itself if the code changes. It will also provide a useful debugger to track the errors if any, in the application.

The Debug mode is enabled by setting the debug property of the application object to True before running or passing the debug parameter to the run() method.

```
app.debug = True
```

```
app.run()
```

app.run(debug = True)

Flask Routing

Modern web frameworks use the routing technique to help a user remember application URLs. It is useful to access the desired page directly without having to navigate from the home page.

The route() decorator in Flask is used to bind URL to a function.

For example:

```
@app.route('/hello')  
  
def hello_world():  
  
    return 'hello world'
```

Here, URL '/hello' rule is bound to the hello_world() function. As a result, if a user visits <http://localhost:5000/hello> URL, the output of the hello_world() function will be rendered in the browser.

Flask App routing

App routing is used to map the specific URL with the associated function that is intended to perform some task. It is used to access some particular page in the web application.

In our first application, the URL ('/') is associated with the home function that returns a particular string displayed on the web page.

In other words, we can say that if we visit the particular URL mapped to some particular function, the output of that function is rendered on the browser's screen.

In flask, the URL ("/") is associated with the root URL. So if our site's domain was www.example.org and we want to add routing to "www.example.org/hello", we would use "/hello". To bind a function to an URL path we use the app.route decorator.

In the below example, we have implemented the above routing in the flask.

main.py

```
from flask import Flask

app = Flask(__name__)

# Pass the required route to the decorator.

@app.route("/hello")
def hello():
    return "Hello, GTTC Hubli"

@app.route("/")
def index():
    return "Homepage of GTTC"

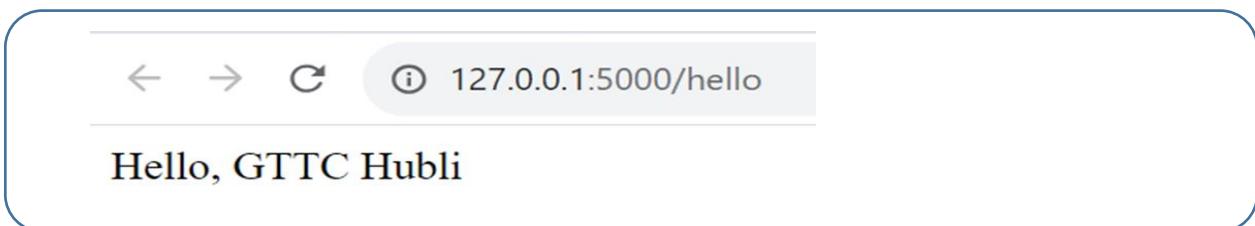
if __name__ == "__main__":
    app.run(debug=True)
```

RESULT

The hello function is now mapped with the “/hello” path and we get the output of the function rendered on the browser.



Open the browser and visit `127.0.0.1:5000/hello`, you will see the following output.



The `add_url_rule()` function

The `add_url_rule()` function of an application object is also available to bind a URL with a function as in the above example, `route()` is used.

The syntax to use this function is

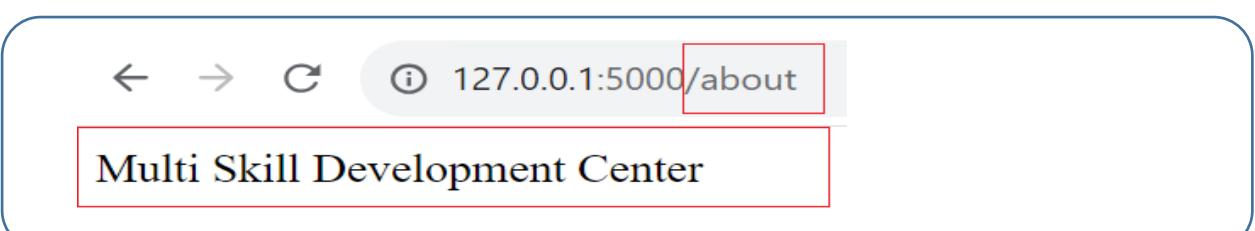
`add_url_rule(<url rule>, <endpoint>, <view function>)`

This function is mainly used in the case if the view function is not given and we need to connect a view function to an endpoint externally by using this function.

Example:

```
from flask import Flask
app = Flask(__name__)
def about():
    return "Multi Skill Development Center"
app.add_url_rule("/about", "about", about)
if __name__ == "__main__":
    app.run(debug=True)
```

OUTPUT:



Flask Variable Rules

It is possible to build a URL dynamically, by adding variable parts to the rule parameter. This variable part is marked as <variable-name>. It is passed as a keyword argument to the function with which the rule is associated.

Flask facilitates us to add the variable part to the URL by using the section. We can reuse the variable by adding that as a parameter into the view function.

The converter can also be used in the URL to map the specified variable to the particular data type. For example, we can provide the integers or float like age or salary respectively.

The following converters are used to convert the default string type to the associated data type.

1. **string:** default
2. **int:** used to convert the string to the integer
3. **float:** used to convert the string to the float.
4. **path:** It can accept the slashes given in the URL.
5. **uuid:** It accepts UUID strings.

Example 1:

```
from flask import Flask

app = Flask(__name__)

@app.route('/course/<name>')

def course(name):

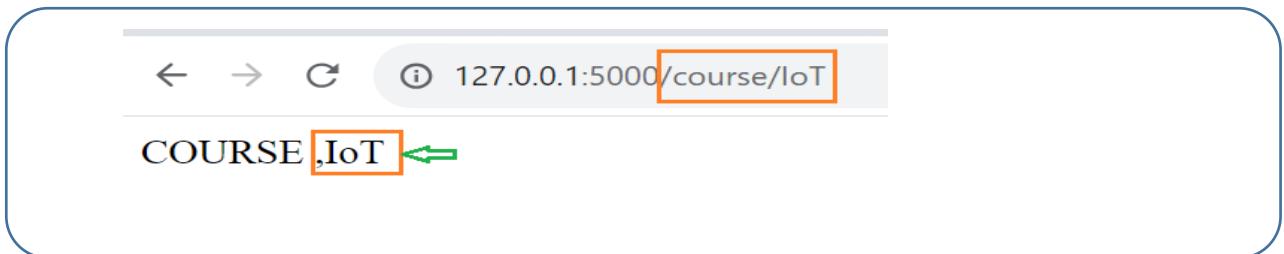
    # Course Name

    return 'COURSE ,'+ name

if __name__ == "__main__":

    app.run(debug=True)
```

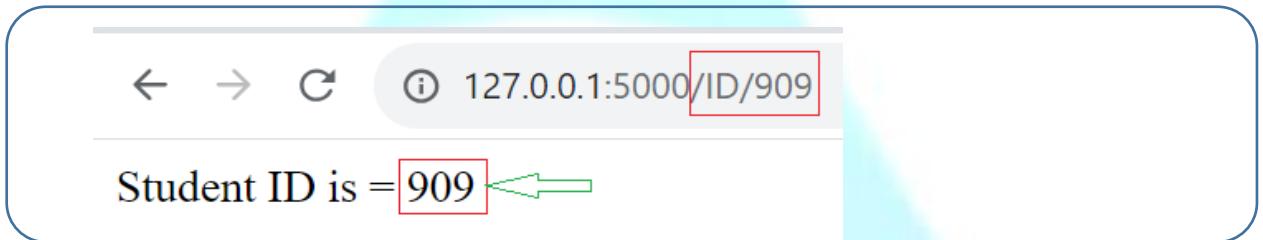
OUTPUT:



Example 2

```
from flask import Flask  
  
app = Flask(__name__)  
  
@app.route('/ID/<int:std_id>')  
  
def id(std_id):  
  
    return "Student ID is = %d"%std_id;  
  
if __name__ == "__main__":  
  
    app.run(debug=True)
```

OUTPUT:



Flask – URL Building

The url_for() function is very useful for dynamically building a URL for a specific function. The function accepts the name of a function as first argument, and one or more keyword arguments, each corresponding to the variable part of URL.

This function is useful in the sense that we can avoid hard-coding the URLs into the templates by dynamically building them using this function.

```
from flask import *
app = Flask(__name__)

@app.route('/')
def dept():
    return 'MSDC'

@app.route('/iot')
def iot():
    return 'Internet of Things'

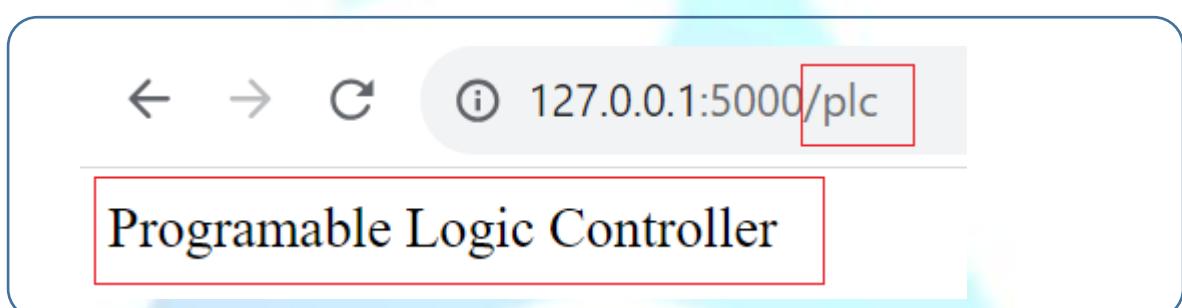
@app.route('/plc')
def plc():
    return 'Programable Logic Controller'

@app.route('/cad')
def cad():
    return 'Computer Adied Design'

@app.route('/user/<name>')
def user(name):
    if name == 'iot':
        return redirect(url_for('iot'))
    if name == 'plc':
        return redirect(url_for('plc'))
    if name == 'cad':
        return redirect(url_for('cad'))

if __name__ == '__main__':
    app.run(debug=True)
```

OUTPUT:



Flask HTTP methods

HTTP is the hypertext transfer protocol which is considered as the foundation of the data transfer in the world wide web. All web frameworks including flask need to provide several HTTP methods for data communication.

We can specify which HTTP method to be used to handle the requests in the route() function of the Flask class. By default, the requests are handled by the GET() method.

The methods are given in the following table

SN	Method	Description
1	GET	It is the most common method which can be used to send data in the unencrypted form to the server.
2	HEAD	It is similar to the GET but used without the response body.
3	POST	It is used to send the form data to the server. The server does not cache the data transmitted using the post method.
4	PUT	It is used to replace all the current representation of the target resource with the uploaded content.
5	DELETE	It is used to delete all the current representation of the target resource specified in the URL.

Flask HTTP GET Method

The Flask framework is a micro web framework for Python, and it provides the functionality to handle various HTTP methods, including GET. The GET method is used to request data from a specified resource.

Handling GET Requests

GET requests are used to request data from a specified resource. The data sent to the server with GET requests can be accessed using **request.args**.

Example: GetForm.html

```

<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Using GET Method</title>
    <style> table, td, th{border: 1px solid black;} </style>
</head>
<body>
    <h1>GET Method form</h1>
    <form action="/register" method="GET">
        <input type="text" id="fname" name="fname" placeholder="First Name"><br><br>
        <input type="date" id="dob" name="dob" placeholder="Date of Birth"><br><br>
        <input type="number" id="age" name="age" placeholder="Age" min="18" max="35"><br><br>
        <input type="submit" value="Submit"><br><br>
    </form>
    <table style="width:70%">
        <tr>
            <th>Name</th>
            <th>date of Birth</th>
            <th>Age</th>
        </tr>
        {% for row in users %}
        <tr>
            <td>{{row[0]}}</td>
            <td>{{row[1]}}</td>
            <td>{{row[2]}}</td>
        </tr>
        {% endfor %}
    </table> </body> </html>

```

Python code

```

from flask import Flask, request, render_template
app = Flask(__name__)
# Initialize an empty list to store user data
users = []
@app.route('/')
def data():
    return render_template('GetForm.html')
@app.route('/register', methods=['GET'])
def signin():
    # Access form data using request.form instead of request.args for a POST request
    fname = request.args.get('fname')
    dob = request.args.get('dob')
    age = request.args.get('age')
    user = []
    user.append(fname)
    user.append(dob)
    user.append(age)
    # Store user data in the users list
    users.append(user)
    print(fname, dob, age)
    # Print user data (for testing purposes)
    # Pass the users list to the template
    return render_template('GetForm.html', users=users)
if __name__ == '__main__':
    app.run(debug=True)

```

Output:

GET Method form

Name	date of Birth	Age
Pushpa L M	1988-01-05	35
Kiran K	2000-09-08	23

Flask HTTP POST Method

Flask is a lightweight web framework for Python that makes it easy to develop web applications. One of the critical features of any web framework is handling HTTP methods. The POST method in Flask, which is used to submit data to be processed to a specified resource.

Handling POST Requests

POST requests are used to send data to the server to create/update a resource. The data sent to the server with POST requests can be accessed using **request.form** or **request.json**.

Example: PostForm.html

```

<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Using GET Method</title>
    <style> table, td, th{border: 1px solid black;} </style>
</head>
<body>
<h1>POST Method form</h1>
<form action="/register" method="POST">
    <input type="text" id="fname" name="fname" placeholder="First Name"><br><br>
    <input type="date" id="dob" name="dob" placeholder="Date of Birth"><br><br>
    <input type="number" id="age" name="age" placeholder="Age" min="18" max="35"><br><br>
    <input type="radio" name="gender" value="Male">Male
        <input type="radio" name="gender" value="Female">Female<br><br>
    <input type="submit" value="Submit"><br><br>
</form>
<table style="width:100%">
    <tr>
        <th>Name</th>
        <th>date of Birth</th>
        <th>Age</th>
        <th>Gender</th>
    </tr>
    {% for row in users %}>
        <tr>
            <td>{{row[0]}}</td>
            <td>{{row[1]}}</td>
            <td>{{row[2]}}</td>
            <td>{{row[3]}}</td>
        </tr>
    {% endfor %}
</table>
</body>
</html>

```

Python Code:

```

from flask import Flask, request, render_template
app = Flask(__name__)
# Initialize an empty list to store user data
users = []
@app.route('/')
def data():
    return render_template('PostForm.html')
@app.route('/register', methods=['POST'])
def signin():
    # Access form data using request.form instead of request.args for a POST request
    if request.method=="POST":
        fname = request.form['fname']
        dob = request.form['dob']
        age = request.form['age']
        gender=request.form['gender']
        user = [ ]
        user.append(fname)
        user.append(dob)
        user.append(age)
        user.append(gender)
    # Store user data in the users list
    users.append(user)
    # Print user data (for testing purposes)
    print(fname, dob, age, gender)
    # Pass the users list to the template
    return render_template('PostForm.html', users=users)
if __name__ == '__main__':
    app.run(debug=True)

```

Browser Output:

POST Method form

 Male FeMale

Name	date of Birth	Age	Gender
Akash V	1990-02-16	33	Male
Deepa Patil	2001-01-05	22	Female
Kavita Desai	1999-02-28	24	Female

Flask Templates

Flask facilitates us to return the response in the form of HTML templates. Rendering external HTML files

Flask facilitates us to render the external HTML file instead of hardcoding the HTML in the view function. Here, we can take advantage of the jinja2 template engine on which the flask is based.

Flask provides us the `render_template()` function which can be used to render the external HTML file to be returned as the response from the view function.

message.html

```
#To render an HTML file from the view function, let's first create an HTML file named as message.html.
```

```
<html>  
<head>  
<title>Message</title>  
</head>  
<body>  
<h1>hi, welcome to the GTTC Hubbli </h1>  
</body>  
</html>
```

script.py

```
from flask import *  
  
app = Flask(__name__)  
  
@app.route('/')  
  
def message():  
  
    return render_template('message.html')  
  
if __name__ == '__main__':  
  
    app.run(debug = True)
```

Run the flask script **script.py** and visit <https://localhost:5000> on the browser the result is.



← → ⌛ ⓘ 127.0.0.1:5000

hi, welcome to the GTTC Hubbli

Delimiters

Jinga 2 template engine provides some delimiters which can be used in the HTML to make it capable of dynamic data representation. The template system provides some HTML syntax which are placeholders for variables and expressions that are replaced by their actual values when the template is rendered.

The jinga2 template engine provides the following delimiters to escape from the HTML.

{% ... %} for statements

{{ ... }} for expressions to print to the template output

{# ... #} for the comments that are not included in the template output

... ## for line statements

Static Files

The static files such as CSS or JavaScript file enhance the display of an HTML web page. A web server is configured to serve such files from the static folder in the package or the next to the module. The static files are available at the path /static of the application.

Example

In the following example, the flask script returns the HTML file, i.e., **message.html** which is styled using the stylesheet **style.css**. The flask template system finds the static CSS file under **static/css** directory. Hence the style.css is saved at the specified path.

script.py

```
from flask import *
app = Flask(__name__)
@app.route('/')
def message():
    return render_template('message.html')
if __name__ == '__main__':
    app.run(debug = True)
```

message.html

```
<html>
<head>
    <title>Message</title>
    <link rel="stylesheet" href= "/static/style.css" >
</head>

<body>
    <h1> Embedded Full Stack IoT </h1>
</body>
</html>
```

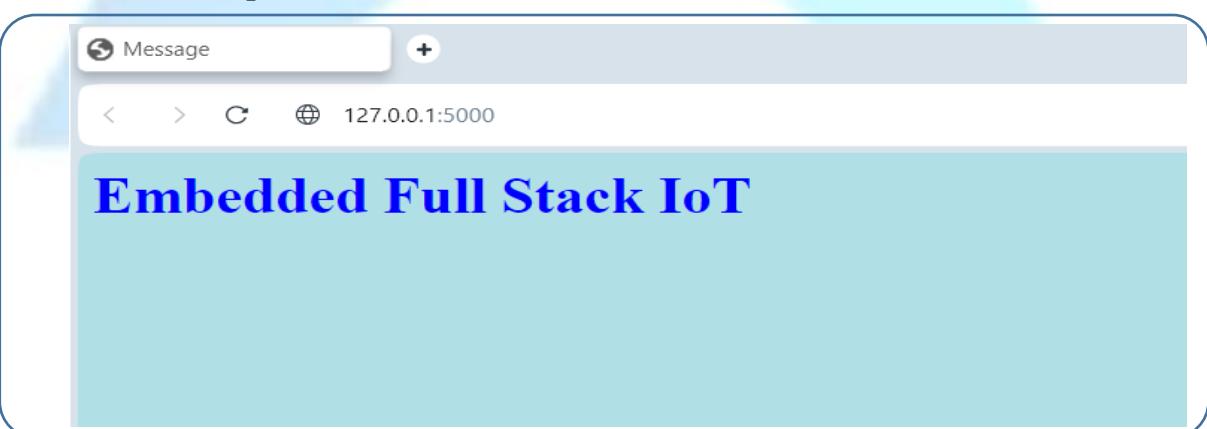
style.css

```
body {
    background-color: powderblue;
}

h1 {
    color: blue;
}

p {
    color: red;
}
```

Browser Output



Flask Request Object

The data from a client's web page is sent to the server as a global request object. In order to process the request data, it should be imported from the Flask module.

Important attributes of request object are listed below –

- Form – It is a dictionary object containing key and value pairs of form parameters and their values.
- args – parsed contents of query string which is part of URL after question mark (?).
- Cookies – dictionary object holding Cookie names and values.
- files – data pertaining to uploaded file.
- method – current request method.

Flask – Sending Form Data to Template

In the http method can be specified in URL rule. The Form data received by the triggered function can collect it in the form of a dictionary object and forward it to a template to render it on a corresponding web page.

In the following example, ‘/’ URL renders a web page (student.html) which has a form. The data filled in it is posted to the ‘/result’ URL which triggers the result() function.

The results() function collects form data present in request.form in a dictionary object and sends it for rendering to result.html.

The template dynamically renders an HTML table of form data.

Main.py

```

from flask import Flask, render_template, request
app = Flask(__name__)

@app.route('/')
def student():
    return render_template('student.html')

@app.route('/result', methods = ['POST', 'GET'])
def result():
    if request.method == 'POST':
        result = request.form
        return render_template("result.html", result = result)

if __name__ == '__main__':
    app.run(debug = True)

```

student.html

```

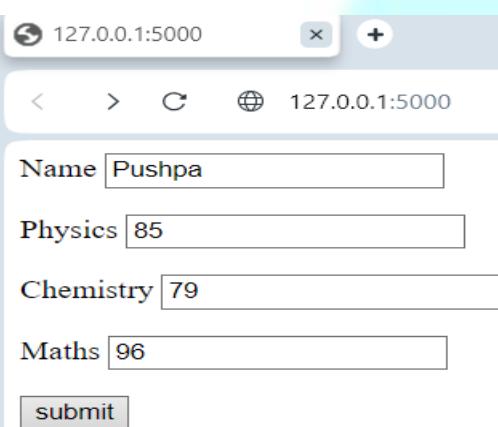
<html>
  <body>
    <form action = "http://localhost:5000/result" method = "POST">
      <p>Name <input type = "text" name = "Name" /></p>
      <p>Physics <input type = "text" name = "Physics" /></p>
      <p>Chemistry <input type = "text" name = "chemistry" /></p>
      <p>Maths <input type = "text" name = "Mathematics" /></p>
      <p><input type = "submit" value = "submit" /></p>
    </form>
  </body>
</html>

```

result.html

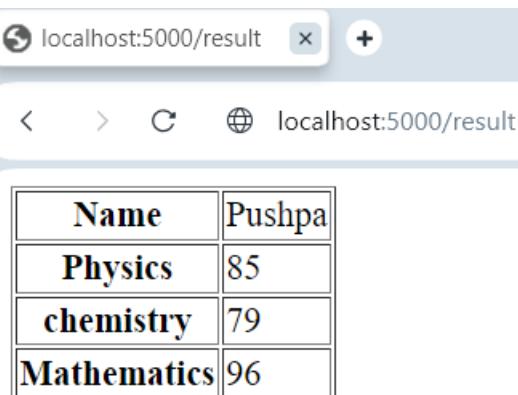
```
<!doctype html>
<html>
<body>
<table border = 1>
{ % for key, value in result.items() %}
<tr>
<th> {{ key }} </th>
<td> {{ value }} </td>
</tr>
{ % endfor %
</table>
</body>
</html>
```

Browser Output:



A screenshot of a web browser window. The address bar shows the URL `127.0.0.1:5000`. The page contains a form with four input fields and a submit button. The fields are labeled "Name", "Physics", "Chemistry", and "Maths". The "Name" field contains "Pushpa", "Physics" contains "85", "Chemistry" contains "79", and "Maths" contains "96". Below the fields is a "submit" button.

When the Submit button is clicked, form data is rendered on result.html in the form of HTML table



A screenshot of a web browser window. The address bar shows the URL `localhost:5000/result`. The page displays a table with four rows. The first row has a single cell containing "Name Pushpa". The subsequent three rows have two cells each, with the first cell being the subject name and the second cell being the score. The subjects and scores are "Physics 85", "chemistry 79", and "Mathematics 96".

Name	Pushpa
Physics	85
chemistry	79
Mathematics	96

Flask SQLite

Flask can make use of the SQLite3 module of the python to create the database web applications.

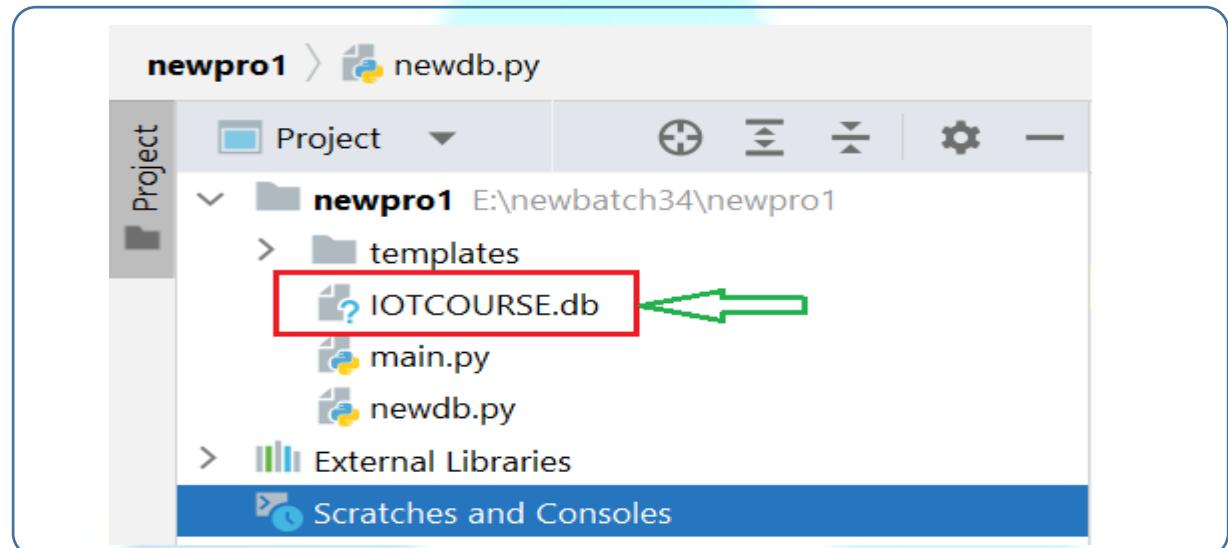
In Flask using SQLite we can create a CRUD (create - read - update - delete) application.

CRUD Application in flask

For this purpose, the database should be created **IOTCOURSE.db**

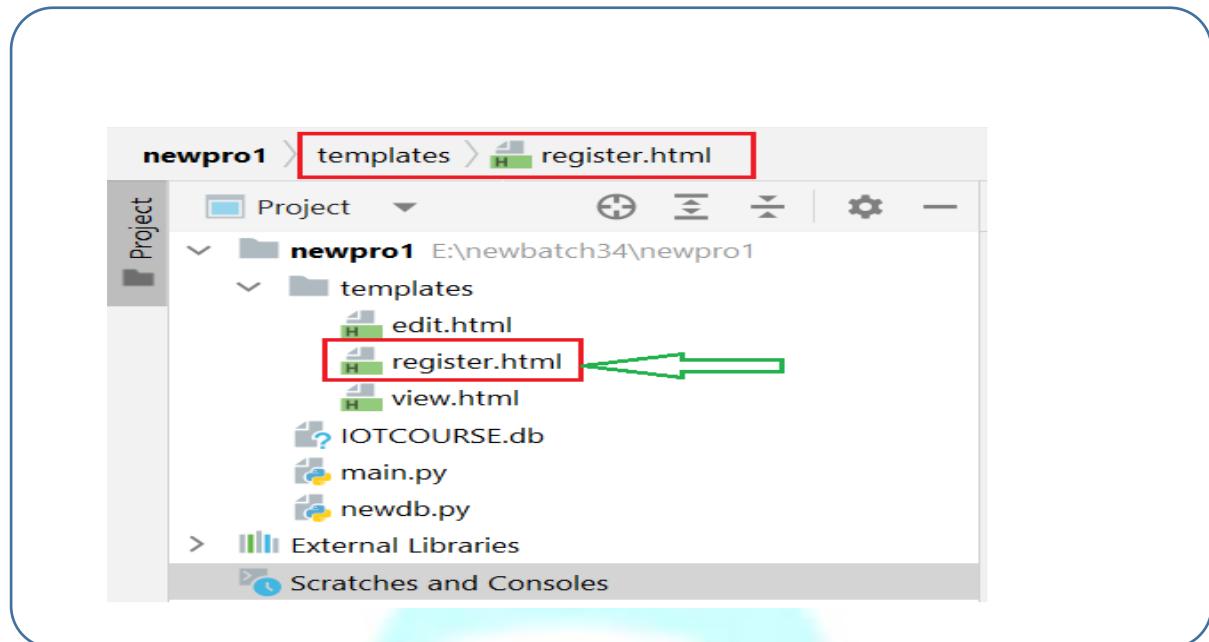
newdb.py

Create a data base using python file



```
import sqlite3
conn = sqlite3.connect("IOTCOURSE.db")
conn.execute("CREATE TABLE Students(id INTEGER PRIMARY KEY AUTOINCREMENT, fname TEXT, mob TEXT, dob TEXT )")
```

Next create the view function: first() which is associated with the URL (/). It renders a template register.html.



register.html.

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Register Form</title>
    <meta name="viewport" content="width=device-width, initial-scale=1">
</head>
<body>
    <div style="text-align:center">
        <h1>Form Design</h1>
        <h4><a href="/view" >VIEW</a> </h4>
        <h4>{{msg}}</h4>
        <form action="/register" method="POST">
            First Name: <input type="text" id="fname" name="fname" required><br><br>
            Mobile No : <input type="text" id="mob" name="mob"><br><br>
            Date of Birth: <input type="date" id="dob" name="dob"><br><br>
            <input type="submit" value="Register">
        </form>
    </div>
</body>
</html>
```

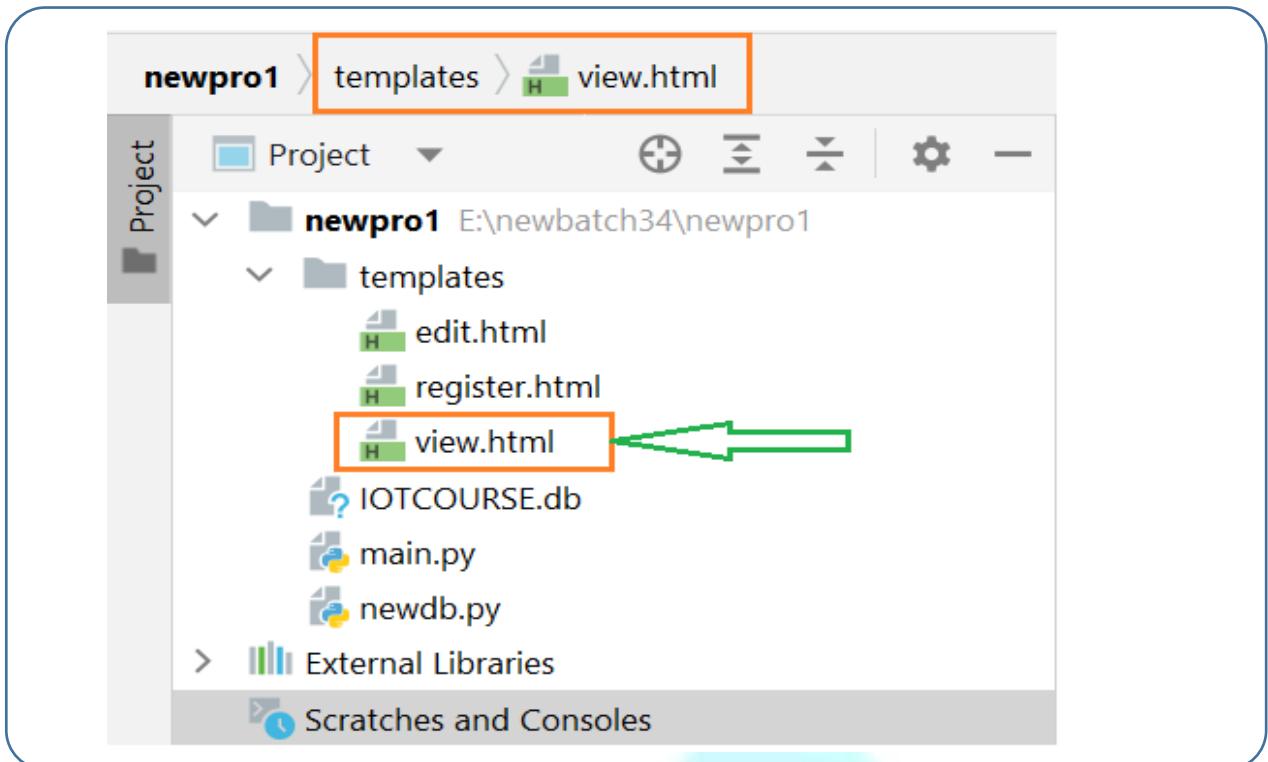
Python code

```
@app.route('/')
def first():
    return render_template("register.html")
```

All the details entered by the students is posted to the URL **/register** which is associated with the function **register()**. The function **register()** is given below which contains the code for extracting the data entered by the students and save that data into the table Students.

```
@app.route('/register', methods=["POST"])
def register():
    if request.method == "POST":
        fname = request.form['fname']
        mob = request.form['mob']
        dob = request.form['dob']
        print(fname, mob, dob)
        conn = sqlite3.connect("IOTCOURSE.db")
        conn.execute("INSERT INTO students(fname, mob, dob)VALUES(?, ?, ?)", (fname, mob, dob))
        conn.commit()
        msg = "Registered successfully!"
        return render_template("register.html", msg=msg)
```

The entered details can be stored in database IOTCOURSE.db . The details can be viewed in renders a template **view.html** with the function **view()**



Python code

```
@app.route('/view')

def view():

    conn = sqlite3.connect("IOTCOURSE.db")

    cur = conn.execute("SELECT * FROM students")

    rows = cur.fetchall()

    return render_template("view.html", rows=rows)
```

view.html

```

<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>View Details</title>
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <style> table, td, th{border:1px solid black; } </style>
</head>
<body>
<div style="text-align:center">
    <h1>View Details</h1>
    <h4><a href="/">Home</a> &ampnbsp <a href="/view">View Details</a>
</h4>
<table style="width:100%">
    <thead>
        <tr>
            <th>Id</th>
            <th>Name</th>
            <th>Mobile</th>
            <th>DOB</th>
            <th>Action</th>
        </tr>
    </thead>
    <tbody>
        { % for row in rows % }
        <tr>
            <td>{{ row[0] }}</td>
            <td>{{ row[1] }}</td>
            <td>{{ row[2] }}</td>
            <td>{{ row[3] }}</td>
            <td><a href="/del/{{ row[0] }}">Delete</a>, <a href="/edit/{{ row[0] }}">Edit</a> </td>
        </tr>
    { % endfor %}
    </tbody>
</table>
</div> </body> </html>

```

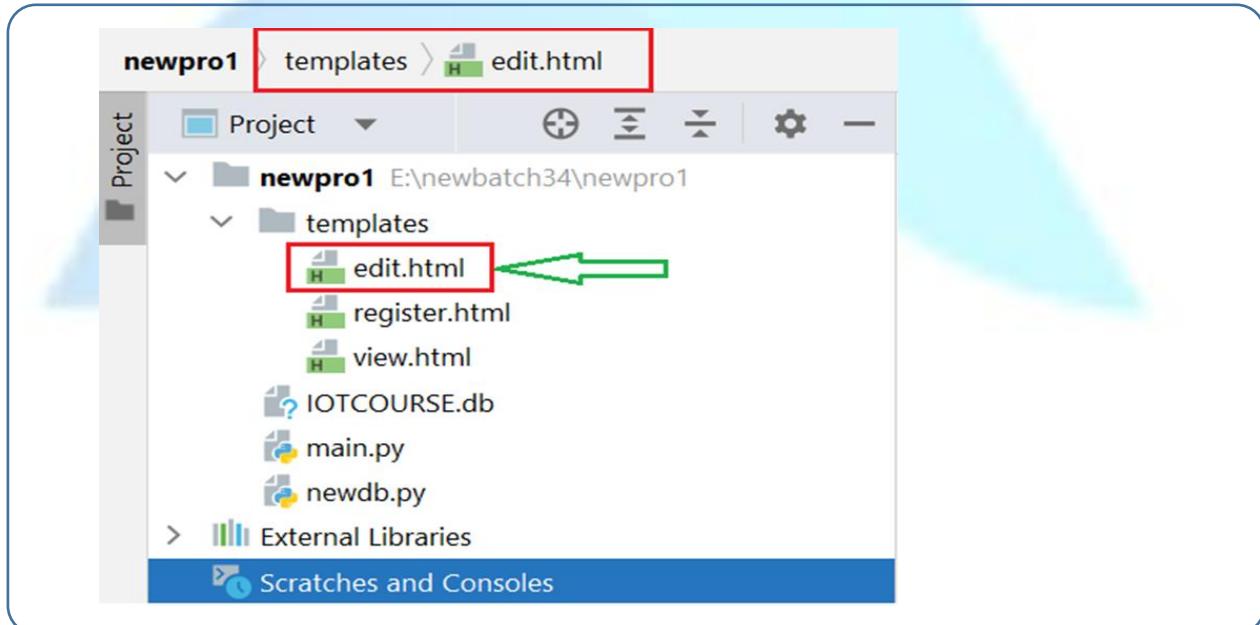
The ID entered by the student is posted to the URL **/del** which contains the python code to establish the connection to the database and then delete all the records for the specified ID. The URL **/del** is associated with the function **delete()** which is given below.

```
@app.route('/del/<a>')
def delete(a):
    con=sqlite3.connect("IOTCOURSE.db")
    con.execute("DELETE FROM students where id=?",(a,))
    con.commit()
    return redirect(url_for('view'))
```

After delete() function It returns a response object and redirects the URL /view

The entered details can be stored in database IOTCOURSE.db . The details can be edited with the URL **/edit** which is associated with the the function **edit()** .

The details can be updated in renders a template **edit.html** with the function **update()** with the URL **/update**



```

@app.route('/edit/<a>')

def edit(a):

    con=sqlite3.connect("IOTCOURSE.db")

    cur=con.execute("SELECT *FROM students where id=?",(a,))

    row=cur.fetchone()

    return render_template("edit.html", row=row)

```

edit.html

```

<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>About</title>
    <meta name="viewport" content="width=device-width, initial-scale=1">
</head>
<body>
<div style="text-align:center">
    <h1>Edit Details</h1>
    <h4><a href="/view" >View Details</a> </h4>
    <h4>{ { msg } }</h4>
    <form action="/update/{ { row[0] } }" method="POST">
        First Name: <input type="text" id="fname" name="fname"
        value="{ { row[1] } }" required><br><br>
        Mobile No : <input type="text" id="mob" name="mob"
        value="{ { row[2] } }"><br><br>
        Date of Birth: <input type="date" id="dob" name="dob"
        value="{ { row[3] } }"><br><br>
        <input type="submit" value="update">
    </form>
</div>
</body>
</html>

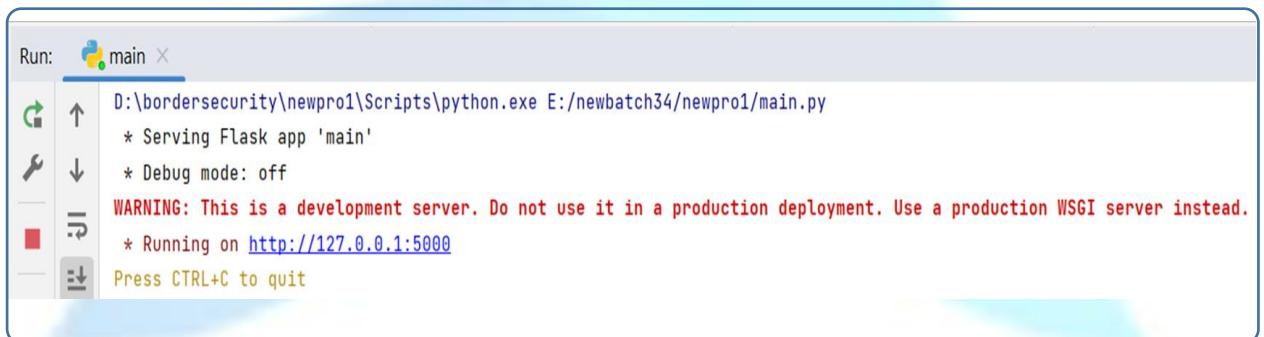
```

The ID entered by the student is posted to the URL **/edit** which contains the python code to establish the connection to the database and then edit all the records for the specified ID. The URL **/update** is associated with the function **update()** which is given below.

```
@app.route('/update/<a>',methods=['POST'])
def update(a):
    if request.method=='POST':
        fname=request.form['fname']
        mob=request.form['mob']
        dob=request.form['dob']
        con=sqlite3.connect('IOTCOURSE.db')
        con.execute('UPDATE students SET fname=?, mob=?,dob=? WHERE id=?',(fname,mob,dob,a))
        con.commit()
        return redirect(url_for('view'))
```

After update() function It returns a response object and redirects the URL /view

Run the flask script main.py and visit <https://localhost:5000> on the browser.



To visit <http://127.0.0.1:5000> on browser

Form Design

[VIEW](#)

First Name:

Mobile No :

Date of Birth: dd - mm - yyyy 

[Register](#)

To visit [view](#) on home page

View Details

[Home](#) [View Details](#)

Id	Name	Mobile	DOB	Action
10	GTTC Hubli	1234554321	2023-11-06	Delete , Edit
11	Asha K	9876556789	2023-11-05	Delete , Edit

To visit edit in view page

Edit Details

[View Details](#)

First Name:

Mobile No :

Date of Birth: 

After row update the result is

View Details

[Home](#) [View Details](#)

Id	Name	Mobile	DOB	Action
10	MSDC	1234554321	2023-11-06	<u>Delete</u> , <u>Edit</u>
11	Asha K	9876556789	2023-11-05	<u>Delete</u> , <u>Edit</u>

Reference

<https://flask.palletsprojects.com/en/3.0.x/>

<https://www.geeksforgeeks.org/flask-tutorial/>

<https://pythonbasics.org/what-is-flask-python/>

<https://www.javatpoint.com/flask-tutorial>

SQL

Introduction to Databases

A database is an organized collection of structured information, or data, typically stored electronically in a computer system. A database is usually controlled by a database management system (DBMS).

Types of Databases

In general, there are two common types of databases:

1. Non-Relational
2. Relational

Non-Relational Database

In a non-relational database, data is stored in **key-value pairs**.

For example:

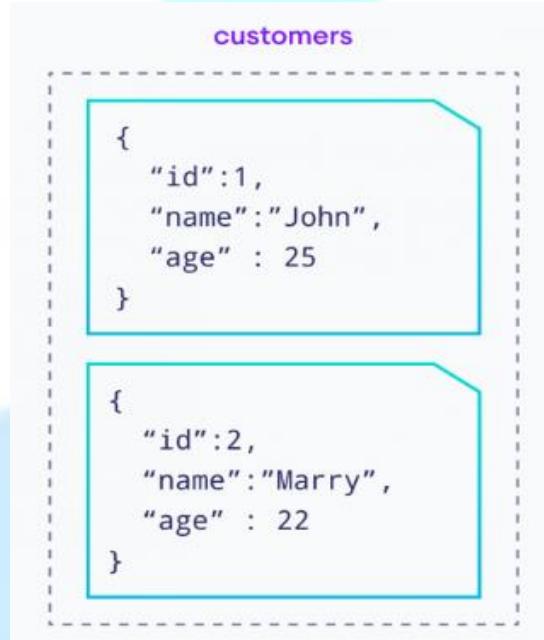


Fig.:Non-Relational Database

Here, customers' data are stored in key-value pairs.

- Commonly used non-relational database management systems (Non-RDBMS) are MongoDB, Amazon DynamoDB, Redis, etc
- Non-relational databases are often used when large quantities of complex and diverse data need to be organized. For example, a large store might have a database in which each customer has their own document containing all of their information, from name and address to order history and credit card information. Despite their differing formats, each of these pieces of information can be stored in the same document.
- Non-relational databases often perform faster because a query does not have to view several tables in order to deliver an answer, as relational datasets often do. Non-relational databases are therefore ideal for storing data that may be changed frequently or for applications that handle many different kinds of data. They can support rapidly developing applications requiring a dynamic database able to change quickly and to accommodate large amounts of complex, unstructured data.

Relational Database

In a relational database, data is stored in tabular format.

For example,

Table: customers

customer_id	first_name	last_name	phone	country
1	John	Doe	817-646-8833	USA
2	Robert	Luna	412-862-0502	USA
3	David	Robinson	208-340-7906	UK
4	John	Reinhardt	307-242-6285	UK
5	Betty	Taylor	806-749-2958	UAE

Here, customers is a table inside the database.

- The first row is the attributes of the table. Each row after that contains the data of a customer.
- Commonly used relational database management systems (RDBMS) are MySQL, PostgreSQL, MSSQL, Oracle etc.
- Relational databases use Structured Query Language (SQL). In relational database design, the database usually contains tables consisting of columns and rows. When new data is added, new records are inserted into existing tables or new tables are added. Relationships can then be made between two or more tables. Relational databases work best when the data they contain does not change very often, and when accuracy is crucial. Relational databases are, for instance, often found in financial applications.

Introduction to RDBMS

RDBMS stands for Relational Database Management Systems. It is a program that allows us to create, delete, and update a relational database. A Relational Database is a database system that stores and retrieves data in a tabular format organized in the form of rows and columns. It is a smaller subset of DBMS which was designed by E.F Codd in the 1970s. The major DBMSs like SQL, My-SQL, and ORACLE are all based on the principles of relational DBMS.

Relational DBMS owes its foundation to the fact that the values of each table are related to others. It has the capability to handle larger magnitudes of data and simulate queries easily. Relational Database Management Systems maintains data integrity by simulating the following features:

- **Entity Integrity:** No two records of the database table can be completely duplicate.
- **Referential Integrity:** Only the rows of those tables can be deleted which are not used by other tables. Otherwise, it may lead to data inconsistency.

- **User-defined Integrity:** Rules defined by the users based on confidentiality and access.
- **Domain integrity:** The columns of the database tables are enclosed within some structured limits, based on default values, type of data or ranges.

RDBMS History

The history of Relational Database Management Systems (RDBMS) starts in the 1970s with E F. Codd's work at IBM. Codd introduced the concept of relational databases in 1970 that use of SQL (Structured Query Language) for querying data. This model revolutionized data management by emphasizing data integrity and reducing redundancy. In the 1980s, commercial RDBMS products such as Oracle, IBM DB2, and Microsoft SQL Server emerged, making relational databases the industry standard for data management. Over the decades, RDBMS technology has continued to evolve, incorporating advancements in scalability, performance, and support for complex queries, cementing its role as a cornerstone of modern database management.

Features of RDBMS

- Data must be stored in tabular form in DB file, that is, it should be organized in the form of rows and columns.
- Each row of table is called record/tuple. Collection of such records is known as the cardinality of the table
- Each column of the table is called an attribute/field. Collection of such columns is called the arity of the table.
- No two records of the DB table can be same. Data duplicity is therefore avoided by using a candidate key. Candidate Key is a minimum set of attributes required to identify each record uniquely.
- Tables are related to each other with the help for foreign keys.
- Database tables also allow NULL values, that is if the values of any of the element of the table are not filled or are missing, it becomes a NULL value, which is not equivalent to zero. (NOTE: Primary key cannot have a NULL value).

Working of Relational Database Model

The relational database, created by IBM's E.F. Codd in the 1970s, enables any table to be associated to another table by means of a common attribute. Codd suggested a change to a data model where data is stored, accessed, and related in tables without restructuring the tables that hold them in place of hierarchical structures.

Each spreadsheet in the relational database model is a table that contains data, which is shown as rows (records or tuples) and columns (attributes).

A data type is specified by an attribute (column), and the value of that particular data type is contained in each record (or row). A primary key is an attribute found in all tables in a relational database that serves as a unique row identifier.

Example: The following table STUDENT consists of three columns Roll Number, Name, Section and four records of students 1, 2, 3 and 4 respectively. The records can't be completely same, the Roll Number acts as a candidate key which separates records.

Roll number	Name	Section
1	Ishita	A
2	Yash	B
3	Ishita	A
4	Mallika	C

Uses of RDBMS

RDBMS is used in Customer Relationship Management.

- It is used in Online Retail Platforms.
- It is used in Hospital Management Systems.
- It is used in Business Intelligence.
- It is used in Data Warehousing

Advantages of RDBMS

- **Easy to Manage:** Each table can be independently manipulated without affecting others.
- **Security:** It is more secure consisting of multiple levels of security. Access of data shared can be limited.
- **Flexible:** Updating of data can be done at a single point without making amendments at multiple files. Databases can easily be extended to incorporate more records, thus providing greater scalability. Also, facilitates easy application of SQL queries.
- **Users:** RDBMS supports client-side architecture storing multiple users together.
- Facilitates storage and retrieval of large amount of data.
- **Easy Data Handling:**
 - Data fetching is faster because of relational architecture.
 - Data redundancy or duplicity is avoided due to keys, indexes, and normalization principles.
 - Data consistency is ensured because RDBMS is based on ACID properties for data transactions(Atomicity Consistency Isolation Durability).
- **Fault Tolerance:** Replication of databases provides simultaneous access and helps the system recover in case of disasters, such as power failures or sudden shutdowns.

Disadvantages of RDBMS

- **High Cost and Extensive Hardware and Software Support:** Huge costs and setups are required to make these systems functional.
- **Scalability:** In case of addition of more data, servers along with additional power, and memory are required.
- **Complexity:** Voluminous data creates complexity in understanding of relations and may lower down the performance.
- **Structured Limits:** The fields or columns of a relational database system is enclosed within various limits, which may lead to loss of data.

Structured Query Language (SQL)

SQL (Structured Query Language) is a powerful and standard query language for relational database systems. This language is used by all the relational database systems. SQL allows users to create databases, add data, modify, maintain data and fetch selected data. It is governed by standards maintained by ISO (International Standards Organization). We use SQL to perform CRUD (Create, Read, Update, Delete) operations on databases along with other various operations like:

- create databases
- create tables in a database
- read data from a table
- insert data in a table
- update data in a table
- delete data from a table
- delete database tables
- delete databases
- and many more database operations

Benefits of SQL:

1. SQL is non procedural, that means you do not need to write lengthy programs to get data, there are commands, expressions and operators to use to get data from tables.
2. SQL is portable, databases using SQL can be moved from one device to another without any problems.
3. Easy to learn as SQL is in form of ENGLISH statements.

SQLite

SQLite is embedded relational database management system. It is self-contained, serverless, zero configuration and transactional SQL database engine.

SQLite is free to use for any purpose commercial or private. In other words, “SQLite is an open source, zero-configuration, self-contained, stand alone, transaction relational database engine designed to be embedded into an application”.

SQLite is different from other SQL databases because unlike most other SQL databases, SQLite does not have a separate server process. It reads and writes directly to ordinary disk files. A complete SQL database with multiple tables, indices, triggers, and views, is contained in a single disk file.

Features of SQLite

- **SQLite is totally free:** SQLite is open-source. So, no license is required to work with it.
- **SQLite is serverless:** SQLite doesn't require a different server process or system to operate.
- **SQLite is very flexible:** It facilitates you to work on multiple databases on the same session on the same time.
- **Configuration Not Required:** SQLite doesn't require configuration. No setup or administration required.
- **SQLite is a cross-platform DBMS:** You don't need a large range of different platforms like Windows, Mac OS, Linux, and Unix. It can also be used on a lot of embedded operating systems like Symbian, and Windows CE.
- **Storing data is easy:** SQLite provides an efficient way to store data.
- **Variable length of columns:** The length of the columns is variable and is not fixed. It facilitates you to allocate only the space a field needs. For example, if you have a varchar(200) column, and you put a 10 characters' length value on it, then SQLite will allocate only 20 characters' space for that value not the whole 200 space.
- **Provide large number of API's:** SQLite provides API for a large range of programming languages. **For example:** .Net languages (Visual Basic, C#), PHP, Java, Objective C, Python and a lot of other programming language.
- **SQLite** is written in **ANSI-C** and provides simple and easy-to-use API.
- **SQLite** is available on **UNIX** (Linux, Mac OS-X, Android, iOS) and **Windows** (Win32, WinCE, WinRT).

SQLite Advantages

SQLite is a very popular database which has been successfully used with on disk file format for desktop applications like version control systems, financial analysis tools, media cataloging and editing suites, CAD packages, record keeping programs etc. There are a lot of advantages to use SQLite as an application file format:

1. Lightweight

SQLite is a very light weighted database so, it is easy to use it as an embedded software with devices like televisions, Mobile phones, cameras, home electronic devices, etc

2. Better Performance

Reading and writing operations are very fast for SQLite database. It is almost 35% faster than File system. It only loads the data which is needed, rather than reading the entire file and hold it in memory. If you edit small parts, it only overwrite the parts of the file which was changed.

3. No Installation Needed

SQLite is very easy to learn. You don't need to install and configure it. Just download SQLite libraries in your computer and it is ready for creating the database.

4. Reliable

It updates your content continuously so, little or no work is lost in a case of power failure or crash. SQLite is less bugs prone rather than custom written file I/O codes. SQLite queries are smaller than equivalent procedural codes so, chances of bugs are minimal.

5. Portable

SQLite is portable across all 32-bit and 64-bit operating systems and big- and little-endian architectures. Multiple processes can be attached with same application file and can read and write without interfering each other. It can be used with all programming languages without any compatibility issue.

Differences between SQL and SQLite

SQL	SQLite
SQL is Structured Query Language used to query Relational Database System. It is written in C language.	SQLite is an Relational Database Management System which is written in ANSI-C.
SQL is standard which specifies how relational schema is created, data is inserted or updated in relations, transactions are started and stopped, etc.	SQLite is file-based. It is different from other SQL databases because unlike most other SQL databases, SQLite does not have separate server process.
Main components of SQL are Data Definition Language(DDL), Data Manipulation Language(DML), Data Control Language(DCL).	SQLite supports many features of SQL and has high performance but does not support stored procedures.
SQL is Structured Query Language which is used with databases like MySQL, Oracle, Microsoft SQL Server, IBM DB2, etc.	SQLite is portable database resource. It could get an extension in whatever programming language used to access that database.
A conventional SQL database needs to be running as service like Oracle DB to connect to and provide lot of functionalities.	SQLite database system does not provide such functionalities.
SQL is query language which is used by other SQL databases. It is not database itself.	SQLite is relational database management system itself which uses SQL.

SQLite Commands

SQLite commands are similar to SQL commands. There are three types of SQLite commands:

- **DDL:** Data Definition Language
- **DML:** Data Manipulation Language
- **DQL:** Data Query Language

Data Definition Language

There are three commands in this group:

- **CREATE:** This command is used to create a table, a view of a table or other object in the database.
- **ALTER:** It is used to modify an existing database object like a table.
- **DROP:** The DROP command is used to delete an entire table, a view of a table or other object in the database.

Data Manipulation language

There are three commands in data manipulation language group:

- **INSERT:** This command is used to create a record.
- **UPDATE:** It is used to modify the records.
- **DELETE:** It is used to delete records.

Data Query Language

- **SELECT:** This command is used to retrieve certain records from one or more table.

SQLite dot Command

The SQLite dot commands are not terminated by a semicolon (;

Creating a Table in SQLite3

Step 1: Start SQLite3

```
import sqlite3
```

Step 2: Create a New Database

```
sqlite3 DatabaseName.db
```

Step 3: Create a New Table

To create a table in SQLite3, you use the `CREATE TABLE` statement.

SQLite Create Database

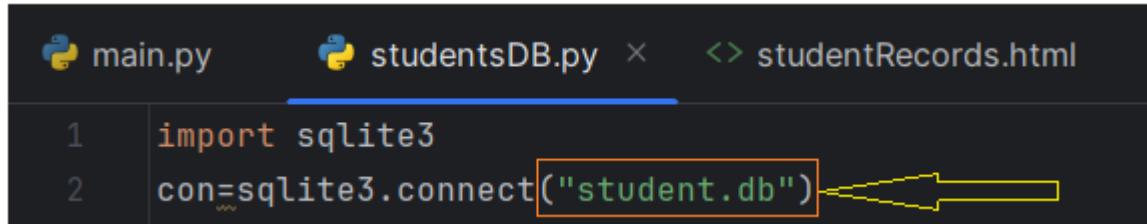
In SQLite, the `sqlite3` command is used to create a new database.

Syntax:

sqlite3 DatabaseName.db

Your database name should be unique within the RDBMS. The `sqlite3` command is used to create database. But, if the database does not exist then a new database file with the given name will be created automatically.

Example



```
main.py          studentsDB.py  x  studentRecords.html
1 import sqlite3
2 con=sqlite3.connect("student.db")
```

CREATE Table

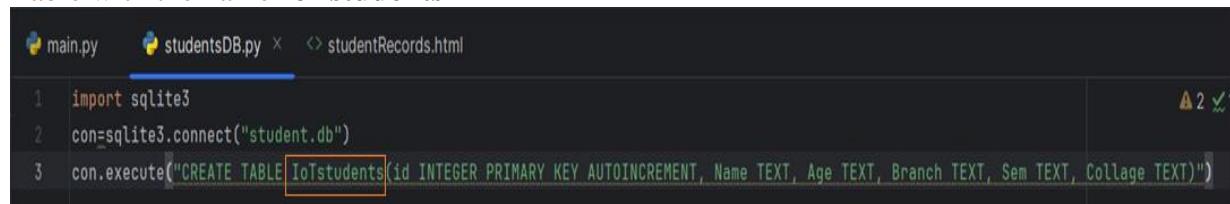
To create a table in SQLite3, you use the `CREATE TABLE` statement. The general syntax is:

```
CREATE TABLE table_name (
    column1 datatype,
    column2 datatype,
    column3 datatype,
    ....
    columnN datatype,
    PRIMARY KEY(one or more columns)
);
```

For example:

```
import sqlite3
con=sqlite3.connect("student.db")
con.execute("CREATE TABLE IoTstudents(id INTEGER PRIMARY KEY AUTOINCREMENT, Name TEXT, Age TEXT, Branch TEXT, Sem TEXT, Collage TEXT)")
```

Table with the name IoTstudents



```

main.py          studentsDB.py x  studentRecords.html

1 import sqlite3
2 con=sqlite3.connect("student.db")
3 con.execute("CREATE TABLE IoTstudents(id INTEGER PRIMARY KEY AUTOINCREMENT, Name TEXT, Age TEXT, Branch TEXT, Sem TEXT, Collage TEXT)")
```

Output of the table IoTstudents

id		Name	Age	Branch	Sem	Collage
	Filter	Filter	Filter	Filter	Filter	Filter
1	1	Asha	19	E&C	4th	KLECET Belgum
2	2	Kiran	22	ME	7th	SDMCET Dharwad
3	3	Suman	20	CS	2nd	Jain Engg. Hubli
4	4	Akash	21	E&C	5th	Jain Engg. Hubli
5	5	Gagan	19	ME	7th	KLECET Belgum
6	6	Deepa	21	E&C	4th	SDMCET Dharwad

SQLite Insert

To insert data into a table, you use the INSERT statement. SQLite provides various forms of the INSERT statements that allow you to insert a single row, multiple rows, and default values into a table. In addition, you can insert a row into a table using data provided by a SELECT statement.

SQLite INSERT – inserting a single row into a table

To insert a single row into a table, you use the following form of the INSERT statement:

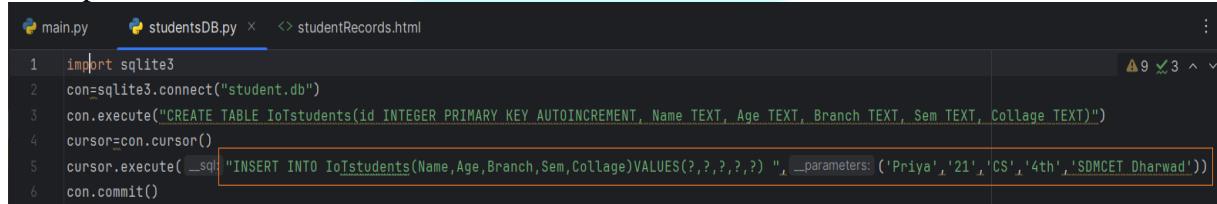
```
INSERT INTO table (column1,column2 ,..) VALUES( value1,value2 ,...);
```

First, specify the name of the table to which you want to insert data after the INSERT INTO keywords.

Second, add a comma-separated list of columns after the table name. The column list is optional. However, it is a good practice to include the column list after the table name.

Third, add a comma-separated list of values after the VALUES keyword. If you omit the column list, you have to specify values for all columns in the value list. The number of values in the value list must be the same as the number of columns in the column list.

Example for INSERT statement:

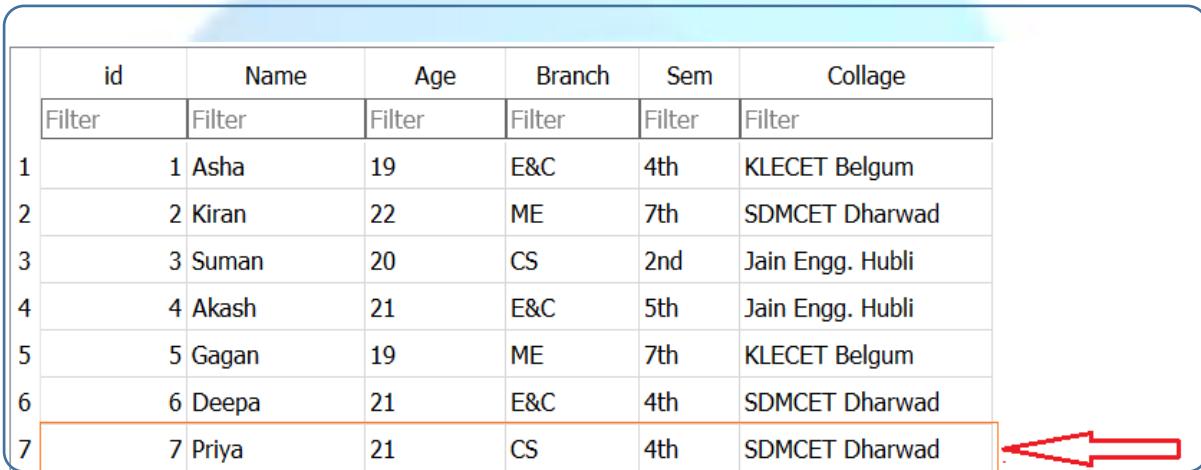


```

main.py      studentsDB.py  studentRecords.html
1 import sqlite3
2 con=sqlite3.connect("student.db")
3 con.execute("CREATE TABLE IoTstudents(id INTEGER PRIMARY KEY AUTOINCREMENT, Name TEXT, Age TEXT, Branch TEXT, Sem TEXT, Collage TEXT)")
4 cursor=con.cursor()
5 cursor.execute(_sql_ "INSERT INTO IoTstudents(Name,Age,Branch,Sem,Collage)VALUES(?,?,?,?,?)", _parameters: ('Priya','21','CS','4th','SDMCET Dharwad'))
6 con.commit()

```

The value is inserted in the table **IoTstudents**



id		Name	Age	Branch	Sem	Collage
Filter						
1	1	Asha	19	E&C	4th	KLECET Belgum
2	2	Kiran	22	ME	7th	SDMCET Dharwad
3	3	Suman	20	CS	2nd	Jain Engg. Hubli
4	4	Akash	21	E&C	5th	Jain Engg. Hubli
5	5	Gagan	19	ME	7th	KLECET Belgum
6	6	Deepa	21	E&C	4th	SDMCET Dharwad
7	7	Priya	21	CS	4th	SDMCET Dharwad

SQLite Select

SQLite SELECT statement is used to fetch the data from a SQLite database table which returns data in the form of a result table. These result tables are also called result sets.

The syntax of SQLite SELECT statement.

```
SELECT column1, column2, columnN FROM table_name;
```

Here, column1, column2 ... are the fields of a table, whose values you want to fetch.
 If you want to fetch all the fields available in the field, then you can use the following syntax
 SELECT * FROM table_name;

Example for SELECT statement:

```
cur=con.execute("SELECT Name, Branch FROM IoTstudents ")
print(cur.fetchall())
con.commit()
```

The Name and Branch is selected FROM IoTstudents

id	Name	Age	Branch	Sem	Collage
Filter	Filter	Filter	Filter	Filter	Filter
1	Asha	19	E&C	4th	KLECET Belgum
2	Kiran	22	ME	7th	SDMCET Dharwad
3	Neha	20	CS	2nd	Jain Engg. Hubli
4	Akash	21	E&C	5th	Jain Engg. Hubli
5	Gagan	19	ME	7th	KLECET Belgum
6	Deepa	21	E&C	4th	SDMCET Dharwad
7	Priya	21	CS	4th	SDMCET Dharwad
8	Priya	21	CS	4th	SDMCET Dharwad

SELECT Name, Branch FROM IoTstudents		
	Name	Branch
1	Asha	E&C
2	Kiran	ME
3	Neha	CS
4	Akash	E&C
5	Gagan	ME
6	Deepa	E&C
7	Priya	CS
8	Priya	CS

SQLite Delete

The SQLite DELETE statement is used to remove rows from a table. The SQLite DELETE statement allows you to delete one row, multiple rows, and all rows in a table.

The syntax of the SQLite DELETE statement is as follows:

DELETE FROM table WHERE search_condition;

In this syntax:

- First, specify the name of the table which you want to remove rows after the DELETE FROM keywords.
- Second, add a search condition in the WHERE clause to identify the rows to remove. The WHERE clause is an optional part of the DELETE statement. If you omit the WHERE clause, the DELETE statement will delete all rows in the table.

Example for DELETE statement

```
con.execute("DELETE FROM IoTstudents WHERE id=4")
con.commit()
```

The id=4 is deleted FROM IoTstudents

	id	Name	Age	Branch	Sem	Collage
	Filter	Filter	Filter	Filter	Filter	Filter
1	1	Asha	19	E&C	4th	KLECET Belgum
2	2	Kiran	22	ME	7th	SDMCET Dharwad
3	3	Neha	20	CS	2nd	Jain Engg. Hubli
4	4	Akash	21	E&C	5th	Jain Engg. Hubli
5	5	Gagan	19	ME	7th	KLECET Belgum
6	6	Deepa	21	E&C	4th	SDMCET Dharwad
7	7	Priya	21	CS	4th	SDMCET Dharwad
8	8	Priya	21	CS	4th	SDMCET Dharwad

DELETE FROM IoTstudents WHERE id=4;						
	id	Name	Age	Branch	Sem	Collage
	Filter	Filter	Filter	Filter	Filter	Filter
1	1	Asha	19	E&C	4th	KLECET Belgum
2	2	Kiran	22	ME	7th	SDMCET Dharwad
3	3	Neha	20	CS	2nd	Jain Engg. Hubli
4	5	Gagan	19	ME	7th	KLECET Belgum
5	6	Deepa	21	E&C	4th	SDMCET Dharwad
6	7	Priya	21	CS	4th	SDMCET Dharwad
7	8	Priya	21	CS	4th	SDMCET Dharwad

SQLite Drop Table

SQLite DROP TABLE statement is used to remove a table definition and all associated data, indexes, triggers, constraints, and permission specifications for that table. To remove a table in a database, use SQLite DROP TABLE statement.

DROP TABLE [IF EXISTS] [schema_name.]table_name;

- SQLite allows you to drop only one table at a time. To remove multiple tables, you need to issue multiple DROP TABLE statements.
- If you remove a non-existing table, SQLite issues an error. If you use IF EXISTS option, then SQLite removes the table only if the table exists, otherwise, it just ignores the statement and does nothing.
- If you want to remove a table in a specific database, you use the [schema_name.] explicitly.
- In case the table has dependent objects such as triggers and indexes, the DROP TABLE statement also removes all the dependent objects.

SQL Truncate Command

The SQL TRUNCATE command is used to delete all rows from a table, efficiently removing all data while retaining the table structure for future use. It is a Data Definition Language (DDL) command that operates differently from the DELETE statement, which is a Data Manipulation Language (DML) command. Understanding the TRUNCATE command is crucial for database management, especially when dealing with large datasets that need to be cleared quickly.

Key Features of SQL TRUNCATE

1. Complete Data Removal:

- The TRUNCATE command removes all rows from a table, leaving the table empty but intact.
- Unlike DELETE, TRUNCATE does not remove individual rows based on a condition; it clears the entire table.

2. Performance:

- TRUNCATE is generally faster than DELETE because it deallocates the data pages used by the table directly, bypassing the need to log individual row deletions.
- It is particularly efficient for large tables where deleting rows one by one would be time-consuming.

3. Transactional Behavior:

- The TRUNCATE command is usually non-transactional, meaning it cannot be rolled back once executed in most database systems. This makes it crucial to use with caution.
- However, in some database systems, TRUNCATE can be wrapped in a transaction, allowing it to be rolled back if needed.

4. Impact on Table Structure and Constraints:

- TRUNCATE resets any auto-increment counters for columns defined with an auto-increment attribute.
- It retains the table structure, including column definitions, constraints, indexes, and triggers.

5. Permission Requirements:

- To execute TRUNCATE, a user must have the necessary privileges, typically requiring the ALTER permission on the table.
- This ensures that only authorized users can perform such a significant operation.

Syntax of TRUNCATE Command

The syntax for the TRUNCATE command is straightforward:

```
TRUNCATE TABLE table_name;
```

- `table_name`: The name of the table from which all rows will be removed.

Example Usage

Consider a table named `employees`:

```
CREATE TABLE employees (
    id INT AUTO_INCREMENT,
    name VARCHAR(100),
    position VARCHAR(50),
    salary DECIMAL(10, 2),
    PRIMARY KEY (id)
);
```

To remove all data from the `employees` table, you would use:

```
TRUNCATE TABLE employees;
```

After executing this command, the `employees` table will be empty, but its structure, including columns, primary key, and any other constraints, will remain unchanged.

Comparison with DELETE Command

While both TRUNCATE and DELETE can be used to remove data from a table, they have distinct differences:

DELETE:

- Removes rows based on a condition (e.g., DELETE FROM employees WHERE position = 'Intern';).
 - Logs individual row deletions, which can be rolled back if used within a transaction.
- Slower for large tables due to logging overhead.

TRUNCATE:

- Removes all rows unconditionally.
- Logs page deallocations, making it faster.
- Typically non-transactional and cannot be rolled back in most systems.



EMBEDDED C PROGRAMMING

Introduction

Embedded C is a programming language that is used in the development of Embedded Systems. Embedded Systems are specialized systems designed to perform very specific functions or tasks. Embedded System is the combination of hardware and software and the software is generally known as firmware which is embedded into the system hardware. Embedded C is used to program a wide range of microcontrollers and microprocessors. Embedded C requires less number of resources to execute in comparison with high-level languages such as assembly programming language.

Features of embedded c programming language:

1. Simple and Efficient:

The basic syntax style of implementing C language is very simple and easy to learn. This makes the language easily comprehensible and enables a programmer to redesign or create a new application. Optimized code for speed and memory usage.

2. Hardware Interaction:

- Direct access to hardware registers.
- Bit manipulation to control individual bits of a port.
- Interfacing with peripherals like timers, ADCs, and communication protocols (SPI, I2C, UART).

3. Efficiency:

- Optimized code for speed and memory usage.
- Minimal runtime overhead.

4. Determinism:

- Predictable execution times, critical for real-time systems.
- Use of fixed-point arithmetic instead of floating-point for performance reasons.

5. Resource Constraints:

- Management of limited memory (RAM and ROM).
- Handling of processor limitations in terms of speed and power consumption.

6. Interrupt Handling:

- Use of interrupt service routines (ISRs) to handle asynchronous events.
- Prioritization and masking of interrupts.

7. Portability and Scalability:

- Code can be ported to different microcontrollers with minimal changes.
- Use of standardized C libraries and header files.

8. Compiler-Specific Extensions:

- Pragmas and intrinsic functions specific to embedded compilers.
- Inline assembly for critical performance sections.

9. Concurrency:

- Handling of multiple tasks and states without traditional multitasking support.
- Use of state machines, polling, and cooperative multitasking.

10. Real-Time Constraints:

- Meeting hard and soft real-time deadlines.
- Use of real-time operating systems (RTOS) when necessary.

11. Debugging and Testing:

- Use of in-circuit debuggers (ICDs) and simulators.
- Techniques like hardware-in-the-loop (HIL) for testing.

12. Memory Management:

- Static and dynamic allocation techniques, though dynamic allocation is often minimized or avoided.
- Use of memory-mapped IO.

13. Minimal Runtime Libraries:

- Custom lightweight libraries instead of full standard libraries.
- Often a subset of standard C libraries tailored for embedded systems.

Embedded C Applications:

Embedded C is used in multiple industries with many uses. In addition, embedded C programming has some real time applications.

Here are some notable areas where Embedded is commonly used:

1. Home Automation Systems:

- Smart Lighting: Embedded systems control lighting based on user input, schedules, or
- Sensor data (e.g., motion detectors).
- Smart Thermostats: Embedded controllers manage home heating and cooling systems
- To optimize energy use.

2. Automotive Systems:

- Engine Control Units (ECUs): Manage engine performance, fuel injection, and emission control systems.
- Anti-lock Braking System (ABS): Monitors wheel speed sensors and controls braking
 - i. pressure to prevent wheel lockup.

3. Consumer Electronics:

- Microwave Ovens: Control cooking time, power levels, and user interface.
- Washing Machines: Manage wash cycles, water levels, and spin speeds based on user
 - i. settings and sensor input.
- Televisions: Handle remote control input, display settings, and smart features like
 - i. internet connectivity.

4. Medical Devices:

- Pacemakers: Monitor heart rhythms and deliver electrical impulses to regulate heartbeats.

Embedded C Syntax

- Embedded C syntax is largely based on standard C, but it includes some specific features and practices tailored for embedded systems programming.
- Direct Register Access: Utilizes pointers to access hardware registers directly, optimizing performance and allowing precise control over peripherals.
- Bit-Level Manipulation: Employs bitwise operators (like &, |, ^, <<, >>) to manipulate individual bits, crucial for configuring hardware settings and status flags.
- Interrupt Handling: Defines Interrupt Service Routines (ISRs) using specific syntax (ISR() macros), ensuring timely response to hardware events without compromising real-time constraints.
- Memory Management: Manages memory resources efficiently, often minimizing dynamic memory allocation due to limited RAM availability typical in embedded systems.

Embedded C Keywords or Reserved Words:

Keywords are predefined, reserved words used in embedded c programming that have special meanings to the compiler. We cannot use a keyword as a variable name, function name, or any other identifier. They are used to define the syntax and structure of the embedded C language.

at	alien	Bit
code	compact	Far
idata	interrupt	Pdata
priority	reentrant	Sfr
small	_task_	Xdata

Embedded C Identifiers:

C identifiers represent the name in the C program, for example, variables, functions, arrays, structures, unions, labels, etc. An identifier can be composed of letters such as uppercase, lowercase letters, underscore, digits, but the starting letter should be either an alphabet or an underscore. If the identifier is not used in the external linkage, then it is called as an internal identifier. If the identifier is used in the external linkage, then it is called as an external identifier

Rules for Naming an Identifier:

- Identifiers cannot be a keyword.
- Identifiers are case-sensitive.
- It can have a sequence of letters and digits. However, it must begin with a letter or `_`.
The first letter of an identifier cannot be a digit.
- It's a convention to start an identifier with a letter rather `_`.
- Whitespaces are not allowed.
- We cannot use special symbols like `!, @, #, $`, and so on.

Rules of c identifiers:

- The first character of an identifier should be either an alphabet or an underscore, and then it can be followed by any of the character, digit, or underscore.
- It should not begin with any numerical digit.
- In identifiers, both uppercase and lowercase letters are distinct. Therefore, we can say that identifiers are case sensitive.
- Commas or blank spaces cannot be specified within an identifier.
- Keywords cannot be represented as an identifier.
- The length of the identifiers should not be more than 31 characters.
- Identifiers should be written in such a way that it is meaningful, short, and easy to read.
- Starting an identifier with a single leading underscore indicates that the identifier is private identifier.
- Starting an identifier with two leading underscores indicates a strongly private identifier.
- If the identifier also ends with two trailing underscores, the identifier is a language-defined special name.

Embedded C Comments

Comments can be used to explain Python code. Comments can be used to make the code more readable. Comments can be used to prevent execution when testing code There are three types of comments available in Python

1. Single line Comments
2. Multiple line Comments

Single line Comments:

A single-line comment of c is the one that has a hashtag `//` at the beginning of it and continues until the finish of the line. If the comment continues to the next line, add a hashtag to the subsequent line and resume the conversation.

EX: `("Hello, World!") // This is a single line comment in c`

Multiline Comments:

A multiple comments you can comment multiple lines as follows

```
EX: //This is comment
    //This is comment, too
    //This is comment, too
```

Multi-Line comments are represented by slash asterisk `* ... *\.`. It can occupy many lines of code, but it can't be nested. Syntax:

EX:

```
/*This is
 a multiline
 comment. */
```

Preprocessor Directives

The C preprocessor is a micro processor that is used by compiler to transform your code before compilation. It is called micro preprocessor because it allows us to add macros.

All preprocessor directives starts with hash # symbol.

Pre-processor	Description
#include	It is used to include a file in a program
#define	It is used to define a macro or constant
#ifdef	It checks if a macro is defined or not
#ifndef	It checks if a macro is not defined
#if	It checks for a condition at compile time
#else	It specifies the alternative statement if the condition is false
#elif	It specifies the next condition if the previous #if statement is false
#endif	It denotes the end of the conditional statements
#pragma	It is used to provide additionl information to the compiler

Global Variables

In C programming, a global variable is a variable that is declared outside of any function and can be accessed by any function in the program. Unlike local variables, which are only accessible within their own function, global variables are visible to the entire program.

Declaring Global Variables in C

To declare a global variable in C, we will simply declare outside of any function using the following syntax:

```
data_type variable_name;
```

For example, to declare a global integer variable named count, we would use the following code:

```
int count;
```

This code declares a global variable named count of type int. Since it is declared outside of any function, it is accessible to all functions in the program.

Infinite Loops

Loops in programming are used to repeat a block of code until the specified condition is met. A loop statement allows programmers to execute a statement or group of statements multiple times without repetition of code.

There are mainly two types of loops in C Programming:

- **Entry Controlled loops:** In Entry controlled loops the test condition is checked before entering the main body of the loop. For Loop and While Loop is Entry-controlled loops.
- **Exit Controlled loops:** In Exit controlled loops the test condition is evaluated at the end of the loop body. The loop body will execute at least once, irrespective of whether the condition is true or false. do-while Loop is Exit Controlled loop.

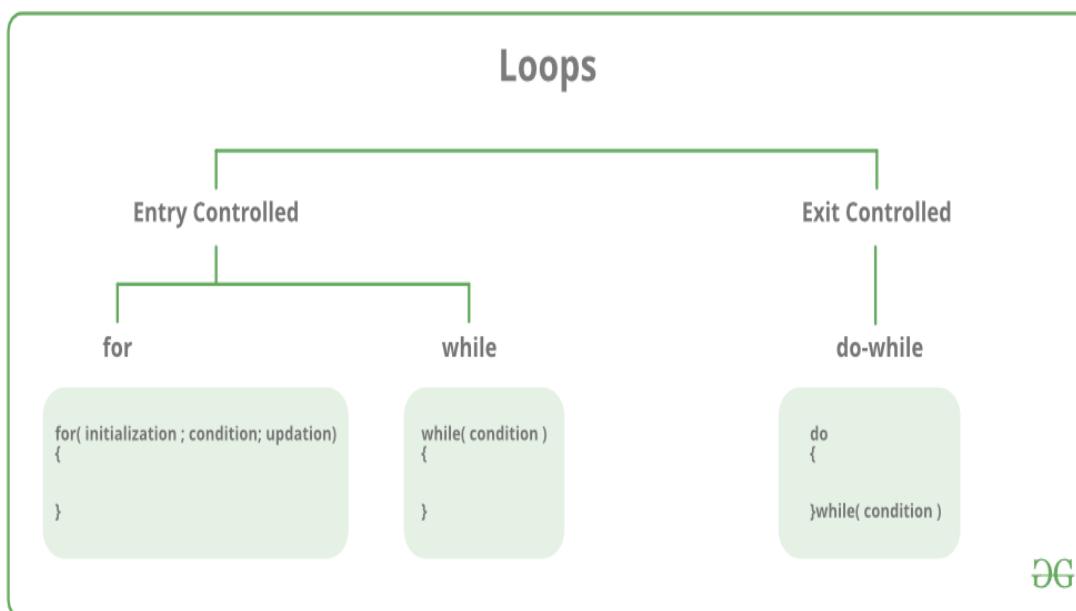


Fig.: Breakdown of Loops

For Loop:

For loop in C programming is a repetition control structure that allows programmers to write a loop that will be executed a specific number of times. for loop enables programmers to perform n number of steps together in a single line.

```

for (initialize expression; test expression; update expression)
{
    //
    // body of for loop
    //
}
    
```

Example:

```
#include<stdio.h>
Int main()
{
    for(int i = 0; i < n; ++i)
    {
        printf("Body of for loop which will execute till n");
    }
}
```

For loop Equivalent Flow Diagram:

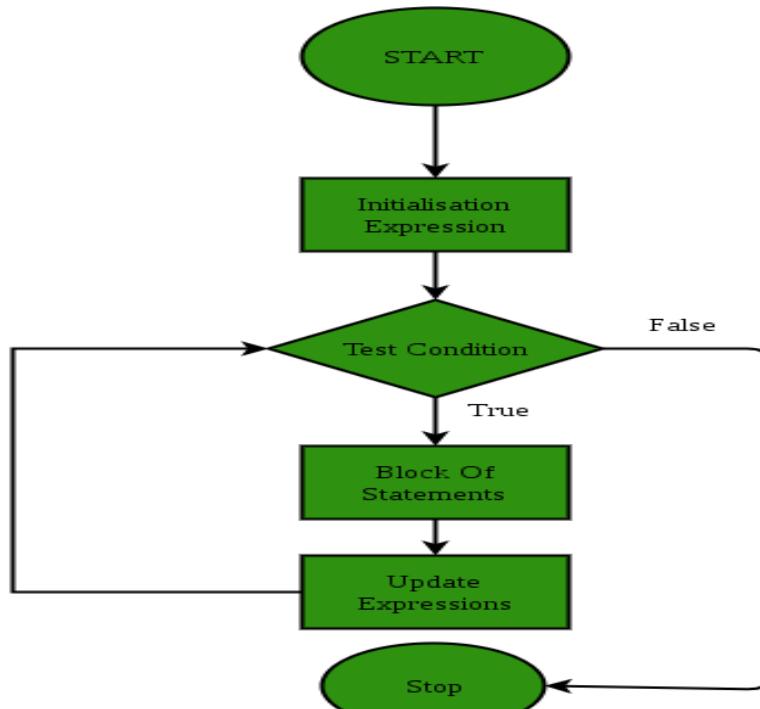


Fig.: Flowchart of For Loop

While Loop:

While loop does not depend upon the number of iterations. In for loop the number of iterations was previously known to us but in the While loop, the execution is terminated on the basis of the test condition. If the test condition will become false then it will break from the while loop else body will be executed.

Syntax:

```
while(initialize expression; test expression)
{
    initialize expression;
    // body of for loop
    update expression;
}
```

Example:

```
#include<stdio.h>
Int main()
{
    while(i < 10)
    {
        // loop body
        printf( "Hello World\n");
        // update expression
        i++;
    }
}
```

Flow Diagram for while loop:

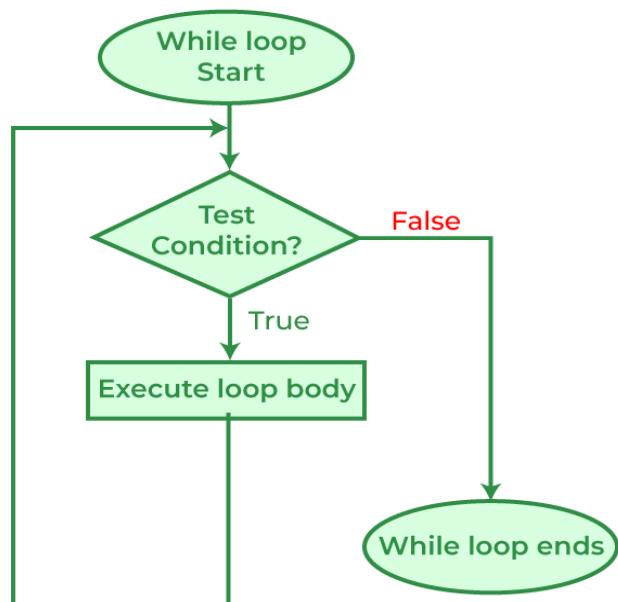


Fig.: Flowchart of while Loop

Do-While Loop:

The do-while loop is one of the three loop statements in C, the others being while loop and for loop. It is mainly used to traverse arrays, vectors, and other data structures.

```
do {
    // body of do-while loop
} while (condition);
```

Example:

```
#include<stdio.h>
Int main()
{
    //loop body
    printf( "Hello World\n");
    // Update expression
    i++;
    // Test expression
    } while (i < 1);
}
```

C do...while Loop Flowchart:

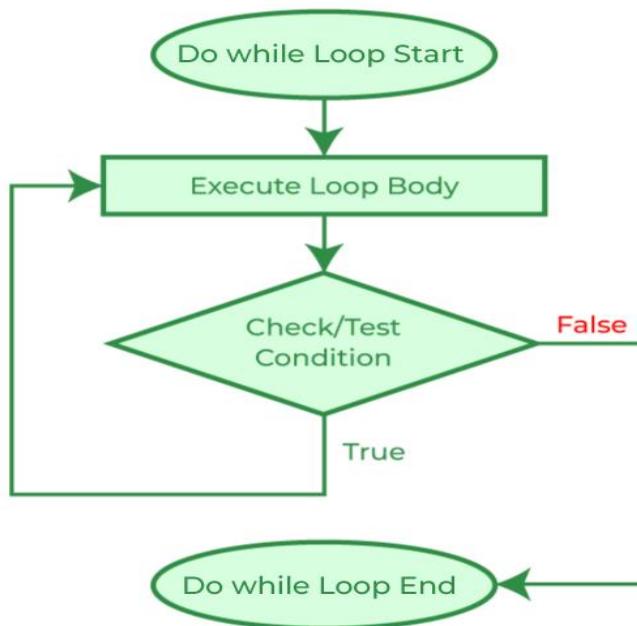


Fig.: Flowchart of do-while Loop

Statements In C

Decision making is anticipation of conditions occurring while execution of the program and specifying actions taken according to the conditions. Decision structures evaluate multiple expressions which produce TRUE or FALSE as outcome.

To determine which action to take and which statements to execute if outcome is TRUE or FALSE otherwise.

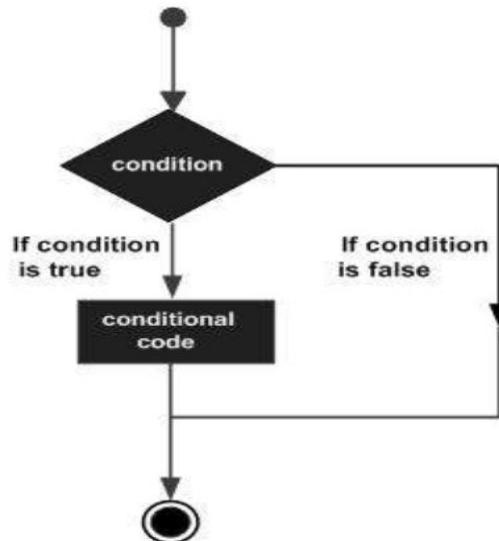


Fig.Work flow of condition

C programming language assumes any non-zero and non-null values as TRUE, and if it is either zero or null, then it is assumed as FALSE value. Decision-making statements in programming languages decide the direction of the flow of program execution. In P, if-else elif statement is used for decision making.

if statement

if statement is the most simple decision-making statement. It is used to decide whether a certain statement or block of statements will be executed or not i.e if a certain condition is true then a block of statement is executed otherwise not.

Syntax:

```

if(condition)
{
    // if body
    // Statements to execute if condition is true
}
  
```

C uses indentation to identify a block. So the block under an if statement will be identified

Example:

```

#include<stdio.h>
int main()
{
    int x = 20;
    int y = 18;
    if(x>y)
    {
        printf("x is greater than y");
    }
}
  
```

Flowchart of C if statement:

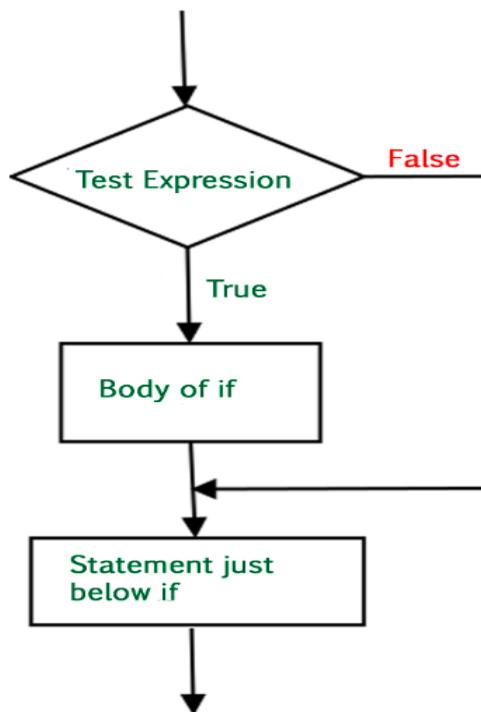


Fig.: Work flow diagram of C statement

- Here, the condition after evaluation will be either true or false.
- if the statement accepts Boolean values – if the value is true then it will execute the block of statements below it otherwise not.

if-else statement:

The if statement alone tells us that if a condition is true it will execute a block of statements and if the condition is false it won't. But what if we want to do something else if the condition is false. Here comes the else statement. We can use the else statement with if statement to execute a block of code when the condition is false.

Syntax:

```

if (condition)
{
    // code executed when the condition is true
}
else {
    // code executed when the condition is false
}
  
```

Example:

```
#include<stdio.h>
Int main()
{
    int time = 20;
    if(time<18)
    {
        Printf("Good day.");
    }
    else
    {
        Printf("Good evening");
    }
}
```

Flow Chart of C If-else statement:

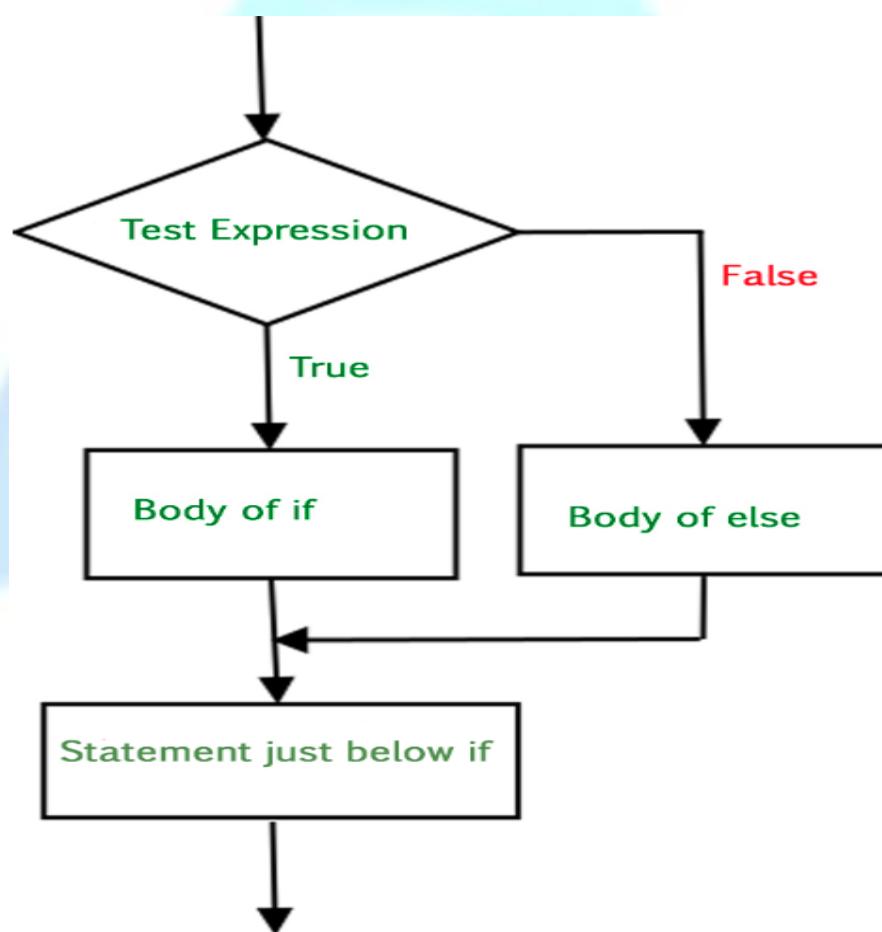


Fig.: Flowchart of if-else statement

nested-if statement:

A nested if is an if statement that is the target of another if statement. Nested if statements mean an if statement inside another if statement. C allows us to nest if statements within if statements. i.e, we can place an if statement inside another if statement.

Syntax:

```
if (condition1)
{
    // Executes when condition1 is true
    if (condition2)
    {
        // Executes when condition2 is true
    }
    else
    {
        // Executes when condition2 is false
    }
}
```

Example:

```
int a = 274;
printf("Value of a is : %d\n", a);
if (a < 100){
    printf("Value of a is less than 100\n");
}
if (a >= 100 && a < 200)
{
    printf("Value of a is between 100 and 200\n" );
}
if (a >= 200)
{
    printf("Value of a is more than 200\n" );
```

Flowchart of C Nested if Statement:

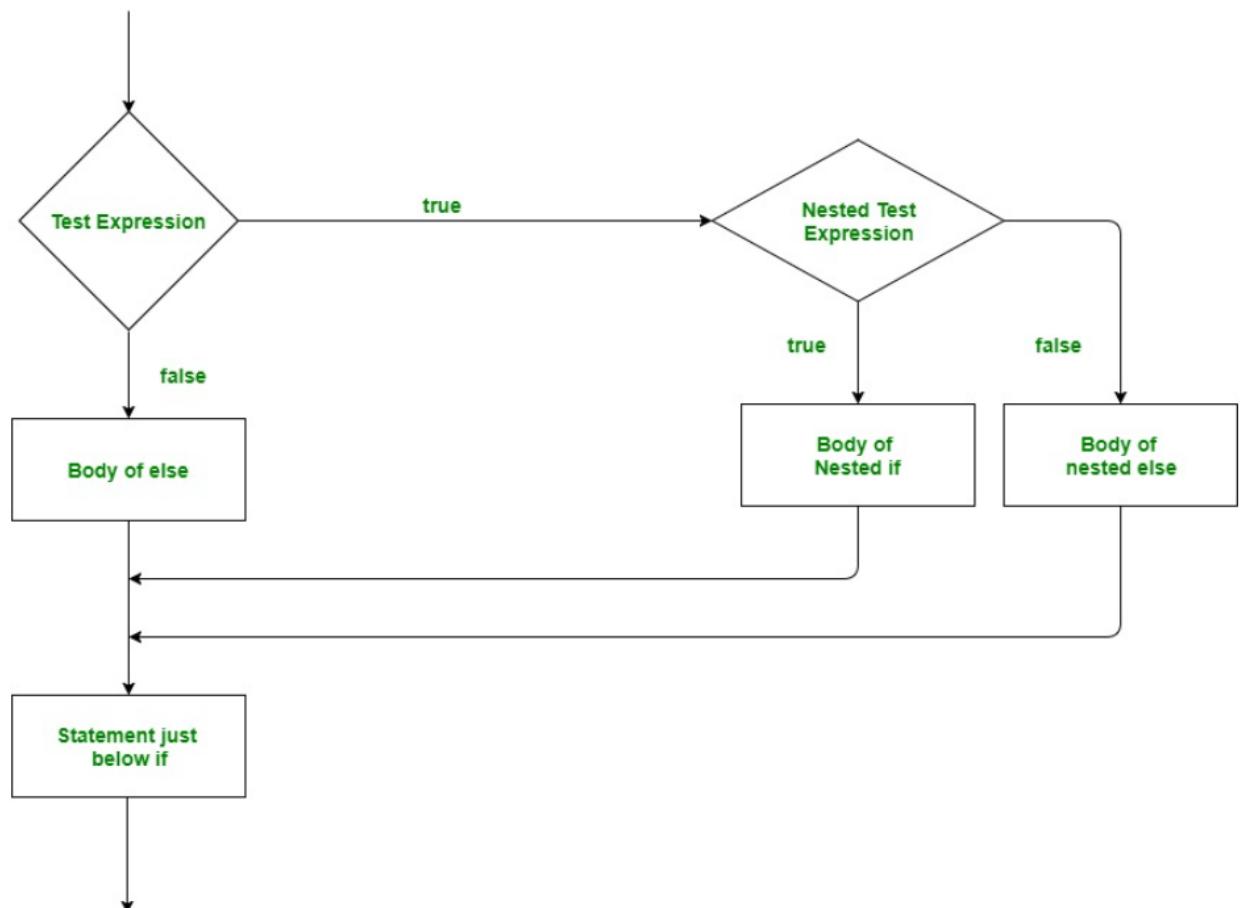


Fig.: Flowchart of nested-if statement

if-elif-else ladder statement:

Here, a user can decide among multiple options. The if statements are executed from the top down. As soon as one of the conditions controlling the if is true, the statement associated with that if is executed, and the rest of the ladder is bypassed. If none of the conditions is true, then the final else statement will be executed.

Syntax:

```

if (condition)
  statement;
else if (condition)
  statement;
  .
  .
else
  statement;
  
```

Example:

```
#include<stdio.h>
int main()
{
    int number=0;
    printf("enter a number:");
    scanf("%d",&number);
    if(number==10){
        printf("number is equals to 10");
    }
    else if(number==50)
    {
        printf("number is equal to 50");
    }
    else if(number==100)
    {
        printf("number is equal to 100");
    }
    else
    {
        printf("number is not equal to 10, 50 or 100");
    }
    return 0;
}
```

Flow Chart of C if elif else statements:

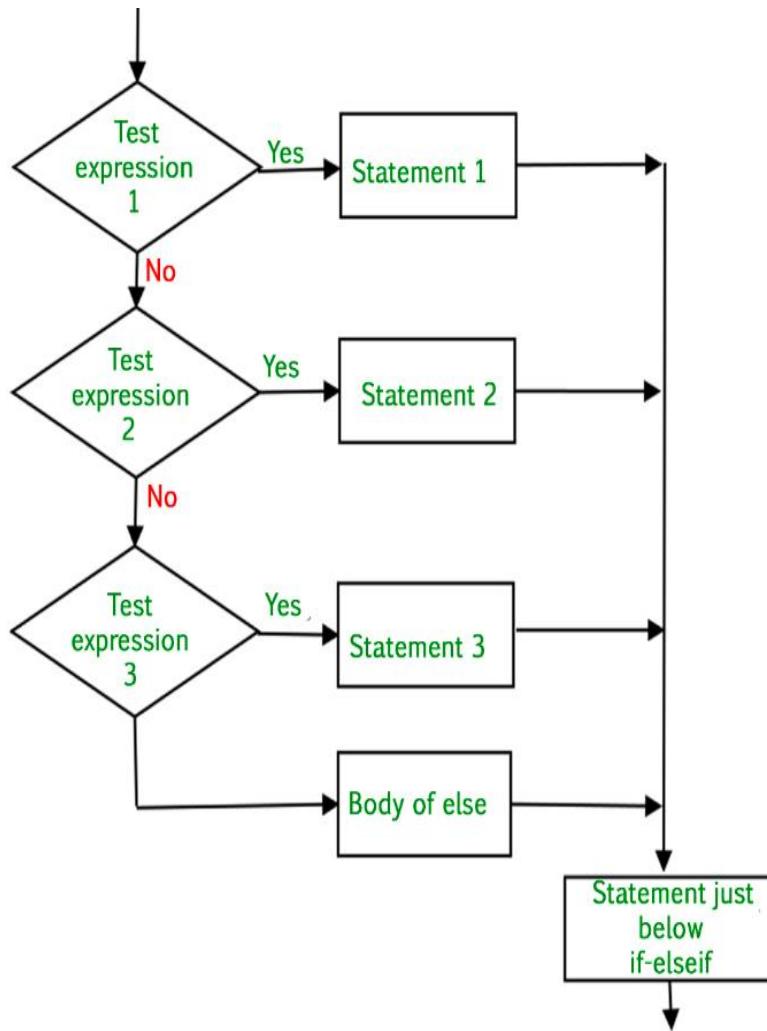


Fig.: Flowchart of if elif else statement

Function In C

Function Declaration:

Function Declaration introduces the name, return type, and parameters of a function to the compiler, while Function Definition provides the actual implementation or body of the function. Declarations enable calling the function elsewhere in the code, while definitions provide the code to be executed when the function is called. In this article, we will learn the difference between ‘function declaration’ and ‘function Definition’.

It provides information about the function’s interface without including its implementation. Function declarations enable other parts of the program to call the function without knowing its internal details.

Syntax:

```
void function_name(parameters);
```

Function call

Function call is a statement that instructs the compiler to execute the function. We use the function name and parameters in the function call.

In the below example, the first sum function is called and 10,30 are passed to the sum function. After the function call sum of a and b is returned and control is also returned back to the main function of the program.

Working of Function in C

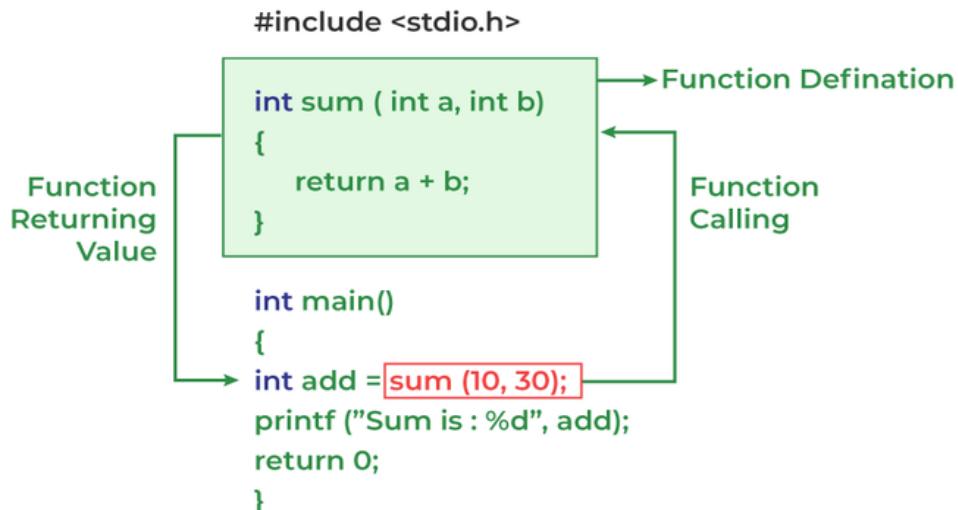


Fig.: C function working flow

Main Function:

The main function is an integral part of the programming languages such as C, C++, and Java. The main function in C is the entry point of a program where the execution of a program starts. It is a user-defined function that is mandatory for the execution of a program because when a C program is executed, the operating system starts executing the statements in the main() function.

Syntax of C main() Function

```

return_type main()
{
    // Statement 1;
    // Statement 2;
    // and so on..
    return;
}

```

Types of C main Functions

1. Main function with no arguments and void return type
2. Main function with no arguments and int return type
3. Main function with the Command Line Arguments

1. Main Function with No Arguments and Void Return Type

Let's see the example of the main function inside which we have written a simple program for printing a string. In the below code, first, we include a header file and then we define a main function inside which we write a statement to print a string using printf() function. The main function is called by the operating system itself and then executes all statements inside this function.

Syntax:

```
void main()
{
    // Function body
}
```

2. Main Function with No Arguments and int Return Type

In this example, we use the int return type in the main() function that indicates the exit status of the program. The exit status is a way for the program to communicate to the operating system whether the program was executed successfully or not. The convention is that a return value of 0 indicates that the program was completed successfully, while any other value indicates that an error occurred.

Syntax:

```
int main()
{
    // Function body
}
```

3. Main Function with the Command Line Arguments

In this example, we have passed some arguments in the main() function as seen in the below code. These arguments are called command line arguments and these are given at the time of executing a program.

The first argument argc means argument count which means it stores the number of arguments passed in the command line and by default, its value is 1 when no argument is passed.

The second argument is a char pointer array argv[] which stores all the command line arguments passed. We can also see in the output when we run the program without passing any command line argument the value of argc is 1.

Syntax:

```
int main(int argc, char* argv[])
{
    // Function body
}
```

Function Definitions

A function is a self-contained block of code that performs a specific task. Functions allow for modular programming, making the code easier to understand, test, and maintain.

Syntax of Function Definition in C/C++:

```
return_type function_name(parameter_list) {  
    // function body  
}
```

Components:

- **Return Type:** Specifies the type of value the function returns. It can be void if no value is returned.
- **Function Name:** Identifies the function. It should be meaningful and indicative of the function's purpose.
- **Parameter List:** Defines the inputs to the function. It consists of data types and variable names, separated by commas. It can be empty if the function takes no parameters.
- **Function Body:** Contains the statements that define what the function does. It is enclosed in curly braces {}.

Example:

```
int add(int a, int b) {  
    int result = a + b;  
    return result;  
}
```

This example defines a function add that takes two integers as parameters, calculates their sum, and returns the result.

Benefits of Using Functions:

- **Modularity:** Functions break down complex problems into smaller, manageable parts.
- **Reusability:** Functions can be reused in different parts of the program or in different programs.
- **Maintainability:** Functions make it easier to update and debug code.

Local Variables

Local variables are declared within a function and are only accessible within that function. They are created when the function is called and destroyed when the function exits, ensuring that their scope is limited to the function.

Key Characteristics:

- **Scope:** The scope of a local variable is limited to the function in which it is declared.
- **Lifetime:** Local variables exist only during the execution of the function. Once the function exits, the local variables are destroyed, and their memory is reclaimed.
- **Storage:** Local variables are typically stored on the stack, which makes access to them very fast.

Example:

```
void exampleFunction() {  
    int localVariable = 5; // local variable  
    printf("Local variable value: %d\n", localVariable);  
}
```

In this example, localVariable is a local variable with a scope limited to exampleFunction. It is created when exampleFunction is called and destroyed when the function exits.

Advantages of Local Variables:

- **Memory Efficiency:** Local variables are temporary and use memory only when needed.
- **Avoiding Conflicts:** Local variables prevent naming conflicts because they are not visible outside their function.
- **Security:** Local variables provide a level of data security since they cannot be accessed outside their function.



Arduino IDE

Introduction

The Arduino IDE provides a user-friendly environment for developing and managing Arduino projects. Key components of the IDE include:

- **Editor Window:** Where you write and edit your code (sketches).
- **Toolbar:** Contains buttons for common actions like verifying (compiling) code, uploading code to the board, opening and saving sketches, and opening the serial monitor.
- **Message Area:** Displays notifications and errors.
- **Text Console:** Shows detailed information about the compilation and upload process.
- **Board and Port Selection:** Allows you to select the connected Arduino board and the corresponding port.

Project Creation

To create a new project (sketch) in the Arduino IDE:

1. Open the Arduino IDE:

Launch the IDE and you will see a blank sketch with the default structure:

```
void setup() {  
    // Initialization code here  
}  
  
void loop() {  
    // Main code here  
}
```

2. Write Your Code:

Enter your initialization code within the setup() function and your main code within the loop() function. For example, a simple LED blink sketch:

```
void setup() {  
    pinMode(LED_BUILTIN, OUTPUT);  
}  
  
void loop() {  
    digitalWrite(LED_BUILTIN, HIGH);  
    delay(1000);  
    digitalWrite(LED_BUILTIN, LOW);  
    delay(1000);  
}
```

3. Save Your Sketch:

Go to File > Save and give your sketch a name.

Board and Library Installation

Before uploading your code, ensure the correct board and libraries are installed.

1. Board Installation:

- Go to Tools > Board > Boards Manager.
- In the Boards Manager window, search for your board (e.g., Arduino Uno) and click Install.

2. Library Installation:

- Libraries extend the functionality of the Arduino environment.
- Go to Sketch > Include Library > Manage Libraries.
- Search for the required library and click Install.

Compiling and Uploading

1. Compile Your Code:

- Click the checkmark button (✓) in the toolbar to compile your code. The IDE will verify your code for errors.

2. Upload Your Code:

- Connect your Arduino board to your computer via USB.
- Select the appropriate board and port from Tools > Board and Tools > Port.
- Click the right arrow button (→) to upload your code to the board.

Arduino Uno: Pin Out, Pin Configuration, and Specifications

Pin Out

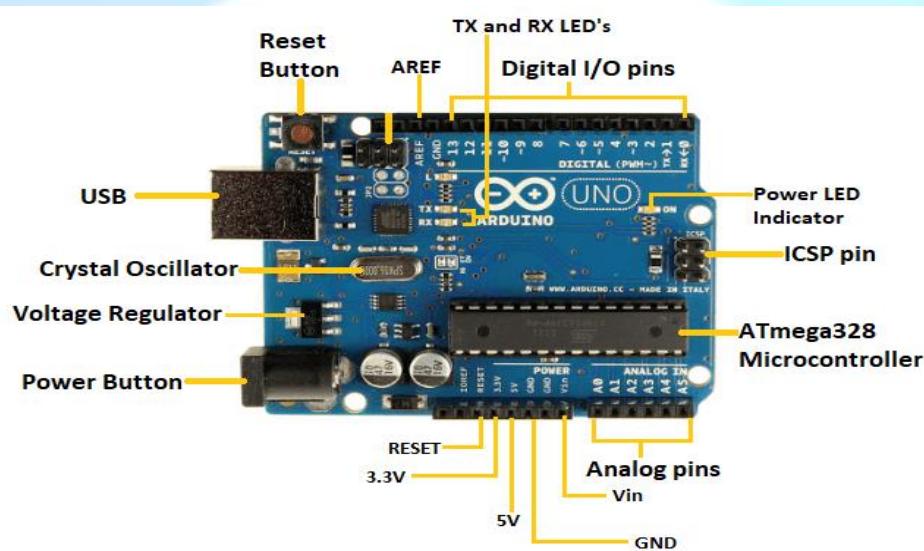


Fig.: Arduino uno pin out

The Arduino Uno board features several key components:

- **Microcontroller:** ATmega328P, which is the brain of the board.
- **Power Supply:** Can be powered via USB or an external power supply.
- **Digital I/O Pins:** 14 digital pins (6 can be used as PWM outputs).
- **Analog Input Pins:** 6 analog input pins.
- **Power Pins:** Includes VIN, 3.3V, 5V, GND, and a reset pin.
- **Crystal Oscillator:** Provides a clock signal to the microcontroller.
- **USB Connection:** Used for programming and communication with the computer.
- **Reset Button:** Resets the microcontroller.
- **Voltage Regulator:** Regulates the voltage supplied to the microcontroller.

Pin Configuration

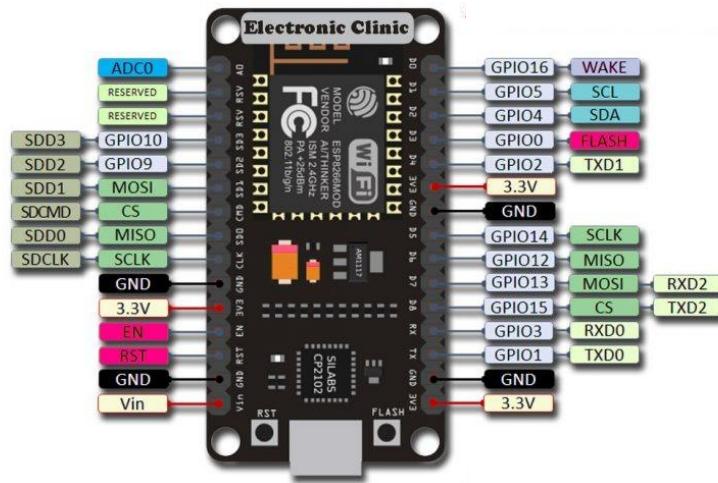
- **Digital Pins (0-13):** Used for general-purpose input/output (GPIO).
- **PWM Pins (3, 5, 6, 9, 10, 11):** Provide pulse-width modulation output.
- **Analog Pins (A0-A5):** Used to read analog sensors, provide a 10-bit resolution.
- **Power Pins:**
 - **VIN:** Input voltage to the Arduino board when using an external power source.
 - **3.3V:** 3.3V supply generated by the onboard regulator.
 - **5V:** 5V supply from the regulator.
 - **GND:** Ground pins.
 - **Reset:** Can reset the microcontroller when connected to ground.

Specifications

- **Microcontroller:** ATmega328P
- **Operating Voltage:** 5V
- **Input Voltage (recommended):** 7-12V
- **Input Voltage (limits):** 6-20V
- **Digital I/O Pins:** 14 (6 PWM outputs)
- **Analog Input Pins:** 6
- **DC Current per I/O Pin:** 20 mA
- **DC Current for 3.3V Pin:** 50 mA
- **Flash Memory:** 32 KB (ATmega328P) of which 0.5 KB used by bootloader
- **SRAM:** 2 KB (ATmega328P)
- **EEPROM:** 1 KB (ATmega328P)
- **Clock Speed:** 16 MHz

ESP8266: Pin Out, Pin Configuration, and Specifications

Pin Out



The ESP8266 Wi-Fi module integrates several key components:

- **Microcontroller:** Tensilica L106 32-bit RISC processor, running at 80 MHz.
- **Wi-Fi Module:** Supports IEEE 802.11 b/g/n standard.
- **Flash Memory:** External flash memory used for storing user programs.
- **GPIO Pins:** General-purpose input/output pins.
- **ADC (Analog to Digital Converter):** For reading analog sensors.
- **UART:** Universal Asynchronous Receiver/Transmitter for serial communication.
- **SPI/I2C:** Serial Peripheral Interface and Inter-Integrated Circuit for peripheral communication.
- **Power Management Unit:** Manages power consumption of the module.

Pin Configuration

- **GPIO Pins:** Used for digital I/O operations. The number of GPIO pins varies with different ESP8266 modules.
- **ADC (A0):** Single analog input pin for reading analog sensors.
- **UART Pins:** TX (Transmit) and RX (Receive) for serial communication.
- **SPI Pins:** SCLK (Clock), MISO (Master In Slave Out), MOSI (Master Out Slave In), and CS (Chip Select).
- **Power Pins:**
 - **VCC:** 3.3V power supply.
 - **GND:** Ground pin.
 - **EN (Enable):** Used to enable or disable the module.
 - **RST (Reset):** Used to reset the module.

Specifications

- **Microcontroller:** Tensilica L106 32-bit RISC processor
- **Operating Voltage:** 3.0V to 3.6V
- **Wi-Fi:** 802.11 b/g/n
- **Flash Memory:** Varies (commonly 4 MB)
- **SRAM:** 160 KB
- **GPIO Pins:** Varies (typically 17 GPIOs)
- **ADC:** 10-bit resolution
- **UART, SPI, I2C:** Supported
- **Clock Speed:** 80 MHz (can be overclocked to 160 MHz)
- **Power Consumption:**
 - **Deep Sleep:** 10 µA
 - **Modem Sleep:** 15 mA
 - **Light Sleep:** 0.9 mA
 - **Active Mode:** 80 mA

Arduino Programming

Arduino is an open-source electronics platform that provides a flexible and user-friendly environment for creating interactive projects. It consists of both hardware and software components. It consists of a circuit board, which can be programmed (referred to as a microcontroller) and a ready-made software called Arduino IDE (Integrated Development Environment), which is used to write and upload the computer code to the physical board.

Arduino boards are able to read analog or digital input signals from different sensors and turn it into an output such as activating a motor, turning LED on/off, connect to the cloud and many other actions. You can control your board functions by sending a set of instructions to the microcontroller on the board via Arduino IDE (referred to as uploading software). Arduino does not need an extra piece of hardware (called a programmer) in order to load a new code onto the board. You can simply use a USB cable. Arduino IDE uses a simplified version of C and C++, making it easier to learn to program. Arduino provides a standard form factor that breaks the functions of the micro-controller into a more accessible package.

Hardware Components:

1. Arduino Board:

The core of the Arduino platform is the Arduino board, which comes in various models (e.g., Arduino Uno, Arduino Nano, Arduino Mega). Each board has a microcontroller, digital and analog pins, power pins, and communication ports.

2. Power Supply:

Arduino boards can be powered through USB, an external power supply, or a battery, depending on the model.

3. Input/Output Pins:

Digital Pins: Used for binary signals (ON/OFF, HIGH/LOW).

Analog Pins: Used for reading analog signals (range of values).

4. Sensors and Actuators:

Connect sensors (e.g., temperature sensors, light sensors) and actuators (e.g., motors, LEDs) to the input/output pins for interaction.

Software Components:

1. Arduino IDE (Integrated Development Environment):

The Arduino IDE is a software application that allows you to write, compile, and upload code to the Arduino board.

2. Arduino Programming Language:

Arduino uses a simplified version of C/C++ programming languages. It has its own set of functions and libraries to make programming easier for beginners.

Arduino Coding Basics

The "Arduino Programming Language", consists of several functions, variables and structures based on the C/C++ language. Arduino IDE (Integrated Development Environment) allows us to draw the sketch and upload it to the various Arduino boards using code.

Arduino programs can be divided in three main parts: Structure, Functions and Values (variables and constants).

Functions: for controlling the Arduino board and performing computations. For example, to read or write a state to a digital pin, map a value or use serial communication.

Variables: the Arduino constants, data types and conversions. E.g. int, boolean, array.

Structure: the elements of the Arduino (C++) code, such as

- sketch (loop(), setup())
- control structure (if, else, while, for)
- arithmetic operators (multiplication, addition, subtraction)
- comparison operators, such as == (equal to), != (not equal to), > (greater than).

Arduino Program Structure

Software structure consist of two main functions void setup() and void loop(). The "void" indicates that nothing is returned on execution.

1. Void setup() function

```
void setup() {
    //program configurations here
}
```

This function is called once when the Arduino starts or resets. It's used for initializing variables, setting up pin modes (input or output), the baud rate of serial communication or the initialization of a library and other one-time configuration tasks.

2. Void loop() function

```
void loop() {
    //main program here
}
```

This function runs continuously after the setup() function. It initializes and sets the initial values, the loop() function does precisely what its name suggests, and loops consecutively, allowing your program to change and respond. Use it to actively control the Arduino board. This is where we write the code that we want to execute over and over again, such as turning on/off a lamp based on an input, or to conduct a sensor reading every X second.

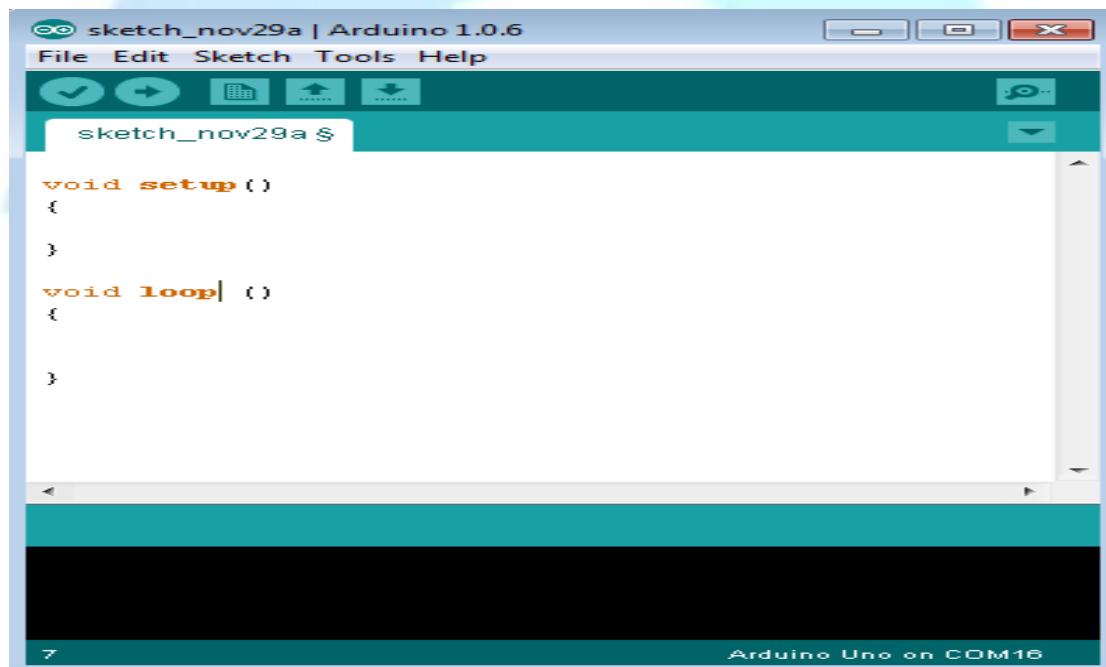


Fig.: Arduino IDE SKETCH

The "Sketch"

In the Arduino project, a program is referred to as a "sketch". A sketch is a file that you write your program inside. It has the .ino extension, and is always stored in a folder of the same name. The folder can include other files, such as a **header file**, that can be included in your sketch.

Libraries

Arduino libraries are an extension of the standard Arduino API, and consists of **thousands of libraries**, both official and contributed by the community.

Libraries simplifies the use of complex code, such as reading a specific sensor, controlling a motor or connecting to the Internet. Instead of having to write all of this code yourself, you can just install a library, include it at the top of your code, and use any of the available functionalities of it. All Arduino libraries are open source and free to use by **anyone**.

To use a library, you need to include it at the top of your code, as the example below:

```
#include <Library.h>
```

Most libraries also have a set of examples that are useful to get started with the library.

Arduino Delay

Arduino Delay specifies the delay() function used in the Arduino programming. The delay() function pauses the program or task for a specified duration of time. The time is specified inside the open and closed parentheses in milliseconds.

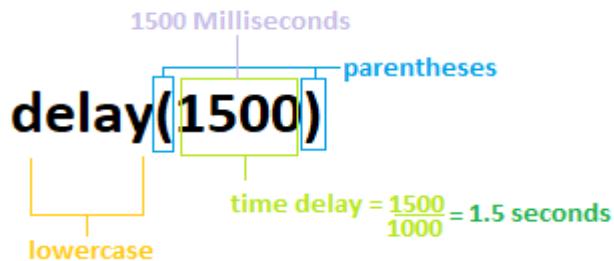


Fig.:Syntax of delay

Where,

1 second = 1000 milliseconds

The program waits for a specified duration before proceeding onto the next line of the code.

The delay() function allows the unsigned long data type in the code.

Arduino Variables

The variables are defined as the place to store the data and values. It consists of a name, value, and type. The variables can belong to any data type such as int, float, char, etc

For example: int pin = 8;

Here, the int data type is used to create a variable named pin that stores the value 8. It also means that value 8 is initialized to the variable pin.

If we have not declared the variable, the value can also be directly passed to the function.

Advantages of Variables

1. We can use a variable many times in a program.
2. The variables can represent integers, strings, characters, etc.
3. It increases the flexibility of the program.
4. We can easily modify the variables. For example, if we want to change the value of variable LEDpin from 8 to 13, we need to change the only point in the code.
5. We can specify any name for a variable. For example, greenpin, bluePIN, REDpin, etc.

Variables Scope

The variables can be declared in two ways in Arduino, they are:

1. Local variables
2. Global variables

Local Variables

The local variables are declared within the function. The variables have scope only within the function. These variables can be used only by the statements that lie within that function.

For example,

```
void setup() {  
    Serial.begin(9600);  
}  
  
void loop() {  
    int x = 3;  
    int b = 4;  
    int sum = 0;  
    sum = x + b;  
    Serial.println(sum);  
}
```

Global Variables

The global variables can be accessed anywhere in the program. The global variable is declared outside the setup() and loop() function.

For example,

Consider the below code.

```
int LEDpin = 8;  
  
void setup() {  
    pinMode(LEDpin, OUTPUT);  
}  
  
void loop() {  
    digitalWrite(LEDpin, HIGH);  
}
```

We can notice that the LEDpin is used both in the loop() and setup() functions.

The basics to start with Arduino programming.

Brackets

There are two types of brackets used in the Arduino coding, which are listed below:

1. Parentheses ()

The parentheses brackets are the group of the arguments, such as method, function, or a code statement. These are also used to group the math equations.

2. Curly Brackets { }

The statements in the code are enclosed in the curly brackets. We always require closed curly brackets to match the open curly bracket in the code or sketch.

Open curly bracket- '{ '

Closed curly bracket - ' } '

Line Comment

There are two types of line comments, which are listed below:

1. Single line comment
2. Multi-line comment

// Single line comment

The text that is written after the two forward slashes are considered as a single line comment. The compiler ignores the code written after the two forward slashes. The comment will not be displayed in the output. Such text is specified for a better understanding of the code or for the explanation of any code statement.

The // (two forward slashes) are also used to ignore some extra lines of code without deleting it.

/* Multi - line comment */

The Multi-line comment is written to group the information for clear understanding. It starts with the single forward slash and an asterisk symbol (/ *). It also ends with the / *. It is commonly used to write the larger text. It is a comment, which is also ignored by the compiler.

Serial Communication

Serial communication is essential to Arduino programming, as it is the easiest way to know what goes on on your board. For this, we can use the **Serial** class.

Serial.begin()

Initializes serial communication between board & computer. This is defined in the void **setup()** function, where you also specify baud rate (speed of communication). The **baud** rate signifies the data rate in bits per second.

The default baud rate in Arduino is **9600 bps (bits per second)**. We can specify other baud rates as well, such as 4800, 14400, 38400, 28800, etc.

```
void setup() {
```

```
    Serial.begin(9600);
```

```
}
```

The `Serial.begin()` is declared in two formats, which are shown below:

```
begin( speed )
```

```
begin( speed, config)
```

Where,

`serial`: It signifies the serial port object.

`speed`: It signifies the baud rate or bps (bits per second) rate. It allows long data types.

`config`: It sets the stop, parity, and data bits.

Example 1:

```
void setup ()
```

```
{
```

```
    Serial.begin(4800);
```

```
}
```

```
void loop ()
```

```
{
```

```
}
```

The `serial.begin(4800)` open the serial port and set the bits per rate to 4800. The messages in Arduino are interchanged with the serial monitor at a rate of 4800 bits per second.

Example 2:

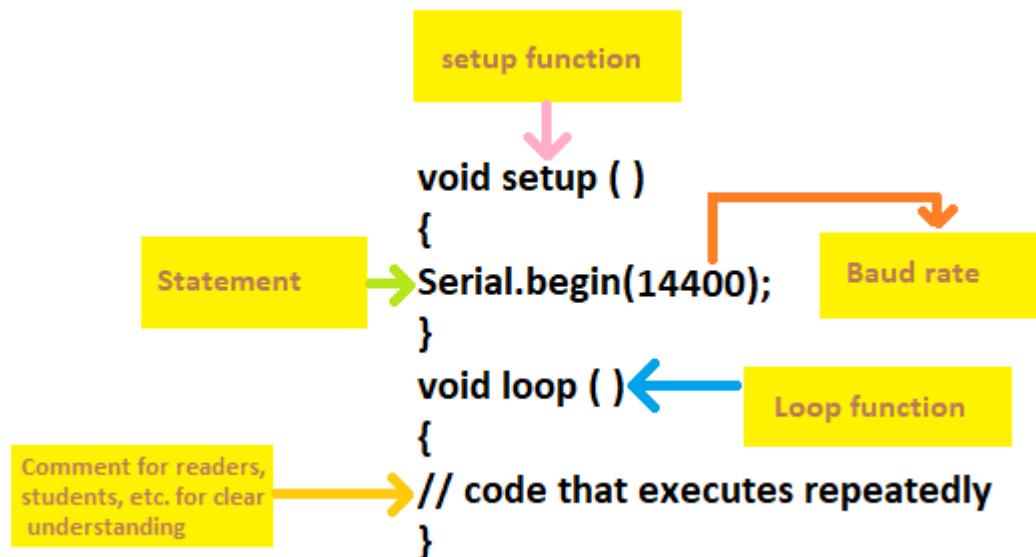


Fig.: Arduino Ide code

Serial.print()

Prints data to the serial port, which can be viewed in the Arduino IDE Serial Monitor tool.

```
void loop() {
  Serial.print();
}
```

Serial.read()

Reads the incoming serial data.

```
void loop() {
  int incomingByte = Serial.read();
```

}

GPIO / Pin Management

Configuring, controlling and reading the state of a digital/analog pin on an Arduino.

pinMode()

Configures a digital pin to behave as an input or output. Is configured inside the void setup() function.

```
pinMode(pin, INPUT); //configures pin as an input  
pinMode(pin, OUTPUT); //configures pin as an output  
pinMode(pin, INPUT_PULLUP); //enables the internal pull-up resistor
```

digitalRead()

Reads the state of a digital pin. Used to for example detect a button click.

There are usually 14 digital pins in the Arduino Boards which can be used for the read and write functions. When the status of any specific pin is to be known then the digitalRead() function is used. This function is a return type function as it will tell the status of the pin in its output.

```
int state = digitalRead(pin); //store the state in the "state" variable
```

digitalWrite()

Writes a high or low state to a digital pin. Used to switch on or off a component. **When a state is to be assigned to any pin then a digitalWrite() function is used.** The digitalWrite() function has two arguments one is the pin number and other is the state that will be defined by the user.

```
digitalWrite(pin, HIGH); // writes a high (1) state to a pin (aka turn it on)
```

```
digitalWrite(pin, LOW); // writes a low (0) state to a pin (aka turn it off)
```

Both functions are of Boolean type so, only two types of states are used in digital write function one is high and the other is low.

analogRead()

Reads the voltage of an analog pin, and returns a value between 0-1023 (10-bit resolution) and for 12 bits resolution the range will be 0 to 4095 . Used to read analog components. The bit resolution is the analog to digital conversion so for 10 bit the range can be calculated by 2^{10} and for 12 bits it will be 2^{12} respectively.

```
sensorValue = analogRead(A1); //stores reading of A1 in "sensorValue" variable
```

analogWrite()

To assign a state to any analog pin on the Arduino board the function analogWrite() is used. It will generate the pulse modulation wave and the state will be defined by giving its duty cycle that ranges from 0 to 255

Writes a value between 0-255 (8-bit resolution). Used for dimming lights or setting the speed of a motor. Also referred to as PWM, or Pulse Width Modulation.

```
analogWrite(pin, value); //write a range between 0-255 to a specific pin
```

PWM is only available on specific pins (marked with a "~" symbol).

Data types in Arduino

Data types in C refers to an extensive system used for declaring variables or functions of different types. The type of a variable determines how much space it occupies in the storage and how the bit pattern stored is interpreted. The data types used in Arduino programming are

void	Boolean	char	Unsigned char	byte	int	Unsigned int	word
long	Unsigned long	short	float	double	array	String-char array	String-object

void

The void keyword is used only in function declarations. It indicates that the function is expected to return no information to the function from which it was called.

Example

```
Void Loop ( ) {
```

```
    // rest of the code  
}
```

Boolean

A Boolean holds one of two values, true or false. Each Boolean variable occupies one byte of memory.

Example

```
boolean val = false ; // declaration of variable with type boolean and initialize it with false  
boolean state = true ; // declaration of variable with type boolean and initialize it with true
```

Char

The char datatype can store any number of character set. An identifier declared as the char becomes a character variable. The literals are written inside a single quote. The char type is often said to be an integer type. It is because, symbols, letters, etc., are represented in memory by associated number codes and that are only integers. The size of character data type is **minimum of 8 bits**. We can use the byte data type for an unsigned char data type of 8 bits or 1 byte. For example, character ' A ' has the ASCII value of 65.

The syntax is:

char var = val;

where,

var= variable

val = The value assigned to the variable.

ASCII TABLE

Decimal	Hex	Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char
0	0	32	20	[SPACE]	64	40	@	96	60	`
1	1	33	21	!	65	41	A	97	61	a
2	2	34	22	"	66	42	B	98	62	b
3	3	35	23	#	67	43	C	99	63	c
4	4	36	24	\$	68	44	D	100	64	d
5	5	37	25	%	69	45	E	101	65	e
6	6	38	26	&	70	46	F	102	66	f
7	7	39	27	'	71	47	G	103	67	g
8	8	40	28	(72	48	H	104	68	h
9	9	41	29)	73	49	I	105	69	i
10	A	42	2A	*	74	4A	J	106	6A	j
11	B	43	2B	+	75	4B	K	107	6B	k
12	C	44	2C	,	76	4C	L	108	6C	l
13	D	45	2D	-	77	4D	M	109	6D	m
14	E	46	2E	.	78	4E	N	110	6E	n
15	F	47	2F	/	79	4F	O	111	6F	o
16	10	48	30	0	80	50	P	112	70	p
17	11	49	31	1	81	51	Q	113	71	q
18	12	50	32	2	82	52	R	114	72	r
19	13	51	33	3	83	53	S	115	73	s
20	14	52	34	4	84	54	T	116	74	t
21	15	53	35	5	85	55	U	117	75	u
22	16	54	36	6	86	56	V	118	76	v
23	17	55	37	7	87	57	W	119	77	w
24	18	56	38	8	88	58	X	120	78	x
25	19	57	39	9	89	59	Y	121	79	y
26	1A	58	3A	:	90	5A	Z	122	7A	z
27	1B	59	3B	:	91	5B	[123	7B	{
28	1C	60	3C	<	92	5C	/	124	7C	
29	1D	61	3D	=	93	5D]	125	7D	}
30	1E	62	3E	>	94	5E	^	126	7E	~
31	1F	63	3F	?	95	5F	_	127	7F	[DEL]

Fig.: ASCII Table

unsigned char

Unsigned Char chr_y = 121 ;// declaration of variable with type Unsigned char and initialize it with character y

Example

Unsigned Char chr_y = 121 ;// declaration of variable with type Unsigned char and initialize it with character y

byte

A byte stores an 8-bit unsigned number, from 0 to 255.

Example

byte m = 25 ;//declaration of variable with type byte and initialize it with 25

int

Integers are the primary data-type for number storage. int stores a 16-bit (2-byte) value. This yields a range of -32,768 to 32,767 (minimum value of -2^{15} and a maximum value of $(2^{15} - 1)$).

The int size varies from board to board. On the Arduino Due, for example, an int stores a 32-bit (4-byte) value. This yields a range of -2,147,483,648 to 2,147,483,647 (minimum value of -2^{31} and a maximum value of $(2^{31}) - 1$).

Example

```
int counter = 32 ;// declaration of variable with type int and initialize it with 32
```

Unsigned int

Unsigned ints (unsigned integers) are the same as int in the way that they store a 2 byte value. Instead of storing negative numbers, however, they only store positive values, yielding a useful range of 0 to 65,535 ($2^{16} - 1$). The Due stores a 4 byte (32-bit) value, ranging from 0 to 4,294,967,295 ($2^{32} - 1$).

Example

```
Unsigned int counter = 60 ; // declaration of variable with
```

type unsigned int and initialize it with 60

Word

On the Uno and other ATMEGA based boards, a word stores a 16-bit unsigned number. On the Due and Zero, it stores a 32-bit unsigned number.

Example

```
word w = 1000 ;//declaration of variable with type word and initialize it with 1000
```

Long

Long variables are extended size variables for number storage, and store 32 bits (4 bytes), from -2,147,483,648 to 2,147,483,647.

Example

```
Long velocity = 102346 ;//declaration of variable with type Long and initialize it with 102346
```

unsigned long

Unsigned long variables are extended size variables for number storage and store 32 bits (4 bytes). Unlike standard longs, unsigned longs will not store negative numbers, making their range from 0 to 4,294,967,295 ($2^{32} - 1$).

Example

```
Unsigned Long velocity = 101006 ;// declaration of variable with
```

type Unsigned Long and initialize it with 101006

short

A short is a 16-bit data-type. On all Arduinos (ATMega and ARM based), a short stores a 16-bit (2-byte) value. This yields a range of -32,768 to 32,767 (minimum value of -2^{15} and a maximum value of $(2^{15}) - 1$).

Example: `short val = 13 ;//declaration of variable with type short and initialize it with 13`

float

Data type for floating-point number is a number that has a decimal point. Floating-point numbers are often used to approximate the analog and continuous values because they have greater resolution than integers.

Floating-point numbers can be as large as $3.4028235E+38$ and as low as $-3.4028235E+38$. They are stored as 32 bits (4 bytes) of information.

Example: `float num = 1.352;//declaration of variable with type float and initialize it with 1.352`

double

On the Uno and other ATMEGA based boards, Double precision floating-point number occupies four bytes. That is, the double implementation is exactly the same as the float, with no gain in precision. On the Arduino Due, doubles have 8-byte (64 bit) precision.

Example

`double num = 45.352 ;// declaration of variable with type double and initialize it with 45.352`

Arduino Operators

The operators are widely used in Arduino programming from basics to advanced levels. The operators are used to solve logical and mathematical problems. For example, to calculate the temperature given by the sensor based on some analog voltage.

The types of Operators classified in Arduino are:

1. Arithmetic Operators
2. Compound Operators
3. Boolean Operators
4. Comparison Operators
5. Bitwise Operators

Arithmetic Operators

There are six basic operators responsible for performing mathematical operations in Arduino, they are:

Assume variable A holds 10 and variable B holds 20 then

Operator name	Operator simple	Description	Example
assignment operator	=	Stores the value to the right of the equal sign in the variable to the left of the equal sign.	A = B
addition	+	Adds two operands	A + B will give 30
subtraction	-	Subtracts second operand from the first	A - B will give -10
multiplication	*	Multiply both operands	A * B will give 200
division	/	Divide numerator by denominator	B / A will give 2
modulo	%	Modulus Operator and remainder of after an integer division	B % A will give 0

Comparison Operators

Assume variable A holds 10 and variable B holds 20 then

Operator name	Operator simple	Description	Example
equal to	==	Checks if the value of two operands is equal or not, if yes then condition becomes true.	(A == B) is not true
not equal to	!=	Checks if the value of two operands is equal or not, if values are not equal then condition becomes true.	(A != B) is true
less than	<	Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true.	(A < B) is true
greater than	>	Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true.	(A > B) is not true
less than or equal to	<=	Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true.	(A <= B) is true
greater than or equal to	>=	Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true.	(A >= B) is not true

Compound Operators

The compound operators perform two or more calculations at once. The result of the right operand is assigned to the left operand. Assume variable A holds 10 and variable B holds 20 then

Operator name	Operator simple	Description	Example
increment	<code>++</code>	Increment operator, increases integer value by one	<code>A++</code> will give 11
decrement	<code>--</code>	Decrement operator, decreases integer value by one	<code>A--</code> will give 9
compound addition	<code>+=</code>	Add AND assignment operator. It adds right operand to the left operand and assign the result to left operand	<code>B += A</code> is equivalent to <code>B = B + A</code>
compound subtraction	<code>-=</code>	Subtract AND assignment operator. It subtracts right operand from the left operand and assign the result to left operand	<code>B -= A</code> is equivalent to <code>B = B - A</code>
compound multiplication	<code>*=</code>	Multiply AND assignment operator. It multiplies right operand with the left operand and assign the result to left operand	<code>B *= A</code> is equivalent to <code>B = B * A</code>
compound division	<code>/=</code>	Divide AND assignment operator. It divides left operand with the right operand and assign the result to left operand	<code>B /= A</code> is equivalent to <code>B = B / A</code>
compound modulo	<code>%=</code>	Modulus AND assignment operator. It takes modulus using two operands and assign the result to left operand	<code>B %= A</code> is equivalent to <code>B = B % A</code>
compound bitwise or	<code> =</code>	bitwise inclusive OR and assignment operator	<code>A = 2</code> is same as <code>A = A 2</code>
compound bitwise and	<code>&=</code>	Bitwise AND assignment operator	<code>A &= 2</code> is same as <code>A = A & 2</code>

Bitwise Operators

The Bitwise operators operate at the **binary level**. These operators are quite easy to use. There are various bitwise operators. Assume variable A holds 60 and variable B holds 13 then

Operator name	Operator simple	Description	Example
and	&	Binary AND Operator copies a bit to the result if it exists in both operands.	(A & B) will give 12 which is 0000 1100
or		Binary OR Operator copies a bit if it exists in either operand	(A B) will give 61 which is 0011 1101
xor	^	Binary XOR Operator copies the bit if it is set in one operand but not both.	(A ^ B) will give 49 which is 0011 0001
not	~	Binary Ones Complement Operator is unary and has the effect of 'flipping' bits.	(~A) will give -60 which is 1100 0011
shift left	<<	Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand.	A << 2 will give 240 which is 1111 0000
shift right	>>	Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand.	A >> 2 will give 15 which is 0000 1111

Boolean Operators

Boolean operators in Arduino programming are essential for making logical decisions based on true or false conditions.

These operators, namely AND (`&&`), OR (`||`), and NOT (`!`), allow us to combine multiple conditions and evaluate whether they are true or false. By doing so, we can control the flow of our Arduino program based on logical outcomes.

Boolean operators in Arduino programming are essential for making logical decisions based on true or false conditions.

Operator name	Operator simple	Description	Example
and	<code>&&</code>	Called Logical AND operator. If both the operands are non-zero then condition becomes true.	(A && B) is true
or	<code> </code>	Called Logical OR Operator. If any of the two operands is non-zero then condition becomes true.	(A B) is true
not	<code>!</code>	Called Logical NOT Operator. Use to reverses the logical state of its operand. If a condition is true then Logical NOT operator will make false.	!(A && B) is false

Arduino Control Statements or Conditional Statements

Decision making structures require that the programmer specify one or more conditions to be evaluated or tested by the program. It should be along with a statement or statements to be executed if the condition is determined to be true, and optionally, other statements to be executed if the condition is determined to be false

The general form of a typical decision making structure is shown in the figure below

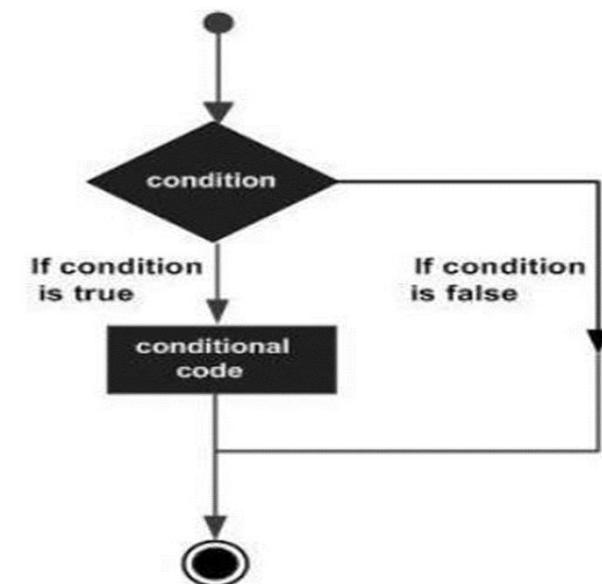


Fig.: Conditional Statements

S.NO.	Control Statement & Description
1	If statement It takes an expression in parenthesis and a statement or block of statements. If the expression is true then the statement or block of statements gets executed otherwise these statements are skipped.
2	If ...else statement An if statement can be followed by an optional else statement, which executes when the expression is false.
3	If...else if ...else statement The if statement can be followed by an optional else if...else statement, which is very useful to test various conditions using single if...else if statement.
4	switch case statement Similar to the if statements, switch...case controls the flow of programs by allowing the programmers to specify different codes that should be executed in various conditions.
5	Conditional Operator ? : The conditional operator ? : is the only ternary operator in C.

Arduino – Loops

Programming languages provide various control structures that allow for more complicated execution paths.

A loop statement allows us to execute a statement or group of statements multiple times and following is the general form of a loop statement in most of the programming languages

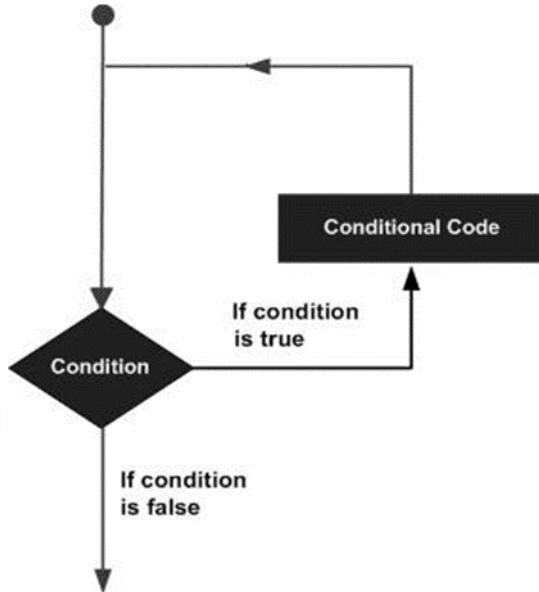


Fig.: Loop flow

S.NO.	Loop & Description
1	while loop while loops will loop continuously, and infinitely, until the expression inside the parenthesis, () becomes false. Something must change the tested variable, or the while loop will never exit.
2	do...while loop The do...while loop is similar to the while loop. In the while loop, the loop-continuation condition is tested at the beginning of the loop before performed the body of the loop.
3	for loop A for loop executes statements a predetermined number of times. The control expression for the loop is initialized, tested and manipulated entirely within the for loop parentheses.
4	Nested Loop C language allows you to use one loop inside another loop. The following example illustrates the concept.
5	Infinite loop It is the loop having no terminating condition, so the loop becomes infinite.

Arduino Functions

Functions allow structuring the programs in segments of code to perform individual tasks. The typical case for creating a function is when one needs to perform the same action multiple times in a program.

Standardizing code fragments into functions

—

1. Functions help the programmer stay organized. Often this helps to conceptualize the program.
2. Functions codify one action in one place so that the function only has to be thought about and debugged once.
3. This also reduces chances for errors in modification, if the code needs to be changed.
4. Functions make the whole sketch smaller and more compact because sections of code are reused many times.
5. They make it easier to reuse code in other programs by making it modular, and using functions often makes the code more readable.

There are two required functions in an Arduino sketch or a program i.e. `setup()` and `loop()`. Other functions must be created outside the brackets of these two functions.

The most common syntax to define a function is –

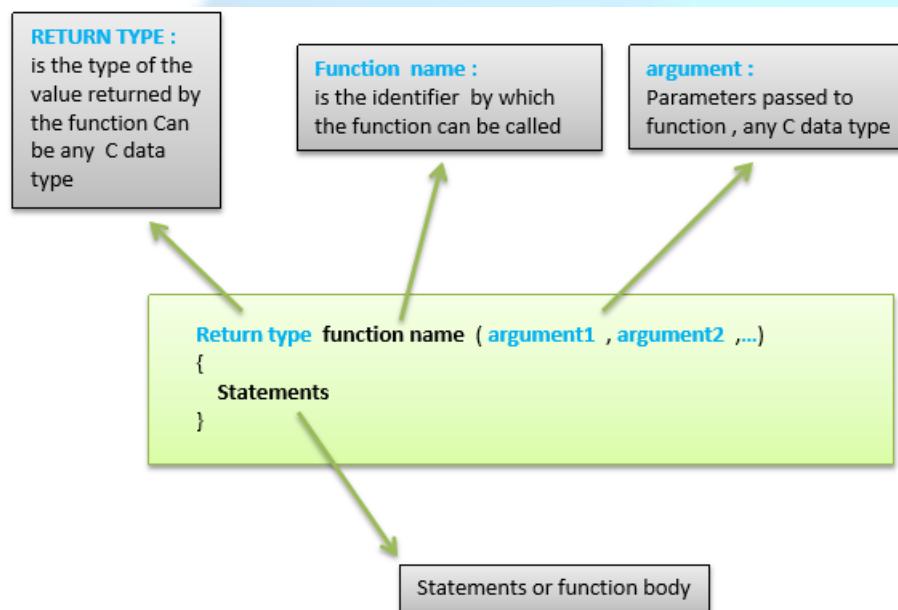


Fig.: Function Syntax

Function Declaration

A function is declared outside any other functions, above or below the loop function.

We can declare the function in two different ways –

The first way is just writing the part of the function called a function prototype above the loop function, which consists of –

- Function return type
- Function name
- Function argument type, no need to write the argument name

Function prototype must be followed by a semicolon (;).

The following example shows the demonstration of the function declaration using the first method.

Example

```
int sum_func (int x, int y) // function declaration {  
    int z = 0;  
    z = x+y ;  
    return z; // return the value  
}  
  
void setup () {  
    Statements // group of statements  
}  
  
void loop () {  
    int result = 0 ;  
    result = Sum_func (5,6) ; // function call  
}
```

The second part, which is called the function definition or declaration, must be declared below the loop function, which consists of –

- Function return type
- Function name

- Function argument type, here you must add the argument name
- The function body (statements inside the function executing when the function is called)

The following example demonstrates the declaration of function using the second method.

Example

```
int sum_func (int , int ) ; // function prototype

void setup () {

    Statements // group of statements

}

Void loop () {

    int result = 0 ;

    result = Sum_func (5,6) ; // function call

}

int sum_func (int x, int y) // function declaration {

    int z = 0;

    z = x+y ;

    return z; // return the value

}
```

The second method just declares the function above the loop function.



Installation

The Arduino Integrated Development Environment (IDE) is a software application used to write, compile, and upload code to Arduino-compatible boards. The installation process is straightforward:

1. Download the Arduino IDE:

- Visit the Arduino Software page and download the version compatible with your operating system (Windows, macOS, or Linux).

2. Install the IDE:

- For **Windows**: Run the downloaded installer and follow the on-screen instructions.
- For **macOS**: Open the downloaded .dmg file and drag the Arduino application into the Applications folder.
- For **Linux**: Extract the downloaded archive and run the install.sh script.

3. Launch the Arduino IDE:

- After installation, open the Arduino IDE from the Start menu (Windows), Applications folder (macOS), or by running the arduino command (Linux).

HARDWARE PROGRAMMING (ESP8266)

ESP8266 INTERFACE WITH IR SENSOR

HARDWARE REQUIREMENTS

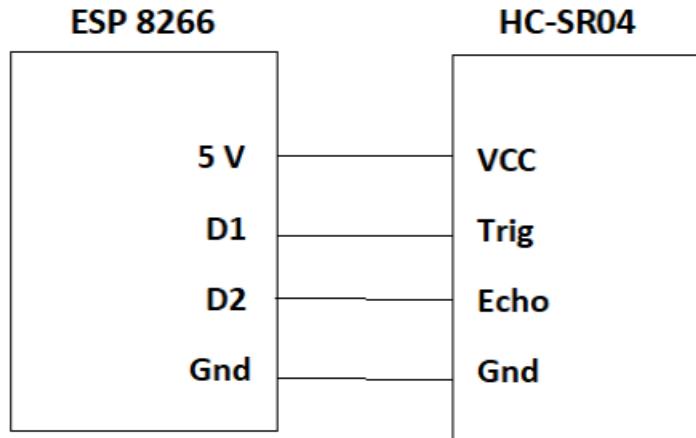
- ESP8266 module
- IR sensor
- Resistors (as needed)
- Breadboard and jumper wires
- Power supply

OBJECTIVES

- Calculating and displaying the distance between the obstacle and the sensor
- Determining the distance to an object.
- Understand the working of Ultrasonic Sensor
- Understand the pin configuration of ESP 8266 and Ultrasonic Sensor
- Understand developing the circuit

- Understand developing code for interfacing ESP 8266 with Ultrasonic sensor

BLOCK DIAGRAM



PROGRAM

```
#define echo 4
#define trig 5
void setup() {
pinMode(trig,OUTPUT) ;
pinMode(echo,INPUT);
Serial.begin(9600);
Serial.println("Object distance calculator");
delay(1000);
}
void loop() {
int distance;
long duration;
digitalWrite(trig,LOW);
delayMicroseconds(2);
digitalWrite(trig,HIGH);
delayMicroseconds(10);
```

```

digitalWrite(trig,LOW);
duration=pulseIn(echo,HIGH);
distance=(0.034*duration)/2;
Serial.println("Distance:" +String(distance));
}

```

PROCEDURE

- Connect the HC-SR01 TRIG pin to the ESP8266 D1 pin.
- Connect the HC-SR01 ECHO pin to the ESP8266 D2 pin.
- Connect the HC-SR01 VCC pin to 5V or 3.3V.
- Connect ground to ground.

APPLICATIONS

- **Obstacle Detection:** Used in robots for obstacle avoidance.
- **Remote Control Systems:** Control various appliances using IR remotes.
- **Automatic Door Openers:** Detects presence and opens doors automatically.
- **Intruder Alarms:** Detects unauthorized entry.

ESP8266 INTERFACE WITH ULTRASONIC SENSOR (HC-SR04)

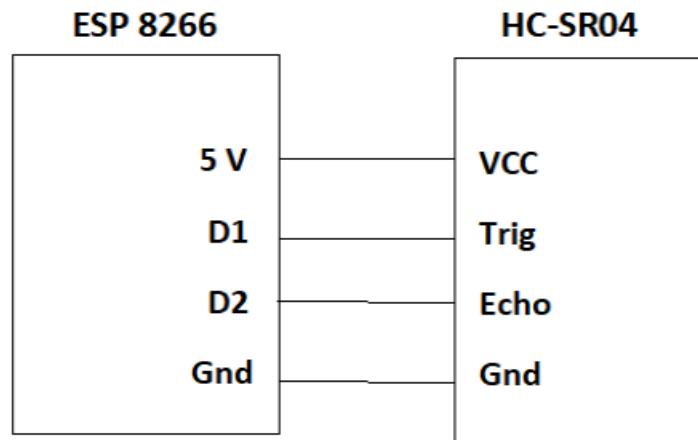
HARDWARE REQUIREMENTS

- ESP8266 module
- Ultrasonic sensor (HC-SR04)
- Resistors
- Breadboard and jumper wires
- Power supply

OBJECTIVES

- Calculating and displaying the distance between the obstacle and the sensor.
- Determining the distance to an object
- Understand the working of Ultrasonic Sensor
- Understand the pin configuration of ESP 8266 and Ultrasonic Sensor
- Understand developing the circuit
- Understand developing code for interfacing ESP 8266 with Ultrasonic sensor

BLOCK DIAGRAM



PROGRAM

```
#define echo 4
#define trig 5
void setup() {
pinMode(trig,OUTPUT) ;
pinMode(echo,INPUT);
Serial.begin(9600);
Serial.println("Object distance calculator");
delay(1000);
}
void loop() {
int distance;
long duration;
digitalWrite(trig,LOW);
delayMicroseconds(2);
digitalWrite(trig,HIGH);
delayMicroseconds(10);
digitalWrite(trig,LOW);
duration=pulseIn(echo,HIGH);
distance=(0.034*duration)/2;
Serial.println("Distance:" +String(distance));
}
```

PROCEDURE

- Connect the HC-SR01 TRIG pin to the ESP8266 D1 pin.
- Connect the HC-SR01 ECHO pin to the ESP8266 D2 pin.
- Connect the HC-SR01 VCC pin to 5V or 3.3V.
- Connect ground to ground.

APPLICATIONS

- **Distance Measurement:** Measure distance in robotic and industrial applications.
- **Level Detection:** Monitor levels in tanks or bins.
- **Obstacle Avoidance:** Used in autonomous robots and vehicles.
- **Parking Assistance:** Guides drivers when parking cars.
- **Security Systems:** Detects movement in secured areas.

ESP8266 INTERFACE WITH PHOTO SENSOR

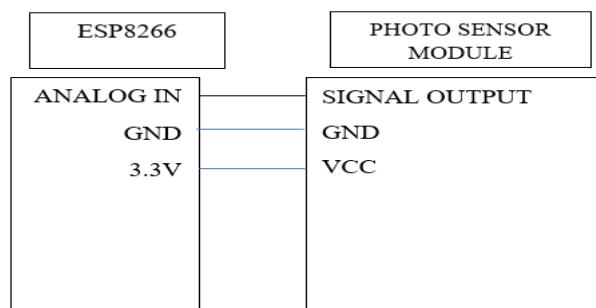
HARDWARE REQUIREMENTS

- ESP8266 module
- Photoresistor (LDR)
- Resistors (as needed)
- Breadboard and jumper wires
- Power supply

OBJECTIVES

- Measuring and displaying the light intensity
- Understanding the working of a photoresistor (LDR)
- Understanding the pin configuration of the ESP8266 and the photoresistor
- Developing code for interfacing ESP8266 with the photoresistor

BLOCK DIAGRAM



PROGRAM

```
const int photoPin = A0; // Define pin for photoresistor (analog pin A0)
```

```

void setup() {
    Serial.begin(115200); // Initialize serial communication at 115200 baud rate
    pinMode(photoPin, INPUT); // Set the photoresistor pin as input
}

void loop() {
    int lightIntensity = analogRead(photoPin); // Read the analog value from the photoresistor
    (0-1023)
    Serial.print("Light Intensity: ");
    Serial.println(lightIntensity); // Print the light intensity value to the serial monitor
    delay(1000); // Wait for 1 second before reading again
}

```

PROCEDURE

- Connect one leg of the photoresistor to the 3.3V pin of the ESP8266.
- Connect the other leg of the photoresistor to a $10\text{k}\Omega$ resistor, and then to GND.
- Connect the junction between the photoresistor and the resistor to the ADC pin (A0) of the ESP8266.
- Ensure all connections are secure and power up the ESP8266.

APPLICATIONS

- **Light Intensity Measurement:** Measure ambient light intensity.
- **Automatic Street Lighting:** Turn street lights on/off based on light conditions.
- **Solar Tracking Systems:** Adjust solar panels to maximize efficiency.
- **Security Systems:** Detect light changes indicating intrusions.
- **Environmental Monitoring:** Monitor daylight exposure for plants.

ESP8266 INTERFACE WITH MAGNETIC SENSORS (FLOATING & DOOR)

HARDWARE REQUIREMENTS

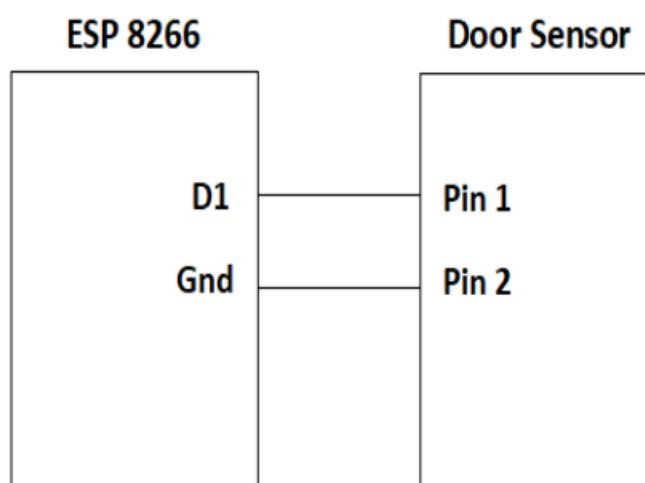
- ESP8266 module
- Magnetic reed switch or hall effect sensor

- Resistors
- Breadboard and jumper wires
- Power supply

OBJECTIVES

- Detect when a door opens or closes: By monitoring the state changes on the ESP8266 input pin, you can detect when a door opens or closes. If the state changes from LOW to HIGH, the door has opened. If the state changes from HIGH to LOW, the door has closed.
- Understand the working of Magnetic Door Sensor
- Understand the pin configuration of ESP 8266 and Magnetic Door Sensor
- Understand developing Electrical circuit
- Understand developing code for interfacing ESP 8266 with Magnetic Door sensor

BLOCK DIGRAM



PROGRAM

```
#define door_sensor 5
```

```

void setup() {
  pinMode(door_sensor,INPUT_PULLUP);
  Serial.begin(9600);
  Serial.println("Home security system");
  delay(1000);
}

void loop() {
  int temp;
  temp = digitalRead(door_sensor);
  Serial.println(temp);
  if (temp == HIGH)
  {
    Serial.println("Door is opened");
  }
  else
  {
    Serial.println("Door is closed");
  }
  delay(2000);
}

```

PROCEDURE

- Connect the Pin 1 pin of the sensor to the D1 pin of the Node MCU.
- Connect the Pin 2 pin of the sensor to the GND pin of the Node MCU.

APPLICATIONS

- **Door Security:** Detects when doors/windows are opened.
- **Position Sensing:** Determine the position of mechanical parts.
- **Proximity Detection:** Used in smart locks and access control systems.
- **Flow Meters:** Measure the flow of liquids using magnetic floats.
- **Industrial Automation:** Detects the status of machinery parts.

ESP8266 INTERFACE WITH RELAY MODULE

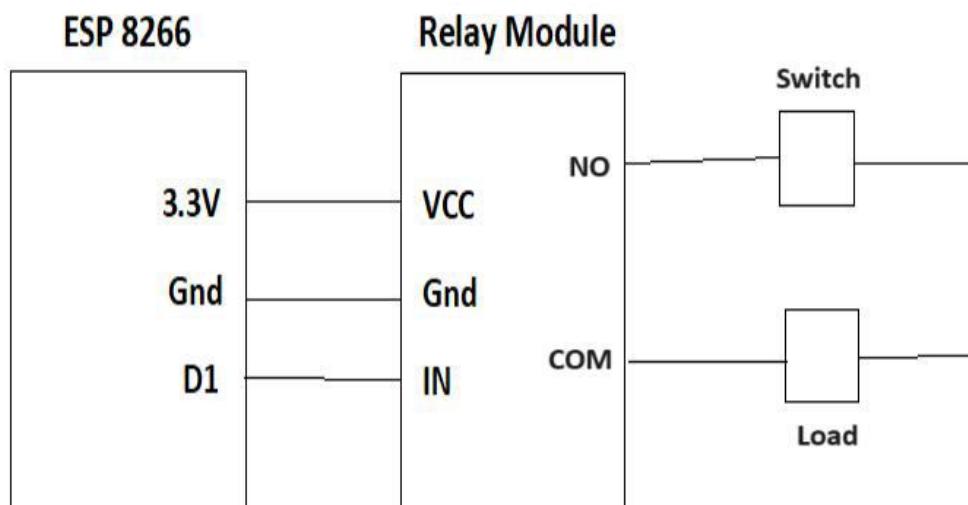
HARDWARE REQUIREMENTS

- ESP8266 module
- Relay module
- Resistors (as needed)
- Breadboard and jumper wires
- Power supply
- Lamp
- Switch

OBJECTIVES

- The ESP8266-based Relay Module can control appliances wirelessly via WiFi.
- It can control up to 250V AC and 30V DC with a load capacity of 10A.
- The module has an Lamp that lights up depending on the module's condition.
- Understand the working of Relay Module
- Understand the pin configuration of ESP 8266 and Relay Module
- Understand developing Electrical circuit
- Understand developing code for interfacing ESP 8266 with Relay Module
- Understand the controlling of 230V AC appliances using NodeMCU and Relay Module.

BLOCK DIAGRAM



PROGRAM

```
#define relay 5
Void setup() {
pinMode(relay,OUTPUT);
}
void loop() {
digitalWrite(relay,LOW);
delay(10000);
digitalWrite(relay, HIGH);
delay(10000);
}
```

PROCEDURE

- 3V3 pin of Node MCU is connected to VCC pin of Relay Module
- Gnd pin of NodeMCU is connected to Gnd pin of Relay Module.
- D1 pin of NodeMCU id connected to Input pin of Relay Module.
- NO of Relay module is connected to one terminal of Switch and Another terminal of Switch is connected to load.
- COM of relay module is connected to another terminal of load.

APPLICATIONS

- **Home Automation:** Control home appliances remotely.
- **Industrial Control:** Control industrial machines and equipment.
- **Automated Lighting:** Control lighting systems based on schedules or sensors.
- **Smart Irrigation:** Control water pumps for irrigation systems.
- **HVAC Control:** Manage heating, ventilation, and air conditioning systems.

ESP8266 INTERFACE WITH TEMPERATURE & HUMIDITY SENSOR (DHT11)

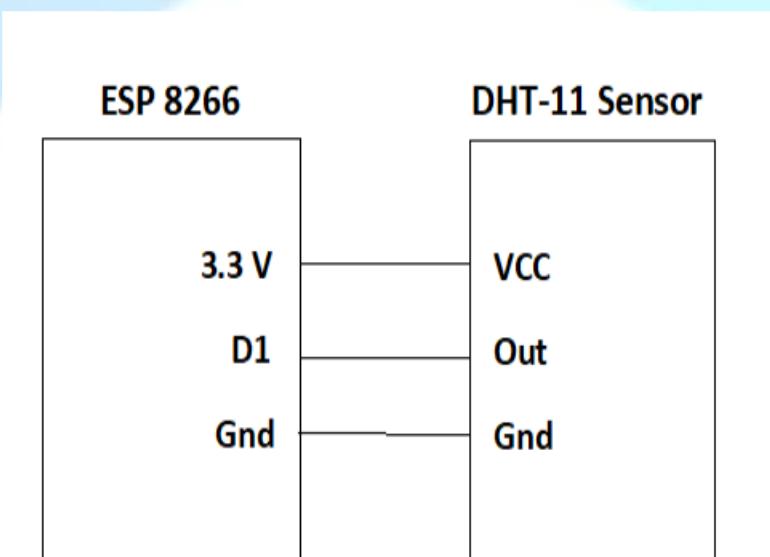
HARDWARE REQUIREMENTS

- ESP8266 module
- DHT11 sensor
- Resistors (as needed)
- Breadboard and jumper wires
- Power supply

OBJECTIVES

- Sensing and displaying the surrounding Humidity and Temperature
- Understand the working of Digital Temperature and Humidity Sensor
- Understand the pin configuration of ESP 8266 and DHT-11 Sensor
- Understand developing Electrical circuit
- Understand developing code for interfacing ESP 8266 with DHT-11 sensor

BLOCK DIAGRAM



PROGRAM

```
#include<DHT.h>
#define DHTPIN 5
#define DHTTYPE DHT11
DHT dht(DHTPIN, DHTTYPE);
void setup() {
Serial.begin(9600);
dht.begin();
Serial.println("DHT-11 Interface with ESP8266");
delay(1000);
}
void loop() {
float temp,humi;
temp=dht.readTemperature();
humi=dht.readHumidity();
Serial.println("Temperature: " +String(temp));
Serial.println("Humidity: " +String(humi));
delay(2000);
}
```

PROCEDURE

- Connect the Vcc pin of the sensor to the 3V3 pin of the Node MCU.
- Connect the GND pin of the sensor to the GND pin of the Node MCU.
- Connect the Output pin of the sensor to the D1 pin of the Node MCU.
- Include DHT.h Library

APPLICATIONS

- **Weather Monitoring:** Track local weather conditions.
- **Smart Thermostats:** Control heating/cooling based on room conditions.
- **Greenhouse Monitoring:** Maintain optimal conditions for plant growth.
- **Data Logging:** Record temperature and humidity over time.
- **Industrial Environment Monitoring:** Ensure safe working conditions.

ESP8266 INTERFACE WITH LCD & I2C MODULE

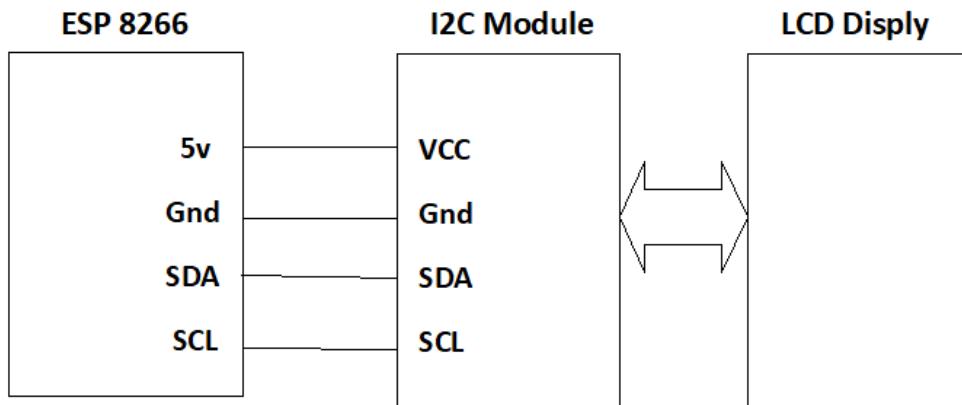
HARDWARE REQUIREMENTS

- ESP8266 module
- LCD display with I2C interface
- Resistors (as needed)
- Breadboard and jumper wires
- Power supply

OBJECTIVES

- Display custom and regular data: The 16x2 LCD display can display any custom as well as regular data in 4-bit mode.
- Display information: 16x2 LCDs are popular and widely used in electronics projects as they are good for displaying information like sensor data from your project.
- Show info for the current state of the sensors and actions: Once a project goes beyond the basic blinking LED, there is always a need for a display to be connected in order to show info for the current state of the sensors and actions.
- Understand the interfacing of I2C Communication Protocol
- Understand the pin configuration of ESP 8266, I2C Module and LCD Display
- Understand developing Electrical circuit
- Understand developing code for interfacing ESP 8266 with LCD Display using I2C Module

BLOCK DIAGRAM



PROGRAM

```

#include <Wire.h>
#include <LiquidCrystal_I2C.h>
LiquidCrystal_I2C lcd(0x27, 16, 2);
void setup()
{
lcd.begin();
lcd.backlight();
lcd.clear();
lcd.setCursor(0, 0);
lcd.print("Hello World");
delay(2000);
}
void loop()
{
lcd.clear();
lcd.setCursor(0, 0);
lcd.print("Hello World");
  
```

PROCEDURE

- **GND** pin of I2C is connected Ground pin (**GND**) of the NodeMCU.

- **VCC** pin of I2C is connected **Vin** pin of the NodeMCU. (Because we need to supply 5v to LCD)
- **SDA** pin of I2C is connected **SDA** of the NodeMCU.
- **SCL** pin of I2C is connected **SCL** pin of the NodeMCU.
- Include Zip Library of LiquidCrystal_I2C.h

APPLICATIONS

- **Information Displays:** Display system information or sensor data.
- **User Interfaces:** Create interactive user interfaces for projects.
- **Data Monitoring:** Real-time display of environmental or process data.
- **Menu Systems:** Provide a navigable menu for selecting options.
- **Clock Displays:** Show time and date information

ESP8266 INTERFACE WITH BLUETOOTH MODULE & ANDROID APP

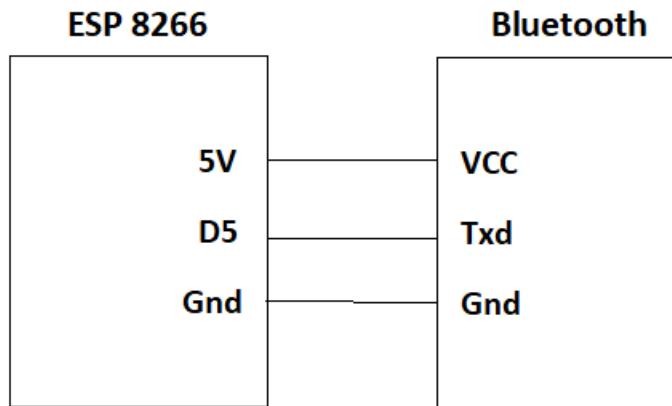
HARDWARE REQUIREMENTS

- ESP8266 module
- Bluetooth module (HC-05 or HC-06)
- Resistors (as needed)
- Breadboard and jumper wires
- Power supply
- Android device with Bluetooth capability

OBJECTIVES

- Bluetooth module can be used in short-distance wireless communication
- Transfer and Receive message between Bluetooth and end device
- Understand the working of Bluetooth Interfacing
- Understand the pin configuration of ESP 8266 with Bluetooth
- Understand Bluetooth UART Communication
- Understand developing code for interfacing ESP 8266 with Bluetooth

BLOCK DIAGRAM



PROGRAM

```

#include<SoftwareSerial.h>

SoftwareSerial bt_serial(14,12); //Rx,Tx
void setup() {
Serial.begin(9600);
bt_serial.begin(9600);
Serial.println("Bluetooth interface with ESP8266");
delay(2000);
}
void loop() {
if(Serial.available())
{
String msg = Serial.readString();
Serial.println(msg);
bt_serial.println(msg);
}
if(bt_serial.available())
{
String msg = bt_serial.readString();
Serial.println(msg);
}
}
  
```

PROCEDURE

- Connect the Vcc pin of the Bluetooth to the Vin pin of the Node MCU.
- Connect the GND pin of the sensor to the GND pin of the Node MCU.

- Connect the Txd pin of the Bluetooth to the D5 pin of the Node MCU.

APPLICATIONS

- **Wireless Data Transfer:** Exchange data between devices.
- **Remote Control:** Control home appliances or robots from a smartphone.
- **Health Monitoring:** Transmit data from health sensors to an app.
- **Proximity-Based Systems:** Trigger actions based on proximity to the smartphone.
- **Smart Home Integration:** Integrate with other smart home devices via Bluetooth.

ESP8266 INTERFACE WITH GSM MODULE

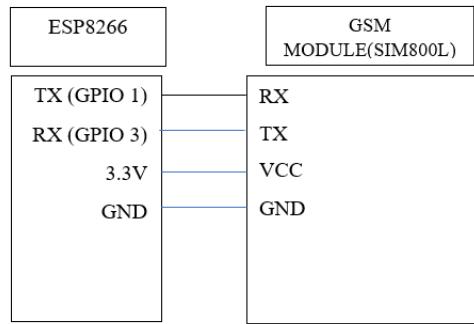
HARDWARE REQUIREMENTS

- ESP8266 module
- GSM module (e.g., SIM800L or similar)
- Resistors (if needed)
- Breadboard and jumper wires
- Power supply
- SIM card (with active data plan if required)

OBJECTIVES

- Send and receive SMS messages using the GSM module.
- Establish GPRS connection to send data to a remote server.
- Make and receive voice calls through the GSM module.
- Understand the working of GSM module.
- Understand the pin configuration and interfacing of ESP8266 with GSM module.
- Develop code for interfacing ESP8266 with GSM module for various applications.

BLOCK DIAGRAM



PROGRAM

```
#include <SoftwareSerial.h>

SoftwareSerial gsm(2, 3); // RX, TX

void setup() {
    Serial.begin(9600);
    gsm.begin(9600);
    Serial.println("Initializing...");
    delay(1000);
    gsm.println("AT"); // Test AT startup
    delay(1000);
    gsm.println("AT+CMGF=1"); // Set SMS mode to text
    delay(1000);
    gsm.println("AT+CNMI=1,2,0,0,0"); // Configure module to send SMS data to serial out
    upon receipt
    delay(1000);
}

void loop() {
    if (gsm.available()) {
        Serial.write(gsm.read());
    }
    if (Serial.available()) {
        gsm.write(Serial.read());
    }
}
```

PROCEDURE

1. Connecting the GSM Module:

- Connect VCC of the GSM module to the 5V pin of the ESP8266.
- Connect GND of the GSM module to the GND pin of the ESP8266.
- Connect RX of the GSM module to TX of the ESP8266 through a voltage divider (if required).
- Connect TX of the GSM module to RX of the ESP8266.

2. Setting Up the GSM Module:

- Insert a SIM card into the GSM module.
- Power the GSM module and wait for it to register on the network.

3. Programming the ESP8266:

- Write the code to initialize the GSM module and set it to SMS text mode.
- Implement functions to send and receive SMS, and optionally make calls.

4. Testing the Setup:

- Upload the code to the ESP8266.
- Open the Serial Monitor and observe the initialization messages.
- Test sending and receiving SMS messages using the Serial Monitor.

APPLICATIONS

- **Remote Monitoring:** Send sensor data via SMS or GPRS.
- **Alert Systems:** Send alerts and notifications based on conditions.
- **Vehicle Tracking:** Track vehicles using GSM-based location data.
- **Remote Control:** Control devices via SMS commands.
- **IoT Projects:** Connect IoT devices to the internet where Wi-Fi is not available.

ESP8266 INTERFACE WITH RFID (MFRC522)

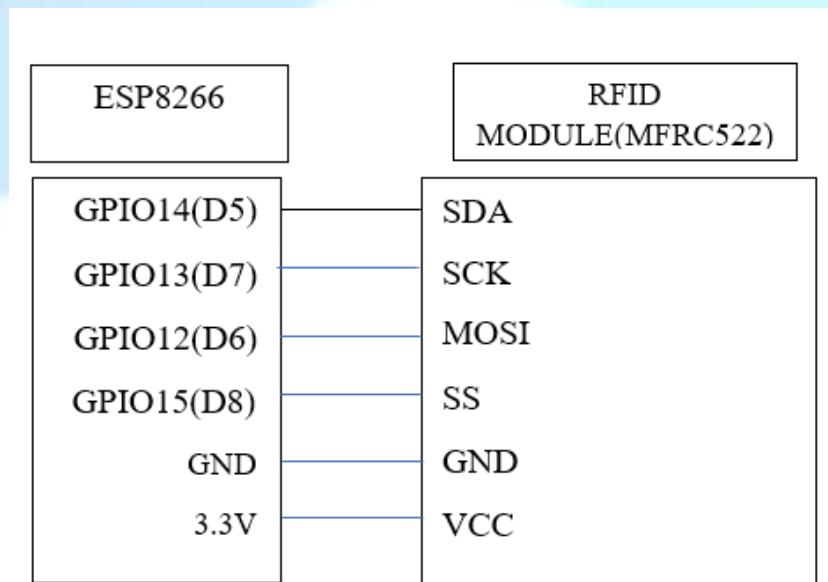
HARDWARE REQUIREMENTS

- ESP8266 module
- RFID reader (MFRC522)
- Resistors
- Breadboard and jumper wires
- Power supply
- RFID tags/cards

OBJECTIVES

- Reading and displaying the unique ID of RFID tags.
- Understanding how to interface the RFID module (MFRC522) with the ESP8266.
- Learning the pin configuration and communication protocol of the RFID module.
- Developing code to read data from RFID tags.
- Implementing an application using RFID tags for access control or tracking.

BLOCK DIAGRAM



PROGRAM

```
#include <SPI.h>
#include <MFRC522.h>
#define SS_PIN D8
#define RST_PIN D3
MFRC522 mfrc522(SS_PIN, RST_PIN); // Create MFRC522 instance
void setup() {
    Serial.begin(115200); // Initialize serial communications with the PC
    SPI.begin(); // Init SPI bus
    mfrc522.PCD_Init(); // Init MFRC522
    Serial.println("Scan an RFID tag");
}
void loop() {
    // Look for new cards
    if (!mfrc522.PICC_IsNewCardPresent()) {
        return;
    }
    // Select one of the cards
    if (!mfrc522.PICC_ReadCardSerial()) {
        return;
    }
    // Dump UID
    Serial.print("UID tag: ");
    String content = "";
    for (byte i = 0; i < mfrc522.uid.size; i++) {
        Serial.print(mfrc522.uid.uidByte[i] < 0x10 ? " 0" : " ");
        Serial.print(mfrc522.uid.uidByte[i], HEX);
        content.concat(String(mfrc522.uid.uidByte[i] < 0x10 ? " 0" : " "));
        content.concat(String(mfrc522.uid.uidByte[i], HEX));
    }
    Serial.println();
    Serial.print("Message: ");
    content.toUpperCase();
}
```

```

if (content.substring(1) == "YOUR_TAG_UID") { // Change YOUR_TAG_UID to your
tag's UID
    Serial.println("Authorized access");
} else {
    Serial.println("Unauthorized access");
}
delay(1000);
}

```

PROCEDURE

➤ **Connections:**

- Connect the VCC of the RFID module (MFRC522) to the 3.3V pin of the ESP8266.
- Connect the GND of the RFID module to the GND pin of the ESP8266.
- Connect the RST of the RFID module to the D3 pin of the ESP8266.
- Connect the MISO of the RFID module to the MISO pin of the ESP8266.
- Connect the MOSI of the RFID module to the MOSI pin of the ESP8266.
- Connect the SCK of the RFID module to the SCK pin of the ESP8266.
- Connect the SDA (SS) of the RFID module to the D8 pin of the ESP8266.

➤ **Setup and Programming:**

- Install the necessary libraries (MFRC522) in the Arduino IDE.
- Upload the provided program to the ESP8266.
- Open the Serial Monitor to see the output.
- Scan an RFID tag to see its unique ID displayed.

APPLICATIONS

1) Access Control:

- Secure entry systems using RFID cards.
- Only authorized personnel can gain access to secured areas.

2) Inventory Management:

- Track items in a warehouse.
- Automatically update inventory records when items are added or removed.

3) Attendance Systems:

- Record attendance in schools or workplaces.
- Automatically log entry and exit times.

4) Payment Systems:

- Implement contactless payment solutions.
- Users can pay using their RFID-enabled cards.

5) Asset Tracking:

- Monitor the location of assets.
- Ensure that valuable items are not moved without authorization.



ARM Controller and Raspberry Pi

Introduction to ARM Controller

ARM (Advanced RISC Machines) Controllers are a family of microcontrollers designed by ARM Holdings, widely used in embedded systems. They are known for their efficiency, low power consumption, and high performance, making them suitable for a range of applications from consumer electronics to industrial automation.

Key Features:

- **RISC Architecture:** ARM controllers use a Reduced Instruction Set Computing (RISC) architecture, which simplifies the instruction set, leading to faster performance and efficient execution.
- **Low Power Consumption:** Ideal for battery-powered and energy-efficient applications.
- **Wide Range of Peripherals:** Support for various communication protocols (SPI, I2C, UART), ADCs, DACs, PWM, etc.
- **Scalability:** Available in various series such as ARM Cortex-M for microcontrollers and Cortex-A for application processors.

Common ARM Controllers:

- **Cortex-M0/M0+:** Ultra-low power, ideal for simple embedded applications.
- **Cortex-M3:** Balanced performance and efficiency for general-purpose applications.
- **Cortex-M4:** Enhanced performance with DSP (Digital Signal Processing) capabilities.
- **Cortex-A series:** High-performance processors for complex applications like smartphones and tablets.

Introduction to Raspberry Pi

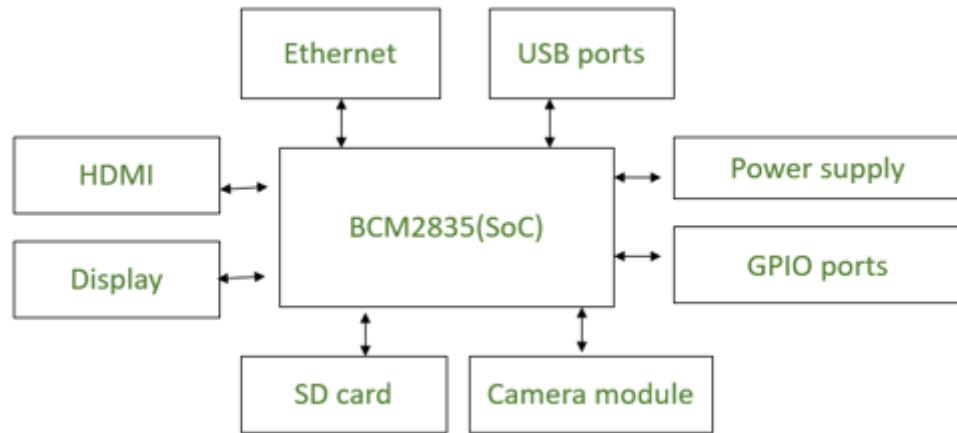
The **Raspberry Pi** is a small, affordable single-board computer developed by the Raspberry Pi Foundation. It is widely used in education, hobbyist projects, and industry due to its versatility and ease of use.

Key Features:

- **Compact Size:** Fits in the palm of your hand, making it ideal for portable and embedded projects.
- **Affordable:** Low cost, making it accessible for educational and hobbyist purposes.
- **General-Purpose Input/Output (GPIO) Pins:** Allow interfacing with various sensors, actuators, and other hardware.
- **Wide Range of Models:** Various models available, such as Raspberry Pi 4, Raspberry Pi 3, and Raspberry Pi Zero, catering to different performance and connectivity needs.

Block Diagram and Specifications of Raspberry Pi

Block Diagram



Raspberry Pi mainly consists of the following blocks:

- **Processor:** Raspberry Pi uses Broadcom BCM2835 system on chip which is an ARM processor and Video core Graphics Processing Unit (GPU). It is the heart of the Raspberry Pi which controls the operations of all the connected devices and handles all the required computations.
- **HDMI:** High Definition Multimedia Interface is used for transmitting video or digital audio data to a computer monitor or to digital TV. This HDMI port helps Raspberry Pi to connect its signals to any digital device such as a monitor digital TV or display through an HDMI cable.
- **GPIO ports:** General Purpose Input Output ports are available on Raspberry Pi which allows the user to interface various I/P devices.
- **Audio output:** An audio connector is available for connecting audio output devices such as headphones and speakers.
- **USB ports:** This is a common port available for various peripherals such as a mouse, keyboard, or any other I/P device. With the help of a USB port, the system can be expanded by connecting more peripherals.
- **SD card:** The SD card slot is available on Raspberry Pi. An SD card with an operating system installed is required for booting the device.
- **Ethernet:** The ethernet connector allows access to the wired network, it is available only on the model B of Raspberry Pi.
- **Power supply:** A micro USB power connector is available onto which a 5V power supply can be connected.
- **Camera module:** Camera Serial Interface (CSI) connects the Broadcom processor to the Pi camera.
- **Display:** Display Serial Interface (DSI) is used for connecting LCD to Raspberry Pi using 15 15-pin ribbon cables. DSI provides a high-resolution display interface that is specifically used for sending video data.

Key Components:

- **CPU (Central Processing Unit):** ARM-based processor (e.g., Cortex-A72 in Raspberry Pi 4).
- **GPU (Graphics Processing Unit):** VideoCore IV for multimedia processing.
- **RAM (Random Access Memory):** Varies by model, e.g., 2GB, 4GB, or 8GB LPDDR4.
- **USB Ports:** For connecting peripherals like keyboards, mice, and storage devices.
- **HDMI Ports:** For video output to monitors or TVs.
- **Ethernet Port:** For wired network connectivity.
- **Wi-Fi and Bluetooth:** For wireless communication.
- **MicroSD Card Slot:** For storage and operating system.
- **GPIO Pins:** For interfacing with external hardware.

Specifications of Raspberry Pi 4 Model B

- **Processor:** Quad-core Cortex-A72 (ARM v8) 64-bit SoC @ 1.5GHz
- **Memory:** 2GB, 4GB, or 8GB LPDDR4-3200 SDRAM
- **Networking:** Gigabit Ethernet, 802.11ac wireless, Bluetooth 5.0
- **USB Ports:** 2 x USB 3.0, 2 x USB 2.0
- **Video Output:** 2 x micro-HDMI ports, supporting up to 4K resolution
- **Storage:** MicroSD card slot
- **GPIO:** 40-pin GPIO header

Raspbian Operating System Installation and Configuration

Raspbian is the official operating system for Raspberry Pi, based on Debian Linux. It is optimized for the Raspberry Pi hardware, providing a robust and user-friendly environment for development and use.

Installation Steps:

1. **Download Raspbian:**
 - Visit the official Raspberry Pi website and download the latest Raspbian image.
 - Download Link
2. **Write Image to MicroSD Card:**
 - Use an image writing tool like **balenaEtcher** or **Raspberry Pi Imager**.
 - Insert the microSD card into your computer.
 - Select the Raspbian image and the microSD card, then click "Write."
3. **Insert MicroSD Card into Raspberry Pi:**
 - Insert the written microSD card into the microSD slot on the Raspberry Pi.
4. **Power Up:**
 - Connect peripherals (keyboard, mouse, monitor) and power the Raspberry Pi using a suitable power supply.
 - The Raspberry Pi will boot into Raspbian.

Configuration Steps:

1. **Initial Setup:**

- On first boot, the Raspberry Pi Configuration tool will guide you through the initial setup, including setting the country, language, time zone, and creating a user account.

2. Update System:

- Open the terminal and update the system with the following commands:
`sudo apt update`
`sudo apt upgrade`

3. Configure Wi-Fi:

- Use the Raspberry Pi Configuration tool or manually edit the `/etc/wpa_supplicant/wpa_supplicant.conf` file to configure Wi-Fi.

Terminal Commands

The terminal in Raspbian allows you to interact with the system using command-line interface (CLI) commands.

Common Terminal Commands:

- **Update Package Lists:**

```
sudo apt update
```

- **Upgrade Installed Packages:**

```
sudo apt upgrade
```

- **Install Software:**

```
sudo apt install <package-name>
```

- **File Operations:**

- **List Files:** `ls`
- **Change Directory:** `cd <directory>`
- **Copy Files:** `cp <source> <destination>`
- **Move Files:** `mv <source> <destination>`
- **Remove Files:** `rm <file>`

Introduction to Thonny Python IDE

Thonny is an Integrated Development Environment (IDE) designed for beginners in Python programming. It is user-friendly and comes pre-installed with Raspbian.

Features:

- **Simple Interface:** Easy to navigate, making it ideal for beginners.
- **Syntax Highlighting:** Helps in identifying different parts of code.
- **Step-by-Step Execution:** Allows you to debug code by running it step by step.
- **Variable Explorer:** Displays variables and their values, aiding in debugging and understanding code flow.

Getting Started with Thonny:

1. **Open Thonny:**

- Find Thonny in the main menu under Programming.

2. Writing Code:

- Write Python code in the editor window.
- Example:

```
print("Hello, Raspberry Pi!")
```

3. Running Code:

- Click the "Run" button or press F5 to execute the code.

Project Creation and Python Library Installation

Creating a Project:

1. New Project:

- Open Thonny and create a new file (File > New).
- Save the file with a .py extension (File > Save As).

2. Write Code:

- Start writing the code for your project.
- Example:

```
import RPi.GPIO as GPIO
import time

GPIO.setmode(GPIO.BCM)
GPIO.setup(18, GPIO.OUT)

while True:
    GPIO.output(18, GPIO.HIGH)
    time.sleep(1)
    GPIO.output(18, GPIO.LOW)
    time.sleep(1)
```

Installing Python Libraries:

1. Using pip:

- Install additional libraries using the pip package manager.
- Open the terminal and type:

```
pip install <library-name>
```

- Example: Installing the RPi.GPIO library:

```
pip install RPi.GPIO
```

Programming with Python on Raspberry Pi

Basic Python Programming:

• Variables and Data Types:

- a) Example:
 - (a) `x = 10`
 - (b) `y = 20.5`
 - (c) `name = "Raspberry Pi"`

• Control Structures:

- a) If-Else:
 - (a) `if x > y:`

```

        (i) print("x is greater")
(b) else:
        (i) print("y is greater")
    
```

- **Loops:**

- i) **For Loop:**

1. for i in range(5):
- a. print(i)
- b)
1. count = 0
2. while count < 5:
3. print(count)
4. count += 1

- **Functions:**

- a) Defining and calling functions:

- (a) def greet(name):
- (i) print("Hello, " + name)
- (b) greet("Raspberry Pi")

- **Interfacing with GPIO:**

- a) Example to blink an LED:

```

        (a) import RPi.GPIO as GPIO
        (b) import time
        (c) GPIO.setmode(GPIO.BCM)
        (d) GPIO.setup(18, GPIO.OUT)

try:
    while True:
        GPIO.output(18, GPIO.HIGH)
        time.sleep(1)
        GPIO.output(18, GPIO.LOW)
        time.sleep(1)
except KeyboardInterrupt:
    GPIO.cleanup()
    
```

Raspberry Pi

Raspberry Pi, developed by Raspberry Pi Foundation in association with Broadcom, is a series of small single-board computers and perhaps the most inspiring computer available today.

From the moment you see the shiny green circuit board of Raspberry Pi, it invites you to tinker with it, play with it, start programming, and create your own software with it. Earlier, the Raspberry Pi was used to teach basic computer science in schools but later, because of its low cost and open design, the model became far more popular than anticipated.

It is widely used to make gaming devices, fitness gadgets, weather stations, and much more. But apart from that, it is used by thousands of people of all ages who want to take their first step in computer science.

Raspberry Pi offers several advantages that contribute to its popularity and versatility in a wide range of applications

Key characteristics of Raspberry Pi include:

1. Single-Board Design:

Raspberry Pi is a credit card-sized computer with all major components, including the CPU, RAM, and I/O ports, integrated onto a single printed circuit board (PCB).

2. Broadcom SoC (System on a Chip):

Raspberry Pi boards are powered by Broadcom SoCs, which include a CPU, GPU, RAM, and other components in a compact package.

3. General-Purpose Input/Output (GPIO) Pins:

Raspberry Pi boards feature GPIO pins that allow users to interface with and control external devices such as sensors, LEDs, motors, and more.

4. Linux-Based Operating System Support:

Raspberry Pi commonly runs a variety of Linux-based operating systems, with Raspbian (now known as Raspberry Pi OS) being the official and recommended distribution. Other operating systems, including various Linux distributions and Windows IoT, are also compatible.

5. Affordability:

Raspberry Pi boards are designed to be affordable, making them accessible to a wide range of users, including students, hobbyists, and professionals.

6. Community and Ecosystem:

Raspberry Pi has a vibrant and active community of users and developers. The ecosystem includes a wide range of accessories, add-on boards (HATs), and software resources.

7. Educational and Hobbyist Use:

Raspberry Pi is widely used in educational settings to teach programming, electronics, and computer science. It is also a popular choice for hobbyist projects, DIY electronics, home automation, media centers, and more.

8. Various Models:

Over the years, several models of Raspberry Pi have been released, each with improvements in terms of hardware specifications and features. Some notable models include Raspberry Pi 4 Model B, Raspberry Pi 3 Model B+, Raspberry Pi Zero, and more.

Raspberry Pi has found applications in a diverse range of projects, from simple programming exercises to complex IoT (Internet of Things) applications. Its versatility, affordability, and community support have contributed to its popularity in various fields.

Architecture of Raspberry Pi

Raspberry Pi is a small single-board computer (SBC). It is a credit card-sized computer that can be plugged into a monitor. It acts as a minicomputer by connecting the keyboard, mouse, and display. Raspberry Pi has an ARM processor and 512MB of RAM. The following diagram shows the architecture of Raspberry Pi:

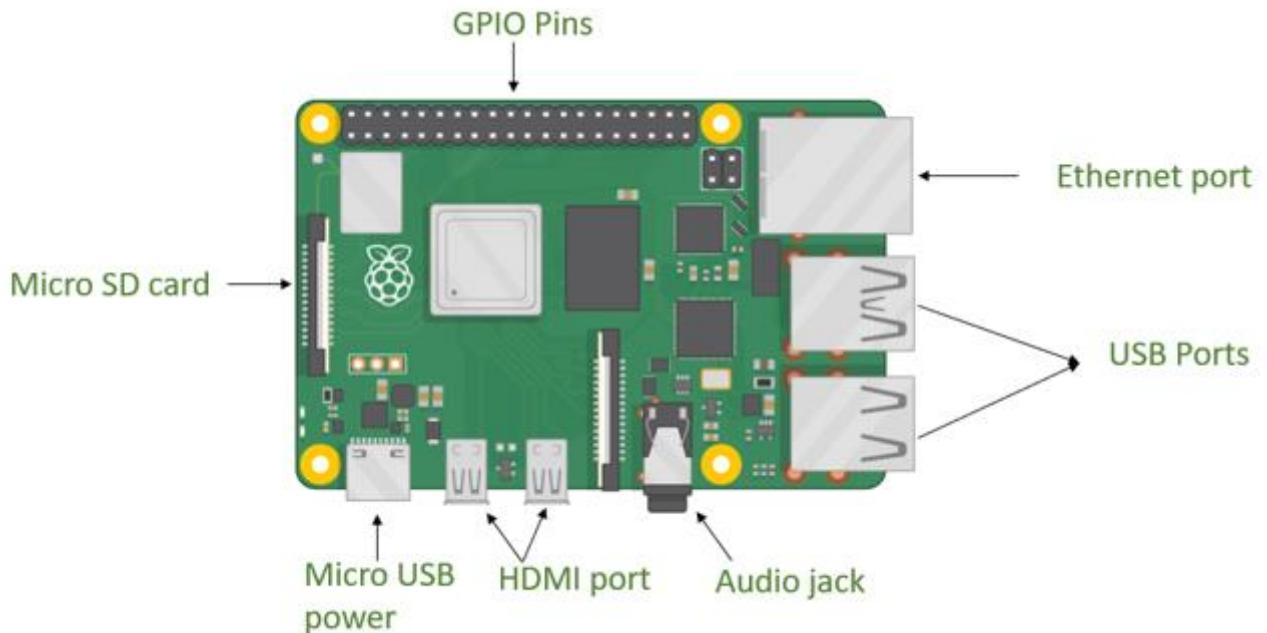


Fig.: Raspberry pi

Fig.: Main blocks of Raspberry pi

Supporting languages of raspberry pi

Raspberry Pi supports a variety of programming languages, offering flexibility for developers and users with different coding preferences. Here are some of the programming languages commonly used on Raspberry Pi:

1. Python:

Python is the most widely used and recommended programming language for Raspberry Pi. It comes pre-installed on Raspbian (now known as Raspberry Pi OS) and is well-supported with a large community. Python is suitable for a broad range of applications, from simple scripts to complex projects.

2. C:

C is a powerful and low-level programming language that can be used for Raspberry Pi development. It is commonly employed when performance optimization or hardware-level access is necessary. Developers can use tools like GCC for compiling C programs on Raspberry Pi.

3. C++:

C++ is an extension of the C programming language and can be used on Raspberry Pi for object-oriented programming. Like C, it is suitable for applications where performance is critical.

4. Java:

Java is a versatile, object-oriented language supported on Raspberry Pi. It is often used for applications where platform independence is essential. The Java Virtual Machine (JVM) is available for Raspberry Pi.

5. JavaScript:

JavaScript is commonly used for web development, and Node.js allows developers to run JavaScript on the Raspberry Pi. This is useful for building web-based applications and IoT projects.

6. Scratch:

Scratch is a visual programming language designed for beginners, and it comes pre-installed on Raspberry Pi OS. It allows users to create programs by stacking code blocks, making it an excellent tool for educational purposes.

7. Ruby:

Ruby is a dynamic, object-oriented programming language that is supported on Raspberry Pi. It is known for its simplicity and readability.

8. PHP:

PHP, a server-side scripting language commonly used in web development, can also be utilized on Raspberry Pi for server-related projects.

9. Shell Scripting (Bash):

Shell scripting, using languages like Bash, allows users to create scripts to automate tasks and manage the Raspberry Pi from the command line.

10. Go (Golang):

Go is a statically typed language developed by Google. It is suitable for building efficient and concurrent programs, and it is supported on Raspberry Pi.

11. Rust:

Rust is a systems programming language that emphasizes safety and performance. It is available on Raspberry Pi and can be used for applications where low-level control is necessary.

12. Perl:

Perl is a versatile scripting language that can be used for various tasks on Raspberry Pi.

13. Lisp, Haskell, and Other Languages:

Raspberry Pi's Linux-based environment allows for the installation and use of a wide range of programming languages, including Lisp, Haskell, and others.

The wide range of supported programming languages makes Raspberry Pi a flexible platform suitable for different development needs and preferences. Depending on the application, developers can choose the language that best fits their requirements and expertise.

Raspberry Pi GPIO connectors:

The Raspberry Pi models have GPIO (General Purpose Input/Output) pins that allow users to interface with external devices such as sensors, LEDs, buttons, and more. The GPIO pins may have different capabilities and functionalities depending on the specific Raspberry Pi model.

GPIO Pinout

One of the powerful features of the Raspberry Pi is the row of GPIO (general-purpose input output) pins and the GPIO Pinout is an interactive reference to these GPIO pins.

Following diagram shows a 40-pin GPIO header, which is found on all the current Raspberry Pi boards

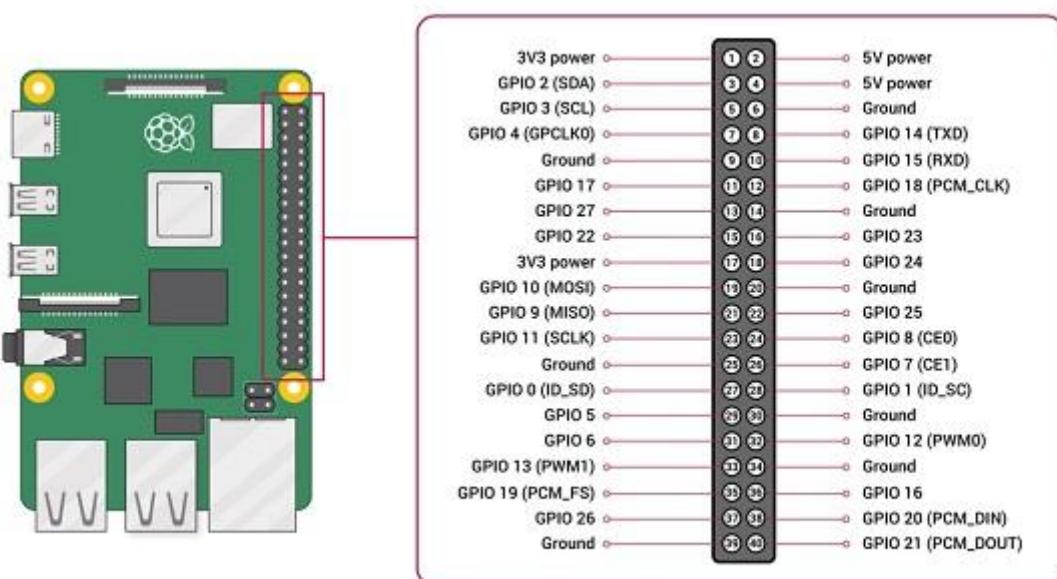


Fig.: 40-pin GPIO header

Raspberry Pi 3 Model B / Raspberry Pi 3 Model B+ / Raspberry Pi 4 Model B:

These models typically follow the Broadcom numbering convention (BCM).

Raspberry Pi 3 GPIO Pinout

GPIO_GEN	Functions	GPIO	Pin	Pin	GPIO	Functions	GPIO_GEN
		3.3V	1	2	5V		
	SDA1 (I ² C)	GPIO2	3	4	5V		
	SCL1 (I ² C)	GPIO3	5	6	GND		
GCLK		GPIO4	7	8	GPIO14	TXD0 (UART)	
		GND	9	10	GPIO15	RXD0 (UART)	
GEN0		GPIO17	11	12	GPIO18	PWM0, CLK (PCM)	GEN1
GEN2		GPIO27	13	14	GND		
GEN3		GPIO22	15	16	GPIO23		GEN4
		3.3V	17	18	GPIO24		GEN5
	MOSI (SPI)	GPIO10	19	20	GND		
	MISO (SPI)	GPIO9	21	22	GPIO25		GEN6
	SCLK (SPI)	GPIO11	23	24	GPIO8	CE0 (SPI)	
		GND	25	26	GPIO7	CE1 (SPI)	
		ID_SD	27	28	ID_SC		
		GPIO5	29	30	GND		
		GPIO6	31	32	GPIO12	PWM0	
	PWM1	GPIO13	33	34	GND		
	FS (PCM), PWM1	GPIO19	35	36	GPIO16		
		GPIO26	37	38	GPIO20	DIN (PCM)	
		GND	39	40	GPIO21	DOUT (PCM)	

Pin Number	BCM GPIO	Physical Pin	Function
3.3V	-	1	Power 3.3V
5V	-	2	Power 5V
GPIO2	2	3	SDA1 (I ² C)
5V	-	4	Power 5V

Pin Number	BCM GPIO	Physical Pin	Function
GPIO3	3	5	SCL1 (I2C)
GND	-	6	Ground
GPIO4	4	7	GPIO
GPIO14	14	8	TXD0 (UART)
GND	-	9	Ground
GPIO15	15	10	RXD0 (UART)
GPIO17	17	11	GPIO
GPIO18	18	12	GPIO, PWM
GPIO27	27	13	GPIO
GND	-	14	Ground
GPIO22	22	15	GPIO
GPIO23	23	16	GPIO
3.3V	-	17	Power 3.3V
GPIO24	24	18	GPIO
GPIO10	10	19	SPI_MOSI
GND	-	20	Ground
GPIO9	9	21	SPI_MISO
GPIO25	25	22	GPIO
GPIO11	11	23	SPI_CLK
GPIO8	8	24	SPI_CE0_N
GND	-	25	Ground
GPIO7	7	26	SPI_CE1_N

Pin Number	BCM GPIO	Physical Pin	Function
ID_SD	-	27	I2C ID EEPROM (Raspberry Pi 4 only)
ID_SC	-	28	I2C ID EEPROM (Raspberry Pi 4 only)
GPIO5	5	29	GPIO
GND	-	30	Ground
GPIO6	6	31	GPIO
GPIO12	12	32	GPIO
GPIO13	13	33	GPIO
GND	-	34	Ground
GPIO19	19	35	GPIO
GPIO16	16	36	GPIO
GPIO26	26	37	GPIO
GPIO20	20	38	GPIO
GND	-	39	Ground
GPIO21	21	40	GPIO

Raspberry Pi Zero and Zero W:

The Raspberry Pi Zero models have a smaller number of GPIO pins.

Pin Number	BCM GPIO	Physical Pin	Function
GND	-	6	Ground
GPIO0	17	11	GPIO
GPIO1	18	12	GPIO
GPIO2	27	13	GPIO
GND	-	14	Ground
GPIO3	22	15	GPIO
GPIO4	23	16	GPIO
3.3V	-	17	Power 3.3V
GPIO17	17	18	GPIO
GPIO27	27	22	GPIO
GPIO22	22	23	GPIO
3.3V	-	1	Power 3.3V
GPIO10	10	24	SPI_MOSI
GPIO9	9	21	SPI_MISO
GPIO11	11	23	SPI_CLK
GPIO8	8	24	SPI_CE0_N
GPIO7	7	26	SPI_CE1_N
GND	-	30	Ground

GPIO Setmodes:

The `GPIO.setmode()` function in the `RPi.GPIO` library for Raspberry Pi is used to set the numbering mode for the GPIO pins. It determines how you refer to the pins in your code. There are two available modes: `GPIO.BCM` and `GPIO.BOARD`. These conventions refer to different ways of numbering the GPIO pins on the Raspberry Pi.

1. BCM (Broadcom) Mode:

Syntax: `GPIO.setmode(GPIO.BCM)`

In BCM mode, GPIO pins are identified by their Broadcom SOC channel numbers. These numbers correspond to the physical pin numbers on the Broadcom SOC (System on a Chip) used in Raspberry Pi. This mode provides a consistent way of identifying pins across different Raspberry Pi models.

Example:

```
import RPi.GPIO as GPIO  
  
GPIO.setmode(GPIO.BCM)  
  
GPIO.setup(18, GPIO.OUT) # BCM GPIO 18 (physical pin 12) configured as output
```

BCM Mode is preferred for consistency across different Raspberry Pi models. Especially useful when migrating code between different Pi models, as the GPIO numbering remains the same.

2. BOARD Mode:

Syntax: GPIO.setmode(GPIO.BOARD)

In BOARD mode, GPIO pins are identified by their physical pin numbers on the Raspberry Pi header. This mode allows you to refer to the pins by their physical locations, making it easier for beginners to understand which pin is being used.

Example:

```
import RPi.GPIO as GPIO  
  
GPIO.setmode(GPIO.BOARD)  
  
GPIO.setup(12, GPIO.OUT) # Physical pin 12 (BCM GPIO 18) configured as output
```

BOARD Mode is Easier for beginners to understand, as it refers to the physical layout of the GPIO pins on the Raspberry Pi header. Board mode is suitable for projects where you are referencing pins based on their physical location on the Pi.

The choice between BCM and BOARD mode depends on personal preference, project requirements, and whether you want consistency across Raspberry Pi models or prefer a more straightforward physical pin reference.

Regardless of the mode used, the actual functionality and capabilities of the GPIO pins remain the same. The only difference is in how you reference and identify the pins in your code.

When using GPIO.setup() or GPIO.cleanup(), the chosen mode affects how you specify the GPIO pin numbers. Always be consistent in using the same mode throughout your code.

Operating Modes Of Raspberry Pi

In Raspberry Pi, the GPIO (General Purpose Input/Output) pins can be configured for different modes depending on how you want to use them. The main modes include Input, Output, and special modes like PWM (Pulse Width Modulation), I2C, SPI, etc.

Here are the common GPIO pin modes used in Raspberry Pi:

1. Input Mode:

Mode Name: GPIO.IN or GPIO.INPUT

Configures the GPIO pin as an input, allowing you to read digital signals (HIGH or LOW) from external devices like buttons, sensors, etc.

Example:

```
import RPi.GPIO as GPIO  
  
GPIO.setmode(GPIO.BCM)  
  
GPIO.setup(GPIO_PIN, GPIO.IN)  
  
input_state = GPIO.input(GPIO_PIN)
```

2. Output Mode:

Mode Name: GPIO.OUT or GPIO.OUTPUT

Configures the GPIO pin as an output, allowing you to send digital signals (HIGH or LOW) to control external devices like LEDs, relays, etc.

Example:

```
import RPi.GPIO as GPIO  
  
GPIO.setmode(GPIO.BCM)  
  
GPIO.setup(GPIO_PIN, GPIO.OUT)  
  
GPIO.output(GPIO_PIN, GPIO.HIGH) # Turn ON  
  
GPIO.output(GPIO_PIN, GPIO.LOW) # Turn OFF
```

3. PWM Mode:

Mode Name: GPIO.PWM

Configures the GPIO pin for Pulse Width Modulation, allowing you to control the brightness of LEDs, the speed of motors, etc.

Example:

```
import RPi.GPIO as GPIO
import time
GPIO.setmode(GPIO.BCM)
GPIO.setup(GPIO_PIN, GPIO.OUT)
pwm = GPIO.PWM(GPIO_PIN, 100) # 100 Hz frequency
pwm.start(50) # 50% duty cycle (0 to 100)
time.sleep(5) # Run for 5 seconds
pwm.stop()
```

4. I2C Mode:

Mode Name: GPIO.I2C

Configures the GPIO pins for I2C (Inter-Integrated Circuit) communication, allowing you to connect to I2C devices.

Example:

```
import smbus
i2c = smbus.SMBus(1) # Use the appropriate bus number
# Perform I2C operations using the smbus library
```

5. SPI Mode:

Mode Name: GPIO.SPI

Configures the GPIO pins for SPI (Serial Peripheral Interface) communication, enabling communication with SPI devices.

Example:

```
import spidev
spi = spidev.SpiDev()
spi.open(0, 0) # Bus 0, Device 0
```

```
# Perform SPI operations using the spidev library
```

6. Serial (UART) Mode:

Mode Name: GPIO.SERIAL or GPIO.UART

Configures the GPIO pins for serial communication (UART), allowing you to communicate with other serial devices.

Example:

```
import serial

ser = serial.Serial("/dev/serial0", baudrate=9600, timeout=1)

# Perform serial communication using the serial library
```

7. Hardware PWM Mode (Raspberry Pi 2 and 3):

Mode Name: GPIO.HARD_PWM

Configures the GPIO pin for hardware PWM. It is similar to PWM but uses hardware support for better accuracy and performance.

8. Default Mode:

Mode Name: GPIO.IN or GPIO.INPUT

In the absence of explicit mode configuration, the GPIO pins are usually configured as inputs by default.

Note:

Always import the RPi.GPIO module and set the GPIO mode using GPIO.setmode() before configuring pins.

Ensure proper cleanup using GPIO.cleanup() when your program exits to release the resources used by GPIO pins.

Libraries or modules of Raspberry Pi

The libraries or modules you need for Raspberry Pi programming depend on the specific tasks or projects you're working on. However, there are some common libraries that are frequently used in Raspberry Pi programming across various applications. Here are some essential libraries:

- **RPi.GPIO:**

- Purpose: Provides access to the GPIO pins on the Raspberry Pi.
- Installation: Most Raspberry Pi OS distributions come with this library pre-installed. If not, you can install it using:
 - i) sudo apt-get update
 - ii) sudo apt-get install python3-rpi.gpio

- **smbus-cffi (for I2C):**

- Purpose: Allows communication with I2C devices.
- Installation:
 - i) sudo apt-get update
 - ii) sudo apt-get install python3-smbus

- **spidev (for SPI):**

- Purpose: Enables communication with SPI devices.
- Installation:
 - i) sudo apt-get update
 - ii) sudo apt-get install python3-spidev

- **serial (for UART):**

- Purpose: Provides serial communication capabilities.
- Installation:
 - i) sudo apt-get update
 - ii) sudo apt-get install python3-serial

1. time:

Purpose: Standard Python library for handling time-related functions.

2. datetime:

Purpose: Standard Python library for working with dates and times.

3. math:

Purpose: Standard Python library for mathematical operations.

4. subprocess:

Purpose: Standard Python library for running shell commands.

5. os:

Purpose: Standard Python library for interacting with the operating system.

6. requests:

Purpose: Useful for making HTTP requests. Useful in projects involving internet connectivity.

Installation:

```
pip3 install requests
```

7. picamera (for Camera Module):

Purpose: Interface for the Raspberry Pi Camera Module.

Installation:

```
sudo apt-get update
```

```
sudo apt-get install python3-picamera
```

8. pygame (for Multimedia Applications):

Purpose: Useful for creating games or multimedia applications.

Installation:

```
sudo apt-get update
```

```
sudo apt-get install python3-pygame
```

9. cv2 (OpenCV for Computer Vision):

Purpose: Useful for computer vision tasks.

Installation:

```
pip3 install opencv-python
```

These libraries cover a range of functionalities, from GPIO access to communication protocols (I2C, SPI, UART), timing, multimedia, and computer vision. Depending on your project requirements, you might need additional specialized libraries. Always check the documentation of the specific sensors or components you are using for any required libraries or drivers.

Text Editors and Integrated Development Environments (IDEs) for Programming on Raspberry Pi

There are several Integrated Development Environments (IDEs) and text editors available for Raspberry Pi programming, catering to different preferences and programming languages. Here are some popular ones:

1. Thonny:

Thonny is a beginner-friendly IDE that comes pre-installed with many Raspberry Pi OS distributions. It is designed to make Python programming easy for beginners.

Installation:

```
sudo apt-get update
```

```
sudo apt-get install thonny
```

2. IDLE:

IDLE (Integrated Development and Learning Environment) is another Python-specific IDE that comes pre-installed with Python. It's lightweight and suitable for Python development.

Installation:

IDLE is included with the Python installation on Raspberry Pi.

3. Geany:

Geany is a lightweight text editor with IDE features. It supports multiple programming languages, including Python.

Installation:

```
sudo apt-get update
```

```
sudo apt-get install geany
```

4. Visual Studio Code (VSCode):

VSCode is a powerful and popular open-source code editor developed by Microsoft. It supports various programming languages, and there are extensions available for Raspberry Pi development.

Installation:

You can download and install VSCode from the official website or use the following commands:

```
sudo apt-get update
```

```
sudo apt-get install code
```

5. Atom:

Atom is a customizable and feature-rich text editor developed by GitHub. It supports various languages and has a large community that contributes packages and themes.

Installation:

You can download Atom from the official website.

6. Emacs:

Emacs is a highly customizable text editor that can function as an IDE. It has a steep learning curve but offers extensive features for those who become proficient.

Installation:

```
sudo apt-get update
```

```
sudo apt-get install emacs
```

7. Nano:

Nano is a simple and lightweight text editor for the command line. It's easy to use and suitable for quick edits or simple coding tasks.

Installation:

Nano is often pre-installed on Raspberry Pi OS.

Choose an IDE or text editor based on your preferences, programming language, and the complexity of your projects. Many developers prefer using a combination of different tools for different tasks.

Basic program structure of Raspberry Pi programming

When programming for Raspberry Pi, you typically write scripts or programs in languages like Python. The basic program structure is similar to general programming practices. Here's a simple example of a Python program structure for Raspberry Pi:

```
# Import necessary libraries
import RPi.GPIO as GPIO
import time

# Set up GPIO mode (BCM or BOARD)
GPIO.setmode(GPIO.BCM)

# Define GPIO pin numbers
led_pin = 18
button_pin = 17

# Set up GPIO pins
GPIO.setup(led_pin, GPIO.OUT)
GPIO.setup(button_pin, GPIO.IN, pull_up_down=GPIO.PUD_DOWN)

try:
```

```
# Main program loop
```

```
while True:
```

```
    # Read button state
```

```
    button_state = GPIO.input(button_pin)
```

```
    # Perform actions based on button state
```

```
    if button_state == GPIO.HIGH:
```

```
        print("Button pressed!")
```

```
        GPIO.output(led_pin, GPIO.HIGH)
```

```
    else:
```

```
        print("Button released.")
```

```
        GPIO.output(led_pin, GPIO.LOW)
```

```
    # Add other program logic here
```

```
    # Introduce a delay to avoid unnecessary CPU usage
```

```
    time.sleep(0.1)
```

```
except KeyboardInterrupt:
```

```
    # Handle keyboard interrupt (Ctrl+C)
```

```
    print("Program terminated by user.")
```

```
finally:
```

```
    # Clean up GPIO settings on program exit
```

```
    GPIO.cleanup()
```

This example assumes you have a button connected to a GPIO pin (button_pin) and an LED connected to another GPIO pin (led_pin). The program continuously reads the state of the button and turns the LED on when the button is pressed.

Key components of the program structure:

1. Import Libraries:

- Import the necessary libraries at the beginning of your program. In this example, the RPi.GPIO library and the time module are imported.

2. Set Up GPIO:

- Set the GPIO mode (BCM or BOARD) using `GPIO.setmode()`.
- Define GPIO pin numbers that you will be using.

3. Set Up GPIO Pins:

- Use `GPIO.setup()` to configure GPIO pins as inputs or outputs.
- Optionally, set pull-up or pull-down resistors for input pins.

4. Main Program Loop:

- Use a while True loop to create the main program loop.
- Read input from sensors, buttons, etc.
- Perform actions based on input.
- Add other program logic as needed.

5. Exception Handling:

- Use a try, except, finally block for proper exception handling.
- The except KeyboardInterrupt block handles the case when the user interrupts the program with Ctrl+C.

6. Clean Up GPIO:

- In the finally block, use `GPIO.cleanup()` to release the resources used by GPIO pins when the program exits.

This is a basic structure, and you can expand and modify it based on your specific project requirements. Always refer to the documentation of the libraries you are using and follow best practices for programming on the Raspberry Pi.

while loop in Raspberry Pi

In the context of Raspberry Pi programming, the while loop is commonly used to create continuous or repetitive tasks. Here are some reasons why while loops are important and commonly employed in Raspberry Pi programming:

1. Continuous Monitoring:

Raspberry Pi often interfaces with sensors, buttons, or external events. A while loop allows continuous monitoring of these inputs, enabling the system to respond in real-time when specific conditions are met.

2. Sensor Readings and Data Collection:

For projects involving data collection from sensors (temperature, humidity, motion, etc.), a while loop is essential. It allows the Raspberry Pi to repeatedly read sensor values and process the data.

3. Event Handling:

Raspberry Pi can be used for event-driven applications. The while loop can continuously check for events or triggers and respond accordingly, whether it's a button press, a change in sensor values, or other input events.

4. User Interaction:

When the Raspberry Pi is running a program that involves user interaction, a while loop can be used to continually check for user input, enabling a responsive and interactive interface.

5. Control of Hardware Components:

In projects involving the control of hardware components (LEDs, motors, relays), a while loop can be used to maintain the state of these components based on the desired logic or conditions.

6. Real-Time Applications:

For real-time applications, such as robotics or automation, a while loop ensures that the system can respond quickly to changing conditions or inputs.

7. Background Tasks:

Some tasks, like background monitoring or maintenance, need to run continuously in the background. A while loop facilitates the creation of such background tasks on the Raspberry Pi.

8. Control of Timing and Delays:

In many scenarios, precise timing is crucial. A while loop can be used in conjunction with functions like `time.sleep()` to introduce delays, control timing, or synchronize activities.

9. State Machine Implementation:

State machines are often used in embedded systems and robotics. A while loop can be used to implement the logic of a state machine, allowing the Raspberry Pi to transition between different states based on conditions.

10. Iterative Processes:

When a process needs to be repeated iteratively until a certain condition is met, a while loop provides a clean and effective way to structure the code.

Advantages of using Raspberry Pi

Raspberry Pi offers several advantages that contribute to its popularity and versatility in a wide range of applications:

1. Affordability:

Raspberry Pi boards are cost-effective, making them accessible for educational purposes, hobbyist projects, and cost-sensitive applications.

2. Compact Size:

The small form factor of Raspberry Pi allows for easy integration into various projects with space constraints, making it suitable for portable and embedded applications.

3. Low Power Consumption:

Raspberry Pi consumes very little power, making it energy-efficient and suitable for applications where power consumption is a critical factor.

4. Community Support:

Raspberry Pi has a large and active community of users and developers. This community support includes forums, documentation, tutorials, and a vast repository of projects, providing assistance and inspiration to users.

5. Versatility:

Raspberry Pi supports a variety of programming languages, making it versatile for a wide range of applications. Python is particularly well-supported and widely used.

6. Educational Tool:

Raspberry Pi is an excellent educational tool for learning programming, electronics, and computer science. It helps bridge the gap between software and hardware.

7. GPIO (General Purpose Input/Output) Pins:

Raspberry Pi's GPIO pins enable interaction with external devices, making it suitable for projects involving sensors, actuators, and other hardware components.

8. Expandability:

Raspberry Pi boards come with USB ports, HDMI, audio output, and camera/display interfaces, providing options for expanding functionality by connecting peripherals and accessories.

9. Open Source Software:

The software ecosystem for Raspberry Pi is based on open-source principles, allowing users to modify and customize software to suit their needs.

10. Rich Set of Accessories:

A wide range of accessories and add-ons are available for Raspberry Pi, including cases, cameras, sensors, and HATs (Hardware Attached on Top), enhancing its capabilities for specific applications.

11. Robust Software Support:

The Raspberry Pi Foundation regularly releases updates and improvements for the Raspberry Pi OS, ensuring a stable and secure operating environment.

12. Media Capabilities:

Raspberry Pi supports multimedia applications and can function as a low-cost media center, providing users with the ability to stream videos, music, and more.

13. Internet of Things (IoT) Applications:

Raspberry Pi's capabilities make it suitable for IoT projects, where it can connect to the internet, collect data from sensors, and interact with other devices.

14. Rapid Prototyping:

Raspberry Pi allows for rapid prototyping of projects, enabling users to quickly test and iterate on ideas before deploying them in a larger-scale environment.

Disadvantages of Raspberry Pi

While Raspberry Pi is a versatile and popular platform, it does have some limitations and potential drawbacks. Here are some of the disadvantages of Raspberry Pi:

1. Limited Processing Power:

Raspberry Pi boards have modest processing power compared to more powerful computers, limiting their performance for resource-intensive tasks.

2. Limited RAM:

The amount of RAM on Raspberry Pi boards is limited, which can be a constraint for memory-intensive applications or large datasets.

3. Limited Graphics Performance:

The graphics capabilities of Raspberry Pi are sufficient for basic tasks, but they may not be suitable for demanding graphical applications or gaming.

4. Limited Storage Options:

Raspberry Pi typically relies on microSD cards for storage. While this is suitable for many applications, it may limit storage capacity and speed compared to traditional hard drives or SSDs.

5. Limited Connectivity:

The number of USB ports on Raspberry Pi boards is limited. This may require the use of USB hubs for projects that involve multiple peripherals.

6. No Real-Time Clock (RTC):

Raspberry Pi lacks a built-in real-time clock, which means it relies on an internet connection to get the current time. This can be a limitation for applications that require accurate timekeeping without internet access.

7. Not Suitable for High-Performance Computing:

Raspberry Pi is not designed for high-performance computing tasks. Applications requiring significant computational power may need more specialized hardware.

8. Limited Availability of GPU Documentation:

While the GPU on Raspberry Pi is powerful, the lack of complete documentation for it can limit the development of certain graphics-intensive applications.

9. Power Requirements:

Raspberry Pi boards have specific power requirements, and using insufficient power sources can lead to instability or malfunctions. Power-related issues can be a concern in certain setups.

10. Limited Audio Output:

The audio output on Raspberry Pi may not be as high-quality as dedicated audio solutions, which can be a consideration for audio-centric projects.

11. Learning Curve for Beginners:

For individuals new to programming and electronics, there may be a learning curve associated with setting up and configuring Raspberry Pi for specific projects.

12. Not Suitable for Real-Time Applications:

Raspberry Pi is not designed for real-time applications. It may not provide the level of precision and timing required for certain critical applications.

13. Single-Board Failure:

If a single-board fails due to hardware issues, the entire system may be affected. Redundancy options are limited.

14. Not Designed for High-End Gaming:

While Raspberry Pi can handle retro gaming and less demanding games, it is not designed for modern, high-end gaming experiences.

Despite these limitations, Raspberry Pi remains an excellent platform for a wide range of applications, and many of these drawbacks can be mitigated or addressed depending on the specific project requirements.

Real World Applications of Raspberry Pi

Raspberry Pi, a versatile and affordable single-board computer, has found applications across various domains due to its compact size, low cost, and low power consumption. Here are some real-world uses and applications of Raspberry Pi:

1. Education:

Teaching Programming: Raspberry Pi is an excellent tool for learning programming and computer science concepts. It helps students understand hardware and software interactions in a practical way.

2. Home Automation:

Smart Home Hub: Raspberry Pi can serve as a central hub for home automation projects, controlling lights, thermostats, cameras, and other smart devices.

3. Media Center:

Kodi: With media center software like Kodi, Raspberry Pi can transform into a low-cost, energy-efficient entertainment hub, streaming music and videos.

4. Network Monitoring:

Pi-hole: Raspberry Pi can run Pi-hole, a network-wide ad blocker. It filters out unwanted ads before they reach your devices, improving network performance.

5. Web Server:

Apache or Nginx: Raspberry Pi can be configured as a web server for hosting small websites or development projects.

6. Security Camera System:

MotionEyeOS: Raspberry Pi can be turned into a security camera system using MotionEyeOS, allowing users to monitor their premises remotely.

7. Weather Station:

Weather Monitoring: Raspberry Pi can collect and analyze weather data using sensors like temperature and humidity sensors, then display the information on a web interface.

8. Gaming Console:

RetroPie: Raspberry Pi can emulate various gaming consoles using RetroPie, enabling users to play classic games from platforms like NES, SNES, and more.

9. Robotics and IoT:

DIY Projects: Raspberry Pi is popular in robotics and Internet of Things (IoT) projects, where it can control sensors, actuators, and communicate with other devices.

10. Desktop Computer:

Basic Computing: Raspberry Pi can function as a basic desktop computer for tasks such as web browsing, word processing, and programming.

11. Network Attached Storage (NAS):

OpenMediaVault: Raspberry Pi can be used to set up a low-cost NAS solution, providing network storage for files and media.

12. Educational Initiatives:

Classroom Projects: Raspberry Pi is used in educational initiatives to engage students in hands-on projects, fostering creativity and problem-solving skills.

13. Telecommunication:

Voice over IP (VoIP): Raspberry Pi can be used to set up VoIP services, enabling cost-effective communication over the internet.

14. AI and Machine Learning:

TensorFlow and OpenCV: Raspberry Pi can run lightweight AI and machine learning models for applications like image recognition and object detection.

These are just a few examples, and the versatility of Raspberry Pi continues to drive innovation in various fields. Its open-source nature and a supportive community contribute to its widespread adoption for both hobbyist and professional projects.



HARDWARE PROGRAMMING (RASPBERRY PI)

RASPBERRY PI INTERFACE WITH ULTRASONIC SENSOR (HC-SR04)

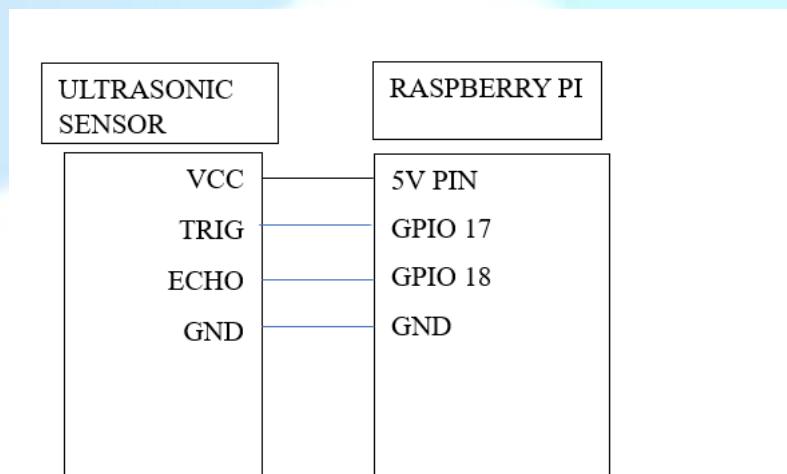
HARDWARE REQUIREMENTS

- Raspberry Pi board
- Ultrasonic Sensor (HC-SR04)
- Breadboard and jumper wires
- 1 kΩ and 2 kΩ resistors (for voltage divider)
- Power supply (5V)

OBJECTIVES

- Calculating and displaying the distance between the obstacle and the sensor
- Determining the distance to an object
- Understand the working of Ultrasonic Sensor
- Understand the pin configuration of Raspberry pi and Ultrasonic sensor
- Understand Installation and writing of OS into Raspberry PI
- Understand developing code for interfacing Raspberry with Ultrasonic sensor

BLOCK DIAGRAM



PROGRAM

```

import RPi.GPIO as GPIO
import time
GPIO.setmode(GPIO.BOARD)
TRIG_PIN = 11
ECHO_PIN = 13
GPIO.setup(TRIG_PIN, GPIO.OUT)
GPIO.setup(ECHO_PIN, GPIO.IN)
def measure_distance():
    GPIO.output(TRIG_PIN, GPIO.HIGH)
    time.sleep(0.00001)
    GPIO.output(TRIG_PIN,
    GPIO.LOW)    while
    GPIO.input(ECHO_PIN) == 0:
        pulse_start_time = time.time()    while
    GPIO.input(ECHO_PIN) == 1:
        pulse_end_time = time.time()
        pulse_duration = pulse_end_time - pulse_start_time
        distance = pulse_duration * 34300 / 2 # Speed of sound is 343
meters/second    return distance try:    while True:
    distance = measure_distance()
    print(f"Distance: {distance:.2f} cm")
    time.sleep(1)
except KeyboardInterrupt:
    GPIO.cleanup()

```

PROCEDURE

- Connect the VCC of the sensor to the pin no 2, 5V of the Raspberry Pi.
- Connect the Trig pin of the sensor to the pin no 11 of the Raspberry Pi.
- Connect the Echo pin of the sensor to the pin no 13 of the Raspberry Pi.
- Connect the Gnd pin of the sensor to the pin no 6, Gnd of the Raspberry Pi.

APPLICATIONS

- Measure the distance of objects for robotics applications.
- Implement obstacle avoidance systems in autonomous vehicles.
- Monitor liquid levels in tanks or reservoirs.
- Create a proximity sensing system for security purposes.
- Design a parking assistance system to aid drivers in tight spaces.

RASPBERRY PI INTERFACE WITH DHT11

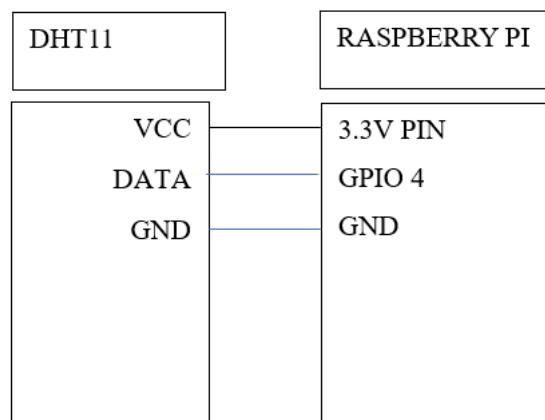
HARDWARE REQUIREMENTS

- Raspberry Pi board
- DHT11 sensor module
- Breadboard and jumper wires
- 10 kΩ resistor

OBJECTIVES

- Measure and display temperature and humidity data.
- Understand the working principles of the DHT11 sensor.
- Learn the pin configuration of the Raspberry Pi and the DHT11 sensor.
- Install and configure the necessary libraries for DHT11 interfacing.
- Develop code to read data from the DHT11 sensor and display it on the console.

BLOCK DIAGRAM



PROGRAM

```

import RPi.GPIO as GPIO
import Adafruit_DHT
import time

DHT_SENSOR = Adafruit_DHT.DHT11
DHT_PIN = 4

GPIO.setmode(GPIO.BCM)

try:
    while True:
        humidity, temperature = Adafruit_DHT.read(DHT_SENSOR, DHT_PIN)
        if humidity is not None and temperature is not None:
            print(f"Temp={temperature:.1f}C Humidity={humidity:.1f}%")
        else:
            print("Failed to retrieve data from humidity sensor")
            time.sleep(2)
except KeyboardInterrupt:
    GPIO.cleanup()

```

PROCEDURE

- Connect the VCC of the DHT11 sensor to the 5V pin of the Raspberry Pi.
- Connect the Data pin of the DHT11 sensor to the GPIO4 pin of the Raspberry Pi.
- Connect the GND pin of the DHT11 sensor to the GND pin of the Raspberry Pi.

APPLICATIONS

- Monitor temperature and humidity in a home automation system.
- Build a weather station to collect environmental data.
- Automate greenhouse climate control based on sensor readings.
- Track indoor climate conditions for HVAC systems.
- Develop a data logging system for scientific experiments.

RASPBERRY PI INTERFACE WITH I2C & LCD MODULE

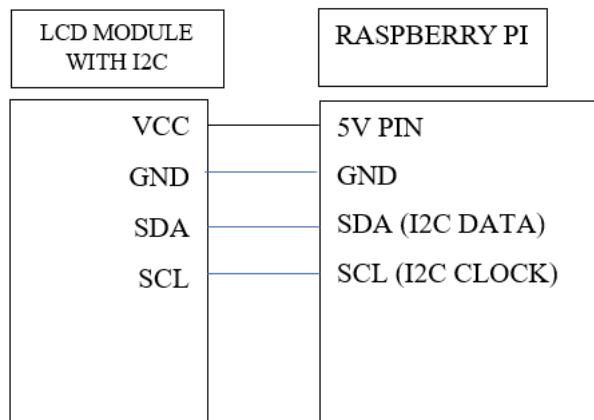
HARDWARE REQUIREMENTS

- Raspberry Pi board (any model)
- I2C LCD module (e.g., 16x2 or 20x4)
- Breadboard and jumper wires
- I2C cable (if needed)

OBJECTIVES

- Display real-time sensor data on an LCD screen.
- Understand the working principles of I2C communication.
- Learn the pin configuration of the Raspberry Pi and the I2C LCD module.
- Install and configure the necessary libraries for I2C communication.
- Develop code to interface the Raspberry Pi with the I2C LCD module.

BLOCK DIAGRAM



PROGRAM

```

import smbus2
import time

# Define some device parameters
I2C_ADDR = 0x27 # I2C device address
LCD_WIDTH = 16 # Maximum characters per line
  
```

```

# Define some device constants

LCD_CHR = 1      # Mode - Sending data
LCD_CMD = 0      # Mode - Sending command

LCD_LINE_1 = 0x80 # LCD RAM address for the 1st line
LCD_LINE_2 = 0xC0 # LCD RAM address for the 2nd line
LCD_BACKLIGHT = 0x08 # On

ENABLE = 0b00000100 # Enable bit

# Timing constants
E_PULSE = 0.0005
E_DELAY = 0.0005

# Open I2C interface
bus = smbus2.SMBus(1) # Rev 2 Pi uses 1

def lcd_byte(bits, mode):
    bits_high = mode | (bits & 0xF0) | LCD_BACKLIGHT
    bits_low = mode | ((bits << 4) & 0xF0) | LCD_BACKLIGHT

    bus.write_byte(I2C_ADDR, bits_high)
    lcd_toggle_enable(bits_high)

    bus.write_byte(I2C_ADDR, bits_low)
    lcd_toggle_enable(bits_low)

def lcd_toggle_enable(bits):
    time.sleep(E_DELAY)
    bus.write_byte(I2C_ADDR, (bits | ENABLE))
    time.sleep(E_PULSE)
    bus.write_byte(I2C_ADDR, (bits & ~ENABLE))
    time.sleep(E_DELAY)

```

```

def lcd_init():
    lcd_byte(0x33, LCD_CMD)
    lcd_byte(0x32, LCD_CMD)
    lcd_byte(0x06, LCD_CMD)
    lcd_byte(0x0C, LCD_CMD)
    lcd_byte(0x28, LCD_CMD)
    lcd_byte(0x01, LCD_CMD)
    time.sleep(E_DELAY)

def lcd_string(message, line):
    message = message.ljust(LCD_WIDTH, " ")
    lcd_byte(line, LCD_CMD)

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

lcd_init()

while True:
    lcd_string("Hello, World!", LCD_LINE_1)
    lcd_string("RPi & LCD I2C", LCD_LINE_2)
    time.sleep(2)

```

PROCEDURE

- Connect the SDA pin of the I2C LCD module to the SDA pin of the Raspberry Pi.
- Connect the SCL pin of the I2C LCD module to the SCL pin of the Raspberry Pi.
- Connect the VCC pin of the I2C LCD module to the 5V pin of the Raspberry Pi.
- Connect the GND pin of the I2C LCD module to the GND pin of the Raspberry Pi.

APPLICATIONS

- Display real-time sensor data on an LCD screen.
- Create an interactive user interface for a DIY project.
- Monitor system status and display alerts.
- Develop an information display for IoT devices.
- Use in educational projects to teach electronics and programming.

RASPBERRY PI INTERFACE WITH RELAY MODULE

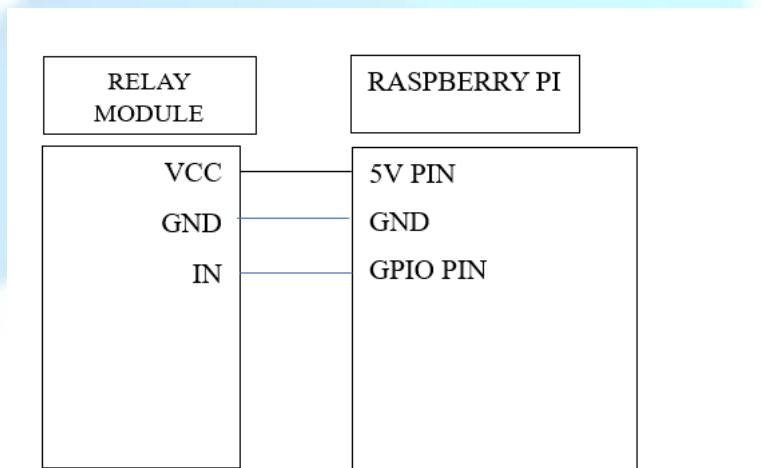
HARDWARE REQUIREMENTS

- Raspberry Pi board
- Relay module (1-channel, 2-channel, or more)
- Breadboard and jumper wires
- External power supply

OBJECTIVES

- Calculating and displaying the distance between the obstacle and the sensor
- Determining the distance to an object
- Understand the working of relay module
- Understand the pin configuration of Raspberry pi and relay module
- Understand Installation and writing of OS into Raspberry PI
- Understand developing code for interfacing Raspberry with relay module

BLOCK DIAGRAM



PROGRAM

```
import RPi.GPIO as GPIO
import time
GPIO.setmode(GPIO.BCM)
RELAY_PIN = 17 # GPIO pin for relay control

GPIO.setup(RELAY_PIN, GPIO.OUT)

def turn_relay_on():
    GPIO.output(RELAY_PIN, GPIO.HIGH)
    print("Relay turned ON")

def turn_relay_off():
    GPIO.output(RELAY_PIN, GPIO.LOW)
    print("Relay turned OFF")

try:
    turn_relay_off()
    time.sleep(2)
    turn_relay_on()
    time.sleep(2)
    turn_relay_off()

except KeyboardInterrupt:
    GPIO.cleanup()
```

PROCEDURE

- Connect the VCC of the sensor to the pin no 2, 5V of the Raspberry Pi
- Connect the Trig pin of the sensor to the pin no 11 of the Raspberry Pi.
- Connect the Echo pin of the sensor to the pin no 13 of the Raspberry Pi.
- Connect the Gnd pin of the sensor to the pin no 6, Gnd of the Raspberry Pi.

APPLICATIONS

- Control home appliances remotely for automation purposes.
- Build industrial automation systems for controlling machinery.
- Implement remote control systems for various devices.
- Automate agricultural systems such as irrigation.
- Create security systems that can control alarms and lights.

RASPBERRY PI INTERFACE WITH OLED DISPLAY

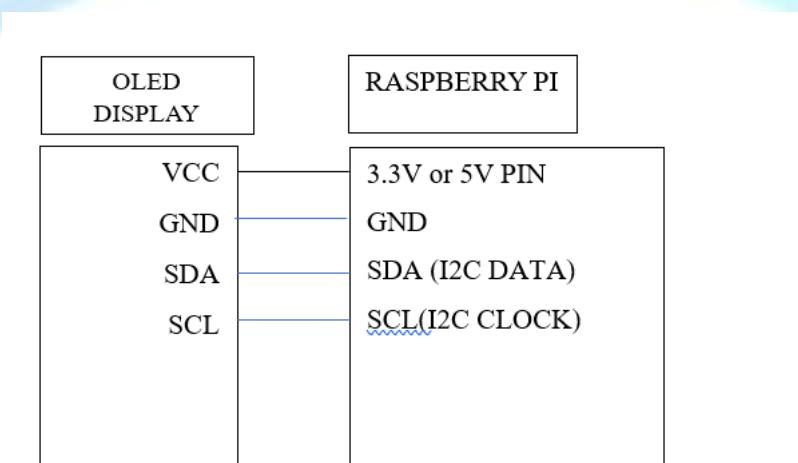
HARDWARE REQUIREMENTS

- Raspberry Pi board
- OLED display (SSD1306)
- Breadboard and jumper wires

OBJECTIVES

- Display sensor data in a compact form factor.
- Understand the working principles of OLED displays.
- Learn the pin configuration of the Raspberry Pi and the OLED display.
- Install and configure the necessary libraries for OLED interfacing.
- Develop code to interface the Raspberry Pi with the OLED display.

BLOCK DIAGRAM



PROGRAM

```
import RPi.GPIO as GPIO
import Adafruit_SSD1306
from PIL import Image, ImageDraw, ImageFont

RST = None
disp = Adafruit_SSD1306.SSD1306_128_64(rst=RST)

disp.begin()
disp.clear()
disp.display()

width = disp.width
height = disp.height
image = Image.new('1', (width, height))

draw = ImageDraw.Draw(image)
draw.rectangle((0, 0, width, height), outline=0, fill=0)

font = ImageFont.load_default()
draw.text((0, 0), "Hello World!", font=font, fill=255)

disp.image(image)
disp.display()
```

PROCEDURE

- Connect the SDA pin of the OLED display to the SDA pin of the Raspberry Pi.
- Connect the SCL pin of the OLED display to the SCL pin of the Raspberry Pi.
- Connect the VCC pin of the OLED display to the 3.3V pin of the Raspberry Pi.
- Connect the GND pin of the OLED display to the GND pin of the Raspberry Pi.

APPLICATIONS

- Display sensor data in a compact form factor.
- Create custom user interfaces for projects.
- Use as status indicators for various applications.
- Develop wearable technology with display capabilities.
- Design portable information displays for quick reference.

RASPBERRY PI INTERFACE WITH ACCELEROMETER

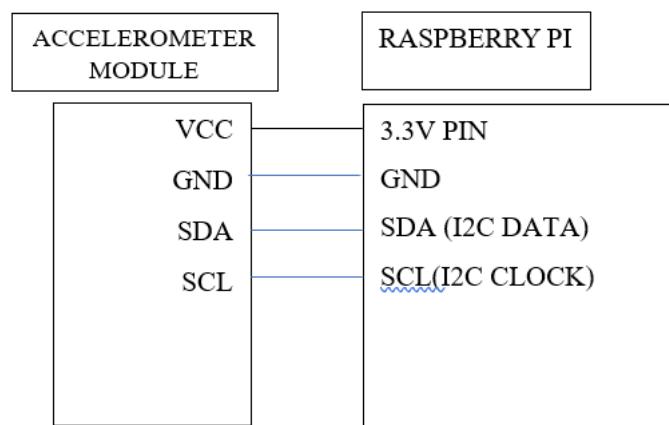
HARDWARE REQUIREMENTS

- Raspberry Pi board
- Accelerometer sensor (e.g., ADXL345)
- Breadboard and jumper wires
- I2C cable

OBJECTIVES

- Detect and measure motion for various applications.
- Understand the working principles of accelerometers.
- Learn the pin configuration of the Raspberry Pi and the accelerometer sensor.
- Install and configure the necessary libraries for accelerometer interfacing.
- Develop code to read data from the accelerometer sensor.

BLOCK DIAGRAM



PROGRAM

```

import RPi.GPIO as GPIO
import smbus
import time

bus = smbus.SMBus(1)
DEVICE = 0x53
bus.write_byte_data(DEVICE, 0x2D, 0x08)

def read_axis(axis):
    bytes = bus.read_i2c_block_data(DEVICE, 0x32, 6)
    value = bytes[axis] | (bytes[axis + 1] << 8)
    if value & (1 << 15):
        value = value - (1 << 16)
    return value

GPIO.setmode(GPIO.BCM)

try:
    while True:
        x = read_axis(0)
        y = read_axis(2)
        z = read_axis(4)
        print(f'X={x}, Y={y}, Z={z}')
        time.sleep(1)
except KeyboardInterrupt:
    GPIO.cleanup()

```

PROCEDURE

- Connect the SDA pin of the accelerometer to the SDA pin of the Raspberry Pi.
- Connect the SCL pin of the accelerometer to the SCL pin of the Raspberry Pi.
- Connect the VCC pin of the accelerometer to the 3.3V pin of the Raspberry Pi.
- Connect the GND pin of the accelerometer to the GND pin of the Raspberry Pi.

APPLICATIONS

- Detect and measure motion for various applications.
- Monitor vibrations in machinery for predictive maintenance.
- Determine orientation and angle in 3D space.
- Track physical activity for fitness applications.
- Implement balancing mechanisms in robotics projects.

RASPBERRY PI INTERFACE WITH BME280

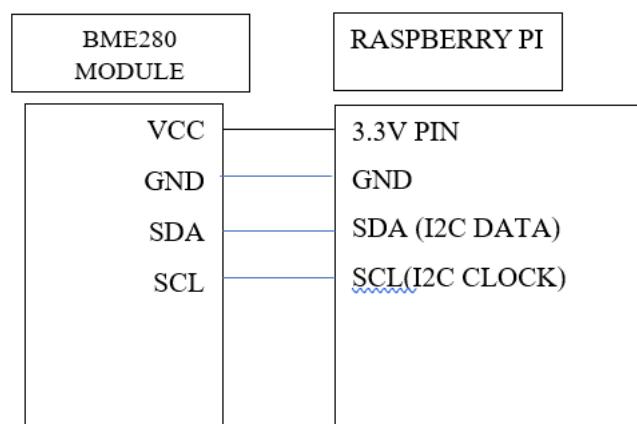
HARDWARE REQUIREMENTS

- Raspberry Pi board
- BME280 sensor module
- Breadboard and jumper wires

OBJECTIVES

- Collect weather data for environmental monitoring.
- Understand the working principles of the BME280 sensor.
- Learn the pin configuration of the Raspberry Pi and the BME280 sensor.
- Install and configure the necessary libraries for BME280 interfacing.
- Develop code to read data from the BME280 sensor and display it.

BLOCK DIAGRAM



PROGRAM

```

import RPi.GPIO as GPIO
import smbus2
import time

bus = smbus2.SMBus(1)
BME280_ADDRESS = 0x76

def read_bme280():
    data = bus.read_i2c_block_data(BME280_ADDRESS, 0x88, 24)
    # Compensation parameters and further data reading logic goes here
    return temperature, humidity, pressure

GPIO.setmode(GPIO.BCM)

try:
    while True:
        temperature, humidity, pressure = read_bme280()
        print(f'Temperature: {temperature:.2f} C')
        print(f'Humidity: {humidity:.2f} %')
        print(f'Pressure: {pressure:.2f} hPa')
        time.sleep(1)
except KeyboardInterrupt:
    GPIO.cleanup()

```

PROCEDURE

- Connect the SDA pin of the BME280 sensor to the SDA pin of the Raspberry Pi.
- Connect the SCL pin of the BME280 sensor to the SCL pin of the Raspberry Pi.
- Connect the VCC pin of the BME280 sensor to the 3.3V pin of the Raspberry Pi.
- Connect the GND pin of the BME280 sensor to the GND pin of the Raspberry Pi.

APPLICATIONS

- Collect weather data for environmental monitoring.
- Measure altitude for geographic applications.
- Monitor indoor air quality in smart home systems.
- Use in IoT projects to gather and analyze environmental data.
- Design scientific instruments for climate research.

RASPBERRY PI INTERFACE WITH USB WEBCAM

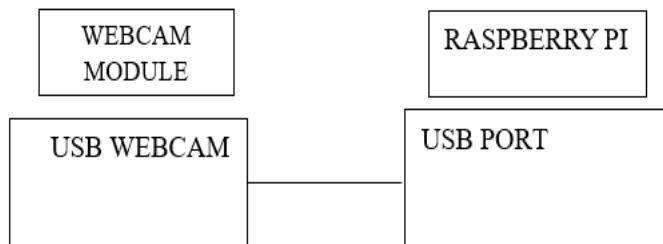
HARDWARE REQUIREMENTS

- Raspberry Pi board
- USB webcam
- USB hub
- External storage
- Breadboard and jumper wires

OBJECTIVES

- Stream video for remote monitoring and surveillance.
- Understand the working principles of USB webcams.
- Learn the pin configuration and setup of the Raspberry Pi for USB webcams.
- Install and configure the necessary software for video streaming.
- Develop code to capture and process video data from the USB webcam.

BLOCK DIAGRAM



PROGRAM

```
import RPi.GPIO as GPIO
import cv2

GPIO.setmode(GPIO.BCM)

cap = cv2.VideoCapture(0)

try:
    while True:
        ret, frame = cap.read()
        if ret:
            cv2.imshow('Webcam', frame)
            if cv2.waitKey(1) & 0xFF == ord('q'):
                break
except KeyboardInterrupt:
    GPIO.cleanup()
    cap.release()
    cv2.destroyAllWindows()
```

PROCEDURE

- Connect the USB webcam to a USB port on the Raspberry Pi.
- Install OpenCV and other necessary libraries for video processing.
- Run the program to capture and display video from the USB webcam.

APPLICATIONS

- Stream video for remote monitoring and surveillance.
- Set up video conferencing systems using open-source software.
- Implement computer vision projects for various applications.
- Record and analyze video for research and development.
- Create interactive video-based applications.



Introduction to IoT and LPWAN

Introduction to IoT



Fig.:Internet of things

Definition:

The Internet of Things (IoT) refers to the network of physical objects—devices, vehicles, buildings, and other items—that are embedded with sensors, software, and other technologies to connect and exchange data with other devices and systems over the internet. These objects are often referred to as "smart" devices due to their ability to collect and transmit data autonomously.

Evolution:

The concept of IoT has evolved significantly since its inception. The idea can be traced back to the early 1980s when a Coca-Cola vending machine at Carnegie Mellon University was modified to report its inventory and whether newly loaded drinks were cold. Over the decades, advancements in technology, particularly in wireless communication, sensor technology, and data analytics, have fueled the growth of IoT. The term "Internet of Things" itself was coined by Kevin Ashton in 1999, but it wasn't until the 2010s that IoT started gaining substantial momentum, driven by the proliferation of smartphones, advancements in cloud computing, and the development of more efficient wireless communication protocols.

Applications in Various Sectors

IoT has a broad range of applications across different sectors, transforming industries and improving efficiency, safety, and quality of life.

1. **Healthcare:**
 - Remote monitoring of patients using wearable devices.
 - Smart medical devices that track and transmit vital signs in real-time.
 - Automated insulin delivery systems for diabetes management.
2. **Agriculture:**
 - soil moisture, nutrient levels, and crop health.
 - Automated irrigation systems that optimize water usage based on real-time data.

3. Transportation:

- Fleet management systems that track vehicle locations, optimize routes, and monitor driver behavior.
- Smart traffic management systems that reduce congestion and enhance road safety.
- Connected cars that provide real-time diagnostics and predictive maintenance alerts.

4. Smart Cities:

- Intelligent street lighting that adjusts brightness based on pedestrian and vehicle movement.
- Waste management systems that optimize collection routes and schedules based on bin fill levels.
- Smart energy grids that manage distribution and consumption more efficiently.

5. Industrial Automation:

- Predictive maintenance systems that monitor equipment health and predict failures before they occur.
- Real-time monitoring of production lines to optimize efficiency and reduce downtime.

Future Trends in IoT

The future of IoT is poised for rapid growth and innovation, with several key trends expected to shape its development:

1. 5G Connectivity:

- The deployment of 5G networks will enhance IoT by providing faster, more reliable, and lower-latency connections, enabling more devices to be connected simultaneously.

2. Edge Computing:

- Shifting data processing to the edge of the network, closer to the source of the data, will reduce latency and bandwidth usage, improving real-time data analysis and decision-making.

3. AI and Machine Learning Integration:

- Incorporating AI and machine learning into IoT systems will enable more advanced analytics, predictive maintenance, and autonomous decision-making.

4. Increased Security Measures:

- As IoT devices become more widespread, ensuring robust security protocols to protect against cyber threats will be critical.

5. Sustainability:

- IoT will play a significant role in promoting sustainability, with applications in energy management, smart agriculture, and environmental monitoring.

Low Power Wide Area Networks (LPWAN)

Definition:

Low Power Wide Area Networks (LPWAN) are types of wireless telecommunication networks designed to allow long-range communications at a low bit rate among connected objects, such as sensors operated on a battery. These networks are optimized for low power consumption, long-range communication, and low data transfer rates, making them ideal for IoT applications.

Characteristics:

- **Long Range:** LPWAN technologies can cover distances of several kilometers, much farther than traditional wireless networks like Wi-Fi or Bluetooth.
- **Low Power Consumption:** Devices connected to LPWAN can operate for several years on small batteries, making them suitable for remote or hard-to-reach locations.
- **Low Data Rates:** LPWANs are designed for applications that require low bandwidth, such as sending small amounts of sensor data periodically.
- **Scalability:** LPWAN can support a large number of devices, making it ideal for large-scale IoT deployments.

Comparison with Other Network Technologies

1. Wi-Fi:

- **Range:** Short (up to 100 meters)
- **Power Consumption:** High
- **Data Rate:** High (up to several Gbps)
- **Use Case:** Suitable for high-bandwidth applications within limited areas, such as homes and offices.

2. Bluetooth:

- **Range:** Short (up to 100 meters)
- **Power Consumption:** Low to medium
- **Data Rate:** Medium (up to 2 Mbps)
- **Use Case:** Ideal for short-range communication between devices, such as wearable tech and peripheral devices.

3. Cellular (3G/4G/5G):

- **Range:** Long (up to several kilometers)
- **Power Consumption:** High
- **Data Rate:** High (up to several Gbps with 5G)
- **Use Case:** Suitable for applications requiring high data rates and mobility, such as smartphones and connected cars.

4. LPWAN:

- **Range:** Very long (up to 15-20 kilometers in rural areas)
- **Power Consumption:** Very low
- **Data Rate:** Low (up to 50 Kbps)
- **Use Case:** Ideal for low-power, long-range applications like smart meters, environmental monitoring, and asset tracking.

Applications of LPWAN in IoT

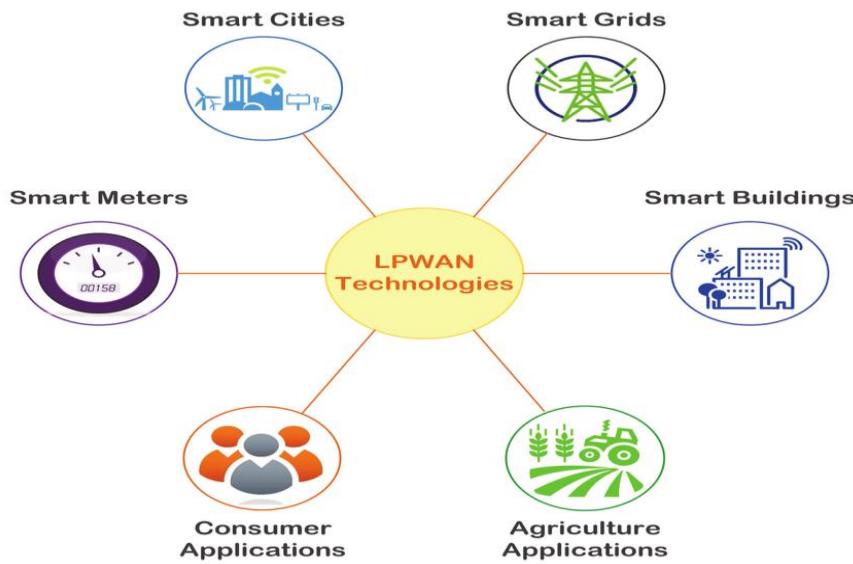


Fig.: Applications of LPWAN in IoT

1. Smart Agriculture:

- LPWAN can be used to connect sensors that monitor soil moisture, weather conditions, and crop health, enabling farmers to make data-driven decisions to optimize yield and reduce resource usage.

2. Asset Tracking:

- LPWAN enables the tracking of assets across vast distances, providing real-time location data for logistics and supply chain management.

3. Smart Cities:

- LPWAN can support various smart city applications, such as monitoring air quality, managing street lighting, and optimizing waste collection.

4. Utilities:

- Utility companies can use LPWAN to connect smart meters for water, gas, and electricity, allowing for remote monitoring and more efficient resource management.

5. Environmental Monitoring:

- LPWAN can facilitate the deployment of sensors in remote locations to monitor environmental parameters like air and water quality, providing critical data for conservation and climate research.

Overview of IoT

The Internet of Things (IoT) refers to the network of physical objects embedded with sensors, software, and other technologies to connect and exchange data with other devices and systems over the internet. These objects, often referred to as "smart" devices, range from ordinary household items to sophisticated industrial tools.

Key Components of IoT

1. **Devices/Things:** Physical objects with sensors and actuators that collect data and perform actions.
2. **Connectivity:** The communication infrastructure that connects devices to the internet and each other.
3. **Data Processing:** Systems and technologies that process, analyze, and act upon the collected data.
4. **User Interface:** Interfaces that allow users to interact with the IoT system, such as mobile apps or web dashboards.

Sensors and Actuators

Sensors

Sensors are devices that detect changes in the environment and generate data based on these changes. They are essential for gathering real-world data and can measure various parameters such as temperature, humidity, light, motion, and more.

Types of Sensors:

- Temperature Sensors: Measure ambient temperature (e.g., thermocouples, RTDs).
- Proximity Sensors: Detect the presence of nearby objects without physical contact (e.g., ultrasonic sensors, infrared sensors).
- Light Sensors: Measure light intensity (e.g., photoresistors, photodiodes).
- Motion Sensors: Detect movement (e.g., accelerometers, gyroscopes).

Actuators

Actuators are devices that receive signals from the IoT system and perform actions in the physical world. They convert electrical signals into physical movement or other forms of output.

Types of Actuators:

- Motors: Convert electrical signals into rotational or linear motion.
- Solenoids: Generate a magnetic field to create mechanical movement.
- Speakers: Convert electrical signals into sound.
- LEDs: Emit light when energized.

Connectivity and Communication Protocols

Connectivity

Connectivity refers to the methods used to connect IoT devices to each other and the internet. Reliable connectivity is crucial for effective data exchange and communication in IoT systems.

Common Connectivity Methods:

- **Wi-Fi:** Offers high data transfer rates and is widely used for home and office IoT devices.
- **Bluetooth:** Ideal for short-range communication, commonly used in wearable devices.
- **Zigbee:** Suitable for low-power, low-data-rate applications like home automation.
- **LoRaWAN:** Provides long-range, low-power communication for remote IoT applications.
- **Cellular:** Uses mobile networks (e.g., 4G, 5G) for IoT devices requiring wide-area coverage.

Communication Protocols

Communication protocols define the rules and formats for data exchange between IoT devices. They ensure interoperability and efficient data transmission.

Common Protocols:

- **MQTT (Message Queuing Telemetry Transport):** A lightweight protocol for publish/subscribe messaging, suitable for low-bandwidth and high-latency networks.
- **CoAP (Constrained Application Protocol):** Designed for simple electronics, it allows devices to communicate over the internet using minimal resources.
- **HTTP/HTTPS:** The foundation of web communication, used for sending data between IoT devices and web servers.

IoT Platforms and Frameworks

IoT Platforms

IoT platforms are comprehensive suites that facilitate the development, deployment, and management of IoT applications. They provide tools and services for device connectivity, data management, and application development.

Popular IoT Platforms:

- **AWS IoT:** Offers cloud services for connecting and managing IoT devices.
- **Microsoft Azure IoT:** Provides a range of tools for IoT development, including data analytics and AI integration.
- **Google Cloud IoT:** Enables secure device connection, data processing, and analysis.

IoT Frameworks

IoT frameworks are software libraries and tools that support the creation of IoT applications. They provide standardized methods for device communication, data handling, and user interaction.

Examples of IoT Frameworks:

- **Thing Speak:** An open-source platform for collecting and analyzing IoT data.
- **Kaa IoT:** A flexible framework for building end-to-end IoT solutions.

Types of IoT Networks

The Internet of Things (IoT) encompasses a vast array of devices and systems that communicate and share data over various types of networks. These networks are generally categorized based on their range, which affects the type of applications they are suited for. Understanding the types of IoT networks is crucial for deploying IoT solutions effectively.

1. Short-Range IoT Networks

Short-range IoT networks are typically used for communication within a limited area, such as a home, office, or a specific part of a factory. They offer high data rates and low latency, making them suitable for applications where fast and frequent data transfer is necessary. Key technologies in this category include Bluetooth, Zigbee, and Wi-Fi.

▪ **Bluetooth**

Bluetooth is a wireless technology standard for exchanging data over short distances using short-wavelength UHF radio waves in the ISM band from 2.4 to 2.485 GHz. It is widely used in consumer electronics such as smartphones, tablets, and wearable devices.

- Bluetooth Classic
- Bluetooth Low Energy (BLE)

▪ **Zigbee**

Zigbee is a specification for a suite of high-level communication protocols using low-power digital radios. It is designed for small, low-cost, low-power IoT applications.

▪ **Wi-Fi**

Wi-Fi is a technology that allows devices to connect to a local area network (LAN) using high-frequency radio waves. It is ubiquitous in homes and businesses.

- **High Data Rates:** Wi-Fi supports high data rates, suitable for video streaming and large data transfers.
- **Power Consumption:** Higher power consumption compared to Bluetooth and Zigbee, making it less suitable for battery-powered IoT devices.
- **Applications:** Smart home devices, security cameras, and connected consumer electronics.

2. Long-Range IoT Networks

Long-range IoT networks are designed to cover larger areas, such as cities or rural regions. They typically support low data rates and are optimized for low power consumption, enabling devices to operate on battery power for extended periods. Key technologies include LPWAN and Cellular IoT.

▪ **LPWAN (Low-Power Wide-Area Network)**

LPWAN technologies provide long-range communication at low bit rates, suitable for applications where devices need to transmit small amounts of data infrequently.

- **LoRa (Long Range)**
- **Sigfox**

- **Cellular IoT**

Cellular IoT leverages existing cellular networks to provide connectivity to IoT devices. This category includes technologies like NB-IoT and LTE-M.

- **NB-IoT (Narrowband IoT)**
- **LTE-M (LTE Cat-M1)**

3. Hybrid IoT Networks

Hybrid IoT networks combine short-range and long-range communication technologies to optimize performance and coverage. This approach leverages the strengths of each type of network to address various IoT application requirements.

- **Gateway Devices:** These devices act as intermediaries, connecting short-range devices (e.g., Bluetooth sensors) to long-range networks (e.g., cellular or LPWAN). This setup enables local data aggregation and transmission to cloud services over long distances.
- **Edge Computing:** Some hybrid networks incorporate edge computing, where data processing occurs close to the data source. This reduces latency and bandwidth usage by processing data locally and only sending necessary information to the cloud.

Understanding LoRa Technology

Introduction to LoRa Technology

LoRa (Long Range) is a wireless modulation technique designed for low-power, wide-area networks (LPWANs) necessary for Internet of Things (IoT) and machine-to-machine (M2M) communications. It enables long-range communication with minimal power consumption, making it ideal for applications where battery life and connectivity range are critical factors.

LoRa Modulation

LoRa modulation is the core technology that enables long-range communication with low power consumption. It uses Chirp Spread Spectrum (CSS) modulation, where data is encoded into chirp signals that vary in frequency over time. This technique allows LoRa to achieve greater communication distances compared to traditional modulation schemes like FSK (Frequency Shift Keying) or ASK (Amplitude Shift Keying).

Technical Details of LoRa Modulation

LoRa operates in the sub-GHz ISM (Industrial, Scientific, and Medical) bands, typically 433 MHz, 868 MHz, or 915 MHz, depending on regional regulations. It uses narrowband signals with bandwidths ranging from 125 kHz to 500 kHz, which contributes to its high sensitivity and efficient use of spectrum.

The modulation schemes in LoRa include:

- **Chirp Spread Spectrum (CSS):** Utilizes chirp signals to encode data, allowing for robust performance in noisy environments and long-range communication.
- **Forward Error Correction (FEC):** Implements error correction techniques to enhance reliability, especially over long distances and in challenging RF conditions.

Advantages of LoRa in IoT Applications

LoRa technology offers several advantages that make it suitable for a wide range of IoT applications:

- **Long Range:** Capable of covering distances of several kilometers in rural areas and dense urban environments.
- **Low Power Consumption:** Devices can operate on battery power for years, reducing maintenance and operational costs.
- **Low Cost:** Cost-effective infrastructure deployment due to the use of unlicensed spectrum and simple network architecture.
- **Scalability:** Supports thousands of nodes per gateway, making it scalable for large-scale IoT deployments.
- **Robustness:** Resilient to interference and capable of maintaining communication in challenging RF environments.

History and Evolution of LoRa

Origins of LoRa Technology

LoRa (Long Range) technology emerged as a significant advancement in the realm of wireless communication, particularly in the domain of Internet of Things (IoT). Developed to address the challenges of long-range, low-power communication, LoRa technology was pioneered by Semtech Corporation, a leading semiconductor company.

The inception of LoRa can be traced back to Semtech's exploration into ultra-low-power radio frequency (RF) solutions in the early 2010s. The goal was to create a robust, energy-efficient wireless communication technology capable of transmitting small packets of data over long distances, while consuming minimal power. This would cater to the growing demand for IoT applications that required connectivity in remote or challenging environments.

Milestones in LoRaWAN Development

A significant milestone in the evolution of LoRa technology was the establishment of the LoRa Alliance in 2015. The LoRa Alliance is a collaborative ecosystem of industry leaders committed to standardizing and promoting the adoption of LoRaWAN, the protocol layer that enables seamless interoperability across different LoRa-based devices and networks.

Key milestones in the development of LoRaWAN include the release of various specifications aimed at enhancing network security, scalability, and efficiency. These specifications, developed through collective expertise within the LoRa Alliance, have contributed to making LoRaWAN a preferred choice for IoT deployments globally.

The LoRa Alliance, comprising a diverse ecosystem of over 500 members, has played a crucial role in driving the adoption of LoRa technology globally. Member contributions span device manufacturing, network deployment, software development, and application deployment across various industries including smart cities, agriculture, logistics, and industrial automation.

Key Developments and Enhancements

Over the years, LoRa technology has undergone several key developments and enhancements:

- **Advanced Modulation Techniques:** Continuous improvements in modulation schemes have expanded the capabilities of LoRa technology, enabling higher data rates and improved spectral efficiency.
- **Geolocation Capabilities:** Integration of geolocation services has facilitated asset tracking and location-based services using LoRa-enabled devices.
- **Battery Optimization:** Ongoing efforts to optimize power consumption have extended battery life for connected devices, enhancing operational efficiency and reducing maintenance costs.

LoRaWAN Protocol

LoRaWAN (Long Range Wide Area Network) is the protocol layer built on top of LoRa modulation, defining the communication protocol and system architecture for LoRa-based networks.

Overview of LoRaWAN Stack

The LoRaWAN protocol stack consists of several layers:

- **Physical Layer (PHY):** Defines the physical parameters such as frequency, modulation, and data rate.
- **Data Link Layer:** Manages the communication between end-devices and gateways, including frame structure, addressing, and security.
- **MAC Layer (Medium Access Control):** Handles the access to the shared medium and ensures efficient communication between devices and gateways.
- **Application Layer:** Contains the application-specific protocols and interfaces for data integration and management.

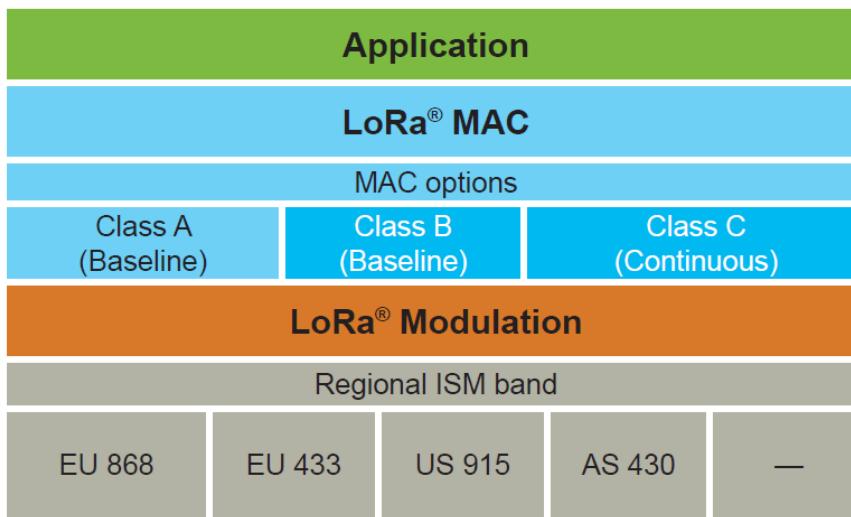


Fig.: Overview of LoRaWAN Stack

Classes of LoRaWAN Devices (Class A, B, C)

LoRaWAN supports three device classes, each with different capabilities and power consumption profiles:

- **Class A:** Lowest power consumption; devices can communicate with the gateway at any time but with asymmetric communication (uplink more frequent than downlink).
- **Class B:** Adds scheduled receive slots for downlink communication, allowing bi-directional communication with slight increases in power consumption.
- **Class C:** Highest power consumption; devices are almost always listening to receive data from the gateway, suitable for applications requiring almost immediate downlink data.

Class A

All LoRaWAN end-devices must support Class A implementation. A Class A device can send an uplink message at any time. Once the uplink transmission is completed, the device opens two short receive windows for receiving downlink messages from the network. There is a delay between the end of the uplink transmission and the start of each receive window, known as RX1 Delay and RX2 Delay, respectively. If the network server does not respond during these two receive windows, the next downlink will be scheduled immediately after the next uplink transmission.

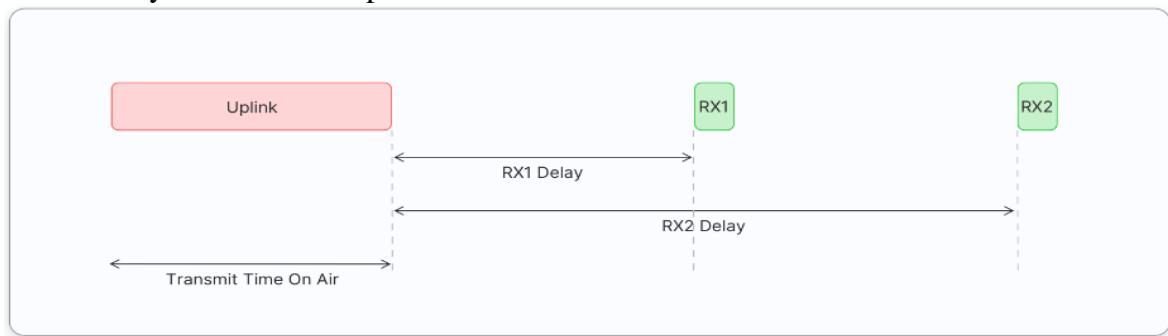


Fig.: Class A

The network server can respond during the first receive window (RX1) or the second receive window (RX2), but does not use both windows. The end device opens both receive windows but it doesn't receive an downlink message during either receive window. The end device receives a downlink during the first receive window and therefore it does not open the second receive window. The end device opens the first receive window but it does not receive a downlink. Therefore it opens the second receive window and it receives a downlink during the second receive window.

Let's consider three situations for downlink messages as illustrated below

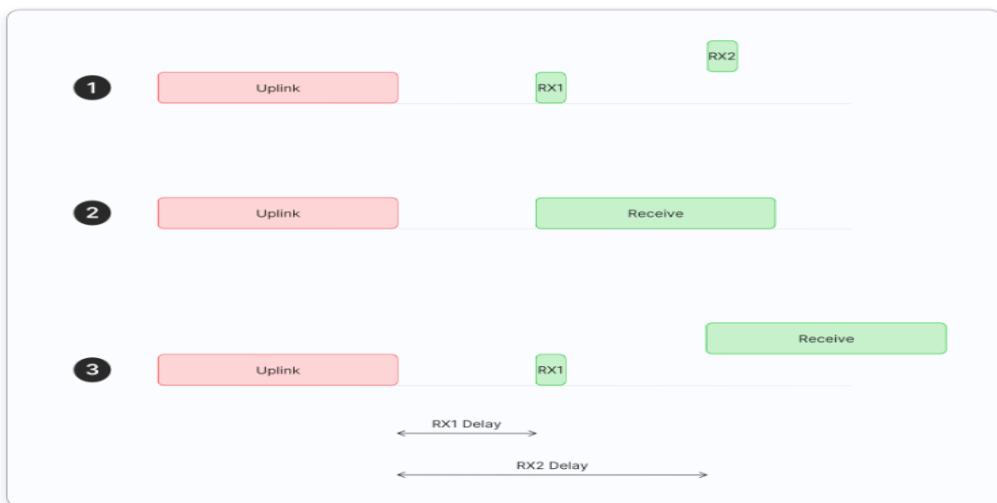


Fig.: 3 situations for downlink messages

Class A end devices have very low power consumption. Therefore, they can operate with battery power. They spend most of their time in sleep mode and usually have long intervals between uplinks. Class A devices have high downlink latency, as they require sending an uplink to receive a downlink.

The use cases for Class A end devices are:

- Environmental monitoring
- Animal tracking
- Forest fire detection
- Water leakage detection
- Smart parking
- Asset tracking
- Waste management

Class B

Class B devices extend Class A capabilities by periodically opening receive windows called ping slots to receive downlink messages. The network broadcasts a time-synchronized beacon (unicast and multicast) periodically through the gateways, which is received by the end devices. These beacons provide a timing reference for the end devices, allowing them to align their internal clocks with the network. This allows the network server to know when to send a downlink to a specific device or a group of devices. The time between two beacons is known as the beacon period.

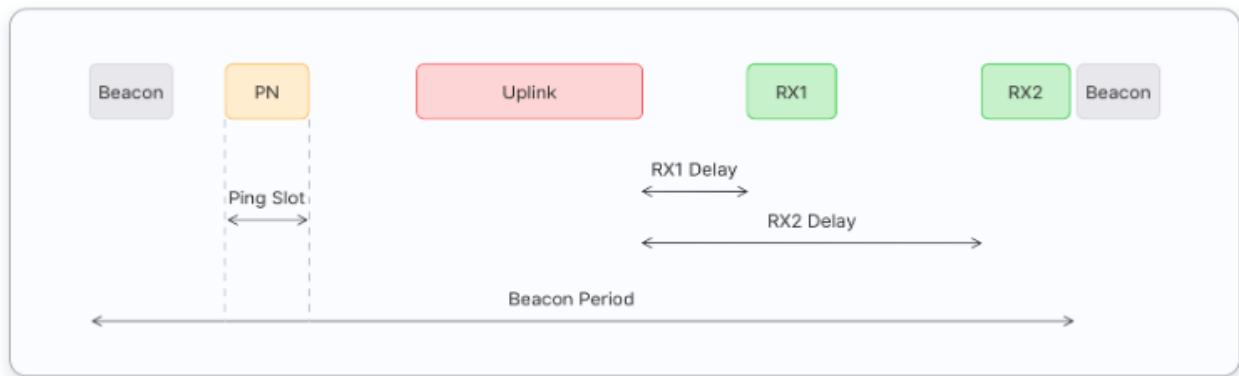


Fig.: Class B

After an uplink, the two short receive windows, RX1 and RX2 will open similar to Class A devices. Class B end devices have low latency for downlinks compared to Class A end devices because they periodically open ping slots. However, they have much higher latency than the Class C end devices. Class B devices are often battery powered. The battery life is shorter in Class B compared to Class A because the devices spend more time in active mode due to receiving beacons and having open ping slots. Because of the low latency for downlinks, Class B mode can be used in devices that require medium-level critical actuation, such as utility meters.

The following are some of the use cases for Class B end devices:

- Utility meters (electrical meters, water meters, etc)
- Street lights

Class B devices can also operate in Class A mode.

Class C

Class C devices extend Class A capabilities by keeping the receive windows open unless transmitting an uplink, as shown in the figure below.

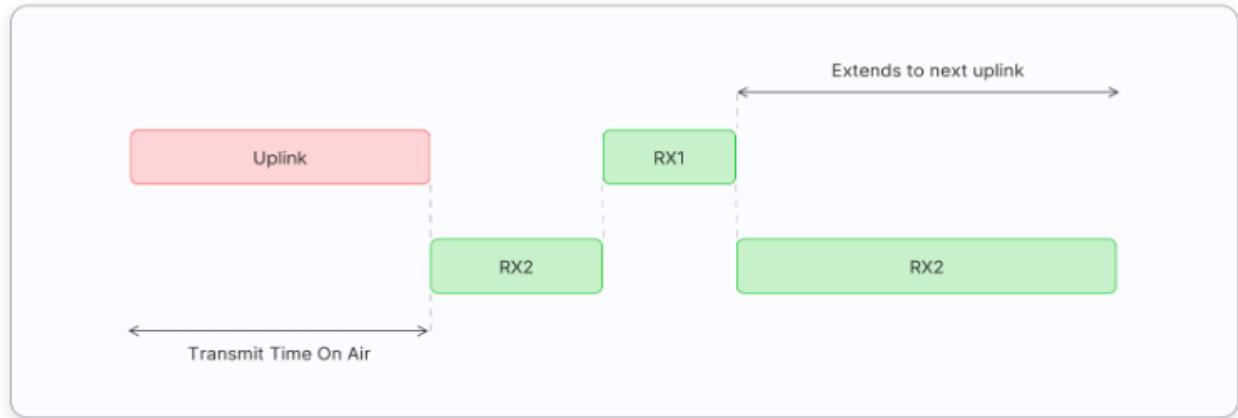


Fig.: Class C

Class C devices can receive downlink messages at almost any time, thus having very low latency for downlinks. These downlink messages can be used to activate certain functions of a device, such as reducing the brightness of a street light or turning on the cut-off valve of a water meter. Class C devices open two receive windows, RX1 and RX2, similar to Class A. However, the RX2 receive window remains open until the next uplink transmission. After the device sends an uplink, a short RX2 receive window opens, followed by a short RX1 receive window, and then the continuous RX2 receive window opens. This RX2 receive window remains open until the next uplink is scheduled. Uplinks are sent when there is no downlink in progress.

Compared to Class A and Class B devices, Class C devices have the lowest latency. However, they consume more power due to the need for opening continuous receive slots. As a result, these devices cannot be operated with batteries for long time therefore they are often mains powered.

The following are some of the use cases for Class C end devices:

- Utility meters (electrical meters, water meters, etc)
- Street lights
- Beacon lights
- Alarms

Class C devices can also operate in Class A mode.

Network Topology and Communication Model

LoRaWAN networks typically follow a star-of-stars topology where multiple end-devices communicate with one or more gateways. Gateways relay data between end-devices and a centralized network server, which manages the entire network and interfaces with applications through standard protocols like MQTT or HTTP.

LoRaWAN Protocol

LoRaWAN (Long Range Wide Area Network) is the protocol layer built on top of LoRa modulation, defining the communication protocol and system architecture for LoRa-based networks.

LoRaWAN Architecture Overview

LoRaWAN architecture consists of several key components that work together to enable communication between IoT devices (nodes) and network servers. The architecture is divided into three main parts: end devices, gateways, and network servers.

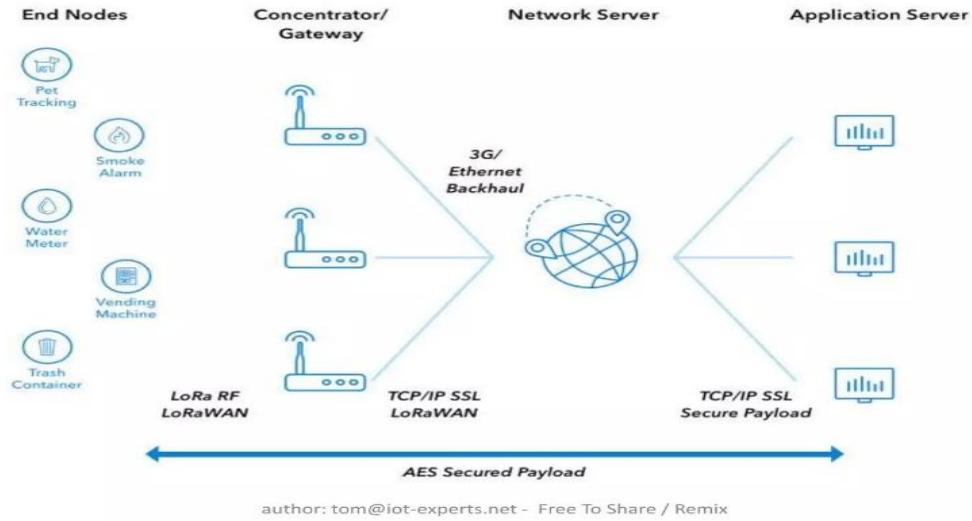


Fig.: LoRaWAN Architecture

Key Components of LoRaWAN

1. End Devices (Nodes):

- **Sensors/Actuators:** These are the physical devices that collect data (sensors) or perform actions (actuators). They are typically battery-powered and transmit data intermittently.
- **LoRa Transceiver:** Each end device is equipped with a LoRa transceiver module that modulates and transmits data using the LoRa modulation technique. This allows for long-range communication with minimal power consumption.

2. Gateways:

- **Receivers:** Gateways receive data from end devices within their range and forward it to the network server. They are usually connected to the internet via Ethernet, Wi-Fi, or cellular networks.
- **Multi-channel Capability:** Gateways support multiple channels and can handle simultaneous communications from multiple end devices.

3. Network Server:

- **Central Coordinator:** The network server manages the entire LoRaWAN network. It authenticates end devices, manages communication sessions, and routes data between end devices and application servers.
- **Security:** It ensures secure communication by managing encryption keys and enforcing security protocols.
- **Integration with Applications:** The network server interfaces with application servers where data is processed, analyzed, and acted upon based on specific IoT applications.

4. Application Server:

- **Data Processing:** Receives data from network servers and processes it based on specific application logic.
- **Integration:** Interfaces with external applications or databases for further analysis, storage, or visualization of IoT data.
- **APIs:** Provides interfaces for developers to retrieve data or send commands to end devices, facilitating real-time interaction with IoT deployments.

LoRa vs. Other Wireless Technologies

Aspect	LoRa	WiFi	Bluetooth
Range	Several km in rural, 100s of meters in urban	Up to 100 meters indoors, more outdoors	Up to 100 meters
Bandwidth	Low (tens to hundreds of kbps)	High (Mbps to Gbps)	Moderate (up to 2 Mbps)
Power Consumption	Low	High	Moderate
Suitable Applications	Agriculture, asset tracking, remote sensing	Video surveillance, industrial automation	Wearable devices, smart home appliances
Strengths	Long-range, low power consumption	High data transfer rates, low latency	Low power consumption, easy connectivity
Weaknesses	Limited bandwidth, longer latency	High power consumption, limited range	Limited range, bandwidth constraints

Gateways and End Devices

Introduction

In the realm of network architecture, gateways and end devices play fundamental roles in facilitating communication and data exchange across various networks. Understanding their functions, characteristics, and interactions is crucial for designing, implementing, and maintaining efficient and reliable network infrastructures.

Definition and Role of Gateways

A gateway serves as a bridge between different networks or network segments that use different protocols. It acts as an interface that enables seamless communication between these networks, ensuring compatibility and efficient data transfer. Gateways operate at the network layer (Layer 3) of the OSI model and perform protocol translation, enabling devices using different communication protocols to communicate effectively.

Gateways are often used to connect networks of different types, such as connecting a local area network (LAN) to the internet or connecting an Ethernet network to a Wi-Fi network. They can also provide additional functions such as security, firewall protection, and network management.

Types of Gateways

LoRaWAN gateways can be categorized into indoor (picocell) and outdoor (macrocell) gateways.

Indoor gateways

- Indoor gateways are cost-effective and suitable for providing coverage in places like deep-indoor locations (spaces covered by multiple walls), basements, and multi-floor buildings.
- These gateways have internal antennas or external ‘pigtail’ antennas. However depending on the indoor physical environment some indoor gateways can receive messages from sensors located several kilometers away.

Outdoor gateways

- Outdoor gateways provide a larger coverage than the indoor gateways. They are suitable for providing coverage in both rural and urban areas.
- These gateways can be mounted on cellular towers, the rooftops of very tall buildings, metal pipes (masts) etc. Usually an outdoor gateway has an external antenna (i.e. Fiberglass antenna) connected using a coaxial cable.

LoRa-based End Devices

A LoRaWAN end device can be a sensor, an actuator, or both. They are often battery operated. These end devices are wirelessly connected to the LoRaWAN network through gateways using LoRa RF modulation. In the majority of applications, an end device is an autonomous, often battery-operated sensor that digitizes physical conditions and environmental events. Typical use cases for an actuator include: street lighting, wireless locks, water valve shut off, leak prevention, among others.

When they are being manufactured, LoRa-based devices are assigned several unique identifiers. These identifiers are used to securely activate and administer the device, to ensure the safe transport of packets over a private or public network and to deliver encrypted data to the Cloud.

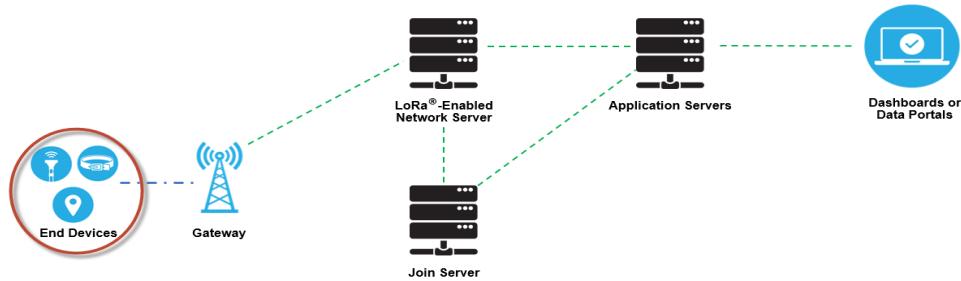


Fig.: LoRa end devices

Characteristics and Functions of End Devices

End devices, also known as hosts, are devices connected to a network that are the ultimate source or destination of data. These devices interact directly with users and perform specific tasks based on user input or network requests. End devices can vary widely in terms of functionality and form factor, including:

- **Computers:** Workstations, laptops, and servers that process and store data.
- **Mobile Devices:** Smartphones, tablets, and wearable devices that provide mobility and access to network resources.
- **Printers and Scanners:** Devices that facilitate document management and reproduction within the network.
- **IoT Devices:** Sensors, actuators, and other devices that collect and transmit data over the internet.

Communication between End Devices and Gateways

End devices communicate with gateways and other network devices using protocols and addressing schemes. The gateway acts as an intermediary that forwards data between end devices and ensures that information reaches its intended destination across different networks or segments.

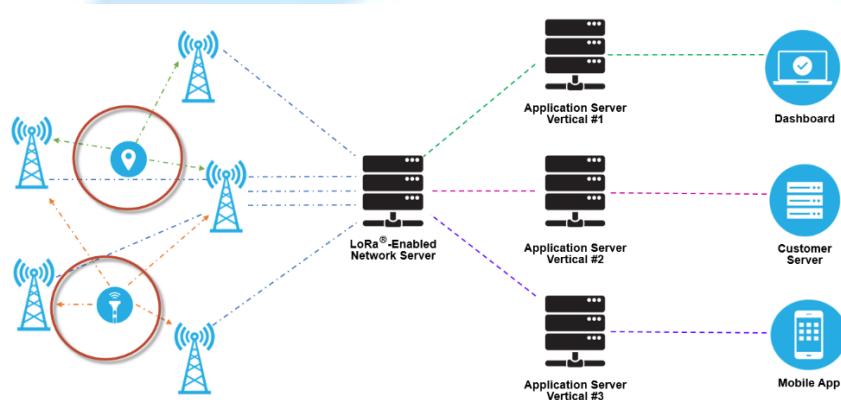


Fig.: Communication between End Devices and Gateways

LoRa allows for scalable, cost-optimized gateway implementation, depending on deployment objectives. For example, in North America, 8-, 16-, and 64-channel gateways are available. The 8-channel gateways are the least expensive. The type of gateway needed will depend on the use case. Eight- and 16-channel gateways are available for both indoor and outdoor use. Sixty-four channel gateways are only available in a carrier-grade variant. This type of gateway is intended for deployment in such places as cell towers, the rooftops of very tall buildings, etc.

Communication between end devices and gateways involves several key processes:

1. **Addressing:** Each end device on a network is assigned a unique identifier, such as an IP address, that enables accurate routing and delivery of data packets.
2. **Routing:** Gateways use routing tables to determine the best path for forwarding data packets to their destination. This process involves evaluating network topology, traffic patterns, and other factors to optimize data transmission.
3. **Protocol Conversion:** Gateways translate data between different communication protocols to facilitate seamless communication between devices that use different standards or formats.

LoRa Network Server (LNS)

Introduction to LoRa Technology

LoRa (Long Range) is a wireless communication technology designed for long-range communication with low power consumption. It is particularly suitable for Internet of Things (IoT) applications that require long battery life and low data rates over long distances.

Components of a LoRaWAN Network

A LoRaWAN (Long Range Wide Area Network) typically consists of three main components:

1. **End Devices (Nodes):** These are sensors or devices that collect data and communicate over the LoRa network.
2. **Gateways:** Devices that receive data from end devices and forward it to the network server.
3. **LoRa Network Server (LNS):** The central component responsible for managing communication between gateways and applications.

Role and Functionality of LoRa Network Server (LNS)

The LNS plays a crucial role in the operation of a LoRaWAN network. Its primary functions include:

1. Device Management

- **Activation:** The LNS manages the activation process of end devices on the network. This includes both Over-the-Air Activation (OTAA) and Activation by Personalization (ABP) methods.
- **Configuration:** It handles configuration parameters for end devices, such as data rate, frequency channels, and transmission power.
- **Firmware Updates:** LNS can facilitate firmware updates to end devices over the air.

2. Network Management

- **Routing:** The LNS determines the optimal route for data transmission from gateways to end devices and vice versa.
- **Security:** It manages security aspects such as message integrity and encryption to ensure data confidentiality.
- **Quality of Service (QoS):** Ensures reliable delivery of messages by implementing acknowledgments and retransmissions.

3. Data Processing and Integration

- **Data Handling:** LNS collects, processes, and stores data received from end devices. It formats data according to application requirements before forwarding it to the appropriate application servers.
- **Integration:** It integrates with application servers and other backend systems using standard protocols like MQTT, HTTP, or CoAP.

4. Monitoring and Diagnostics

- **Monitoring:** LNS monitors the health and status of gateways and end devices within the network.
- **Diagnostics:** It provides tools for troubleshooting network issues and analyzing performance metrics.

Architecture of LoRa Network Server (LNS)

The architecture of an LNS typically involves several components:

- **Message Broker:** Manages communication between gateways and application servers.
- **Network Controller:** Orchestrates device management and network operations.
- **Database:** Stores device information, configuration data, and historical records.
- **API Gateway:** Provides interfaces for external applications to interact with the LNS.

Characteristics of LNS

Scalability

- LNS should be designed to scale horizontally to support a growing number of devices and gateways.

Security

- Strong encryption and authentication mechanisms are crucial to protect data transmitted over the network.

Interoperability

- LNS should support interoperability with various types of gateways and application servers.

Performance

- Low latency and high reliability are essential for real-time applications and critical services.

Uses of LoRa Network Server (LNS)

Smart Cities

- Monitoring environmental parameters such as air quality and noise levels.
- Controlling street lighting based on traffic conditions.

Industrial IoT

- Tracking assets and monitoring equipment performance in factories.
- Optimizing logistics and supply chain management.

Application Servers and Backend Integration

Introduction

In the landscape of modern software development, Application Servers and Backend Integration play pivotal roles in ensuring the functionality, scalability, and interoperability of applications. These components are fundamental to the architecture of web and enterprise applications, facilitating seamless communication between front-end user interfaces and backend systems.

What is an Application Server?

An Application Server, often referred to as an App Server, is a software framework that provides an environment in which applications can run, including middleware services such as transaction management, security, data access, messaging, and more. It acts as a bridge between the front-end (user interface) and the backend (database and other services), ensuring efficient processing of client requests and responses.

Components of an Application Server:

1. **Servlet/JSP Container:** Manages the execution of Java Servlets and JavaServer Pages (JSP), handling HTTP requests from clients and generating dynamic content.
2. **Enterprise JavaBeans (EJB) Container:** Supports the deployment and execution of enterprise-level components, managing transactions, security, concurrency, and persistence.
3. **Web Services Support:** Facilitates the integration and interoperability of distributed applications through web services standards such as SOAP (Simple Object Access Protocol) and REST (Representational State Transfer).
4. **Connection Pooling:** Optimizes resource utilization by maintaining a pool of reusable database connections, enhancing performance and scalability.

5. **Security Infrastructure:** Provides mechanisms for authentication, authorization, and secure communication between components and clients, ensuring data integrity and confidentiality.

Backend Integration

Backend Integration involves the seamless connection of various backend systems, including databases, legacy systems, external APIs, and other services, to support the functionality of an application. This integration ensures that data flows efficiently between different components, enabling real-time processing and retrieval of information.

Techniques for Backend Integration:

1. **RESTful APIs:** Representational State Transfer (REST) APIs facilitate lightweight and scalable integration between applications using standard HTTP methods (GET, POST, PUT, DELETE). They promote interoperability and are widely adopted for building microservices architectures.
2. **Message-Oriented Middleware (MOM):** Uses asynchronous messaging protocols such as JMS (Java Message Service) or AMQP (Advanced Message Queuing Protocol) to decouple components and enable reliable communication between disparate systems.
3. **Enterprise Integration Patterns (EIP):** Defines standardized ways to handle common integration challenges such as message routing, transformation, and aggregation. Patterns like publish-subscribe, request-reply, and content-based routing are used to design robust integration solutions.
4. **Data Synchronization and ETL (Extract, Transform, Load):** Involves processes for extracting data from various sources, transforming it into a compatible format, and loading it into a target database or data warehouse. ETL tools automate these processes, ensuring data consistency and accuracy.

Practical Implications and Considerations

Successful implementation of Application Servers and Backend Integration requires careful consideration of several factors:

1. **Scalability and Performance:** App Servers should be capable of handling increasing loads and concurrent users without compromising performance. Backend integration should be designed to minimize latency and maximize throughput.
2. **Security and Compliance:** Robust security measures must be implemented to protect sensitive data and prevent unauthorized access. Compliance with regulatory requirements such as GDPR, HIPAA, or PCI DSS is essential for handling personal and financial information.
3. **Monitoring and Management:** Tools for monitoring server health, performance metrics, and transaction logs are critical for identifying bottlenecks, optimizing resource utilization, and ensuring high availability.
4. **Versioning and Compatibility:** As applications evolve, maintaining backward compatibility and managing versioned APIs and services becomes crucial to prevent disruptions in service.

Advantages of LoRa Devices

The adoption of LoRa devices and sensors has been driven by several compelling advantages:

- **Long Range:** LoRa devices can achieve communication ranges of several kilometers in urban environments and over 10 kilometers in rural settings, depending on factors like antenna quality and environmental conditions.
- **Low Power Consumption:** LoRa devices are designed to operate on low power, making them ideal for battery-operated applications. This efficiency extends battery life significantly compared to traditional cellular or Wi-Fi devices.
- **Scalability:** LoRa networks can support thousands of devices per gateway, making them highly scalable for large-scale IoT deployments. This scalability is crucial for applications in smart cities, industrial monitoring, and precision agriculture.
- **Cost-Effectiveness:** The infrastructure costs associated with deploying LoRa networks are relatively low compared to other IoT technologies. This affordability has democratized IoT solutions, enabling small and medium-sized enterprises to adopt smart technologies.

Applications of LoRa Devices and Sensors

LoRa technology has found diverse applications across various industries:

- **Smart Cities:** In urban environments, LoRa enables efficient management of street lighting, waste management systems, parking spaces, and environmental monitoring (air quality, noise levels).
- **Agriculture:** LoRa sensors are used for precision farming, enabling farmers to monitor soil moisture, temperature, and crop health remotely. This data-driven approach improves crop yield while conserving water and reducing costs.
- **Industrial IoT (IIoT):** Industries utilize LoRa for asset tracking, predictive maintenance of machinery, and monitoring of manufacturing processes. These applications enhance operational efficiency and reduce downtime.
- **Utilities:** LoRa-based smart meters provide utilities with real-time data on energy consumption, enabling them to optimize distribution, detect anomalies, and improve overall service reliability.

Introduction to LoRa Devices

LoRa (Long Range) devices are a crucial component of Low Power Wide Area Network (LPWAN) technology, designed for efficient long-distance communication with minimal power consumption. This makes them ideal for applications requiring low data rates and extended battery life in remote or urban environments. Understanding the types and functionalities of LoRa devices is essential for designing and implementing robust IoT (Internet of Things) solutions.

Types of LoRa Devices

1. LoRa Modules

LoRa modules are compact, integrated devices that encapsulate the necessary components for LoRa communication. They typically include:

Transceiver: Handles the modulation/demodulation of radio signals using the LoRa protocol.

Microcontroller: Manages data processing, interfacing with sensors, and executing communication protocols.

Antenna Interface: Connects to an external antenna for transmitting and receiving signals.

Power Management Circuitry: Regulates power supply to ensure efficient operation and battery longevity.

Interfaces (UART, SPI, I2C): Allows communication with external devices and sensors.

Modules come in various form factors (e.g., surface-mount, through-hole) and may incorporate additional features such as onboard sensors (temperature, humidity) or GPS for location tracking.

2. LoRa Sensors

LoRa sensors are specialized devices that integrate LoRa communication capabilities with sensor functionalities. They are designed to collect specific types of data and transmit it over LoRa networks. Examples include:

Environmental Sensors: Measure parameters like temperature, humidity, air quality, and ambient light.

Motion Sensors: Detect movement or vibrations, used in security systems or industrial monitoring.

Industrial Sensors: Monitor parameters such as pressure, flow rates, or machine status in industrial automation.

Smart City Sensors: Deployed in urban environments for monitoring traffic, parking availability, noise levels, etc.

Agricultural Sensors: Monitor soil moisture, temperature, and crop health in precision agriculture applications.

LoRa sensors often feature low-power consumption and can operate on battery power for extended periods, making them suitable for remote and outdoor deployments.

3. LoRa Gateways

LoRa gateways serve as bridge devices between end-node LoRa devices and the internet or a private network server. Key features include:

Radio Interface: Receives LoRa signals from multiple devices within its range.

Network Interface: Connects to the backend server or cloud platform via Ethernet, Wi-Fi, or cellular networks.

Processing Power: Handles packet forwarding, data aggregation, and network management functions.

Antenna Configuration: Utilizes multiple antennas for optimal coverage and signal reception.

Security Features: Ensures secure data transmission and device authentication.

Gateways are essential for creating LoRaWAN (LoRa Wide Area Network) infrastructure, enabling scalable IoT deployments across large geographical areas.

4. LoRaWAN End Devices

LoRaWAN end devices encompass both LoRa modules and sensors that conform to the LoRaWAN protocol specifications. They communicate with LoRa gateways to transmit data to the network server or cloud platform.

Key characteristics include:

Low Power Consumption: Prolongs battery life, crucial for remote and battery-operated applications.

Long Range: Capable of communication over several kilometers in open spaces, depending on environmental conditions.

Scalability: Supports deployment of thousands of devices within a single network infrastructure.

Secure Communication: Implements encryption and authentication mechanisms to protect data integrity and privacy.

Applications of LoRa Devices

LoRa devices find applications across various industries and use cases, including:

Smart Agriculture: Monitoring soil conditions, weather data, and crop health.

Smart Cities: Tracking environmental parameters, managing utilities, and optimizing traffic flow.

Connecting sensors to LoRa modules

Selecting Sensors for Integration

Before connecting sensors to LoRa modules, it's crucial to select sensors that are compatible with the LoRa communication protocol and suitable for the specific application requirements. Common sensor types include:

- **Environmental Sensors:** Measure temperature, humidity, air quality, etc.
- **Motion Sensors:** Detect movement or vibration.
- **Level Sensors:** Monitor liquid or gas levels.
- **Biometric Sensors:** Measure physiological parameters (heart rate, etc.).
- **Position Sensors:** Track GPS coordinates or positional data.
- **Light Sensors:** Measure ambient light levels.

The choice of sensor depends on factors such as the environment, power requirements, data accuracy, and the specific use case.

Configuring the LoRa Module

1. **Hardware Setup:** Connect the LoRa module to the microcontroller or development board (e.g., Arduino, Raspberry Pi) that interfaces with the sensor(s).
2. **Software Configuration:** Utilize the LoRaWAN (Long Range Wide Area Network) protocol stack or proprietary LoRa protocol libraries provided by the module manufacturer. Configure parameters such as frequency band, spreading factor, and transmission power to optimize communication range and energy efficiency.
3. **Integration with Microcontroller:** Write or adapt firmware to read data from sensors and format it for transmission via the LoRa module. Implement error checking and data encryption if necessary to ensure data integrity and security.

Establishing Communication Protocols

LoRa modules typically communicate using the LoRaWAN protocol, which defines how sensor data is transmitted to and received from a central gateway. Key steps in establishing communication include:

- **Joining a LoRaWAN Network:** Register the LoRa module with a LoRaWAN network server provided by a network operator or deploy a private LoRaWAN network using a gateway.
- **Data Transmission:** Define data formats (payloads) and message rates according to the application's needs. LoRa supports bi-directional communication, allowing both sensor data transmission and control commands to be sent back to the sensors.

Data Transmission and Reception

1. **Transmission Process:** Sensors collect data at predefined intervals or based on triggers (e.g., thresholds). The microcontroller processes this data and transmits it through the LoRa module at scheduled intervals or in response to specific events.
2. **Reception at Gateway:** LoRa gateways receive transmissions from multiple LoRa modules within their range and forward the data to a network server via Ethernet, Wi-Fi, or cellular connection.
3. **Data Handling and Analysis:** At the network server, data packets are decrypted, processed, and stored in a database or forwarded to an application server for further analysis. Cloud-based platforms or edge computing devices often handle data aggregation, analytics, and visualization.

Message Types in LoRaWAN

The different message types used in LoRaWAN 1.0.x and 1.1. These message types are used to transport MAC commands and application data.

- Uplink and downlink messages.
- MAC Message types and their uses.
- Sending MAC commands in the FOpts field.
- Sending MAC commands and application data in the FRMPayload field.
- Keys used to encrypt each field that carries MAC Commands and application data.
- Keys used to calculate the Message Integrity Code (MIC) of each message.

Uplink and Downlink Messages

LoRa messages can be divided into uplink and downlink messages based on the direction they travel.

- **Uplink messages** - Uplink messages are sent by end devices to the Network Server relayed by one or many gateways. If the uplink message belongs to the Application Server or the Join Server, the Network server forwards it to the correct receiver.
- **Downlink messages** - Each downlink message is sent by the Network Server to only one end device and is relayed by a single gateway. This includes some messages initiated by the Application Server and the Join Server too.

Join-request

- The Join-request message is always initiated by an end device and sent to the Network Server.
- In LoRaWAN versions earlier than 1.0.4 the Join-request message is forwarded by the Network Server to the Application Server. In LoRaWAN 1.1 and 1.0.4+, the Network Server forwards the Join-request message to the device's Join Server.
- The Join-request message is not encrypted.

Join-accept

- In LoRaWAN versions **earlier** than 1.0.4 the Join-accept message is generated by the Application Server. In LoRaWAN 1.1 and 1.0.4+ the Join-accept message is generated by the Join Server. In both cases the message passes through the Network Server. Then the Network Server routes the Join-accept message to the correct end-device.
- The Join-accept message is encrypted as follows.
 - In LoRaWAN 1.0, the Join-accept message is encrypted with the AppKey.
 - In LoRaWAN 1.1, the Join-accept message is encrypted with different keys as shown in the table below.

If triggered by	Encryption Key
Join-request	NwkKey
Rejoin-request type 0, 1, and 2	JSEncKey

Rejoin-request

- The Rejoin-request message is always initiated by an end device and sent to the Network Server.
- There are three types of Rejoin-request messages: Type 0, 1, and 2.
- These message types are used to initialize the new session context for the end device. For the Rejoin-request message, the network replies with a Join-accept message.

Data Transmission and Payload Formats

Introduction to Data Transmission

Data transmission is the process of sending digital or analog data over a communication channel from a source to a destination. It is fundamental to modern telecommunications, networking, and computer systems. Efficient data transmission involves considerations such as speed, reliability, and the format in which data is encoded and decoded.

Components of Data Transmission

1. **Sender and Receiver:** The sender initiates the transmission by encoding data into signals suitable for the transmission medium. The receiver decodes these signals back into usable data.
2. **Transmission Medium:** This can be wired (e.g., copper cables, fiber optics) or wireless (e.g., radio waves, infrared). Each medium has its advantages and limitations in terms of bandwidth, distance, and susceptibility to interference.
3. **Protocols and Standards:** These define rules and procedures for data transmission, ensuring compatibility between different systems. Examples include TCP/IP (Transmission Control Protocol/Internet Protocol) for the internet and USB (Universal Serial Bus) for peripherals.

Types of Data Transmission

1. **Serial vs. Parallel Transmission:**
 - **Serial Transmission:** Sends data one bit at a time over a single channel, suitable for long-distance communication due to reduced interference.
 - **Parallel Transmission:** Sends multiple bits simultaneously over separate channels, typically faster but more susceptible to timing issues over longer distances.
2. **Analog vs. Digital Transmission:**
 - **Analog Transmission:** Uses continuous signals to represent data, suitable for voice and audio transmissions.
 - **Digital Transmission:** Uses discrete, binary signals (0s and 1s) for transmitting data, offering higher fidelity and less susceptibility to noise.

Payload Formats

Payload refers to the actual data being transmitted, distinct from the overhead (such as headers and error-checking codes) necessary for transmission and reception. The format of payload data depends on the application and the type of data being transmitted. Common payload formats include:

1. **Text-based Formats:**
 - **ASCII (American Standard Code for Information Interchange):** A widely used format for encoding text characters using 7 or 8 bits per character.
 - **Unicode:** Supports a wider range of characters (including non-Latin scripts) using variable bit lengths per character.
2. **Binary Formats:**
 - **Binary Coded Decimal (BCD):** Represents numeric data where each decimal digit is encoded in 4 bits.
 - **Floating Point:** Represents real numbers using a sign bit, exponent, and mantissa, allowing for a wide range of values with varying precision.
3. **Image and Multimedia Formats:**
 - **JPEG (Joint Photographic Experts Group):** Compresses photographic images using lossy compression, balancing image quality and file size.
 - **MP3 (MPEG-1 Audio Layer 3):** Compresses audio data by discarding less perceptible sounds, maintaining high fidelity with reduced file size.
4. **Structured Data Formats:**
 - **XML (Extensible Markup Language):** Represents hierarchical data structures using tags, facilitating data interchange between different systems.
 - **JSON (JavaScript Object Notation):** Lightweight format for storing and exchanging data objects, widely used in web APIs.

Data Messages

- There are 4 data message types used in both LoRaWAN 1.0.x and 1.1. These data message types are used to transport both MAC commands and application data which can be combined together in a single message.
- Data messages can be confirmed or unconfirmed. Confirmed data messages must be acknowledged by the receiver whereas unconfirmed data messages do not need to be acknowledged by the receiver.

- A data message is constructed as shown below:



Fig.: PHY layer

- MAC payload of the data messages consists of a frame header (FHDR) followed by an optional port field (FPort) and an optional frame payload (FRM Payload).

Error Detection and Correction

Ensuring data integrity during transmission involves error detection and correction mechanisms:

- **Checksums:** Summarize data using mathematical algorithms to detect errors caused by noise or transmission faults.
- **Parity Bits:** Append an additional bit to data bytes to detect single-bit errors, commonly used in serial communication.
- **Forward Error Correction (FEC):** Adds redundancy to data before transmission, allowing the receiver to correct errors without retransmission.

Security in IoT

The Internet of Things (IoT) refers to the network of interconnected devices that communicate and share data over the internet, often without direct human intervention. These devices range from consumer products like smart thermostats and wearables to industrial machinery and critical infrastructure components.

Challenges in IoT Security

1. **Heterogeneity of Devices:** IoT encompasses a vast array of devices with different capabilities, architectures, and levels of security.
2. **Resource Constraints:** Many IoT devices have limited processing power, memory, and energy resources, making it challenging to implement robust security measures.
3. **Data Privacy:** IoT devices collect and transmit sensitive data, raising concerns about data privacy and unauthorized access.
4. **Lack of Standards:** There is a lack of universally adopted standards for IoT security, leading to inconsistencies in implementation across devices and platforms.

Common Security Threats in IoT

1. **Unauthorized Access:** Attackers may exploit vulnerabilities to gain unauthorized access to IoT devices or networks.
2. **Data Interception:** As data travels between IoT devices and the cloud, it may be intercepted and manipulated by malicious actors.
3. **Denial-of-Service (DoS) Attacks:** Attackers can overwhelm IoT devices or networks with traffic, causing disruption or downtime.
4. **Device Compromise:** Vulnerable IoT devices can be compromised and used as entry points into larger networks.

Security Measures in IoT

1. **Authentication and Authorization:** Strong authentication mechanisms (e.g., passwords, biometrics) ensure that only authorized entities can access IoT devices and data.
2. **Encryption:** Data encryption (both at rest and in transit) protects sensitive information from being intercepted and read by unauthorized parties.
3. **Secure Boot and Firmware Updates:** Ensuring that only trusted software can run on IoT devices and timely application of security patches are crucial for mitigating vulnerabilities.
4. **Network Segmentation:** Segregating IoT devices into separate networks or VLANs can limit the impact of a security breach.
5. **Monitoring and Incident Response:** Continuous monitoring of IoT devices and networks for suspicious activities allows for early detection and response to security incidents.

LoRaWAN Security

LoRaWAN is a Low Power Wide Area Network (LPWAN) protocol designed for long-range communication between IoT devices and gateways. It operates on unlicensed radio frequency bands, making security considerations paramount.

Key Security Features of LoRaWAN

1. **Encryption:** LoRaWAN utilizes strong end-to-end encryption to protect data transmitted between devices and application servers. AES-128 encryption is used for payload encryption, ensuring data confidentiality.
2. **Authentication:** Devices and gateways must authenticate each other before communication begins. This mutual authentication prevents unauthorized devices or gateways from participating in the network.
3. **Device Keys:** Each LoRaWAN device has unique keys (e.g., AppKey, DevEUI, AppEUI) used for authentication and encryption. These keys are securely provisioned during device provisioning.
4. **Message Integrity:** LoRaWAN uses Message Integrity Codes (MIC) to ensure that data payloads are not tampered with during transmission. MIC provides data integrity and detects any unauthorized modifications.

5. **Join Procedure:** Devices undergo a secure join procedure to establish a secure session with the network server. This procedure involves cryptographic challenges and responses to prevent replay attacks.

Security Keys

Security keys are the basis of maintaining data and device integrity. LoRaWAN relies on a unique security key for each end-device to minimize the damage of a stolen key. Therefore, it is not acceptable to use the same root key on multiple devices. Additionally, root keys cannot be based on DevEUI or any other easily-guessed scheme. Keys should not be all zeros or all ones.

To obtain security keys, use a state-of-the-art cryptographic process that allows minimal transport of keys in plain text. Nonces and other methods used to generate keys should vary based on blocks of devices. For example, you can change the inputs, such as nonce values, for key generation every 64,000 devices. You can also use an alternate JoinEUI to ease the burden of looking up the exact input needed to regenerate the key, if required. LoRaWAN 1.0 specifies a number of security keys: NwkSKey, AppSKey and AppKey. All keys have a length of 128 bits. The algorithm used for this is AES-128, similar to the algorithm used in the 802.15.4 standard.

Session Keys

- When a device joins the network (this is called a join or activation), an application session key AppSKey and a network session key NwkSKey are generated.
- The NwkSKey is shared with the network, while the AppSKey is kept private. These session keys will be used for the duration of the session.

Network Session Key

- The Network Session Key (NwkSKey) is used for interaction between the Node and the Network Server.
- This key is used to validate the integrity of each message by its Message Integrity Code (MIC check). This MIC is similar to a checksum, except that it prevents intentional tampering with a message. For this, LoRaWAN uses AES-CMAC.
- In the backend of The Things Network this validation is also used to map a non-unique device address (DevAddr) to a unique DevEUI and AppEUI.

Application Session Key

- The Application Session Key (AppSKey) is used for encryption and decryption of the payload.
- The payload is fully encrypted between the Node and the Handler/Application Server component of The Things Network (which you will be able to run on your own server). This means that nobody except you is able to read the contents of messages you send or receive.

These two session keys (NwkSKey and AppSKey) are unique per device, per session. If you dynamically activate your device (OTAA), these keys are re-generated on every activation. If you statically activate your device (ABP), these keys stay the same until you change them.

Application Key

- The application key (AppKey) is only known by the device and by the application.
- Dynamically activated devices (OTAA) use the Application Key (AppKey) to derive the two session keys during the activation procedure.
- In The Things Network you can have a default AppKey which will be used to activate all devices, or customize the AppKey per device.

Frame Counters

- Replay attacks can be detected and blocked using frame counters. When a device is activated, these frame counters (FCntUp and FCntDown) are both set to 0. Every time the device transmits an uplink message, the FCntUp is incremented and every time the network sends a downlink message, the FCntDown is incremented. If either the device or the network receives a message with a frame counter that is lower than the last one, the message is ignored.
- This security measure has consequences for development devices, which often are statically activated (ABP). When you do this, you should realize that these frame counters reset to 0 every time the device restarts (when you flash the firmware or when you unplug it). As a result, The Things Network will block all messages from the device until the FCntUp becomes higher than the previous FCntUp. Therefore, you should re-register your device in the backend every time you reset it.

End Device Activation

- Every end device must be registered with a network before sending and receiving messages. This procedure is known as **activation**.
- There are two activation methods available:
 - **Over-The-Air-Activation (OTAA)** - the most secure and recommended activation method for end devices. Devices perform a join procedure with the network, during which a dynamic device address is assigned and security keys are negotiated with the device.
 - **Activation By Personalization (ABP)** - requires hardcoding the device address as well as the security keys in the device. ABP is **less secure** than OTAA and also has the downside that devices can not switch network providers without manually changing keys in the device.
- The join procedure for LoRaWAN 1.0.x and 1.1 is slightly different.

What is OTA (Over-the-Air Update)

OTA, or Over-the-Air update, refers to the process of remotely updating the firmware or software of devices in the field, without the need for physical access. This capability is crucial for managing and maintaining IoT devices, enabling updates for security patches, feature enhancements, and bug fixes.

OTA in LoRaWAN 1.0.x

In the context of LoRaWAN 1.0.x, OTA is particularly important given the deployment of numerous devices across large geographic areas. The protocol supports OTA updates to ensure devices can receive critical updates while minimizing power consumption and maintaining long-range connectivity.

Mechanism of OTA Updates in LoRaWAN 1.0.x

1. Firmware and Software Management:

- **Firmware** refers to the low-level control software that operates the hardware of the device.
- **Software** encompasses higher-level applications and services running on the device.
- OTA updates can target both firmware and software, allowing comprehensive management of the device.

2. Network Components:

- **End Devices**: These are the IoT devices equipped with LoRa transceivers.
- **Gateways**: Devices that relay messages between end devices and the network server.
- **Network Server**: Manages network traffic, ensuring secure data transmission and device updates.
- **Application Server**: Hosts the applications that interact with the end devices.

3. OTA Update Process:

- **Initiation**: The update process is usually initiated by the application server, which decides that an update is necessary based on version checks, schedules, or security needs.
- **Notification**: The application server notifies the end device about the availability of a new update. This notification is typically sent as a downlink message.
- **Download**: The end device downloads the update. Given the limited data rates and power constraints, this process may involve downloading the update in small chunks over multiple transmission sessions.
- **Verification**: The integrity of the downloaded update is verified through checksums or digital signatures to ensure it has not been corrupted or tampered with during transmission.
- **Installation**: Once verified, the update is installed. This may involve writing the new firmware or software to the device's memory and rebooting the device.
- **Rollback Mechanism**: To ensure reliability, devices typically include a rollback mechanism that reverts to the previous firmware if the update fails or causes issues.

4. Security Aspects:

- **Encryption**: Data transmitted during the OTA update process is encrypted to prevent interception and tampering.
- **Authentication**: Devices authenticate update packages to ensure they originate from a trusted source.
- **Integrity Checks**: Use of cryptographic hashes and digital signatures to validate the integrity of the update files.

5. Power Management:

- **Duty Cycle:** LoRaWAN devices often operate under strict duty cycle regulations to limit transmission time, conserving power and adhering to regional regulations.
- **Adaptive Data Rate (ADR):** LoRaWAN uses ADR to optimize data rates, airtime, and energy consumption. This is crucial during OTA updates to balance the power consumption and update speed.

6. Challenges and Considerations:

- **Fragmentation:** OTA updates may need to be fragmented into smaller pieces due to payload size limitations.
- **Reliability:** Ensuring updates are received correctly and applied without bricking devices is critical.
- **Bandwidth:** Limited bandwidth in LoRaWAN networks can slow down the update process, requiring careful management of update scheduling and delivery.

7. LoRaWAN 1.0.x Specifics:

- **Versioning:** LoRaWAN 1.0.x has its own specifications for version management and compatibility between different versions of the protocol.
- **Class A, B, C Devices:** Different classes of devices have different behaviors in terms of receiving downlink messages. Class A devices only open a receive window after an uplink transmission, Class B devices have scheduled receive windows, and Class C devices are almost always in receive mode, influencing how OTA updates are managed.

Over The Air Activation in LoRaWAN 1.0.x

- In LoRaWAN 1.0.x, the join procedure requires two MAC messages to be exchanged between the end device and the Network Server:
 - Join-request - from end device to the Network Server
 - Join-accept - from Network Server to the end device
- Before activation, the AppEUI, DevEUI, and AppKey should be stored in the end device.
- The AppKey is an AES-128 bit secret key known as the root key. The same AppKey should be provisioned onto the network where the end device is going to register.
- The AppEUI and DevEUI are not secret and are visible to everyone.
- The AppKey is never sent over the network.
- The following steps describe the Over-The-Air-Activation (OTAA) procedure

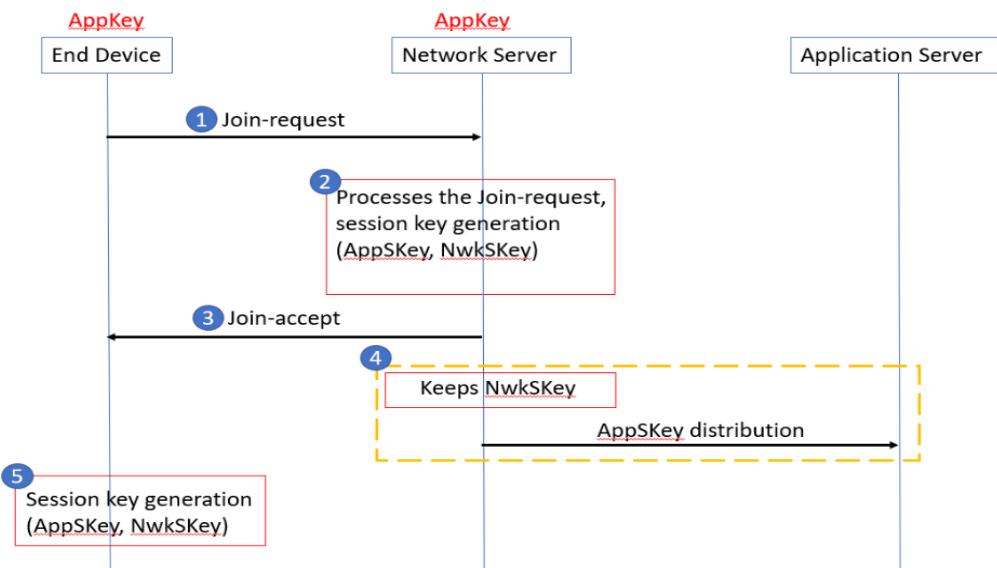


Fig.: OTAA message flow in LoRaWAN 1.0

Over-The-Air-Activation in LoRaWAN 1.1

- In LoRaWAN 1.0.x, the join procedure requires two MAC messages to be exchanged between the end device and the Join Server:
 - Join-request - from end device to the Join Server
 - Join-accept - from Join Server to the end device
- Before activation, the JoinEUI, DevEUI, AppKey, and NwkKey should be stored in the end device. The AppKey and NwkKey are AES-128 bit secret keys known as root keys.
- The matching AppKey, NwkKey, and DevEUI should be provisioned onto the Join Server that will assist in the processing of the join procedure and session key derivation. The JoinEUI and DevEUI are not secret and visible to everyone.
- The AppKey and NwkKey are never sent over the network.

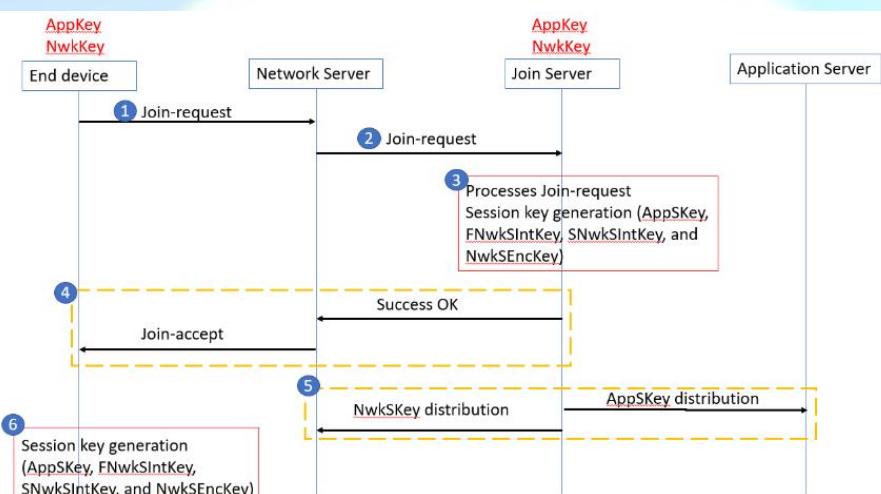


Fig.: OTAA message flow in LoRaWAN 1.1

Activation By Personalization

- Activation By Personalization (ABP) directly ties an end-device to a pre-selected network, bypassing the over-the-air-activation procedure.
- Activation by Personalization is the less secure activation method, and also has the downside that devices can not switch network providers without manually changing keys in the device. A Join Server is not involved in the ABP process.
- An end device activated using the ABP method can only work with a single network and keeps the same security session for its entire lifetime.

Activation By Personalisation in LoRaWAN 1.0.x

The **DevAddr** and the two session keys **NwkSKey** and **AppSKey** are directly stored into the end-device instead of the DevEUI, AppEUI, and the AppKey. Each end device should have a unique set of NwkSKey and AppSkey. The same **DevAddr** and **NwkSKey** should be stored in the Network Server and the **AppSKey** should be stored in the Application Server



Fig.: Pre-sharing DevAddr and session keys for ABP in LoRaWAN 1.0

Activation By Personalisation in LoRaWAN 1.1

The DevAddr and the four-sessionkeys

FNwkSIntKey, SNwkSIntKey, NwkSEncKey, and AppSKey are directly stored into the end device instead of the DevEUI, JoinEUI, AppKey, and NwkKey. The same DevAddr, FNwkSIntKey, SNwkSIntKey, and NwkSEncKey should be stored in the Network Server and the and AppSKey should be stored in the Application Server.

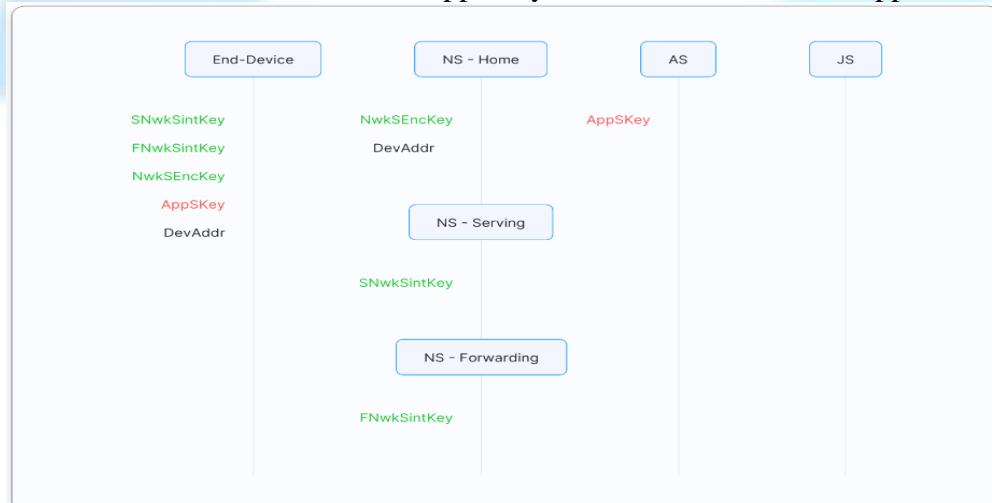


Fig.: Activation By Personalisation in LoRaWAN 1.1

Security Best Practices for LoRaWAN

1. **Key Management:** Implement robust key management practices to securely store, distribute, and rotate encryption keys.
2. **Secure Deployment:** Ensure that LoRaWAN devices are deployed in physically secure locations and are protected against tampering or unauthorized access.
3. **Network Monitoring:** Continuously monitor LoRaWAN traffic and device behavior for anomalies that may indicate security breaches.
4. **Firmware Updates:** Regularly update device firmware to patch known vulnerabilities and improve overall security posture.

Encryption: Protecting Data from Prying Eyes

Definition and Purpose: Encryption is the process of encoding information in such a way that only authorized parties can access it. It transforms plaintext (readable data) into ciphertext (encoded data) using mathematical algorithms and keys. The primary goal of encryption is confidentiality — ensuring that even if intercepted, the data remains unreadable to unauthorized users.

Types of Encryption:

1. **Symmetric Encryption:** In symmetric encryption, the same key is used for both encryption and decryption. Algorithms like AES (Advanced Encryption Standard) and DES (Data Encryption Standard) are widely used due to their efficiency in securing data within closed systems.
2. **Asymmetric Encryption (Public-Key Encryption):** Asymmetric encryption uses a pair of keys: a public key for encryption and a private key for decryption. This method, exemplified by RSA (Rivest-Shamir-Adleman) and Elliptic Curve Cryptography (ECC), facilitates secure communication over open networks like the internet.

Encryption Algorithms and Strengths: The effectiveness of encryption depends on the strength of the algorithm and the length of the encryption keys. Modern algorithms employ complex mathematical operations that make brute-force attacks impractical within reasonable timeframes.

Applications of Encryption: Encryption is essential in various domains, including:

- **Data Storage:** Encrypting files and databases to protect sensitive information.
- **Communications:** Securing email, messaging apps, and online transactions.
- **Regulatory Compliance:** Meeting data protection standards such as GDPR and HIPAA.

Authentication: Verifying Identities and Ensuring Trust

Definition and Importance: Authentication is the process of verifying the identity of a user or system entity. It ensures that the sender or recipient of data is who they claim to be, thereby establishing trust in digital interactions. Authentication mechanisms prevent unauthorized access and protect against impersonation and fraud.

Types of Authentication:

1. **Single-Factor Authentication (SFA):** SFA relies on a single verification method, such as a password, PIN, or biometric scan (fingerprint, facial recognition). While straightforward, SFA is vulnerable to brute-force attacks and phishing.
2. **Multi-Factor Authentication (MFA):** MFA combines two or more authentication factors (typically something you know, have, or are) to enhance security. This could involve a password, a code sent to a mobile device (something you have), and a biometric scan (something you are).

Authentication Protocols:

- **OAuth:** Used for delegated authorization, enabling users to grant third-party applications limited access to their resources.
- **OpenID Connect:** An identity layer built on OAuth 2.0, providing authentication and user profile information.

Biometric Authentication: Leveraging unique biological traits for identity verification, biometric methods offer enhanced security and user convenience. Technologies such as fingerprint recognition, iris scanning, and facial recognition are increasingly integrated into devices and systems.

Integration and Future Trends

Integration of Encryption and Authentication: Combining encryption with robust authentication mechanisms forms the cornerstone of secure systems. Data encrypted with strong algorithms remains protected even if intercepted, while rigorous authentication ensures only authorized entities can decrypt and access it.

Emerging Trends:

- **Post-Quantum Cryptography:** Addressing vulnerabilities posed by quantum computers to current encryption standards.
- **Zero Trust Security:** Moving beyond perimeter-based security models to verify and authenticate every access request.
- **Homomorphic Encryption:** Allowing computations on encrypted data without decrypting it first, enabling secure data processing in the cloud.

Best Practices for Securing LoRa Networks

1. Key Management

Proper key management is essential for maintaining the confidentiality and integrity of data. Keys should be securely stored and rotated periodically to reduce the impact of potential key compromise.

2. Secure Join Process

LoRaWAN specifies a secure join process where devices authenticate themselves to the network server using pre-shared keys. Implementing this process ensures that only authorized devices can join the network.

3. Device Management

Maintain visibility and control over devices throughout their lifecycle. Implement secure provisioning, firmware updates, and remote management capabilities to address vulnerabilities promptly and efficiently.

4. Network Segmentation

Segmenting the network can limit the impact of a security breach by containing it within specific areas. Use firewalls and access control policies to restrict unauthorized access between network segments.

5. Monitoring and Logging

Continuous monitoring of network traffic and device behavior can help detect anomalies indicative of security incidents. Log and analyze events to identify potential threats and respond proactively.

6. Physical Security

Protect devices and gateways from physical tampering or theft. Deploy them in secure locations and use tamper-resistant hardware where necessary to prevent unauthorized access.

7. Regulatory Compliance

Ensure compliance with relevant regulations and standards governing IoT and wireless communications. Compliance frameworks such as GDPR, HIPAA, or industry-specific standards provide guidelines for securing data and maintaining privacy.

LoRa Network Deployment

Network Deployment Considerations

Deploying a LoRa network involves several considerations to ensure reliable and efficient operation:

- **Coverage Planning:** Conducting a site survey to determine optimal gateway locations based on terrain, obstacles, and desired coverage area.
- **Gateway Placement:** Strategically placing gateways to maximize coverage overlap while minimizing interference and ensuring sufficient signal strength at all points of interest.
- **Antenna Selection:** Choosing antennas based on the frequency band and desired coverage pattern (e.g., omnidirectional vs. directional antennas).
- **Network Architecture:** Deciding between public LoRaWAN networks (managed by third-party providers) and private deployments (self-managed networks with greater control over security and customization).
- **Power Considerations:** Optimizing power consumption for end devices to maximize battery life, often achieved through adaptive data rate (ADR) algorithms that adjust transmission parameters based on signal conditions.

Practical Applications of LoRa Networks

LoRa technology finds applications across various industries and use cases:

- **Smart Cities:** Monitoring environmental parameters (air quality, noise levels) and managing utilities (water metering, street lighting).
- **Agriculture:** Precision farming through soil moisture monitoring, weather stations, and livestock tracking.
- **Industrial IoT:** Asset tracking, predictive maintenance, and remote monitoring of equipment and machinery.
- **Logistics and Supply Chain:** Tracking shipments in real-time, monitoring temperature-sensitive goods, and optimizing route planning.

Steps to Setting Up a LoRa Network

Planning Phase

Identify Use Case

1. **Define Application Requirements:** Determine the specific needs of your application (e.g., smart agriculture, smart city infrastructure monitoring, industrial IoT). This includes understanding data requirements, device types (sensors, actuators), and expected communication patterns.
2. **Define Use Case Scenarios:** Identify typical use scenarios to understand the volume of data, frequency of transmissions, and criticality of data delivery. This helps in designing the network architecture and selecting appropriate hardware.

Coverage Planning

1. **Conduct Site Survey:** Perform a physical survey of the deployment area to assess terrain, buildings, and potential obstacles that could affect radio frequency (RF) propagation.
2. **Gateway Placement Strategy:** Determine optimal locations for gateways based on coverage requirements. Factors to consider include:
 - **Line-of-Sight:** Prefer locations with clear line-of-sight to maximize range.
 - **Antenna Height:** Higher placement improves coverage.
 - **Density:** Ensure adequate overlap between coverage areas to avoid dead zones.
3. **Gateway Density Calculation:** Estimate the number of gateways needed based on coverage area size and density of deployed end devices.

Select Frequency Band

1. **Regulatory Considerations:** Choose from available ISM bands (433 MHz, 868 MHz, 915 MHz) based on local regulations and spectrum availability.
2. **Propagation Characteristics:** Consider environmental factors such as building materials, foliage, and weather conditions that can affect signal propagation at different frequencies.

Implementation Phase

Hardware Selection

1. **End Devices (Nodes):**
 - Choose LoRa-compatible sensors or actuators that match the requirements of your application (e.g., temperature sensors, motion detectors).
 - Consider factors such as power consumption, communication range, and sensor data formats.
2. **Gateways:**
 - Select LoRa gateways that support the chosen frequency band and have sufficient capacity to handle anticipated traffic.
 - Ensure compatibility with network server software and protocols (e.g., LoRaWAN).
3. **Network Servers:**
 - Choose a network server that integrates well with your selected gateways and application servers.
 - Configure server settings including encryption keys, data rates, and device management policies.

Gateway Deployment

1. **Installation Planning:**
 - Install gateways in planned locations identified during the site survey phase.
 - Ensure stable power supply and secure mounting to prevent tampering or damage.
2. **Configuration and Connectivity:**
 - Configure gateways to connect to the network server using appropriate backhaul options (Ethernet, Wi-Fi, cellular).
 - Verify connectivity and data forwarding to the network server.

End Device Integration

1. **Sensor/Actuator Integration:**
 - o Integrate sensors or actuators with LoRa nodes according to manufacturer guidelines.
 - o Configure node parameters such as data transmission intervals and payload formats.
2. **Gateway-Node Communication:**
 - o Test communication between nodes and gateways to ensure data transmission reliability and integrity.

Testing and Optimization

Functional Testing

1. **End-to-End Communication:**
 - o Conduct tests to verify data transmission from end devices through gateways to the network and application servers.
 - o Validate data reception and processing at each network layer.

Performance Optimization

1. **Parameter Tuning:**
 - o Optimize transmission parameters such as data rate, spreading factor, and transmit power to achieve optimal network performance.
 - o Adjust antenna placement and orientation to improve coverage and signal strength.

Security Testing

1. **Security Audits:**
 - o Perform vulnerability assessments and penetration testing to identify and mitigate potential security risks.
 - o Implement encryption mechanisms and secure authentication protocols to protect data integrity and privacy.

Maintenance and Scaling

Monitoring and Maintenance

1. **Network Monitoring:**
 - o Deploy monitoring tools to continuously track network performance, including gateway uptime, packet loss rates, and data throughput.
 - o Establish alerts for network anomalies or equipment failures.
2. **Regular Maintenance:**
 - o Schedule routine maintenance tasks such as firmware updates for gateways and end devices to ensure optimal performance and security.

Scaling

1. Expansion Strategy:

- Plan for network expansion by adding more gateways and nodes as the application scales.
- Upgrade network infrastructure to handle increased data traffic and device density while maintaining performance standards.

Programming LoRa Modules

Programming LoRa modules involves configuring communication parameters such as frequency, spreading factor, bandwidth, and coding rate. These parameters affect the range, data rate, and power consumption of the communication. Here's a step-by-step guide on how to program LoRa modules:

1. Setting Up Development Environment

LoRa Module Selection:

Choose a LoRa module based on your project's requirements. Popular choices include the SX127x series (e.g., SX1276, SX1278) by Semtech, which are widely supported and provide flexibility in terms of frequency bands and features.

Microcontroller:

Select a microcontroller compatible with your chosen LoRa module. Commonly used microcontrollers include Arduino boards (with LoRa shields or integrated LoRa modules) and STM32 microcontrollers, which offer higher processing power and more advanced features suitable for complex applications.

Integrated Development Environment (IDE):

Use an IDE like Arduino IDE, PlatformIO, or STM32CubeIDE for writing, compiling, and uploading code to the microcontroller. These environments provide libraries, examples, and tools necessary for LoRa development.

2. Installing Libraries and Drivers

LoRa Library:

For Arduino, utilize libraries such as:

- **LoRa library by Sandeep Mistry:** Provides a straightforward interface for configuring and using LoRa modules.
- **RadioHead library by AirSpayce:** Offers more advanced features and compatibility with various radio transceivers, including LoRa modules.

Include the library in your project to access functions that handle LoRa communication protocols efficiently.

Driver Installation:

Ensure that the appropriate drivers for your microcontroller and LoRa module are installed correctly. Manufacturers typically provide documentation and software tools to facilitate driver installation and configuration.

3. Writing LoRa Code

Example Code (Arduino IDE):

```
#include <LoRa.h>

void setup() {
    Serial.begin(9600);
    while (!Serial);

    if (!LoRa.begin(915E6)) { // Set frequency to 915 MHz
        Serial.println("LoRa initialization failed. Check your connections.");
        while (true);
    }
    Serial.println("LoRa initialized successfully.");
}

void loop() {
    // Send packet
    String message = "Hello, LoRa!";
    LoRa.beginPacket();
    LoRa.print(message);
    LoRa.endPacket();
    Serial.println("Message sent.");
    delay(5000); // Wait 5 seconds before sending next message
}
```

4. Configuring LoRa Parameters

Key Parameters:

- **Frequency:** Set according to local regulations and regional frequency bands (e.g., 433MHz, 868MHz, 915MHz). Ensure compatibility with your LoRa module and gateway.
- **Spreading Factor:** Determines the trade-off between range and data rate. Higher spreading factors (e.g., SF12) provide longer range but lower data rates.
- **Bandwidth:** Defines the spectral width of the signal. Narrower bandwidths conserve power but may reduce data throughput.
- **Coding Rate:** Influences error correction capabilities. Higher coding rates enhance reliability but decrease overall throughput.

Adjust these parameters in your code based on environmental conditions, desired range, and power consumption requirements.

5. Testing and Deployment

Testing:

Verify LoRa communication by:

- Sending and receiving test messages between LoRa modules.
- Using tools like LoRaWAN packet analyzers to monitor signal strength, data integrity, and packet loss.

Deployment Considerations:

- **Antenna Placement:** Optimize antenna positioning for maximum signal strength and reliability.
- **Environmental Factors:** Consider factors such as terrain, obstructions, and interference sources that may affect LoRa communication performance.
- **Regulatory Compliance:** Adhere to local regulations regarding frequency bands and transmission power limits.

Building IoT Applications with LoRa



HARDWARE PROGRAMS(LoRA)

1. Home Automation and Building Automation

HARDWARE REQUIREMENTS:

- LoRa Gateway
- IO Controller
- Light
- Fan

OBJECTIVES:

Understanding LoRa Technology and Communication: Learn about the LoRa (Long Range) communication protocol and how it can be used for wireless communication over long distances with low power consumption.

Developing and Integrating IoT Devices: Understand the process of integrating various IoT devices and components such as lights and fans into a home automation system.

Creating a Centralized Control System: Create a centralized system to control multiple devices within a smart home setup.

Exploring Remote Monitoring and Control: Enable remote monitoring and control of home appliances.

Energy Efficiency and Automation: Explore how home automation can contribute to energy efficiency.

Security and Reliability: Ensure that the home automation system is secure and reliable.

Data Logging and Analysis: Collect and analyze data from the home automation system for insights and optimization.

IO CONTROLLER:



Features:

- STM32L072xxxx MCU
- SX1276/78 LoRa Wireless Chip
- LoRaWAN Class A & Class C protocol
- Optional Customized LoRa Protocol
- Bands: CN470/EU433/KR920/US915/EU868/AS923/AU915
- AT Commands to change parameters

Applications:

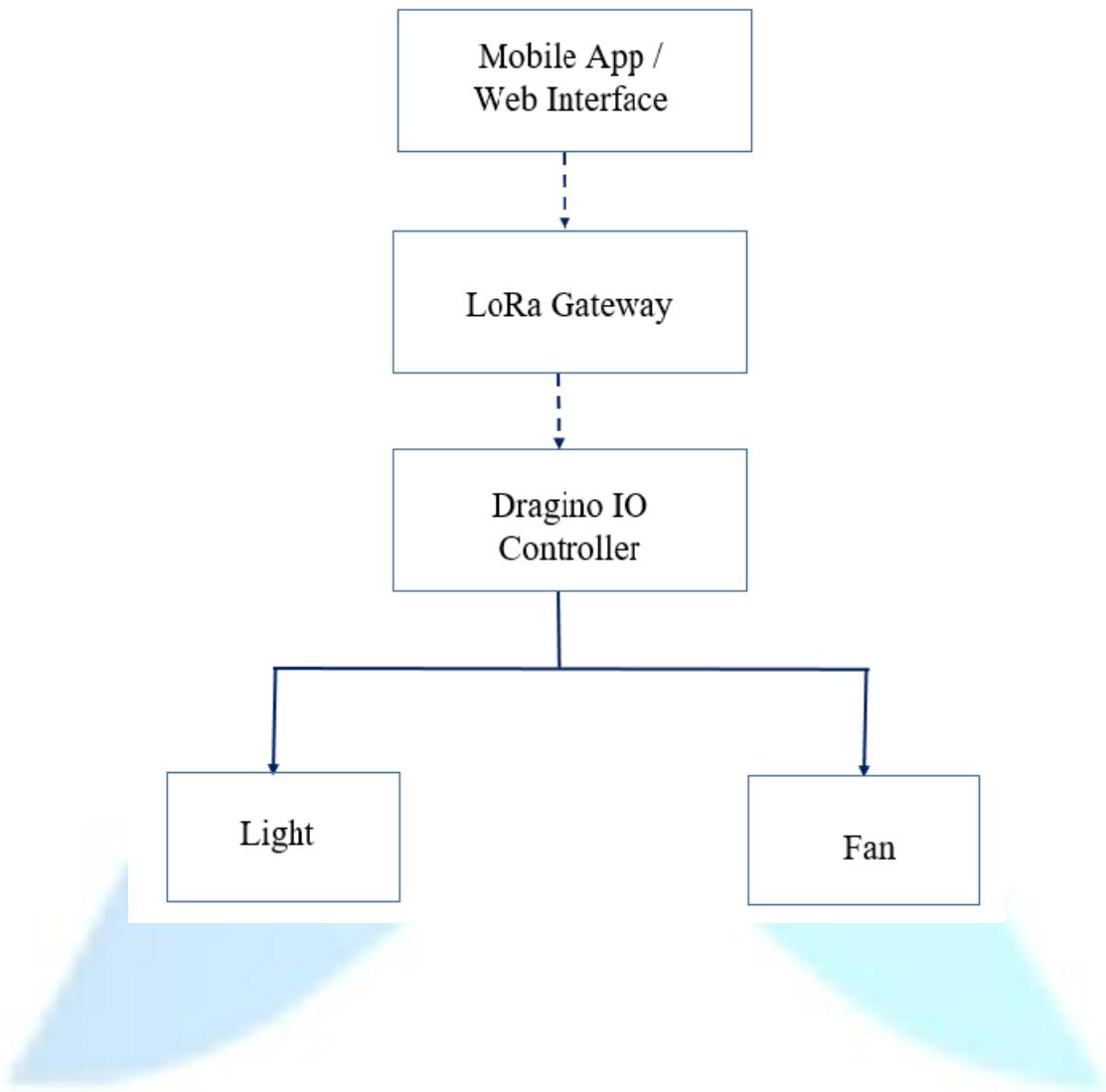
- Smart Buildings & Home Automation
- Logistics and Supply Chain Management
- Smart Metering
- Smart Agriculture
- Smart Cities
- Smart Factory

Interfaces:

- 2 x Digital Input (bi-direction)
- 2 x Digital Output
- 2 x Relay Output (5A@250VAC / 30VDC)
- 2 x 0~20mA Analog Input (res:0.01mA)

- 2 x 0~30V Analog Input (res:0.01v)
- Power Input 0~24V

BLOCK DIAGRAM



PROGRAM:

```
import paho.mqtt.publish as publish
import time

publish.single("v3/iot-io-controller@ttn/devices/eui-a8404129418379e7/down/push",
'{"downlinks": [{"f_port": 2, "frm_payload": "AwAA", "priority": "NORMAL"}]}',
hostname="au1.cloud.thethings.network", port=1883, auth={'username':'iot-io-
controller@ttn','password':'NNSXS.QQQKAVZLNKAVMJTCHV6KZKXJ6R2HMXWM2
YA2AEI.TXXEDUQUWXDV6DPIJE3WZJGIIZAOKVOTMXDOJTEA3UKQAMBJQPL
Q"})
```

PROCEDURE:

Step 1: Set Up the Hardware

1. LoRa Gateway:

- Install and configure your LoRa gateway following the manufacturer's instructions.
- Connect the gateway to the internet via Ethernet or Wi-Fi.

2. Dragino IO Controller:

- Connect the Dragino IO Controller to the LoRa network.
- Ensure the IO Controller is powered on and within the range of the LoRa gateway.

3. Light and Fan:

- Connect the light and fan to the Dragino IO Controller according to the controller's documentation.
- Make sure the connections are secure and the devices are properly powered.

Step 2: Set Up The Things Network (TTN)

1. Create a TTN Account:

- Sign up for an account at The Things Network.

2. Register Your LoRa Gateway:

- Log in to TTN Console.
- Register your LoRa gateway under the "Gateways" section.

3. Register Your Device (Dragino IO Controller):

- Register the Dragino IO Controller under the "Devices" section.

- Note down the Device EUI (eui-a8404129418379e7) and the Application ID (iot-io-controller@ttn).

Step 3: Configure MQTT

1. MQTT Configuration on TTN:

- In the TTN Console, navigate to your application.
- Enable the MQTT integration.
- Note down the MQTT broker address, port, username, and password.

Step 4: Write and Execute the MQTT Publish Script

1. Install Paho MQTT Library:

- If you haven't already, install the Paho MQTT library:

```
pip install paho-mqtt
```

2. Write the Publish Script:

- Use the provided script to send commands to the Dragino IO Controller. This example turns off the light and fan:

3. Run the Script:

- Execute the script to send the MQTT message:

Step 5: Verify Operation

1. Check Device Status:

- Observe the light and fan to ensure they turn off as commanded.

2. Monitor TTN Console:

- Check the TTN Console to see the downlink message being sent to the Dragino IO Controller.
- Verify that the device acknowledges the command.

Step 6: Extend Functionality (Optional)

1. Control Light and Fan Separately:

- Modify the frm_payload in your script to control the light and fan independently. For example, turning on the light:

```
# Adjust this based on your payload codes
frm_payload = "AwAB"  # Light ON
frm_payload = "AwEA"  # Fan ON
frm_payload = "AwEB"  # Both Fan and Light ON
frm_payload = "AwAA"  # Both Fan and Light OFF
```

2. Automate Control:

- Add logic to automate the control based on conditions or schedules using additional scripts or integration with home automation platforms (e.g., Home Assistant).

FLASK WEB APPLICATION CREATION CODE FOR HOME AUTOMATION:

```

from flask import Flask, render_template, redirect, url_for
import paho.mqtt.publish as publish
import sqlite3
app = Flask(__name__)
MQTT_HOST = "au1.cloud.thethings.network"
MQTT_PORT = 1883
MQTT_AUTH = {'username': "iot-io-controller@ttn", 'password':
"NNSXS.QQQKAVZLNKAVMJTCHV6KZKXJ6R2HMXWM2YA2AEI.TXXEDUQUWX
DV6DPIJE3WZJGIIZAOKVOTMXDOJTEA3UKQAMBJQPLQ"}
def publish_message(payload):
    publish.single("v3/iot-io-controller@ttn/devices/eui-a8404129418379e7/down/push",
    '{"downlinks": [{"f_port": 2, "frm_payload": "' + payload + '", "priority": "NORMAL"}]}',
    hostname=MQTT_HOST, port=MQTT_PORT, auth=MQTT_AUTH)

@app.route("/")
def index():
    return render_template('index.html')
@app.route("/light/<a>")
def light(a):
    con=sqlite3.connect("ioDataBase.db")
    con.execute("UPDATE iocontroller SET Light=? WHERE id=?", (a,1))
    con.commit()
    return redirect(url_for('status'))
@app.route("/fan/<a>")
def fan(a):
    con=sqlite3.connect("ioDataBase.db")
    con.execute("UPDATE iocontroller SET Fan=? WHERE id=?", (a,1))
    con.commit()
    return redirect(url_for('status'))
@app.route('/status')

```

```

def status():

    conn = sqlite3.connect("ioDataBase.db")

    cur = conn.execute("SELECT * FROM iocontroller WHERE id=?",(1,))

    status = cur.fetchone()

    if status[1]=='on' and status[2]=='off':

        payload='AwAB'

        publish_message(payload)

        msg1="Light is ON"

        msg2="Fan is OFF"

    elif status[1]=='on' and status[2]=='on':

        payload='AwEB'

        publish_message(payload)

        msg1 = "Light is ON"

        msg2 = "Fan is ON"

    elif status[1]=='off' and status[2]=='on':

        payload='AwEA'

        publish_message(payload)

        msg1 = "Light is OFF"

        msg2 = "Fan is ON"

    elif status[1]=='off' and status[2]=='off':

        payload='AwAA'

        publish_message(payload)

        msg1 = "Light is OFF"

        msg2 = "Fan is OFF"

    return render_template('index.html', msg1=msg1,msg2=msg2)

if __name__ == '__main__':
    app.run()

```

2. Agri IoT and Irrigation System

a. Interfacing of Soil NPK Sensor

HARDWARE REQUIREMENTS :

- LSNPK01 - Soil NPK Sensor
- LoRa Gateway
- Soil

OBJECTIVES:

- **Measure Soil Nutrient Levels:** Accurately measure the levels of Nitrogen, Phosphorus, and Potassium in the soil using the NPK sensor.
- **Transmit Data via LoRa:** Use LoRa technology to wirelessly transmit the sensor data to a LoRa Gateway for remote monitoring.
- **Real-time Data Monitoring:** Set up a system for real-time monitoring of soil nutrient levels to aid in precision agriculture.
- **Data Logging and Analysis:** Log the received data for further analysis and to make informed decisions about soil and crop management.

SOIL NPK SENSOR



Features :

- LoRaWAN 1.0.3 Class A
- Ultra-low power consumption
- Monitor Soil Nitrogen
- Monitor Soil Phosphorus
- Monitor Soil Potassium
- Monitor Battery Level
- Bands: CN470/EU433/KR920/US915/EU868/AS923/AU915/IN865
- AT Commands to change parameters
- Uplink on periodically
- Downlink to change configure
- IP66 Waterproof Enclosure
- IP68 rate for the Sensor Probe
- 8500mAh Battery for long term use

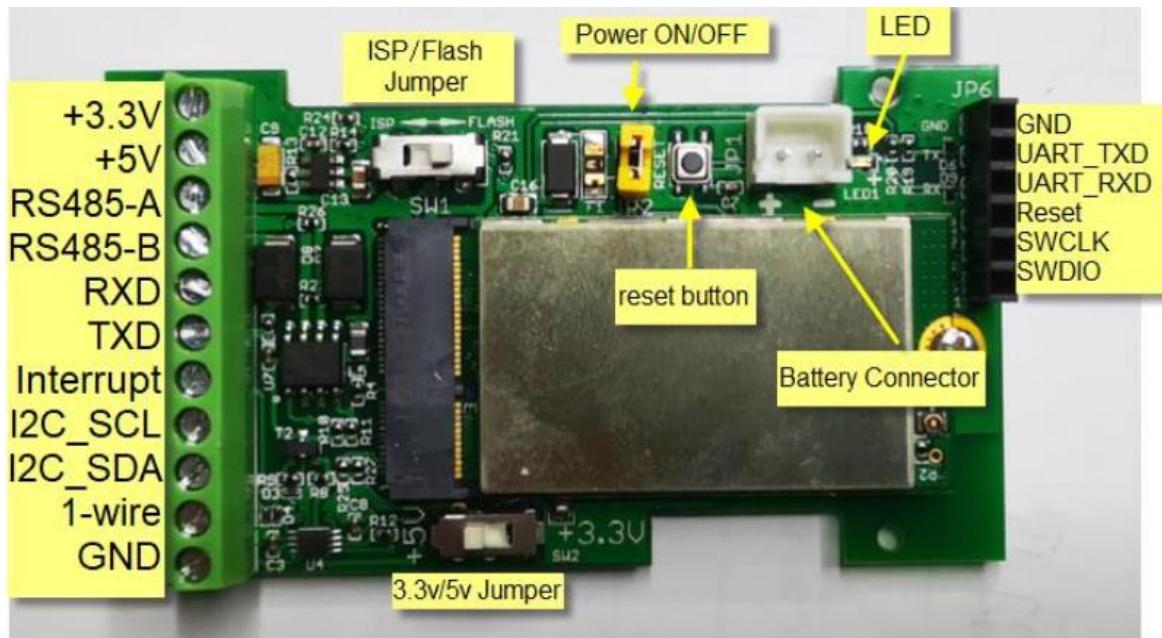
NPK Probe Specification

- Range 1-1999 mg/kg
- Resolution: 1 mg/kg
- Accuracy: $\pm 2\%$ FS
- Material: Stainless Steel Probe
- P68 Protection
- Length: 2 meters

Applications

- Smart Agriculture

Pin mapping and power on



Uplink Payload Format

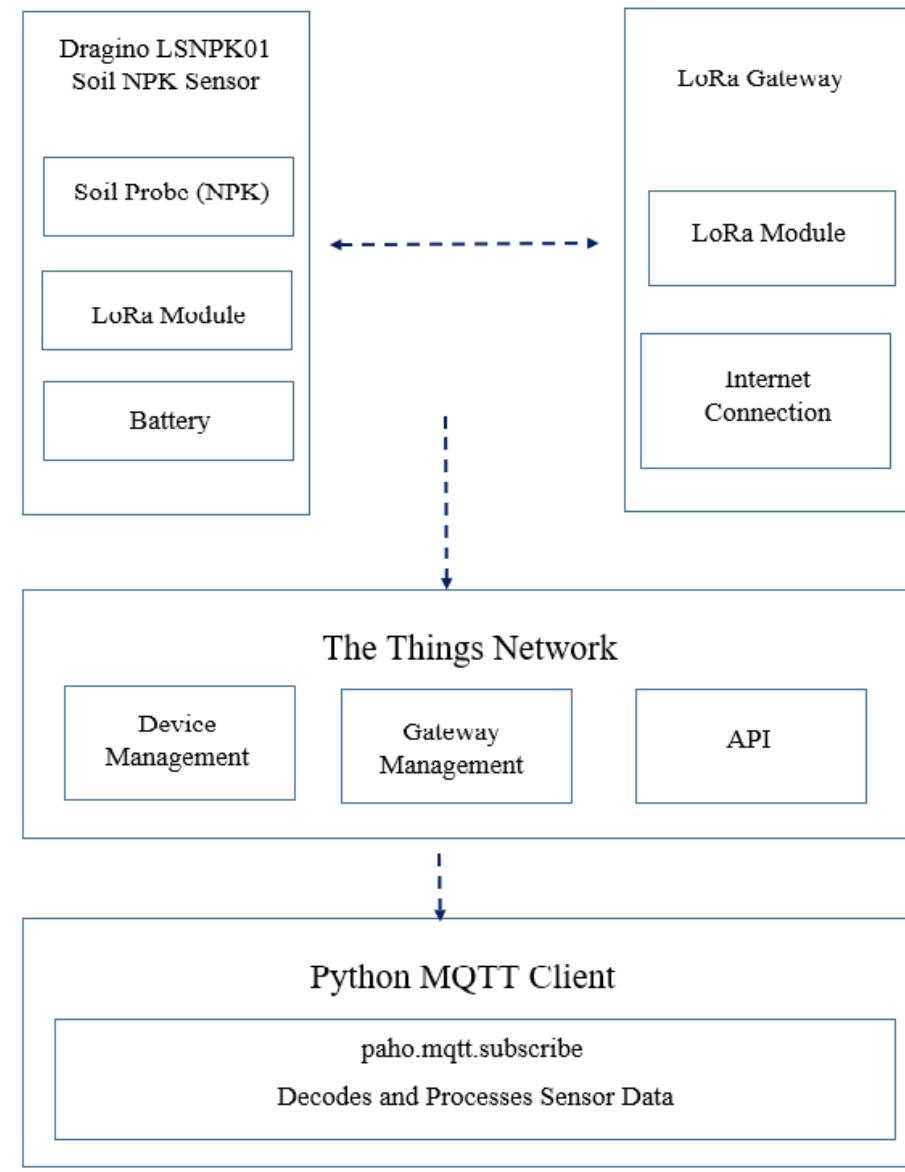
LSNPK01 - Soil NPK Sensor will uplink payload via LoRaWAN with below payload format:

Uplink payload includes in total 9 bytes.

Normal uplink payload:

Size(bytes)	2	1	2	2	2
Value	BAT	Digital Interrupt (Optional)	Soil Phosphorus	Soil Nitrogen	Soil Potassium

BLOCK DIAGRAM



PROGRAM:

Uplink Code:

```

function decodeUplink(input) {
    // Initialize the object to hold the decoded data
    var decoded = {};

    // Decode bytes from the payload
    var bytes = input.bytes;
  
```

```

// Check if the payload length matches the expected 9 bytes
if (bytes.length !== 9) {
    return {
        errors: ["Invalid payload length"]
    };
}

// Decode Battery value (bytes 0-1)
var battery = (bytes[0] << 8) | bytes[1];
decoded.Bat = battery / 1000; // Convert from millivolts to volts

// Decode Nitrogen (N) value (bytes 2-3)
var nitrogen = (bytes[5] << 8) | bytes[6];
// Apply scaling factor (example: divide by 10 to get value in percentage)
decoded.N_SOIL = nitrogen; // Adjust scaling factor as needed

// Decode Phosphorus (P) value (bytes 4-5)
var phosphorus = (bytes[3] << 8) | bytes[4];
// Apply scaling factor (example: divide by 10 to get value in percentage)
decoded.P_SOIL = phosphorus; // Adjust scaling factor as needed

// Decode Potassium (K) value (bytes 6-7)
var potassium = (bytes[7] << 8) | bytes[8];
// Apply scaling factor (example: divide by 10 to get value in percentage)
decoded.K_SOIL = potassium; // Adjust scaling factor as needed

// Decode interrupt (byte 8)
var interrupt = bytes[2];
decoded.Interrupt = interrupt; // No scaling applied to the interrupt value

decoded.device_id = "NPK01";

// Return the decoded object with the specified structure
return {

```

```

    data: decoded
};

}

```

Downlink Code:

```

function encodeDownlink(input) {
    return {
        bytes: [],
fPort: 2,
        warnings: [],
        errors: []
    };
}

function decodeDownlink(input) {
    return {
        data: {
            bytes: input.bytes
        },
        warnings: [],
        errors: []
    }
}

```

Python Script Code:

```

import paho.mqtt.subscribe as subscribe
import json
while True:
    m = subscribe.simple(topics=['#'], hostname="au1.cloud.thethings.network", port=1883,
                         auth={'username': "soilnpk@ttn",
                                'password':
"NNXSX.X7YC3OCZTUYGM2Z3YDGGUSRCH4C6NYH4D5WHK3I.FML37TAETYWV
ZVC3NIPDQCRUNWZ67VTGM6OCXKPVNYKLB6JHBFPQ"},

                         msg_count=2)
    for message in m:

```

```

payload_dict = json.loads(message.payload.decode('utf-8'))
decoded_payload = payload_dict.get('uplink_message', {}).get('decoded_payload')
print(decoded_payload)
if (decoded_payload):
    battery=decoded_payload.get('Bat')
    phos=decoded_payload.get('P_SOIL')
    Nito=decoded_payload.get('N_SOIL')
    pota=decoded_payload.get('K_SOIL')
    print("Potassium :", pota)
    print('Phosphorus', phos)
    print('Nitrogen :',Nito)

```

PROCEDURE:

Configure The Things Network (TTN)

1. Log in to The Things Network (TTN) Console:

- Go to TTN Console and log in with your credentials.

2. Create an Application:

- In the TTN Console, navigate to the "Applications" section.
- Click on "Add application" and fill in the required details such as Application ID, Description, and Handler Registration.

3. Register the Device:

- Within the created application, navigate to the "Devices" section.
- Click on "Register Device."
- Enter the Device ID, Device EUI, and App Key. These keys should be provided with your LSNPK01 sensor.
- Click on "Register" to complete the registration.

Configure Uplink Decoder

1. Add Custom Payload Formatter:

- Navigate to the "Payload Formats" tab in your application.
- Select "Custom" and enter the above uplink JavaScript code for the uplink decoder:

2. Save the Formatter:

- Click "Save payload functions" to apply the custom payload formatter.

Configure Downlink Encoder and Decoder

1. Add Custom Encoder and Decoder:

- In the "Payload Formats" tab, add the above JavaScript code for the downlink encoder and decoder:

2. Save the Formatter:

- Click "Save payload functions" to apply the custom encoder and decoder.

Python Script for Data Subscription

1. Write the Python Script:

- Write the above python code in the python file to subscribe to the MQTT topic and decode the payload:

2. Run the Script:

- Execute the script to start receiving and decoding data:

FLASK WEB APPLICATION CREATION CODE FOR SOIL NPK SENSOR

```

import paho.mqtt.subscribe as subscribe
import json
import sqlite3
from flask import Flask, render_template
import threading
app = Flask(__name__)
# MQTT subscription and database update function
def mqtt_subscribe_and_update_db():
    while True:
        m = subscribe.simple(topics=['#'], hostname="au1.cloud.thethings.network", port=1883,
                             auth={'username': "soilnpk@ttn",
                                    'password':
"NNXSX.X7YC3OCZTUYGM2Z3YDGGUSRCH4C6NYH4D5WHK3I.FML37TAETYWV
ZVC3NIPDQCRUNWZ67VTGM6OCXKPVNYKLB6JHBFPQ"},

                             msg_count=2)
        for message in m:
            payload_dict = json.loads(message.payload.decode('utf-8'))
            decoded_payload = payload_dict.get('uplink_message', {}).get('decoded_payload')
            print(decoded_payload)
            if decoded_payload:
                battery = decoded_payload.get('Bat')
                phos = decoded_payload.get('P_SOIL')
                nito = decoded_payload.get('N_SOIL')
                pota = decoded_payload.get('K_SOIL')
                print("Potassium :", pota)
                print('Phosphorus', phos)
                print('Nitrogen :', nito)

# Update the database
            conn = sqlite3.connect('npkData.db')
            cur = conn.execute("SELECT * FROM npk WHERE id=1")
            row = cur.fetchone()

```

```

if row is None:
    conn.execute("INSERT INTO npk (Battery, Potassium, Nitrogen, Phosphorus)
VALUES (?, ?, ?, ?)",
             (battery, pota, nito, phos))
else:
    conn.execute("UPDATE npk SET Battery=?, Potassium=?, Nitrogen=?,
Phosphorus=? WHERE id=?",
             (battery, pota, nito, phos, 1))
conn.commit()
conn.close()

# Start the MQTT subscription function in a separate thread
threading.Thread(target=mqtt_subscribe_and_update_db, daemon=True).start()

@app.route('/')
def npk():
    conn = sqlite3.connect('npkData.db')
    cur = conn.execute("SELECT * FROM npk WHERE id=1")
    row = cur.fetchone()
    if row:
        battery = row[1]
        pota = row[2]
        nito = row[3]
        phos = row[4]
    else:
        battery = pota = nito = phos = 'N/A'
    conn.close()
    return render_template('index.html', battery=battery, pota=pota, nito=nito, phos=phos)
if __name__ == "__main__":
    app.run(host="0.0.0.0")

```

b. Interfacing of Leaf Moisture Sensor

HARDWARE REQUIREMENTS :

- LLMS01-LoRaWAN Leaf Moisture Sensor
- LoRa Gateway
- Plants

OBJECTIVES

1 .Sensor Setup and Configuration:

- Learn how to correctly install and configure the LLMS01 Leaf Moisture Sensor.
- Understand the LoRaWAN protocol and how to integrate the sensor with a LoRa Gateway.

2. Data Collection and Transmission:

- Collect leaf moisture data from the LLMS01 sensor.
- Transmit the collected data to a LoRa Gateway.
- Ensure reliable communication between the sensor and the gateway.

3. Data Analysis and Interpretation:

- Analyze the leaf moisture data to determine the moisture levels on the plant leaves.
- Correlate the leaf moisture data with soil moisture levels and irrigation needs.
- Understand the impact of environmental factors on leaf moisture readings.

4. Integration with IoT Platforms:

- Forward the data received by the LoRa Gateway to an IoT platform for remote monitoring.
- Set up dashboards and alerts to monitor plant health in real-time.

5. Practical Applications:

- Utilize the leaf moisture data to optimize irrigation schedules, ensuring efficient water use.
- Monitor plant health to detect early signs of disease or water stress.
- Enhance agricultural practices by integrating sensor data with other environmental data sources.

Leaf Moisture Sensor



Features

- LoRaWAN 1.0.3 Class A
- Ultra-low power consumption
- Monitor Leaf moisture
- Monitor Leaf temperature
- Monitor Battery Level
- Bands: CN470/EU433/KR920/US915/EU868/AS923/AU915/IN865
- AT Commands to change parameters
- Uplink on periodically
- Downlink to change configure
- IP66 Waterproof Enclosure
- IP67 rate for the Sensor Probe
- 8500mAh Battery for long term use

Probe Specification

- Leaf Moisture: percentage of water drop over total leaf surface
- Range 0-100%
- Resolution: 0.1%
- Accuracy: $\pm 3\%$ (0-50%) ; $\pm 6\%$ ($>50\%$)
- IP67 Protection
- Length: 3.5 meters

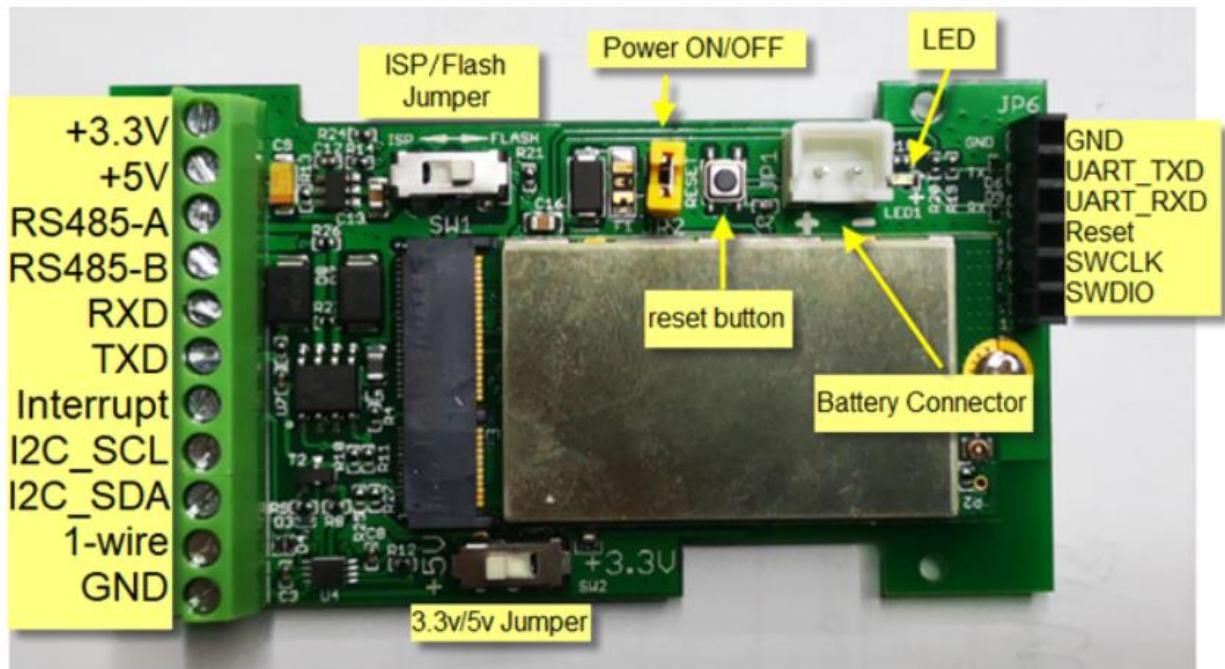
Leaf Temperature:

- Range $-50^{\circ}\text{C} \sim 80^{\circ}\text{C}$
- Resolution: 0.1°C
- Accuracy: $\leq \pm 0.5^{\circ}\text{C}$ ($-10^{\circ}\text{C} \sim 70^{\circ}\text{C}$), $\leq \pm 1.0^{\circ}\text{C}$ (others)
- IP67 Protection
- Length: 3.5 meters

Applications

- Smart Agriculture

Pin mapping and power on



Uplink Payload Format

LLMS01 will uplink payload via LoRaWAN with below payload format:

Uplink payload includes in total 11 bytes.

Normal uplink payload:

Size(bytes)	2	2	2	2	1	1	1
Value	BA T	Temperatur e (Optional)	Leaf Moistur e	Leaf Temperatur e	Digital Interrupt (Optional)	Reserv e	Messag e Type

PROGRAM

UPLINK Code:

```
function Decoder(bytes, port) {
    // Parse the raw payload data
    var battery = (bytes[0] << 8) | bytes[1]; // Assuming battery level is a 16-bit integer
    var moisture = (bytes[4] << 8) | bytes[5]; // Assuming Moisture is a 16-bit integer
    var Leaf_temperature = (bytes[6] << 8) | bytes[7]; // Assuming temperature is a 16-bit
    integer

    // Convert raw values to actual measurements
    var batteryVoltage = battery / 1000.0; // Convert to voltage assuming 1 mV resolution
    var leafMoisture = moisture/ 10.0; // Convert Leaf Moisture
    var leafTemperature= Leaf_temperature / 10.0; // Convert to Celsius assuming 1/100th
    degree resolution

    // Prepare the output object
    var decoded = {
        batteryVoltage: batteryVoltage,
        leafMoisture: leafMoisture,
        leafTemperature: leafTemperature,
        device_id: "leafsensor2"
    };
    return decoded;
}
```

DOWNLINK Code:

```
function encodeDownlink(input) {
    return {
        bytes: [],
        fPort: 2,
        warnings: [],
        errors: []
    };
}
```

```
function decodeDownlink(input) {
    return {
        data: {
            bytes: input.bytes
        },
        warnings: [],
        errors: []
    }
}
```

PYTHON Script:

```
import paho.mqtt.subscribe as subscribe
import json
while True:
    m = subscribe.simple(topics=['#'], hostname="au1.cloud.thethings.network", port=1883,
                         auth={'username': "leafsensor2@ttn",
                                'password':
"NNXSX.3WVCXG6AKMSSX6XVRYH5CE2SMPV23S45GWMNLQ.JFQ462YGPCQT
3TA4KMIAGZDCG4DWCLZNT2VQEIJ5LAWXQB4YWCD"}, msg_count=2)
    for message in m:
        payload_dict = json.loads(message.payload.decode('utf-8'))
        decoded_payload = payload_dict.get('uplink_message', {}).get('decoded_payload')
```

```

print(decoded_payload)
if (decoded_payload):
    battery=decoded_payload.get('batteryVoltage')
    leafMoisture=decoded_payload.get('leafMoisture')
    leafTemperature=decoded_payload.get('leafTemperature')
    print("batteryVoltage :" ,battery)
    print("leafMoisture :", leafMoisture)
    print("leafTemperature :", leafTemperature)

```

Procedure for Interfacing the Leaf Moisture Sensor (LLMS01)

Step 1: Hardware Setup

1. LLMS01 Sensor Installation:

- Place the LLMS01 Leaf Moisture Sensor in a suitable position where it can accurately measure the moisture level on the plant leaves.
- Ensure the sensor is securely attached to the leaf and connected properly to its power source.

2. LoRa Gateway Setup:

- Connect and power on your LoRa Gateway.
- Ensure it has a stable internet connection for data transmission to the IoT platform.

Step 2: Configuring the Sensor

1. Register the Sensor:

- Register the LLMS01 sensor on the LoRaWAN network server (e.g., The Things Network).
- Use the unique keys provided with the sensor for registration.

Step 3: Writing the Decoder and Downlink Code

1. Uplink Decoder Function:

- Create a JavaScript decoder function to parse the incoming data from the sensor.

2. Downlink Encoder and Decoder Functions:

- Write the functions for encoding and decoding downlink messages.

Step 4: Python Script for Data Collection

1. Install Required Libraries:

- Install the paho-mqtt library if not already installed.

```
pip install paho-mqtt
```

2. Write the Python Script:

- Use the provided Python script to subscribe to the MQTT topics and decode the payload.

Step 5: Deploy and Test

1. Deploy the Setup:

- Deploy the sensor in the field and start the LoRa Gateway.
- Ensure the sensor is actively transmitting data to the LoRa Gateway.

2. Monitor the Data:

- Use the Python script to monitor the incoming data.
- Check the console output for battery voltage, leaf moisture, and leaf temperature readings.

3. Troubleshooting:

- If data is not received, check the sensor's power supply, placement, and connection to the LoRa network.
- Ensure the LoRa Gateway is functioning correctly and connected to the internet.

3. Interfacing of LoRaWAN Distance Detection Sensor

HARDWARE REQUIREMENTS:

- LDDS75 Ultrasonic Sensor
- LoRa Gateway

OBJECTIVES:

- **Sensor Setup and Configuration:**
 - Properly install and configure the LDDS75 sensor.
 - Understand the LoRaWAN protocol and how to integrate the sensor with a LoRa Gateway.
- **Data Collection and Transmission:**
 - Collect distance measurements from the LDDS75 sensor.
 - Transmit the collected data to a LoRa Gateway.
 - Ensure reliable communication between the sensor and the gateway.
- **Data Analysis and Interpretation:**
 - Analyze the distance data to interpret the sensor readings.
 - Use the data to monitor and manage distance-related applications.
- **Integration with IoT Platforms:**
 - Forward the data received by the LoRa Gateway to an IoT platform for remote monitoring.
 - Set up dashboards and alerts to monitor distance measurements in real-time.
- **Practical Applications:**
 - Utilize the distance data for various applications such as tank level monitoring, object detection, and more.
 - Improve efficiency and accuracy in distance measurement-related applications.
- **Evaluation and Troubleshooting:**

- Evaluate the performance of the LDDS75 sensor and the LoRa communication system.
- Troubleshoot any issues related to data transmission, sensor accuracy, or integration with the IoT platform.

LDDS75 LoRaWAN Distance Detection Sensor



Features:

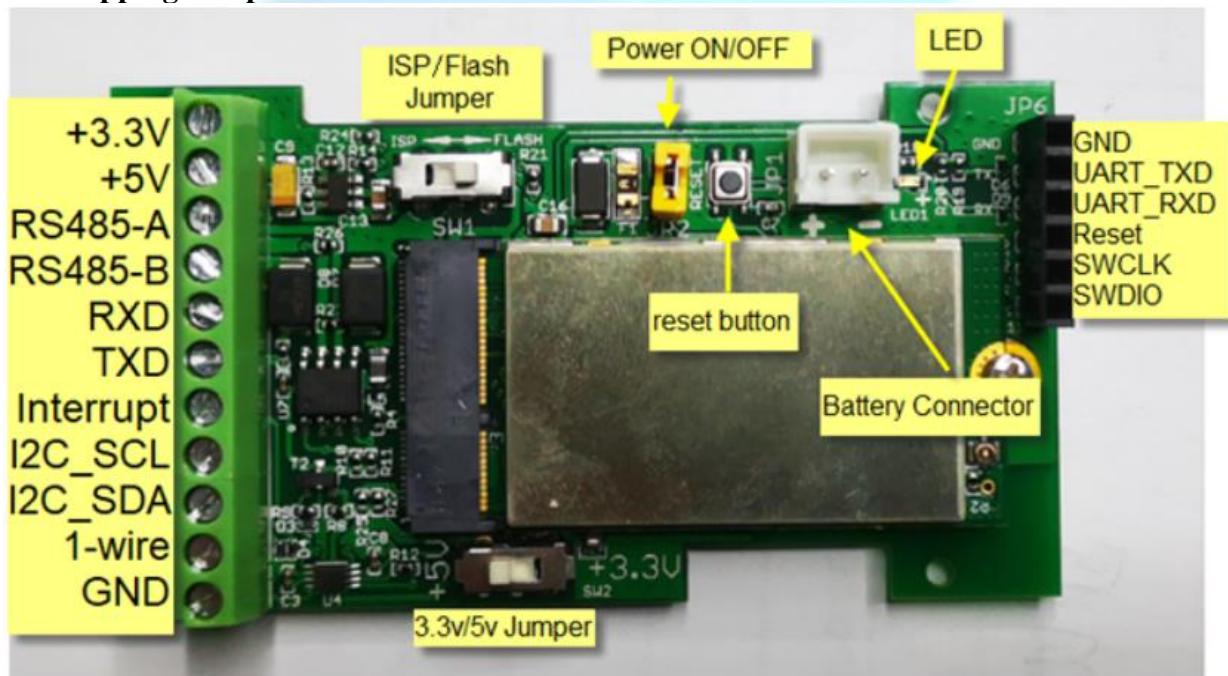
- LoRaWAN 1.0.3 Class A
- Ultra low power consumption
- Distance Detection by Ultrasonic technology
- Flat object range 280mm - 7500mm
- Accuracy: $\pm(1\text{cm}+S \times 0.3\%)$ (S: Distance)
- Measure Angle: 40°

- Cable Length : 25cm
- Temperature Compensation
- Bands: CN470/EU433/KR920/US915/EU868/AS923/AU915/IN865
- AT Commands to change parameters
- Uplink on periodically
- Downlink to change configure
- IP66 Waterproof Enclosure
- 4000mAh or 8500mAh Battery for long term use

Applications:

- Horizontal distance measurement
- Liquid level measurement
- Parking management system
- Object proximity and presence detection
- Intelligent trash can management system
- Robot obstacle avoidance
- Automatic control, sewer
- Bottom water level monitoring

Pin mapping and power on



Uplink Payload

LDDS75 will uplink payload via LoRaWAN with below payload format:

Uplink payload includes in total 4 bytes.

Payload for firmware version v1.1.4. . Before v1.1.3, there is one two fields: BAT and Distance

Size (bytes)	2	2	1	2	1
Value	BAT	Distance (unit: mm)	Digital Interrupt (Optional)	Temperature (Optional)	Sensor Flag

PROGRAM:

UPLINK Code

```
function Decoder(bytes, port) {
    // Parse the raw payload data
    var battery = (bytes[0] << 8) | bytes[1]; // Assuming battery level is a 16-bit integer
    var distance = (bytes[2] << 8) | bytes[3]; // Assuming distance is a 16-bit integer
    var temperature = (bytes[4] << 8) | bytes[5]; // Assuming temperature is a 16-bit integer

    // Convert raw values to actual measurements
    var batteryVoltage = battery / 1000.0; // Convert to voltage assuming 1 mV resolution
    var distanceInMeters = distance / 100.0; // Convert to meters assuming 1 cm resolution
    var temperatureInCelsius = temperature / 100.0; // Convert to Celsius assuming 1/100th
    degree resolution

    // Prepare the output object
    var decoded = {
        batteryVoltage: batteryVoltage,
        distanceInMeters: distanceInMeters,
        temperatureInCelsius: temperatureInCelsius
    };

    return decoded;
}
```

DLINK Code

```
function encodeDownlink(input) {
    return {

```

```

bytes: [],
fPort: 2,
warnings: [],
errors: []
};

}

function decodeDownlink(input) {
  return {
    data: {
      bytes: input.bytes
    },
    warnings: [],
    errors: []
  }
}

```

Python Script

```

import paho.mqtt.subscribe as subscribe
import json
while True:
  m = subscribe.simple(topics=['#'], hostname="au1.cloud.thethings.network", port=1883,
                       auth={'username': "loradustbin@ttn",
                              'password':
"NNXSX.E4JILCUICFZKDHPYZKS6GTXRC7PP4M5SXW7WWHA.EY45UPSTJGIPOF
ZEGQ7WZB5LQCQCH3CLMJ3GEPTZGPKFFVWH6ANQ"}, msg_count=2)
  for message in m:
    payload_dict = json.loads(message.payload.decode('utf-8'))
    decoded_payload = payload_dict.get('uplink_message', {}).get('decoded_payload')
    print(decoded_payload)
    if (decoded_payload):
      distance=decoded_payload.get('distanceInMeters')

```

```
battery=decoded_payload.get('batteryVoltage')
deviceID=decoded_payload.get('device_id')
print('distance=' ,distance)
print('battery=', battery)
print('deviceID=',deviceID)
```



Procedure for Interfacing the LDDS75 LoRaWAN Distance Detection Sensor

Step 1: Hardware Setup

1. LDDS75 Sensor Installation:

- Mount the LDDS75 Distance Detection Sensor in a suitable position where it can accurately measure distances.
- Ensure the sensor is securely attached and connected properly to its power source.

2. LoRa Gateway Setup:

- Connect and power on your LoRa Gateway.
- Ensure it has a stable internet connection for data transmission to the IoT platform.

Step 2: Configuring the Sensor

1. Register the Sensor:

- Register the LDDS75 sensor on the LoRaWAN network server (e.g., The Things Network).
- Use the unique keys provided with the sensor for registration.

Step 3: Writing the Decoder and Downlink Code

1. Uplink Decoder Function:

- Create a JavaScript decoder function to parse the incoming data from the sensor.

2. Downlink Encoder and Decoder Functions:

- Write the functions for encoding and decoding downlink messages.

Step 4: Python Script for Data Collection

1. Install Required Libraries:

- Install the paho-mqtt library if not already installed.

```
pip install paho-mqtt
```

2. Write the Python Script:

- Use the provided Python script to subscribe to the MQTT topics and decode the payload.

Step 5: Deploy and Test

1. Deploy the Setup:

- Deploy the sensor in the field and start the LoRa Gateway.
- Ensure the sensor is actively transmitting data to the LoRa Gateway.

2. Monitor the Data:

- Use the Python script to monitor the incoming data.
- Check the console output for distance readings, battery voltage, and temperature.

3. Troubleshooting:

- If data is not received, check the sensor's power supply, placement, and connection to the LoRa network.
- Ensure the LoRa Gateway is functioning correctly and connected to the internet.





