Internet of Things (DI05000041)

Milav Dabgar

Lecturer, GP Palanpur

July 22, 2026

Table of Contents I

Table of Contents II

Table of Contents III

Introduction to IoT

- Course: Internet of Things (DI05000041)
- Unit 1: Fundamentals of Internet of Things (IoT)
- Lecture 1.1: Introduction to IoT (Part 1)
- Topics: Definition, Evolution, History, and Need in the Modern World

Agenda

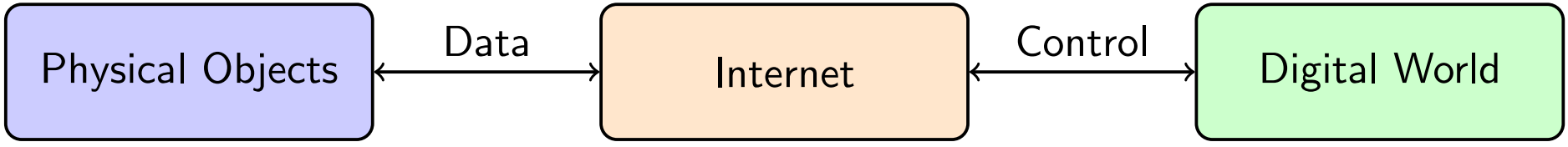
- Definition of IoT
- Core Components and Characteristics
- Evolution and History of IoT
- Key Milestones (RFID, WSN, IPv6)
- Need of IoT in the Modern World
- Summary & Next Lecture Preview

Objectives

- Understand the formal definition of the Internet of Things.
- Identify the key historical milestones that shaped IoT.
- Comprehend the technological evolution enabling IoT.
- Analyze the need for IoT in contemporary applications.

Definition of IoT

- IoT (Internet of Things) is a network of physical objects ('things') embedded with sensors, software, and other technologies.
- The purpose is connecting and exchanging data with other devices and systems over the internet.
- It merges physical and digital worlds, allowing real-world data collection and control.
- Examples range from ordinary household objects (smart bulbs) to sophisticated industrial tools.



What is a 'Thing'?

- A 'Thing' can be a person with a heart monitor implant.
- A farm animal with a biochip transponder.
- An automobile that has built-in sensors to alert the driver when tire pressure is low.
- Any natural or man-made object assigned an IP address and capable of transferring data over a network.

Early History

- 1832: The electromagnetic telegraph enabled direct human-to-machine communication.
- 1982: The first 'smart' device - a modified Coca-Cola vending machine at Carnegie Mellon University.
- It could report its inventory and whether newly loaded drinks were cold or not.
- Early concepts involved telemetry and basic M2M (Machine-to-Machine) communication.

Coining the Term: 1999

- 1999: The term 'Internet of Things' was coined by Kevin Ashton.
- He used it during a presentation at Procter & Gamble (P&G).
- The presentation linked the new idea of RFID (Radio-Frequency Identification) in P&G's supply chain to the internet.
- Ashton stated that adding RFID and other sensors to everyday objects would allow computers to manage and inventory them.

Key Enablers: RFID and WSN

- RFID (Radio Frequency Identification): Crucial early technology for tracking goods passively without power.
- WSN (Wireless Sensor Networks): Allowed multiple distributed sensors to monitor physical or environmental conditions.
- The convergence of wireless technologies, MEMS (micro-electromechanical systems), and the internet propelled IoT.
- Miniaturization allowed sensors to be cheap and small enough to put on anything.

The Role of IPv6

- IPv4 provided about 4.3 billion addresses, which was not enough for billions of IoT devices.
- IPv6 introduced 128-bit addresses, allowing 3.4×10^{38} unique addresses.
- Steven Leibson famously said, "we could assign an IPv6 address to every atom on the surface of the earth".
- IPv6 ensures that every single 'thing' can have a unique public IP address.

Need of IoT in the Modern World

- Data-Driven Decision Making: IoT provides real-time data for analytics (e.g., predictive maintenance).
- Automation and Efficiency: Reduces human intervention, lowering operational costs and errors.
- Quality of Life: Smart homes, smart health wearables, and smart cities improve daily living.
- Resource Management: Smart grids and precision agriculture optimize energy and water usage.

Key Application Areas

- Industrial IoT (IIoT): Manufacturing automation, supply chain tracking, and digital twins.
- Healthcare (IoMT): Remote patient monitoring, connected pacemakers.
- Smart Cities: Traffic management, smart street lighting, waste management.
- Agriculture: Soil moisture sensors, drone crop monitoring.

Evolution of IoT

Era	Key Developments
Pre-2000s	Telemetry, M2M, ARPANET Coca-Cola machine.
2000s	RFID adoption, Kevin Ashton's vision, basic WSNs.
2010s	Smart home boom, IPv6 launch, cloud computing integration.
2020s	AI integration (AIoT), 5G networks enabling massive IoT deployments, Edge computing.

Summary

- IoT connects physical objects equipped with sensors to the internet for data exchange.
- The term was coined by Kevin Ashton in 1999, initially focusing on RFID.
- Technologies like WSN, MEMS, and IPv6 (providing massive address space) enabled its growth.
- IoT is essential today for automation, data-driven decisions, and resource optimization across industries.

Next Lecture Preview

- Characteristics of IoT systems.
- Applications of IoT in various domains.
- Deep dive into Smart Homes and Smart Cities.

Agenda

- Learning Objectives
- Fundamental Characteristics of IoT
- Dynamic and Self-Adapting Systems
- Interconnectivity and Heterogeneity
- IoT Ecosystem Overview
- Key Layers of the IoT Ecosystem
- Real-World IoT Examples (Smart Home, Industry, Healthcare)
- Summary and Key Takeaways

Learning Objectives

- Identify and explain the core characteristics of an IoT system.
- Understand the components that make up the IoT ecosystem.
- Analyze the role of different layers within the IoT architecture.
- Recognize practical implementations of IoT in various industries.

Characteristics of IoT Systems

IoT systems are distinct from traditional computing networks due to several key traits. Core characteristics include:

- **Dynamic & Self-Adapting:** Ability to change state based on context.
- **Interconnectivity:** Everything is connected to the global information and communication infrastructure.
- **Heterogeneity:** Devices from different vendors using different protocols working together.
- **Enormous Scale:** Billions of devices communicating simultaneously.

Dynamic and Self-Adapting

- **Dynamic State Changes:**
  - Devices frequently change states (e.g., sleeping, waking, transmitting).
  - Operating environments change (temperature, location, network availability).
- **Self-Adapting Mechanisms:**
  - Devices can upgrade their software autonomously over-the-air (OTA).
  - Systems adapt to changing contexts (e.g., a smart camera switching to night vision based on ambient light).
  - Networks can self-heal or reroute data if a node fails.

Interconnectivity & Heterogeneity

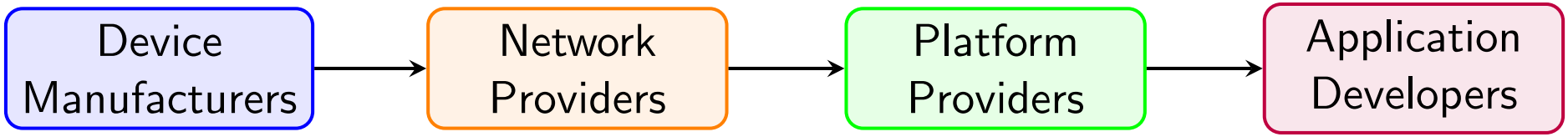
- **Interconnectivity:**
  - Seamless integration of physical and virtual objects.
  - Enables 'Anytime, Anywhere, Any Media, Anything' communication.
- **Heterogeneity:**
  - Devices use different hardware platforms (ARM, x86, RISC-V).
  - Use of diverse communication protocols (Wi-Fi, Zigbee, LoRaWAN, BLE).
  - IoT platforms act as middleware to bridge these heterogeneous elements into a unified system.

Enormous Scale & Safety

- **Enormous Scale:**
  - The number of connected devices vastly exceeds human populations.
  - Requires IPv6 to address the massive number of nodes.
  - Data generation is continuous and massive, requiring Big Data analytics.
- **Safety & Security:**
  - Physical security of exposed edge devices.
  - Data privacy and network security.
  - Safety-critical systems (e.g., medical implants, autonomous vehicles) where failure threatens lives.

IoT Ecosystem Overview

The IoT ecosystem is a complex network of interconnected components involving hardware, software, connectivity, and human interfaces.



- **Core Participants:** Device Manufacturers (Hardware), Network Providers (Connectivity), Platform Providers (Cloud/Middleware), Application Developers (Software/Services)

Ecosystem Layers: Devices & Connectivity

- **Device/Perception Layer:**
  - Physical objects embedded with sensors (data collection) and actuators (taking action).
  - Edge computing capabilities for localized processing.
- **Connectivity/Network Layer:**
  - Transmits data from the perception layer to the cloud.
  - Includes gateways that translate protocols (e.g., Zigbee to IP).
  - Technologies: Cellular, Wi-Fi, Ethernet, LPWAN, Bluetooth.

Ecosystem Layers: Data Processing & Application

- **Data Processing/Cloud Layer:**
  - Ingestion of massive data streams.
  - Storage (Databases, Data Lakes).
  - Analytics and Machine Learning to derive insights.
- **Application/User Interface Layer:**
  - Delivers value to the end-user.
  - Smart dashboards, mobile apps, alert systems.
  - Industry-specific applications (Smart Agriculture dashboard vs. Smart Home app).

Examples of IoT: Smart Home

Smart Home Automation:

- **Lighting:** Smart bulbs adjusting color and intensity based on time of day.
- **Climate Control:** Smart thermostats (e.g., Nest) learning user preferences to optimize HVAC energy usage.
- **Security:** Connected doorbells, smart locks, and window sensors integrated with mobile alerts.
- **Appliances:** Refrigerators that monitor inventory and suggest recipes.

Examples of IoT: Industrial & Healthcare

Industrial IoT (IIoT)	Healthcare (IoMT)
Predictive Maintenance: Sensors on factory motors detect abnormal vibrations, predicting failure before it happens.	**Remote Patient Monitoring:** Wearables tracking ECG, heart rate, and blood oxygen, sending data directly to physicians.
Supply Chain: RFID and GPS tracking of goods in real-time.	**Smart Hospitals:** Asset tracking for critical equipment like wheelchairs and infusion pumps.

Key Takeaways

- IoT systems are inherently dynamic, self-adapting, heterogeneous, and operate at a massive scale.
- The IoT ecosystem is a multi-layered architecture comprising Devices, Connectivity, Data Processing, and Applications.
- Gateways play a crucial role in bridging heterogeneous edge devices to the internet.
- IoT applications span diverse domains, from consumer Smart Homes to critical Industrial and Healthcare systems.

Next Lecture Preview

- **Topic:** 1.2 Physical Design of IoT
- **Focus Areas:**
  - Things in IoT (Sensors, Actuators, Microcontrollers)
  - IoT Protocols Overview (Link layer, Network layer)
  - Exploring the physical hardware that makes up the Perception Layer

Agenda

- Introduction to IoT Architecture
- The Four-Layer Architecture Model
- Sensing Layer (Perception)
- Network Layer (Transport)
- Data Processing Layer (Middleware)
- Application Layer
- Protocols and Technologies at each layer

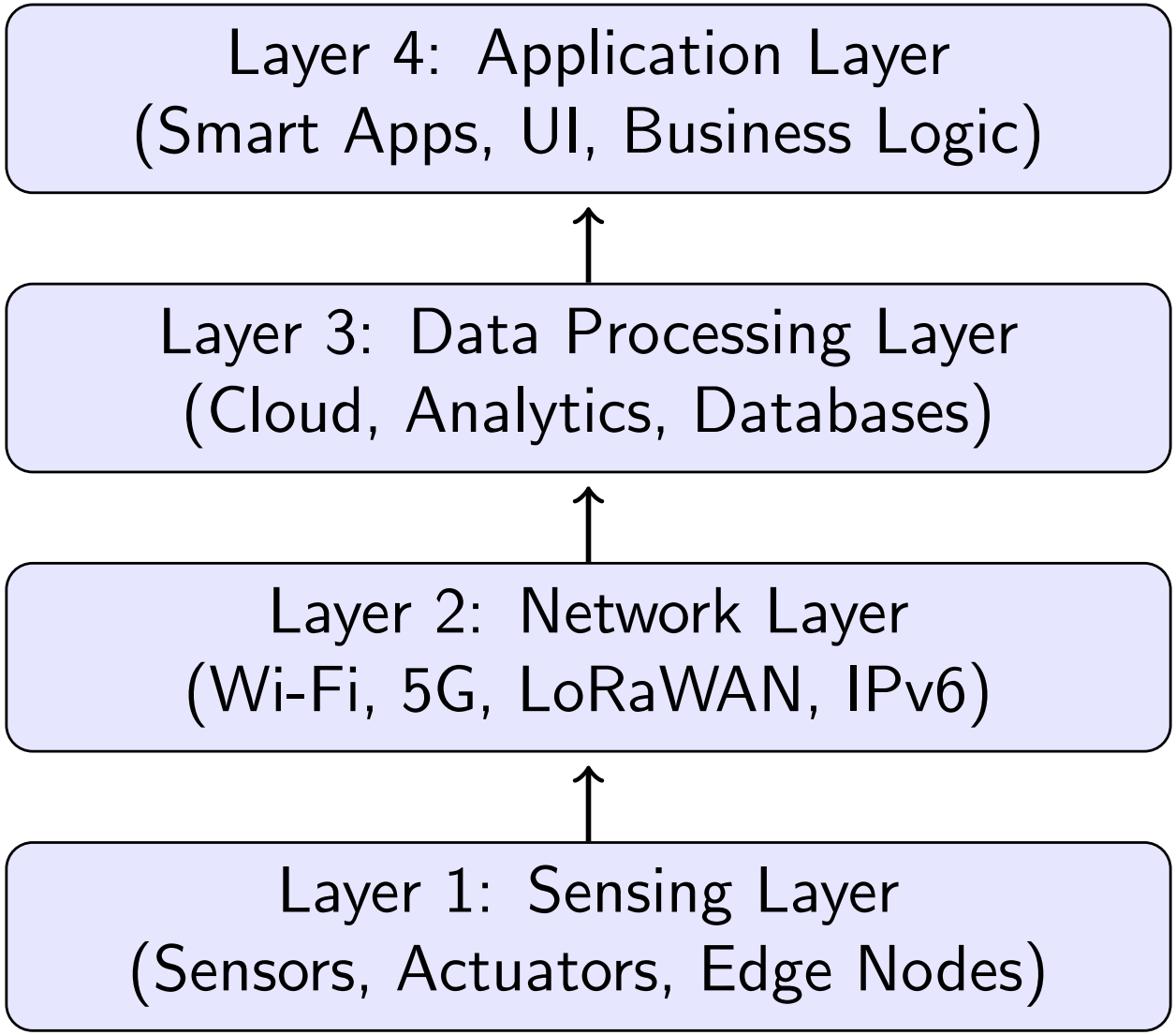
Learning Objectives

- Understand the necessity of a layered architecture in IoT.
- Identify the four major layers of the IoT architecture.
- Explain the function of the Sensing Layer in gathering physical data.
- Describe how the Network Layer transports data securely.
- Analyze the role of the Data Processing layer in analytics and storage.
- Recognize the Application layer's purpose in delivering end-user value.

Introduction to IoT Architecture

- IoT systems are highly complex, involving hardware, networks, cloud, and user interfaces.
- A layered architecture simplifies design, troubleshooting, and scalability.
- It decouples hardware from software, allowing interoperability between heterogeneous devices.
- While 3-layer, 5-layer, and 7-layer models exist, the 4-layer model provides an optimal balance for learning and industrial application.

The 4-Layer IoT Architecture Model



Sensing Layer (Perception Layer)

- The lowest layer of the IoT architecture, interacting directly with the physical world.
- Primary function: Data acquisition and physical actuation.
- Components: Sensors (temperature, humidity, motion), Actuators (motors, relays, valves), and embedded microcontrollers (like ESP32).
- Converts physical parameters (analog) into digital signals.

Sensing Layer Technologies

- Sensors: DHT11 (Temperature/Humidity), LDR (Light), HC-SR04 (Ultrasonic).
- Microcontrollers: ESP32, Raspberry Pi, Arduino.
- Data Acquisition (DAQ) units filter and amplify signals.
- Key Challenges: Power consumption, environmental durability, and calibration.

Network Layer (Transport Layer)

- Responsible for transmitting data from the Sensing Layer to the Data Processing Layer.
- Acts as a bridge connecting edge devices to the internet or local gateway.
- Handles data routing, transmission, and network security.
- Accommodates various communication mediums (wired and wireless).

Network Layer Protocols & Standards

Category	Examples
Short-range	Bluetooth Low Energy (BLE), Zigbee, RFID, NFC
Medium-range	Wi-Fi (802.11)
Long-range (LPWAN)	LoRaWAN, Sigfox, NB-IoT
Cellular	4G LTE, 5G
Networking Protocols	IPv4, IPv6, 6LoWPAN

Data Processing Layer (Middleware/Cloud)

- Receives vast amounts of data from the Network Layer.
- Core functions: Data storage, aggregation, filtering, and analysis.
- Utilizes Cloud Computing (AWS, Azure) and Edge Computing.
- Implements Machine Learning and AI algorithms to extract meaningful insights.

Data Processing Functions & Protocols

- Messaging Protocols: MQTT, CoAP, AMQP, HTTP/REST.
- Databases: Time-series databases (InfluxDB), NoSQL (MongoDB).
- Event Processing: Rule engines that trigger alerts based on thresholds.
- Edge vs. Cloud: Processing can happen at the edge (closer to sensors) to reduce latency and bandwidth.

Application Layer

- The topmost layer, visible to the end-user.
- Delivers application-specific services and user interfaces (Dashboards, Mobile Apps).
- Translates processed data into business value.
- Examples: Smart Home Automation apps, Industrial SCADA dashboards, Fleet tracking systems.

Key Takeaways

- Sensing Layer gathers physical data using sensors and edge devices.
- Network Layer transmits data securely using protocols like Wi-Fi, LoRa, and BLE.
- Data Processing Layer stores and analyzes data, often using Cloud and MQTT.
- Application Layer provides the UI and specific business services to the user.
- Layered architecture ensures decoupling and interoperability.

Next Lecture Preview

- Topic: 1.3 Characteristics and Requirements of IoT.
- Understanding what makes a device 'smart'.
- Exploring scalability, dynamic adaptation, and self-configuration.
- Security and privacy requirements in modern IoT ecosystems.

1.3 Physical and Logical Design of IoT

- Understanding IoT Components and Architecture

Agenda

- Introduction to IoT System Design
- Physical Components: Nodes, Sensors, Actuators
- Logical Design: IoT Functional Blocks
- IoT Communication Models
- Request-Response vs Publish-Subscribe
- IoT Communication APIs: REST & WebSockets

Learning Objectives

- Differentiate between physical and logical design in IoT.
- Identify the roles of sensors, actuators, and edge devices.
- Describe the six primary functional blocks of an IoT system.
- Compare Request-Response and Publish-Subscribe communication models.
- Evaluate the use cases for RESTful APIs and WebSocket APIs in IoT.

Introduction to IoT Design

- **Physical Design:** Refers to the physical devices and components that make up the IoT system (Things).
- **Logical Design:** Refers to an abstract representation of the entities and processes without tying to specific hardware.
- Both designs work in tandem to create a cohesive ecosystem from edge sensing to cloud processing.

Physical Design: Things in IoT

- **Things (Nodes):** Embedded devices capable of sensing, actuating, and communicating.
- **Key characteristics:**
  - Unique identities (e.g., IP address, MAC address).
  - Ability to perform remote sensing and monitoring.
  - Form factor varies from micro-chips to heavy industrial machinery.
- **Processing Constraints:** Often battery-powered with limited compute and memory (e.g., ESP32, Arduino).

Physical Design: Sensors

- **Definition:** Devices that detect changes in the physical environment and convert them into measurable electrical signals.
- **Types of Sensors:**
  - **Analog:** Produce continuous voltage signals (e.g., LDR, LM35).
  - **Digital:** Output discrete digital signals (e.g., DHT11 for Temp/Humidity).
- **Role:** They act as the 'eyes and ears' of the IoT system, gathering raw data for processing.

Physical Design: Actuators

- **Definition:** Devices that convert electrical energy into physical motion or action.
- **Common Actuators in IoT:**
  - Relays (to switch high power loads).
  - Servo and Stepper Motors (precise movement).
  - Solenoid valves (fluid control).
- **Role:** They allow the IoT system to physically affect its environment based on sensor data or remote commands.

Logical Design: IoT Functional Blocks

- An IoT system consists of several functional blocks:
  - **Device:** Senses, actuates, and processes locally.
  - **Communication:** Handles connectivity (Wi-Fi, LoRa, BLE).
  - **Services:** Device modeling, data publishing, discovery.
  - **Management:** Device provisioning, monitoring, and firmware updates.
  - **Security:** Authentication, authorization, and data encryption.
  - **Application:** The user interface or business logic layer.

IoT Communication Models

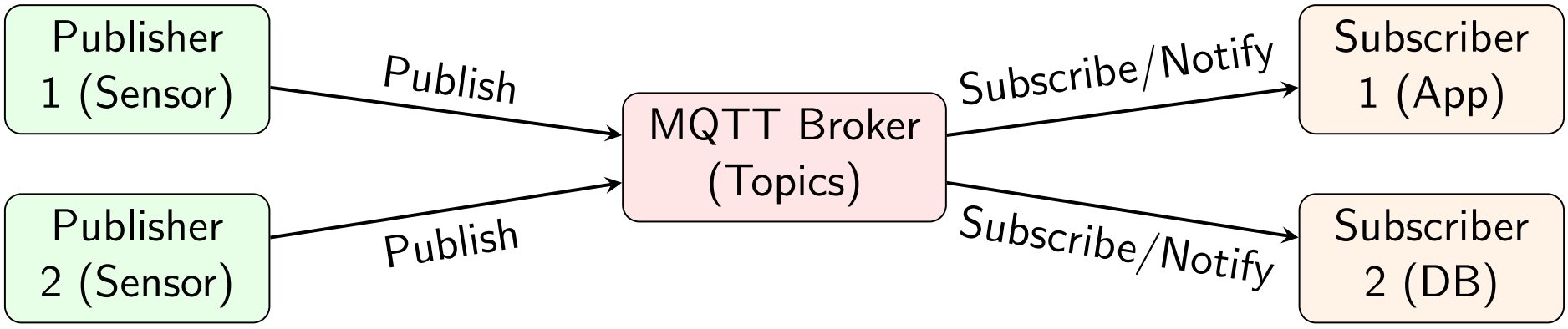
- IoT devices communicate with servers and each other using specific models.
- **Main Models:**
  - Request-Response
  - Publish-Subscribe
  - Push-Pull
  - Exclusive Pair
- **Selection Criteria:** Depends on network latency, power constraints, data volume, and real-time requirements.

Model: Request-Response

- **Architecture:** Client-Server model.
- **Mechanism:**
  - Client requests data from a Server.
  - Server processes the request and sends a Response.
- **Characteristics:**
  - Stateless and typically synchronous.
  - High overhead due to headers and connection setup.
  - Example Protocol: HTTP/HTTPS.

Model: Publish-Subscribe

- **Architecture:** Publisher, Broker, Subscriber.
- **Mechanism:**
  - Publishers send data to specific 'Topics' on a Broker.
  - Subscribers listen to specific 'Topics'.
  - Broker routes data from Publishers to Subscribers.
- **Characteristics:**
  - Asynchronous and decoupled.
  - Low bandwidth overhead, ideal for many-to-many communication.
  - Example Protocol: MQTT.



IoT Communication APIs: REST vs WebSocket

- **REST-based APIs:**
  - Follows Request-Response model over HTTP.
  - Uses standard HTTP methods (GET, POST, PUT, DELETE).
  - Best for stateless, occasional data transfers.
- **WebSocket-based APIs:**
  - Follows Exclusive Pair model over a single TCP connection.
  - Full-duplex, bidirectional communication.
  - Best for real-time, continuous data streams (e.g., live dashboards).

Feature	**REST API**	**WebSocket API**
Model	Request-Response	Exclusive Pair
Connection	Stateless (HTTP)	Stateful (TCP)
Communication	Unidirectional	Bidirectional (Full-duplex)
Best For	Occasional Data	Real-time Streams

Key Takeaways

- Physical design encompasses nodes, sensors, and actuators acting on the environment.
- Logical design provides an abstract architecture via six primary functional blocks.
- Request-Response (HTTP) is synchronous; Publish-Subscribe (MQTT) is asynchronous and broker-based.
- REST APIs are suitable for stateless operations, while WebSockets excel in low-latency, real-time bidirectional data flow.

Next Lecture Preview

- **Topic:** 1.4 IoT Enabling Technologies
- **Key Areas to Cover:**
  - Wireless Sensor Networks (WSN).
  - Cloud Computing and Big Data Analytics.
  - Embedded Systems in IoT.
  - Communication Protocols (Overview).

Introduction

- **author**: IoT Instructor

Overview of IoT Enabling Technologies

- Overview of IoT Enabling Technologies
- Wireless Sensor Networks (WSN) Concepts
- WSN Node Architecture
- WSN Topologies (Star, Mesh)
- Cloud Computing Basics for IoT
- Cloud Service Models (IaaS, PaaS, SaaS)

Learning Objectives

- Understand the role of enabling technologies in IoT systems.
- Describe the components and architecture of a WSN node.
- Differentiate between Star and Mesh topologies in WSN.
- Explain the basic concepts of Cloud Computing.
- Identify the differences between IaaS, PaaS, and SaaS in an IoT context.

IoT Enabling Technologies

- IoT ecosystems rely on several foundational technologies.
- Key technologies: WSN, Cloud Computing, Big Data Analytics, Embedded Systems, and Communication Protocols.
- Function: They bridge the gap between the physical world (sensors/actuators) and the digital world (data processing/insights).
- Today's focus: WSN for data gathering and Cloud Computing for data storage and processing.

Wireless Sensor Networks (WSN)

- Definition: A network of spatially distributed autonomous sensors.
- Purpose: Monitor physical or environmental conditions (temperature, sound, pressure).
- Operation: Nodes cooperatively pass their data through the network to a main location (sink or gateway).
- Key Characteristics: Low power consumption, scalability, and ability to withstand harsh environmental conditions.

WSN Node Architecture

- A sensor node is the fundamental building block of a WSN.
- Core Components:
  - Sensing Unit: Consists of sensors and Analog-to-Digital Converters (ADCs).
  - Processing Unit: Microcontroller or microprocessor managing node tasks.
  - Transceiver Unit: Handles radio communication (transmit/receive).

WSN Node Power & Optional Components

- Power Unit: Usually a battery, sometimes augmented with energy harvesting (solar, thermal).
- Optional Components:
  - Location Finding System (e.g., GPS for tracking nodes).
  - Mobilizer: Required if the node needs to move physically.
- Power management is the most critical aspect, driving the design of the processing and transceiver units.

WSN Topologies

- Star Topology: Every node connects directly to a central hub (gateway).
- Mesh Topology: Nodes connect to multiple other nodes. Data hops from node to node to reach the gateway.

Topology	Pros	Cons
Star	Simple, low latency	Single point of failure, limited range
Mesh	High reliability, extended range, self-healing	Higher power consumption, complex routing

Cloud Computing Basics for IoT

- Definition: Delivery of computing services (servers, storage, databases, networking, software) over the internet.
- Role in IoT: WSNs generate massive amounts of data; Cloud provides the necessary scalable storage and computational power.
- Benefits for IoT: Ubiquitous access, rapid elasticity, pay-as-you-go pricing, and high availability.
- Challenges: Latency (time taken to send data to the cloud and back) and bandwidth constraints.

Cloud Service Models: IaaS

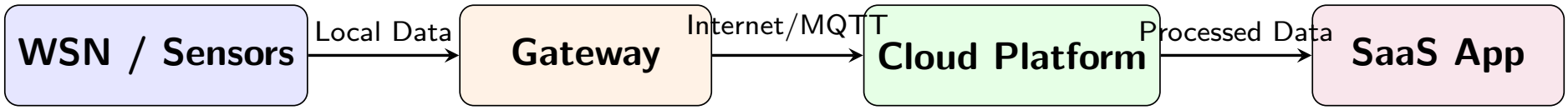
- Infrastructure as a Service (IaaS): Provides virtualized computing resources over the internet.
- IoT Use Case: Renting raw servers and storage to host custom IoT databases and backend services.
- Examples: Amazon EC2, Google Compute Engine.
- User Responsibility: OS, middleware, runtime, data, and applications.

Cloud Service Models: PaaS & SaaS

- Platform as a Service (PaaS): Provides hardware and software tools—usually for application development.
  - IoT Use Case: Managed IoT platforms (e.g., AWS IoT Core) where developers focus on IoT logic.
- Software as a Service (SaaS): Delivers complete software applications over the internet.
  - IoT Use Case: End-user IoT dashboards and smart home apps (e.g., Google Nest app).

End-to-End IoT Architecture

- Sensors (WSN) collect real-time physical data.
- Gateway acts as a bridge, translating WSN protocols to internet protocols (e.g., HTTP/MQTT).
- Cloud Platform receives, stores, and performs big data analytics on the ingested data.
- SaaS Applications visualize the processed data for users and trigger automated actions back to the devices.



Summary

- WSN and Cloud Computing are foundational enabling technologies for IoT.
- A WSN node consists of sensing, processing, transceiver, and power units.
- Mesh topologies offer reliability for WSNs, while star topologies offer simplicity.
- Cloud computing provides essential scalability for IoT data storage and processing.
- IaaS, PaaS, and SaaS serve different levels of IoT architecture needs, from raw infrastructure to end-user applications.

Next Lecture Preview

- Big Data Analytics in IoT.
- Communication Protocols (overview).
- Embedded Systems role in IoT.
- How these remaining technologies complete the IoT ecosystem.

Agenda

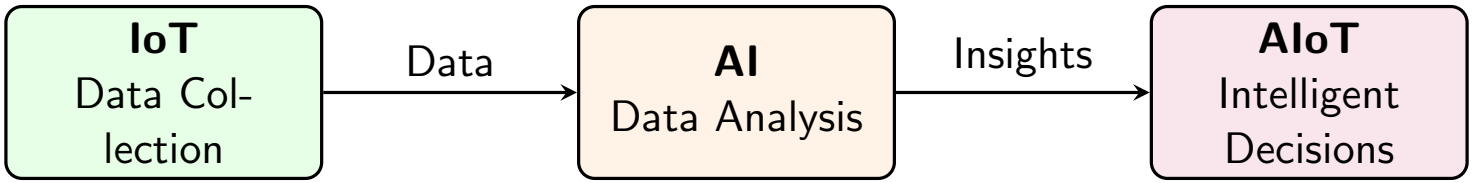
- Introduction to AI in IoT (AIoT)
- Why Combine AI and IoT?
- Machine Learning at the Edge (Edge AI)
- Predictive Maintenance in IoT
- Computer Vision in IoT
- Challenges and Future Trends

Objectives

- Understand the basic concept of Artificial Intelligence in IoT (AIoT).
- Explain the architecture and benefits of Machine Learning at the edge.
- Describe how predictive maintenance leverages IoT data.
- Identify applications of computer vision within IoT systems.

AIoT = AI + IoT

- AIoT = Artificial Intelligence + Internet of Things.
- IoT acts as the digital nervous system, collecting vast amounts of data via sensors.
- AI acts as the brain, analyzing data, recognizing patterns, and making autonomous decisions.
- Moves IoT from 'connected devices' to 'intelligent devices'.



Why Combine AI and IoT?

- **Data Utilization:** Processing the zetabytes of data generated by IoT devices efficiently.
- **Automation:** Enabling autonomous actions without human intervention (e.g., smart HVAC adjusting to occupancy).
- **Real-time Decision Making:** Reducing latency by processing data close to the source.
- **Personalization:** Learning user behaviors to tailor experiences (e.g., smart home routines).

Machine Learning at the Edge (Edge AI)

- **Definition:** Deploying ML algorithms directly on edge devices (gateways, ESP32, Raspberry Pi) rather than the cloud.
- **Mechanism:** A model is trained in the cloud, optimized (quantization, pruning), and then deployed to the edge device.
- **TinyML:** A subfield focusing on running ML on low-power microcontrollers (like ESP32).

Advantages of Edge AI

- **Low Latency:** Immediate inference (e.g., autonomous braking in vehicles).
- **Bandwidth Conservation:** Only sending insights (e.g., 'Person Detected') instead of raw video streams.
- **Privacy & Security:** Sensitive data (voice, images) never leaves the local device.
- **Offline Operation:** System continues to function without internet connectivity.

Feature	**Edge AI**	**Cloud AI**
Latency	Low (Real-time)	High (Network dependent)
Bandwidth	Low (Send insights)	High (Send raw data)
Privacy	High (Local processing)	Low (Data sent to cloud)
Connectivity	Works Offline	Requires Internet

Predictive Maintenance in IoT

- **Definition:** Using data analysis tools to detect anomalies and predict equipment failure before it happens.
- **Contrast with Reactive:** Fixing after it breaks (high downtime).
- **Contrast with Preventive:** Routine scheduled maintenance (often unnecessary, wastes resources).
- **Predictive:** 'Just-in-time' maintenance based on actual equipment health.

How Predictive Maintenance Works

- **Data Collection:** IoT sensors monitor vibration, temperature, acoustic emissions, and current draw.
- **Data Processing:** Edge devices filter and pre-process the signals.
- **AI Modeling:** ML anomaly detection models (e.g., Autoencoders, Random Forests) identify deviations from normal baselines.
- **Alerting:** System alerts technicians with a probability of failure and estimated time to failure.

Computer Vision in IoT

- **Definition:** Enabling IoT systems to derive meaningful information from digital images, videos, and visual inputs.
- **Hardware:** Uses cameras as the primary 'sensor' (e.g., ESP32-CAM).
- **Techniques:** Object detection, facial recognition, image classification, and semantic segmentation.

Applications of Computer Vision in IoT

- **Smart Retail:** Automated checkout, foot traffic analysis, shelf inventory monitoring.
- **Industrial Quality Control:** Detecting microscopic defects in manufacturing lines using high-speed cameras.
- **Smart Cities:** Traffic flow optimization, parking space availability, license plate recognition.
- **Security:** Intrusion detection with false-alarm filtering (ignoring pets, detecting humans).

Challenges and Future Trends

- **Resource Constraints:** Running complex AI models on devices with limited RAM and CPU (like ESP32).
- **Data Quality:** AI models are only as good as the sensor data ('Garbage In, Garbage Out').
- **Security Risks:** AI models can be vulnerable to adversarial attacks or data poisoning.
- **Management:** Updating and deploying new ML models to thousands of remote edge devices.

Summary

- AIoT integrates AI's intelligence with IoT's data gathering capabilities.
- Edge AI (and TinyML) allows ML models to run locally, ensuring low latency, privacy, and offline capability.
- Predictive maintenance uses IoT sensor data and AI to foresee and prevent equipment failures.
- Computer vision turns cameras into powerful IoT sensors for retail, industry, and security.

Next Lecture Preview

- In our next lecture, we will move to Unit 1, Topic 1.5: IoT Communication Overview.
- We will discuss M2M, D2D, Device-to-Cloud communication, and the critical role of the Internet in IoT.

Agenda

- Machine-to-Machine (M2M) Communication
- Understanding Device-to-Device (D2D) Networks
- Device-to-Cloud Communication Paradigms
- The Role of the Internet in IoT
- Comparative Analysis: M2M vs. IoT

Learning Objectives

- Understand the foundational concepts of M2M, D2D, and Device-to-Cloud architectures.
- Analyze the role of IP networking and the broader Internet in scaling IoT solutions.
- Identify the architectural and protocol differences between closed M2M telemetry and open IoT ecosystems.
- Evaluate use cases for different communication models based on latency, power, and bandwidth.

Machine-to-Machine (M2M) Communication

- Direct communication between devices using wired or wireless channels.
- No human intervention required during standard operation.
- Historically relies on point-to-point communications.
- Primary focus: Telemetry, remote monitoring, and industrial control.
- Typically uses proprietary protocols or specialized cellular connections (e.g., GSM/CDMA for SCADA).

Key Takeaways

M2M is built for isolated telemetry.
Relies on point-to-point connections without human input.

The Closed Nature of M2M

- Operates in closed, siloed networks ("Intranet of Things").
- Hardware-specific: Sensors and receivers are tightly coupled.
- Scalability issues: Adding new types of devices often requires new infrastructure.
- Security is primarily achieved through physical isolation and obscure proprietary protocols.
- Data is rarely shared across different enterprise applications.

Key Takeaways

M2M systems are typically closed and siloed.
Integration with other enterprise systems is difficult.

Internet of Things (IoT): Open and IP-Based

- Leverages standard Internet Protocols (IP) like IPv4 and specifically IPv6.
- Open systems: Hardware is decoupled from the application layer.
- Highly scalable: IP networks can support billions of distinct, addressable nodes.
- Enables data sharing across disparate platforms and cloud-based analytics.
- Uses standardized application layer protocols (e.g., MQTT, CoAP, HTTP).

Key Takeaways

IoT utilizes standard IP protocols.
Data is decoupled from hardware, promoting interoperability.

Key Differences: M2M vs. IoT

- Connectivity: M2M uses Point-to-Point (often proprietary); IoT uses standard IP networks.
- Architecture: M2M is Closed/Siloed; IoT is Open/Interoperable.

Feature	**M2M**	**IoT**
Connectivity	Point-to-Point, Proprietary	IP-based, Open Standards
Architecture	Closed, Siloed	Open, Interoperable
Data Use	Single Application	Multiple Enterprise Applications
Internet	Not Strictly Required	Relies on Internet & Cloud
Hardware	Tightly Coupled	Decoupled (Edge vs Cloud)

Device-to-Device (D2D) Communication

- Direct communication between two adjacent nodes without an intermediary server.
- Typically relies on short-range wireless protocols (Bluetooth, Zigbee, Z-Wave).
- Reduces latency since data does not travel to the cloud and back.
- Preserves bandwidth on the broader network.
- Ideal for constrained environments and local automation.

Key Takeaways

D2D is direct, peer-to-peer communication.
Uses short-range protocols and reduces latency.

Device-to-Device (D2D): Use Cases & Limitations

- Use Cases: Smart Home automation (e.g., thermostat talking to a smart vent), wearables syncing with smartphones.
- Limitations: Confined to proximity (usually < 100 meters).
- Limitations: Interoperability challenges if devices use different D2D protocols (e.g., Zigbee vs. BLE).
- Security concerns: Eavesdropping and spoofing on local RF bands.
- Often requires a Gateway to eventually bridge D2D networks to the Internet.

Key Takeaways

Best for local, low-latency automation.
Restricted by short physical ranges.

Device-to-Cloud Communication

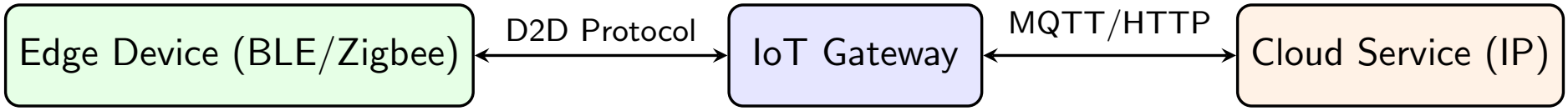
- Devices connect directly to an Internet cloud service (e.g., AWS IoT, Azure IoT Hub).
- Utilizes standard IP connections via Wi-Fi, Ethernet, or Cellular (4G/5G/NB-IoT).
- Ideal for heavy computation, historical data storage, and global accessibility.
- Enables remote monitoring and control from anywhere in the world.
- Protocols like MQTT and AMQP provide robust message brokering to cloud endpoints.

Key Takeaways

Requires IP-capable edge devices.
Enables global access and heavy cloud-side processing.

Device-to-Cloud: The Role of Gateways

- Not all devices can handle TCP/IP (due to power or processing constraints).
- Device-to-Gateway-to-Cloud model bridges the gap.
- Gateway translates local D2D protocols (Zigbee/BLE) into IP protocols (MQTT/HTTP).
- Gateways can perform edge computing: filtering and aggregating data before sending it to the cloud.



The Role of the Internet in IoT

- Provides the ubiquitous transport layer for global device connectivity.
- Enables IPv6 addressing, ensuring a unique IP for billions of devices.
- Facilitates integration with cloud computing platforms for Big Data analytics.
- Supports standardized security layers like TLS/DTLS over the public network.
- Without the Internet, IoT reverts to localized M2M networks.

Key Takeaways

The Internet provides routing, addressing (IPv6), and global reach.
It is the bridge between edge devices and centralized cloud intelligence.

Summary & Key Takeaways

- M2M focuses on closed, telemetry-based, point-to-point communication.
- IoT is characterized by open, IP-based networks that integrate with cloud services.
- D2D provides low-latency local communication using protocols like BLE and Zigbee.
- Device-to-Cloud allows for global access and centralized analytics, often leveraging MQTT over IP.
- Gateways are essential for bridging constrained D2D networks to the broader Internet.

Key Takeaways

Understand the IP vs. non-IP distinction.
Match the communication model to the specific IoT use case.

Next Lecture Preview

- Topic: IoT Architecture Layers
- Exploring the 3-layer, 4-layer, and 5-layer IoT architectures.
- Deep dive into the Perception, Network, and Application layers.
- Understanding how data flows from physical sensors up to user interfaces.

Agenda

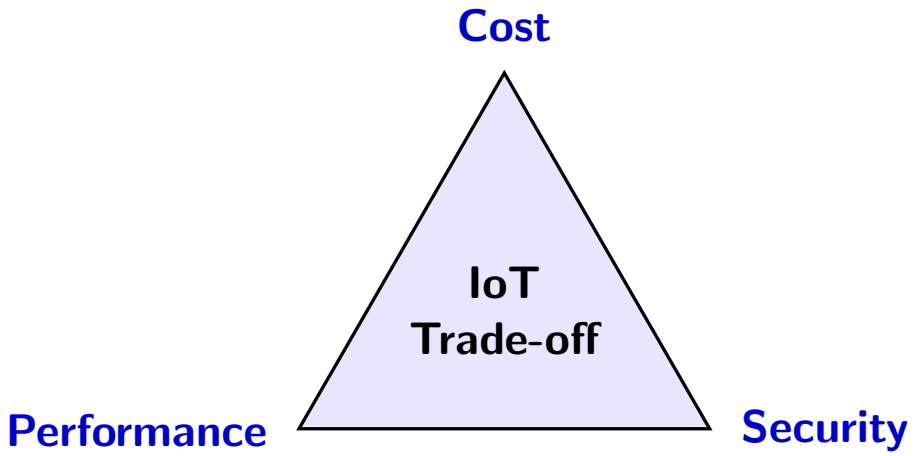
- Introduction to IoT Challenges
- Design Challenges: Power & Resources
- Design Challenges: Interoperability
- Security: Authentication & DDoS
- Security: Data Privacy & Physical Security
- Deployment Challenges

Learning Objectives

- Identify key design constraints in IoT end-devices.
- Analyze the critical security threats targeting IoT networks.
- Understand the impact of DDoS attacks and authentication failures.
- Evaluate the deployment challenges related to scalability and physical security.

Introduction to IoT Challenges

- IoT environments are highly heterogeneous, involving varied hardware, protocols, and platforms.
- Unlike traditional IT systems, IoT integrates the cyber world with physical processes.
- Challenges are generally grouped into Design, Security, and Deployment.
- The trade-off triangle: Cost vs. Performance vs. Security.



Design Challenges: Power

- Most edge nodes are battery-operated and deployed in hard-to-reach locations.
- Continuous wireless transmission (Wi-Fi, Cellular) drains battery rapidly.
- Requires Low-Power Wide-Area Networks (LPWAN) like LoRa or NB-IoT.
- Energy harvesting (solar, thermal, vibration) is often necessary but unpredictable.

Design Challenges: Resource Constraints

- Microcontrollers (MCUs) have limited processing power (e.g., tens of MHz).
- Severely constrained memory (few KBs of RAM and Flash).
- Inability to run standard operating systems; requires RTOS (Real-Time Operating System) or bare-metal code.
- Limits the ability to perform complex on-device analytics or run heavy cryptographic algorithms.

Component	**Traditional System**	**IoT Edge Device**
CPU	Multi-core GHz	Single-core MHz
Memory	Gigabytes (GB)	Kilobytes (KB)
Power	Mains powered	Battery/Harvested
OS	Full Linux/Windows	RTOS / Bare-metal

Design Challenges: Interoperability

- Fragmented ecosystem with no single universally accepted standard.
- Multiple communication protocols operating at different frequencies (Zigbee, Z-Wave, BLE, Thread).
- Vendor lock-in: Devices from different manufacturers often cannot communicate natively.
- Requires complex middleware and gateways to translate payloads.

Security: Authentication

- Millions of devices require unique, verifiable identities.
- Default or hardcoded passwords (e.g., 'admin/admin') are a massive vulnerability.
- Lack of mutual authentication allows rogue devices to join networks.
- Public Key Infrastructure (PKI) is difficult to implement on constrained nodes.

Security: DDoS

- IoT devices are prime targets for forming botnets (e.g., Mirai botnet).
- A compromised IoT fleet can launch massive Distributed Denial of Service (DDoS) attacks.
- Insecure network interfaces (open Telnet, SSH, or HTTP ports).
- Lack of Network Address Translation (NAT) or firewalls on edge devices.

Security: Data Privacy

- IoT devices collect sensitive personal or industrial data (health metrics, location, production rates).
- Data in transit often lacks end-to-end encryption due to computational overhead.
- Data at rest on the device is rarely encrypted, making physical theft a data breach risk.
- Compliance with privacy regulations (GDPR, CCPA) is difficult when data ownership is unclear.

Security: Physical Threats

- Devices are often deployed in unattended, public, or hostile environments.
- Susceptible to physical tampering, node capture, and extraction of firmware.
- Attackers can extract cryptographic keys directly from the flash memory via JTAG/UART.
- Hardware trojans and side-channel attacks (power analysis) are real threats.

Deployment Challenges

- Managing firmware Over-The-Air (OTA) updates for millions of heterogeneous devices.
- Network congestion: Millions of nodes attempting to transmit simultaneously.
- Addressing constraints: Transitioning from IPv4 to IPv6 (6LoWPAN) to handle the volume of devices.
- Cloud infrastructure costs scale non-linearly with massive data ingestion.

Summary

- Power and resource constraints severely limit the complexity of IoT edge devices.
- Interoperability remains a barrier, requiring gateways and middleware.
- Poor authentication and open ports lead to catastrophic DDoS botnets (Mirai).
- Physical security and scalable deployment mechanisms (OTA) are non-negotiable for real-world IoT.

Next Steps

- Introduction to standard IoT Reference Architectures.
- The 3-layer, 4-layer, and 5-layer IoT models.
- Edge, Fog, and Cloud computing paradigms in IoT.

Agenda

- Need for Networking in IoT
- Client-Server Architecture
- Publish-Subscribe Architecture
- IP, MAC, and Port Numbers
- URLs and Domain Names

Learning Objectives

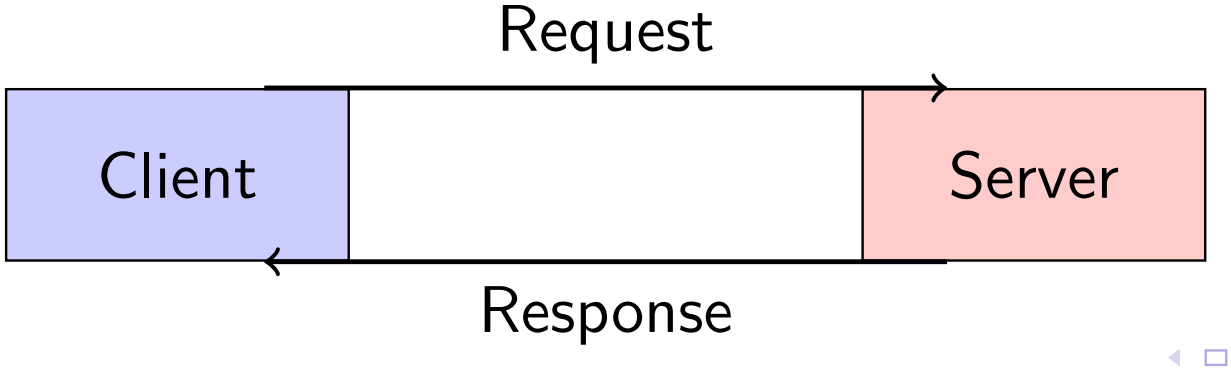
- Understand the necessity of robust networking in IoT deployments.
- Differentiate between Client-Server and Publish-Subscribe communication models.
- Explain the roles of IPv4, IPv6 (6LoWPAN), MAC addresses, and Port numbers.
- Describe how URLs and Domain Names facilitate IoT device accessibility.

Need for Networking in IoT

- **Data Transmission:** Sensors collect data that must be transmitted to edge gateways or cloud servers for processing.
- **Remote Control & Actuation:** Users or automated systems need to send commands back to IoT devices (e.g., turning on a pump).
- **Device-to-Device (D2D) Communication:** Local coordination between devices (e.g., smart lights syncing with motion sensors).
- **Scalability & Interoperability:** Standardized networking protocols allow diverse devices from different manufacturers to interoperate seamlessly.

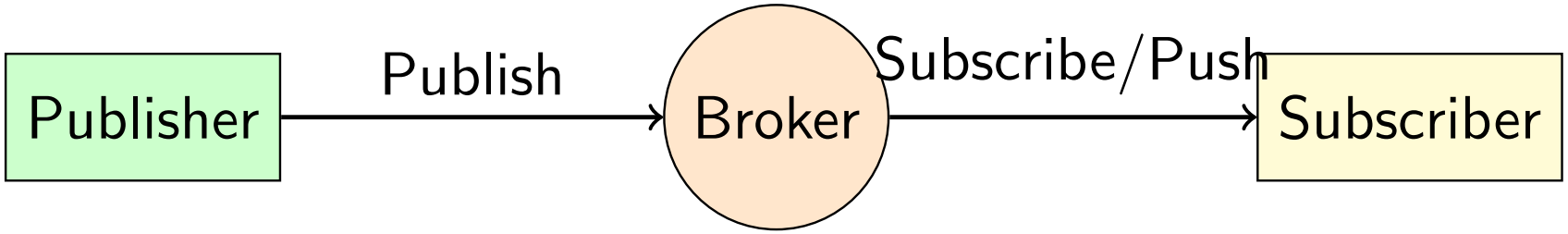
Client-Server Architecture

- **Definition:** A request-response model where a client requests resources or services, and a central server provides them (e.g., HTTP).
- **Synchronous Communication:** The client typically waits for a response from the server.
- **Coupling:** High spatial and temporal coupling. The client must know the server's address, and both must be online simultaneously.
- **IoT Challenges:** Polling for updates drains battery. Scaling to millions of devices overloads the central server with constant requests.



Publish-Subscribe (Pub-Sub) Architecture

- **Definition:** A decoupled messaging pattern where senders (publishers) send messages to topics, and receivers (subscribers) read from those topics via a central Message Broker.
- **Asynchronous Communication:** Publishers and subscribers do not wait for each other.
- **Decoupling:**
  - **Spatial:** Publishers don't know who subscribers are (no direct IP addressing between them).
  - **Temporal:** They don't need to be online at the same time (Broker handles queuing).
- **Examples:** MQTT, AMQP.



Client-Server vs Publish-Subscribe in IoT

Feature	**Client-Server**	**Publish-Subscribe**
Communication Paradigm	Request/Response	Event-driven/Push
State & Connection	Often active connections maintained	Clients only connect to broker
Bandwidth Efficiency	Heavier headers (e.g., HTTP)	Lightweight headers (e.g., MQTT)
Coupling	High (Spatial & Temporal)	Low (Decoupled)
Typical IoT Use Case	Firmware update, static config	Continuous telemetry data

IP Addressing in IoT: IPv4 vs IPv6

- **IPv4 Limitations:** 32-bit addresses ($\sim$4.3 billion). Exhausted rapidly due to the explosion of IoT devices.
- **IPv6 Solution:** 128-bit addresses. Provides an effectively infinite number of addresses (3.4×10^{38}), essential for billions of sensors.
- **LoWPAN (IPv6 over Low-Power Wireless Personal Area Networks):**
  - Adapts IPv6 for constrained devices (e.g., IEEE 802.15.4 / Zigbee).
  - Features header compression to fit IPv6 packets into tiny 127-byte frames.
  - Enables end-to-end IP connectivity even for tiny battery-powered sensors.

MAC Addresses in IoT

- **Definition:** Media Access Control (MAC) address is a unique, hardware-level identifier assigned to a network interface controller (NIC).
- **Layer 2 Addressing:** Operates at the Data Link layer of the OSI model.
- **Format:** 48-bit address (e.g., `00:1A:2B:3C:4D:5E`). Globally unique, assigned by the manufacturer.
- **Role in IoT:**
  - Used for local communication within the same network segment (e.g., sensor to local gateway).
  - Used in ARP (Address Resolution Protocol) to map IP addresses to physical MAC addresses.

Port Numbers and Communication

- **Definition:** A 16-bit logical construct (0 to 65535) identifying a specific process or service on a device.
- **Role in IoT:** Directs incoming network traffic to the correct application layer protocol.
- **Common IoT Ports:**
  - **1883:** MQTT (unencrypted)
  - **8883:** MQTT over TLS/SSL (encrypted)
  - **5683:** CoAP (Constrained Application Protocol)
  - **5684:** CoAP over DTLS
- **Socket:** An IP address paired with a Port number (e.g., `192.168.1.50:1883`) forms a socket for an end-to-end connection.

URLs and Domain Names in IoT

- **URL (Uniform Resource Locator):** A reference to a web resource specifying its location and the protocol to retrieve it.
  - Example: `mqtts://broker.hivemq.com:8883/topic` or `coap://sensor.local/temp`
- **Domain Name:** A human-readable string (e.g., `broker.hivemq.com`) that abstracts complex IP addresses.
- **Need in IoT:**
  - Hardcoding IP addresses in IoT firmware is a bad practice. Cloud IPs change frequently.
  - Domain names allow dynamic IP reassignment without updating millions of deployed devices.

DNS Resolution Process in IoT

- **DNS (Domain Name System):** The internet's phonebook, translating domain names into IP addresses.
- **Process for an IoT Device:**
  1. IoT device needs to connect to `api.iot-platform.com`.
  2. Device sends a DNS Query to a local DNS server (provided by DHCP).
  3. DNS server responds with the IP address (e.g., `203.0.113.45`).
  4. Device establishes a TCP/UDP socket connection using the resolved IP.
- **mDNS (Multicast DNS):** Often used in local IoT networks (e.g., smart home) to resolve `.local` names without a central DNS server.

Key Takeaways

- Publish-Subscribe is the dominant IoT communication model due to temporal and spatial decoupling.
- IPv6 is essential for IoT scale, and 6LoWPAN compresses it for low-power wireless networks.
- MAC addresses handle local delivery, while IP addresses handle global routing.
- Port numbers direct traffic to the correct IoT application protocol (like 1883 for MQTT).
- Always use Domain Names in IoT firmware instead of hardcoded IPs to ensure maintainability.

Next Lecture Preview

- **Topic:** 2.2 Network Topologies and Connectivity
- **Subtopics:**
  - Star, Mesh, and Tree Topologies in IoT.
  - Personal Area Networks (PAN) vs Wide Area Networks (WAN).
  - Introduction to Wi-Fi and Bluetooth Low Energy (BLE) for IoT.

IoT Communication Protocols (Part 1)

- HTTP protocol and REST architecture
- MQTT protocol: Working, Broker, Topics, and QoS levels

Agenda

- Learning Objectives
- HTTP Protocol and REST Architecture in IoT
- The Challenge: TCP/IP Overhead
- Introduction to MQTT
- MQTT Architecture: Pub/Sub and Broker
- MQTT Topics and Filtering
- MQTT Quality of Service (QoS) Levels 0, 1, and 2
- Summary & Next Lecture Preview

Learning Objectives

- Explain the working of HTTP and REST APIs in an IoT context.
- Analyze the overhead of traditional TCP/IP protocols for resource-constrained devices.
- Describe the MQTT Publish/Subscribe architecture and the role of the Broker.
- Differentiate between MQTT QoS levels (0, 1, 2) and their use cases.

HTTP Protocol Basics in IoT

- **HTTP (Hypertext Transfer Protocol)**: A synchronous, request-response protocol running over TCP.
- **Client-Server Model**: IoT device (client) sends an HTTP Request (GET, POST, PUT, DELETE); Server responds.
- **Heavy Headers**: HTTP requests contain verbose ASCII headers (e.g., User-Agent, Accept, Content-Type).
- **Connection Overhead**: Requires opening and closing TCP connections or maintaining keep-alives, which drains battery.

REST Architecture in IoT

- **REST (Representational State Transfer)**: Architectural style for web services using HTTP methods.
- **Resources**: Data (e.g., sensor readings) is treated as a resource identified by a URI (e.g., `/api/v1/sensors/temperature`).
- **Stateless**: Each request from client to server must contain all information needed to understand the request.
- **Payloads**: Typically uses JSON or XML for payload formatting.
- **Use Case**: Best for integration with enterprise systems, web dashboards, and device configuration.

The Challenge: TCP/IP & HTTP Overhead

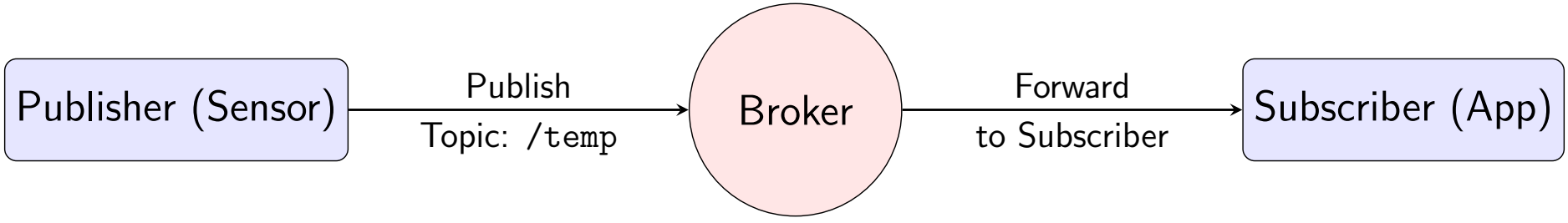
- **TCP 3-Way Handshake**: SYN, SYN-ACK, ACK required before sending any payload.
- **TLS/SSL Handshake**: Adding security requires multiple additional round trips.
- **Header Size**: TCP (20 bytes) + IP (20 bytes) + HTTP headers (often 200+ bytes).
- **Payload Ratio**: Sending a 2-byte temperature value might require 300+ bytes of protocol overhead.
- **Impact**: High latency, high bandwidth consumption, and rapid battery depletion in constrained edge devices.

Introduction to MQTT

- **MQTT (Message Queuing Telemetry Transport)**: An OASIS standard messaging protocol for IoT.
- **Lightweight**: Minimal packet overhead (fixed header is just 2 bytes).
- **Data Agnostic**: Payload can be anything (binary, JSON, text).
- **Asynchronous**: Designed for intermittent, unreliable networks with high latency.
- **Stateful Connection**: Maintains a persistent connection, avoiding repeated handshakes.

MQTT Architecture: Pub/Sub Model

- **Publish/Subscribe Pattern**: Decouples the sender (Publisher) from the receiver (Subscriber) in space and time.
- **Clients**: Devices that publish data or subscribe to data. They never connect directly to each other.
- **Broker**: The central server that receives all messages, filters them, and routes them to interested subscribers.
- **Workflow**: Sensor (Pub) $\rightarrow$ Broker $\rightarrow$ Mobile App (Sub).



MQTT Topics and Filtering

- **Topics**: A UTF-8 string used by the broker to filter messages for each connected client.
- **Hierarchy**: Topics use a forward slash / as a delimiter (e.g., `home/livingroom/temperature`).
- **Wildcards (Subscriptions Only)**:
  - **Single-level (+)**: Matches one level (e.g., `home/+/temperature` matches livingroom and kitchen).
  - **Multi-level (#)**: Matches all subsequent levels (e.g., `home/#` matches everything in home).
- **No Pre-configuration**: Topics are created dynamically when a client publishes to them.

MQTT QoS Levels: Overview

- **Quality of Service (QoS)**: An agreement between the sender and receiver regarding the guarantee of message delivery.
- MQTT provides 3 QoS levels:
  - **QoS 0**: At most once (Fire and forget).
  - **QoS 1**: At least once (Acknowledged delivery).
  - **QoS 2**: Exactly once (Assured delivery).
- **Trade-off**: Higher QoS guarantees delivery but increases protocol overhead and latency.

QoS Level	Name	Guarantee	Overhead
0	At most once	None (Fire & forget)	Low
1	At least once	Acknowledged	Medium
2	Exactly once	Assured delivery	High

MQTT QoS 0: At most once

- **Mechanism**: 'Fire and forget'. The message is sent once without any acknowledgment.
- **Overhead**: Minimal. No retries, no queuing if the client is disconnected.
- **Reliability**: Relies purely on the underlying TCP connection.
- **Use Case**: High-frequency telemetry data where the loss of a single reading doesn't matter (e.g., reporting temperature every second).

MQTT QoS 1 & 2: Higher Guarantees

- **QoS 1 (At least once)**:
  - Sender stores message until it receives a PUBACK from the receiver.
  - Message may be delivered multiple times if the PUBACK is lost and a retry occurs.
  - **Use Case**: Critical alerts where duplicates can be handled.
- **QoS 2 (Exactly once)**:
  - Highest overhead; uses a 4-step handshake (PUBLISH, PUBREC, PUBREL, PUBCOMP).
  - Ensures message is delivered exactly once, eliminating duplicates.
  - **Use Case**: Financial transactions, precise control commands (e.g., incrementing a counter).

Summary

- **HTTP/REST** relies on heavy, synchronous request-response cycles, resulting in large TCP/IP overhead.
- **MQTT** is a lightweight, binary protocol designed specifically for constrained M2M communication.
- **Pub/Sub Model**: MQTT uses a Broker to route messages via hierarchical **Topics**.
- **QoS Levels**: MQTT provides flexible delivery guarantees (QoS 0, 1, 2) to balance reliability against bandwidth usage.

Next Lecture Preview

- **Topic**: IoT Communication Protocols (Part 2)
- **Key Concepts**:
  - CoAP (Constrained Application Protocol) and its similarities to HTTP.
  - WebSockets for full-duplex communication over HTTP.
  - AMQP for enterprise messaging.
  - Comparing MQTT, CoAP, and HTTP for specific IoT use cases.

Agenda

- Introduction to CoAP
- CoAP Architecture and RESTful Model
- CoAP over UDP and Message Types
- URI Mapping in CoAP
- The Observe Pattern
- Comparison: HTTP vs MQTT vs CoAP

Learning Objectives

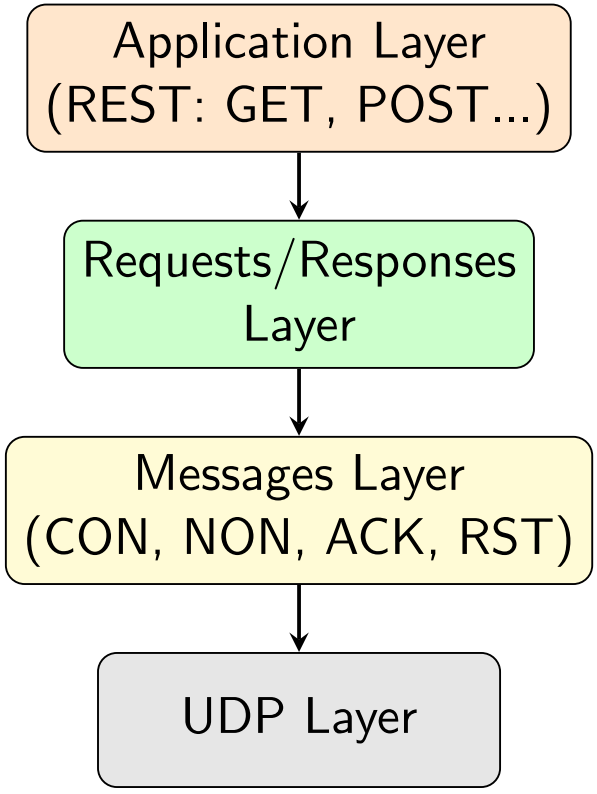
- Understand the necessity and architecture of CoAP for constrained devices.
- Analyze the mechanisms of CoAP messaging over UDP.
- Comprehend CoAP's URI mapping and the Observe pattern for resource monitoring.
- Evaluate and compare HTTP, MQTT, and CoAP based on technical parameters.

Introduction to CoAP

- Constrained Application Protocol (CoAP) defined in RFC 7252.
- Designed specifically for constrained devices (low RAM/ROM) and lossy networks.
- Provides a RESTful framework (like HTTP) but with much lower overhead.
- Ideal for M2M (Machine-to-Machine) communication in smart energy and building automation.

CoAP Architecture and Request/Response Model

- Follows a Client-Server architecture.
- Utilizes standard REST methods: GET, POST, PUT, DELETE.
- Features a dual-layer approach: Messages layer (for reliability) and Request/Response layer.



CoAP over UDP: Why UDP?

- CoAP primarily uses UDP (User Datagram Protocol) rather than TCP.
- Lower Overhead: UDP eliminates TCP's 3-way handshake and connection state maintenance.
- Reduced Latency: Faster transmission suited for small IoT data packets.
- Suitability for LPWAN: UDP is better for low-bandwidth, high-latency networks.

CoAP Message Types

- **Confirmable (CON):** Requires an Acknowledgement (ACK) from the receiver.
- **Non-Confirmable (NON):** Does not require an ACK (fire-and-forget).
- **Acknowledgement (ACK):** Acknowledges a specific CON message.
- **Reset (RST):** Indicates a message was received but context is missing or it cannot be processed.

CoAP URI Mapping and RESTful Approach

- Uses URIs similar to HTTP: `coap://server.com:5683/sensors/temperature`
- Default UDP port for CoAP is 5683 (5684 for secure CoAPS).
- Resources are discoverable via a standard endpoint: `/.well-known/core`
- Payloads are often in lightweight formats like CBOR, JSON, or plain text.

```
GET /sensors/temperature
Host: server.com
Port: 5683
```

The CoAP Observe Pattern

- Described in RFC 7641, extends the basic GET method.
- Client registers interest in a resource using the 'Observe' option.
- Server sends asynchronous notifications whenever the resource state changes.
- Eliminates the need for continuous polling, saving battery and bandwidth.

Protocol Comparison: HTTP vs MQTT vs CoAP

- HTTP: Heavyweight, document-centric, synchronous Request/Response over TCP.
- MQTT: Lightweight, data-centric, asynchronous Publish/Subscribe over TCP.
- CoAP: Lightweight, resource-centric, asynchronous Request/Response over UDP.
- Selection depends on device constraints, network reliability, and architectural needs.

Comparison Table: Architecture & Transport

Feature	**HTTP**	**MQTT**	**CoAP**
Architecture	Client-Server	Pub/Sub (Broker)	Client-Server
Transport Layer	TCP	TCP	UDP
Header Size	Large (Bytes/KB)	Small (2 Bytes min)	Small (4 Bytes min)
Statefulness	Stateless	Stateful (Sessions)	Stateless

Comparison Table: Messaging & Reliability

Feature	**HTTP**	**MQTT**	**CoAP**
Messaging Model	Synchronous	Asynchronous	Asynchronous
Reliability/QoS	Relies on TCP	QoS 0, 1, 2	CON/NON messages
Security	TLS	TLS	DTLS (Datagram TLS)
Resource Discovery	None standard	None standard	Supported (.well-known/core)

Key Takeaways

- CoAP adapts the RESTful web model for highly constrained IoT devices and networks.
- Operating primarily over UDP reduces connection overhead and latency.
- The Observe pattern provides efficient asynchronous notifications without polling.
- MQTT is preferred for scalable Pub/Sub telemetry, while CoAP excels in lightweight Client-Server control scenarios.

Next Lecture Preview

- Topic: 2.3 Short Range Wireless Protocols
- Introduction to Bluetooth Low Energy (BLE)
- Zigbee Architecture and Topologies
- Application areas for short-range protocols

Agenda

- Introduction to Wireless Communication in IoT
- Wi-Fi (IEEE 802.11) Overview and Applications
- Wi-Fi Pros & Cons for IoT
- Bluetooth Classic vs. Bluetooth Low Energy (BLE)
- BLE Concepts: Advertising vs. Connections
- BLE GATT Architecture (Services & Characteristics)
- Wi-Fi vs. BLE: Power Consumption & Use Cases

Learning Objectives

By the end of this lecture, students will be able to:

- Explain the role and limitations of Wi-Fi (IEEE 802.11) in IoT applications.
- Differentiate between Bluetooth Classic and Bluetooth Low Energy (BLE).
- Understand BLE operation modes: Advertising and Connection-oriented.
- Describe the structure of the BLE Generic Attribute Profile (GATT) using Services and Characteristics.
- Compare Wi-Fi and BLE based on power consumption, range, and bandwidth.

Introduction to Wireless Communication in IoT

- **Role in IoT:** Enables untethered connectivity for sensors and actuators to edge gateways or the cloud.
- **Design Trade-offs:**
  - Range (Local area vs. Wide area)
  - Bandwidth (High data rate vs. Low data rate)
  - Power Consumption (Mains powered vs. Battery powered)
- **Local Area Technologies:** Wi-Fi, Bluetooth, Zigbee
- **Wide Area Technologies:** Cellular, LoRaWAN, NB-IoT (Covered in future lectures)

Wi-Fi (IEEE 802.11) Overview

- **Standard:** IEEE 802.11 family (a/b/g/n/ac/ax/ah)
- **Frequencies:** Primarily 2.4 GHz and 5 GHz ISM bands.
- **Topology:** Star topology (Access Point/Router to Stations/Clients).
- **High Bandwidth:** Capable of transmitting large amounts of data (video, audio, bulk data).
- **IP Native:** Directly supports TCP/IP, allowing seamless integration with the Internet and Cloud services without translation gateways.

Wi-Fi in IoT: Pros & Cons

- **Pros:**
  - High data throughput (Megabits to Gigabits per sec).
  - Existing infrastructure (routers everywhere).
  - Direct IP connectivity (HTTP, MQTT, WebSockets).
- **Cons:**
  - **High Power Consumption:** Constantly maintaining a connection drains batteries quickly.
  - Complex provisioning (requires SSID and Password setup on headless devices).
  - Sub-optimal for small, battery-operated coin-cell sensors.

Bluetooth Classic vs. BLE (Bluetooth Low Energy)

- **Bluetooth Classic (BR/EDR):**
  - Designed for continuous streaming (audio headsets, file transfers).
  - Higher throughput, higher power consumption.
- **Bluetooth Low Energy (BLE / Bluetooth Smart):**
  - Introduced in Bluetooth 4.0.
  - Designed for short bursts of data (sensor readings, state changes).
  - Sleeps most of the time, resulting in ultra-low power consumption.
  - Incompatible with Classic Bluetooth at the physical layer.

BLE Basic Concept & Architecture

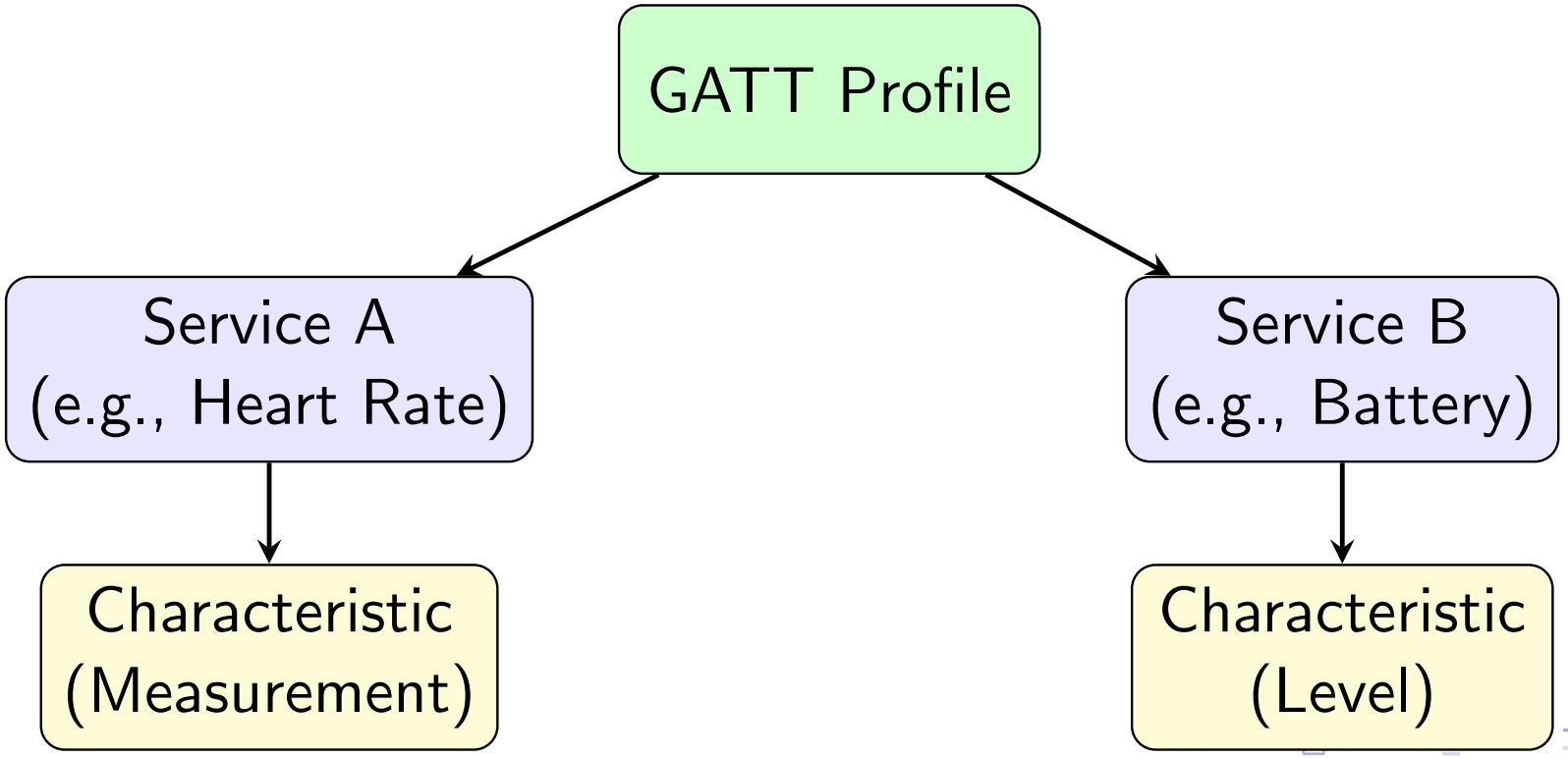
- **Topology:** Star network (Central device connects to multiple Peripherals).
- **Roles:**
  - **Peripheral:** The sensor/device (e.g., heart rate monitor, ESP32 acting as a sensor). Advertises its presence.
  - **Central:** The smartphone or gateway. Scans for peripherals and initiates connection.
- **Frequency:** 2.4 GHz ISM band, uses 40 channels (3 for advertising, 37 for data) with Frequency Hopping Spread Spectrum (FHSS) to avoid interference.

BLE: Advertising vs. Connections

- **Advertising (Broadcasting):**
  - Peripheral blindly transmits small packets (up to 31 bytes) at set intervals.
  - One-to-many communication.
  - Examples: iBeacons, Eddystone, temperature broadcasts.
  - No pairing required.
- **Connections:**
  - Central discovers Peripheral via advertising, then sends a connect request.
  - One-to-one, bidirectional, secure communication.
  - Advertising stops once connected.

BLE GATT Architecture

- **GATT:** Generic Attribute Profile. Defines how data is organized and exchanged over a BLE connection.
- **Client-Server Model:**
  - **GATT Server:** Holds the data (usually the Peripheral/Sensor).
  - **GATT Client:** Reads/writes the data (usually the Central/Smartphone).
- **Data Hierarchy:** Profile $\rightarrow$ Services $\rightarrow$ Characteristics.



BLE GATT: Services and Characteristics

- **Service:** A collection of related data. Identified by a UUID (16-bit or 128-bit).
  - *Example:* Heart Rate Service (UUID: 0x180D)
- **Characteristic:** A specific data value within a Service. Has properties (Read, Write, Notify, Indicate).
  - *Example:* Heart Rate Measurement (UUID: 0x2A37)
- **Descriptors:** Additional metadata for a characteristic (e.g., human-readable description, unit of measurement).

Comparison: Wi-Fi vs. BLE

Feature	Wi-Fi (802.11 b/g/n)	Bluetooth Low Energy (BLE)
Power Consumption	High (mains power preferred)	Ultra-Low (coin cell for years)
Data Rate	High (up to hundreds of Mbps)	Low (1-2 Mbps max)
Range	$\sim$50-100 meters	$\sim$10-50 meters (typically)
Internet Connect.	Direct (Native IP)	Requires Gateway (Smartphone/Hub)
Best Use Case	Video streaming, heavy data IoT, smart plugs	Wearables, smart tags, battery sensors

Key Takeaways

- **Wi-Fi** offers high bandwidth and native IP connectivity but consumes too much power for tiny battery-operated sensors.
- **BLE** is optimized for ultra-low power consumption through short data bursts and long sleep cycles.
- BLE operates using **Advertising** for broadcasting and **Connections** for two-way communication.
- The **GATT** profile structures BLE data into Services and Characteristics, operating on a Client-Server model.

Next Lecture Preview

In the next lecture, we will cover:

- **Topic 2.4: Wireless Communication Technologies (Part 2)**
- Zigbee Protocol and Mesh Networking
- Cellular IoT (NB-IoT and LTE-M)
- Introduction to LPWAN (LoRaWAN)

Wireless Communication Technologies (Part 2)

- Unit 2: IoT Communication Technologies
- Focus: LoRa, NB-IoT, and Comparative Analysis

Agenda

- Introduction to LPWAN
- LoRa & LoRaWAN Basics
- Chirp Spread Spectrum (CSS) in LoRa
- Introduction to NB-IoT
- NB-IoT Cellular Bands
- Comparison: Wi-Fi vs. LoRa vs. NB-IoT

Learning Objectives

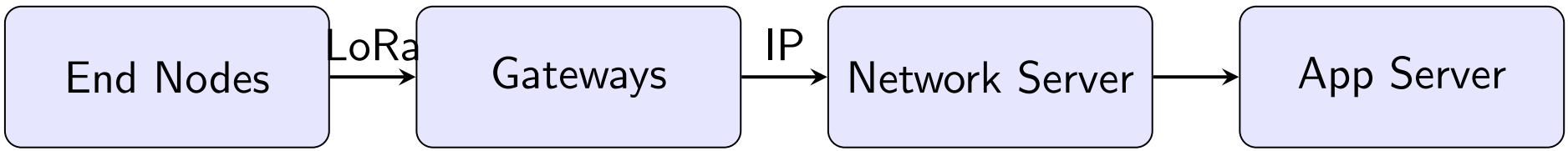
- Understand the characteristics and importance of LPWANs.
- Explain the working principles of LoRa and Chirp Spread Spectrum (CSS).
- Describe NB-IoT architecture and its operation in cellular bands.
- Compare Wi-Fi, LoRa, and NB-IoT based on range, bandwidth, and power consumption.

Introduction to LPWAN

- **LPWAN**: Low Power Wide Area Network.
- Designed for IoT devices needing long-range communication at a low bit rate.
- **Key Characteristics**:
  - **Long Range**: Kilometers in urban areas, tens of kilometers in rural.
  - **Low Power**: Battery life of up to 10 years or more.
  - **Low Cost**: Reduced infrastructure and endpoint costs.

LoRa Basics & Architecture

- **LoRa (Long Range)**: Proprietary physical layer (PHY) modulation technique developed by Semtech.
- Operates in unlicensed sub-GHz ISM bands (e.g., 868 MHz in Europe, 915 MHz in North America, 865-867 MHz in India).
- **Network Architecture**: Star-of-stars topology.



Chirp Spread Spectrum (CSS)

- LoRa uses **Chirp Spread Spectrum (CSS)** modulation.
- **Chirp**: A signal whose frequency increases (up-chirp) or decreases (down-chirp) over time.
- **Benefits of CSS**:
  - High resilience to interference, multi-path fading, and Doppler effect.
  - Extremely low signal-to-noise ratio (SNR) requirements (can receive signals below the noise floor).
  - **Spreading Factor (SF)**: Determines the chirp duration. Higher SF = longer range, lower data rate.

LoRaWAN Protocol & Security

- **LoRaWAN**: The open MAC layer protocol standard built on top of LoRa PHY.
- Defines device classes (Class A, B, C) based on downlink latency and power requirements.
- **Security Features**:
  - Two layers of cryptography: Network session key (NwkSKey) and Application session key (AppSKey).
  - AES-128 encryption ensures data integrity and confidentiality from end-to-end.

Introduction to NB-IoT

- **NB-IoT (Narrowband IoT)**: A 3GPP cellular standard (LTE Cat NB1/NB2) designed for LPWAN.
- Operates in licensed spectrum, managed by telecom operators.
- **Key Goals**:
  - Massive connection density (up to 100k devices per cell).
  - Deep indoor penetration (20 dB better than standard LTE).
  - Simplified modem design for low cost.

NB-IoT Cellular Bands & Deployment

- Requires only 180 kHz of bandwidth (a single LTE Physical Resource Block).
- **Three Deployment Modes**:
  - **In-Band**: Utilizes resource blocks within a normal LTE carrier.
  - **Guard Band**: Deployed in the unused resource blocks between LTE carriers.
  - **Standalone**: Re-purposing GSM (2G) spectrum.

Key Features of NB-IoT

- **Power Saving Mode (PSM)**: Allows the device to go into a deep sleep while remaining registered to the network.
- **Extended Discontinuous Reception (eDRX)**: Lengthens the paging cycle, allowing the device to sleep longer between listening for network pages.
- **Security**:
  - Leverages existing 3GPP LTE security mechanisms (SIM-based authentication).

Comparison: Wi-Fi, LoRa, and NB-IoT

Feature	Wi-Fi	LoRa	NB-IoT
Range	Short ($<$ 100m)	Long (up to 15km)	Long (up to 15km, deep penetration)
Bandwidth	High (Mbps/Gbps)	Low (0.3 - 50 kbps)	Low (up to 250 kbps)
Power	High	Very Low	Low
Spectrum	Unlicensed	Unlicensed	Licensed

Applications & Use Cases

- **LoRa Use Cases**:
  - Smart Agriculture (soil moisture sensing).
  - Supply chain tracking in rural areas.
  - Campus-wide smart lighting.
- **NB-IoT Use Cases**:
  - Smart Metering (water/gas meters in basements).
  - Smart City infrastructure (parking sensors).
  - Wearable health monitors.

Key Takeaways

- LPWAN technologies prioritize range and battery life over data rates.
- LoRa uses CSS modulation in unlicensed bands and is ideal for private network deployments.
- NB-IoT uses licensed cellular bands, offering high reliability, deep penetration, and QoS.
- Selecting between Wi-Fi, LoRa, and NB-IoT depends entirely on the specific application's range, data, and power requirements.

Next Lecture Preview

- Unit 2: 2.4 Wireless Sensor Networks (WSN)
- Topics to cover:
  - Architecture of WSNs
  - Sensor node components
  - Routing protocols in WSN

Agenda

- Concept of Application Programming Interfaces (APIs)
- Introduction to REST API Architecture
- JSON Data Format Fundamentals
- Working with JSON Key-Value Pairs
- Device-to-Server Data Exchange
- HTTP GET and POST with JSON Payloads
- Parsing JSON on Microcontrollers (ESP32)

Learning Objectives

- Define the role of APIs as communication bridges in IoT systems.
- Explain the core principles of RESTful architecture and stateless communication.
- Construct and validate proper JSON data structures using key-value pairs.
- Differentiate between HTTP GET and POST methods in the context of JSON payloads.
- Describe the memory constraints and techniques for parsing JSON on microcontrollers.

Concept of API (Application Programming Interface)

- Definition: A set of rules and protocols allowing different software applications to communicate.
- IoT Context: Acts as the intermediary between hardware (edge devices) and cloud services.
- Abstraction: Hides the complex backend database operations and exposes standard endpoints.
- Standardization: Ensures that an ESP32, a mobile app, and a web dashboard can all consume the same data seamlessly.
- Authentication: APIs often require API Keys or tokens to ensure secure access.

REST API Architecture

- REST: Representational State Transfer, an architectural style for web services.
- Stateless: Every HTTP request contains all necessary information; the server stores no client session context.
- Resource-Based: Data entities (like 'sensors' or 'users') are exposed as URIs (Uniform Resource Identifiers).
- Standard Methods: Maps CRUD (Create, Read, Update, Delete) operations to HTTP verbs.
- Uniform Interface: Standardized communication decoupling the client architecture from the server architecture.

Introduction to JSON Data Format

- JSON: JavaScript Object Notation.
- Purpose: A lightweight, text-based, language-independent data interchange format.
- Why JSON in IoT?: Much less overhead compared to XML, resulting in smaller payloads and lower bandwidth usage.
- Readability: Easy for humans to read and write, while being simple for machines to parse and generate.
- Widespread Support: Virtually all modern cloud platforms (AWS, Azure, GCP) use JSON natively.

JSON Key-Value Pairs and Structure

- Objects: Enclosed in curly braces {}. Represents a collection of name/value pairs.
- Key-Value Pairs: Formatted as `"key": value`. Keys MUST be strings enclosed in double quotes.
- Data Types Supported:
  - String: `"ESP32_Node"`
  - Number (Integer/Float): `24.5`
  - Boolean: `true` or `false`
  - Array: Ordered list enclosed in brackets `[]`, e.g., `[1, 2, 3]`
  - Null: `null` representing empty values.

JSON Payload Example for IoT

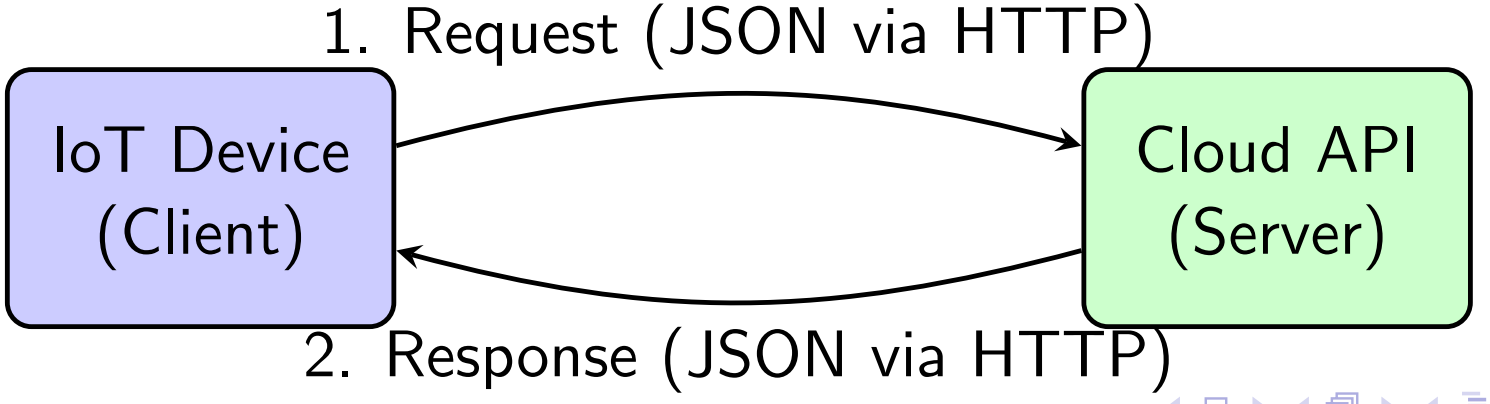
Example of a telemetry payload from a weather station:

```
{
  "device_id": "weather_esp_01",
  "timestamp": 1690003200,
  "sensors": {
    "temperature": 28.4,
    "humidity": 65
  },
  "status_ok": true
}
```

Note the nested 'sensors' object and the mix of string, integer, float, and boolean types.

Data Exchange Between Device and Server

- Client-Server Model: The IoT device usually acts as the HTTP Client, initiating requests to the Cloud HTTP Server.
- Serialization: The process of converting MCU variables (C/C++ structs) into a JSON formatted string for transmission.
- Deserialization: The process of parsing an incoming JSON string back into MCU variables.
- API Endpoints: Specific URLs defined by the server for data exchange (e.g., `api.example.com/v1/telemetry`).
- Headers: Required metadata, particularly `Content-Type: application/json`.



HTTP GET Requests in IoT

- Purpose: Used to retrieve data or configuration from the server.
- Mechanics: Parameters are passed in the URL (Query String), NOT in the body.
- IoT Use Case: An ESP32 booting up and requesting its latest configuration or operating thresholds.
- Response: The server replies with an HTTP 200 OK and a JSON body containing the requested data.
- Idempotent: Executing a GET request multiple times does not alter the server's state.

HTTP POST Requests with JSON Payloads

- Purpose: Used to submit new data to the server (creating resources).
- Mechanics: The JSON payload is placed in the HTTP Request Body.
- Mandatory Header: `Content-Type: application/json` tells the server how to interpret the body.
- IoT Use Case: Pushing periodic sensor telemetry data to a cloud dashboard or database.
- Security: Often requires an Authorization header (e.g., Bearer Token) to validate the device.

Method	**Data Location**	**Common IoT Usage**
GET	URL (Query string)	Retrieve settings, sync time
POST	Request Body (JSON)	Send telemetry, upload logs

Parsing JSON on Microcontrollers

- Challenge: JSON strings are dynamic, but MCUs like the ESP32 have limited RAM and lack built-in garbage collection.
- Standard Library: `ArduinoJson` is the industry standard for C++ IoT applications.
- Memory Management: Requires pre-allocating a `JsonDocument`.
- `StaticJsonDocument`: Allocates on the stack (for small, fixed payloads).
- `DynamicJsonDocument`: Allocates on the heap (for larger, variable payloads).
- Extraction syntax:

```
float temp = doc["temperature"];
bool status = doc["status_ok"];
```

Key Takeaways

- APIs provide a standardized interface for hardware devices to interact with cloud services.
- RESTful architecture relies on stateless, resource-based HTTP communication.
- JSON is the dominant IoT data format due to its lightweight, key-value structure.
- HTTP POST is used to upload sensor telemetry (JSON body), while GET is used to retrieve configurations.
- Microcontrollers require specialized memory-managed libraries (like ArduinoJson) to safely parse JSON strings.

Next Lecture Preview

- Topic: 2.5 Message Queuing Telemetry Transport (MQTT) Protocol
- The Publish-Subscribe Architecture
- MQTT Broker, Clients, and Topics
- Comparing MQTT vs. HTTP for IoT applications
- Quality of Service (QoS) levels in MQTT

Intro to IoT Cloud Computing (Part 1)

- Course: Internet of Things
- Unit 2
- Lecture: Lecture 2.7: Intro to IoT Cloud Computing (Part 1)

Agenda

- Concept of Cloud Computing
- Essential Characteristics of Cloud
- The Need for Cloud Computing in IoT
- Introduction to Cloud Service Models
- IaaS: Infrastructure as a Service
- PaaS: Platform as a Service (AWS IoT Core, Azure IoT Hub)
- SaaS: Software as a Service

Learning Objectives

- Define Cloud Computing and identify its core characteristics.
- Understand the critical need for cloud infrastructure in scalable IoT deployments.
- Differentiate between IaaS, PaaS, and SaaS cloud service models.
- Relate PaaS specifically to IoT backend platforms like AWS IoT Core and Azure IoT Hub.

Concept of Cloud Computing

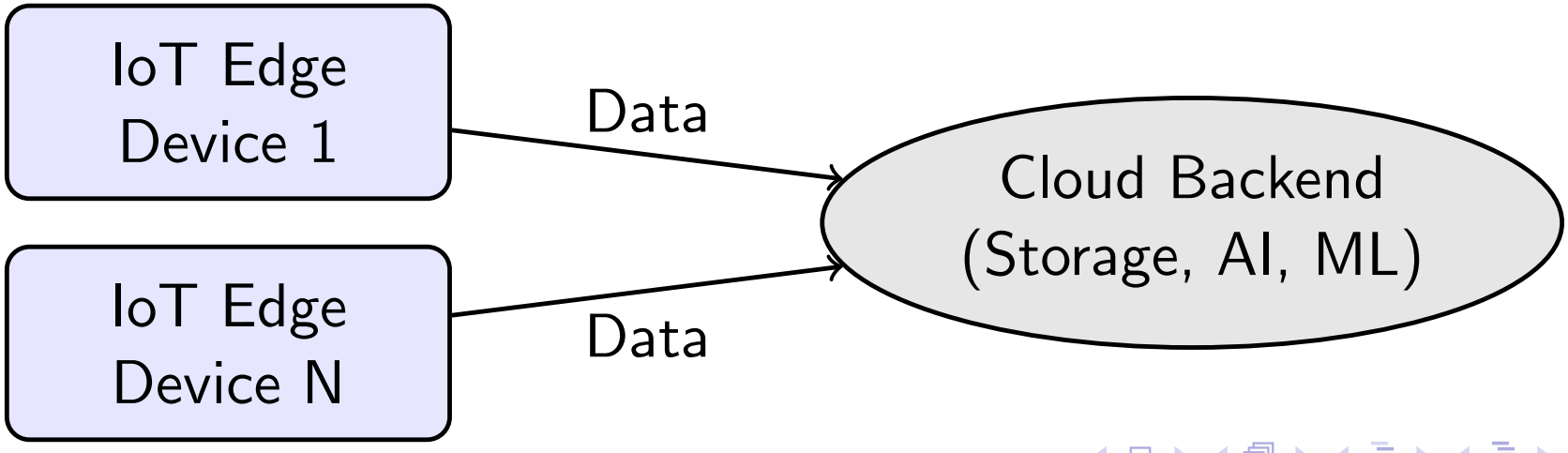
- Cloud Computing is the on-demand availability of computer system resources, especially data storage and computing power.
- It eliminates the need for direct active management by the user.
- Provides shared pools of configurable resources (networks, servers, storage, applications, and services).
- Resources can be rapidly provisioned and released with minimal management effort.

Essential Characteristics of the Cloud

- On-demand self-service: Users can provision resources without human interaction with the service provider.
- Broad network access: Capabilities are available over the network through standard mechanisms.
- Resource pooling: Provider's computing resources are pooled to serve multiple consumers using a multi-tenant model.
- Rapid elasticity: Capabilities can be elastically provisioned and released to scale rapidly outward and inward.
- Measured service: Cloud systems automatically control and optimize resource use (pay-as-you-go).

The Need for Cloud Computing in IoT

- Massive Data Storage: Millions of IoT edge devices generate petabytes of telemetry data needing centralized storage.
- Computational Power: Complex analytics, Machine Learning (ML), and AI require more processing power than edge devices possess.
- Global Scalability: Cloud allows IoT applications to scale seamlessly from hundreds to millions of connected devices.
- Ubiquitous Access: Centralized cloud backends allow users to monitor and control IoT devices from anywhere in the world.



Challenges Solved by Cloud in IoT

- Device Management: Cloud platforms provide registries to track, update (OTA), and secure fleets of devices.
- Interoperability: Cloud acts as a middleware hub, translating protocols (MQTT, CoAP, HTTP) to unify diverse hardware.
- High Availability: Cloud providers ensure 99.99% uptime, crucial for mission-critical IoT systems (e.g., healthcare monitoring).
- Cost Efficiency: Eliminates upfront capital expenditure (CapEx) for on-premise servers in favor of operational expenditure (OpEx).

Introduction to Cloud Service Models

- Cloud computing is typically categorized into three standard service models.
- IaaS (Infrastructure as a Service): Renting fundamental computing resources (virtual machines, storage).
- PaaS (Platform as a Service): Providing a platform allowing customers to develop, run, and manage applications.
- SaaS (Software as a Service): Delivering software applications over the internet, on demand and typically on a subscription basis.

Layer	**IaaS**	**PaaS**	**SaaS**
Applications	User	User	Provider
Data	User	User	Provider
Runtime, Middleware, OS	User	Provider	Provider
Virtualization, Servers, Storage, Networking	Provider	Provider	Provider

IaaS: Infrastructure as a Service

- Provides highly scalable and automated compute resources, storage, and networking.
- User manages: Operating System, Middleware, Runtime, Data, Applications.
- Provider manages: Virtualization, Servers, Storage, Networking.
- Examples: Amazon EC2, Google Compute Engine (GCE), Microsoft Azure VMs.

PaaS: Platform as a Service

- Provides a framework for developers to build upon and create customized applications.
- Abstracts away underlying infrastructure (OS, servers, storage, networking).
- User manages: Applications and Data.
- Provider manages: Everything else, including runtime, middleware, and O/S.
- Examples: AWS Elastic Beanstalk, Google App Engine, Heroku.

PaaS as IoT Backend Platforms

- In IoT, PaaS manifests as specialized 'IoT Platforms'.
- AWS IoT Core: A managed cloud platform that lets connected devices easily and securely interact with cloud applications.
- Azure IoT Hub: A managed service hosted in the cloud that acts as a central message hub for bi-directional communication between IoT application and devices.
- These platforms provide built-in MQTT brokers, device shadows/twins, rules engines, and security frameworks.

SaaS: Software as a Service

- Delivers fully functional, ready-to-use software applications over the internet.
- User manages: Nothing (except configuring user preferences and access).
- Provider manages: The entire stack, from infrastructure to the application code and data storage.
- IoT Examples: A complete smart home dashboard app, fleet tracking software, or an industrial IoT analytics portal (e.g., Datadog, Salesforce IoT).

Key Takeaways

- Cloud Computing provides scalable, on-demand compute and storage, essential for processing massive IoT telemetry data.
- IaaS gives you virtual machines and storage, requiring you to manage the OS and middleware.
- PaaS abstracts infrastructure, allowing developers to focus on applications. AWS IoT Core and Azure IoT Hub are prime examples of IoT PaaS.
- SaaS delivers fully managed end-user applications over the web.

Next Lecture Preview

- Topic: Intro to IoT Cloud Computing (Part 2)
- Deep dive into Cloud Deployment Models.
- Public vs. Private vs. Hybrid vs. Community Clouds.
- Understanding which deployment model is best for different IoT scenarios (e.g., Industrial vs. Consumer IoT).

Agenda

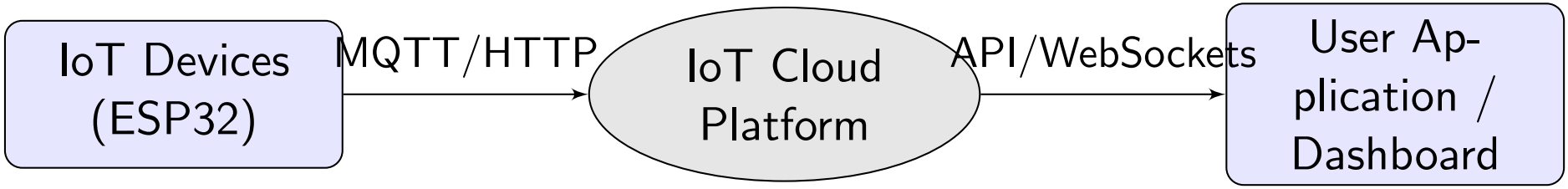
- Recap of IoT Cloud Architecture
- ThingSpeak Platform
- Blynk IoT Platform
- Arduino IoT Cloud
- Ubidots Platform
- ThingsBoard Overview
- Comparative Analysis of Platforms

Learning Objectives

- Understand the role of specific IoT cloud platforms.
- Identify the key features of ThingSpeak, Blynk, Arduino IoT Cloud, Ubidots, and ThingsBoard.
- Evaluate which platform is best suited for different IoT use cases.
- Learn how these platforms integrate with hardware like ESP32.

Recap: The Role of IoT Cloud Platforms

- Acts as the central nervous system for IoT networks.
- Provides services for Data Ingestion, Storage, and Analytics.
- Enables Device Management and over-the-air (OTA) updates.
- Offers visualization tools (dashboards) and rule engines (alerts).
- Facilitates secure communication via MQTT, HTTP, and CoAP.



ThingSpeak: Data Collection and Analysis

- Developed by MathWorks, tailored for IoT analytics.
- Core Feature: Seamless integration with MATLAB for data analysis.
- Channels: Data is stored in 'Channels' (up to 8 fields of data, plus location and status).
- Communication: Supports REST API and MQTT.
- Use Case: Sensor data logging and predictive maintenance.

ThingSpeak: Features and Limitations

Features:

- Real-time data visualization via charts.
- Event scheduling (TimeControl) and action triggers (React).
- Public and private data channels.

Limitations:

- Free tier limits update rates (15 seconds per message).
- UI is less customizable compared to other platforms.

Blynk: Rapid Mobile App Development

- Hardware-agnostic IoT platform designed for rapid prototyping.
- Blynk 2.0: Includes Web Dashboard, Mobile App (iOS/Android), and Cloud.
- Virtual Pins: A unique concept to send/receive data between hardware and app.
- Communication: Highly optimized custom binary protocol over TCP/IP.
- Use Case: Smart home control, remote switching, and monitoring.

Blynk: Architecture and Capabilities

- Blynk Server: Can use the public cloud or host a private local server.
- Blynk Library: Runs on ESP32/Arduino, handling connection management.

Features:

- Device provisioning (WiFi setup via app).
- OTA (Over-The-Air) firmware updates.
- Role-based access and device sharing.

Arduino IoT Cloud

- An all-in-one platform for creating IoT applications easily.
- Thing-centric approach: Define variables, and the cloud auto-generates the sketch.
- Device Support: Officially supports Arduino boards, ESP8266, and ESP32.

Features:

- Web Editor integration.
- Automatic variable synchronization.
- Built-in dashboards.

Ubidots: Application Enablement

- A platform focused on data analytics, visualization, and application enablement.
- Data Hierarchy: Devices $\rightarrow$ Variables $\rightarrow$ Values.

Features:

- Synthetic Variables: Compute new data from existing variables using math expressions.
- High-quality, highly customizable drag-and-drop dashboards.
- Email, SMS, and webhook alerts.

ThingsBoard: Open-Source IoT Platform

- A robust, open-source platform for data collection, processing, and device management.
- Architecture: Highly scalable, built on Java and Cassandra/PostgreSQL.

Features:

- Rule Engine: Complex event processing via drag-and-drop node graph.
- Multi-tenancy: Support for multiple customers and users in one instance.
- Protocols: MQTT, CoAP, and HTTP.

Comparison of IoT Platforms

Platform	Best For	Key Feature
ThingSpeak	Data Logging + Analytics (MATLAB)	Basic UI, MathWorks backend
Blynk	Mobile App Control + Prototyping	Rapid UI, Virtual Pins
Arduino IoT Cloud	Ease of use, Beginners	Auto-generated code
Ubidots	Professional Dashboards	Cloud-side math
ThingsBoard	Enterprise, Open-Source	Complex Rules, Multi-tenancy

Summary

- IoT cloud platforms provide essential infrastructure for device connectivity, data storage, and user interfaces.
- ThingSpeak focuses on analytics and MATLAB integration.
- Blynk provides a rapid way to build mobile and web interfaces using Virtual Pins.
- Arduino IoT Cloud simplifies development through auto-generated code.
- Ubidots and ThingsBoard offer professional, scalable solutions with advanced logic and dashboards.

Next Lecture Preview

- Topic: 2.6 IoT Edge Computing
- What is Edge Computing?
- Cloud vs. Edge: The Paradigm Shift
- Implementing basic Edge processing on ESP32
- Reducing latency and bandwidth in IoT systems

Agenda

- Introduction to ESP32
- ESP32-WROOM-32 Module Overview
- Key Features of ESP32
- Tensilica Xtensa Dual-Core LX6 Processor
- CPU Clocks and Performance
- Internal Memory (ROM and SRAM)
- External Flash Memory
- Wireless Capabilities
- Power Management
- Summary and Next Lecture Preview

Learning Objectives

- Understand the architecture and ecosystem of the ESP32 SoC.
- Identify the components and pinout of the ESP32-WROOM-32 module.
- Analyze the specifications of the Tensilica Xtensa Dual-Core LX6 processor.
- Comprehend the memory organization (ROM, SRAM, Flash) in ESP32.

Introduction to ESP32

- **Creator:** Developed by Espressif Systems.
- **Nature:** Low-cost, low-power system on a chip (SoC).
- **Integration:** Highly integrated with built-in Wi-Fi and dual-mode Bluetooth.
- **Target Applications:** Designed for mobile, wearable electronics, and Internet of Things (IoT) applications.
- **Successor:** The successor to the highly popular ESP8266 microcontroller.

ESP32-WROOM-32 Module Overview

- **Module vs. SoC:** The ESP32 is the bare chip; the WROOM-32 is a certified module containing the chip.
- **Components:** Includes the ESP32-D0WDQ6 chip, a crystal oscillator (typically 40 MHz), filtering capacitors, and matching networks.
- **Antenna:** Features an integrated PCB trace antenna or an IPEX connector for an external antenna (in variants like WROVER).
- **Shielding:** Enclosed in a metal shield to reduce EMI and aid in FCC/CE certification.

Key Features of ESP32

- **Processing Power:** Dual-core processor with high clock speeds.
- **Wireless Connectivity:** 2.4 GHz Wi-Fi (802.11 b/g/n) and Bluetooth (v4.2 BR/EDR and BLE).
- **Rich Peripherals:** Capacitive touch, ADCs, DACs, I2C, SPI, UART, I2S, CAN bus.
- **Low Power:** Advanced power management with deep sleep modes consuming as little as 10 µA.
- **Security:** Hardware accelerators for AES, SHA-2, RSA, and secure boot.

Tensilica Xtensa Dual-Core LX6 Processor

- **Architecture:** 32-bit RISC architecture by Cadence Tensilica.
- **Cores:** Two independently controlled CPU cores (PRO_CPU and APP_CPU).
- **PRO_CPU (Protocol CPU):** Typically handles Wi-Fi, Bluetooth, and internal background tasks.
- **APP_CPU (Application CPU):** Available exclusively for user application code (when using FreeRTOS).
- **Instructions:** Supports 16-bit and 32-bit instructions, optimizing code density.

Clock Speed and CPU Performance

- **Clock Frequency:** Adjustable from 80 MHz to 240 MHz.
- **Performance:** Up to 600 DMIPS (Dhrystone Million Instructions Per Second) at 240 MHz.
- **Dynamic Frequency Scaling:** The clock speed can be reduced to save power when high performance is not required.
- **FPU:** Includes a Single-Precision Floating Point Unit (FPU) for complex mathematical operations.

Internal Memory (ROM and SRAM)

- **ROM (448 KB):** Used for booting and core functions. Contains the bootloader and low-level drivers.
- **SRAM (520 KB):** Fast on-chip memory for data and instructions.
- **SRAM Organization:**
  - **SRAM0 (192 KB):** Instruction RAM (IRAM).
  - **SRAM1 & SRAM2 (328 KB):** Data RAM (DRAM).
- **RTC FAST/SLOW Memory (8 KB each):** Retained during Deep Sleep. Accessed by the ULP (Ultra-Low Power) coprocessor.

External Flash Memory

- **No Internal Flash:** The ESP32 SoC does not contain internal flash memory for storing user code.
- **SPI Flash:** Relies on external SPI Flash memory (integrated inside the WROOM-32 module).
- **WROOM-32 Capacity:** Typically comes with 4 MB of SPI Flash (some variants offer 8 MB or 16 MB).
- **Memory Mapping:** The ESP32 can map up to 16 MB of external flash directly into the CPU instruction space using cache.
- **PSRAM:** Some modules (like WROVER) add external Pseudo-SRAM for memory-intensive applications.

Wireless Capabilities

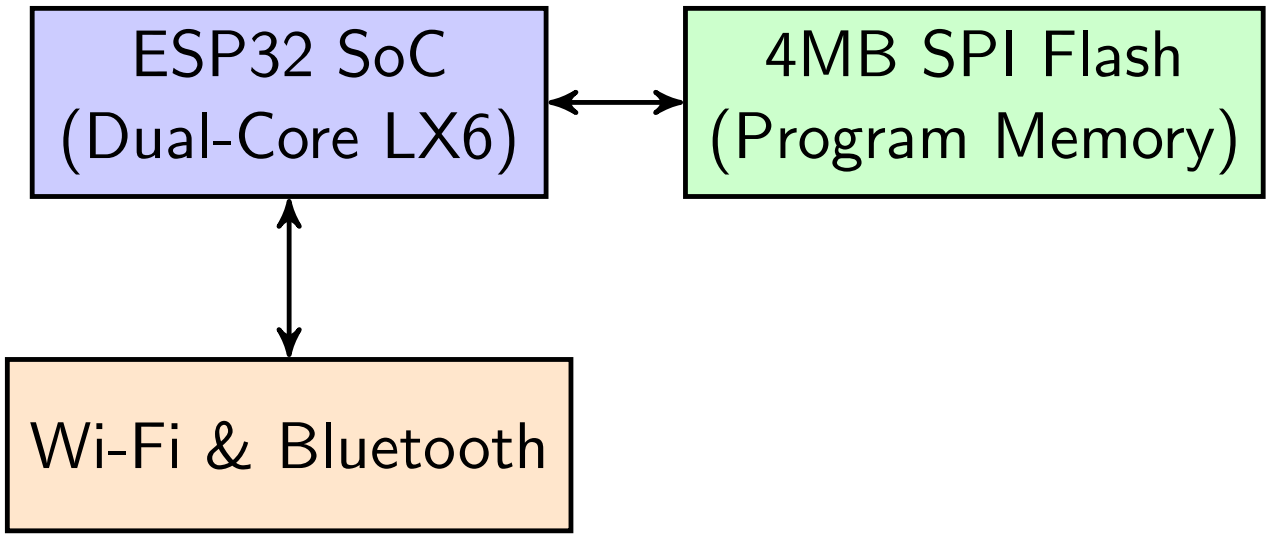
- **Wi-Fi:** 802.11 b/g/n, supporting Station (Client), SoftAP (Access Point), and Station+SoftAP modes.
- **Wi-Fi Security:** WPA/WPA2/WPA3-Personal and Enterprise.
- **Bluetooth:** Dual-mode Bluetooth.
  - **Classic Bluetooth:** (BR/EDR) for audio streaming and legacy devices.
  - **Bluetooth Low Energy (BLE):** For low-power IoT sensors and beacons.
- **Coexistence:** Hardware and software support for Wi-Fi and Bluetooth to share the same antenna seamlessly.

Power Management

- **Active Mode:** Normal execution, highest power consumption (∼160-260 mA with Wi-Fi active).
- **Modem Sleep:** Wi-Fi/Bluetooth disabled, CPU running. Saves significant power.
- **Light Sleep:** CPU paused, RAM retained. Wakes up quickly on interrupts.
- **Deep Sleep:** CPU and main RAM powered down. Only RTC memory and ULP coprocessor run (∼10 µA).
- **Hibernation:** Everything off except RTC timer (∼5 µA).

Summary

- The ESP32-WROOM-32 is a certified module based on the ESP32 SoC.
- It features a powerful dual-core Tensilica Xtensa LX6 processor running up to 240 MHz.
- Memory architecture includes internal SRAM and relies on external SPI Flash for program storage.
- It provides highly integrated Wi-Fi and dual-mode Bluetooth with advanced power management.



Next Lecture Preview

- Detailed ESP32 Pinout and Functionality.
- Understanding Strapping Pins and Boot Modes.
- GPIO capabilities: ADC, DAC, PWM, and Touch Pins.
- Hardware Peripherals overview (I2C, SPI, UART).

Agenda

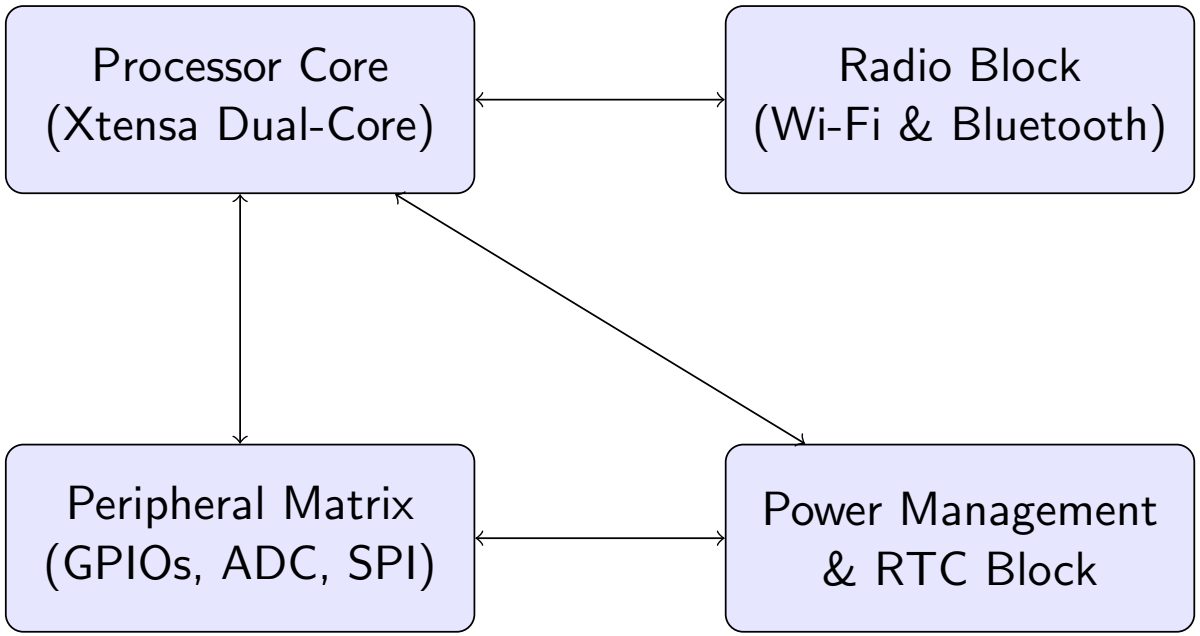
- ESP32 Block Diagram
- CPU and Architecture Details
- Memory Overview (ROM, SRAM, RTC Fast/Slow Memory)
- Wi-Fi and Bluetooth Capabilities
- Pin Configuration and GPIO Mapping
- Specific Pin Functions (ADC, DAC, Touch)

Learning Objectives

- Analyze the ESP32 block diagram and identify major subsystems.
- Understand the CPU architecture and memory allocation of the ESP32.
- Describe the Wi-Fi and Bluetooth protocol support built into the ESP32.
- Map out the GPIO pins and identify specialized pins like ADC, DAC, and capacitive touch pins.

ESP32 Block Diagram

The ESP32 architecture is divided into three main blocks: the Processor Core (CPU, Memory), the Radio block (Wi-Fi, Bluetooth), and the Peripherals block.



- High performance with low power consumption.
- Extensive peripheral matrix connects internal hardware to GPIO pins.
- Power management and RTC block handles ultra-low-power deep sleep modes.

CPU Architecture

The ESP32 is powered by a dual-core CPU designed for performance and efficiency.

- Xtensa Dual-Core 32-bit LX6 Microprocessor.
- Clock frequency adjustable from 80 MHz to 240 MHz.
- Performance up to 600 DMIPS.
- Includes a hardware Floating Point Unit (FPU) for faster math operations.
- Ultra-low-power (ULP) coprocessor handles basic tasks in deep sleep.

Memory Overview

Memory in the ESP32 is segmented for different operational needs.

Memory Type	**Size**	**Purpose**
Internal ROM	448 KB	Booting and core functions
Internal SRAM	520 KB	Data and instructions
RTC Fast Memory	8 KB	Data storage and main CPU during RTC boot
RTC Slow Memory	8 KB	ULP coprocessor access during deep sleep
External Flash	Up to 16 MB	Supports external SPI flash (commonly 4MB)

Wi-Fi Capabilities

The ESP32 features a robust 802.11 b/g/n Wi-Fi subsystem.

- Supports Station (STA), SoftAP, and Station + SoftAP modes.
- Data rates up to 150 Mbps (802.11n).
- Hardware accelerators for cryptographic protocols (AES, SHA-2, RSA, ECC).
- Antenna diversity and WPA/WPA2/WPA3 enterprise security support.

Bluetooth Capabilities

Dual-mode Bluetooth sets the ESP32 apart from its predecessor, the ESP8266.

- Classic Bluetooth (BR/EDR) for audio streaming and legacy devices.
- Bluetooth Low Energy (BLE) for low-power sensor networks and beacons.
- Simultaneous multi-connection support.
- Co-existence interface ensures Wi-Fi and Bluetooth can operate together smoothly sharing the same antenna.

Pin Configuration and GPIO Mapping

The ESP32 has 34 programmable GPIO pins, but some have restrictions.

- Input/Output: Most pins (e.g., GPIO 0-19, 21-23, 25-27, 32-33) support both digital read and write.
- Input Only: GPIOs 34, 35, 36 (VP), and 39 (VN) are strictly input-only and do not have internal pull-up/pull-down resistors.
- Strapping Pins: GPIO 0, 2, 5, 12, and 15 are used during bootup to determine boot mode (Flash vs UART download). Use with caution.

Analog-to-Digital Converters (ADC)

The ESP32 features two 12-bit SAR ADCs.

- ADC1: 8 channels attached to GPIOs 32-39.
- ADC2: 10 channels attached to GPIOs 0, 2, 4, 12-15, 25-27.
- 12-bit resolution means values range from 0 to 4095.
- WARNING: ADC2 cannot be used when Wi-Fi is active, as the Wi-Fi driver uses it. Always use ADC1 for sensors if using Wi-Fi.

DAC and Capacitive Touch Pins

The ESP32 also generates analog signals and reads capacitive touch.

- DAC (Digital-to-Analog Converter): Two 8-bit DAC channels on GPIO 25 and GPIO 26.
- DAC allows generation of true analog voltages (not just PWM) for audio or control signals.
- Capacitive Touch: 10 internal capacitive touch sensors on specific GPIOs (e.g., T0 on GPIO4, T2 on GPIO2).
- Touch pins can wake the ESP32 from deep sleep upon human contact.

Other Peripherals (SPI, I2C, UART)

Standard serial communication protocols are fully supported.

- UART: 3 hardware UART interfaces (UART0 usually for USB-serial programming).
- I2C: 2 I2C bus interfaces, can be mapped to almost any GPIO.
- SPI: 3 SPI interfaces (SPI, HSPI, VSPI). VSPI and HSPI are available for user applications.
- Other built-in hardware includes I2S for high-quality audio, CAN bus, and PWM controllers.

Key Takeaways

- ESP32 relies on a powerful dual-core CPU with a segmented memory structure including RTC memory for deep sleep.
- Provides both Wi-Fi and dual-mode Bluetooth (Classic + BLE) connectivity.
- Care must be taken with GPIOs: noting input-only pins (34-39) and boot strapping pins.
- Features 12-bit ADCs (ADC1 is safe with Wi-Fi, ADC2 is not), two 8-bit DACs, and capacitive touch sensors.

Next Lecture Preview

- Setting up the ESP32 in the Arduino IDE.
- Writing our first ESP32 program (Blink and Serial Print).
- Understanding the build process and flashing the microcontroller.

Agenda

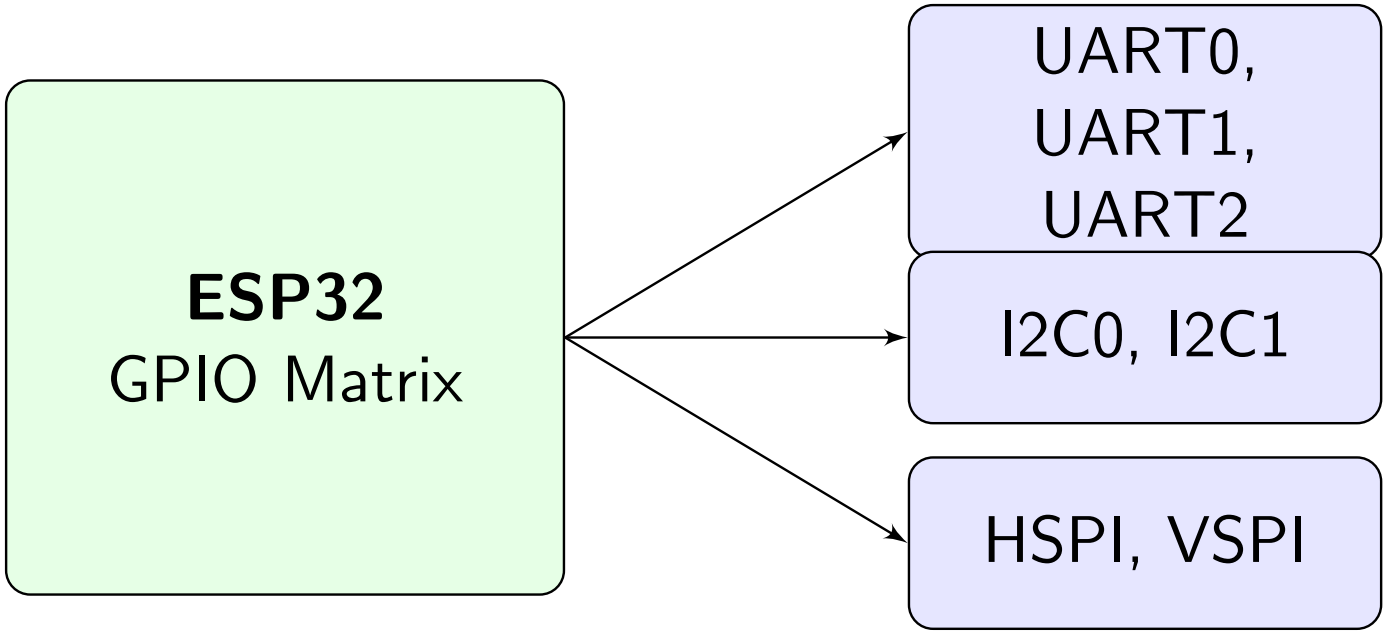
- Overview of Communication Peripherals
- UART Interfaces on ESP32
- I2C Bus Features and Configuration
- SPI Bus Specifications
- ESP32 vs. ESP8266
- ESP32 vs. Arduino UNO (RAM & Architecture)

Learning Objectives

- Identify and configure the UART, I2C, and SPI peripherals on the ESP32.
- Understand the multiplexing capabilities of ESP32 GPIOs for communication.
- Analyze the architectural and memory differences between ESP32, ESP8266, and Arduino UNO.
- Select the appropriate microcontroller based on project requirements (e.g., RAM, processing power, wireless needs).

Communication Peripherals Overview

- **Rich Interface Support:** The ESP32 is designed for complex IoT nodes requiring multiple sensors and modules.
- **UART:** 3 hardware interfaces.
- **I2C:** 2 hardware interfaces.
- **SPI:** 4 hardware interfaces (3 available to users).



UART on ESP32

- **UART Interfaces:** UART0, UART1, and UART2.
- **Speeds:** Supports baud rates up to 5 Mbps.
- **UART0:** Typically used for flashing and logging (connected to USB-to-TTL on dev boards via GPIO 1/3).
- **UART1:** Often overlapping with flash memory pins; requires remapping via GPIO matrix to be usable.
- **UART2:** Free for user applications (default GPIO 16/17 on many boards).
- **Hardware Flow Control:** Supported (RTS/CTS) for reliable high-speed data transfer.

I2C on ESP32 - Deep Dive

- **I2C Interfaces:** Can operate as Master or Slave.
- **Speeds:** Standard mode (100 Kbit/s) and Fast mode (400 Kbit/s). Some implementations support up to 5 MHz, but 400 Kbit/s is standard for stability.
- **Pin Assignment:** Default is usually SDA on GPIO 21 and SCL on GPIO 22, but can be routed to any output-capable GPIO.
- **Multi-Master Support:** Supports multiple masters on the same bus.
- **Use Cases:** Ideal for displaying data on OLEDs (SSD1306) and reading sensors (BME280, MPU6050).

SPI on ESP32 - Deep Dive

- **SPI Buses:** SPI0, SPI1, HSPI, and VSPI.
- **Usability:** SPI0 and SPI1 are internally used to communicate with the integrated flash memory. Do not use them.
- **Available Buses:** HSPI and VSPI are fully available to the user.
- **VSPI Defaults:** MOSI (GPIO 23), MISO (GPIO 19), CLK (GPIO 18), CS (GPIO 5).
- **HSPI Defaults:** MOSI (GPIO 13), MISO (GPIO 12), CLK (GPIO 14), CS (GPIO 15).
- **High Speed:** Can reach clock speeds up to 80 MHz, ideal for TFT screens and SD card modules.

ESP32 vs ESP8266: Hardware Specs

- **Processor:** ESP32 is Dual-Core (160-240 MHz); ESP8266 is Single-Core (80-160 MHz).
- **RAM:** ESP32 has 520 KB SRAM; ESP8266 has ~160 KB (mostly used by WiFi stack).
- **Connectivity:** ESP32 features Wi-Fi + Bluetooth (Classic & BLE); ESP8266 is Wi-Fi only.
- **GPIO Pins:** ESP32 has 34+ usable GPIOs; ESP8266 has ~11 usable GPIOs.
- **Analog Inputs:** ESP32 has 18x 12-bit ADCs; ESP8266 has 1x 10-bit ADC.

ESP32 vs ESP8266: Which one to choose?

When to choose ESP8266:

- Simple, single-sensor IoT projects.
- Strict budget constraints (ESP8266 is slightly cheaper).
- Legacy projects or simple Wi-Fi relays (e.g., Sonoff devices).

When to choose ESP32:

- Requires Bluetooth/BLE.
- Complex processing (SSL/TLS encryption, audio processing, machine learning).
- Needs multiple sensors or high-resolution analog readings.

ESP32 vs Arduino UNO: Architecture

- **Architecture:** ESP32 is a 32-bit Xtensa LX6; Arduino UNO is an 8-bit AVR (ATmega328P).
- **Clock Speed:** ESP32 runs up to 240 MHz; Arduino UNO runs at 16 MHz.
- **Operating Voltage:** ESP32 operates at 3.3V (NOT 5V tolerant); Arduino UNO operates at 5V.
- **Connectivity:** ESP32 has built-in Wi-Fi and Bluetooth; Arduino UNO has none natively.

ESP32 vs Arduino UNO: RAM & Memory

- **SRAM (Random Access Memory):**
  - ESP32: 520 KB
  - Arduino UNO: 2 KB (2048 bytes)
- **Flash Memory (Storage):**
  - ESP32: Typically 4 MB to 16 MB (external SPI flash).
  - Arduino UNO: 32 KB (internal).
- **Implications of RAM Differences:**
  - 2 KB on UNO severely limits dynamic memory allocation, JSON parsing, and handling web requests.
  - 520 KB on ESP32 allows for running RTOS, handling complex secure web sockets, and buffering large data payloads (like images or audio).

Summary of Comparisons

Feature	Arduino UNO	ESP8266	ESP32
Processor	8-bit, 16 MHz	32-bit, 80/160 MHz	32-bit Dual, 240 MHz
RAM	2 KB	$\sim$160 KB	520 KB
Wireless	None	Wi-Fi	Wi-Fi + BLE
Voltage	5V	3.3V	3.3V
Best For	Learning, strict 5V hardware	Basic IoT	Advanced IoT, Edge ML

Key Takeaways

- ESP32 provides highly flexible communication via 3 UARTs, 2 I2Cs, and 2 usable SPIs, enhanced by a versatile GPIO matrix.
- ESP32 represents a massive performance upgrade over the ESP8266, adding a core, BLE, and vastly improved analog features.
- Compared to an Arduino UNO, the ESP32's 32-bit architecture and 520 KB RAM enable modern, secure networking and complex data handling.
- Care must be taken with ESP32's 3.3V logic levels when interfacing with legacy 5V sensors.

Next Lecture Preview

Topic: 3.2 ESP32 Pinout and GPIOs (Part 1)

What we will cover:

- Detailed analysis of the ESP32 Pinout.
- Digital Input and Output operations.
- Pull-up and Pull-down resistors.
- Pins to avoid using during boot.

Basic Terms and Dev Environment (Part 1)

- Unit 3: ESP32 Ecosystem
- Topic: 3.2 Basic Terms and Dev Environment (Part 1)
- Course: IoT and ESP32 (DI05000041)
- Week 7, Lecture 2

Agenda

- Basic Terms: Firmware, Wi-Fi, MAC, SSID, IP
- ESP32 Firmware Concepts
- Introduction to ESP-IDF and Arduino Core
- ESP-IDF vs. Arduino Core Comparison
- Flashing Firmware to ESP32

Learning Objectives

- By the end of this lecture, students will be able to:
- Define key networking and embedded systems terms (Firmware, MAC, IP, SSID).
- Explain the concept of firmware specific to the ESP32 architecture.
- Differentiate between ESP-IDF and Arduino Core development environments.
- Understand the underlying process of compiling and flashing firmware to the ESP32 via UART.

What is Firmware?

- **Definition**: Permanent software programmed into a read-only memory (ROM) or flash memory.
- **Purpose**: Provides low-level control for a device's specific hardware.
- **In IoT**: Bridges the gap between hardware execution and higher-level software applications.
- **Difference from Software**: Tightly coupled to hardware; typically runs directly on the bare metal or a lightweight RTOS.

Networking Terms: MAC Address

- **MAC (Media Access Control) Address**:
- Unique identifier assigned to a network interface controller (NIC).
- Hardware address, typically burnt into ROM by the manufacturer.
- Format: 6 bytes (48 bits), usually represented as Hex (e.g., `24:0A:C4:00:11:22`).
- **ESP32 Context**:
- Has distinct MAC addresses for Wi-Fi Station, Wi-Fi SoftAP, and Bluetooth.
- Base MAC address is stored in eFuse memory.

Networking Terms: IP Address

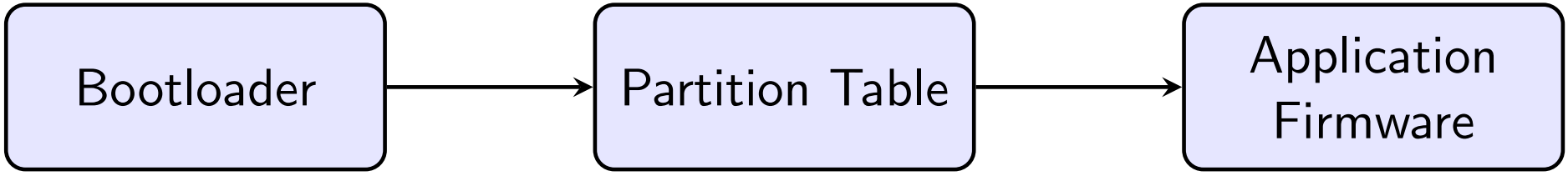
- **IP (Internet Protocol) Address**:
- Logical numeric label assigned to each device connected to a computer network.
- Used for host or network interface identification and location addressing.
- **IPv4**: 32-bit address (e.g., `192.168.1.100`).
- **IPv6**: 128-bit address (e.g., `fe80::240a:c4ff:fe00:1122`).
- **Dynamic vs. Static**: Assigned by a DHCP server (Dynamic) or manually configured (Static).

Wireless Terms: Wi-Fi and SSID

- **Wi-Fi**: A family of wireless network protocols based on the IEEE 802.11 family of standards.
- ESP32 supports 802.11 b/g/n (2.4 GHz).
- **SSID (Service Set Identifier)**:
- The public name of a wireless network.
- Maximum 32 alphanumeric characters.
- Example: `MyHomeNetwork`, `GTU_Campus_WiFi`.
- **BSSID**: The MAC address of the wireless access point offering the SSID.

ESP32 Firmware Concept

- **Bootloader**: First program that runs, validates, and loads the application.
- **Partition Table**: Divides flash memory into distinct areas (NVS, PHY init data, Factory app, OTA apps).
- **Application Firmware**: The actual compiled C/C++ code, running on top of FreeRTOS.
- **Binary Image (`.bin`)**: The compiled firmware file that gets flashed onto the ESP32's SPI Flash.



Arduino Core for ESP32

- **What is it?**: A wrapper that allows programming the ESP32 using the familiar Arduino API (`setup()`, `loop()`).
- **Pros**:
- Huge ecosystem of libraries.
- Extremely gentle learning curve.
- Fast prototyping.
- **Cons**:
- Adds overhead, less efficient.
- Abstracts away fine-grained hardware control and multi-core features.
- Built on top of pre-compiled ESP-IDF components.

ESP-IDF (IoT Development Framework)

- **What is it?**: Espressif's official development framework for ESP32. Based on FreeRTOS.
- **Pros**:
- Full access to all hardware features (multi-core, ULP, cryptography).
- Highly optimized and configurable via `menuconfig`.
- Professional-grade toolchain.
- **Cons**:
- Steeper learning curve.
- Requires deep understanding of C and FreeRTOS.
- Less 'plug-and-play' third-party libraries compared to Arduino.

ESP-IDF vs Arduino Core: A Comparison

Feature	Arduino Core	ESP-IDF
Target User	Beginners, Hobbyists	Professionals, Enterprise
Language	C++ (Arduino flavor)	Standard C / C++
OS Base	FreeRTOS (Hidden)	FreeRTOS (Explicit)
Hardware Control	Limited (Abstracted)	Complete
Build System	Arduino Builder/PlatformIO	CMake, Ninja
Configuration	Static	Highly Customizable (`menuconfig`)

Flashing Firmware to ESP32

- **Process**: Transferring the compiled `.bin` file from PC to ESP32 Flash memory.
- **Hardware Interface**: Uses USB-to-UART bridge (CP2102, CH340) communicating via UART0 (TX/RX).
- **Boot Modes**: ESP32 has GPIO0 used as a strapping pin.
- GPIO0 LOW during reset = Download Boot (Flashing) Mode.
- GPIO0 HIGH during reset = SPI Boot (Normal Run) Mode.
- **Tools**: `esptool.py` is the official Python utility used under the hood by Arduino IDE and ESP-IDF.

Key Takeaways

- **MAC and IP**: MAC is physical and static; IP is logical and network-specific.
- **Firmware**: Structured software in flash memory (Bootloader + Partition + App).
- **Arduino vs IDF**: Arduino offers simplicity; ESP-IDF offers comprehensive control and performance.
- **Flashing**: Accomplished via UART using `esptool.py`, controlled by the state of GPIO0 during reset.

Next Lecture Preview

- **Topic: 3.3 Basic Terms and Dev Environment (Part 2)**
- Deep dive into Setting up the Toolchain.
- Installing ESP-IDF and necessary drivers.
- Setting up VS Code and PlatformIO.
- Writing and flashing the first 'Hello World' program.

Agenda

- Introduction to the Arduino IDE
- Why use Arduino IDE for ESP32?
- Downloading and Installing the IDE
- Configuring the ESP32 Board Manager URL
- Installing the ESP32 Package
- USB-to-UART Bridge Drivers (CP2102/CH340)
- Selecting the Board and COM Port

Learning Objectives

- Understand the role of the Arduino IDE in IoT development.
- Successfully install and configure the Arduino IDE for ESP32.
- Install and verify USB-to-UART drivers (like CP2102).
- Correctly select the ESP32 board and communication port.

Introduction to Arduino IDE

- An open-source integrated development environment (IDE).
- Designed to make coding for microcontrollers accessible.
- Uses a simplified version of C/C++.
- Cross-platform: Available for Windows, macOS, and Linux.

Why Arduino IDE for ESP32?

- **Huge Community:** Massive repository of libraries and examples.
- **Familiarity:** If you know Arduino, you can program ESP32.
- **C++ Wrapper:** Espressif provides an Arduino core that wraps the native ESP-IDF.
- **Simplicity:** Hides complex toolchain setups required by raw ESP-IDF.

Downloading & Installing Arduino IDE

- Download from the official site: arduino.cc/en/software.
- Recommend using **Arduino IDE 2.x** for modern features (autocomplete, debugger).
- Run the installer and follow the default prompts.
- Ensure USB drivers are checked during Windows installation.

Configuring ESP32 Board Manager URL

- By default, Arduino IDE only supports standard Arduino boards.
- To add ESP32, we must provide an Additional Board Manager URL.
- **Path:** File → Preferences → Additional Boards Manager URLs.
- **URL:**

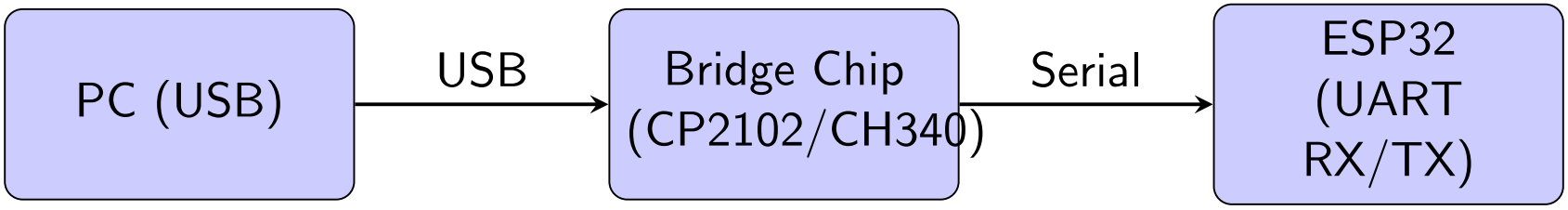
```
https://dl.espressif.com/dl/package_esp32_index.
    json
```

Installing the ESP32 Board Package

- Open **Boards Manager** (Tools → Board → Boards Manager).
- Search for `esp32`.
- Look for the package named **esp32 by Espressif Systems**.
- Click **Install** (this may take several minutes to download the toolchain).

Understanding USB-to-UART Bridges

- ESP32 chips don't typically have native USB (except newer ones like ESP32-S2/S3).
- Development boards use a **USB-to-UART bridge** chip to communicate with the PC.
- Common chips: **Silicon Labs CP2102** and **WCH CH340**.
- These chips convert PC USB signals into Serial (TX/RX) signals for the ESP32.



Installing USB Drivers (CP2102/CH340)

- If your board isn't recognized, you need drivers.
- **CP210x:** Download 'CP210x USB to UART Bridge VCP Drivers' from Silicon Labs.
- **CH340:** Download CH341SER drivers from WCH.
- Extract, run the installer, and restart the IDE.

Chip	**Manufacturer**	**Driver Name**
CP2102	Silicon Labs	CP210x VCP Drivers
CH340	WCH	CH341SER

Selecting the ESP32 Board

- Go to **Tools** → **Board** → **esp32**.
- Select your specific board model.
- Common choice for generic 30-pin/38-pin boards: **DOIT ESP32 DEVKIT V1**.
- Selecting the wrong board may cause pin mapping issues or upload failures.

Selecting the COM Port

- Go to **Tools** $\rightarrow$ **Port**.
- Select the COM port that appeared when you plugged in the board.
- **Windows:** `COM3`, `COM4`, etc. (Check Device Manager if unsure).
- **Mac/Linux:** `/dev/cu.SLAB_USBtoUART` or `/dev/ttyUSB0`.

Key Takeaways

- Arduino IDE is the most accessible environment for programming the ESP32.
- The ESP32 board manager URL must be added to Preferences to install the ESP32 core.
- USB-to-UART drivers (CP2102/CH340) are critical for PC communication.
- Board model and COM port must be correctly selected before programming.

Next Lecture Preview

- Topic: 3.3 ESP32 Pinout and GPIO Configuration
- Understanding digital and analog pins.
- Power pins and safety limits.
- Special function pins (I2C, SPI, UART, ADC).

Agenda

- Learning Objectives
- Introduction to ESP32 Programming Structure
- The setup() Function
- The loop() Function
- Writing a Basic Sketch (Blink)
- Creating and Saving Sketches
- Compiling and Verifying Code
- Uploading Code to the ESP32
- Key Takeaways
- Next Lecture Preview

Learning Objectives

- Understand the foundational structure of an Arduino IDE program (sketch).
- Differentiate between initialization code and execution code.
- Learn how to create, save, and organize ESP32 sketches.
- Master the compile (verify) and upload process in the Arduino IDE.

Introduction to ESP32 Programming Structure

- Arduino IDE programs are called **Sketches**.
- Sketches are essentially C++ programs, but the IDE abstracts away complex C++ boilerplate (like `int main()`).
- Every standard sketch must contain at least two core functions:
- `setup()`: Runs once at boot.
- `loop()`: Runs continuously after setup completes.

Key Takeaways:

- Sketches are written in C++.
- The main() function is hidden by the Arduino core.
- setup() and loop() are mandatory.

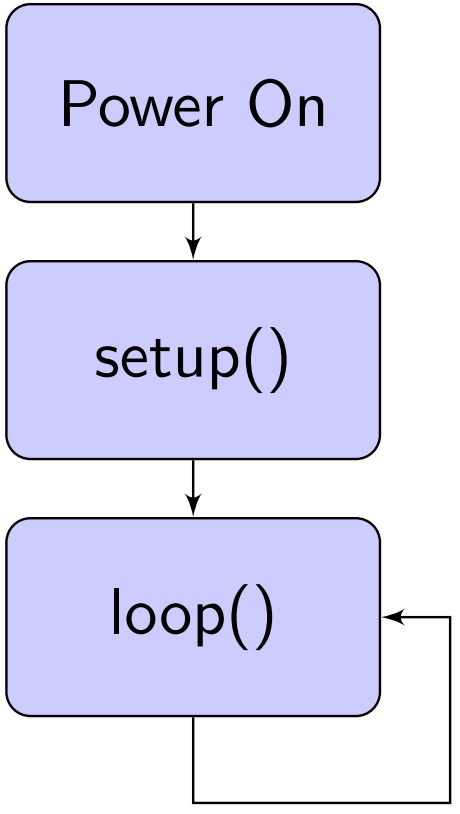
Basic Structure of an ESP32 Program

- This is the bare minimum code required for a valid sketch.
- `void` indicates that these functions do not return any value.
- Comments starting with `//` are ignored by the compiler.

```
void setup() {
  // put your setup code here, to run once:

}

void loop() {
  // put your main code here, to run repeatedly:

}
```

The setup() Function

- **Purpose:** Initialization and configuration.
- **Execution:** Called exactly once when the ESP32 is powered on or reset.
- **Common Tasks:**
  - Initializing variables and hardware states.
  - Configuring pin modes (e.g., `pinMode(LED_PIN, OUTPUT)`).
  - Starting serial communication (e.g., `Serial.begin(115200)`).
  - Connecting to Wi-Fi networks.



The loop() Function

- **Purpose:** Core application logic.
- **Execution:** Called consecutively in an infinite loop immediately after `setup()` finishes.
- **Common Tasks:**
  - Reading sensor data (e.g., `digitalRead()`, `analogRead()`).
  - Processing logic and making decisions.
  - Controlling outputs (e.g., `digitalWrite()`, `ledcWrite()`).
  - Handling network requests.

Example: The Classic Blink Sketch

- Declares a constant for the LED pin.
- `setup()` configures pin 2 as an `OUTPUT`.
- `loop()` toggles the pin state with a 1000ms delay between states.

```
const int ledPin = 2; // Onboard LED for most ESP32s

void setup() {
  pinMode(ledPin, OUTPUT); // Configure pin as output
}

void loop() {
  digitalWrite(ledPin, HIGH); // Turn LED on
  delay(1000);                // Wait for 1 second (1000ms)
  digitalWrite(ledPin, LOW);  // Turn LED off
  delay(1000);                // Wait for 1 second
}
```

Creating a New Sketch

- Open the Arduino IDE.
- Go to **File** $\rightarrow$ **New Sketch** (or use `Ctrl+N`).
- A new window opens with an empty `setup()` and `loop()` structure.
- Go to **File** $\rightarrow$ **Save As...** to save your project.
- The IDE automatically creates a folder with the same name as your `.ino` sketch file.

Compiling / Verifying Code

- **What is Compiling?**
  - Translating human-readable C++ code into machine code (binary/hex) that the ESP32 can execute.
- **How to Verify:**
  - Click the **Checkmark** (Verify) icon in the top left corner.
  - Or go to **Sketch** $\rightarrow$ **Verify/Compile** (`Ctrl+R`).
- **What happens:**
  - Checks for syntax errors (missing semicolons, undefined variables).
  - Builds the core ESP32 libraries alongside your code.
  - Reports memory usage in the output console.

Uploading Code to the ESP32

- **Prerequisites:** Board selected (e.g., 'DOIT ESP32 DEVKIT V1') and COM Port selected.
- **How to Upload:**
  - Click the **Right Arrow** (Upload) icon.
  - Or go to **Sketch** $\rightarrow$ **Upload** (`Ctrl+U`).
- **The Process:**
  - Code is compiled (verified) first.
  - IDE opens a serial connection to the ESP32 via the USB-to-UART bridge.
  - The binary file is flashed into the ESP32's flash memory.
  - 'Hard resetting via RTS pin...' indicates success.

Key Takeaways:

- Some ESP32 boards require holding the 'BOOT' button during the 'Connecting...' phase to enter flash mode.

The Output Console

- Located at the bottom of the Arduino IDE window.
- Displays real-time feedback during compilation and uploading.
- **Key Information Displayed:**
  - **Errors:** Syntax issues, missing libraries.
  - **Warnings:** Potential issues that won't stop compilation.
  - **Memory Stats:** RAM and Flash memory usage of your compiled sketch.
  - **Upload Progress:** Shows percentage of binary written to flash.

Key Takeaways

- Arduino sketches run on a C++ framework hiding the main loop complexity.
- `setup()` is for one-time initialization (pin modes, serial config).
- `loop()` is for continuous execution of the main program logic.
- Verifying checks syntax and compiles the code.
- Uploading flashes the compiled binary to the ESP32 memory.

Next Lecture Preview

- **Topic:** Program Development using Arduino IDE (Part 2)
- **Subtopics:**
  - Basic debugging techniques.
  - Using the Serial Monitor.
  - `Serial.print()` and `Serial.println()`.
  - Reading data from the ESP32.

Program Development (Part 2)

- Course: Internet of Things (DI05000041)
- Unit 3: ESP32-WROOM-32 Architecture and Programming Fundamentals
- Lecture 3.7: Program Development using Arduino IDE (Part 2)
- Topics: Variables, data types, constants, operators, and control loops

Agenda

- Variables and Data Types
- Variable Scope
- Built-in Constants
- Operators (Arithmetic, Relational, Logical)
- Conditional Statements (if/else, switch)
- Loops (for, while, do-while)
- Practical Arduino Code Examples

Learning Objectives

- Declare and initialize variables using appropriate data types in Arduino C++.
- Utilize Arduino's built-in constants for pin states and modes.
- Apply operators for mathematical and logical evaluations.
- Control program flow using if/else, switch case, and loops (for, while).

Variables and Variable Scope

- A variable is a named space in memory used to store a value.
- Syntax: `type variableName = value;` (e.g., `int ledPin = 13;`)
- Global variables: Declared outside of setup() and loop(), accessible anywhere.
- Local variables: Declared inside a function (e.g., inside loop()), accessible only within that function.

```
int globalVar = 10; // Global scope

void setup() {
  int localVar = 5; // Local scope
}
```

Arduino Data Types

- Choosing the right data type is crucial for memory management.

Type	Size / Value	Description
`int`	16-bit	Integer numbers (-32,768 to 32,767)
`float`	32-bit	Floating-point numbers for decimals (e.g., 3.14)
`boolean`	8-bit	True (1) or False (0)
`char`	8-bit	Single character (e.g., 'A')
`String`	Object	Sequence of characters (memory-heavy)
`byte`	8-bit	Unsigned number (0 to 255), efficient for pin numbers

Built-in Constants

- Pin Levels: HIGH (1, 5V/3.3V) and LOW (0, 0V).
- Pin Modes: INPUT (read signal), OUTPUT (send signal), INPUT_PULLUP (enable internal pull-up resistor).
- Built-in LED: LED_BUILTIN (usually maps to pin 13 on UNO).
- Constants make code readable without memorizing specific values.

```
pinMode(LED_BUILTIN, OUTPUT);
digitalWrite(LED_BUILTIN, HIGH);
```

Operators (Arithmetic & Relational)

- Arithmetic Operators: +, -, *, /, % (modulo).
- Assignment: =, +=, -=, *=, /= (e.g., x += 5 is x = x + 5).
- Relational Operators: == (equal), != (not equal), ¡, ¿, ¡=, ¿=.
- Be careful not to confuse assignment '=' with equality '=='!

```
int a = 10;
int b = 3;
int rem = a % b; // rem = 1
bool isGreater = (a > b); // true
```

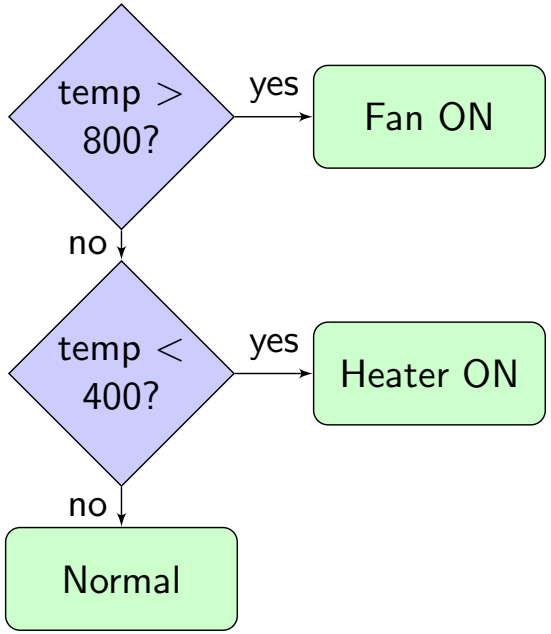
Logical and Bitwise Operators

- Logical AND (&&): True if both operands are true.
- Logical OR (——): True if at least one operand is true.
- Logical NOT (!): Inverts the boolean state.
- Bitwise Operators: & (AND), — (OR), ˆ (XOR), ¡¡ (left shift), ¿¿ (right shift) for low-level register manipulation.

```
if (sensorValue > 100 && buttonPressed == true) {
  // Execute only if both conditions are met
}
```

Conditional Statements (if/else)

- Execute code conditionally based on evaluations.
- Chain conditions with 'else if'. 'else' is a catch-all.




```
int temp = analogRead(A0);
if (temp > 800) { digitalWrite(FAN_PIN, HIGH); }
else if (temp < 400) { digitalWrite(HEATER_PIN, HIGH); }
else { /* Temperature is normal */ }
```

Conditional Statements (switch)

- Used for evaluating a single variable against multiple exact values.
- Often cleaner than long chains of if-else statements.
- Must use the 'break' statement to exit the block, otherwise execution 'falls through' to the next case.
- The 'default' case handles anything not explicitly matched.

```
switch(sensorState) {
  case 1: turnOnRed(); break;
  case 2: turnOnGreen(); break;
  default: turnOffAll();
}
```

Loops (for)

- Used when the number of iterations is known in advance.
- Syntax: `for (initialization; condition; increment) { code }`
- Great for arrays, stepping through PWM duty cycles, or fading LEDs.

```
// Fade an LED using PWM
for (int brightness = 0; brightness <= 255; brightness += 5)
      {
  analogWrite(LED_PIN, brightness);
  delay(30);
}
```

Loops (while and do-while)

- `while` loop: Evaluates condition before executing. May execute 0 times if condition is initially false.
- `do-while` loop: Evaluates condition after executing. Guaranteed to execute at least once.
- Warning: Ensure the condition eventually becomes false to avoid infinite loops.

```
int count = 0;
while (count < 5) {
  Serial.println(count);
  count++;
}

do {
  buttonState = digitalRead(BTN_PIN);
} while (buttonState == HIGH);
```

Summary

- Variables store data; scope determines where they can be accessed.
- Choosing memory-efficient data types like 'byte' is crucial on Arduino.
- Built-in constants (HIGH, LOW, INPUT) simplify hardware interaction.
- Control structures (if, switch) and loops (for, while) form the logic of any smart IoT device.

Next Lecture Preview

- Creating custom functions (void and return types).
- Passing arguments to functions.
- Including and using external Arduino Libraries.
- Writing cleaner, modular Arduino code.

Agenda

- Built-in Library Functions
- Timer and Delay Functions (`delay`, `millis`)
- Blocking vs. Non-blocking Code
- Creating User-Defined Functions
- Examples and Best Practices

Learning Objectives

By the end of this lecture, you will be able to:

- Identify and use common Arduino built-in library functions.
- Implement precise timing using `delay()`, `delayMicroseconds()`, and `millis()`.
- Write non-blocking code for multitasking on ESP32/Arduino.
- Design and implement user-defined functions to improve code reusability.

Built-in Library Functions

- Arduino core provides numerous pre-defined functions to simplify hardware interaction and data manipulation.
- These functions abstract the underlying hardware registers, making code portable across different microcontrollers like AVR, ESP8266, and ESP32.

Category	**Common Functions**
Digital I/O	`pinMode()`, `digitalWrite()`, `digitalRead()`
Analog I/O	`analogRead()`, `analogWrite()`
Advanced I/O	`tone()`, `noTone()`, `shiftOut()`, `pulseIn()`
Communication	`Serial.begin()`, `Serial.print()`

Math and Characters Functions

Arduino also includes standard C++ math and character manipulation functions:

- **Math Functions:** `min(x, y)`, `max(x, y)`, `abs(x)`, `constrain(x, a, b)`, `map(value, fromLow, ...)`, `pow()`, `sqrt()`
- **Trigonometry:** `sin(rad)`, `cos(rad)`, `tan(rad)`
- **Random Numbers:** `randomSeed(seed)`, `random(min, max)`
- **Characters:** `isAlpha()`, `isDigit()`, `isSpace()`, `toUpperCase()`

Note: The `map()` function is extremely useful for scaling analog sensor readings to desired output ranges (e.g., 0-4095 from an ESP32 ADC to 0-100%).

Timer/Delay Functions: delay()

`delay(ms)` pauses the program for the amount of time (in milliseconds) specified as parameter.

```
void loop() {
  digitalWrite(LED_BUILTIN, HIGH);
  delay(1000); // Wait for 1 second
  digitalWrite(LED_BUILTIN, LOW);
  delay(1000); // Wait for 1 second
}
```

- **Pros:** Extremely simple to use for basic blinking or sequencing.
- **Cons:** It is a **blocking** function. While `delay()` is running, the microcontroller cannot process other inputs, update outputs, or perform calculations (except for interrupts).

Timer/Delay Functions: delayMicroseconds()

- `delayMicroseconds(us)` pauses the program for the specified number of microseconds.
- 1 millisecond = 1,000 microseconds.
- Useful for precise timing required by specific hardware protocols (e.g., generating a very short pulse for an ultrasonic sensor).

```
digitalWrite(TRIGGER_PIN, LOW);
delayMicroseconds(2);
digitalWrite(TRIGGER_PIN, HIGH);
delayMicroseconds(10); // Send 10us pulse
digitalWrite(TRIGGER_PIN, LOW);
```

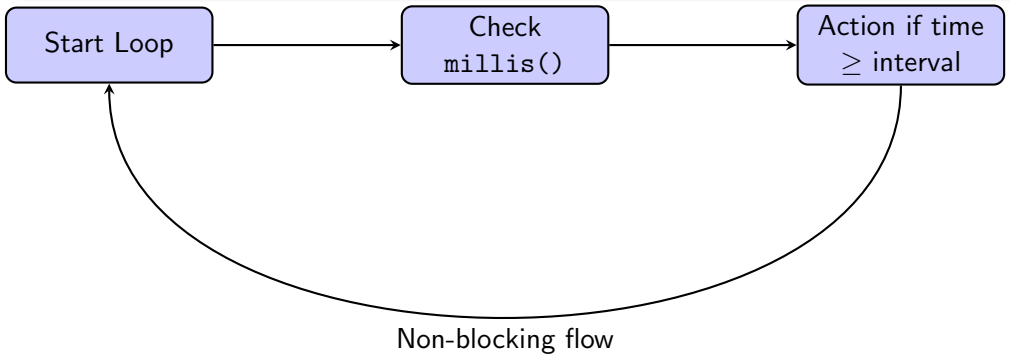
Note: Like `delay()`, it is also a blocking function.

Timer Functions: millis()

- `millis()` returns the number of milliseconds passed since the Arduino board began running the current program.
- Returns an `unsigned long` data type.
- Overflows (goes back to zero) after approximately 50 days.
- `micros()` is similar but returns microseconds (overflows after ~70 minutes).

Key Concept

Instead of pausing the program, `millis()` allows you to check the "current time" on the microcontroller's internal clock and make decisions based on elapsed time.



millis() vs delay(): Non-blocking Code

To write non-blocking code, you track the *previous time* an event happened and compare it to the *current time*. **Standard Pattern:**

```
unsigned long previousMillis = 0;
const long interval = 1000; // Interval in ms
void loop() {
  unsigned long currentMillis = millis();
  if (currentMillis - previousMillis >= interval) {
    previousMillis = currentMillis; // Reset timer
    // Do something here (e.g., toggle LED)
  }
  // Other code here runs continuously!
}
```

Real Code Example: Multitasking

```
unsigned long prevLedTime = 0;
unsigned long prevPrintTime = 0;
bool ledState = LOW;

void loop() {
  unsigned long currentMillis = millis();

  // Task 1: Blink LED every 500ms
  if (currentMillis - prevLedTime >= 500) {
    prevLedTime = currentMillis;
    ledState = !ledState;
    digitalWrite(LED_PIN, ledState);
  }

  // Task 2: Print to Serial every 2000ms
  if (currentMillis - prevPrintTime >= 2000) {
    prevPrintTime = currentMillis;
    Serial.println("System␣Running...");
  }
} // Loop never stops!
```

User-Defined Functions

As programs grow, writing everything in `loop()` makes code unreadable and hard to debug. User-defined functions allow you to group code into reusable blocks. **Syntax:**

```
returnType functionName(parameter1, parameter2) {
  // Function Body
  // Code to be executed
  return value; // If returnType is not void
}
```

- **returnType:** Data type of the value returned (`void`, `int`, `float`, etc.).
- **functionName:** A descriptive name for the function.
- **parameters:** Variables passed into the function (optional).

User-Defined Functions: Example

```
// Function declaration and definition
float calculateTemperature(int adcValue) {
  float voltage = adcValue * (3.3 / 4095.0); // ESP32 ADC
  float temperature = (voltage - 0.5) * 100.0; // TMP36
      formula
  return temperature;
}

void loop() {
  int rawADC = analogRead(34);

  // Calling the function
  float tempC = calculateTemperature(rawADC);

  Serial.print("Temp: ");
  Serial.println(tempC);
  delay(1000);
}
```

This keeps complex mathematical conversions out of the main logic, enhancing readability.

Summary (Key Takeaways)

1. **Built-in Functions:** Arduino provides an extensive standard library for I/O, math, and character manipulation.
2. **Blocking `delay()`:** Halts all program execution. Avoid in complex IoT projects.
3. **Non-blocking `millis()`:** Uses internal timers to track time, allowing for concurrent multitasking.
4. **User-Defined Functions:** Essential for organizing code into modular, reusable, and readable components.
5. **Good Practices:** Keep `loop()` clean and delegate specific tasks to specialized functions.

Next Lecture Preview

In the next lecture, we will explore:

- **Program Development (Part 4)**
- String formatting and manipulations.
- Understanding Arrays in C++.
- Using Arrays for sensor data buffering and averaging.

3.4 Digital, Analog, PWM and Serial Ops (Part 1)

- Focus: GPIO Configuration, Digital I/O, LED & Switch Interfacing

Agenda

- Introduction to GPIOs
- ESP32 GPIO Capabilities
- GPIO Configuration (`pinMode`)
- Digital Output (`digitalWrite`) and LED Interfacing
- Digital Input (`digitalRead`)
- Internal Pull-up/Pull-down Resistors
- Switch Interfacing
- Combining Input and Output

Learning Objectives

By the end of this lecture, you will be able to:

- Explain the purpose of GPIO pins on a microcontroller.
- Configure ESP32 pins as inputs or outputs using `pinMode()`.
- Control external devices like LEDs using `digitalWrite()`.
- Read digital states from switches using `digitalRead()`.
- Utilize internal pull-up resistors to ensure stable input readings.

Introduction to GPIO

General Purpose Input/Output (GPIO):

- Uncommitted digital signal pins on an integrated circuit or electronic circuit board.
- Can be controlled by the user at runtime.
- **Digital Input**: Reads discrete states (HIGH/1 or LOW/0).
- **Digital Output**: Sets discrete states (HIGH/3.3V or LOW/0V).
- Serves as the primary bridge between the microcontroller brain and the physical world.

ESP32 GPIO Capabilities

- **Voltage Level**: ESP32 operates at **3.3V** logic level (NOT 5V tolerant!).
- **Pin Count**: Most development boards expose around 30+ GPIO pins.
- **Current Limits**: Maximum current drawn per pin is typically around 12mA to 40mA (depends on specific pin and config; safer to stay under 20mA).
- **Multiplexing**: Most pins have multiple functions (e.g., a pin can be a simple GPIO, an ADC input, or a UART TX pin).

GPIO Configuration: `pinMode()`

Before using a pin, you must tell the microcontroller how it should behave.

Syntax:

```
pinMode(pin, mode);
```

- **pin**: the GPIO number (e.g., 2, 4, 15).
- **mode**:
  - `INPUT`: Pin reads external signals.
  - `OUTPUT`: Pin drives external signals.
  - `INPUT_PULLUP`: Input with internal resistor to 3.3V.
  - `INPUT_PULLDOWN`: Input with internal resistor to GND.

Digital Output: `digitalWrite()`

Used to set an OUTPUT pin to a HIGH or LOW state.

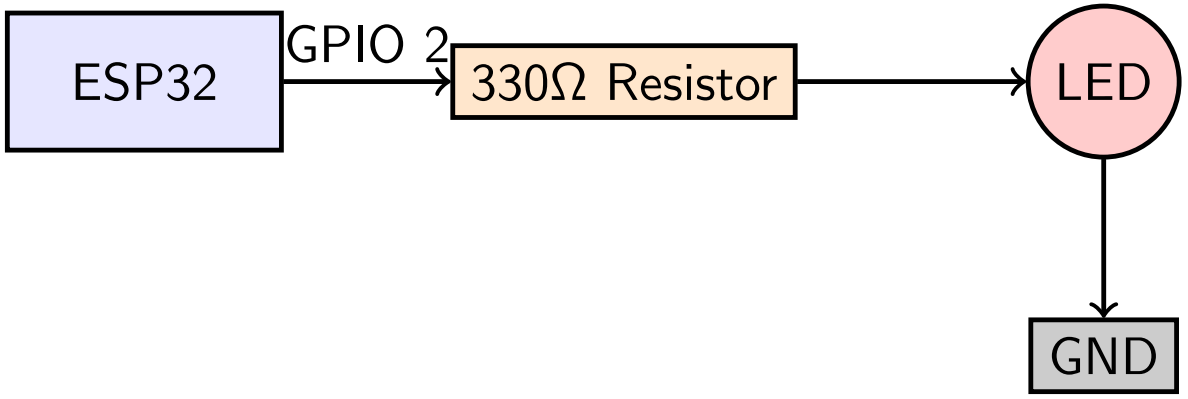
Syntax:

```
digitalWrite(pin, value);
```

- **pin**: the GPIO number.
- **value**:
  - `HIGH`: Sets the pin to 3.3V.
  - `LOW`: Sets the pin to 0V (Ground).

Interfacing an LED

Hardware Setup: Connect GPIO → Resistor (e.g., 330Ω) → LED Anode (+) → LED Cathode (-) → GND.




```
const int ledPin = 2; // Onboard LED
void setup() {
  pinMode(ledPin, OUTPUT);
}
void loop() {
  digitalWrite(ledPin, HIGH); // Turn LED on
  delay(1000);                // Wait 1 second
  digitalWrite(ledPin, LOW);  // Turn LED off
  delay(1000);
}
```

Digital Input: `digitalRead()`

Used to read the state of an INPUT pin.

Syntax:

```
int state = digitalRead(pin);
```

- Returns `HIGH` (1) if voltage is near 3.3V.
- Returns `LOW` (0) if voltage is near 0V.

Floating Pins & Pull Resistors

The Floating Pin Problem:

- If a pin is set as INPUT but not connected to anything, its voltage is undefined. It will pick up electrical noise and read random HIGH/LOW values.

The Solution:

- **Pull-up Resistor**: Connects pin to 3.3V (default HIGH). Switch connects to GND.
- **Pull-down Resistor**: Connects pin to GND (default LOW). Switch connects to 3.3V.
- **Internal Resistors**: ESP32 has built-in pull resistors! Use `INPUT_PULLUP` or `INPUT_PULLDOWN`.

Switch Interfacing (using INPUT_PULLUP)

Hardware Setup: Connect Switch between GPIO 4 and GND. No external resistor needed!

Code Snippet:

```
const int buttonPin = 4;
void setup() {
  Serial.begin(115200);
  // Enable internal pull-up
  pinMode(buttonPin, INPUT_PULLUP);
}
void loop() {
  int buttonState = digitalRead(buttonPin);
  // Note: LOW means pressed because of PULLUP
  if (buttonState == LOW) {
    Serial.println("Button is Pressed!");
  }
  delay(100);
}
```

Combining Input and Output

Goal: Press button to turn on LED.

```
const int ledPin = 2;
const int buttonPin = 4;

void setup() {
  pinMode(ledPin, OUTPUT);
  pinMode(buttonPin, INPUT_PULLUP);
}

void loop() {
  int buttonState = digitalRead(buttonPin);

  if (buttonState == LOW) { // Button pressed
    digitalWrite(ledPin, HIGH);
  } else {                  // Button released
    digitalWrite(ledPin, LOW);
  }
}
```

Key Takeaways

- `pinMode()` is mandatory for setting pin direction (`OUTPUT`, `INPUT`, `INPUT_PULLUP`).
- `digitalWrite()` controls output state (3.3V or 0V).
- `digitalRead()` senses input state.
- ESP32 relies on 3.3V logic.
- Floating inputs cause erratic behavior; always use pull-up or pull-down resistors (internal or external).

Next Lecture Preview

Digital, Analog, PWM and Serial Ops (Part 2)
We will explore:

- **Analog Inputs**: Reading variable voltages using the Analog-to-Digital Converter (ADC).
- **Analog Outputs**: Simulating variable voltage using Pulse Width Modulation (PWM) with `ledc`.
- Fading an LED instead of just blinking it.

3.4 Digital, Analog, PWM (Part 2)

- Analog data reading (ADC)
- POT interfacing
- PWM generation
- Duty cycle control (LED brightness)

Agenda

- Understanding Analog Signals vs Digital Signals
- ESP32 ADC (Analog to Digital Converter) Features
- Interfacing a Potentiometer (POT) with ESP32
- Writing Code for Analog Read
- Understanding Pulse Width Modulation (PWM)
- ESP32 LEDC PWM API
- Code for LED Brightness Control

Learning Objectives

- Understand the difference between discrete digital signals and continuous analog signals.
- Learn how the ESP32's 12-bit ADC converts analog voltage into digital values.
- Interface a potentiometer and write code to read its value.
- Comprehend the concept of Duty Cycle and Frequency in PWM.
- Implement PWM using the ESP32's LEDC peripheral to fade an LED.

Understanding Analog Signals

ESP32 ADC (Analog to Digital Converter)

- Resolution: ESP32 features two 12-bit ADCs (ADC1 and ADC2).
- bit Value: Means the digital output ranges from 0 to 4095 ($2^{12} - 1$).
- Voltage Range: Default attenuation maps 0V to 0 and 3.3V to 4095.
- ADC1: 8 channels (GPIOs 32-39). Highly recommended for use.
- ADC2: 10 channels. Shared with Wi-Fi; cannot be used reliably when Wi-Fi is active.

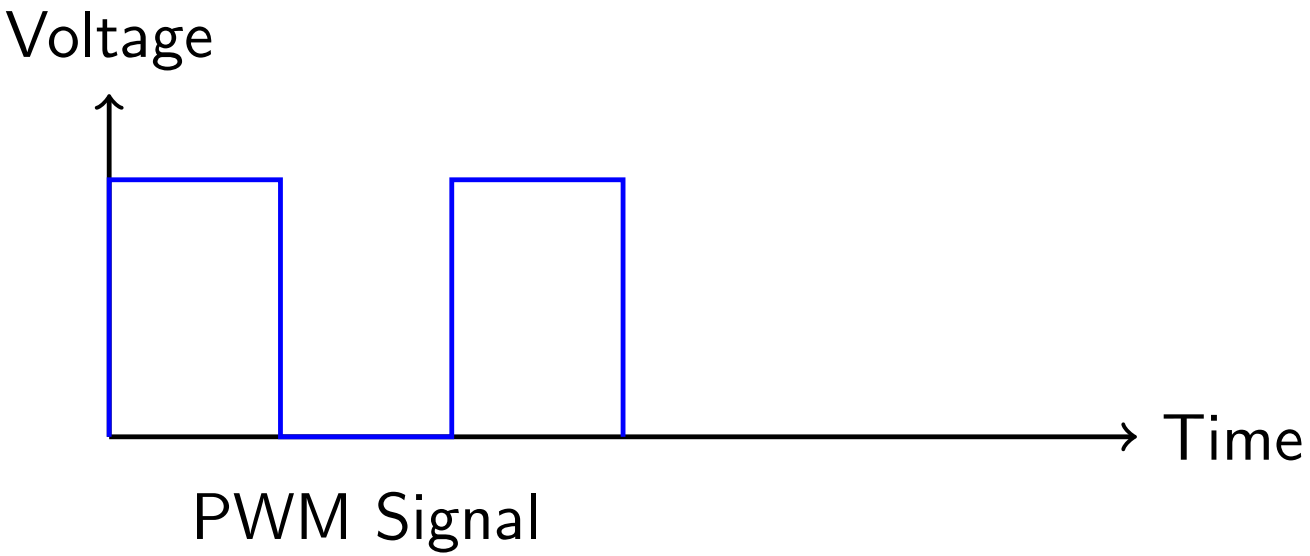
Interfacing a Potentiometer (POT)

- A Potentiometer is a three-terminal variable resistor acting as a voltage divider.
- Wiring to ESP32:
- Outer Pin 1: VCC (3.3V)
- Outer Pin 2: GND (0V)
- Center Pin (Wiper): ADC Pin (e.g., GPIO 34)
- Turning the knob changes the resistance, which changes the voltage at the center pin between 0V and 3.3V.

Code: Reading Analog Data (analogRead)

```
const int potPin = 34; // ADC1 channel
int potValue = 0;
void setup() {
Serial.begin(115200);
// Optional: analogReadResolution(12);
}
void loop() {
potValue = analogRead(potPin);
Serial.print("Analog␣Value:␣");
Serial.println(potValue); // Prints 0 to 4095
delay(100);
}
```

Understanding PWM (Pulse Width Modulation)



PWM Parameters: Frequency and Duty Cycle

- Period (T): The time it takes for one complete ON and OFF cycle.
- Frequency (f): The number of cycles per second ($f = 1/T$). Measured in Hertz (Hz).
- Duty Cycle: The percentage of time the signal is ON during one period.
- 0
- 50
- 100
- Resolution: Number of discrete steps for the duty cycle (e.g., 8-bit = 256 steps).

ESP32 PWM Generation (LEDC API)

- Unlike standard Arduino (‘analogWrite‘), ESP32 has a dedicated LED PWM (LEDC) peripheral.
- Key Features of LEDC:
- 16 independent PWM channels.
- Configurable frequency and resolution.
- Can be routed to almost any GPIO pin.
- Steps to configure LEDC:
- 1. ‘ledcSetup(channel, freq, resolution)‘
- 2. ‘ledcAttachPin(gpio, channel)‘
- 3. ‘ledcWrite(channel, dutyCycle)‘

Fading an LED with PWM

- Hardware Setup:
- Connect an LED anode (long leg) to GPIO 16 via a 220 Ω resistor.
- Connect LED cathode (short leg) to GND.
- PWM Configuration:
- Channel: 0 (0 to 15 available)
- Frequency: 5000 Hz
- Resolution: 8-bit (Values from 0 to 255)
- By varying the duty cycle from 0 to 255, the LED brightness changes from fully off to fully on.

Code: LED Brightness Control (ledcWrite)

```
const int ledPin = 16;
const int freq = 5000;          // 5 kHz
const int ledChannel = 0;
const int resolution = 8;       // 8-bit (0-255)
void setup() {
ledcSetup(ledChannel, freq, resolution);
ledcAttachPin(ledPin, ledChannel);
}
void loop() {
for(int duty = 0; duty <= 255; duty++) { // Fade in
ledcWrite(ledChannel, duty);
delay(15);
}
for(int duty = 255; duty >= 0; duty--) { // Fade out
ledcWrite(ledChannel, duty);
delay(15);
}
}
```

Key Takeaways

- ESP32 features two 12-bit ADCs. ADC1 is preferred when using Wi-Fi.
- ‘analogRead()‘ returns values from 0 to 4095 corresponding to 0V to 3.3V.
- PWM simulates analog output by rapidly toggling a digital pin.
- Duty cycle determines the average output voltage.
- ESP32 uses the ‘ledc‘ API (‘ledcSetup‘, ‘ledcAttachPin‘, ‘ledcWrite‘) for robust PWM generation.

Next Lecture Preview

- Topic: 3.5 Serial Communication (Part 1)
- Overview of UART communication.
- Hardware serial vs. Software serial concepts.
- Interfacing the ESP32 with external serial devices.

Agenda

What we will cover today

- Introduction to Serial Communication
- Hardware UARTs in ESP32
- Initializing Serial: `Serial.begin()`
- Sending Data: `Serial.print()` & `Serial.println()`
- Receiving Data: `Serial.available()` & `Serial.read()`
- Hands-on Echo Program
- Debugging Techniques using Serial Monitor

Learning Objectives

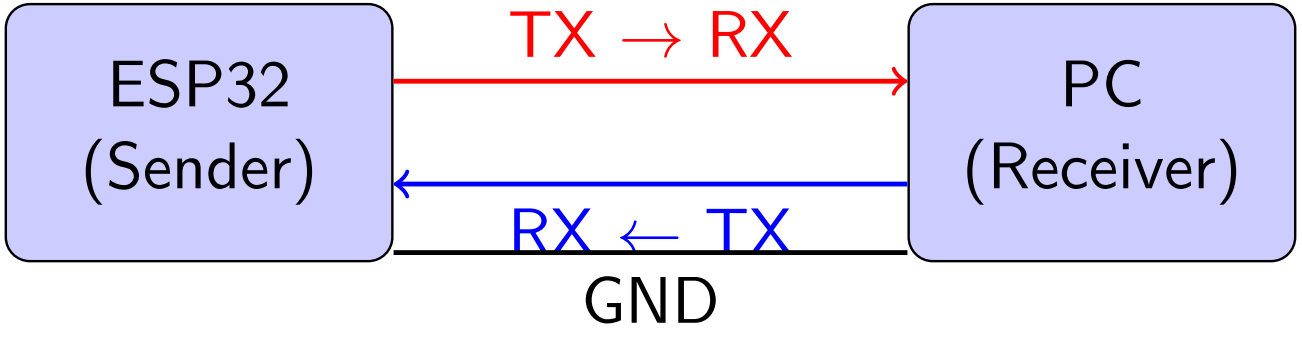
By the end of this lecture, students will be able to:

- Understand the principles of asynchronous serial communication.
- Configure the ESP32 UART interface using the Arduino IDE.
- Send and receive data between the ESP32 and a computer.
- Utilize the Serial Monitor effectively for debugging code logic and hardware issues.

Introduction to Serial Communication

Asynchronous Data Transfer

- **What is it?** The process of sending data one bit at a time, sequentially, over a communication channel.
- **Asynchronous:** No shared clock signal between sender and receiver. They must agree on the transmission speed (baud rate) beforehand.
- **Lines Used:** Typically requires three lines: TX (Transmit), RX (Receive), and GND (Common Ground).
- **Protocol:** Data is framed with a start bit, 8 data bits (usually), an optional parity bit, and stop bit(s) (8N1 configuration is standard).



Hardware UARTs in ESP32

Universal Asynchronous Receiver-Transmitter

- The ESP32 chip has **three** hardware UART interfaces: UART0, UART1, and UART2.

Interface	**TX Pin**	**RX Pin**	**Typical Use**
UART0 (Serial)	GPIO 1	GPIO 3	Programming, default Serial Monitor debugging.
UART1 (Serial1)	GPIO 10	GPIO 9	Often tied to flash memory, requires pin remapping.
UART2 (Serial2)	GPIO 17	GPIO 16	Freely available (e.g., GPS, serial displays).

Understanding Baud Rate

Speed of Communication

- **Baud Rate:** The rate at which information is transferred in a communication channel. Measured in bits per second (bps).
- Both the ESP32 and the connected device (e.g., Serial Monitor) **MUST** be set to the exact same baud rate.
- **Common Rates:** 9600, 115200, 921600.
- **ESP32 Standard:** While older Arduinos default to 9600, the ESP32 typically uses **115200** baud as standard for faster firmware uploading and debugging output.

Initializing Serial: Serial.begin()

Setting up the UART

- To use serial communication, it must be initialized in the `setup()` function.
- **Syntax:** `Serial.begin(baud_rate);`

Example Code:

```
void setup() {
    // Initialize UART0 at 115200 bps
    Serial.begin(115200);
    // Wait for serial port to connect
    delay(1000);
    Serial.println("Serial initialized successfully!");
}
```

Sending Data

Serial.print() & Serial.println()

- `Serial.print(data)`: Prints data to the serial port as human-readable ASCII text.
- `Serial.println(data)`: Does the same, but appends a carriage return (`\r`) and newline (`\n`) character, moving the cursor to the next line.

Example Code:

```
void loop() {
    int sensorValue = analogRead(34);
    Serial.print("Sensor Reading: "); // Stays on same line
    Serial.println(sensorValue);      // Prints value and
        moves to next line
    delay(500);
}
```

Receiving Data

Serial.available() & Serial.read()

- `Serial.available()`: Returns the number of bytes available to read in the serial receive buffer.
- `Serial.read()`: Reads the first available byte of incoming serial data. Returns -1 if no data is available.

Example Code:

```
void loop() {
    // Check if data has been received
    if (Serial.available() > 0) {
        // Read the incoming byte
        char incomingByte = Serial.read();
        Serial.print("I received: ");
        Serial.println(incomingByte);
    }
}
```

Putting it Together: Echo Program

Two-way Communication

This program reads whatever you type in the Serial Monitor and sends it back in uppercase.

```
void setup() {
  Serial.begin(115200);
  Serial.println("Echo Server Ready. Type something:");
}

void loop() {
  if (Serial.available()) {
    String incomingString = Serial.readStringUntil('\n');
    incomingString.trim(); // Remove whitespace/newlines
    incomingString.toUpperCase();

    Serial.print("ESP32 Echo: ");
    Serial.println(incomingString);
  }
}
```

Debugging using Serial Monitor

Your Best Friend in IoT

- **Purpose:** The Serial Monitor is an essential tool to see what is happening inside the microcontroller.
- **Tracking Execution:** Use `Serial.println("Entered Function X");` to verify program flow.
- **Variable Inspection:** Print sensor values, loop counters, and state variables to ensure they hold expected data.
- **Error Messages:** Explicitly print error codes if a peripheral (like WiFi or a sensor) fails to initialize.

Common Serial Communication Issues

Troubleshooting Tips

- **Garbage Characters:** Caused by baud rate mismatch. Ensure `Serial.begin(X)` matches the monitor's dropdown.
- **Board Not Detected / COM Port Missing:** Check USB cable (must support data, not just charging), install CP2102/CH340 drivers.
- **Upload Fails (Connecting...):** Sometimes the ESP32 needs manual boot mode entry. Hold the 'BOOT' button when connecting starts.
- **Missing First Messages:** ESP32 boots faster than the serial monitor connects. Add a short `delay(1000);` after `Serial.begin()`.

Key Takeaways

Lecture Summary

- Serial communication provides an asynchronous data link between the ESP32 and a PC or other devices.
- The ESP32 has three hardware UARTs, with UART0 used by default for the USB connection.
- `Serial.begin(115200)` initializes the connection.
- `Serial.print()` and `Serial.println()` are used for transmitting data.
- `Serial.available()` and `Serial.read()` handle incoming data reception.
- The Serial Monitor is the most crucial tool for debugging ESP32 applications.

Next Lecture Preview

Moving from Basic I/O to Timers

- We have now covered Digital I/O, Analog I/O, PWM, and Serial Communication.
- Next, we will start **Unit 4: Timers and Interrupts**.
- We will explore how to schedule tasks without using blocking `delay()` functions.
- Introduction to Hardware Timers in the ESP32.
- Understanding the concept of Interrupt Service Routines (ISRs).

Agenda

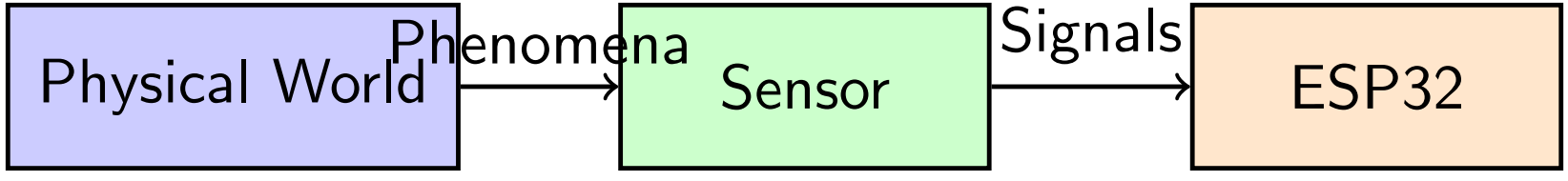
- Introduction to Sensor Interfacing
- DHT11 & DHT22 Temperature and Humidity Sensors
- Programming DHT Sensors (Arduino C++)
- Light Dependent Resistor (LDR)
- Passive Infrared (PIR) Motion Sensor
- Programming PIR Sensors

Learning Objectives

- Understand the principles of interfacing digital and analog sensors.
- Learn how to wire and program DHT11/DHT22 sensors for environmental monitoring.
- Comprehend the working mechanism of an LDR for light intensity measurement.
- Implement motion detection using a PIR sensor with digital reads.

Introduction to Sensor Interfacing

- Connecting the Physical World to IoT
- **Sensors** convert physical phenomena (temperature, light, motion) into electrical signals.
- **Analog Sensors**: Output a continuous voltage. Require ADC (Analog-to-Digital Converter) to read.
- **Digital Sensors**: Output discrete values (High/Low) or communicate via protocols (I2C, SPI, One-Wire).
- **Interfacing Steps**: Power supply (3.3V/5V), Signal connection (GPIO/ADC), and Ground (GND).



DHT11 & DHT22 Sensors

- Temperature and Humidity
- **DHT11**: Low-cost, basic sensor. Temp range: $0 - 50^{\circ}C$ ($\pm 2^{\circ}C$), Humidity: $20 - 80\%$ ($\pm 5\%$).
- **DHT22 (AM2302)**: Higher precision. Temp range: -40 to $80^{\circ}C$ ($\pm 0.5^{\circ}C$), Humidity: $0 - 100\%$ ($\pm 2\%$).
- **Protocol**: Uses a proprietary single-wire digital interface.
- Both consist of a capacitive humidity sensing element and a thermistor.

Feature	**DHT11**	**DHT22 (AM2302)**
Temp Range	$0 - 50^{\circ}C$ ($\pm 2^{\circ}C$)	-40 to $80^{\circ}C$ ($\pm 0.5^{\circ}C$)
Humidity Range	$20 - 80\%$ ($\pm 5\%$)	$0 - 100\%$ ($\pm 2\%$)
Cost	Low	Medium

DHT Pinout and Connection

- Wiring to ESP32
- **Pin 1 (VCC)**: 3.3V to 5V power supply.
- **Pin 2 (DATA)**: Connect to any digital GPIO (e.g., GPIO 4).
- **Pin 3 (NC)**: Not connected.
- **Pin 4 (GND)**: Ground.
- **Pull-up Resistor**: A 10kΩ pull-up resistor is required between VCC and DATA pin (often built-in on breakout boards).

Programming DHT Sensors

- Arduino C++ Code Snippet

```
#include "DHT.h"
#define DHTPIN 4     // Digital pin connected to the DHT
    sensor
#define DHTTYPE DHT11   // or DHT22

DHT dht(DHTPIN, DHTTYPE);

void setup() {
  Serial.begin(115200);
  dht.begin();
}

void loop() {
  delay(2000); // DHT needs 2s between reads
  float h = dht.readHumidity();
  float t = dht.readTemperature();

  if (isnan(h) || isnan(t)) {
    Serial.println("Failed to read from DHT sensor!");
    return;
  }
  Serial.printf("Temp: %.2f C, Humidity: %.2f %%\n", t, h);
}
```

Light Dependent Resistor (LDR)

- Photoresistor Basics
- **Function**: A variable resistor whose resistance decreases as light intensity increases.
- **Analog Sensor**: Requires a voltage divider circuit to produce a readable voltage.
- **Applications**: Automatic streetlights, day/night detection, solar tracking.
- **Characteristics**: Non-linear response to light, slow reaction time.

LDR Interfacing and Code

- Reading Analog Values
- **Circuit**: Connect LDR and a fixed resistor (e.g., 10kΩ) in series between VCC and GND. Connect the midpoint to an ADC pin (e.g., GPIO 34).

```
#define LDR_PIN 34

void setup() {
  Serial.begin(115200);
  // Optional: analogReadResolution(12); for ESP32
}

void loop() {
  int ldrValue = analogRead(LDR_PIN);
  Serial.print("Light␣Level␣(0-4095):␣");
  Serial.println(ldrValue);
  delay(500);
}
```

PIR Motion Sensor

- Passive Infrared Sensor (HC-SR501)
- **Function**: Detects motion by measuring changes in the infrared (heat) levels emitted by surrounding objects.
- **Digital Output**: Outputs a logic HIGH (3.3V) when motion is detected, and LOW when idle.
- **Coverage**: Typically up to 7 meters and a 110° cone angle.
- **Applications**: Security alarms, automated lighting, occupancy detection.

PIR Pinout and Adjustments

- HC-SR501 Module
- **Pinout**: VCC (5V recommended), OUT (3.3V Logic), GND.
- **Potentiometer 1 (Sensitivity/Distance)**: Adjusts the detection range (3m to 7m).
- **Potentiometer 2 (Time Delay)**: Adjusts how long the OUT pin stays HIGH after motion stops (0.3s to 5 mins).
- **Trigger Jumper**: Selects between Single Trigger (L) and Repeatable Trigger (H).

Programming PIR Sensor

- Reading Digital State

```
#define PIR_PIN 27

void setup() {
  Serial.begin(115200);
  pinMode(PIR_PIN, INPUT);
  Serial.println("PIR␣Sensor␣Calibrating...␣wait␣30s");
}

void loop() {
  int motionState = digitalRead(PIR_PIN);

  if (motionState == HIGH) {
    Serial.println("Motion␣Detected!");
  } else {
    Serial.println("All␣clear.");
  }
  delay(500);
}
```

Key Takeaways

- **DHT Sensors**: Require specific libraries and strict timing (2s delays) to read temperature and humidity reliably.
- **LDRs**: Analog sensors that need a voltage divider and ADC pin to quantify light intensity.
- **PIR Sensors**: Provide simple digital logic outputs for motion detection, with hardware tunability for sensitivity and delay.
- ESP32 pin configuration (3.3V logic, ADC resolutions) is critical when choosing and wiring these sensors.

Next Lecture Preview

- **Interfacing and Programming Sensors (Part 2)**
- Ultrasonic Distance Sensor (HC-SR04).
- Soil Moisture Sensor.
- Gas Sensors (MQ series).
- Integrating multiple sensors into a single ESP32 project.

4.1 Interfacing and Programming Sensors (Part 2)

- Focus on Gas (MQ-4), Ultrasonic (HC-SR04), and Water Level Sensors

Agenda

- Introduction to Gas Sensors (MQ-4)
- MQ-4 Interfacing & Code
- Ultrasonic Sensor (HC-SR04) Principle
- HC-SR04 Interfacing & Code
- Water Level Sensor Overview
- Water Level Sensor Code

Learning Objectives

- By the end of this lecture, you will be able to:
- Understand the working mechanism of the MQ-4 gas sensor and read analog data.
- Measure distance using the HC-SR04 ultrasonic sensor and time-of-flight principles.
- Interface and program an analog water level sensor with the ESP32.

Gas Sensor (MQ-4): Introduction

- **Purpose**: Detects Methane (CH4) and Natural Gas (CNG).
- **Working Principle**: Contains a sensing element (SnO2) whose resistance changes based on gas concentration.
- **Output**: Provides both Analog (A0) and Digital (D0) outputs.
- **Heater**: Requires a small internal heater to reach operating temperature, drawing around 150mA.

MQ-4 Pinout & Interfacing

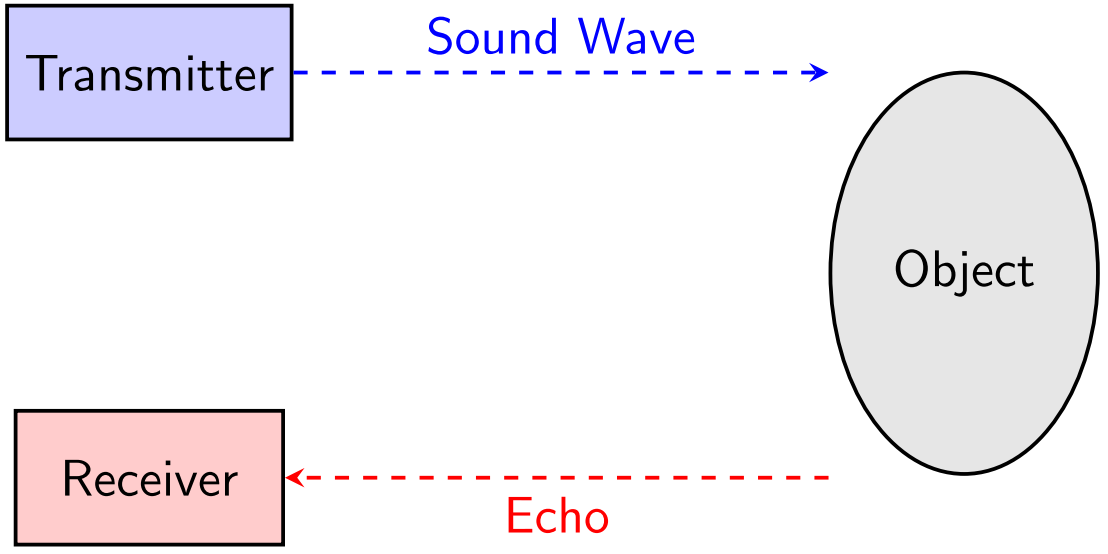
- Pins:
- **VCC**: 5V (Note: ESP32 ADC pins are 3.3V tolerant, a voltage divider is recommended for safety if output exceeds 3.3V).
- **GND**: Ground.
- **A0**: Analog Output (Proportional to gas concentration).
- **D0**: Digital Output (Adjustable via onboard potentiometer).
- ESP32 Connection:
- Connect A0 to an ADC-capable pin (e.g., GPIO 34).

Code Snippet: MQ-4 Analog Reading

```
const int mq4Pin = 34; // ADC pin for MQ-4

void setup() {
  Serial.begin(115200);
}

void loop() {
  // Read 12-bit ADC value (0-4095)
  int sensorValue = analogRead(mq4Pin);
  Serial.print("MQ-4 Gas Level: ");
  Serial.println(sensorValue);

  if (sensorValue > 2000) {
    Serial.println("WARNING: High Gas Concentration!");
  }
  delay(1000);
}
```

Ultrasonic Sensor (HC-SR04)

- **Purpose**: Non-contact distance measurement (2cm to 400cm).
- **Principle**: Time-of-Flight (ToF). Uses high-frequency sound waves (40 kHz).
- **Components**: One transmitter (speaker) and one receiver (microphone).
- **Formula**: Distance = (Speed of Sound $\times$ Time) / 2
- **Speed of sound**: $\sim$343 m/s (or 0.0343 cm/μs).



HC-SR04 Pinout & Working Cycle

- Pins:
- **VCC**: 5V power supply.
- **Trig**: Trigger input (requires a 10μs HIGH pulse to start).
- **Echo**: Outputs a HIGH pulse whose duration equals the sound travel time.
- **GND**: Ground.
- Sequence:
- Pull Trig HIGH for 10μs.
- Sensor sends 8-cycle 40kHz sonic burst.
- Echo pin goes HIGH until the wave returns.

Interfacing HC-SR04 with ESP32

- **Trig Pin**: Can be connected to any standard GPIO (e.g., GPIO 5).
- **Echo Pin**: Connect to any standard GPIO (e.g., GPIO 18).
- **Level Shifting**: HC-SR04 outputs a 5V signal on the Echo pin. A simple voltage divider (e.g., 1kΩ and 2kΩ resistors) should be used to step it down to 3.3V for the ESP32 input.

Code Snippet: HC-SR04 pulseIn()

```
const int trigPin = 5; const int echoPin = 18;
void setup() {
  Serial.begin(115200);
  pinMode(trigPin, OUTPUT);
  pinMode(echoPin, INPUT);
}
void loop() {
  digitalWrite(trigPin, LOW); delayMicroseconds(2);
  digitalWrite(trigPin, HIGH); delayMicroseconds(10);
  digitalWrite(trigPin, LOW);

  long duration = pulseIn(echoPin, HIGH);
  float distance_cm = duration * 0.034 / 2;

  Serial.print("Distance: "); Serial.print(distance_cm);
  Serial.println(" cm"); delay(1000);
}
```

Water Level Sensor: Overview

- **Purpose**: Detect the depth of water or presence of liquid.
- **Construction**: A series of exposed parallel copper traces.
- **Principle**: Acts as a variable resistor. Water bridges the traces, changing the conductivity.
- **Output**: Analog voltage proportional to the submerged depth.
- **Pros/Cons**: Very cheap and simple, but prone to corrosion (electrolysis) over time.

Code Snippet: Water Level Sensor

```
const int waterSensorPin = 32; // ADC1 channel

void setup() {
  Serial.begin(115200);
}

void loop() {
  int waterLevel = analogRead(waterSensorPin);
  Serial.print("Water Level (Raw): ");
  Serial.println(waterLevel);

  if (waterLevel == 0) Serial.println("Status: Dry");
  else if (waterLevel > 0 && waterLevel < 1500) Serial.
      println("Status: Shallow");
  else Serial.println("Status: Deep");

  delay(1000);
}
```

Key Takeaways

- **MQ-4 Gas Sensor**: Uses heated SnO2 to detect CH4; requires analog reading and calibration.
- **HC-SR04 Ultrasonic**: Uses time-of-flight sound pulses to measure distance. Requires 5V-to-3.3V logic shifting on the Echo pin. Calculated via duration $\times$ 0.034 / 2.
- **Water Level Sensor**: Uses conductivity across PCB traces to measure liquid depth; outputs analog voltage.

Next Lecture Preview

- **Topic**: 4.1 Interfacing and Programming Sensors (Part 3)
- Interfacing the PIR (Passive Infrared) Motion Sensor
- Interfacing the LDR (Light Dependent Resistor)
- Interrupt-driven sensor programming in ESP32

Basics of Data Acquisition (DAQ)

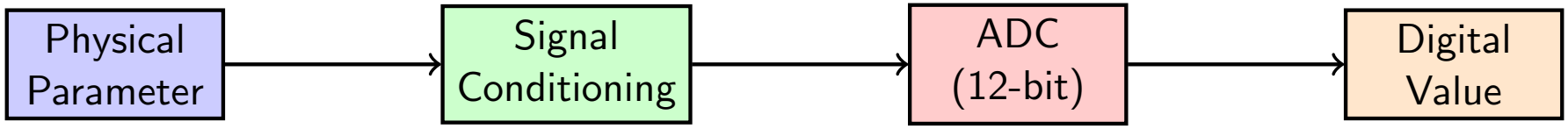
- Basics of Data Acquisition (DAQ)
- Sampling Rates and Aliasing
- Filtering Sensor Noise in Software
- Serial Communication Fundamentals
- Formatting Data as CSV for Serial Monitor
- Introduction to JSON Data Format
- ArduinoJson Library for ESP32
- Formatting Data as JSON over Serial

Learning Objectives

- Understand the principles of Data Acquisition (DAQ) and sampling rates.
- Apply simple software filters (like moving average) to raw sensor data.
- Transmit sensor data effectively over the ESP32 Serial interface.
- Format data outputs into CSV (Comma-Separated Values).
- Format data outputs into JSON using the ArduinoJson library.

Data Acquisition (DAQ)

- Data acquisition is the process of sampling signals that measure real-world physical conditions and converting the resulting samples into digital numeric values.
- Key Components:
  - **Sensors:** Convert physical parameters (temperature, light) to electrical signals.
  - **Signal Conditioning:** Amplification, filtering (hardware).
  - **Analog-to-Digital Converter (ADC):** ESP32 has 12-bit ADCs (0-4095 range).



Sampling Rates and Aliasing

- The **Sampling Rate** is how often the ESP32 reads the sensor.
- **Nyquist-Shannon Sampling Theorem:** The sampling frequency must be at least twice the highest frequency component of the signal to prevent aliasing.
- **Polling vs. Interrupts:**
  - Polling (using `delay()` or `millis()`) is common for slow-changing signals (temperature).
  - Interrupts or hardware timers are used for high-frequency DAQ (audio, vibrations).

Filtering Sensor Noise in Software

- Sensors often produce noisy data. A simple way to smooth this is a Moving Average filter.

```
const int numReadings = 10;
int readings[numReadings];
int readIndex = 0;
long total = 0;

void setup() {
  for (int i = 0; i < numReadings; i++) readings[i] = 0;
}

int getSmoothedData(int pin) {
  total = total - readings[readIndex];
  readings[readIndex] = analogRead(pin);
  total = total + readings[readIndex];
  readIndex = (readIndex + 1) % numReadings;
  return total / numReadings;
}
```

Serial Communication Fundamentals

- The ESP32 communicates with the computer via UART (Universal Asynchronous Receiver-Transmitter).
- Initialization: `Serial.begin(115200);` (115200 baud is standard for ESP32).
- Sending Data: `Serial.print()` and `Serial.println()`.
- Why format? Raw streams of numbers are hard for external Python scripts, Node-RED, or databases to parse automatically.

Formatting Data as CSV for Serial Monitor

- CSV is the simplest way to transmit multiple sensor readings simultaneously.
- Format: `timestamp,sensor1,sensor2\n`

```
unsigned long lastTime = 0;

void loop() {
  if (millis() - lastTime > 1000) {
    lastTime = millis();
    int tempRaw = analogRead(34);
    int humidRaw = analogRead(35);

    // Output: time,temp,humid
    Serial.print(lastTime);
    Serial.print(",");
    Serial.print(tempRaw);
    Serial.print(",");
    Serial.println(humidRaw);
  }
}
```

Arduino IDE Serial Plotter

- The Arduino IDE includes a 'Serial Plotter' tool.
- It automatically parses CSV (or space-separated) data over the Serial port.
- It graphs the variables in real-time.
- To label the axes, print a header row in the `setup()` function.

```
void setup() {
  Serial.begin(115200);
  Serial.println("Timestamp,Temperature,Humidity"); //
      Header for Plotter
}
```

Introduction to JSON Data Format

- JSON (JavaScript Object Notation) is a lightweight data-interchange format.
- Human-readable and machine-parseable.
- Based on key-value pairs.
- Extensively used in web APIs and IoT platforms (AWS IoT, Google Cloud).

```
{
  "device_id": "ESP32_Node1",
  "temperature": 24.5,
  "humidity": 60
}
```

ArduinoJson Library for ESP32

- To handle JSON in C++ efficiently, we use the **ArduinoJson** library by Benoit Blanchon.
- Must be installed via the Library Manager.
- Uses a `JsonDocument` (replaces older Dynamic/StaticJsonDocument in v7) to allocate memory for the JSON object.
- Handles serialization (C++ variables to JSON string) and deserialization (JSON string to C++ variables).

Formatting Data as JSON over Serial

```
#include <ArduinoJson.h>

void loop() {
  // 1. Create a JSON document
  JsonDocument doc;

  // 2. Add data (key-value pairs)
  doc["sensor"] = "DHT22";
  doc["time"] = millis();
  doc["data"][0] = 23.5; // Nested array for temperature
  doc["data"][1] = 55.2; // Nested array for humidity

  // 3. Serialize to Serial port
  serializeJson(doc, Serial);
  Serial.println(); // Add newline for readability

  delay(2000);
}
```

Summary

- Data acquisition involves sampling continuous signals into discrete digital values.
- Software filters, like the moving average, are crucial for mitigating sensor noise.
- Formatting data as CSV allows for easy visualization in the Serial Plotter.
- JSON is the standard format for self-describing data in modern IoT applications.
- The ArduinoJson library provides a robust, memory-safe way to serialize C++ data into JSON.

Next Lecture Preview

- In the next lecture, we will dive into Unit 4.2: Actuators and Output Devices.
- Topics:
  - Controlling LEDs with PWM
  - Interfacing Relays for high-power devices
  - Driving Servo and Stepper Motors

Agenda

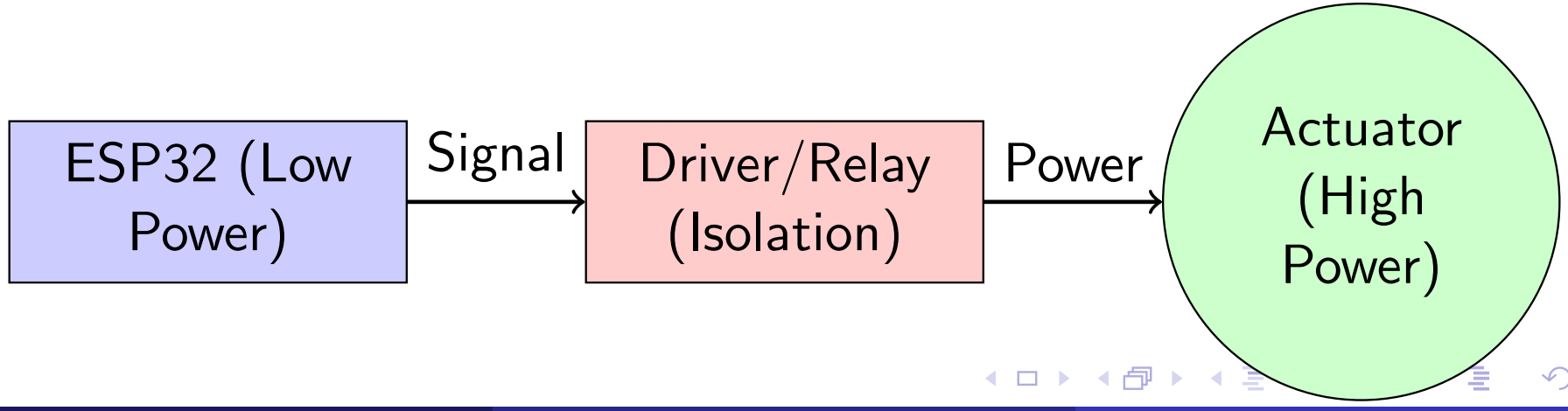
- Learning Objectives
- Introduction to Actuators
- Relay Module Overview
- Interfacing Relay with ESP32
- Code: Controlling a Relay
- DC Motor Basics & Challenges
- L298N Motor Driver Overview
- Interfacing L298N with ESP32
- Code: DC Motor Control
- Stepper Motor Introduction
- Key Takeaways
- Next Lecture Preview

Learning Objectives

- Understand the role of actuators in IoT systems.
- Interface and program a relay module to control high-voltage/current loads.
- Comprehend the need for motor drivers like the L298N.
- Interface and program a DC motor for speed and direction control.
- Differentiate between DC and stepper motors.

Introduction to Actuators

- **Actuators** are devices that convert electrical control signals into physical or mechanical action.
- **Common Types in IoT:**
- **Relays:** Electromechanical switches for AC/DC loads.
- **DC Motors:** Provide continuous rotation.
- **Stepper/Servo Motors:** Provide precise angular or linear positioning.
- **Why ESP32 needs help:** Microcontroller GPIO pins can only source/sink a few milliamperes (typically ∼12-40mA at 3.3V). Actuators require much higher current and sometimes higher voltages.



Relay Module Overview

- A **Relay** is an electrically operated switch.
- **Components of a Relay Module:**
- **Electromagnet (Coil):** Energized by the control signal.
- **Armature & Contacts:** Physically move to open/close the circuit.
- **Optocoupler:** Often included in modules for electrical isolation.
- **Flyback Diode:** Protects the driving circuit from voltage spikes when the coil de-energizes.
- **Terminals:**
- **Control Side:** VCC, GND, IN (Signal)
- **Load Side:** COM (Common), NO (Normally Open), NC (Normally Closed)

Interfacing Relay with ESP32

- **Wiring connections:**
- Relay **VCC** $\rightarrow$ ESP32 **5V (VIN)** (Ensure external power if needed, relays draw $\sim$70mA).
- Relay **GND** $\rightarrow$ ESP32 **GND**.
- Relay **IN** $\rightarrow$ ESP32 **GPIO (e.g., GPIO 26)**.
- **Active Low vs Active High:**
- Many 5V relay modules are **Active Low** (a LOW signal turns the relay ON).
- Check the module specifications or test carefully.

Code: Controlling a Relay (Arduino C++)

```
const int relayPin = 26; // GPIO connected to Relay IN

void setup() {
  pinMode(relayPin, OUTPUT);
  // For Active Low relay, set HIGH to turn OFF initially
  digitalWrite(relayPin, HIGH);
}

void loop() {
  digitalWrite(relayPin, LOW);  // Turn Relay ON
  delay(2000);
  digitalWrite(relayPin, HIGH); // Turn Relay OFF
  delay(2000);
}
```

DC Motor Basics & Challenges

- **DC Motors** convert direct current into rotational motion.
- **Control Requirements:**
- **Direction:** Reversed by swapping the polarity of the applied voltage.
- **Speed:** Controlled by varying the average voltage using **PWM** (Pulse Width Modulation).
- **Challenges:**
- **High Current Draw:** Stall currents can be very high.
- **Inductive Kickback:** Motors are inductors; turning them off generates high voltage spikes.
- **Solution:** H-Bridge circuit with flyback diodes.

L298N Motor Driver Overview

- The **L298N** is a dual H-Bridge motor driver module.
- **Features:**
- Can control up to 2 DC motors (or 1 stepper motor).
- Operating voltage: 5V to 35V.
- Continuous current: Up to 2A per channel.
- **Pins:**
- **Power:** 12V (Motor Power), GND, 5V (Logic power).
- **Motor Outputs:** OUT1/OUT2 (Motor A), OUT3/OUT4 (Motor B).
- **Control Pins:** ENA (PWM A), IN1, IN2, IN3, IN4, ENB (PWM B).

Motor Control	**Pin 1 (e.g., IN1)**	**Pin 2 (e.g., IN2)**
Forward	HIGH	LOW
Backward	LOW	HIGH
Stop	LOW	LOW

Interfacing L298N with ESP32

- **Wiring for Motor A:**
- **ESP32 GPIO 14** $\rightarrow$ **IN1** (Direction pin 1)
- **ESP32 GPIO 27** $\rightarrow$ **IN2** (Direction pin 2)
- **ESP32 GPIO 26** $\rightarrow$ **ENA** (PWM for speed)
- **ESP32 GND** $\rightarrow$ **L298N GND** (CRITICAL: Common Ground!)
- **Motor & Power:**
- Motor connected to OUT1 and OUT2.
- External Battery (e.g., 9V) $\rightarrow$ L298N 12V terminal.
- Battery GND $\rightarrow$ L298N GND.

Code: DC Motor Control with L298N (Arduino C++)

```
const int in1 = 14, in2 = 27, enA = 26;

void setup() {
  pinMode(in1, OUTPUT); pinMode(in2, OUTPUT);
  // Configure LED PWM functionalities (ESP32 specific)
  ledcSetup(0, 5000, 8); // Channel 0, 5kHz, 8-bit
  ledcAttachPin(enA, 0);
}

void loop() {
  // Move Forward at 50% speed
  digitalWrite(in1, HIGH); digitalWrite(in2, LOW);
  ledcWrite(0, 127); // 127 is half of 255
  delay(2000);

  // Stop
  digitalWrite(in1, LOW); digitalWrite(in2, LOW);
  delay(1000);
}
```

Stepper Motor Introduction

- A **Stepper Motor** divides a full rotation into a large number of discrete steps.
- **Characteristics:**
- High precision positioning without feedback (open-loop).
- High holding torque at low speeds.
- **Types:**
- **Unipolar:** Typically 5 or 6 wires. Current flows in one direction through coils. (e.g., 28BYJ-48 with ULN2003).
- **Bipolar:** 4 wires. Requires reversing current flow through coils (e.g., NEMA 17 with A4988).

Key Takeaways

- Actuators require external power sources and intermediary driver circuits to protect the ESP32.
- Relay modules provide simple on/off control with electrical isolation for high-power AC/DC loads.
- DC motors require an H-Bridge (like L298N) for bidirectional control and PWM for speed regulation.
- Sharing a common ground between the microcontroller and motor driver is essential.
- Stepper motors provide precise rotational steps for positional accuracy.

Next Lecture Preview

- **Topic:** 4.2 Interfacing and Programming of Actuators (Part 2)
- **Focus Areas:**
- Programming Stepper Motors with ESP32 (ULN2003 driver).
- Servo Motor Interfacing.
- Precision control using PWM for Servos.

Interfacing and Programming of Actuators (Part 2)

- Course: Internet of Things (DI05000041)
- Unit 4: Interfacing and Internet-Based Control using ESP32
- Lecture 4.5: Interfacing and Programming of Actuators (Part 2)
- Topics: Servo motor, Solenoid valve, Buzzer

Agenda

- Learning Objectives
- Introduction to Servo Motors
- Servo Motor Working Principle
- Interfacing Servo with Microcontroller
- Programming the Servo Motor
- Introduction to Solenoid Valves
- Interfacing Solenoid Valves via Relay
- Programming the Solenoid Valve
- Introduction to Buzzers (Active & Passive)
- Programming Buzzers (Digital & PWM)

Learning Objectives

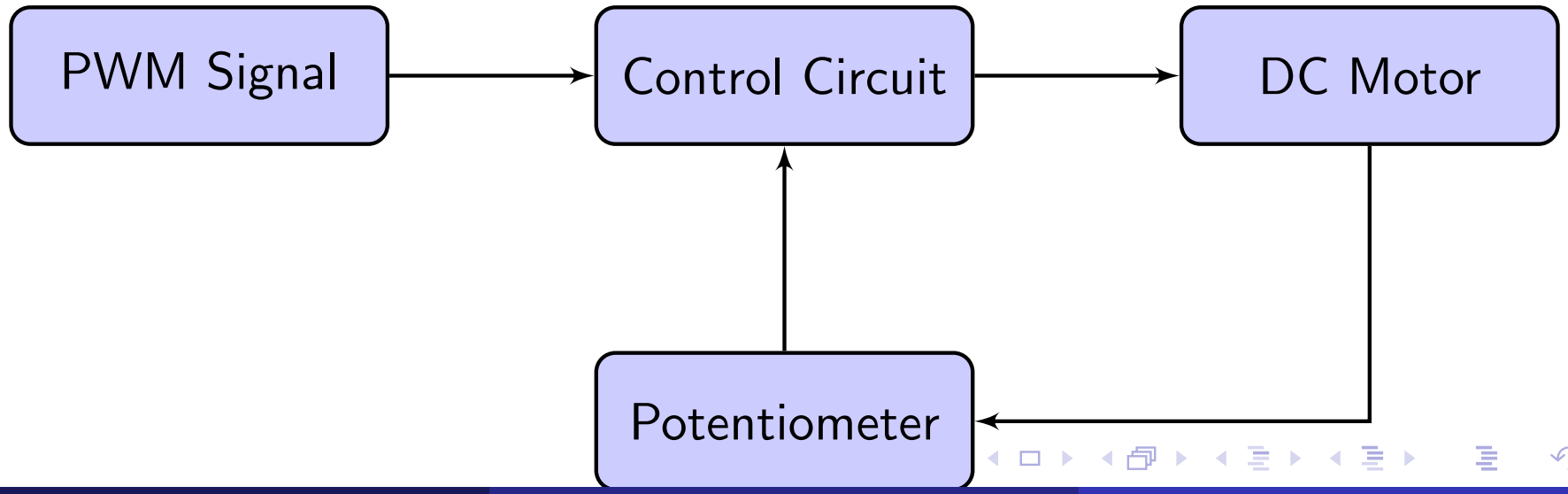
- By the end of this lecture, students will be able to:
- Explain the internal mechanism and PWM control of a Servo motor.
- Write Arduino C++ code using `Servo.h` to control angular position.
- Design an interfacing circuit for high-power Solenoid valves using a relay or MOSFET.
- Differentiate between Active and Passive buzzers.
- Generate simple tones using digital output and PWM for passive buzzers.

Introduction to Servo Motor

- A **Servo motor** is a rotary actuator or linear actuator that allows for precise control of angular or linear position, velocity, and acceleration.
- **Components**: Consists of a suitable motor coupled to a sensor for position feedback.
- **Control Signal**: Driven by Pulse Width Modulation (PWM) signals.
- **Applications**: Robotics, CNC machinery, automated manufacturing, and steering systems.

Servo Motor Working Principle

- **PWM Control**: The angle of rotation is determined by the duration of a pulse applied to the control wire.
- **Standard Timing**: A pulse of $\sim$ 1.0 ms typically moves the servo to 0 degrees, 1.5 ms to 90 degrees (neutral), and 2.0 ms to 180 degrees.
- **Refresh Rate**: The pulse must be repeated every 20 ms (50 Hz).
- **Feedback Mechanism**: A potentiometer is attached to the output shaft to measure the current angle and correct it using an error amplifier.



Interfacing Servo Motor

- Connecting a standard servo (like SG90 or MG996R) to a microcontroller:
- **VCC (Red wire)**: Connect to 5V (or 3.3V depending on the servo rating). *Note: High torque servos need an external power supply.*
- **GND (Brown/Black wire)**: Connect to the system Ground.
- **Signal (Yellow/Orange wire)**: Connect to a PWM-capable digital pin on the microcontroller (e.g., Pin 9 on Arduino Uno or GPIO 18 on ESP32).

Programming: Servo Motor Control

```
#include <Servo.h>

Servo myServo;  // Create servo object
int servoPin = 9;

void setup() {
  myServo.attach(servoPin); // Attach servo to pin 9
}

void loop() {
  // Sweep from 0 to 180 degrees
  for (int pos = 0; pos <= 180; pos += 1) {
    myServo.write(pos); // Tell servo to go to 'pos'
    delay(15);          // Wait for servo to reach position
  }
  // Sweep from 180 back to 0 degrees
  for (int pos = 180; pos >= 0; pos -= 1) {
    myServo.write(pos);
    delay(15);
  }
}
```

Introduction to Solenoid Valve

- A **Solenoid valve** is an electromechanically operated valve used to control the flow of liquids or gases.
- **Mechanism**: An electric current passes through a solenoid coil, creating a magnetic field that moves a plunger to open or close the valve.
- **Types**: Normally Closed (NC) - opens when energized; Normally Open (NO) - closes when energized.
- **Applications**: Irrigation systems, washing machines, industrial fluid control.

Interfacing Solenoid Valve

- **Power Requirements**: Most solenoid valves require 12V or 24V DC and draw substantial current (e.g., 500mA - 1A).
- **Interfacing Circuit**: Use a Relay module or a Power MOSFET (like IRF520 or IRLZ44N) to switch the high power load from a 3.3V/5V logic signal.
- **Flyback Diode**: A flyback diode (e.g., 1N4007) MUST be placed in parallel with the solenoid coil (reverse biased) to protect the switching transistor from high voltage spikes when the coil is turned off.

Programming: Solenoid Valve via Relay

```
const int relayPin = 7; // Pin connected to relay IN

void setup() {
  pinMode(relayPin, OUTPUT);
  digitalWrite(relayPin, LOW); // Ensure valve is off
      initially
}

void loop() {
  // Open the Solenoid valve (turn on relay)
  digitalWrite(relayPin, HIGH);
  delay(5000); // Keep open for 5 seconds

  // Close the Solenoid valve (turn off relay)
  digitalWrite(relayPin, LOW);
  delay(5000); // Keep closed for 5 seconds
}
```

Introduction to Buzzer

- A **Buzzer** is an audio signaling device.
- **Active Buzzer**: Contains an internal oscillator. It generates a fixed tone as soon as a DC voltage is applied. Simple to use (just ON/OFF).
- **Passive Buzzer**: Does not have an internal oscillator. Requires an AC signal (PWM/square wave) to produce sound. Allows generation of different frequencies (melodies).
- **Applications**: Alarms, user feedback, error notifications.

Feature	**Active Buzzer**	**Passive Buzzer**
Oscillator	Internal	None
Control Signal	DC (High/Low)	AC (PWM/Square Wave)
Output	Fixed Tone	Variable Frequencies
Complexity	Low	Medium

Programming: Buzzer Control

Active Buzzer (Digital Control)

```
int activeBuzzer = 8;
void setup() { pinMode(activeBuzzer, OUTPUT); }
void loop() {
  digitalWrite(activeBuzzer, HIGH); delay(1000);
  digitalWrite(activeBuzzer, LOW); delay(1000);
}
```

Passive Buzzer (PWM / Tone Control)

```
int passiveBuzzer = 9;
void setup() {}
void loop() {
  tone(passiveBuzzer, 1000); // 1 kHz tone
  delay(1000);
  noTone(passiveBuzzer);     // Stop tone
  delay(1000);
}
```

Key Takeaways

- Servo motors use PWM signals ($\sim$ 50Hz) for precise angular positioning and are controlled using the `Servo.h` library.
- Solenoid valves control fluid flow, are inductive loads, and require relays or MOSFETs with flyback diodes for safe interfacing.
- Active buzzers generate a fixed tone with simple DC voltage, while passive buzzers require an oscillating signal (like `tone()`) to produce varying frequencies.

Next Lecture Preview

- In our next lecture, we will begin Unit 4, Topic 4.3: Wi-Fi Based Data Transmission and Control.
- We will learn how to connect the ESP32 to Wi-Fi and transmit sensor data wirelessly.

Agenda

- Introduction to ESP32 Wi-Fi Capabilities
- The `WiFi.h` Library
- Modes of Wi-Fi Operation (STA, AP)
- Connecting ESP32 to Wi-Fi (Station Mode)
- Code Example: Wi-Fi Connection
- Transferring Sensor Data through Wi-Fi
- HTTP GET Requests
- Code Example: Sending Data

Learning Objectives

By the end of this lecture, you will be able to:

- Understand the different Wi-Fi operating modes of the ESP32.
- Use the `WiFi.h` library to connect an ESP32 to a local network.
- Implement code to verify Wi-Fi connection status.
- Transmit simulated or real sensor data over Wi-Fi using HTTP requests.

ESP32 Wi-Fi Capabilities

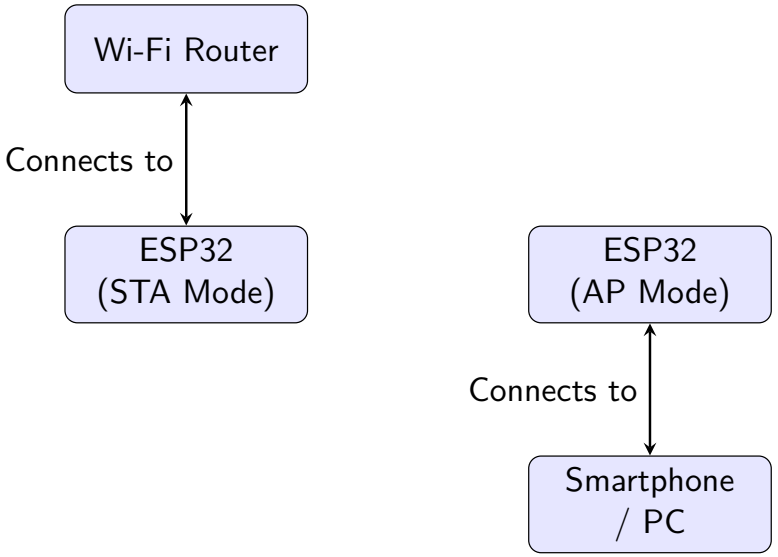
- **Built-in Wi-Fi:** 802.11 b/g/n support (up to 150 Mbps).
- **Frequency:** 2.4 GHz band.
- **Security:** Supports WPA/WPA2/WPA3 personal and enterprise security protocols.
- **Integrated TCP/IP Stack:** Allows the ESP32 to communicate over the internet effortlessly.

The WiFi.h Library

- Core library for Wi-Fi functionalities on ESP32.
- Included by default in the ESP32 Arduino Core.
- Provides classes and methods to manage network connections.
- **Header to include:** `#include <WiFi.h>`
- Replaces the `ESP8266WiFi.h` library used in older ESP8266 boards.

Modes of Wi-Fi Operation

- **Station Mode (STA):** Connects to an existing Wi-Fi router (Access Point). The ESP32 gets an IP address from the router.
- **Access Point Mode (AP):** The ESP32 acts as a router. Other devices (like phones or laptops) can connect directly to the ESP32.
- **STA+AP Mode:** Operates as both simultaneously.



Connecting to Wi-Fi (Station Mode)

Steps to connect:

- Set mode to WIFI_STA: `WiFi.mode(WIFI_STA);`
- Begin connection with SSID and Password: `WiFi.begin(ssid, password);`
- Wait for connection using a while loop checking `WiFi.status()` against `WL_CONNECTED`.
- Print the assigned local IP address: `WiFi.localIP()`.

Code Snippet: Basic Wi-Fi Connection

```
#include <WiFi.h>

const char* ssid = "YOUR_SSID";
const char* password = "YOUR_PASSWORD";

void setup() {
  Serial.begin(115200);
  WiFi.begin(ssid, password);
  Serial.print("Connecting to Wi-Fi");
  while (WiFi.status() != WL_CONNECTED) {
    delay(500);
    Serial.print(".");
  }
  Serial.println("\nConnected!");
  Serial.println(WiFi.localIP());
}
void loop() {}
```

Checking Connection Status

Important `WiFi.status()` return values:

- `WL_CONNECTED`: Assigned when connected to a Wi-Fi network.
- `WL_NO_SHIELD`: Assigned when no Wi-Fi shield is present.
- `WL_IDLE_STATUS`: Temporary status assigned when `WiFi.begin()` is called.
- `WL_CONNECT_FAILED`: When the connection fails (e.g., wrong password).
- `WL_DISCONNECTED`: When disconnected from a network.

Transferring Sensor Data: Overview

- Once connected, the ESP32 can send data over the internet.
- Common protocols for IoT: HTTP/HTTPS, MQTT, WebSockets.
- In this lecture, we use **HTTP GET/POST** requests to send data to a web server (like ThingSpeak or a custom backend).
- The `HTTPClient.h` library simplifies HTTP communications.

HTTP GET Request for Data Transmission

Sending data via URL parameters:

- Structure:
  `http://api.example.com/update?sensor=temperature&value=25.5`
- The `?` starts the query string.
- Key-value pairs are separated by `&`.
- The ESP32 acts as an HTTP Client, sending this URL to the server.

Code Snippet: Sending Data via HTTP

```
#include <HTTPClient.h>
// Inside loop(), after reading sensor data:
if(WiFi.status() == WL_CONNECTED){
  HTTPClient http;
  float temp = 24.5; // Example sensor data
  String url = "http://api.thingspeak.com/update?";
  url += "api_key=YOUR_KEY&field1=" + String(temp);

  http.begin(url);
  int httpResponseCode = http.GET();

  if(httpResponseCode > 0) {
    Serial.print("Data␣Sent,␣Response:␣");
    Serial.println(String(httpResponseCode));
  } else {
    Serial.println("Error␣sending␣data");
  }
  http.end();
}
```

Key Takeaways

- The ESP32's built-in Wi-Fi makes it a powerful IoT device.
- Use `WiFi.h` and `WiFi.begin()` to connect to a network in Station Mode.
- Always verify the connection status using `WiFi.status() != WL_CONNECTED`.
- The `HTTPClient` library allows sending sensor data to web servers using simple GET requests.

Next Lecture Preview

- **Lecture 34: Wi-Fi Based Data Transmission and Control (Part 2)**
- Hosting a web server on the ESP32.
- Controlling ESP32 GPIO pins from a web browser.
- Advanced Wi-Fi features (AutoConnect).

Wi-Fi Based Data Transmission (Part 2)

- Sending Data to Cloud Platforms and Receiving Remote Commands

Agenda

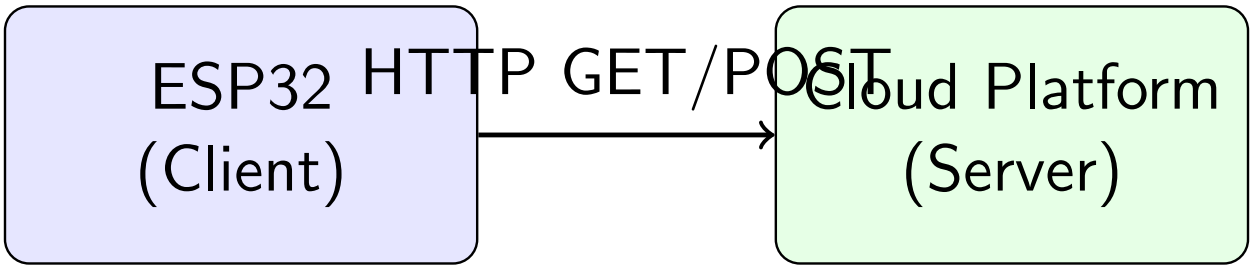
- Cloud Integration Basics
- HTTP Protocol in IoT
- Introduction to HTTPClient Library
- Sending Data via HTTP GET
- Sending Data via HTTP POST (JSON Payload)
- Receiving Remote Commands
- Parsing Responses with ArduinoJson

Learning Objectives

- Understand how ESP32 interacts with cloud platforms using RESTful APIs.
- Learn to construct and execute HTTP GET requests to send query parameters.
- Learn to construct and execute HTTP POST requests with JSON payloads.
- Extract and interpret server responses to trigger local hardware actions.

Cloud Integration Basics

- **IoT Cloud Platforms:** Services like ThingSpeak, AWS IoT, or custom servers that aggregate and visualize IoT data.
- **Role of ESP32:** Acts as an IoT edge device, collecting data from sensors and transmitting it over Wi-Fi.
- **Communication Paradigm:** Client-Server model. The ESP32 is the Client, making requests to the Cloud Server.
- **REST APIs:** Representational State Transfer APIs are standard web endpoints used by platforms to ingest data.



HTTP Protocol in IoT

- **HTTP (Hypertext Transfer Protocol):** The foundation of data communication on the Web.
- **HTTP Methods for IoT:**
- `GET`: Requests data from a specified resource. Also used to send simple data via URL query strings (e.g., `?temp=25`).
- `POST`: Submits data to be processed to a specified resource, typically in the message body (e.g., JSON payload).
- **Response Codes:** `200 OK` indicates success, `404 Not Found`, `500 Server Error`.

Introduction to HTTPClient Library

- **HTTPClient.h:** A built-in ESP32 library that simplifies making HTTP requests.
- **Key Functions:**
- `http.begin(url)`: Initializes the connection to the specified URL.
- `http.addHeader(name, value)`: Adds custom HTTP headers (e.g., Content-Type).
- `http.GET()` / `http.POST(payload)`: Executes the request and returns the HTTP status code.
- `http.getString()`: Retrieves the payload response from the server.
- `http.end()`: Closes the connection to free resources.

Sending Data: HTTP GET Request

- **Mechanism:** Data is appended to the URL as query parameters.
- **Example Format:**
  `http://api.thingspeak.com/update?api_key=XYZ&field1=24.5`
- **Pros:** Extremely simple to implement, easily testable in a web browser.
- **Cons:** Data is visible in the URL, limited payload size, not ideal for complex data structures.

Code Example: HTTP GET

```
#include <WiFi.h>
#include <HTTPClient.h>
void sendDataGET(float temperature) {
  if (WiFi.status() == WL_CONNECTED) {
    HTTPClient http;
    String url = "http://api.example.com/update?";
    url += "temp=" + String(temperature);
    http.begin(url);
    int httpCode = http.GET();
    if (httpCode > 0) {
      String payload = http.getString();
      Serial.println("Response: " + payload);
    }
    http.end();
  }
}
```

Sending Data: HTTP POST Request (JSON)

- **Mechanism:** Data is sent in the body of the HTTP request, leaving the URL clean.
- **Content-Type:** Usually requires setting the header `Content-Type: application/json`.
- **Payload:** A structured JSON string. Example: `{"temperature": 24.5, "humidity": 60}`
- **Pros:** Secure (especially with HTTPS), handles complex and large data structures easily.

Code Example: HTTP POST

```
void sendDataPOST(float temp, float hum) {
  if(WiFi.status() == WL_CONNECTED) {
    HTTPClient http;
    http.begin("http://api.example.com/data");
    http.addHeader("Content-Type", "application/json");
    String jsonPayload = "{\"temperature\":";
    jsonPayload += String(temp) + ",\"humidity\":";
    jsonPayload += String(hum) + "}";
    int httpCode = http.POST(jsonPayload);
    if(httpCode == 200) {
      Serial.println("Data␣POSTed␣successfully");
    }
    http.end();
  }
}
```

Receiving Remote Commands

- **Polling vs Pushing:** ESP32 typically 'polls' a server to get remote commands via HTTP GET.
- **Process:**
- ESP32 sends a GET request to an endpoint (e.g., `/get_status`).
- The server responds with the desired state (e.g., `{"relay1": "ON"}`).
- ESP32 parses the response and updates its GPIO pins accordingly.
- **Limitations:** Polling continuously consumes bandwidth and has latency. For real-time, WebSockets or MQTT are better.

Parsing Responses with ArduinoJson

```
#include <ArduinoJson.h>
// Inside HTTP GET response block:
String payload = http.getString();
StaticJsonDocument<200> doc;
DeserializationError error = deserializeJson(doc, payload);
if (!error) {
  const char* relayState = doc["relay1"];
  if (strcmp(relayState, "ON") == 0) {
    digitalWrite(RELAY_PIN, HIGH);
  } else {
    digitalWrite(RELAY_PIN, LOW);
  }
}
```

Summary (Key Takeaways)

- The HTTPClient library simplifies RESTful communication on the ESP32.
- HTTP GET is useful for simple data transmission via URL parameters.
- HTTP POST with JSON payloads is standard for structured data transfer.
- ESP32 can receive remote commands by polling an API endpoint.
- Libraries like ArduinoJson are used to parse complex server responses.

Next Lecture Preview

- **Topic:** 4.4 MQTT Protocol for IoT
- **Subtopics:**
- Introduction to Publish/Subscribe architecture
- Setting up an MQTT Broker
- Connecting ESP32 to MQTT using PubSubClient library

Agenda

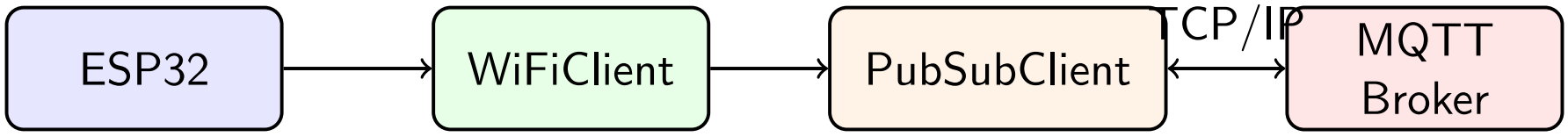
- Learning Objectives
- The PubSubClient Library Overview
- Setting Up Global Variables & Wi-Fi
- Initializing the MQTT Client
- Implementing the Reconnection Logic
- Publishing Messages to a Topic
- Subscribing to a Topic
- Handling Incoming Messages (Callback)
- Summary & Next Lecture Preview

Learning Objectives

- Understand the role of the PubSubClient library in Arduino IDE.
- Write Arduino C++ code to connect the ESP32 to a Wi-Fi network and an MQTT Broker.
- Implement non-blocking reconnection logic for robust IoT operations.
- Develop code to publish sensor data to specific MQTT topics.
- Configure topic subscriptions and process incoming payload data via a callback function.

The PubSubClient Library

- PubSubClient is a popular Arduino library for MQTT messaging.
- Supports MQTT 3.1.1 protocol, which is the industry standard for lightweight IoT.
- Handles the underlying socket connections using the generic `Client` interface (like `WiFiClient`).
- Features include connecting with credentials, publishing payloads, subscribing to topics, and keep-alive pings.



Including Libraries and Global Definitions

```
#include <WiFi.h>
#include <PubSubClient.h>

const char* ssid = "YOUR_SSID";
const char* password = "YOUR_PASSWORD";
const char* mqtt_server = "broker.hivemq.com";

WiFiClient espClient;
PubSubClient client(espClient);
```

- We include `WiFi.h` for network connectivity and `PubSubClient.h` for MQTT.
- `WiFiClient espClient` handles the TCP layer.
- `PubSubClient client(espClient)` binds the MQTT client to our Wi-Fi connection.

Connecting to Wi-Fi

```
void setup_wifi() {
  delay(10);
  WiFi.begin(ssid, password);
  while (WiFi.status() != WL_CONNECTED) {
    delay(500);
  }
}
```

- Before any MQTT communication can occur, the ESP32 must be connected to a Local Area Network with Internet access.
- The `WiFi.begin()` function initiates the connection, and we loop until `WL_CONNECTED` is achieved.

Configuring the MQTT Broker

```
void setup() {
  Serial.begin(115200);
  setup_wifi();
  client.setServer(mqtt_server, 1883);
  client.setCallback(callback);
}
```

- `client.setServer(domain, port)` tells the library where to connect.
- Port 1883 is the default unencrypted MQTT port.
- `client.setCallback(callback)` assigns a function to trigger whenever a message arrives on a subscribed topic.

Port	Usage
1883	Unencrypted MQTT
8883	Encrypted MQTT (TLS/SSL)

The Reconnect Logic (Part 1)

- MQTT connections can drop due to network instability.
- A robust IoT device must automatically attempt to reconnect to the broker.
- The `client.connected()` boolean checks the active session status.
- If disconnected, we use `client.connect(clientID)` to re-establish the link.

The Reconnect Logic (Part 2 - Code)

```
void reconnect() {
  while (!client.connected()) {
    String clientId = "ESP32Client-" + String(random(0xffff)
        , HEX);
    if (client.connect(clientId.c_str())) {
      client.subscribe("esp32/commands");
    } else {
      delay(5000);
    }
  }
}
```

- We generate a random Client ID to avoid collisions.
- Upon successful connection, we re-subscribe to our necessary topics.
- If it fails, we wait 5 seconds before trying again.

Publishing Data via MQTT

```
void loop() {
  if (!client.connected()) {
    reconnect();
  }
  client.loop(); // Keeps connection alive

  // Publish Example
  float temp = 24.5;
  char tempString[8];
  dtostrf(temp, 1, 2, tempString);
  client.publish("esp32/temperature", tempString);
  delay(5000);
}
```

- `client.publish(topic, payload)` sends data to the broker.
- Payloads must be strings or byte arrays (character arrays).

Subscribing to MQTT Topics

- Subscribing allows the ESP32 to receive control commands (e.g., turning on an LED or Relay).
- As seen in the reconnect function: `client.subscribe("esp32/commands");`
- You can subscribe to multiple topics by calling the function multiple times.
- You can also use wildcards like + (single level) or # (multi-level), e.g., `client.subscribe("esp32/+");`

Handling Incoming Messages (Callback)

```
void callback(char* topic, byte* payload, unsigned int
    length) {
  String messageTemp;
  for (int i = 0; i < length; i++) {
    messageTemp += (char)payload[i];
  }

  if (String(topic) == "esp32/commands") {
    if (messageTemp == "on") { digitalWrite(LED_PIN, HIGH);
        }
    else if (messageTemp == "off") { digitalWrite(LED_PIN,
        LOW); }
  }
}
```

- The callback receives the topic name, the byte payload, and the length.
- We convert the byte payload to a String to easily parse commands.

Key Takeaways

- The `PubSubClient` library provides standard MQTT 3.1.1 functionality for the ESP32.
- A robust implementation requires a `reconnect()` loop that automatically recovers lost connections.
- Topic subscriptions must be renewed every time the client re-establishes a connection to the broker.
- `client.loop()` is critical in the `loop()` function to manage keep-alives and incoming packets.
- The `callback` function is used to handle incoming messages asynchronously from subscribed topics.

Next Lecture Preview

- Unit 5: Introduction to Cloud Platforms for IoT
- What is an IoT Cloud Platform?
- Evaluating different Cloud Services (AWS IoT, ThingsBoard, Blynk)
- Integrating MQTT data with cloud dashboards for visualization.

Agenda

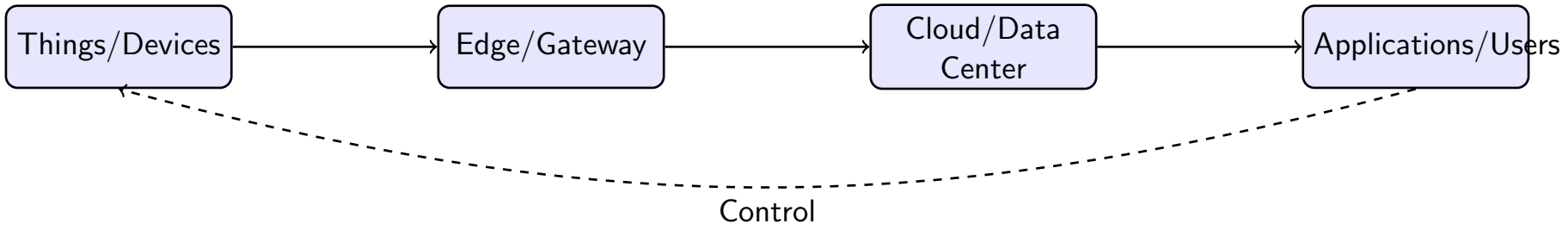
- Overview of End-to-End IoT Architecture
- The Four Main Pillars of IoT Systems
- Layer 1: Perception/Sensing Layer
- Layer 2: Network/Connectivity Layer
- Edge/Fog Computing in the Data Flow
- Layer 3: Cloud/Data Processing Layer
- Layer 4: Application/Actuation Layer
- Complete Data Flow: Sensor to Actuator
- Real-world Example: Smart Irrigation

Learning Objectives

- Understand the fundamental architecture of an end-to-end IoT system.
- Identify the role of sensors, gateways, cloud, and actuators in the data flow.
- Analyze the complete loop of data from generation to actionable physical response.
- Visualize the architecture using block diagrams.

Overview of End-to-End IoT Architecture

- A typical end-to-end IoT architecture is multi-tiered.
- It bridges the physical world (Things) with the digital world (Cloud/Applications).



The Four Main Pillars of IoT Architecture

- **Perception Layer (Sensors/Actuators)**: Interacts directly with the physical environment.
- **Network Layer (Gateways/Connectivity)**: Transmits data between the edge and the cloud.
- **Processing Layer (Cloud/Edge Analytics)**: Stores, analyzes, and makes decisions based on the data.
- **Application Layer (UI/Business Logic)**: Delivers services to users and triggers automated workflows.

Layer 1: Perception/Sensing Layer

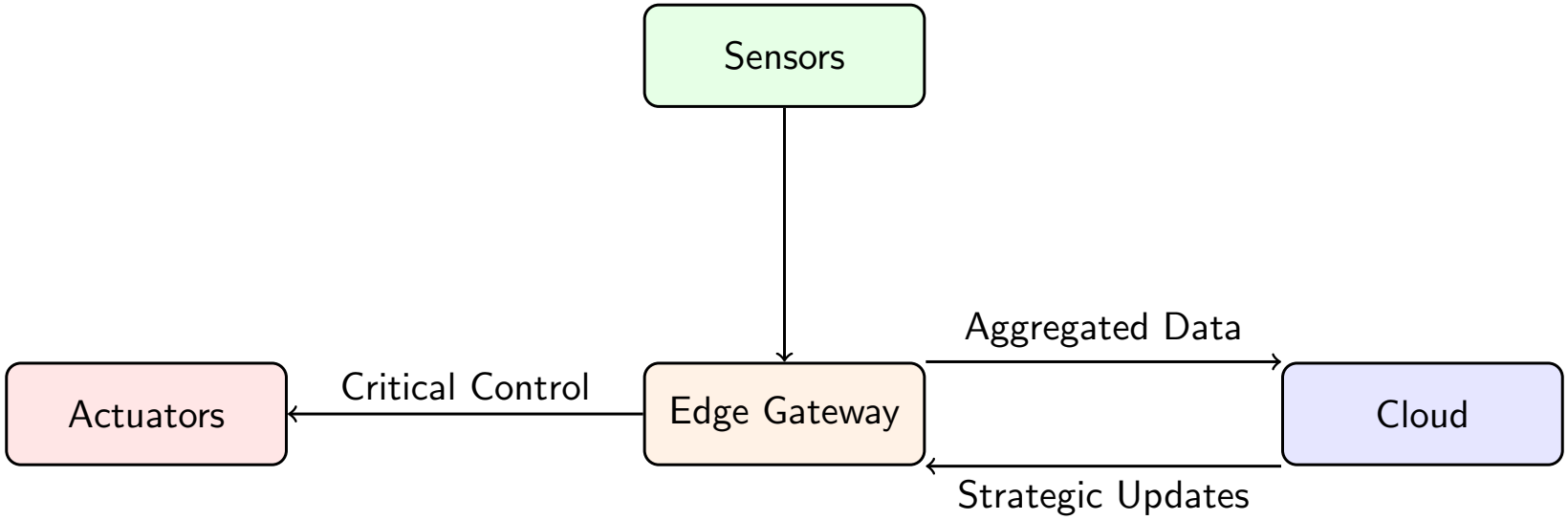
- **Role:** The eyes and ears of the IoT system.
- **Components:**
  - **Sensors:** Convert physical parameters (temperature, motion, light) into electrical signals.
  - **Actuators:** Convert electrical control signals back into physical actions (motors, valves, relays).
  - **Microcontrollers (MCUs):** E.g., ESP32, which reads sensor data and formats it for transmission.
- **Key Challenges:** Power consumption, operating environment, and calibration.

Layer 2: Network/Connectivity Layer

- **Role:** The nervous system routing data from the edge to the cloud.
- **Components:**
  - **Local Connectivity:** Wi-Fi, Bluetooth/BLE, Zigbee, LoRa (connecting sensors to gateways).
  - **Gateways:** Act as intermediaries, performing protocol translation (e.g., Zigbee to IP).
  - **Wide Area Networks (WAN):** Cellular (4G/5G, NB-IoT), Ethernet, routing data to cloud servers.
- **Key Protocols:** MQTT, CoAP, HTTP/REST.

Edge/Fog Computing in the Data Flow

- Not all data needs to go to the cloud.
- Edge computing processes data locally at the gateway level to reduce latency and save bandwidth.



Layer 3: Cloud/Data Processing Layer

- **Role:** The brain of the system for heavy lifting.
- **Components:**
  - **Data Ingestion:** MQTT Brokers (e.g., Mosquitto, AWS IoT Core) handling thousands of connections.
  - **Storage:** Time-series databases (e.g., InfluxDB) for sensor data, relational databases for user data.
  - **Analytics Engine:** Rule engines, machine learning models, and stream processing.
- **Function:** Aggregates massive datasets, detects trends, and evaluates complex rules (e.g., If temperature ¿ 30C for 10 mins, trigger cooling).

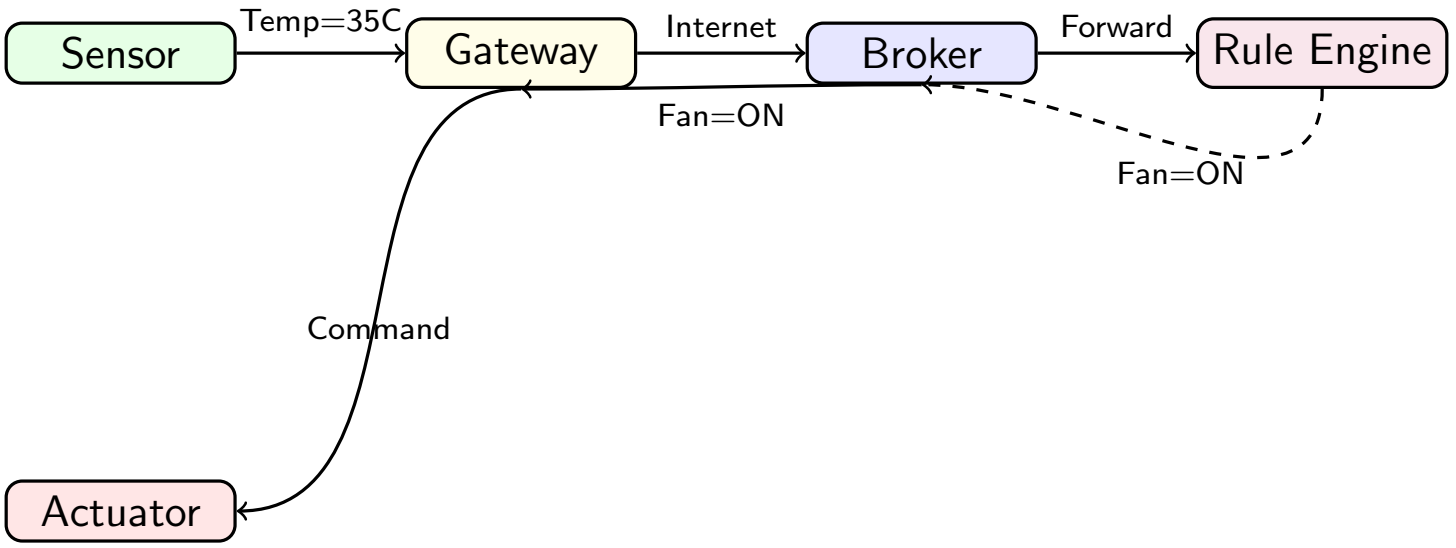
Layer 4: Application/Actuation Layer

- **Role:** The interface for users and automated business processes.
- **Components:**
  - **Dashboards/UIs:** Web apps, mobile apps for real-time monitoring and manual control.
  - **APIs:** Integrations with enterprise systems (ERP, CRM).
  - **Alerting Systems:** SMS, Email, Push notifications.
- **Command Downlink:** The application generates a payload which is routed back down the layers to the specific actuator.

```
{"pump": "ON"}
```

Complete Data Flow: Sensor to Actuator

- Tracing a single event through the entire loop.



Real-world Example: Smart Irrigation System

- **Perception:** Soil moisture sensor detects dry soil. ESP32 reads it, converts to digital (e.g., 20%).
- **Network:** ESP32 connects to farm Wi-Fi, publishes MQTT message.

```
{"moisture": 20}
```

- **Cloud:** AWS IoT Core receives data. Rule Engine checks: If moisture ¡ 30%, trigger pump.
- **Application/Command:** Cloud logs data and publishes command.

```
{"pump_status": 1}
```

- **Actuation:** ESP32 receives message, sets GPIO high, turning on water pump.

Key Takeaways

- IoT architecture consists of Perception, Network, Processing, and Application layers.
- Data flows upstream (telemetry) from sensors to cloud, and downstream (commands) from cloud to actuators.
- Gateways and Edge computing play crucial roles in bridging local and global networks and reducing latency.
- The complete loop connects physical events to automated physical responses.

Next Lecture Preview

- **Topic:** 4.4 End-to-End IoT System Architecture (Part 2)
- **Focus:**
  - Deep dive into specific cloud platforms (AWS IoT, Azure IoT).
  - Security considerations across the entire architecture.
  - Hands-on demo of data flowing from ESP32 to a cloud dashboard.

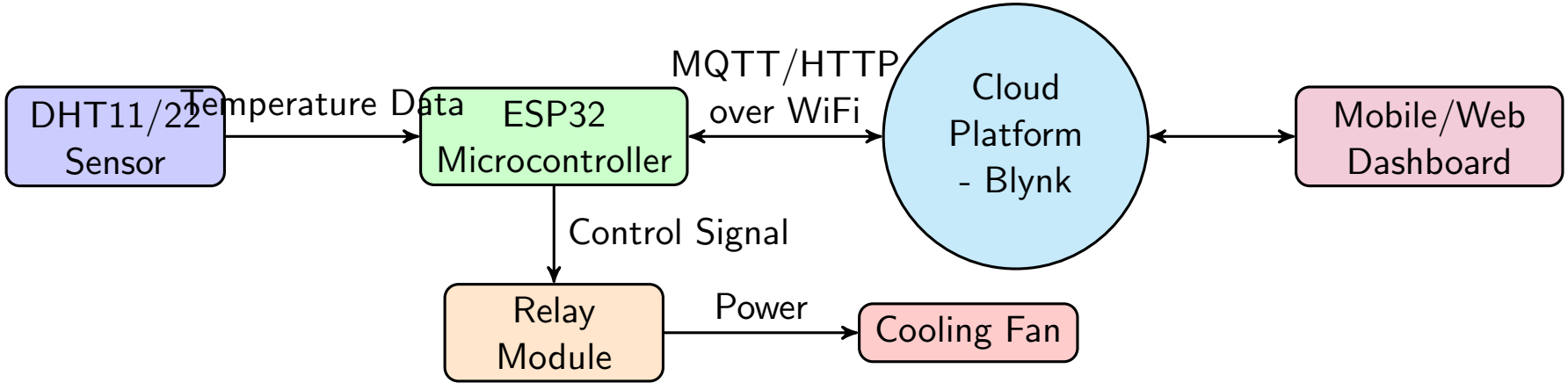
Agenda

- Learning Objectives
- System Architecture Overview
- Hardware Selection & Circuit Wiring
- Cloud Platform Selection
- Firmware: Initialization and Libraries
- Firmware: Sensor Acquisition
- Firmware: Cloud Telemetry
- Firmware: Actuator Control
- System Integration and Testing
- Summary & Next Steps

Learning Objectives

- **Design** an end-to-end IoT system architecture combining telemetry and command/control.
- **Interface** a DHT temperature sensor and a Relay module with the ESP32.
- **Implement** local control logic (Edge computing) to trigger a cooling fan.
- **Integrate** cloud connectivity to monitor environmental data in real-time.

System Architecture Overview



- **Perception Layer:** DHT Sensor (Input)
- **Network/Edge Layer:** ESP32 (Processing & WiFi)
- **Application Layer:** Cloud Dashboard (Monitoring)
- **Actuation:** Relay + Fan (Output)

Hardware Components

- **ESP32 Development Board:** The core processor providing WiFi capabilities and GPIO for hardware interfacing.
- **DHT11 / DHT22 Sensor:** Digital humidity and temperature sensor. DHT22 offers better precision ($\pm 0.5^{\circ}$C) than DHT11 ($\pm 2^{\circ}$C).
- **5V Relay Module:** An electromechanical switch isolated via an optocoupler. Allows the 3.3V ESP32 to safely switch high-power loads.
- **Cooling Fan (12V DC):** The actuator that drives airflow, requiring a separate power supply switched by the relay.

Circuit Wiring & Pinouts

- **DHT Sensor:**
  - VCC $\rightarrow$ 3.3V (with 10k Pull-up if bare sensor)
  - DATA $\rightarrow$ GPIO 4
  - GND $\rightarrow$ GND
- **Relay Module:**
  - VCC $\rightarrow$ 5V (VIN on ESP32 if USB powered)
  - IN $\rightarrow$ GPIO 5
  - GND $\rightarrow$ GND
- **Fan & Power Supply:**
  - 12V (+) $\rightarrow$ Relay COM (Common)
  - Relay NO (Normally Open) $\rightarrow$ Fan (+)
  - Fan (-) $\rightarrow$ 12V GND

Cloud Platform Setup (Blynk IoT)

- **Device Provisioning:** Create a new Template in Blynk. Register the device to obtain the `BLYNK_TEMPLATE_ID`, `BLYNK_DEVICE_NAME`, and `BLYNK_AUTH_TOKEN`.
- **Datastreams:**
  - `V0` (Virtual Pin 0): Float type - For Temperature display.
  - `V1` (Virtual Pin 1): Float type - For Humidity display.
  - `V2` (Virtual Pin 2): Integer type (0/1) - For Fan Status LED/Button.
- **Dashboard:** Drag and drop Guage widgets for V0, V1, and an LED widget for V2.

Firmware: Initialization and Libraries

```
#define BLYNK_TEMPLATE_ID "TMPLxxxx"
#define BLYNK_AUTH_TOKEN "Your_Auth_Token"

#include <WiFi.h>
#include <WiFiClient.h>
#include <BlynkSimpleEsp32.h>
#include <DHT.h>

#define DHTPIN 4
#define DHTTYPE DHT11
#define RELAY_PIN 5

DHT dht(DHTPIN, DHTTYPE);
char ssid[] = "YourNetwork";
char pass[] = "YourPassword";

void setup() {
  Serial.begin(115200);
  pinMode(RELAY_PIN, OUTPUT);
  digitalWrite(RELAY_PIN, LOW); // Fan OFF initially
  dht.begin();
  Blynk.begin(BLYNK_AUTH_TOKEN, ssid, pass);
}
```

Firmware: Sensor Acquisition

```
float readTemperature() {
  float t = dht.readTemperature();
  if (isnan(t)) {
    Serial.println("Failed to read from DHT sensor!");
    return -999.0; // Error value
  }
  Serial.print("Temperature: ");
  Serial.print(t);
  Serial.println(" *C");
  return t;
}
```

- Use `isnan()` to validate sensor readings before processing.
- DHT sensors require at least 2 seconds between reads.

Firmware: Edge Control Logic

```
const float TEMP_THRESHOLD = 30.0; // Celsius
bool fanStatus = false;

void controlFan(float temperature) {
  if (temperature > TEMP_THRESHOLD) {
    digitalWrite(RELAY_PIN, HIGH); // Turn ON Fan
    fanStatus = true;
  } else {
    digitalWrite(RELAY_PIN, LOW);  // Turn OFF Fan
    fanStatus = false;
  }
}
```

- **Edge Computing Benefit:** Even if WiFi disconnects, the ESP32 continues to regulate the temperature locally.

Firmware: Cloud Telemetry Integration

```
BlynkTimer timer;

void sendSensorData() {
  float temp = readTemperature();
  if (temp != -999.0) {
    controlFan(temp);

    // Send data to Blynk Cloud
    Blynk.virtualWrite(V0, temp);
    Blynk.virtualWrite(V2, fanStatus ? 1 : 0);
  }
}

void setup() {
  // ... previous setup code ...
  timer.setInterval(2500L, sendSensorData); // Run every 2.5
      s
}

void loop() {
  Blynk.run();
  timer.run();
}
```

System Integration & Testing

- **Testing Checklist:**
- **Serial Monitor Validation:** Ensure WiFi connects and DHT returns valid floats.
- **Hardware Actuation Test:** Apply a warm object near the DHT sensor. Verify the relay clicks and fan spins when $T > 30^{\circ}C$.
- **Cloud Synchronization:** Open the Blynk app. Verify the gauge updates and the Fan LED widget toggles synchronously with physical actuation.
- **Fail-over Test:** Turn off the WiFi router. Apply heat again. The fan should still turn on, proving edge-resilience.

Key Takeaways

- **Holistic Design:** Integrated hardware (DHT, Relay, ESP32) with a cloud platform (Blynk) via software.
- **Robust Firmware:** Used `isnan()` for data integrity and non-blocking timers (`BlynkTimer`) for network stability.
- **Edge Computing over Cloud Dependence:** Maintained the fan control logic on the microcontroller, ensuring the system remains operational without internet connectivity.
- **Digital Twin Concept:** Represented the physical state of the fan and environment in the cloud via virtual pins.

Next Lecture Preview

- **Next Topic: 4.5 Security in IoT Systems (Part 1)**
- Moving beyond functionality: How do we secure our architecture?
- Understanding IoT threat vectors.
- Introduction to TLS/SSL for secure MQTT/HTTP communications on ESP32.
- Implementing Over-The-Air (OTA) updates securely.
- *Preparation: Read up on basic cryptographic concepts and symmetric vs. asymmetric encryption.*

Smart Motion Detection and Security Alert System

- Unit 4: End-to-End System Architecture
- Part 3: Integrating ESP32, PIR, Buzzer, and MQTT

Agenda

- System Overview and Real-world Applications
- Hardware Components: PIR Sensor & Active Buzzer
- End-to-End System Architecture Diagram
- Circuit Connections & GPIO Mapping
- MQTT Topic Strategy & Payload Structure
- Firmware Implementation: Setup, Loop, and MQTT Callbacks
- System Testing and Edge Cases

Learning Objectives

- Understand the working principle of HC-SR501 PIR motion sensors.
- Design a hybrid local/remote alert system using buzzers and MQTT.
- Implement non-blocking code on ESP32 to handle sensor interrupts and network states.
- Construct an end-to-end telemetry flow from sensor node to cloud dashboard.

System Overview

- Use Case: Intrusion detection for smart homes or restricted industrial areas.
- Local Action: Triggering a high-decibel active buzzer immediately to deter intruders.
- Remote Action: Sending an instant MQTT payload to notify the owner's smartphone or dashboard.
- Control Flow: ESP32 reads PIR state $\rightarrow$ Activates GPIO for Buzzer $\rightarrow$ Publishes JSON to MQTT Broker.

Hardware: HC-SR501 PIR Sensor

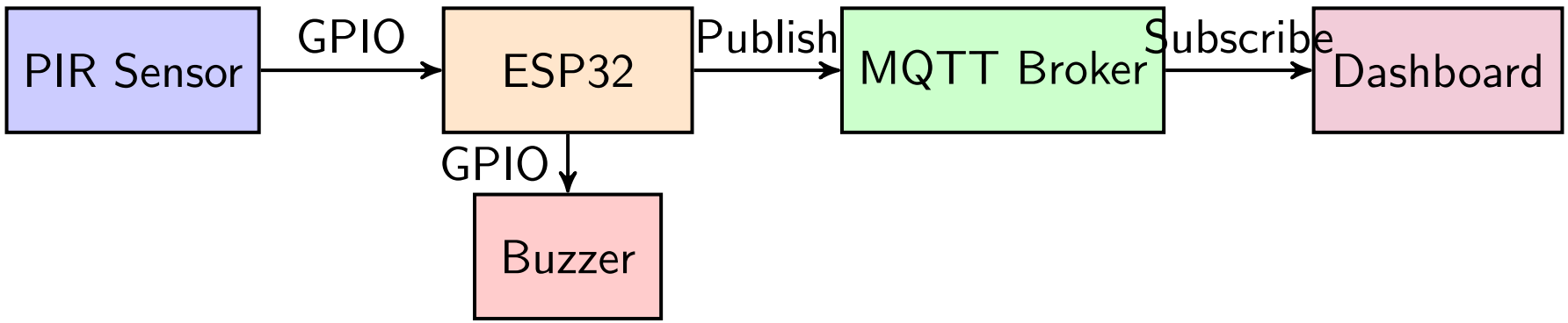
- Operating Principle: Detects changes in infrared radiation emitted by moving bodies.
- Key Pins: VCC (5V), GND, OUT (3.3V Logic High on detection).
- Adjustments: Sensitivity potentiometer (distance) and Time Delay potentiometer (hold time).
- Trigger Modes: Single Trigger (L) vs. Repeatable Trigger (H).

Hardware: Active Buzzer

- Active vs. Passive: Active buzzers have an internal oscillator; they sound simply by applying DC voltage.
- Current Draw: Typical small modules draw 15-30mA, safe for ESP32 GPIOs (which max out at $\sim$40mA), but a transistor is recommended for larger alarms.
- Logic: Driving the GPIO HIGH turns the buzzer ON.

End-to-End System Architecture

- Sensor Node: ESP32 + PIR + Buzzer.
- Network Layer: 2.4GHz Wi-Fi connecting to an MQTT Broker (e.g., Mosquitto on Raspberry Pi or Cloud).
- Application Layer: Node-RED or Home Assistant subscribing to the MQTT broker.
- Diagram Flow: PIR $\rightarrow$ ESP32 $\rightarrow$ Broker $\rightarrow$ Dashboard.



Circuit Connections

- ESP32 to PIR Sensor:
  - PIR VCC $\rightarrow$ ESP32 VIN (5V)
  - PIR GND $\rightarrow$ ESP32 GND
  - PIR OUT $\rightarrow$ ESP32 GPIO 27
- ESP32 to Active Buzzer:
  - Buzzer VCC $\rightarrow$ ESP32 GPIO 26
  - Buzzer GND $\rightarrow$ ESP32 GND

MQTT Topic Strategy

- Publish Topics (ESP32 $\rightarrow$ Broker):
  - `home/security/motion` : Payload `"status": "detected", "timestamp": 169000000`
- Subscribe Topics (Broker $\rightarrow$ ESP32):
  - `home/security/alarm_control` : Payload `ON` or `OFF` to manually trigger/silence buzzer.
- QoS Level: QoS 1 recommended to ensure alert delivery.

Code: Initialization & Setup

- Configure pins and initialize network.

```
#define PIR_PIN 27
#define BUZZER_PIN 26

void setup() {
  pinMode(PIR_PIN, INPUT);
  pinMode(BUZZER_PIN, OUTPUT);
  digitalWrite(BUZZER_PIN, LOW);
  setup_wifi();
  client.setServer(mqtt_server, 1883);
  client.setCallback(callback);
}
```

Code: Motion Detection Logic

- State change detection to avoid spamming the broker.

```
bool motionState = false;

void loop() {
  int pirVal = digitalRead(PIR_PIN);
  if (pirVal == HIGH && !motionState) {
    digitalWrite(BUZZER_PIN, HIGH);
    client.publish("home/security/motion", "{\"alert\":\"
        Motion␣Detected!\"}");
    motionState = true;
  } else if (pirVal == LOW && motionState) {
    digitalWrite(BUZZER_PIN, LOW);
    client.publish("home/security/motion", "{\"alert\":\"
        Clear\"}");
    motionState = false;
  }
  client.loop();
}
```

Code: Remote Alarm Control

- Handling remote silence commands via MQTT.

```
void callback(char* topic, byte* payload, unsigned int
    length) {
  String msg;
  for (int i = 0; i < length; i++) msg += (char)payload[i];

  if (String(topic) == "home/security/alarm_control") {
    if (msg == "OFF") {
      digitalWrite(BUZZER_PIN, LOW);
      // Implement further override logic if required
    }
  }
}
```

Key Takeaways

- Integration of hardware inputs (PIR) and outputs (Buzzer) with cloud connectivity enables hybrid local/remote systems.
- State-tracking logic is essential to prevent network congestion from rapid sensor fluctuations.
- Proper voltage level matching (5V power, 3.3V logic) ensures stable sensor operation.
- Bi-directional MQTT communication allows for remote override and control of local hardware.

Next Lecture Preview

- Unit 5: Introduction to Cloud IoT Platforms
- Connecting our ESP32 systems to AWS IoT Core and ThingsBoard
- Understanding Device Shadows and Digital Twins
- Moving from local Mosquitto brokers to managed cloud infrastructure

Agenda

- Overview of IoT Application Categories
- Consumer IoT (CIoT)
- Commercial IoT
- Industrial IoT (IIoT)
- Infrastructure IoT
- Internet of Military Things (IoMT)

Learning Objectives

- **Classify** IoT applications into distinct broad categories.
- **Understand** the fundamental differences and requirements of each IoT domain.
- **Identify** real-world examples of IoT deployments in consumer, commercial, industrial, infrastructure, and military sectors.
- **Analyze** how IoT addresses specific challenges in different industries.

Overview of IoT Application Categories

- **The IoT Spectrum:** IoT is not a monolithic technology; it serves varied use cases.
- **Primary Categories:**
  - **Consumer IoT (CIoT):** Personal and home use.
  - **Commercial IoT:** Healthcare, transportation, building management.
  - **Industrial IoT (IIoT):** Manufacturing, heavy industry, agriculture.
  - **Infrastructure IoT:** Smart cities, energy grids.
  - **Military IoT (IoMT):** Defense and combat operations.
- **Distinguishing Factors:** Scale, reliability, latency, security, and power requirements differ vastly across categories.

Consumer IoT (CIoT) - Concepts

- **Definition:** Applications designed for everyday consumer use.
- **Focus:** Convenience, lifestyle improvement, home automation, and personal health.
- **Key Characteristics:**
  - High volume of devices.
  - Cost-sensitive market.
  - User-friendly interfaces (often mobile apps).
  - Generally lower reliability and security requirements compared to industrial or military.
  - Heavy reliance on Wi-Fi, Bluetooth, and Zigbee.

Consumer IoT - Examples

- **Smart Home Automation:**
  - Smart thermostats (e.g., Nest) optimizing HVAC energy usage.
  - Smart lighting (e.g., Philips Hue) controllable via voice assistants.
  - Smart appliances (refrigerators, washing machines).
- **Wearable Technology:**
  - Smartwatches (Apple Watch, Fitbit) tracking heart rate, sleep, and activity.
- **Elder Care:**
  - Fall detection wearables and remote monitoring for seniors living alone.

Commercial IoT - Concepts

- **Definition:** IoT applications used in commercial environments to improve business efficiency and customer experience.
- **Focus:** Retail, healthcare, transportation, and building management.
- **Key Characteristics:**
  - Moderate to high scale.
  - Higher security and privacy requirements (e.g., HIPAA for healthcare).
  - Integration with enterprise IT systems (ERP, CRM).
  - Emphasis on operational efficiency and cost reduction.

Commercial IoT - Examples

- **Smart Healthcare (IoMT - Medical):**
  - Continuous glucose monitors sending data to doctors.
  - Asset tracking for wheelchairs and expensive medical equipment.
- **Smart Buildings / Commercial Real Estate:**
  - Automated climate control based on room occupancy.
  - Smart security and access control systems.
- **Retail:**
  - Automated checkout systems (e.g., Amazon Go).
  - Smart shelves monitoring inventory in real-time.

Industrial IoT (IIoT) - Concepts

- **Definition:** The application of IoT technologies in manufacturing, agriculture, and industrial sectors.
- **Focus:** Industry 4.0, automation, predictive maintenance, and supply chain optimization.
- **Key Characteristics:**
  - **Mission-Critical Reliability:** Failures can cause immense financial loss or physical harm.
  - **Strict Latency Requirements:** Real-time control systems.
  - **Robust Security:** Protecting critical industrial infrastructure.
  - Interoperability with legacy OT (Operational Technology) systems.

Industrial IoT - Examples

- **Manufacturing:**
  - **Predictive Maintenance:** Sensors detecting vibration anomalies in motors to predict failures before they happen.
  - Automated guided vehicles (AGVs) on factory floors.
- **Agriculture:**
  - Soil moisture and nutrient sensors optimizing irrigation.
  - Drone-based crop monitoring.
- **Logistics & Supply Chain:**
  - Fleet management with GPS and telematics.
  - Cold chain monitoring for temperature-sensitive pharmaceuticals.

Infrastructure IoT - Concepts & Examples

- **Definition:** IoT applications for monitoring and controlling civil and environmental infrastructure.
- **Focus:** Smart cities, energy grids, and public works.
- **Real-World Examples:**
  - **Smart Grids:** Advanced Metering Infrastructure (AMI) for real-time electricity demand management.
  - **Smart Cities:** Intelligent traffic lights optimizing flow based on congestion data.
  - **Waste Management:** Smart trash cans that signal when they need emptying.
  - **Structural Health Monitoring:** Sensors on bridges and dams monitoring stress and vibrations to prevent collapse.

Internet of Military Things (IoMT)

- **Definition:** The application of IoT technologies for combat operations, warfare, and defense.
- **Focus:** Situational awareness, command and control, and troop safety.
- **Characteristics & Examples:**
  - **Extreme Security & Resilience:** Must withstand jamming and cyberattacks.
  - **Tactical Wearables:** Biometric sensors monitoring soldier health and stress levels in the field.
  - **Autonomous Vehicles & Drones:** Unmanned aerial vehicles (UAVs) for surveillance and strike missions.
  - **Smart Bases:** Automated logistics and perimeter security for military installations.

Summary / Key Takeaways

- IoT applications are broadly categorized by their target domain and operational requirements.
- **Consumer IoT** focuses on convenience (Smart Homes, Wearables) with high volume but lower critical reliability.
- **Commercial IoT** bridges consumer and industrial, focusing on business efficiency (Retail, Healthcare).
- **Industrial IoT (IIoT)** demands mission-critical reliability and low latency for manufacturing and agriculture.
- **Infrastructure IoT** operates at a macro scale (Smart Cities, Grids) for public management.
- **Internet of Military Things (IoMT)** requires extreme security and resilience for defense applications.

Next Lecture Preview

- **Topic:** 5.2 Specific IoT Applications
- **What we will cover:**
  - Deep dive into Home Automation architectures.
  - Detailed analysis of Smart City deployments.
  - Case studies of Smart Grids.
  - Specific implementations in Agriculture and Manufacturing.

Emerging Domains of IoT (Part 1)

- Internet of Battlefield Things (IoBT)
- Internet of Vehicles (IoV)
- Internet of Underwater Things (IoUT)
- Internet of Drones (IoD)

Agenda

- Introduction to Specialized IoT Domains
- Internet of Battlefield Things (IoBT) & Tactical Edge
- Internet of Vehicles (IoV) & V2X Communication
- Internet of Underwater Things (IoUT)
- Internet of Drones (IoD)
- Summary & Next Lecture Preview

Learning Objectives

- Understand the concept and necessity of specialized IoT networks in extreme or highly dynamic environments.
- Analyze the architecture and tactical edge computing in the Internet of Battlefield Things (IoBT).
- Explain V2X communication models within the Internet of Vehicles (IoV).
- Identify the unique acoustic communication challenges in the Internet of Underwater Things (IoUT).
- Describe the use cases and networking aspects of the Internet of Drones (IoD).

Introduction to Emerging IoT Domains

- **Evolution:** IoT has evolved from generic sensor networks to highly domain-specific architectures.
- **Environmental Constraints:** These domains operate in environments with extreme mobility (IoV, IoD), signal attenuation (IoUT), or adversarial threats (IoBT).
- **Customized Protocols:** Standard IoT protocols (like MQTT or CoAP) often require heavy modification or entirely new standards (like DSRC for IoV or acoustic modems for IoUT).
- **Focus:** Today we focus on how IoT adapts to these extreme constraints to deliver mission-critical data.

Internet of Battlefield Things (IoBT)

- **Definition:** An interconnected network of combat equipment, autonomous systems, and human soldiers.
- **Tactical Edge Computing:** Due to limited and frequently contested bandwidth, data must be processed at the 'tactical edge' (near the sensors) rather than in a central cloud.
- **Key Components:** Smart combat gear, autonomous UAVs/UGVs, weapon systems, and command & control (C2) nodes.
- **Objective:** Provide superior situational awareness, predictive logistics, and accelerated decision-making in highly adversarial environments.

IoBT: Challenges and Use Cases

- **Challenges:**
  - Interoperability among legacy and modern defense systems.
  - Electronic Warfare (EW): Susceptibility to jamming and spoofing.
  - SWaP-C Constraints: Strict limits on Size, Weight, Power, and Cost for soldier-carried gear.
- **Use Cases:**
  - **Smart Armor:** Vitals monitoring and automated triage alerts.
  - **Swarm Intelligence:** Drone swarms for reconnaissance and target acquisition.
  - **Predictive Maintenance:** Monitoring vehicle health to prevent breakdowns in combat zones.

Internet of Vehicles (IoV)

- **Definition:** The evolution of Vehicular Ad-hoc Networks (VANETs), integrating vehicles with the internet and urban infrastructure.
- **V2X Communication (Vehicle-to-Everything):**
  - **V2V (Vehicle-to-Vehicle):** Collision avoidance, platoon driving.
  - **V2I (Vehicle-to-Infrastructure):** Traffic light sync, toll collection.
  - **V2P (Vehicle-to-Pedestrian):** Safety alerts for smartphones.
  - **V2N (Vehicle-to-Network):** Cloud services, real-time traffic routing.

IoV: Architecture and Technologies

- **Communication Technologies:**
  - **DSRC (Dedicated Short-Range Communications):** Wi-Fi based (IEEE 802.11p), low latency, short range.
  - **C-V2X (Cellular V2X):** Uses 4G LTE and 5G, offers longer range and higher throughput.
- **Edge Computing in IoV:** Multi-Access Edge Computing (MEC) is deployed at cell towers to process V2X data locally, reducing latency to milliseconds.
- **Challenges:** High mobility causing frequent network topology changes, privacy of location data, and heterogeneous network integration.

Internet of Underwater Things (IoUT)

- **Definition:** A network of smart interconnected underwater objects, often conceptualized as Underwater Wireless Sensor Networks (UWSNs).
- **The Communication Challenge:**
  - RF (Radio Frequency) waves attenuate rapidly in water (only travel a few meters).
  - **Acoustic Communication:** The primary method for IoUT. Sound travels far in water but has very low bandwidth, high latency (speed of sound in water is $\sim$1500 m/s), and high error rates.
  - **Optical Communication:** Used for short-range, high-speed data transfer, but susceptible to turbidity (murky water).

IoUT: Architecture and Applications

- **Architecture:**
  - **Bottom Nodes:** Sensors anchored to the seabed.
  - **Relay Nodes/AUVs:** Autonomous Underwater Vehicles that collect data and move it upwards.
  - **Surface Buoys:** Act as gateways, translating acoustic signals from underwater to RF/Satellite signals to the cloud.
- **Applications:**
  - Marine biology and coral reef monitoring.
  - Offshore oil and gas exploration.
  - Tsunami and earthquake early warning systems.

Internet of Drones (IoD)

- **Definition:** A coordinated architecture tailored for the control, routing, and management of Unmanned Aerial Vehicles (UAVs).
- **FANETs (Flying Ad hoc Networks):** A highly dynamic mesh network formed by drones in mid-air.
- **Airspace Management:** Requires integration with Unmanned Aircraft System Traffic Management (UTM) for collision avoidance and restricted airspace (geofencing) compliance.
- **Communication:** Uses 4G/5G for control and telemetry, and point-to-point RF or optical links for drone-to-drone communication.

IoD: Applications and Challenges

- **Applications:**
  - **Precision Agriculture:** Crop monitoring and targeted pesticide spraying.
  - **Logistics:** Automated last-mile delivery.
  - **Disaster Management:** Establishing temporary communication networks (cell-towers-in-the-sky) and search & rescue.
- **Challenges:**
  - Limited battery life constraining flight time and computation.
  - Line-of-Sight (LoS) blockages in urban canyons.
  - Regulatory hurdles and privacy concerns.

Key Takeaways

- **IoBT:** Focuses on tactical edge computing and resilience in highly adversarial and constrained environments.
- **IoV:** Driven by V2X communication (DSRC/C-V2X) to enable autonomous driving and smart transport infrastructure.
- **IoUT:** Overcomes the limitations of underwater RF by utilizing acoustic communication and surface buoy gateways.
- **IoD:** Utilizes FANETs for 3D mobility, enabling applications like precision agriculture and disaster recovery.

Next Lecture Preview

- **Topic:** 5.2 Emerging Domains of IoT (Part 2)
- **Focus Areas:**
  - Internet of Nano Things (IoNT)
  - Internet of Space Things (IoST)
  - Wearables and Implantable IoT
  - Convergence of AI and Emerging IoT Domains

Emerging Domains of IoT (Part 2)

- Internet of People (IoP)
- Internet of Nano Things (IoNT)
- Internet of Everything (IoE)

Agenda

- Learning Objectives
- Internet of People (IoP)
- Internet of Nano Things (IoNT)
- Internet of Everything (IoE): The Four Pillars
- IoT vs. IoE Comparison
- Key Takeaways
- Next Lecture Preview

Learning Objectives

- Understand the concept of the Internet of People (IoP) and its human-centric approach.
- Explore the architecture and applications of the Internet of Nano Things (IoNT).
- Analyze the Internet of Everything (IoE) framework and its four fundamental pillars.
- Differentiate between IoT and IoE in terms of scope and functionality.

Internet of People (IoP): Introduction

- Definition: A human-centric paradigm where personal devices and individuals form a dynamic network.
- Shift in Focus: Moves from machine-to-machine (M2M) communication to a model where humans are nodes in the network.
- Key Enablers: Smartphones, wearables, and social networks.
- Core Philosophy: Emphasizes privacy, localized data processing, and user control over data.

Internet of People: Applications & Impact

- Social Networking Integration: Devices autonomously share contextual information (e.g., location, activity) based on user consent.
- Healthcare & Well-being: Wearables monitoring vitals and alerting medical professionals or family members.
- Crowdsensing: Aggregating data from multiple users to provide real-time traffic updates or environmental monitoring.
- Impact: Enhances personalized services but raises significant privacy and security challenges.

Internet of Nano Things (IoNT): Introduction

- Definition: The interconnection of nanoscale devices (nanoscale sensors/actuators) with existing communication networks.
- Scale: Devices range from 1 to 100 nanometers in size.
- Communication Protocols: Utilizes terahertz (THz) band frequencies and molecular communication.
- Motivation: Enables sensing and actuation in environments previously inaccessible to macroscopic devices.

IoNT Architecture & Components

- Nano-nodes: The smallest sensing/computing units (e.g., biological or synthetic nanosensors).
- Nano-routers: Aggregate data from nano-nodes and route it to gateways.
- Nano-micro Gateway: Acts as a bridge between the nanoscale network and traditional macroscale networks (like the internet).
- Macro-network: The standard internet or cloud infrastructure where data is stored and analyzed.

IoNT Applications: Healthcare & Industry

- Intrabody Networks: Nanosensors circulating in the bloodstream for early disease detection and targeted drug delivery.
- Environmental Monitoring: Deploying 'smart dust' to detect chemical leaks, pollution, or biohazards at a granular level.
- Advanced Manufacturing: Monitoring the structural integrity of materials at the molecular level.
- Challenges: Biocompatibility, energy harvesting at nanoscale, and security of nano-networks.

Internet of Everything (IoE): Definition & Concept

- Definition: A broad concept coined by Cisco, describing the intelligent connection of People, Process, Data, and Things.
- Philosophy: It is not just about connecting physical devices, but connecting elements in a way that creates unprecedented value and new capabilities.
- Scope: Encompasses IoT but expands it by adding human intelligence, business processes, and data analytics.
- Goal: To turn information into action that creates new capabilities, richer experiences, and unprecedented economic opportunity.

Four Pillars of IoE: People, Process, Data, Things

- People: Connecting people in more relevant, valuable ways (e.g., via wearables, smartphones).
- Process: Delivering the right information to the right person (or machine) at the right time.
- Data: Leveraging data into more useful information for decision making (analytics, AI).
- Things: Physical devices and objects connected to the Internet and each other (the core of IoT).

IoE: Real-world Applications

- Smart Cities: Integrating traffic lights (Things), commuter patterns (Data), emergency routing (Process), and citizens (People) for optimized urban living.
- Modern Healthcare: Wearables (Things) track patient vitals (Data), triggering an alert (Process) to a doctor (People) if anomalies are detected.
- Smart Manufacturing: Sensors predict machine failure (Data/Things), automatically ordering parts (Process), and notifying maintenance staff (People).

Comparing IoT vs. IoE

- Scope: IoT focuses heavily on physical objects (Things). IoE encompasses Things, People, Data, and Process.
- Focus: IoT is about M2M (Machine-to-Machine) communication. IoE includes M2M, M2P (Machine-to-People), and P2P (People-to-People).
- Complexity: IoE is inherently more complex due to the integration of business processes and human behavior.
- Evolution: IoT is considered a subset and the foundational building block of the broader IoE ecosystem.

Key Takeaways

- IoP places humans at the center of the network, leveraging smartphones and wearables for personalized digital experiences.
- IoNT shrinks connectivity down to the nanoscale, utilizing THz frequencies for applications like intrabody networks.
- IoE is the comprehensive integration of People, Process, Data, and Things to drive complex, valuable outcomes.
- Evolution: IoT serves as the hardware and connectivity baseline upon which advanced paradigms like IoE operate.

Next Lecture Preview

- Topic: 5.3 Edge and Fog Computing in IoT
- Introduction to Edge Computing
- Fog Computing Architecture
- Why Cloud Computing is sometimes not enough for IoT
- Use cases for Edge and Fog Computing

Modern IoT Case Studies (Part 1)

- Smart Home Automation System

Agenda

- Introduction to Smart Home Automation
- Core Components of a Smart Home
- Smart Home Architecture
- Communication Protocols
- Sensors and Actuators
- Cloud vs Edge Computing
- Security and Privacy Challenges
- Case Studies: Lighting and HVAC

Learning Objectives

- By the end of this lecture, you will be able to:
- Define what constitutes a smart home automation system.
- Identify the core components and their roles in the system.
- Explain the architectural layers of a smart home network.
- Analyze the security and privacy implications of smart homes.
- Evaluate real-world applications like smart lighting and HVAC.

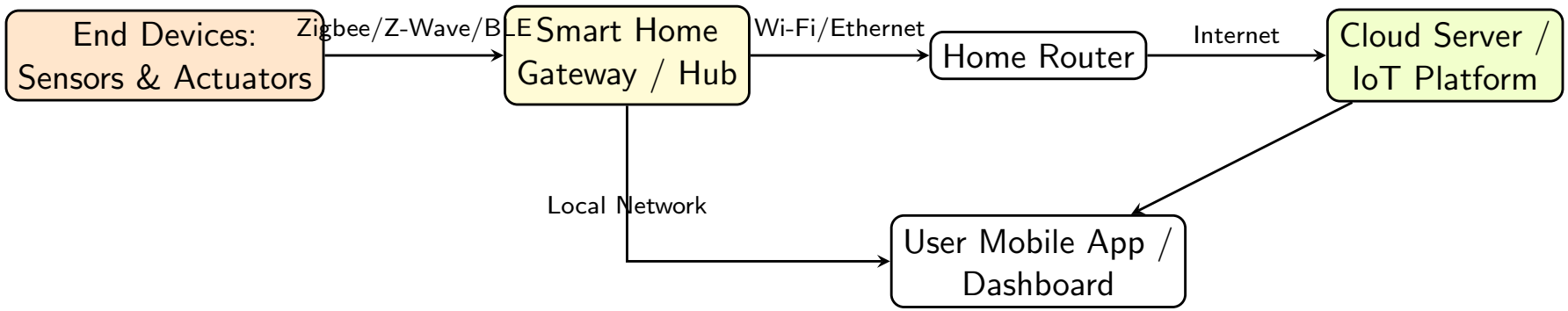
Introduction to Smart Home Automation

- What is a Smart Home?
- A residence equipped with smart devices that automate and control household functions.
- Devices are connected to a central network, allowing remote monitoring and management.
- Primary Goals: Energy efficiency, convenience, security, and comfort.
- Evolution: From simple programmable thermostats to AI-driven predictive environments.

Core Components of a Smart Home

- Perception Layer (Sensors/Actuators): Gathers data and performs actions.
- Network Layer (Gateways/Routers): Connects devices and bridges to the internet.
- Processing Layer (Local Hub/Cloud): Analyzes data and makes decisions.
- Application Layer (User Interfaces): Mobile apps, voice assistants (e.g., Alexa, Google Assistant).

Smart Home Architecture



Communication Protocols in Smart Homes

- Wi-Fi: High bandwidth, high power. Used for cameras and streaming hubs.
- Zigbee / Z-Wave: Mesh networking, low power, standard for sensors and light bulbs.
- Bluetooth Low Energy (BLE): Direct phone-to-device communication, short range.
- Thread / Matter: Emerging IPv6-based mesh protocols aimed at unifying smart home ecosystems.

Sensors and Actuators in Smart Homes

- Sensors (Inputs):
- PIR (Passive Infrared) for motion detection.
- DHT11/DHT22 for temperature and humidity.
- LDR (Light Dependent Resistor) for ambient light.
- Actuators (Outputs):
- Smart Relays for toggling power to appliances.
- Servo/Stepper motors for smart blinds and locks.
- Dimmable LEDs for lighting control.

Cloud vs Edge Computing in Smart Homes

- Cloud Computing:
- Pros: High computational power, accessible anywhere, easy integration with third-party services (e.g., weather APIs).
- Cons: Latency, dependent on internet connection, privacy concerns.
- Edge Computing (Local Hubs):
- Pros: Low latency, works offline, better privacy (data stays local).
- Cons: Limited processing power, harder to access remotely without secure tunnels.

Security and Privacy Challenges

- Security Risks:
- Device hijacking (e.g., Mirai botnet using weak default passwords).
- Man-in-the-Middle (MitM) attacks intercepting unencrypted local traffic.
- Privacy Risks:
- Continuous data collection (audio, video, habits) by big tech companies.
- Mitigation Strategies:
- Use strong, unique passwords and MFA.
- Network segmentation (VLANs for IoT devices).
- End-to-end encryption and regular firmware updates.

Case Study 1: Smart Lighting System

- Scenario: Automated lighting based on occupancy and ambient light.
- Architecture: Motion sensors (BLE/Zigbee) $\rightarrow$ Smart Hub $\rightarrow$ Cloud Rules Engine $\rightarrow$ Smart Bulbs.
- Features:
- Circadian rhythm lighting (color temperature shifts throughout the day).
- Energy savings by ensuring lights are off in empty rooms.
- Technical Challenge: Minimizing latency so lights turn on instantly when someone enters.

Case Study 2: Smart HVAC (Heating, Ventilation, Air Conditioning)

- Scenario: Predictive temperature control using smart thermostats (e.g., Nest).
- Architecture: Multi-room temp sensors + Geofencing via mobile app $\rightarrow$ Cloud AI $\rightarrow$ HVAC Controller.
- Features:
- Learns user schedules to pre-heat or pre-cool the house.
- Geofencing turns off system when everyone leaves.
- Technical Challenge: Integrating with legacy 24V HVAC wiring and managing complex thermal dynamics.

Key Takeaways

- Smart homes rely on a layered architecture: Perception, Network, Processing, and Application.
- Choosing the right protocol (Zigbee, Wi-Fi, Matter) is essential for power and bandwidth management.
- A balance between edge and cloud computing provides the best reliability and features.
- Security and privacy must be designed into the system from the start.
- Real-world applications like lighting and HVAC demonstrate tangible energy and convenience benefits.

Next Lecture Preview

- Topic: Modern IoT Case Studies (Part 2) - Smart Cities & Industrial IoT
- Smart City Infrastructure: Intelligent Traffic Management and Smart Grids.
- Industrial IoT (IIoT): Predictive maintenance and smart manufacturing.
- Scaling IoT solutions from single homes to entire cities.

Agenda

- IoT in Healthcare: Overview
- Health Monitoring System Architecture
- Key Healthcare Sensors
- Cloud & Edge Computing in Health IoT
- Smart City Street Lighting: Introduction
- Smart Lighting Architecture & Sensors
- Energy Efficiency Mechanisms
- Implementation Challenges

Learning Objectives

By the end of this lecture, students will be able to:

- **Analyze** the architectural components of an IoT-based health monitoring system.
- **Identify** specific biosensors used in continuous patient monitoring.
- **Explain** the operational model of a smart city street light monitoring system.
- **Evaluate** the energy-saving mechanisms enabled by adaptive smart lighting.

IoT in Healthcare: Introduction & Significance

The Shift to Connected Care

- **Remote Patient Monitoring (RPM):** Transitioning care from hospital beds to home environments.
- **Continuous Data Acquisition:** 24/7 monitoring of vital signs replacing periodic clinical checks.
- **Preventive Healthcare:** Early detection of anomalies (e.g., cardiac arrhythmias) before they become critical.
- **Resource Optimization:** Reducing hospital readmission rates and optimizing medical staff allocation.

Health Monitoring System: Core Architecture

Multi-Tiered Healthcare IoT Architecture

- **Perception Layer (Body Area Network - BAN):** Wearable or implantable sensors collecting physiological data.
- **Gateway/Edge Layer:** Local aggregation devices (smartphones, dedicated hubs) using BLE or Zigbee to collect BAN data and perform initial filtering.
- **Network Layer:** Wi-Fi, 4G/5G, or LoRaWAN transmitting filtered data to healthcare clouds.
- **Application/Cloud Layer:** Medical databases, AI-driven diagnostic algorithms, and dashboard applications for doctors.

Key Sensors in Health Monitoring

Physiological Data Acquisition

- **Pulse Oximeter (MAX30100/30102):** Measures Heart Rate (BPM) and Blood Oxygen Saturation (SpO2) via photoplethysmography (PPG).
- **ECG Sensor (AD8232):** Single-lead heart rate monitor frontend extracting electrical activity of the heart.
- **Temperature Sensor (DS18B20/LM35):** Medical-grade skin temperature monitoring.
- **Galvanic Skin Response (GSR):** Measures skin conductance, indicating physiological arousal or stress levels.

Edge & Cloud Processing in Health IoT

Data Handling Strategies

- **Edge Processing:**
  - Filtering high-frequency noise (e.g., 50/60Hz AC noise from ECG).
  - Detecting immediate critical thresholds (e.g., heart rate > 150 BPM) for instant local alarms.
- **Cloud Processing:**
  - Long-term trend analysis (e.g., sleep apnea detection over weeks).
  - Machine Learning models predicting deterioration based on multiple fused sensor inputs.
  - EHR (Electronic Health Record) integration.

Smart City Infrastructure: The Role of Smart Lighting

Beyond Basic Illumination

- **Traditional Inefficiencies:** Fixed-schedule or manual operation leads to massive energy waste during low traffic hours.
- **Smart Lighting Benefits:**
  - **Adaptive Dimming:** Adjusting brightness based on ambient light and real-time presence.
  - **Predictive Maintenance:** Automatic reporting of fused bulbs or electrical faults.
  - **Urban Backbone:** Light poles act as real-estate for other smart city sensors (air quality, cameras, 5G small cells).

Smart Street Light System: Architecture

Centralized Control Topology

- **End Nodes (Lamp Posts):** Equipped with LED drivers, microcontrollers (ESP32/STM32), and communication modules.
- **Mesh/Star Networking:** Nodes communicate via RF, Zigbee, or LoRa to a local Street Cabinet Gateway.
- **Gateway:** Aggregates data from hundreds of poles and backhauls via Cellular (LTE-M/NB-IoT) to the central server.
- **Central Management System (CMS):** Dashboard for city administrators to monitor energy usage, schedule lighting, and dispatch repair crews.

Key Sensors in Smart Lighting

Environmental and Operational Sensors

- **LDR (Light Dependent Resistor) / Ambient Light Sensor (BH1750):** Detects dawn/dusk and localized weather darkness (e.g., heavy clouds).
- **PIR (Passive Infrared) / Microwave Radar:** Detects pedestrian or vehicular movement to trigger brightness increases.
- **Current/Voltage Sensors (ACS712 / ZMPT101B):** Monitors power consumption and detects lamp failures (zero current when on).
- **Environmental Sensors (MQ135):** Often added for localized air quality monitoring.

Energy Efficiency & Adaptive Mechanisms

Algorithmic Energy Savings

- **Astronomical Timers:** Base schedules calculated dynamically from GPS coordinates and time of year.
- **Dynamic Dimming Profiles:** e.g., 100% from 7 PM-10 PM, 50% from 10 PM-2 AM, 30% from 2 AM-5 AM (overridden by motion).
- **Neighbor-Node Triggering:** When pole 'A' detects a fast-moving car, it signals poles 'B' and 'C' ahead to illuminate before the car arrives.

Challenges in Modern IoT Systems

Cross-Domain Implementation Hurdles

- **Healthcare Constraints:**
  - **Data Privacy & Security:** Strict compliance (HIPAA, GDPR) required for sensitive medical data.
  - **Reliability:** System failures can be life-threatening.
- **Smart City Constraints:**
  - **Harsh Environments:** Electronics must survive extreme temperatures, rain, and physical vandalism.
  - **Scale and Maintenance:** Deploying firmware updates (OTA) to 50,000 streetlights reliably requires robust rollback mechanisms.

Key Takeaways

- Healthcare IoT utilizes BANs and specialized biosensors (ECG, SpO2) for continuous remote monitoring.
- Edge computing is critical in healthcare for immediate anomaly detection.
- Smart street lighting uses motion and ambient light sensors to dramatically reduce urban energy consumption.
- Lighting networks often use mesh topologies (Zigbee/LoRa) connecting to central cellular gateways.
- Both domains face distinct challenges: privacy in healthcare and environmental durability in smart cities.

Next Lecture Preview

Next Session: 5.4 Industrial IoT (IIoT)

- Difference between IoT and IIoT.
- Industry 4.0 concepts.
- Predictive Maintenance in Manufacturing.
- SCADA vs. IIoT architectures.
- Industrial protocols (Modbus, OPC-UA).

Modern IoT Case Studies (Part 3)

- EV Battery Monitoring System (BMS)
- Cloud Telemetry and Data Ingestion
- SOC and SOH Estimation Algorithms

Agenda

- Introduction to EV Battery Management Systems (BMS)
- Traditional vs. IoT-Enabled BMS
- Core Architecture of an IoT BMS
- Sensor Data Acquisition
- Edge Processing and Vehicle Networks (CAN Bus)
- Cloud Telemetry and Data Ingestion
- State of Charge (SOC) Estimation
- State of Health (SOH) Analysis
- Predictive Maintenance & Alerting

Learning Objectives

- Explain the role and limitations of a traditional offline BMS.
- Map the end-to-end architecture of an IoT-connected EV battery system.
- Describe edge-to-cloud data ingestion protocols used in automotive telemetry.
- Differentiate between SOC and SOH and their calculation methods.
- Design a basic cloud-based predictive maintenance workflow for battery packs.

Introduction to EV Battery Management Systems

- **What is a BMS?**: An electronic system that manages a rechargeable battery pack.
- **Primary Functions**:
  - Protecting the battery from operating outside its safe operating area (SOA).
  - Monitoring its state, calculating secondary data, and reporting.
  - Controlling its environment (thermal management).
  - Authenticating and balancing cells (passive or active balancing).
- **Crucial Role**: Li-ion cells are volatile; overcharging or deep discharging can cause thermal runaway or permanent damage.

Traditional vs. IoT-Enabled BMS

- **Traditional BMS**:
  - Localized processing; acts as an isolated embedded system.
  - Diagnostics require physical connection (e.g., via OBD-II port).
  - Reactive maintenance; alarms trigger only upon threshold breach.
- **IoT-Enabled BMS**:
  - Continuous bidirectional communication via Cellular (4G/5G) or Wi-Fi.
  - Digital Twin creation in the cloud for fleet-wide analytics.
  - Over-The-Air (OTA) updates to refine charging algorithms.
  - Predictive maintenance using cloud-side machine learning.

Core Architecture of an IoT BMS

- **Perception Layer (Edge)**: Current sensors (shunts, Hall effect), NTC thermistors, Voltage tap wires on individual cells.
- **Processing Layer (Edge)**: Local BMS Microcontroller (e.g., TI BQ series or NXP battery controllers).
- **Network Layer**: In-vehicle CAN (Controller Area Network) bus linked to an IoT Telematics Gateway.
- **Cloud/Platform Layer**: MQTT Broker (AWS IoT Core, Azure IoT Hub), Stream processing, Time-series databases (InfluxDB).
- **Application Layer**: Fleet management dashboards, mobile apps for the EV owner.

Sensor Data Acquisition at the Edge

- **Voltage Monitoring**: Measuring individual cell voltages (typically 3.0V - 4.2V for Li-ion) to ensure no cell drops below minimum.
- **Current Measurement**: Using precision shunt resistors or Hall-effect sensors to track the instantaneous current flow (Charge/Discharge).
- **Temperature Sensing**: NTC (Negative Temperature Coefficient) thermistors placed across modules to detect localized hot spots.
- **Sampling Rates**: Voltage and current are sampled at high frequencies (e.g., 10-100 Hz) for accurate safety triggers, while telemetry might aggregate this to 1 Hz.

Edge Processing and Vehicle Networks

- **Real-Time OS (RTOS)**: BMS microcontrollers run RTOS to guarantee deterministic response times for safety disconnects (contactors).
- **CAN Bus Integration**: The BMS broadcasts aggregated battery states onto the vehicle's CAN bus (standard ISO 11898).
- **Telematics Control Unit (TCU)**:
  - Acts as the IoT Edge Gateway.
  - Subscribes to specific CAN IDs containing battery data.
  - Formats raw CAN frames into JSON or Protocol Buffers.
  - Encrypts payloads using TLS 1.2/1.3 before cellular transmission.

Cloud Telemetry and Data Ingestion

- **Protocol**: MQTT (Message Queuing Telemetry Transport) over Cellular (LTE-M / NB-IoT / 4G).
- **Payload Optimization**: Due to bandwidth costs, data is often batched and compressed (e.g., using Protobuf instead of plain JSON).
- **Data Pipeline**:
  - AWS IoT Core / Azure IoT Hub (Ingestion).
  - Stream Analytics (Apache Kafka, AWS Kinesis) for real-time anomaly detection.
  - Time-Series Database (InfluxDB, AWS Timestream) for efficient storage of timestamped telemetry.
- **Digital Twin**: Real-time representation of the physical battery pack in the cloud ecosystem.

State of Charge (SOC) Estimation

- **Definition**: The equivalent of a fuel gauge; represents the remaining capacity of the battery as a percentage.
- **Method 1: Open Circuit Voltage (OCV)**: Relies on the steady-state voltage, but inaccurate under load.
- **Method 2: Coulomb Counting (Ampere-hour integration)**: Integrates the current over time. Prone to cumulative sensor drift.
- **Method 3: Extended Kalman Filter (EKF)**:
  - Advanced mathematical algorithm running on edge and cloud.
  - Fuses OCV, current integration, and temperature data.
  - Continuously corrects itself to provide highly accurate SOC.

State of Health (SOH) and Degradation

- **Definition**: A figure of merit representing the battery's condition relative to its ideal, factory-new condition.
- **Indicators of Degradation**:
  - **Capacity Fade**: The maximum energy the battery can hold decreases over cycles.
  - **Power Fade**: The internal resistance (IR) increases, reducing peak power output.
- **Cloud-Side SOH Calculation**:
  - Requires historical data analysis (months or years).
  - Machine Learning models in the cloud analyze charge/discharge curves to predict nonlinear degradation.
  - Influences warranty claims and second-life battery viability.

Predictive Maintenance & Alerting

- **Proactive vs. Reactive**: Identifying a failing cell before it causes a pack failure or thermal event.
- **Cloud-Based Anomaly Detection**:
  - Monitoring delta-V (voltage difference) between adjacent cells over time.
  - Detecting unusual temperature spikes during standard fast-charging sessions.
- **Alerting Workflows**:
  - **Critical Alert**: Push notification to user ('Stop driving, seek service').
  - **Degradation Alert**: Automated ticket to service center to replace a specific module under warranty.
  - **Software Mitigation**: Pushing an OTA update to lower the max charging rate for a degraded pack.

Key Takeaways

- An IoT-enabled BMS combines edge safety controls with cloud-based analytics.
- Telematics units map CAN bus data to secure, IP-based MQTT streams.
- SOC (State of Charge) acts as the fuel gauge, often requiring complex Kalman filtering.
- SOH (State of Health) tracks long-term degradation like capacity fade and internal resistance.
- Cloud telemetry enables predictive maintenance, anomaly detection, and OTA updates, greatly extending EV lifecycle.

Next Lecture Preview

- **Topic: 5.4 Modern IoT Case Studies (Part 4)**
- Smart Healthcare and Wearable IoT Devices
- Continuous patient monitoring.
- Data privacy and HIPAA compliance in IoT.
- Edge ML for biosignal analysis.

Modern IoT Case Studies: Waste Management

- Course: Internet of Things
- Unit 5:
- Lecture: Modern IoT Case Studies: Waste Management

Agenda

- Traditional Waste Management Challenges
- Concept of IoT-Based Smart Waste Management
- System Architecture
- Sensors in Smart Bins
- Communication Protocols
- Dynamic Route Optimization
- Key Takeaways
- Full Course Recap

Learning Objectives

- Understand the inefficiencies in traditional solid waste collection.
- Explain the hardware and software architecture of an IoT waste management system.
- Identify the sensors used to monitor waste levels and bin status.
- Analyze how route optimization algorithms reduce fuel consumption and operational costs.
- Reflect on the key concepts learned throughout the IoT course.

Challenges in Traditional Waste Management

- Fixed Collection Schedules: Trucks visit bins regardless of whether they are full or empty.
- Overflowing Bins: Unpredictable waste generation leads to overflowing bins, causing health hazards.
- High Operational Costs: Wasted fuel and excessive wear on vehicles due to inefficient routes.
- Lack of Data: Municipalities have no real-time insights into waste generation patterns.

IoT-Based Smart Waste Management

- Core Concept: Equipping trash bins with sensors to monitor fill levels in real-time.
- Data-Driven Decisions: Collection schedules are generated dynamically based on bin status.
- Smart Bins: Containers with integrated IoT nodes (microcontroller + sensors + transceiver).
- Goal: Optimize logistics, reduce carbon footprint, and improve urban sanitation.

System Architecture

- Perception Layer: Ultrasonic sensors, load cells, and temperature sensors inside the bin.
- Network Layer: Low-power Wide Area Networks (LoRaWAN, NB-IoT) transmitting data to gateways.
- Middleware Layer: Cloud servers ingesting and storing telemetry data.
- Application Layer: Web/Mobile dashboards for municipal workers and route optimization engines.

Sensors Used in Smart Bins

- Ultrasonic Sensors (e.g., HC-SR04): Mounted on the lid facing down to measure the distance to the trash surface, calculating fill percentage.
- Load Cells (Weight Sensors): Placed at the base to measure the total weight of the waste.
- Gas Sensors (e.g., MQ-135): Detect foul odors or harmful gases (methane) indicating decomposing organic waste.
- Temperature & Flame Sensors: Detect potential fires inside the bin.

Connectivity & Energy Management

- Protocol Choice: LoRaWAN or NB-IoT are preferred over Wi-Fi due to long range and low power requirements.
- Power Source: Battery-operated nodes, often supplemented by small solar panels on the bin lid.
- Sleep Modes: The ESP32 or STM32 microcontroller remains in deep sleep, waking up every 30-60 minutes to take a reading and transmit.
- Payload Optimization: Sending only a few bytes (Bin ID, Fill %, Battery Level) minimizes energy use.

Cloud Platform & Analytics

- Data Ingestion: MQTT brokers receive data from LoRa gateways.
- Predictive Analytics: Machine learning models predict when a bin will reach 100% based on historical generation rates.
- Geospatial Mapping: Visualizing bin locations and statuses (Green = Empty, Red = Full) on a map.
- Alerting System: Automated SMS or push notifications for fire detection or rapid overflow.

Dynamic Route Optimization

- The Traveling Salesperson Problem (TSP): Applied to waste collection.
- Dynamic Routing: Algorithms (like Dijkstra's or A*) calculate the most efficient path connecting only bins that are ¿80% full.
- Traffic Integration: Factoring in real-time traffic data to avoid congestion.
- Driver Navigation: Providing turn-by-turn navigation to the truck driver via a mobile tablet.

Benefits of Smart Waste Management

- Cost Reduction: Up to 30-50% savings in fuel and operational costs.
- Environmental Impact: Lower CO2 emissions from garbage trucks.
- Improved Hygiene: Elimination of overflowing bins and associated pests/odors.
- Resource Allocation: Municipalities can allocate staff and vehicles more efficiently.

Challenges & Future Scope

- High Initial Cost: Retrofitting thousands of bins with sensors requires significant capital.
- Vandalism and Theft: Sensors placed in public bins are susceptible to damage or theft.
- Maintenance: Batteries need replacement, and sensors require periodic cleaning.
- Future Scope: Automated robotic sorting of waste, integration with smart city grids, and automated garbage collection vehicles.

Key Takeaways: Waste Management

- Smart waste management shifts collection from fixed schedules to data-driven dynamic routes.
- Ultrasonic sensors and LoRaWAN form the backbone of edge bin nodes.
- Cloud analytics and route optimization algorithms are critical for realizing cost and environmental benefits.
- Ruggedization and power management are key hardware challenges.

Full Course Recap

- Unit 1: Fundamentals of IoT, architecture, and impact on modern society.
- Unit 2: Hardware platforms (ESP32), sensors, actuators, and interfacing techniques.
- Unit 3: Communication protocols (MQTT, CoAP, HTTP) and LPWAN technologies.
- Unit 4: Cloud platforms (AWS, ThingsBoard), Edge computing, and data analytics.
- Unit 5: Real-world Case Studies (Smart Home, Agriculture, Industrial IoT, Waste Management).
- Congratulations on completing the Internet of Things course!