ENDTERM 2024 IOT PAPER SOLUTION EXPANDED

END TERM EXAMINATION - SIXTH SEMESTER [B.TECH] JUNE 2024
Subject: Introduction to Internet of Things
Paper Code: CIE-330T/ICE-328T/IOT-324T

Q1. Attempt all questions

(a) Write the characteristics of IOT system. (3)

Answer:
The characteristics of an IoT system are fundamental to its design and operation. They define how IoT devices and networks function and interact. The key characteristics are:

1. **Connectivity:** This is an essential requirement. Things of IoT *must* be connected to the IoT infrastructure (e.g., the internet or a local network). This allows devices to communicate with each other, with backend systems, and with users. The connection should ideally be available "anyone, anywhere, anytime." *(Ref: Unit 1 Notes, Section 5.1)*
2. **Intelligence and Identity:**
    - **Intelligence:** IoT systems aim to extract meaningful knowledge and insights from the vast amounts of data generated by sensors and devices. This often involves data processing, analytics, and even artificial intelligence to make data useful.
    - **Identity:** Each IoT device has a unique identifier (e.g., IP address, MAC address, custom ID). This identity is crucial for tracking the device, querying its status, managing it remotely, and ensuring secure communication. *(Ref: Unit 1 Notes, Section 5.2)*
3. **Scalability:** IoT systems must be designed to handle a massive and ever-increasing number of connected devices and the enormous volume of data they generate. The architecture should support expansion without significant degradation in performance. *(Ref: Unit 1 Notes, Section 5.3)*
4. **Dynamic and Self-Adapting (Complexity):** IoT devices and systems should be able to dynamically adapt to changing contexts, environments, and operational scenarios. For example, a smart thermostat adjusts based on occupancy, or a surveillance camera adapts to different lighting conditions. *(Ref: Unit 1 Notes, Section 5.4)*
5. **Architecture:** IoT architecture is typically heterogeneous and hybrid, meaning it must support devices and protocols from different manufacturers and integrate various technologies. IoT is not owned by a single engineering branch but is a convergence of multiple domains. *(Ref: Unit 1 Notes, Section 5.5)*
6. **Safety (Security and Physical Safety):**

- **Data Security & Privacy:** Protecting sensitive user data and device data from unauthorized access, breaches, and misuse is a major challenge and a critical characteristic.
- **Equipment Safety:** Ensuring the physical safety of the deployed IoT devices (which can be in harsh or remote environments) and the overall network infrastructure is also vital. *(Ref: Unit 1 Notes, Section 5.6)*

7. **Self-Configuring:** IoT devices should ideally be able to configure themselves, connect to the network, and update their software with minimal user intervention. This simplifies deployment and maintenance of large-scale IoT systems. *(Ref: Unit 1 Notes, Section 5.7)*

(b) Why gateway is important for device management in IOT systems? (3)

Answer:
A gateway is critically important for device management in IoT systems for several key reasons:

1. **Protocol Translation and Interoperability:** IoT devices often use a variety of low-power, short-range communication protocols (e.g., Zigbee, Bluetooth LE, Z-Wave, LoRaWAN) that are not directly compatible with standard internet protocols (TCP/IP). The gateway acts as a bridge, translating data between these different protocols, allowing diverse devices to connect to the internet and backend systems. This enables centralized management of heterogeneous devices. *(Ref: Unit 1 Notes, Section 16.2 & 25)*
2. **Connectivity Aggregation:** A gateway aggregates data from multiple local IoT devices and sensors and forwards it to the cloud or a central server using a single, more robust connection (e.g., Wi-Fi, Ethernet, Cellular). This reduces the need for each individual device to have its own direct internet connection, which can be costly and power-intensive. *(Ref: Unit 1 Notes, Section 16.2)*
3. **Security Management:** The gateway can act as a critical security checkpoint. It can implement security measures such as firewalling, intrusion detection/prevention, encryption/decryption of data to and from devices, and authentication/authorization of devices attempting to join the network. This helps protect resource-constrained IoT devices that may have limited onboard security capabilities. *(Ref: Unit 1 Notes, Section 16.2 & 25)*
4. **Local Processing and Edge Analytics (Device Management aspect):** Gateways can perform preliminary data processing, filtering, and aggregation locally (edge computing). For device management, this means the gateway can monitor device health, perform diagnostics, and even execute simple management commands locally without constant cloud intervention, reducing latency and bandwidth usage. It can pre-fetch management instructions or buffer data if cloud connectivity is intermittent. *(Ref: Unit 1 Notes, Section 16.2 & 25)*
5. **Device Provisioning and Configuration (OMA-DM context):** In some device management architectures (like OMA-DM), the gateway facilitates communication between a central Device Management (DM) server and the end devices. The gateway can:
   - **Forward** management commands if the device and DM server can interact directly.
   - Perform **protocol conversion** if the device and DM server use distinct protocols.

- Act as a **proxy**, for instance, if an intermediate pre-fetch of management data is required in a lossy environment.
  This allows for assigning device IDs, activating, configuring parameters, registering/de-registering devices, and managing subscriptions. *(Ref: Unit 1 Notes, Section 25)*

6. **Buffering and Offline Operation:** If the connection to the central server or cloud is lost, a gateway can buffer data from devices and store management commands, synchronizing when connectivity is restored. This ensures data isn't lost and devices can still be managed to some extent.

In essence, the gateway simplifies the IoT architecture, enhances security, improves efficiency, and enables effective management of a diverse and numerous set of IoT devices.

(c) List the basic difference between transducers, sensors, and actuators. (3)

Answer:
The basic differences between transducers, sensors, and actuators are as follows:

Feature	Transducer	Sensor	Actuator
Primary Role	Converts energy from one form to another.	Detects a specific physical quantity or change in the environment and produces an output (often electrical).	Converts an electrical signal (or other energy form) into a physical action or movement.
Input	Any form of energy/signal.	Physical phenomenon/stimulus from the environment (e.g., light, temperature, pressure, motion).	Electrical signal (or hydraulic/pneumatic pressure) typically from a controller or system.
Output	A different form of energy/signal.	Typically an electrical signal (voltage, current, digital data) representing the sensed quantity.	Physical action (e.g., motion, light emission, sound production, valve opening/closing).
Direction of Interaction	Can be input (like a microphone) or output (like a loudspeaker).	Primarily an input device for a system (senses the environment).	Primarily an output device for a system (acts upon the environment).
Relationship	A sensor is a type of transducer (converts physical phenomenon to electrical signal). An actuator can also be a type of transducer (converts electrical signal to physical action).	A device that *uses* a transducer principle to detect a specific physical property.	A device that *uses* a transducer principle to produce a physical effect based on an input signal.
Example	Microphone (sound to electrical),	Temperature sensor, Light sensor, Motion	Electric motor, Solenoid valve, LED,

Feature	Transducer	Sensor	Actuator
	Loudspeaker (electrical to sound), Thermocouple (temperature to voltage).	sensor, Proximity sensor.	Relay, Hydraulic piston.
Purpose in IoT	The underlying principle of energy conversion used by sensors and actuators.	To gather data about the physical world.	To interact with and control the physical world based on data or commands.

(Ref: Unit 1 Notes, Section 11 & 12; Unit 2 Notes, Section 4 & 5)

(d) Why is an IDE required for prototyping the embedded device platform? (3)

Answer:
An Integrated Development Environment (IDE) is crucial for prototyping embedded device platforms for several key reasons:

1. **Code Development and Editing:** IDEs provide a specialized text editor with features like syntax highlighting, auto-completion, and code formatting. This makes writing the software (firmware) for the embedded device easier, faster, and less prone to errors. For prototyping, rapid iteration of code is essential. *(Ref: Unit 2 Notes, Section 3 - Basic Concept of Embedded System)*
2. **Compilation and Build Management:** Embedded devices typically use microcontrollers that understand machine code. An IDE integrates a compiler (and linker) that translates the high-level programming language (like C, C++, or Arduino's language) into executable machine code that the target microcontroller can understand. It also manages the build process, handling dependencies and creating the final binary file for the prototype. *(Ref: Unit 2 Notes, Section 3 - Basic Concept of Embedded System)*
3. **Debugging and Testing:** Prototyping involves significant debugging. IDEs offer debugging tools such as:
    - **Simulators:** Allow testing code logic without the actual hardware, catching errors early. *(Ref: Unit 2 Notes, Section 3 - Basic Concept of Embedded System)*
    - **In-circuit debuggers (with compatible hardware):** Allow stepping through code on the actual prototype, inspecting variables, setting breakpoints, and analyzing real-time behavior.
    - **Serial Monitor:** Many IDEs (like Arduino IDE) include a serial monitor to print debug messages from the device, helping to understand its state and data flow during prototyping.
4. **Code Uploading (Flashing):** Once the code is compiled, the IDE facilitates the process of transferring (uploading or "flashing") the executable code onto the embedded device's microcontroller memory (e.g., Flash memory). This is a critical step in making the prototype functional. *(Ref: Unit 2 Notes, Section 3 - Basic Concept of Embedded System)*

5. **Library Management:** Many embedded platforms (like Arduino) rely heavily on libraries to simplify interaction with hardware components (sensors, actuators) and communication protocols. IDEs often provide tools to easily manage, include, and update these libraries, speeding up the prototyping process by leveraging pre-written and tested code. *(Ref: Unit 4 Notes, Section 7)*
6. **Hardware Abstraction and Board Support:** IDEs are often tailored for specific embedded platforms. They provide board support packages (BSPs) that abstract low-level hardware details, allowing developers to work with higher-level functions for pins, communication interfaces, etc., which is very helpful during the rapid experimentation phase of prototyping. *(Ref: Unit 4 Notes, Section 4 - Arduino IDE setup)*
7. **Integrated Toolchain:** An IDE brings together all necessary tools (editor, compiler, debugger, uploader) into a single, cohesive environment. This streamlines the development workflow, making it more efficient to move from an idea to a working prototype quickly.

Without an IDE, developers would have to manage these tools and processes separately, significantly slowing down the prototyping phase and increasing complexity.

(e) What is a smart sensor, how it is different from sensor node. (3)

Answer:
Smart Sensor:
A **smart sensor** is a sensor that integrates additional onboard electronics and processing capabilities beyond basic sensing. These capabilities often include:

- **Signal Conditioning:** Amplification, filtering of the raw sensor signal.
- **Data Conversion:** Analog-to-Digital Conversion (ADC) if the sensing element is analog.
- **Microcontroller/Processing Unit:** For local data processing, calculations, or decision-making (e.g., calibration, self-diagnosis, applying simple algorithms to the sensed data).
- **Communication Interface:** To transmit processed data or alerts (wired or wireless).
- **Memory:** To store calibration data, configuration parameters, or temporarily buffer readings.

A smart sensor can perform some level of data interpretation or analysis locally before transmitting data, potentially reducing the amount of raw data sent and the processing load on subsequent components.

Sensor Node:
A **sensor node** is a fundamental component of a Wireless Sensor Network (WSN) or a broader IoT system. It is a device deployed in the environment to monitor physical or environmental conditions. A typical sensor node consists of:

- One or more **sensors** (which may or may not be "smart" sensors themselves).
- A **microcontroller** (or microprocessor) for controlling the node's operation, processing data, and managing communication.
- A **transceiver** (radio module) for wireless communication.

- A **power source** (often a battery).
- Optionally, memory for data storage.

The primary role of a sensor node is to collect data via its sensors and transmit this data (raw or partially processed) to other nodes or a central gateway/base station.

Differences between Smart Sensor and Sensor Node:

Feature	Smart Sensor	Sensor Node
Definition	A sensor with integrated processing, conditioning, and communication capabilities.	A (networked) device in a WSN/IoT system that includes sensor(s), MCU, transceiver, and power source.
Scope	Refers to the capabilities of the sensing *element* itself and its immediate integrated electronics.	Refers to the entire *device unit* that performs sensing and participates in a network.
Processing	Performs local data processing, calibration, self-diagnosis, simple algorithms.	May perform some processing via its MCU, but its primary sensor(s) might be basic (not necessarily smart).
Composition	Is a single integrated component.	Is a system composed of multiple components, including one or more sensors (which could be smart or basic).
Output	Often provides more refined, processed, or digital data directly.	Can output raw sensor data or data processed by its onboard MCU.
"Intelligence"	Has inherent "intelligence" due to its onboard processing.	"Intelligence" depends on its MCU and the sophistication of its sensors (if smart sensors are used).
Example	A temperature sensor with built-in ADC, linearization, and an I2C interface.	A battery-powered box with a basic thermistor, an MCU, and a LoRa radio, deployed outdoors.

In essence:

- A **smart sensor** is an advanced *sensing component*.
- A **sensor node** is a *system component* that *uses* sensors (which can be basic or smart) to gather data and communicate it within a network.
  A sensor node *can contain* one or more smart sensors, but a smart sensor itself is more focused on the advanced capabilities of the individual sensing unit.

(Ref: Unit 1 Notes, Section 8 & 27; Unit 2 Notes, Section 4)

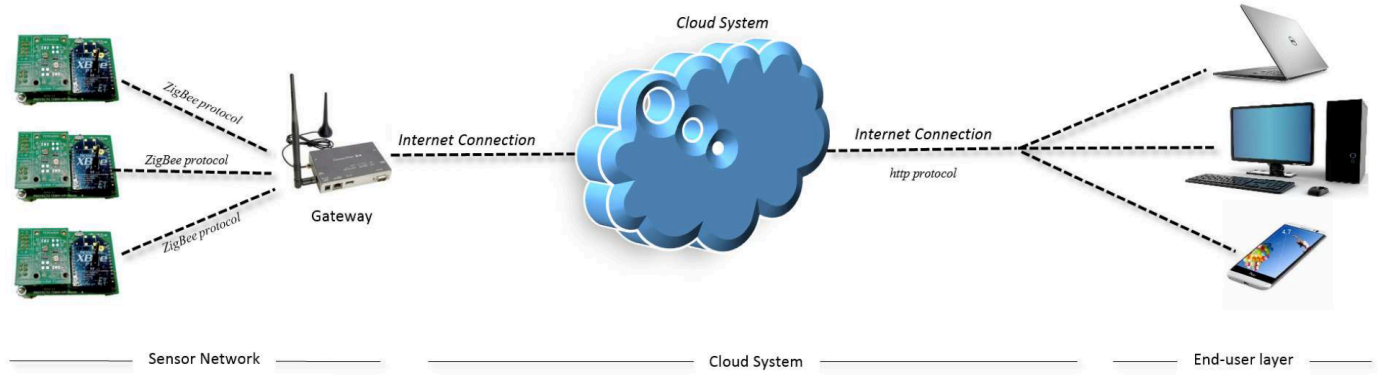
UNIT-I

Q2. (a) Explain the conceptual model and capabilities of an IoT solution with a neat diagram. (10)

Answer:
An IoT (Internet of Things) solution transforms physical objects into "smart" entities by connecting them to the internet, enabling data collection, communication, analysis, and action. A conceptual model helps understand its fundamental building blocks and how data flows through the system.

Conceptual Model of an IoT Solution:

A widely accepted conceptual model for an IoT solution involves several key layers or stages, often depicted as a flow from the physical world to applications and users.



(This diagram shows: Sensors/Devices -> Connectivity (Gateway/Network) -> Cloud (Data Processing/Analytics/Storage) -> Application/User Interface -> Actuators (optional feedback loop))

The key components and data flow can be broken down as:

1. **Things/Devices (with Sensors and Actuators):**
    - **Role:** These are the physical objects at the edge of the IoT solution. They interact directly with the environment or a process.
    - **Sensors:** Collect data from the physical world (e.g., temperature, light, motion, location, biometrics). They convert physical parameters into electrical signals.
        - *(Ref: Unit 1 Notes, Section 11; Unit 2 Notes, Section 4)*
    - **Actuators:** Act upon the physical world based on commands received from the system or decisions made from analyzed data (e.g., turning a light on/off, adjusting a motor, opening a valve). They convert electrical signals into physical actions.
        - *(Ref: Unit 1 Notes, Section 12; Unit 2 Notes, Section 5)*
    - **Identity:** Each device has a unique identity.
        - *(Ref: Unit 1 Notes, Section 5.2)*
2. **Connectivity (Gateways and Networks):**
    - **Role:** Enables data transfer from devices to the processing infrastructure and vice-versa.
    - **Local Networks:** Devices often connect to a local gateway using short-range wireless technologies (e.g., Wi-Fi, Bluetooth, Zigbee, LoRaWAN) or wired connections.
        - *(Ref: Unit 1 Notes, Section 17 & 18; Unit 3 Notes - various MAC/Comm protocols)*

- **Gateways:** Act as an intermediary. They aggregate data from local devices, perform protocol translation (if needed), and forward data to the cloud/backend via longer-range networks (e.g., Ethernet, Cellular, Wi-Fi). Gateways can also provide security, local processing (edge computing), and device management functions.
    - *(Ref: Unit 1 Notes, Section 16.2 & 25)*
- **Wide Area Networks (WAN):** The internet itself, or cellular networks (3G/4G/5G), satellite, etc., used for communication between gateways and the cloud.

3. **Data Processing and Management (Cloud/Edge Platform):**
    - **Role:** This is where raw data from devices is processed, stored, analyzed, and transformed into meaningful information.
    - **Cloud Platforms (e.g., AWS IoT, Azure IoT Hub, Google Cloud IoT):** Provide scalable infrastructure for:
        - **Data Ingestion:** Receiving and validating data from gateways/devices.
        - **Data Storage:** Storing large volumes of IoT data (e.g., time-series databases, data lakes).
        - **Data Processing & Analytics:** Applying rules, algorithms (including machine learning), and analytics to extract insights, detect patterns, and make predictions.
            - *(Ref: Unit 1 Notes, Section 16.3 & 16.4)*
        - **Device Management:** Managing the lifecycle of IoT devices (provisioning, configuration, monitoring, updates, decommissioning).
    - **Edge Computing:** Processing data closer to the source (on devices or gateways) to reduce latency, conserve bandwidth, and enable faster responses for time-critical applications.
        - *(Ref: Unit 1 Notes, Section 17)*
4. **Application and User Interface:**
    - **Role:** Presents the processed information and insights to users and allows them to interact with and control the IoT system.
    - **Applications:** These are software programs that leverage the IoT data to provide specific services or functionalities (e.g., smart home control app, industrial monitoring dashboard, predictive maintenance system).
    - **User Interfaces (UI):** Web portals, mobile apps, dashboards that visualize data, show alerts, and allow users to send commands to devices.
        - *(Ref: Unit 1 Notes, Section 16.5)*

Capabilities of an IoT Solution:

IoT solutions offer a wide range of capabilities that drive value across various domains:

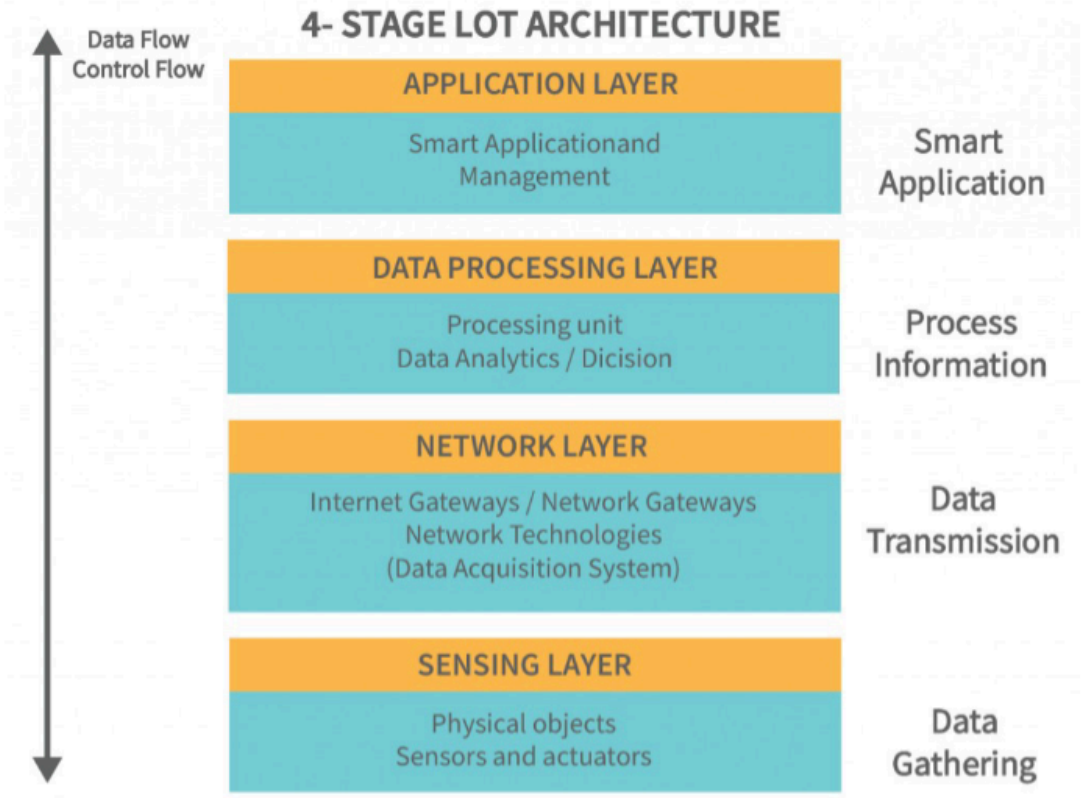
1. **Sensing and Data Collection:** The ability to gather real-time or periodic data about physical parameters, environmental conditions, device status, and user interactions.

2. **Connectivity and Communication:** Enabling seamless and reliable communication between devices, gateways, cloud platforms, and applications using various network technologies and protocols.
3. **Remote Monitoring and Control:** Allowing users or systems to monitor the status of assets, environments, or processes remotely and to control devices or systems from afar.
4. **Data Processing and Analytics:** Transforming raw sensor data into actionable insights through processing, analysis, visualization, and the application of advanced analytics like machine learning. This enables:
    - **Descriptive Analytics:** What happened?
    - **Diagnostic Analytics:** Why did it happen?
    - **Predictive Analytics:** What will happen?
    - **Prescriptive Analytics:** What should be done about it?
5. **Automation and Optimization:** Automating processes and optimizing operations based on real-time data and insights, leading to increased efficiency, reduced costs, and improved resource utilization.
    - *Example:* Automated irrigation based on soil moisture. *(Ref: Unit 1 Notes, Section 2)*
6. **Enhanced Decision Making:** Providing users and systems with timely and relevant information to make better, data-driven decisions.
7. **Scalability and Management:** The ability to support a large number of diverse devices and manage their lifecycle effectively.
8. **Security:** Implementing mechanisms to protect data, devices, and the network from unauthorized access and cyber threats. *(Ref: Unit 1 Notes, Section 5.6)*
9. **Interoperability:** The capability for different devices and systems (potentially from different vendors) to exchange and make use of information.
10. **New Services and Business Models:** Enabling the creation of innovative services and business models based on the data and capabilities offered by connected products.

A well-designed IoT solution integrates these components and capabilities to solve specific problems or create new opportunities.

(b) Explain the role of four-layers in a smart city architectural framework. (5)

Answer:
A smart city utilizes IoT and other information and communication technologies (ICT) to improve the quality of life for its citizens, enhance the efficiency of urban services, and promote sustainability. A layered architectural framework is often used to structure the complexity of a smart city. While various models exist, a common four-layer model includes:



(This is a generic 4-layer IoT architecture diagram. For smart city context, the examples within each layer would be city-specific.)

The role of these four layers in a smart city architectural framework is as follows:

1. **Perception/Sensing Layer (or Device Layer):**
    - **Role:** This is the foundational layer that directly interacts with the physical environment of the city. Its primary role is to collect raw data from various sources across the urban landscape.
    - **Components:**
        - **Sensors:** Deployed throughout the city to monitor various parameters, e.g., traffic sensors, air quality sensors, water level sensors, smart meters (electricity, water, gas), waste bin fill-level sensors, public safety cameras, parking sensors, weather stations.
        - **Actuators:** Devices that can effect changes in the city's infrastructure based on commands, e.g., smart traffic lights, smart streetlights, automated water valves, public announcement systems.
        - **IoT Devices:** Wearables for citizen health monitoring, connected vehicles, smart building components.
    - **Function in Smart City:** Gathers real-time data about urban conditions, resource usage, public safety, and environmental factors.
        - *(Ref: Unit 1 Notes, Section 10.1)*
2. **Network Layer (or Connectivity Layer):**
    - **Role:** Responsible for transmitting the data collected by the Perception Layer to processing centers and for delivering commands from higher layers back to actuators. It provides the

communication backbone for the smart city.

- **Components & Technologies:**
  - **Communication Networks:** Wired (fiber optics, Ethernet) and wireless (Wi-Fi, Cellular like 4G/5G, LoRaWAN, NB-IoT, Zigbee) networks providing connectivity across the city.
  - **Gateways:** Aggregate data from sensor networks and translate protocols for transmission over wider networks.
- **Function in Smart City:** Ensures reliable and efficient data transfer between the vast number of distributed devices and central or distributed processing platforms. Manages network traffic and ensures Quality of Service (QoS) for critical data.
  - *(Ref: Unit 1 Notes, Section 10.2; Unit 3 for MAC/Comm protocols)*

3. **Processing Layer (or Data Management/Middleware Layer):**
   - **Role:** This layer is the "brain" where raw data from the Network Layer is stored, processed, analyzed, and transformed into meaningful information and actionable insights.
   - **Components & Technologies:**
     - **Cloud Platforms & Data Centers:** For scalable storage and computation.
     - **Big Data Analytics Platforms:** To process and analyze massive datasets from various city sources.
     - **Databases:** Time-series databases, geospatial databases, etc.
     - **Middleware:** Software that enables communication and data management between different applications and services.
     - **Artificial Intelligence (AI) and Machine Learning (ML) algorithms:** For pattern recognition, predictive analytics (e.g., predicting traffic congestion, energy demand, potential equipment failures).
   - **Function in Smart City:** Aggregates data from diverse urban systems, performs complex event processing, runs predictive models, and generates alerts and reports for city services and decision-makers. It enables data fusion from different city domains (e.g., correlating traffic data with air quality data).
     - *(Ref: Unit 1 Notes, Section 10.3)*
4. **Application Layer (or Service Layer):**
   - **Role:** This is the top-most layer that delivers specific smart city services and applications to end-users (citizens, city administrators, businesses). It utilizes the processed information and insights from the Processing Layer.
   - **Components & Examples:**
     - **Smart City Applications:** Smart traffic management systems, smart parking solutions, smart waste management, smart energy grids, e-governance portals, public safety and emergency response systems, smart healthcare services, environmental monitoring dashboards, citizen engagement apps.

        - **User Interfaces:** Mobile apps, web dashboards, control room interfaces.
        - **APIs (Application Programming Interfaces):** To allow third-party developers to build innovative services on top of the smart city platform.
    - **Function in Smart City:** Provides tangible benefits to citizens and improves the efficiency of city operations. It is where the "smartness" of the city becomes visible and usable. This layer enables services like optimized traffic flow, efficient resource allocation, improved public safety, and better citizen services.
        - *(Ref: Unit 1 Notes, Section 10.4)*

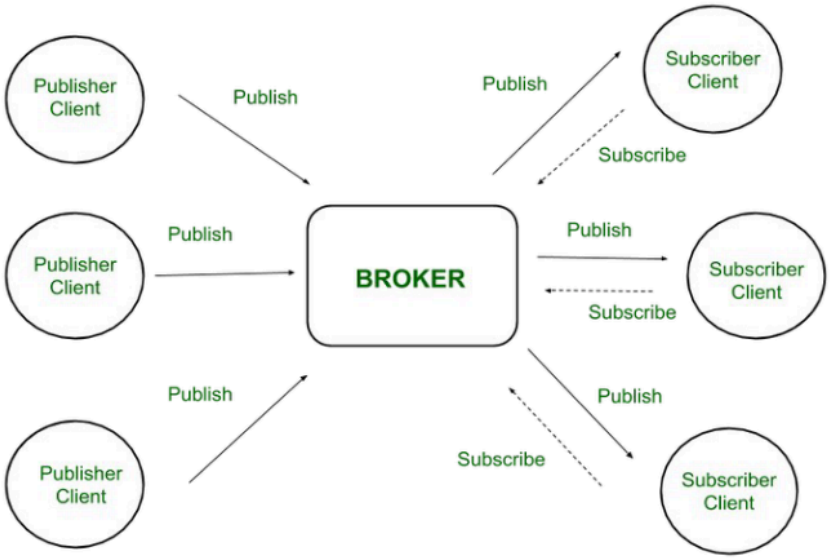
Each layer builds upon the capabilities of the layer below it, creating a comprehensive system that enables a city to become "smarter" by leveraging data and connectivity.

Q3. (a) Specify functions of COAP, RESTful HTTP, MQTT and XMPP (Extensible Messaging and Presence Protocol) in IoT applications. (7.5)

Answer:
CoAP, RESTful HTTP, MQTT, and XMPP are all application layer protocols, but they serve different functions and are suited for different types of IoT applications due to their distinct characteristics.

1. **CoAP (Constrained Application Protocol):**
    - **Designed for:** Constrained devices and constrained (low-power, lossy) networks (LLNs). It is an IETF standard.
    - **Functions in IoT Applications:**
        - **Lightweight Resource Manipulation:** Provides a way for resource-constrained devices (sensors, simple actuators) to expose and interact with their resources (e.g., sensor readings, actuator states) in a web-like manner, similar to HTTP but much lighter.
        - **Request/Response Model:** Operates on a client/server request-response model. Clients request actions (GET, POST, PUT, DELETE) on resources hosted by servers (often the IoT devices themselves).
            - *(Ref: Unit 1 Notes, Section 22 - Application Layer Protocol)*
        - **Asynchronous Message Exchanges:** Supports both synchronous (piggybacked response) and asynchronous (separate response) message exchanges, suitable for devices that may sleep to conserve power.
        - **UDP-based Communication:** Runs over UDP, which is connectionless and has lower overhead than TCP, making it suitable for lossy networks and reducing energy consumption.
        - **Reliability Options:** Offers optional reliability (confirmable messages with retransmissions) without the full overhead of TCP.
        - **Observe Functionality:** Allows clients to "observe" resources on a server. The server then notifies the client whenever the state of the observed resource changes, enabling efficient event-driven communication instead of constant polling.

        - **Resource Discovery:** Supports basic resource discovery mechanisms.
        - **Security:** Integrates with DTLS (Datagram Transport Layer Security) for secure communication over UDP.
    - **Typical IoT Use Cases:** Remote sensor monitoring, smart metering, simple control of actuators in WSNs, and other M2M scenarios where devices have limited processing power, memory, and energy.
2. **RESTful HTTP (Representational State Transfer over Hypertext Transfer Protocol):**
    - **Designed for:** Web services and general internet communication. It is a widely adopted architectural style, not a protocol itself, but typically implemented using HTTP.
    - **Functions in IoT Applications:**
        - **Interfacing with Web Services/Cloud Platforms:** Commonly used for IoT devices or gateways to communicate with cloud-based IoT platforms, web servers, and enterprise applications. This is because most cloud platforms expose their services via REST APIs.
        - **Resource-Oriented Interaction:** Devices and their data are treated as resources, identified by URIs. Standard HTTP methods (GET, POST, PUT, DELETE) are used to interact with these resources (CRUD operations - Create, Read, Update, Delete).
        - **Stateless Communication:** Each request from a client to a server must contain all the information needed to understand the request. The server does not store any client context between requests. This simplifies server design and improves scalability.
        - **Standardized Data Formats:** Often uses standard data formats like JSON or XML for message payloads, ensuring interoperability.
        - **Leveraging Existing Web Infrastructure:** Can utilize existing web infrastructure like proxies, caches, and security mechanisms (HTTPS).
    - **Typical IoT Use Cases:** Sending sensor data from gateways to cloud platforms, device provisioning and management via cloud APIs, mobile applications interacting with IoT backends, integration with enterprise systems. It's less common for direct device-to-device communication in constrained environments due to HTTP's overhead.
3. **MQTT (Message Queuing Telemetry Transport):**
    - **Designed for:** Low-bandwidth, high-latency, or unreliable networks. It is a lightweight publish/subscribe messaging protocol.
    - **Functions in IoT Applications:**
        - **Publish/Subscribe Messaging:** Decouples message producers (publishers) from message consumers (subscribers) through a central message broker. Publishers send messages to "topics," and subscribers interested in those topics receive the messages.
            - *(Ref: Unit 1 Notes, Section 22 - Application Layer Protocol)*



- **Efficient Data Distribution:** Ideal for distributing data from one source (e.g., a sensor) to many recipients (e.g., multiple applications or dashboards) or vice-versa.
- **Low Overhead:** Designed with a very small message header and minimal protocol commands, making it efficient in terms of bandwidth and processing power.
- **Reliability Levels (Quality of Service - QoS):**
  - **QoS 0 (At most once):** Fire-and-forget, no acknowledgment.
  - **QoS 1 (At least once):** Message is guaranteed to be delivered, but duplicates may occur. Requires acknowledgment.
  - **QoS 2 (Exactly once):** Message is guaranteed to be delivered exactly once. Highest reliability, most overhead.
- **Session Awareness (Persistent Sessions):** The broker can store session information for clients, including subscriptions and undelivered messages, so if a client disconnects and reconnects, it can resume its session.
- **Last Will and Testament (LWT):** Allows a client to specify a message to be published by the broker on its behalf if the client disconnects ungracefully. Useful for device status monitoring.
- **Security:** Can be secured using TLS/SSL for encrypted communication.

- **Typical IoT Use Cases:** Remote monitoring of sensors, telemetry data collection from vehicles or industrial equipment, mobile applications receiving real-time updates, smart home automation, messaging in environments with intermittent connectivity.

4. **XMPP (Extensible Messaging and Presence Protocol):**
   - **Designed for:** Real-time communication, originally for instant messaging (IM) and presence information. It is an open, XML-based protocol.
   - **Functions in IoT Applications:**
     - **Real-time Communication & Messaging:** Facilitates near real-time exchange of structured data between entities (devices, users, servers). Its push mechanism is efficient.
       - *(Ref: Unit 1 Notes, Section 22 - Application Layer Protocol)*

        - **Presence Indication:** Can be used to indicate the online/offline/busy status of IoT devices or users interacting with them.
        - **Decentralized Architecture:** Similar to email, XMPP allows for a federated (decentralized) server architecture, meaning anyone can run their own XMPP server, promoting interoperability.
        - **Extensibility:** Based on XML, making it highly extensible. Custom data formats and functionalities can be easily added through protocol extensions (XEPs - XMPP Extension Protocols).
        - **Persistent Connections:** Typically relies on persistent TCP connections, which can be power-intensive for some battery-operated IoT devices but good for real-time updates.
        - **Addressing (JIDs - Jabber IDs):** Uses JIDs (e.g., `device@domain/resource`) for addressing entities.
        - **Security:** Supports TLS for encrypting connections and SASL (Simple Authentication and Security Layer) for authentication.
    - **Typical IoT Use Cases:** Real-time notifications from devices, remote control of devices requiring immediate response, smart home automation where device presence and real-time status are important, integration with chat applications for device interaction (e.g., chatbots controlling devices), middleware communication without human intervention (e.g., Google Cloud Print, Logitech Harmony Hub). While powerful, its XML verbosity can be a concern for very constrained devices.

In summary, CoAP is for constrained M2M, RESTful HTTP for web service integration, MQTT for lightweight pub/sub messaging in unreliable networks, and XMPP for real-time, extensible communication with presence capabilities.

(b) Correlate M2M architectural domains with IoT architecture levels. (7.5)

Answer:
Machine-to-Machine (M2M) communication forms a foundational component of many Internet of Things (IoT) systems. While M2M often focuses on direct device-to-device or device-to-application communication for specific tasks like remote monitoring and control, IoT expands on this by integrating diverse M2M systems into larger, interconnected ecosystems, often leveraging cloud platforms and advanced analytics.

We can correlate M2M architectural domains (often described by standards bodies like ETSI) with common IoT architecture levels (like the 4-layer or 6-layer models).

ETSI M2M Architectural Domains:

ETSI defines two main domains for M2M architecture:

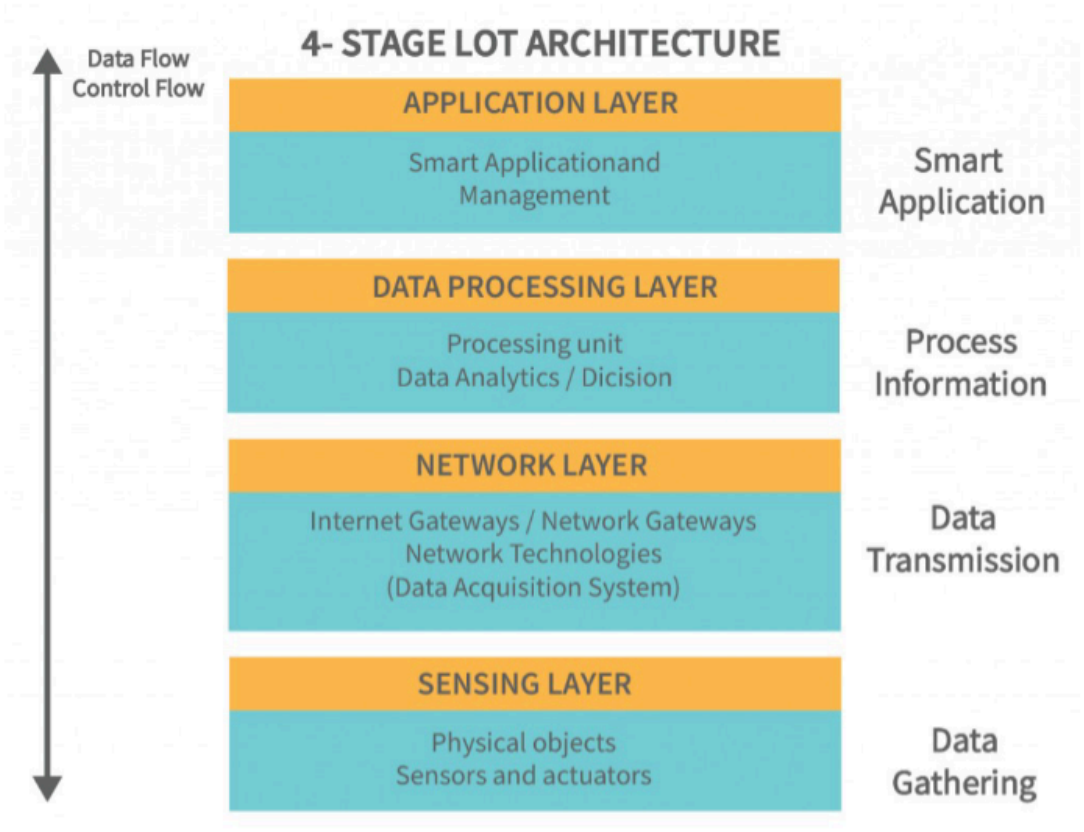
1. **M2M Device and Gateway Domain:** This domain includes:

- **M2M Devices:** The actual "things" with sensors/actuators.
- **M2M Area Network:** Local networks connecting M2M devices (e.g., using Zigbee, Bluetooth, PLC).
- **M2M Gateway:** Acts as an aggregator and an interworking point between the M2M Area Network and the wider Communication Network. It handles protocol translation, data aggregation, and local device management.

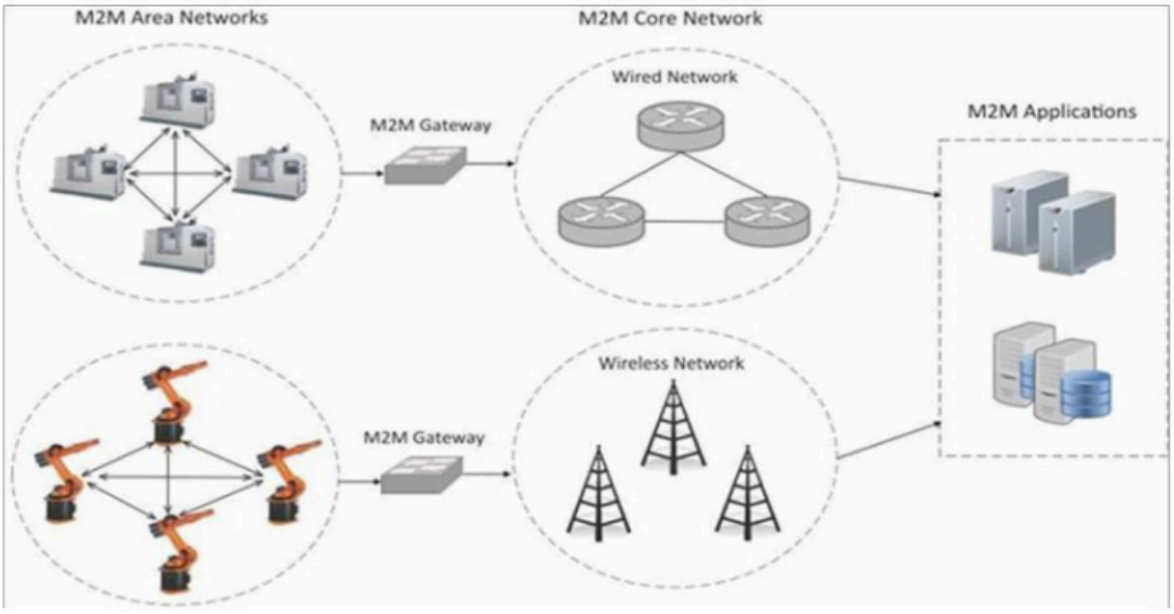
2. **M2M Network Domain (Communication Network & Applications/Services):** This domain includes:
   - **Access Network:** Provides connectivity from the gateway to the Core Network (e.g., xDSL, Cellular).
   - **Core Network:** The backbone network (e.g., IP MPLS, operator's core).
   - **M2M Service Capabilities Layer (SCL):** Provides common functionalities and services that can be used by various M2M applications (e.g., data management, device management, security functions). This is a crucial middleware layer.
   - **M2M Applications:** The specific applications that utilize the data and services provided by the M2M system (e.g., fleet management, smart metering application).

Correlation with a Common 4-Layer IoT Architecture:

Let's correlate these with a typical 4-Layer IoT Architecture (Sensing, Network, Processing, Application):



(Generic 4-layer IoT architecture)

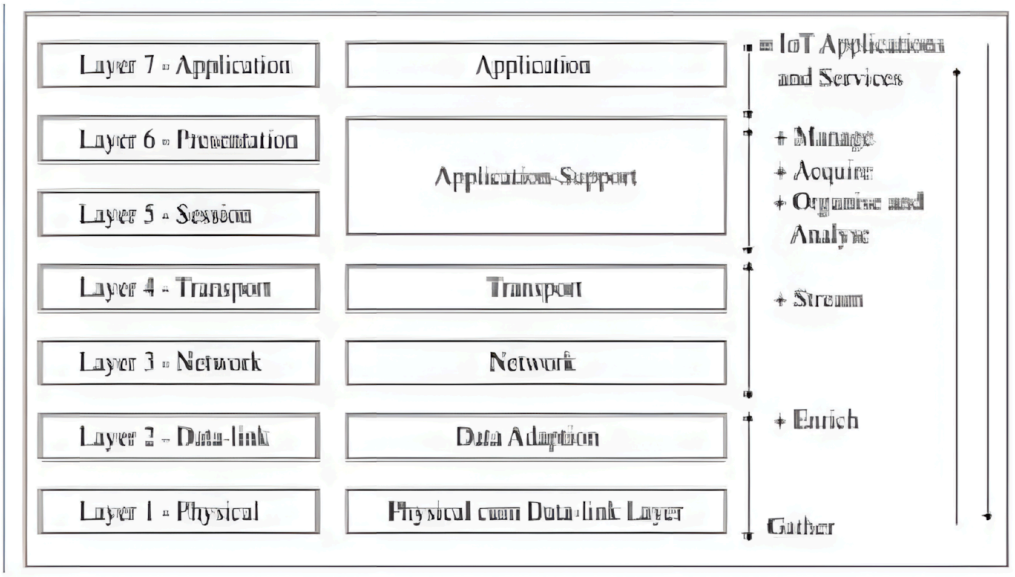


(ETSI M2M Domains for comparison context)

1. **IoT Perception/Sensing Layer:**
    - **Corresponds to: M2M Devices** within the ETSI M2M Device and Gateway Domain.
    - **Function:** Physical devices with sensors collecting data and actuators performing actions. This is where the raw interaction with the physical world occurs.
2. **IoT Network Layer (Connectivity Layer):**
    - **Corresponds to:**
        - **M2M Area Network** (local connectivity from devices to gateway).
        - **M2M Gateway** (for local aggregation and protocol translation).
        - **Access Network** (connecting gateway to the core network).
        - **Core Network** (providing wider network transport).
    - **Function:** Responsible for transmitting data from devices/gateways to higher layers and commands back down. This encompasses all communication infrastructure from the device up to the point where data reaches a processing or service platform.
3. **IoT Processing Layer (Data Management/Middleware Layer):**
    - **Corresponds to: M2M Service Capabilities Layer (SCL)** in the ETSI M2M Network Domain.
    - **Function:** This layer in IoT involves data ingestion, storage, processing, analytics, and device management. The M2M SCL provides these common reusable functions that are essential for processing M2M data and managing devices before the data is consumed by specific applications. Edge processing on the M2M Gateway can also be considered part of this layer.
4. **IoT Application Layer (Service Layer):**
    - **Corresponds to: M2M Applications** in the ETSI M2M Network Domain.
    - **Function:** This is where the value of the collected and processed data is realized through specific applications tailored for end-users or business processes (e.g., smart city traffic

management app, industrial predictive maintenance dashboard).

Correlation with IETF 6-Layer Modified OSI Model for IoT/M2M:



(IETF 6-Layer Model)

- **ETSI M2M Devices & M2M Area Network** primarily map to **L1 (Physical cum Data-Link Layer)** and **L2 (Data-Adaptation Layer - especially the gateway part)** of the IETF model.
- **ETSI Access Network & Core Network** map to **L3 (Network Layer)** and **L4 (Transport Layer)** of the IETF model, handling data routing and end-to-end communication setup.
- **ETSI M2M Service Capabilities Layer (SCL)** aligns well with **L5 (Application-Support Layer)** of the IETF model, which handles data managing, acquiring, organizing, analyzing, and uses application-support protocols like CoAP.
- **ETSI M2M Applications** directly correspond to **L6 (Application Layer)** of the IETF model, where the final applications and services reside.

Summary of Correlation:

ETSI M2M Domain/Component	Typical 4-Layer IoT Architecture Level	IETF 6-Layer IoT/M2M Model Level(s)
M2M Devices	Perception/Sensing Layer	L1 (Physical part), L2 (Device side of Data Adaptation)
M2M Area Network	Network Layer (local part)	L1 (Data-Link part), L2 (Local network within Data Adaptation)
M2M Gateway	Network Layer (edge connectivity), Processing Layer (edge processing)	L2 (Gateway in Data Adaptation), potentially extending to L5 for edge functions
Access Network, Core Network	Network Layer (WAN part)	L3 (Network), L4 (Transport)

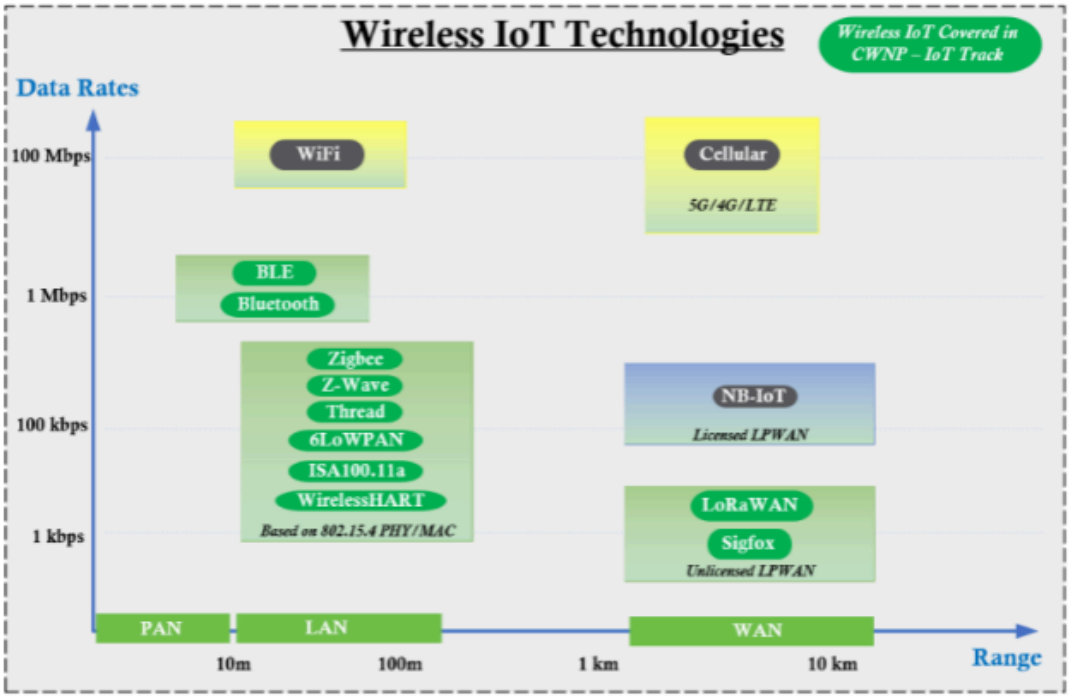
ETSI M2M Domain/Component	Typical 4-Layer IoT Architecture Level	IETF 6-Layer IoT/M2M Model Level(s)
M2M Service Capabilities Layer	Processing Layer (Middleware, Data Management, Device Management)	L5 (Application-Support Layer)
M2M Applications	Application Layer	L6 (Application Layer)

The M2M architectural domains provide a structured view of the components needed for machine-centric communication. IoT architecture levels build upon this, often emphasizing broader connectivity, cloud integration, and more sophisticated data analytics and application services. The correlation shows how M2M provides the foundational elements for many IoT deployments.

UNIT-II

Q4. (a) Compare NFC and RFID protocols which can be used for device communication in IoT. (7.5)

Answer:
NFC (Near Field Communication) and RFID (Radio Frequency Identification) are both wireless communication technologies used for identifying and tracking objects, and they play significant roles in various IoT applications. While NFC is a specialized subset of RFID technology, they have distinct characteristics and use cases.



(Contextual image for wireless communication)

NFC (Near Field Communication):

- **Principle:** Operates on the HF (High Frequency) band of RFID at 13.56 MHz. It enables short-range (typically a few centimeters, up to 20 cm) two-way communication between devices when they are

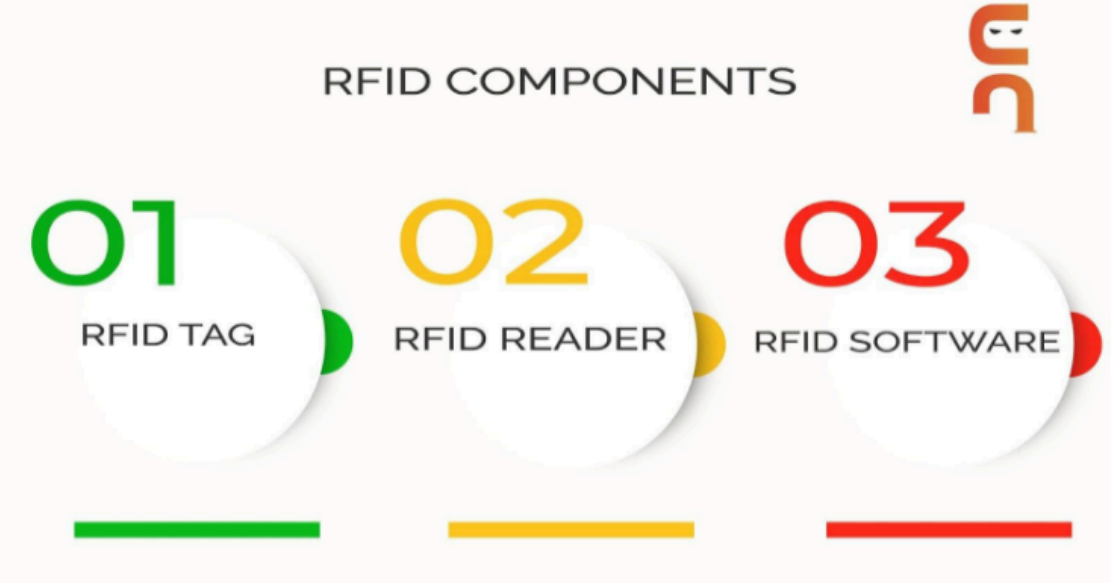
brought into close proximity ("tap" or "touch").

- **Modes of Operation:**
  - **Card Emulation Mode:** An NFC-enabled device (like a smartphone) acts as a contactless smart card (e.g., for payments, access control).
  - **Reader/Writer Mode:** An NFC device can read information from or write information to passive NFC tags embedded in objects.
  - **Peer-to-Peer Mode:** Two NFC-enabled devices can exchange data directly (e.g., sharing contacts, photos).
- **Key Features:**
  - **Short Range:** Provides inherent security as eavesdropping is difficult.
  - **Low Power Consumption:** Especially for passive tags powered by the reader's field.
  - **Ease of Use:** Intuitive "tap-and-go" interaction.
  - **Data Rate:** Up to 424 kbit/s.
  - **Standardization:** Well-standardized, ensuring interoperability.
- **IoT Applications:** Contactless payments (Apple Pay, Google Pay), smart ticketing for public transport, access control systems (e-keys), smart posters for interactive marketing, pairing Bluetooth devices, inventory tracking (item-level), healthcare (patient identification, medical device interaction). *(Ref: Unit 1 Notes, Section 23; Unit 2 Notes, Section 7)*

RFID (Radio Frequency Identification):

- **Principle:** Uses radio waves to transmit the identity (in the form of a unique serial number) of an object or person wirelessly. An RFID reader emits radio waves, and an RFID tag responds with its identification information.
- **Components:**
  - **Tag (Transponder):** A microchip attached to an antenna.
    - **Passive Tags:** No internal power source; powered by the electromagnetic field generated by the reader. Shorter read range, lower cost.
    - **Active Tags:** Have their own battery power source. Longer read range, can include sensors, more expensive.
    - **Semi-Passive (or Battery-Assisted Passive - BAP) Tags:** Use a battery to power the chip but use the reader's field for communication.
  - **Reader (Interrogator):** Transmits and receives radio signals to communicate with tags.

- **Antenna:** Emits radio signals to activate tags and read/write data.



(RFID Components)

- **Frequency Bands:**
  - **Low Frequency (LF):** ~125-134 kHz (short read range, less affected by materials like water/metal).
  - **High Frequency (HF):** 13.56 MHz (moderate read range, NFC is based on this).
  - **Ultra-High Frequency (UHF):** ~860-960 MHz (longer read range, faster data transfer, more sensitive to environment).
  - **Microwave:** ~2.45 GHz, ~5.8 GHz (very long range, specialized applications).
- **Key Features:**
  - **Variable Read Range:** From centimeters to many meters, depending on frequency and tag type.
  - **No Line-of-Sight Required:** Tags can often be read through non-metallic materials.
  - **Multiple Tag Reading:** Some RFID systems can read multiple tags simultaneously.
  - **Durability:** Tags can be ruggedized for harsh environments.
- **IoT Applications:** Supply chain management and logistics (pallet/case tracking), inventory control, asset tracking, livestock identification, toll collection systems, library management, access control, healthcare (patient tracking, asset management), anti-counterfeiting.
  (Ref: Unit 1 Notes, Section 23; Unit 2 Notes, Section 7)

Comparison Table: NFC vs. RFID

Feature	NFC (Near Field Communication)	RFID (Radio Frequency Identification)
Based On	A specialized subset of HF RFID technology.	Broader technology encompassing various frequencies and tag types.
Operating Frequency	Standardized at 13.56 MHz.	Operates across multiple frequency bands (LF, HF, UHF, Microwave).

Feature	NFC (Near Field Communication)	RFID (Radio Frequency Identification)
Communication Range	Very short: typically < 10 cm, maximum around 20 cm.	Variable: centimeters (LF/HF passive) to many meters (UHF/Microwave active).
Communication Type	Two-way (supports peer-to-peer, reader/writer, card emulation).	Primarily one-way (reader interrogates tag, tag responds). Some systems support tag writing.
Data Rate	Up to 424 kbit/s.	Varies widely depending on frequency and tag type (can be higher or lower than NFC).
Power (Tag)	Passive tags powered by reader; active devices have own power.	Passive (reader-powered), Active (battery-powered), BAP (battery-assisted).
Interaction	User-initiated "tap" or "touch."	Can be automated (e.g., passing through a reader portal) or user-initiated.
Complexity & Cost (Tag)	Tags are generally low cost, similar to passive HF RFID tags.	Passive tags very cheap; active tags are more complex and expensive.
Reader Cost	NFC readers often integrated into smartphones.	Dedicated RFID readers can vary significantly in cost and complexity.
Security	Short range provides inherent physical security; supports encryption.	Security varies; passive tags can be vulnerable. Active tags can offer more robust security.
Standardization	Highly standardized (ISO/IEC 18092, ISO/IEC 14443).	Multiple standards exist for different frequencies and applications (e.g., EPC Gen2 for UHF).
Primary IoT Focus	Secure transactions, simple data exchange, device pairing, interactive experiences.	Identification, tracking, inventory management over various distances.
Multiple Tag Reading	Typically one-to-one interaction at very close range.	Many RFID systems (especially UHF) can read hundreds of tags simultaneously.

Conclusion:

- **NFC** is ideal for secure, short-range, intuitive interactions, often involving user participation (like payments or tapping a smart poster). Its integration into smartphones makes it very accessible for consumer IoT applications.
- **RFID** is a more versatile technology suited for a broader range of identification and tracking applications, especially where longer read distances or simultaneous reading of many items are required (like logistics and inventory).

Both protocols are valuable for device communication in IoT, with the choice depending on the specific application requirements regarding range, security, cost, data volume, and interaction type.

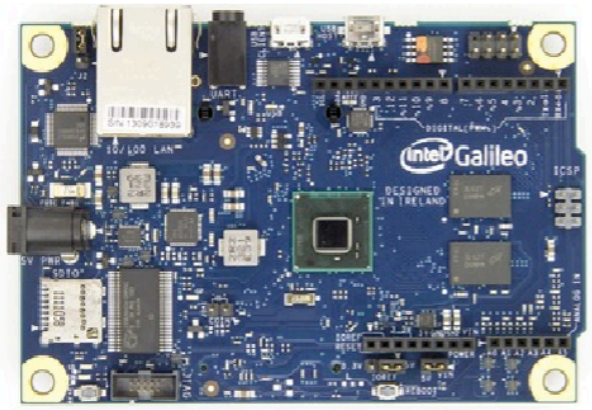
(Ref: Unit 1 Notes, Section 23; Unit 2 Notes, Section 7)

(b) Describe usages of Intel Galileo, Raspberry and BeagleBone boards for IoT applications. (7.5)

Answer:
Intel Galileo, Raspberry Pi, and BeagleBone are popular single-board computers (SBCs) or microcontroller boards frequently used for prototyping and deploying IoT applications. Each has distinct features making them suitable for different types of projects.

(Ref: Unit 2 Notes, Section 4 for all boards)

1. **Intel Galileo:**
    - **Key Characteristics:** Arduino-certified board based on the Intel Quark SoC X1000 (a 32-bit Pentium-class processor). It offers x86 architecture compatibility and can run a full Linux operating system. It is hardware and software pin-compatible with Arduino Uno R3 shields.



    - **Usages in IoT Applications:**
        - **Prototyping Complex IoT Devices:** Its ability to run Linux and support for a wider range of programming languages (beyond just Arduino sketches, by leveraging Linux) makes it suitable for more complex IoT prototypes that might require more processing power or intricate software stacks than a typical Arduino.
        - **Educational Purposes & x86 Development:** Useful for developers familiar with the x86 architecture or those wanting to develop IoT solutions on an Intel-based platform.
        - **Networked IoT Applications:** With built-in Ethernet and support for Wi-Fi via mini-PCIe, it's well-suited for IoT projects requiring robust network connectivity (e.g., gateways, data loggers connected to a local network or the internet).
        - **Interfacing with Arduino Ecosystem:** Its Arduino compatibility allows leveraging the vast array of existing Arduino shields and sensors for rapid hardware prototyping, while still offering more processing power.
        - **Projects Requiring Linux Capabilities:** For IoT applications that need multi-tasking, advanced file system operations, or running existing Linux-based software components (e.g., web servers, databases for local data storage).

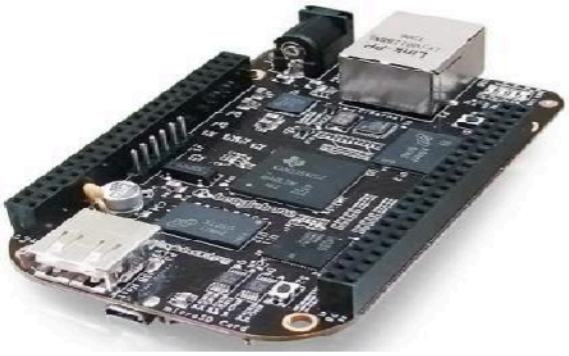
        - **Data Aggregation and Edge Processing:** Can be used as a simple edge gateway to collect data from multiple sensors (via Arduino shields or other interfaces) and perform some initial processing or filtering before sending data to the cloud.

2. **Raspberry Pi:**
    - **Key Characteristics:** A low-cost, credit-card-sized single-board computer based on an ARM processor. It runs a full-fledged Linux-based OS (like Raspberry Pi OS) and has significant processing power, RAM, and graphics capabilities compared to microcontrollers. It offers excellent connectivity options (Ethernet, Wi-Fi, Bluetooth, HDMI, USB).

    - **Usages in IoT Applications:**
        - **IoT Gateway / Hub:** One of its most popular uses. It can aggregate data from various sensors (connected via GPIO, USB, or wireless protocols like Bluetooth/Zigbee with appropriate dongles/HATs) and communicate with cloud platforms using its built-in Ethernet or Wi-Fi.
        - **Home Automation Systems:** Can act as the central controller for smart home devices, running software like Home Assistant or OpenHAB.
        - **Media Centers with IoT Integration:** Its HDMI output and processing power allow it to be a media center (e.g., running Kodi) while also controlling IoT devices.
        - **Data Logging and Local Servers:** Can host lightweight web servers, databases (e.g., MQTT brokers, InfluxDB) to store and serve IoT data locally.
        - **Robotics and Drones:** Provides sufficient processing power for controlling robots, processing sensor data for navigation, and image processing in drone applications.
        - **Environmental Monitoring Systems:** Can collect data from various environmental sensors, process it, and display it or send it to the cloud.
        - **Industrial IoT (IIoT) Prototyping:** For less critical IIoT applications, it can be used for monitoring machine status, collecting production data, or as a simple HMI.
        - **Educational IoT Projects:** Widely used in education due to its affordability, versatility, and large community support.

3. **BeagleBone (e.g., BeagleBone Black):**
    - **Key Characteristics:** A low-cost, open-source, community-supported single-board computer based on ARM Cortex-A series processors (e.g., Sitara AM335x on BeagleBone Black). Known for its extensive number of GPIO pins and real-time processing capabilities due to its

Programmable Real-Time Units (PRUs). Runs Linux (Debian, Angstrom, Ubuntu).



- **Usages in IoT Applications:**
  - **Robotics and Mechatronics:** The PRUs allow for precise, low-latency control of motors and other actuators, making it excellent for robotics projects that require real-time performance. Its numerous GPIOs are also beneficial.
  - **Industrial Control and Automation:** Suitable for applications requiring deterministic I/O control, such as controlling industrial machinery, 3D printers, or CNC machines.
  - **IoT Gateways with Custom Interfacing:** Its extensive I/O capabilities make it a good choice for gateways that need to interface with a wide variety of sensors and communication modules, including custom or legacy protocols.
  - **Data Acquisition Systems:** Can be used to build systems for collecting data from multiple analog and digital sensors simultaneously, leveraging its ADC and GPIOs.
  - **Embedded Linux Projects with Hardware Interaction:** When a project needs the power of Linux combined with fine-grained control over hardware interfaces, BeagleBone is a strong contender.
  - **Prototyping IoT Devices with Specific I/O Needs:** If an IoT device requires many digital or analog connections, or specialized timing for I/O, BeagleBone's hardware features are advantageous.
  - **Projects requiring Cape Add-on Boards:** Similar to Arduino shields or Raspberry Pi HATs, BeagleBone "Capes" provide specialized hardware functionalities (e.g., motor drivers, sensor interfaces, communication modules) that can be easily integrated for specific IoT tasks.

Summary of Suitability:

- **Intel Galileo:** Good for those needing x86 compatibility, Linux capabilities, and Arduino shield integration for projects that might be more computationally intensive than a standard Arduino can handle.
- **Raspberry Pi:** Best for IoT applications requiring a full OS, significant processing power (especially for tasks like image/video processing, running web servers, complex analytics), robust network connectivity, and ease of use with Python and a large software ecosystem. Excellent as an IoT hub or gateway.

- **BeagleBone:** Shines in applications demanding extensive I/O, real-time control capabilities (thanks to PRUs), and interfacing with a diverse set of hardware. Strong for robotics, industrial control, and custom data acquisition in IoT.

The choice among these boards depends on the specific IoT application's requirements for processing power, I/O needs, real-time capabilities, operating system, programming environment, and community support.

Q5. (a) What are the data-link, network, security and application layer protocols used in the WSNs? (10)

Answer:
Wireless Sensor Networks (WSNs) have unique characteristics such as resource-constrained nodes (low power, memory, processing capability), dense deployment, and often unreliable wireless links. These characteristics necessitate specialized protocols at various layers of the communication stack.

(Ref: Unit 2 Notes, Section 1 - WSN Protocols)

1. **Data-Link Layer Protocols in WSNs:**
    - **Primary Functions:** Responsible for reliable data transfer between two directly connected (neighboring) nodes, framing, medium access control (MAC), and error control.
    - **Key Considerations for WSNs:** Energy efficiency is paramount. MAC protocols must minimize collisions, idle listening, and overhead.
    - **Common Standards/Protocols:**
        - **IEEE 802.15.4:** A widely adopted standard that defines the Physical (PHY) layer and MAC sublayer for Low-Rate Wireless Personal Area Networks (LR-WPANs). It's the foundation for protocols like Zigbee, 6LoWPAN, and WirelessHART.
            - **MAC features:** CSMA/CA for contention-based access, optional Guaranteed Time Slots (GTS) for contention-free access, low power consumption modes.
            - *(Ref: Unit 1 Notes, Section 23 - IEEE 802.15.4; Unit 3 Notes, Section 6.1.D - CSMA/CA)*
        - **Specialized WSN MAC Protocols (often research-based or application-specific, building on or adapting IEEE 802.15.4 concepts):**
            - **S-MAC (Sensor-MAC):** A contention-based protocol that uses periodic listen and sleep cycles to reduce energy consumption due to idle listening. Nodes coordinate their sleep schedules.
            - **T-MAC (Timeout-MAC):** An improvement over S-MAC, T-MAC ends the active period dynamically if no activation event occurs for a certain time, further saving energy.
            - **B-MAC (Berkeley MAC):** Provides a flexible and configurable low-power MAC layer, offering clear channel assessment (CCA) and optional acknowledgments.
            - **Contention-Free (Scheduled) MAC protocols (e.g., TDMA-based):** Nodes are assigned specific time slots for transmission, avoiding collisions and enabling predictable

latency, but requiring synchronization.
        - *(Ref: Unit 3 Notes, Section 6.3.B - TDMA)*
    - **Other considerations:** Link layer acknowledgments and retransmissions for reliability, frame formatting suitable for small packet sizes.
2. **Network Layer Protocols in WSNs:**
    - **Primary Functions:** Responsible for routing data packets from the source sensor node to the sink node or base station, potentially across multiple hops.
    - **Key Considerations for WSNs:** Energy-efficient routing, scalability, fault tolerance, data aggregation capabilities, and adapting to dynamic topologies.
    - **Common Protocols/Approaches:**
        - **RPL (Routing Protocol for Low-Power and Lossy Networks):** [PYQ Q6a (June 2024)] An IETF standard distance-vector routing protocol designed specifically for LLNs, including WSNs and 6LoWPAN networks. It builds Destination Oriented Directed Acyclic Graphs (DODAGs).
            - *(Ref: Unit 2 Notes, Section 8.3.A)*
        - **Ad-hoc Routing Protocols (adapted for WSNs):**
            - **AODV (Ad-hoc On-Demand Distance Vector):** Reactive protocol; routes are discovered only when needed.
            - **DSR (Dynamic Source Routing):** Reactive protocol; the entire route is included in the packet header.
                - *(Ref: Unit 2 Notes, Section 8.3.E)*
        - **Hierarchical/Clustering Protocols:**
            - **LEACH (Low Energy Adaptive Clustering Hierarchy):** Nodes organize into clusters; cluster heads collect data from members and transmit aggregated data to the sink. Rotates cluster heads to distribute energy load.
                - *(Ref: Unit 2 Notes, Section 8.3.A)*
        - **Geographic Routing Protocols:** Use location information of nodes to make routing decisions (e.g., Greedy Routing).
            - *(Ref: Unit 2 Notes, Section 8.3.B)*
        - **Data-Centric Routing:** Data is named by attributes, and sinks request data based on these attributes. Protocols like Directed Diffusion fall into this category.
        - **6LoWPAN (IPv6 over Low-Power Wireless Personal Area Networks):** While an adaptation layer protocol, it enables the use of IPv6 for addressing and routing in WSNs, often working in conjunction with RPL for routing.
            - *(Ref: Unit 1 Notes, Section 23 - 6LoWPAN)*
3. **Security Protocols/Mechanisms in WSNs:**

- **Primary Functions:** Provide confidentiality, integrity, authenticity, and availability of data and network operations.
- **Key Considerations for WSNs:** Resource constraints limit the complexity of cryptographic algorithms and protocols.
- **Mechanisms at Different Layers:**
  - **Link Layer Security:** IEEE 802.15.4 provides security features like AES-CCM* for encryption and authentication at the MAC layer.
  - **Network Layer Security:** Secure routing protocols aim to prevent attacks like sinkhole, wormhole, Sybil attacks.
  - **Transport Layer Security (though less common in raw WSNs):** DTLS (Datagram TLS) can be used if UDP is employed (e.g., with CoAP).
  - **Application Layer Security:** End-to-end encryption can be implemented by the application if lower layers don't provide sufficient security.
- **Specific WSN Security Protocols/Frameworks:**
  - **SPINS (Security Protocols for Sensor Networks):** A suite of security building blocks, including SNEP (Sensor Network Encryption Protocol) for data confidentiality and authentication, and µTESLA for authenticated broadcast.
  - **TinySec:** A link-layer security architecture for WSNs, providing message authentication, integrity, and replay protection.
- **Key Management:** Securely distributing and managing cryptographic keys is a major challenge due to resource constraints and ad-hoc deployment.

4. **Application Layer Protocols in WSNs:**
   - **Primary Functions:** Enable communication between the WSN application and the sensor nodes or the sink. Defines the format and meaning of data exchanged.
   - **Key Considerations for WSNs:** Lightweight, low overhead, suitable for small data payloads, and efficient power usage.
   - **Common Protocols/Approaches:**
     - **CoAP (Constrained Application Protocol):** [PYQ Q3a (June 2024)] Designed for constrained environments, runs over UDP. Provides a RESTful interface (GET, POST, PUT, DELETE) for interacting with resources on sensor nodes. Supports observe functionality for event-driven updates.
       - *(Ref: Unit 1 Notes, Section 22 - Application Layer Protocol)*
     - **MQTT / MQTT-SN (MQTT for Sensor Networks):** [PYQ Q3a (June 2024)]
       - **MQTT:** A lightweight publish/subscribe messaging protocol. MQTT-SN is a version optimized for WSNs, running over UDP or non-IP networks. It uses a gateway to bridge to a standard MQTT broker.
       - *(Ref: Unit 1 Notes, Section 22 - Application Layer Protocol)*

- **Custom/Proprietary Protocols:** Many WSN deployments use application-specific protocols designed to be extremely lightweight and tailored to the exact needs of the application.
- **Data Representation Formats:** Often use compact binary formats rather than verbose formats like XML or JSON to reduce payload size (e.g., CBOR - Concise Binary Object Representation, often used with CoAP).

The choice of protocols at each layer depends heavily on the specific application requirements, network size, node capabilities, energy constraints, and security needs of the WSN.

(b) Explain various node behaviours in WSN? (5)

Answer:
In a Wireless Sensor Network (WSN), nodes exhibit various behaviours depending on their roles, the network architecture, the application requirements, and energy conservation strategies. These behaviours are crucial for the overall functioning and longevity of the network.

(Ref: Unit 2 Notes, Section 1 - Node Behaviours in WSN)

1. **Sensing Behaviour:**
   - **Role:** This is the primary function of most sensor nodes.
   - **Action:** Nodes periodically or aperiodically (event-driven) activate their onboard sensors to measure physical parameters from the surrounding environment (e.g., temperature, humidity, light, motion, chemical concentrations).
   - **Data Conversion:** The sensed analog signals are typically converted to digital data by an ADC (Analog-to-Digital Converter).
2. **Processing Behaviour:**
   - **Role:** Nodes may perform local data processing on the sensed data before transmission.
   - **Action:** This can range from simple tasks like data formatting, thresholding (comparing sensed value against a pre-set limit to detect an event), and filtering noise, to more complex tasks like data compression or feature extraction.
   - **Data Aggregation:** Some nodes (especially cluster heads or intermediate nodes) might aggregate data from multiple sources to reduce redundancy and the total amount of data transmitted, thereby saving energy.
3. **Communication Behaviour (Transmitting & Receiving):**
   - **Role:** Exchanging data with neighboring nodes, cluster heads, or the sink/base station.
   - **Action:**
     - **Transmission:** Sending sensed data, control messages, or routing information.
     - **Reception:** Receiving data packets, queries from the sink, commands for actuation, or routing updates from neighbors.

- **Protocols:** This behaviour is governed by MAC, routing, and application layer protocols to manage channel access, find paths, and ensure reliable delivery.

4. **Routing/Relaying Behaviour:**
   - **Role:** In multi-hop WSNs, many nodes act as relays or routers.
   - **Action:** They forward data packets received from other nodes towards the intended destination (usually the sink). This extends the network's communication range beyond the direct reach of individual nodes.
   - **Decision Making:** Involves using routing protocols to select the next hop for a packet based on criteria like link quality, energy levels, or distance to the sink.
5. **Sleeping and Active Behaviour (Duty Cycling):**
   - **Role:** Critical for energy conservation, especially in battery-powered nodes.
   - **Action:** Nodes typically operate in a duty-cycled manner, alternating between:
     - **Active Mode:** The node is fully operational – sensing, processing, and communicating.
     - **Sleep Mode:** The node powers down most of its components (MCU, radio, sometimes sensors) to minimize energy consumption. It wakes up periodically or based on an external trigger (e.g., an event detected by a low-power sensor).
   - **Coordination:** Sleep schedules often need to be coordinated with neighboring nodes to ensure data can be relayed effectively.
6. **Coordinator Behaviour (Network Management):**
   - **Role:** In some WSN architectures (e.g., star or cluster-based like Zigbee or LEACH), a specific node (or nodes) acts as a coordinator or cluster head.
   - **Action:**
     - **Network Formation:** Initiating the network and allowing other nodes to join.
     - **Synchronization:** Providing timing references for TDMA-based MAC protocols or coordinated sleep schedules.
     - **Address Assignment:** Allocating network addresses to nodes.
     - **Resource Management:** Managing communication slots or bandwidth.
     - **Data Fusion/Aggregation:** Cluster heads often aggregate data from their member nodes before forwarding it to the sink.
7. **Node Discovery Behaviour:**
   - **Role:** When a new node is deployed or an existing node restarts, it needs to discover its neighbors and integrate into the network.
   - **Action:** This can involve broadcasting "hello" or discovery messages, listening for beacons from coordinators or established nodes, and exchanging information to establish communication links and routing paths.
8. **Fault Tolerance/Self-Healing Behaviour:**

- **Role:** WSNs should ideally be resilient to node failures.
- **Action:** If a node fails or a link breaks, surrounding nodes might exhibit behavior to reconfigure routing paths, find alternative routes, or report the failure, thus maintaining network connectivity and functionality to the extent possible.

These varied behaviours allow WSNs to operate autonomously, efficiently, and adaptively in diverse and often challenging environments.

UNIT-III

Q6. (a) Explain, why IoT device nodes use RPL in place of IPv6 and IPv4, why a CoAP client in place of HTTP client and 6LoWPAN at the adaptation layer in place of MAC. (10)

Answer:
The choices of RPL, CoAP, and 6LoWPAN in IoT device nodes are driven by the severe resource constraints (processing power, memory, energy) of these devices and the characteristics of Low-Power and Lossy Networks (LLNs) in which they often operate. Standard internet protocols like IPv4/IPv6 (for routing as-is) and HTTP are too heavy or unsuitable for such environments.

1. **Why IoT device nodes use RPL (Routing Protocol for Low-Power and Lossy Networks) in place of standard IPv6 and IPv4 routing protocols:**
    - **Standard IPv4/IPv6 Routing Protocol Overhead:** Traditional routing protocols designed for wired internet (e.g., OSPF, BGP for inter-domain, RIP, EIGRP for intra-domain) are generally too complex and resource-intensive for constrained IoT devices.
        - **Large Routing Tables:** They often require nodes to maintain large routing tables, consuming significant memory.
        - **Frequent Control Message Exchange:** They involve frequent and verbose control message exchanges (e.g., hello packets, link-state updates) to maintain topology information, which consumes considerable bandwidth and energy.
        - **High Computational Requirements:** Path calculation algorithms can be computationally intensive.
    - **Suitability of RPL for LLNs:** RPL is specifically designed by the IETF for LLNs.
        - **Optimized for Constrained Devices:** It has low memory and processing requirements.
        - **Supports Various Traffic Patterns:** While primarily designed for multipoint-to-point traffic (common in WSNs where data flows to a sink/gateway), it can also support point-to-multipoint and point-to-point traffic.
        - **Topology Construction (DODAGs):** RPL constructs a Destination-Oriented Directed Acyclic Graph (DODAG). Nodes establish routes towards a root node (often a gateway or border router). This structure is efficient for upward data flows.
            - *(Ref: Unit 2 Notes, Section 8.3.A)*

        - **Trickle Timer for Control Messages:** RPL uses a Trickle timer mechanism to control the rate of sending control messages (e.g., DIO - DODAG Information Object). This reduces control overhead by sending messages less frequently when the network is stable and more frequently when changes occur.
        - **Objective Functions (OF):** RPL is flexible and allows different Objective Functions to be used for selecting parents and constructing routes based on various metrics (e.g., ETX (Expected Transmissions), hop count, energy levels), making it adaptable to different network goals.
        - **Loop Avoidance and Detection:** Includes mechanisms for loop avoidance and detection.
        - **Repair Mechanisms:** Can repair routes locally or globally when links fail.
    - **IPv4 Unsuitability:** IPv4 has a limited address space, which is a major issue for the billions of devices envisioned in IoT. While NAT can extend its life, it adds complexity and breaks end-to-end connectivity.
    - **IPv6 Suitability (Addressing, but not standard routing):** IPv6 provides a vast address space crucial for IoT. However, standard IPv6 routing protocols are still too heavy. RPL operates *over* IPv6, providing the routing logic specifically tailored for LLNs.
2. **Why a CoAP (Constrained Application Protocol) client in place of HTTP client:**
    - **HTTP Overhead:** HTTP is a text-based protocol with verbose headers and typically runs over TCP. This makes it:
        - **Bandwidth Intensive:** Large headers and text payloads consume significant bandwidth, which is scarce in LLNs.
        - **Processing Intensive:** Parsing text-based headers and managing TCP connections requires more processing power and memory than available on many constrained IoT devices.
        - **Power Intensive:** TCP's connection setup, maintenance (keep-alives), and acknowledgments, along with larger packet sizes, lead to higher energy consumption.
    - **Suitability of CoAP for Constrained Environments:**
        - **Lightweight and Binary:** CoAP uses a compact binary header (4 bytes) and concise message formats, significantly reducing overhead compared to HTTP.
            - *(Ref: Unit 1 Notes, Section 22 - Application Layer Protocol; PYQ Q3a (June 2024))*
        - **UDP-based:** Runs over UDP, which is connectionless and has lower overhead than TCP. This is suitable for lossy networks and devices that need to sleep frequently.
        - **RESTful Model:** Provides a RESTful interaction model (GET, POST, PUT, DELETE) similar to HTTP, making it familiar and allowing easy mapping to web services, but in a constrained-friendly way.
        - **Asynchronous Communication:** Supports both confirmable (reliable) and non-confirmable (unreliable) messages, and separate responses, allowing for asynchronous interactions suitable for sleepy devices.

- **Observe Functionality:** Enables clients to subscribe to resource changes on a server, avoiding the need for inefficient polling by the client. The server notifies observers of changes.
- **Built-in Resource Discovery:** Has simple mechanisms for discovering resources on a CoAP server.
- **DTLS for Security:** Can be secured using DTLS (Datagram TLS) for secure communication over UDP.

3. **Why 6LoWPAN (IPv6 over Low-Power Wireless Personal Area Networks) at the adaptation layer in place of MAC:**
    - **MAC Layer Function:** The MAC (Medium Access Control) layer (e.g., IEEE 802.15.4 MAC) is responsible for framing data for transmission over the physical wireless medium, managing access to the shared medium (e.g., using CSMA/CA), and addressing within the local link (using MAC addresses). It does *not* inherently provide internetworking capabilities or global addressing like IP.
        - *(Ref: Unit 3 Notes, Section 3 & 6.1.D)*
    - **Need for IP in IoT:** For IoT devices to be truly part of the "Internet of Things," they need to be addressable and communicate using Internet Protocol (IP). IPv6 is preferred due to its vast address space.
    - **The "Impedance Mismatch":**
        - **IPv6 Packet Size:** Standard IPv6 packets have a minimum MTU (Maximum Transmission Unit) of 1280 bytes.
        - **IEEE 802.15.4 Frame Size:** The MAC frames of low-power wireless standards like IEEE 802.15.4 are much smaller (e.g., 127 bytes total, with a payload of around 80-100 bytes after headers and security).
        - Directly sending an IPv6 packet over an IEEE 802.15.4 frame is not possible without significant fragmentation.
    - **Role of 6LoWPAN as an Adaptation Layer:** 6LoWPAN is an adaptation layer that sits *between* the IP (Network) layer and the MAC layer (e.g., IEEE 802.15.4). Its functions are:
        - **Header Compression:** It compresses the verbose IPv6 headers (and UDP headers) to reduce their size significantly, making them fit within the small MAC frames of LLNs.
            - *(Ref: Unit 1 Notes, Section 23 - 6LoWPAN)*
        - **Fragmentation and Reassembly:** If an IPv6 packet (even after compression) is still too large for a single MAC frame, 6LoWPAN defines mechanisms to fragment the packet into smaller pieces at the sender and reassemble them at the receiver.
        - **Link-Layer Addressing:** Facilitates the mapping between IPv6 addresses and link-layer (MAC) addresses.
        - **Mesh Under Routing:** Can support mesh routing where routing decisions are made at the 6LoWPAN layer (below IP but above MAC) using link-layer addresses if full IP routing is too

heavy for all nodes.

- **Bootstrapping:** Helps in network auto-configuration.

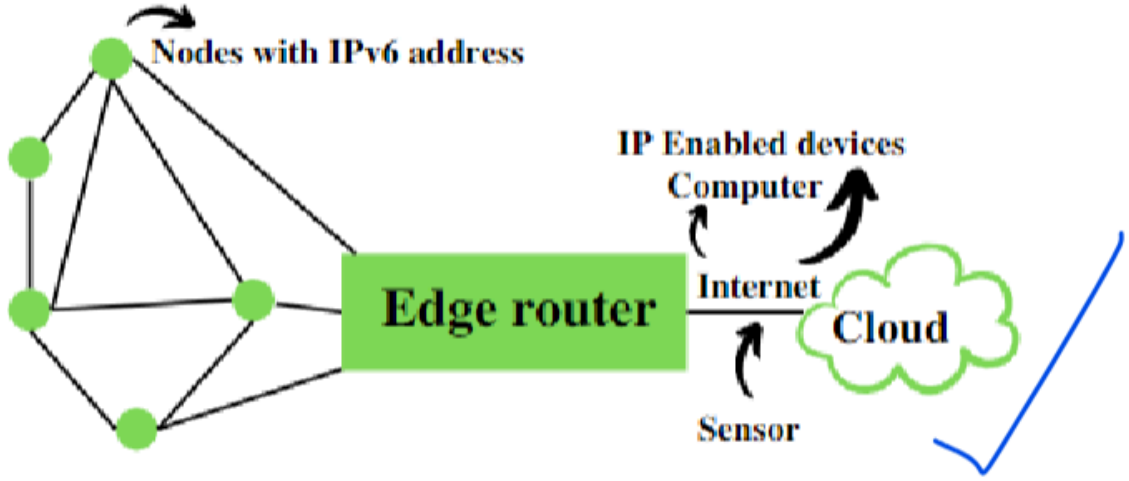
- **Why not just MAC?** Using only MAC layer protocols would limit communication to the local link layer network. Devices wouldn't have global IP addresses, couldn't directly participate in IP-based internet communication, and internetworking with other IP networks would require complex application-layer gateways. 6LoWPAN enables end-to-end IP connectivity for even very constrained devices by adapting IPv6 to the specific constraints of low-power wireless MAC layers.

In summary, RPL, CoAP, and 6LoWPAN are specifically designed or adapted to overcome the limitations of constrained devices and lossy networks, enabling them to participate in IP-based IoT ecosystems more efficiently than their traditional internet counterparts (standard IP routing, HTTP, and direct MAC layer usage for internetworking).

(b) What are the header fields in 6LoWPAN? (5)

Answer:
6LoWPAN (IPv6 over Low-Power Wireless Personal Area Networks) is an adaptation layer designed to allow IPv6 packets to be carried efficiently over low-power, lossy networks (LLNs) like those based on IEEE 802.15.4. A key function of 6LoWPAN is header compression to reduce the overhead of standard IPv6 and UDP headers, making them fit into the small frames of LLNs.

The 6LoWPAN specification defines several "dispatch" types and associated header formats that indicate how an IPv6 packet (or parts of it) is encapsulated and compressed. Instead of a single fixed header, 6LoWPAN uses a chain of headers identified by dispatch values.



(Contextual 6LoWPAN diagram)

The primary header fields introduced or managed by 6LoWPAN relate to:

1. **Dispatch Header (Type field):**

- This is the first part of a 6LoWPAN frame and indicates the type of header or payload that follows. It's a variable-length field (typically 1 or 2 bytes).
- Specific bit patterns in the dispatch field identify:
  - **HC1 (IPv6 Header Compression):** Indicates that IPv6 header compression is being used. Further bits specify how fields like Traffic Class, Flow Label, Source/Destination Addresses, and Next Header are compressed or elided.
  - **HC_UDP (UDP Header Compression):** If UDP is the next header, this indicates UDP header compression, compressing source/destination ports and checksum.
  - **IPHC (IP Header Compression):** A more advanced and common form of IPv6 header compression (RFC 6282) that replaces the older HC1. It uses context information to achieve high compression ratios.
  - **Fragmentation Headers (FRAG1, FRAGN):** Used when an IPv6 packet needs to be fragmented to fit into multiple IEEE 802.15.4 frames.
    - **FRAG1:** For the first fragment. Contains `datagram_size` (total size of the original packet) and `datagram_tag` (unique identifier for the packet).
    - **FRAGN:** For subsequent fragments. Contains `datagram_size`, `datagram_tag`, and `datagram_offset` (offset of this fragment within the original packet).
  - **Mesh Addressing Headers:** Used in mesh-under routing scenarios where routing is done at the 6LoWPAN layer using link-layer addresses. Contains fields for originator and final destination link-layer addresses, and a hop limit.
  - **Broadcast Header:** For link-layer broadcast in a mesh network.
  - **ESC (Escape Dispatch):** Indicates that the following byte is an IPv6 Next Header field (if no compression is applied for that specific header) or another dispatch type.
  - **Uncompressed IPv6:** A dispatch value indicating that a full, uncompressed IPv6 header follows (used when compression is not possible or not configured).

2. **IPHC (IP Header Compression) Encoded Fields (RFC 6282):**
   - The IPHC dispatch byte itself is followed by one or more bytes that encode how various IPv6 header fields are compressed. Key aspects include:
     - **TF (Traffic Class & Flow Label):** Specifies how these fields are compressed (e.g., elided, carried inline).
     - **NH (Next Header):** Specifies if the Next Header field is carried inline or inferred (e.g., UDP if UDP compression follows).
     - **HLIM (Hop Limit):** Specifies how the Hop Limit is carried (e.g., inline, common values like 1, 64, 255).
     - **CID (Context Identifier Extension):** If context-based compression is used for addresses.
     - **SAC (Source Address Compression):** Specifies if the source address is stateless (derived from link-layer, elided) or stateful (context-based, short address).

        - **SAM (Source Address Mode):** Further details on source address compression (e.g., 16-bit, 64-bit inline, fully inline).
        - **M (Multicast Compression):** Indicates if multicast address compression is used.
        - **DAC (Destination Address Compression):** Similar to SAC for the destination address.
        - **DAM (Destination Address Mode):** Similar to SAM for the destination address.
    - Depending on these flags, some IPv6 fields might be elided entirely, carried as short identifiers, or carried inline partially or fully.
3. **UDP Header Compression Fields (LOWPAN_NHC for UDP - RFC 6282):**
    - If UDP is the next header and compression is enabled (via IPHC NH flag), specific bits indicate:
        - **Port Compression:** Source and destination ports can be compressed if they fall within a predefined range (e.g., 61616-61631) or carried inline (8-bit or 16-bit).
        - **Checksum Compression:** The UDP checksum can be elided (and recalculated by the decompressor) or carried inline.

In summary, 6LoWPAN doesn't define a single fixed header format like IPv6 or TCP. Instead, it uses a system of dispatch values to indicate a chain of (potentially compressed) headers or specific 6LoWPAN functionalities like fragmentation or mesh addressing. The most important "header fields" are within these dispatch types, particularly for IPHC, UDP compression, and fragmentation, which manage how the original IPv6/UDP headers are reduced in size or how large packets are segmented.

Q7. (a) What is MAC address? How does a MAC address assign to an IoT node? How does address resolved to enable packets in Ip network reach the node. (10)

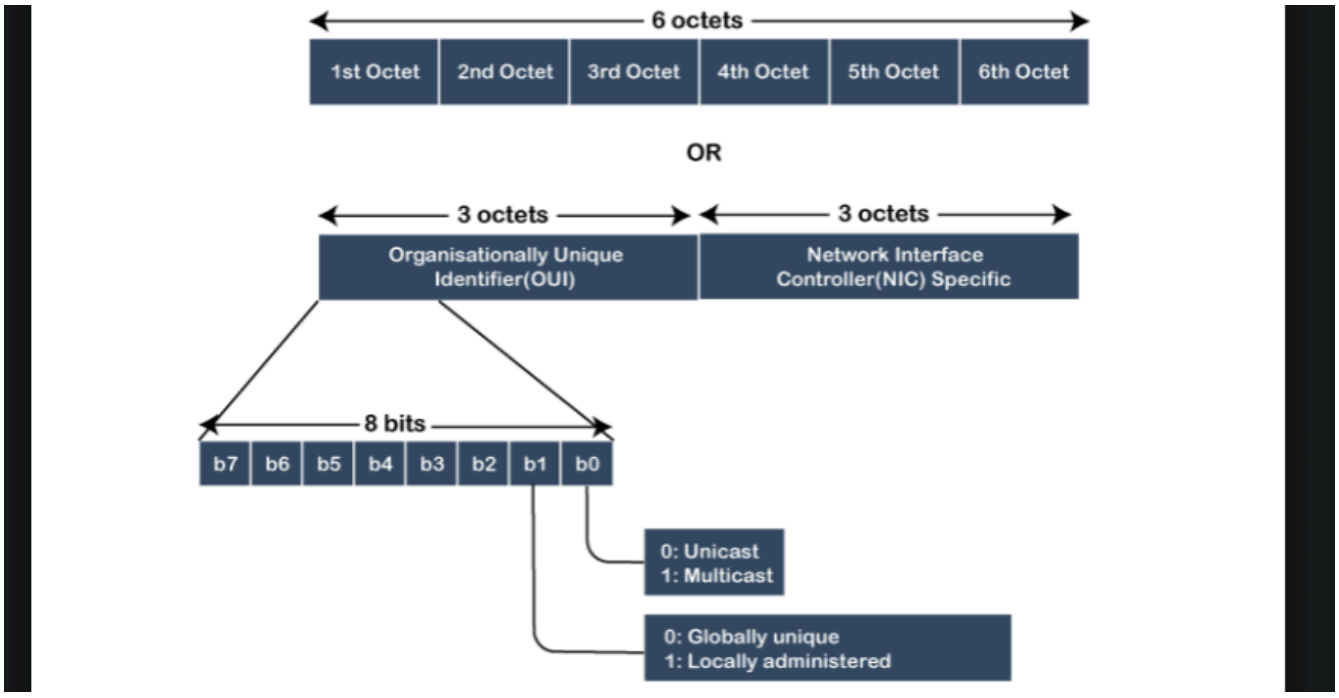
Answer:

(Ref: Unit 3 Notes, Section 4 - MAC (Medium Access Control) Layer, specifically subsections 4.1, 4.2, 4.4, 4.5)

What is a MAC Address?

A **MAC (Medium Access Control) address** is a unique hardware identification number assigned to a Network Interface Controller (NIC) for use as a network address in communications within a network segment (like a Local Area Network - LAN). It is also known as a physical address or hardware address.

- **Format:** A MAC address is a 48-bit (6-byte) number, typically represented as 12 hexadecimal digits, often grouped in pairs separated by colons or hyphens (e.g., `00:1A:2B:3C:4D:5E` or `00-1A-2B-3C-4D-5E`).



(Illustrative MAC address format)

- **Structure:**
  - **First 24 bits (OUI - Organizationally Unique Identifier):** Assigned by the IEEE (Institute of Electrical and Electronics Engineers) to the manufacturer of the NIC. This part identifies the vendor.
  - **Last 24 bits (NIC Specific/Device ID):** Assigned by the manufacturer and is unique to that specific piece of hardware (NIC) within that OUI.
- **Layer of Operation:** MAC addresses operate at the Data Link Layer (Layer 2) of the OSI model.
- **Purpose:** To uniquely identify devices on a local network segment and enable data frames to be delivered to the correct physical device within that segment.

How does a MAC address assign to an IoT node?

A MAC address is assigned to an IoT node (or more precisely, to its network interface(s)) in the following way:

1. **Burned-In Address (BIA):**
   - The MAC address is typically "burned-in" or hard-coded into the firmware or Read-Only Memory (ROM) of the Network Interface Controller (NIC) chip by the manufacturer during the manufacturing process. This is the primary and physical MAC address.
   - Manufacturers purchase blocks of OUIs from the IEEE and then assign unique 24-bit extensions to each NIC they produce, aiming for global uniqueness for each hardware interface.
2. **Multiple Interfaces:**
   - An IoT node might have multiple network interfaces (e.g., a Wi-Fi interface and an Ethernet interface, or a Bluetooth interface). Each of these distinct network interfaces will have its own unique MAC address assigned by its respective chip manufacturer.

3. **Software-Assigned (Less Common for True Physical Address):**
    - While the BIA is the true hardware address, some operating systems or device drivers allow the MAC address to be changed or "spoofed" in software. This doesn't change the BIA on the hardware itself but alters the MAC address the device presents to the network. This is sometimes used for network testing, to bypass MAC filtering, or for privacy reasons, but it's not how the *original* MAC address is assigned.

So, for an IoT node, each of its network communication modules (Wi-Fi chip, Ethernet controller, Bluetooth module, LoRa module with an EUI, etc.) will come with a pre-assigned, burned-in MAC address (or a similar unique Layer 2 identifier like EUI-64 for Zigbee/IEEE 802.15.4).

How does address resolved to enable packets in IP network reach the node?

When a device (Source A) wants to send an IP packet to another device (Destination B) on the *same local IP network segment*, it needs to know Destination B's MAC address to create the Layer 2 frame that will carry the IP packet. If they are on different IP networks, Source A will need the MAC address of its default gateway (router). The process of resolving an IP address to a MAC address is typically done using:

1. **ARP (Address Resolution Protocol) for IPv4:**
    - **Scenario:** Device A knows the IP address of Device B (e.g., `192.168.1.10`) but not its MAC address.
    - **ARP Request:** Device A broadcasts an ARP Request packet onto the local network. This packet essentially asks, "Who has the IP address `192.168.1.10`? Tell my MAC address (`AA:AA:AA:AA:AA:AA`)."
    - **ARP Reply:** All devices on the local network receive the broadcast. Device B, recognizing its own IP address in the request, responds directly to Device A with an ARP Reply packet. This packet says, "IP address `192.168.1.10` is at MAC address `BB:BB:BB:BB:BB:BB`."
    - **ARP Cache:** Device A receives the ARP reply and stores the IP-to-MAC address mapping (`192.168.1.10` -> `BB:BB:BB:BB:BB:BB`) in its ARP cache (or ARP table) for a limited time. This avoids repeating the ARP process for recent communications.
    - **Frame Creation:** Device A can now create an Ethernet frame (or other Layer 2 frame) with Destination B's MAC address (`BB:BB:BB:BB:BB:BB`) and its own source MAC address, encapsulating the IP packet destined for Device B.
2. **NDP (Neighbor Discovery Protocol) for IPv6:**
    - NDP serves a similar purpose for IPv6 as ARP does for IPv4, but it uses ICMPv6 messages.
    - **Neighbor Solicitation (NS):** Similar to an ARP Request, a device sends an NS message to discover the link-layer (MAC) address of a neighbor on the same link, or to verify reachability.
    - **Neighbor Advertisement (NA):** Similar to an ARP Reply, a device responds to an NS message with its link-layer address, or sends unsolicited NAs to announce address changes.
    - **Neighbor Cache:** Devices maintain a neighbor cache similar to an ARP cache.

Process for IP Packet Delivery to an IoT Node:

1. **Application Data:** An application on a source device wants to send data to an IoT node's IP address.
2. **Transport Layer:** Data is encapsulated with a TCP or UDP header (creating a segment/datagram).
3. **Network Layer:** The segment/datagram is encapsulated with an IP header (creating an IP packet), containing source and destination IP addresses.
4. **Routing Decision:**
    - **Same Network:** If the destination IP address is on the same local network as the source, the source device uses ARP/NDP to find the destination IoT node's MAC address.
    - **Different Network:** If the destination IP address is on a different network, the source device uses ARP/NDP to find the MAC address of its default gateway (router). The IP packet is destined for the remote IoT node, but the Layer 2 frame is addressed to the gateway's MAC address.
5. **Data Link Layer Framing:** The IP packet is encapsulated into a Layer 2 frame (e.g., Ethernet frame). This frame includes:
    - Destination MAC Address (either the IoT node's MAC or the gateway's MAC).
    - Source MAC Address.
    - Payload (the IP packet).
6. **Physical Layer Transmission:** The frame is transmitted as bits over the physical medium.
7. **Switching/Routing:**
    - **Switches** (Layer 2) on the local network use the destination MAC address to forward the frame to the correct port leading to the IoT node or the next switch.
    - If sent to a **router** (default gateway), the router de-encapsulates the IP packet, checks its routing table for the destination IP network, determines the next hop router (or if it's directly connected), performs ARP/NDP for the next hop's MAC address (if on an Ethernet-like segment), and re-encapsulates the IP packet into a new Layer 2 frame for the next link. This process repeats until the packet reaches the router connected to the IoT node's local network.
8. **Final Delivery:** The router on the IoT node's local network uses ARP/NDP to find the IoT node's MAC address and delivers the frame. The IoT node's NIC receives the frame, checks the destination MAC address, and if it matches, passes the IP packet up to its network layer for further processing.

This resolution of IP addresses to MAC addresses is fundamental for enabling IP packets to traverse local network segments and ultimately reach the target IoT node.

(b) A subnet mask is 11111111 11111111 11111111 10010000. IP address is 198.136.56.2. How do these figures provide subnet and host addresses? (5)

Answer:
To understand how the subnet mask and IP address provide subnet and host addresses, we first need to convert them into a more common and readable format.

1. Convert Subnet Mask to Dotted Decimal Notation:
The given subnet mask in binary is:
`11111111 11111111 11111111 10010000`

Let's convert each octet (8-bit group) to decimal:

- `11111111` = 255
- `11111111` = 255
- `11111111` = 255
- `10010000` = (1 * 128) + (0 * 64) + (0 * 32) + (1 * 16) + (0 * 8) + (0 * 4) + (0 * 2) + (0 * 1) = 128 + 16 = 144

So, the subnet mask in dotted decimal notation is: **255.255.255.144**

2. Convert IP Address to Binary Notation:
The given IP address is: **198.136.56.2**

Let's convert each octet to binary:

- `198` = `11000110`
- `136` = `10001000`
- `56` = `00111000`
- `2` = `00000010`

So, the IP address in binary is:
`11000110 . 10001000 . 00111000 . 00000010`

3. Understanding Subnet Mask Function:
A subnet mask is used to divide an IP address into two parts:

- **Network/Subnet Portion:** The bits in the IP address that correspond to the '1's in the subnet mask. This part identifies the specific network or subnet.
- **Host Portion:** The bits in the IP address that correspond to the '0's in the subnet mask. This part identifies a specific device (host) within that network/subnet.

4. Determining the Network/Subnet Address:
To find the network (or subnet) address, we perform a bitwise AND operation between the IP address and the subnet mask.

IP Address (binary): `11000110 . 10001000 . 00111000 . 00000010`
Subnet Mask (binary): `11111111 . 11111111 . 11111111 . 10010000`

Network Address (binary): `11000110 . 10001000 . 00111000 . 00000000`

Converting the Network Address back to dotted decimal:

- `11000110` = 198
- `10001000` = 136
- `00111000` = 56
- `00000000` = 0

So, the **Network/Subnet Address is: 198.136.56.0**

5. Determining the Host Address Portion:
The host portion is represented by the bits in the IP address where the subnet mask has '0's.
Subnet Mask: `11111111.11111111.11111111.10010000`
The first 24 bits + 2 bits from the last octet (24+2 = 26 bits) are for the network/subnet.
The remaining 32 - 26 = 6 bits are for the host portion.

IP Address (last octet): `00000010`
Subnet Mask (last octet):`10010000`

The host bits in the IP address are the last 6 bits of the last octet: `000010`.
In the given IP address `198.136.56.2`, the host part is specifically `000010` which is decimal 2.

How these figures provide subnet and host addresses:

- **Subnet Identification:**
  - The subnet mask `255.255.255.144` indicates that the first 26 bits (all of the first three octets and the first two bits of the fourth octet) represent the network or subnet portion.
  - The network address `198.136.56.0` is the identifier for this specific subnet. All devices within this subnet will share this same network prefix.
- **Host Identification:**
  - The remaining 6 bits (32 total bits - 26 network/subnet bits) in the IP address are used to identify individual hosts within the `198.136.56.0` subnet.
  - For the given IP address `198.136.56.2`, the host portion in binary is `000010` (decimal 2). This uniquely identifies this specific device on the `198.136.56.0` subnet.
- **Number of Subnets and Hosts (Additional Information):**
  - The fourth octet of the subnet mask is `10010000`. The '1's that extend beyond the natural class boundary (assuming this was originally a Class C, the natural mask is /24) indicate subnetting. Here, 2 bits (`10` from `10010000`) are borrowed from the host portion for subnetting (if we

consider the base network to be `198.136.56.0/24`). This would allow for $2^2 = 4$ subnets if those were the only subnet bits. However, given the mask `10010000`, it implies specific subnet ranges.

- The number of bits for hosts is 6 (the number of '0's in the host portion of the mask). This allows for $2^6 - 2 = 64 - 2 = 62$ usable host addresses per subnet (we subtract 2 for the network address itself and the broadcast address).
- The subnets created by `255.255.255.144` would increment in blocks. The interesting bits in the fourth octet are `1001xxxx`. The block size is 256 - 144 = 112, which is not standard for subnetting directly from a /24. Let's look at the bits:
  `10010000` means the network bits are the first 26.
  The last octet increments in blocks based on the position of the last '1' in the mask.
  Mask: `10010000`
  This specific mask might be unusual, but if interpreted as the network boundary:
  The subnets available would be:
  `198.136.56.0` (Host range: 198.136.56.1 to 198.136.56.62, Broadcast: 198.136.56.63)
  `198.136.56.64` (Host range: 198.136.56.65 to 198.136.56.126, Broadcast: 198.136.56.127)
  `198.136.56.128` This one doesn't quite align if we strictly interpret 10010000.

Let's re-evaluate the subnet bits if we consider a Class C base: `198.136.56.0 /24`.
The subnet mask `255.255.255.144` is `/26` because `11111111.11111111.11111111.10010000`.
The mask should be contiguous 1s followed by 0s. `10010000` is not a standard representation for the end of network bits in a simple subnet mask if it implies subnetting a /24 network. A standard /26 mask would be `11000000` (192).
Assuming the question implies the mask is exactly as given and valid:
The network portion is defined by the 1s: The first three octets are fully network (`198.136.56`).
In the last octet, the mask is `10010000`. The IP's last octet is `00000010`.
ANDing them:
`00000010` (IP)
`10010000` (Mask)

`00000000` (Network part of last octet)
So the **Subnet Address is 198.136.56.0**.

The host bits are where the mask has 0s. In `10010000`, the host bits are at positions represented by `x` in `1001xxxx`.
Actually, it's simpler: Network bits are where mask is 1. Host bits are where mask is 0.
Mask: `11111111.11111111.11111111.10010000`

IP: `11000110.10001000.00111000.00000010`
Network part from IP: `11000110.10001000.00111000.00000000` (where mask has 1 at the first bit of last octet, and another 1 at the fourth bit of last octet). This interpretation of "network part" from a non-contiguous mask is non-standard.

Standard Interpretation (assuming the mask defines the boundary between network and host bits contiguously):
If the mask implies the *number of network bits*, and `10010000` is just the binary of the last octet (144):
The mask `255.255.255.144` has `8+8+8+2 = 26` leading ones if it were `255.255.255.192` (11000000) or 27 if `255.255.255.224` (11100000). The mask `10010000` (144) is not a standard contiguous CIDR mask.

However, if we take the problem literally, the bitwise AND for the network address is correct:
Subnet Address: 198.136.56.0

To find the host address within this subnet, we invert the subnet mask and AND it with the IP address:

Subnet Mask (binary): `11111111 . 11111111 . 11111111 . 10010000`

Inverted Mask (binary): `00000000 . 00000000 . 00000000 . 01101111`

IP Address (binary): `11000110 . 10001000 . 00111000 . 00000010`

Host Part (binary): `00000000 . 00000000 . 00000000 . 00000010`
(Result of IP AND Inverted_Mask)

So, the **Host address component is 0.0.0.2** relative to the subnet.
The specific host identifier on the subnet `198.136.56.0` is `.2`.

In summary:

- The **Subnet Address** is found by `IP_Address AND Subnet_Mask`, which is `198.136.56.0`.
- The **Host Address** portion within that subnet is identified by the bits in the IP address where the subnet mask is 0. For IP `198.136.56.2` on subnet `198.136.56.0` (with mask `255.255.255.144`), the host identifier effectively corresponds to the value `2` in the host part of the address. The non-contiguous nature of '1's in the last octet of the provided mask (`10010000`) is unusual for standard subnetting practices which use a contiguous block of '1's followed by '0's. If interpreted strictly by bitwise AND for network address, the result stands.

UNIT-IV

Q8. What is Arduino UNO? Explain its components with Pin Structure. List its features. (15)

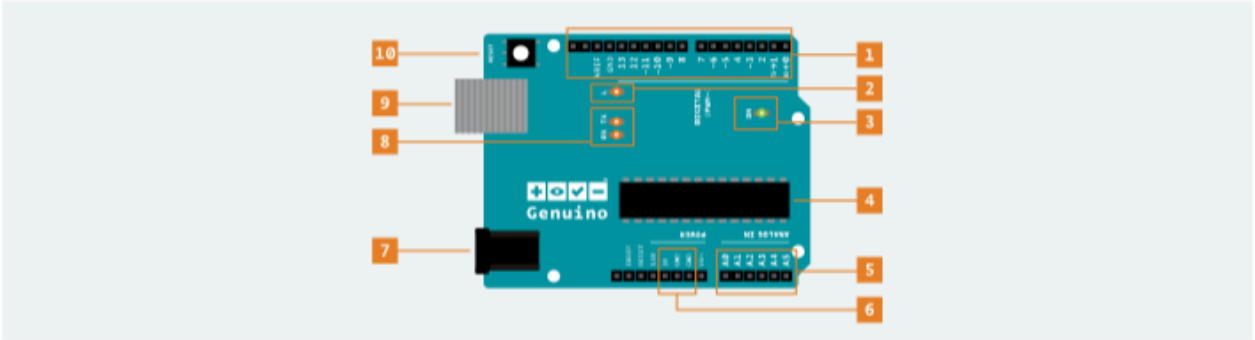
Answer:

(Ref: Unit 2 Notes, Section 4.A; Unit 4 Notes, Section 2 & 3)

What is Arduino UNO?

The Arduino Uno is one of the most popular and widely used microcontroller development boards in the Arduino family. It is an open-source electronics platform based on the Microchip ATmega328P microcontroller. It is designed to be easy to use, making it an excellent choice for beginners, hobbyists, students, and professionals for prototyping interactive electronic projects, especially in the context of IoT.

The "Uno" means 'one' in Italian and was named to mark the release of Arduino Software (IDE) 1.0. It provides a simple hardware and software environment to get started with microcontrollers and electronics. Users can write programs (called "sketches") using the Arduino IDE and upload them to the board to control various electronic components like sensors, LEDs, motors, and displays.



(Arduino Uno Board Image)

Components of Arduino UNO and Pin Structure:

The Arduino Uno board has several key components and a well-defined pin structure:

Architecture of Arduino board :

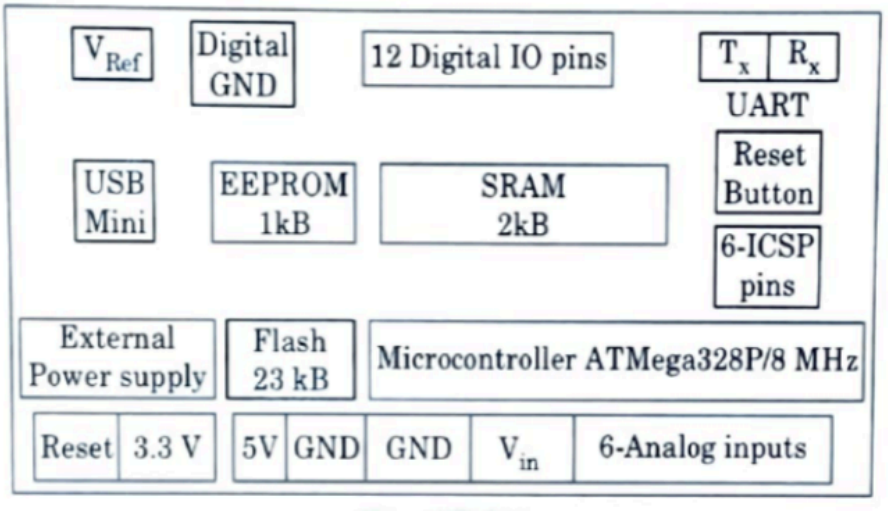
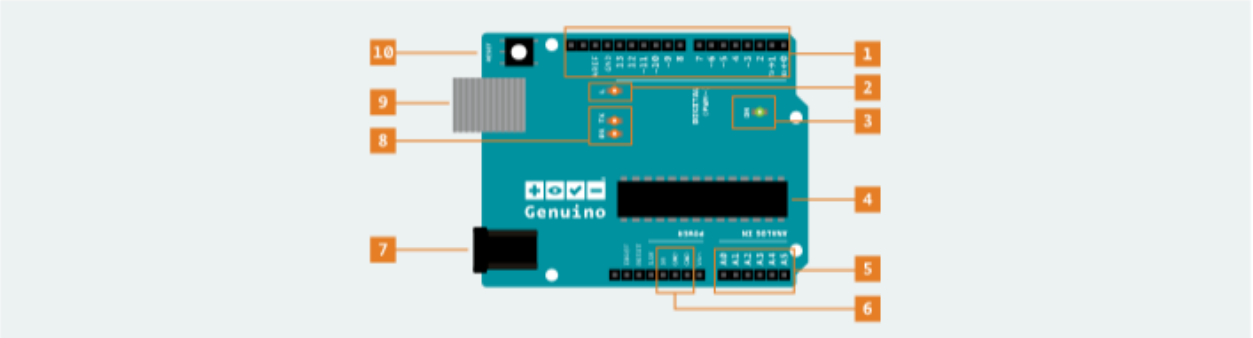


Fig. 2.17.1.

(Arduino Uno Architecture/Block Diagram)



(Annotated Arduino Uno Board for Pin Structure)

1. **Microcontroller (ATmega328P):**
    - This is the "brain" of the Arduino Uno. It's an 8-bit AVR microcontroller.
    - **Function:** Executes the uploaded sketches (programs), performs logical operations, and controls all other components and connected peripherals.
    - **Specifications:** Typically includes Flash memory (32KB, with 0.5KB used by bootloader for Uno R3) for storing sketches, SRAM (2KB) for storing variables during runtime, and EEPROM (1KB) for non-volatile data storage. Operates at a clock speed of 16 MHz.
2. **Digital I/O Pins (Pins 0-13):**
    - **Structure:** 14 digital input/output pins.
    - **Function:** Can be configured as either an INPUT or an OUTPUT using the `pinMode()` function.
        - As **INPUT:** Can read digital signals (HIGH or LOW, corresponding to 5V or 0V).
        - As **OUTPUT:** Can output digital signals (HIGH or LOW) to control devices like LEDs, relays.
    - **PWM (Pulse Width Modulation):** Pins marked with a tilde (~) (pins 3, 5, 6, 9, 10, 11 on the Uno) can provide analog-like output using PWM via the `analogWrite()` function. This is useful for controlling LED brightness or DC motor speed.
    - **Serial Communication:** Pins 0 (RX - Receive) and 1 (TX - Transmit) are used for TTL serial communication. These are connected to the ATmega16U2 USB-to-TTL Serial chip for communication with a computer via USB. *Using these pins for digital I/O will interfere with serial communication.*
3. **Analog Input Pins (Pins A0-A5):**
    - **Structure:** 6 analog input pins.
    - **Function:** Can read analog voltage signals (typically 0-5V) from analog sensors (e.g., potentiometers, temperature sensors like LM35, light-dependent resistors - LDRs).
    - **ADC (Analog-to-Digital Converter):** These pins are connected to a built-in 10-bit ADC, which converts the analog voltage into a digital value ranging from 0 to 1023. This value can be read using the `analogRead()` function.
    - They can also be used as digital I/O pins if needed.
4. **Power Pins:**

- **Vin:** Input voltage for the Arduino board when using an external power source (recommended 7-12V).
- **5V:** Provides a regulated 5V output from the onboard voltage regulator or USB power. Can be used to power external components.
- **3.3V:** Provides a regulated 3.3V output, generated by the onboard voltage regulator. Useful for powering sensors and modules that require 3.3V.
- **GND:** Ground pins. Multiple GND pins are available.
- **IOREF:** Provides the voltage reference with which the microcontroller operates (e.g., 5V for the Uno). Shields can use this to select the appropriate power level.
- **Reset Pin:** A pin that, when brought LOW, will reset the microcontroller.

5. **USB Port (Type B):**
   - **Function:**
     - **Programming:** Used to upload sketches from the Arduino IDE on a computer to the ATmega328P microcontroller.
     - **Power:** Can power the Arduino board (provides 5V).
     - **Serial Communication:** Enables serial communication between the Arduino and a computer (e.g., for sending debug messages using `Serial.println()`, or receiving data from the computer).
6. **Power Connector (Barrel Jack):**
   - **Function:** Allows the Arduino Uno to be powered from an external power supply (e.g., an AC-to-DC adapter or a battery).
   - **Voltage:** Accepts voltages between 7-12V (recommended). The onboard voltage regulator converts this to a stable 5V for the microcontroller and other components.
7. **Reset Button:**
   - **Function:** A physical button that, when pressed, resets the ATmega328P microcontroller, causing the sketch to restart from the beginning.
8. **Crystal Oscillator:**
   - **Function:** Provides the clock signal for the microcontroller. On the Arduino Uno, it is typically a 16 MHz crystal oscillator. This ensures accurate timing for the microcontroller's operations.
9. **ICSP (In-Circuit Serial Programming) Header:**
   - **Structure:** A 6-pin header (2x3 pins).
   - **Function:** Allows for direct programming of the ATmega328P microcontroller (and the ATmega16U2 USB interface chip on some revisions) using an external programmer, bypassing the bootloader. Useful for burning a new bootloader or for low-level programming.
10. **LEDs:**
    - **Power LED (ON):** Indicates that the board is powered on.

- **Pin 13 LED (L):** An onboard LED connected to digital pin 13. Useful for simple visual output and debugging (e.g., the "Blink" sketch).
- **TX and RX LEDs:** Transmit (TX) and Receive (RX) LEDs flicker when data is being transmitted or received via serial communication (e.g., during sketch upload or when using `Serial` functions).

11. **Voltage Regulator:**
    - **Function:** Takes the input voltage from the barrel jack or Vin pin and regulates it down to a stable 5V (and also provides 3.3V) to power the microcontroller and other components safely.

List of Features of Arduino UNO:

- **Open-Source Hardware and Software:** Schematics, board layouts, and software are freely available, fostering a large community and enabling customization.
- **Ease of Use:** Designed for beginners with a simple programming language (based on C/C++) and an intuitive IDE.
- **Cross-Platform IDE:** The Arduino IDE runs on Windows, macOS, and Linux.
- **Microcontroller Based:** Uses the ATmega328P, a capable 8-bit microcontroller.
- **Input/Output Capabilities:** 14 digital I/O pins (6 of which provide PWM output) and 6 analog input pins.
- **Communication Interfaces:** Built-in support for Serial (UART), I2C, and SPI communication.
- **USB Connectivity:** For programming, power, and serial communication.
- **Extensive Libraries:** A vast collection of pre-written code libraries simplifies interfacing with various sensors, actuators, and communication modules.
- **Large Community Support:** Abundant tutorials, forums, and projects available online, making it easy to find help and inspiration.
- **Expandability (Shields):** Supports add-on boards called "shields" that plug into the Arduino headers to provide additional functionalities (e.g., Ethernet, Wi-Fi, motor drivers, GPS).
- **Affordability:** Relatively low cost for the board and components.
- **Versatile Power Options:** Can be powered via USB or an external power supply.
- **Onboard LEDs:** For power indication, serial communication status (TX/RX), and a user-programmable LED on pin 13.
- **Reset Button:** For easy restarting of the microcontroller program.
- **ICSP Header:** For advanced programming and bootloader flashing.

Q9. Draw a circuit diagram of connecting DHT sensor with Arduino. Explain its pin structure. Write a program to connect DHT sensor with Arduino to read the Temperature and Humidity. (15)

Answer:

The DHT series of sensors (e.g., DHT11, DHT22/AM2302) are popular low-cost sensors for measuring temperature and relative humidity. They use a digital one-wire protocol for communication.

1. DHT Sensor Pin Structure (Common for DHT11/DHT22):

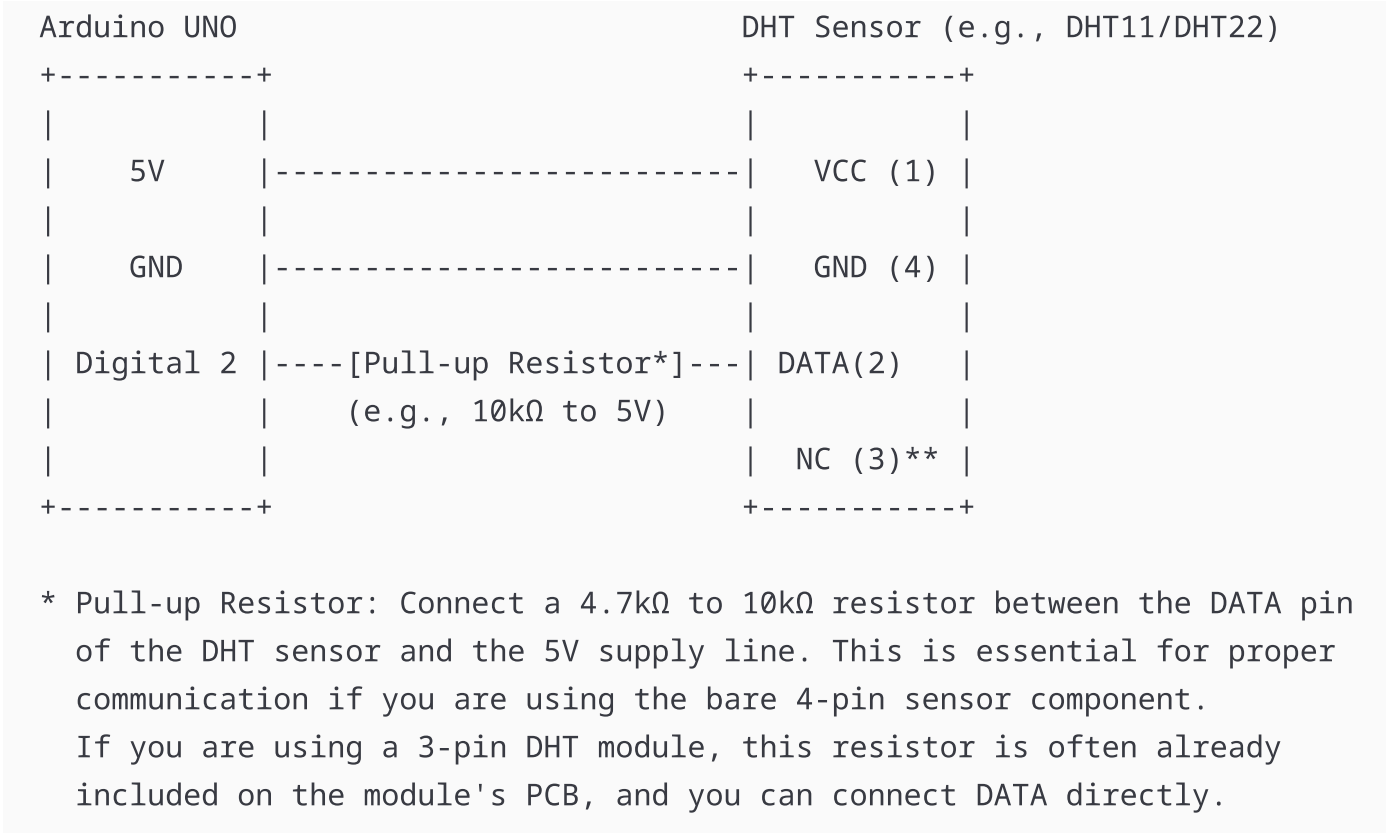
Most DHT sensors have 3 or 4 pins. If it's a 4-pin version, one pin is usually not connected (NC). For a 3-pin module version, the pull-up resistor is often already included.

- **Typical 4-pin DHT Sensor:**
  1. **VCC:** Power supply (typically 3.3V to 5V).
  2. **DATA:** Digital data output pin. This is the pin used for one-wire communication with the Arduino.
  3. **NC:** Not Connected (no internal connection).
  4. **GND:** Ground.
- **Typical 3-pin DHT Sensor Module (often on a small PCB):**
  1. **VCC / + :** Power supply (typically 3.3V to 5V).
  2. **DATA / OUT / S:** Digital data output pin.
  3. **GND / - :** Ground.

  Note: Pin order can vary between manufacturers and models. Always check the datasheet or markings on your specific DHT sensor/module.

2. Circuit Diagram of Connecting DHT Sensor with Arduino UNO:

For this example, we'll assume a common DHT11 or DHT22 sensor and connect its DATA pin to Arduino digital pin 2. A pull-up resistor (typically 4.7kΩ to 10kΩ) is required between the DATA pin and VCC if you are using the bare sensor component (not a module that already includes it).



```
  ** NC: Pin 3 is Not Connected on many 4-pin DHT sensors.
```

3. Program to Connect DHT Sensor with Arduino to Read Temperature and Humidity:

To use the DHT sensor, you'll typically need a library. The "DHT sensor library" by Adafruit is very common.

Steps to use the library:

1. Open Arduino IDE.
2. Go to Sketch > Include Library > Manage Libraries...
3. Search for "DHT sensor library" by Adafruit.
4. Install it. (This library might also require the "Adafruit Unified Sensor" library, which the IDE should prompt you to install if needed).

Arduino Sketch:

```
// Include the DHT library
#include "DHT.h"

// Define the type of DHT sensor you are using (DHT11, DHT22, DHT21)
#define DHTTYPE DHT11   // If using DHT11
// #define DHTTYPE DHT22  // If using DHT22 (AM2302)
// #define DHTTYPE DHT21  // If using DHT21 (AM2301)

// Define the digital pin connected to the DHT sensor's DATA pin
#define DHTPIN 2     // DHT sensor connected to digital pin 2

// Initialize DHT sensor object
DHT dht(DHTPIN, DHTTYPE);

void setup() {
  // Initialize serial communication for displaying readings
  Serial.begin(9600);
  Serial.println("DHT Sensor Test!");

  // Initialize the DHT sensor
  dht.begin();
}

void loop() {
  // Wait a few seconds between measurements.
  // DHT sensors are slow and readings should not be taken more than once
```

```
every 2 seconds.
  delay(2000);

  // Read humidity
  // The readHumidity() function returns a float.
  float humidity = dht.readHumidity();

  // Read temperature as Celsius (the default)
  // The readTemperature() function returns a float.
  float temperatureC = dht.readTemperature();

  // Read temperature as Fahrenheit (isFahrenheit = true)
  // float temperatureF = dht.readTemperature(true);

  // Check if any reads failed and exit early (to try again).
  if (isnan(humidity) || isnan(temperatureC)) {
    Serial.println("Failed to read from DHT sensor!");
    return; // Exit the loop function to try again
  }

  // --- Displaying the readings ---

  // Humidity
  Serial.print("Humidity: ");
  Serial.print(humidity);
  Serial.print("%  ");

  // Temperature in Celsius
  Serial.print("Temperature: ");
  Serial.print(temperatureC);
  Serial.println("°C ");

  /*
  // Temperature in Fahrenheit (optional)
  Serial.print("Temperature: ");
  Serial.print(temperatureF);
  Serial.println("°F");
  */
}
```

Explanation of the Program:

1. **`#include "DHT.h"`:** Includes the DHT sensor library, which provides functions to easily read data from the sensor.

2. **`#define DHTTYPE DHT11` (or `DHT22`/`DHT21`):** Specifies the type of DHT sensor being used. You need to uncomment the line corresponding to your sensor.
3. **`#define DHTPIN 2`:** Defines that the DATA pin of the DHT sensor is connected to digital pin 2 of the Arduino.
4. **`DHT dht(DHTPIN, DHTTYPE);`:** Creates a DHT object named `dht`, configured for the specified pin and sensor type.
5. **`setup()` function:**
    - `Serial.begin(9600);`: Initializes serial communication at a baud rate of 9600 bits per second. This allows the Arduino to send data to the computer, which can be viewed in the Arduino IDE's Serial Monitor.
    - `Serial.println("DHT Sensor Test!");`: Prints an initial message to the Serial Monitor.
    - `dht.begin();`: Initializes the DHT sensor.
6. **`loop()` function:**
    - `delay(2000);`: DHT sensors are relatively slow. It's recommended to wait at least 2 seconds between readings to ensure accurate data.
    - `float humidity = dht.readHumidity();`: Calls the library function to read the humidity. The value is stored as a float.
    - `float temperatureC = dht.readTemperature();`: Calls the library function to read the temperature in Celsius. The value is stored as a float. (You can use `dht.readTemperature(true);` to get Fahrenheit).
    - **Error Checking (`if (isnan(humidity) || isnan(temperatureC))`)**: Checks if the readings are "Not a Number" (NaN). This can happen if the sensor fails to provide a valid reading (e.g., due to wiring issues or sensor malfunction). If a failure occurs, an error message is printed, and the loop restarts.
    - **Serial Printing:** The humidity and temperature values are printed to the Serial Monitor with appropriate labels and units.

To see the output, after uploading the sketch to your Arduino, open the Serial Monitor in the Arduino IDE (Tools > Serial Monitor or Ctrl+Shift+M), ensuring the baud rate is set to 9600. You should see temperature and humidity readings updated every 2 seconds.
