Introduction to IoT, IIIT Hyderabad, spring 2025

Prompt Engineered by Suresh Purini
ChatGPT Generated

Please check for the correctness of the material.

Notes is provided for big picture and important discussion points.

March 10, 2025

Contents

Chapter 1

Introduction to IoT Using Case Studies

1.1 What is an IoT System?

An Internet of Things (IoT) system is an interconnected network of **THINGS**. A **THING** is any device or object that can:

- **Sense**: Collect data from its surroundings.
- **Process**: Analyze and interpret the data.
- **Act**: Perform an action based on the processed data.
- **Communicate**: Exchange information with other **THINGS** (using e.g., Wi-Fi, LoRaWAN, Cellular).

Together, these **THINGS** collaborate to achieve a shared objective, making systems smarter, more efficient, and capable of addressing complex challenges.

IoT Network Architectures

There are multiple ways IoT devices can be connected to form a network, depending on the system's design and requirements:

1. **Peer-to-Peer Network**: Devices communicate directly with each other without relying on centralized nodes.
2. **Star Network**: All devices are connected to a central hub, which manages communication and data exchange.
3. **Hierarchical Network (Cloud-Fog-Edge)**: Combines different levels of processing and storage:
    - **Cloud**: Centralized, large-scale analytics and storage.
    - **Fog**: Localized processing close to the devices.
    - **Edge**: Real-time processing directly on the devices.

1.2 Case Studies

To illustrate the core components and their roles, we explore four real-world IoT applications: ride-hailing services, smart road traffic monitoring, smart water meters, and sports vests.

1.2.1 Case Study 1: Ride-Hailing Services (Uber/Ola)

- **Overview**: Ride-hailing platforms like Uber and Ola use IoT to connect drivers and passengers efficiently.
- **Key Components**:
  - **Sensors**: GPS (for location tracking), accelerometer (driver behavior monitoring).
  - **Communication Network**: Cellular (4G/5G) for real-time updates.
  - **Processing**:
    - **Cloud**: Dynamic pricing, route optimization, and driver-passenger matching.
    - **Edge**: Device-based location updates and basic navigation.
- **Considerations**:
  - **Latency**: Minimal delay is critical for route updates.
  - **Privacy**: Protecting sensitive data like location and payment details.
  - **Scalability**: Handling millions of users simultaneously.

1.2.2 Case Study 2: Smart Road Traffic Monitoring (Including Air Pollution)

- **Overview**: IoT-based systems monitor road traffic and air quality to improve urban planning and reduce pollution.
- **Key Components**:
  - **Sensors**:
    - Traffic Cameras: For vehicle detection and traffic density estimation.
    - Air Quality Sensors: Measure pollutants such as PM2.5 and NOx.
  - **Communication Network**:
    - LoRaWAN for low-power, wide-area air quality data transfer.
    - Cellular/Wi-Fi for high-bandwidth traffic camera feeds.
  - **Processing**:
    - **Edge**: Basic object detection and filtering in smart cameras.
    - **Cloud**: Aggregated traffic analysis and pollution mapping.
- **Considerations**:

  - **Bandwidth**: High data transmission requirements for video.
  - **Integration**: Combining traffic and pollution data.
  - **Power Efficiency**: Optimizing energy use for remote air quality sensors.

1.2.3 Case Study 3: Smart Water Meters

- **Overview**: Smart water meters automate the monitoring and management of water usage.
- **Key Components**:
  - **Sensors**: Flow sensors for water usage measurement.
  - **Communication Network**: Narrowband IoT (NB-IoT) or LoRaWAN for low-power communication.
  - **Processing**:
    * **Edge**: Basic anomaly detection, such as identifying leaks.
    * **Cloud**: Long-term storage for billing and trend analysis.
- **Considerations**:
  - **Power Efficiency**: Ensuring battery longevity in remote devices.
  - **Data Volume**: Optimizing periodic updates for bandwidth efficiency.
  - **Real-Time Alerts**: Detecting and addressing leaks or unusual usage patterns.

1.2.4 Case Study 4: Sports Vests

- **Overview**: IoT-enabled sports vests provide real-time monitoring of athletes' health and performance.
- **Key Components**:
  - **Sensors**: Heart rate monitors, GPS, accelerometers.
  - **Communication Network**: Bluetooth/Wi-Fi for close-range connectivity.
  - **Processing**:
    * **Edge**: Real-time heart rate and motion tracking.
    * **Cloud**: Long-term performance analysis and training optimization.
- **Considerations**:
  - **Real-Time Feedback**: Vital for in-game monitoring.
  - **Wearability**: Lightweight and non-intrusive design.
  - **Data Fusion**: Integrating multiple sensor inputs for meaningful insights.

1.3 Comparative Analysis of Case Studies

Aspect	Ride-Hailing Services	Smart Road Traffic Monitoring	Smart Water Meters	Sports Vests
Sensors	GPS, Accelerometer	Traffic Cameras, Air Sensors	Flow Sensors	Heart Rate, GPS
Communication Network	Cellular (4G/5G)	LoRaWAN, Cellular, Wi-Fi	NB-IoT, LoRaWAN	Bluetooth, Wi-Fi
Edge Processing	Location Updates	Object Detection	Leak Detection	Real-Time Monitoring
Cloud Processing	Route Optimization	Traffic Analysis, Pollution Maps	Billing, Trend Analysis	Training Recommendations
Key Considerations	Latency, Privacy	Bandwidth, Power Efficiency	Power Efficiency, Alerts	Wearability, Data Fusion

1.4 IoT Design Considerations

Designing effective IoT systems requires careful evaluation of the components and architecture, as well as the parameters impacting performance. The design process can be divided into two parts: components and parameters.

1.4.1 IoT Components

- **Sensors/Actuators**:
  - Collect data from the environment (e.g., GPS, flow sensors) or act upon the system (e.g., valves, motors).
  - Require optimization for power, durability, and accuracy.
- **Edge vs. Cloud Processing**:
  - **Edge Processing**:
    - Ideal for real-time tasks and latency-sensitive operations.
    - Reduces bandwidth requirements by filtering and processing data locally.
  - **Cloud Processing**:
    - Suitable for large-scale analytics and storage.
    - Supports centralized control and integration of complex models.
- **Communication Networks**:
  - Enable data transfer between IoT devices and the processing systems.
  - Choice depends on bandwidth needs, power constraints, and range requirements (e.g., LoRaWAN, 5G).
- **Security and Privacy**:

  - IoT devices must implement encryption, secure boot, and robust protocols to protect against breaches.
  - Centralized data (in the cloud) must follow compliance regulations (e.g., GDPR).

- **Software Stack**:
  - Includes on-device software (lightweight OS like FreeRTOS), middleware (for communication and device management), and cloud platforms (for analytics and storage).

1.4.2 Key Parameters for IoT Design

- **Processing Capability**:
  - Assess the computational needs of the application.
  - High-end applications (e.g., video analytics) require more powerful edge devices and robust cloud infrastructure.
- **Communication Requirements**:
  - Data volume and bandwidth determine the choice of communication protocol.
  - For instance, LoRaWAN is suitable for low-data applications, while 5G handles high-bandwidth needs.
- **Real-Time vs Non-Real-Time**:
  - Real-time applications (e.g., autonomous vehicles) demand ultra-low latency.
  - Non-real-time systems (e.g., smart water meters) prioritize power and cost efficiency.
- **Power Efficiency**:
  - Crucial for battery-powered IoT devices.
  - Includes considerations for sleep modes, energy-efficient communication protocols, and optimized hardware.
- **Storage**:
  - Edge devices may have limited local storage for real-time operations.
  - Cloud platforms offer scalable storage but require efficient data synchronization.

1.5 Key Takeaways

- **IoT Applications** span various domains, from transportation to health and environmental monitoring.
- **Cloud vs. Edge**: Choosing the right processing architecture depends on latency, bandwidth, and power requirements.
- **Design Considerations**: Effective IoT systems must balance efficiency, scalability, and privacy.

1.6 Discussion Questions

1. What are the trade-offs between processing data on the edge vs. on the cloud in IoT?
2. How can IoT systems ensure scalability and security in large-scale applications?
3. Propose another application of IoT and identify its core components and considerations.

Chapter 2

Microcontrollers

2.1 Why Do We Need a Microcontroller? Why Not Just Use a Microprocessor?

2.1.1 Microcontroller vs. Microprocessor

- **Microcontroller (MCU):** A microcontroller is a compact, self-contained system that includes a **CPU, memory (RAM and flash), and peripherals (I/O ports, timers, ADCs, etc.)** all on a single chip. It is designed specifically for **embedded systems** where a specific task or set of tasks needs to be performed efficiently, often in real-time. *Examples: Arduino, STM32.*

- **Microprocessor (MPU):** A microprocessor is primarily a **CPU** and relies on external components (RAM, ROM, I/O devices, etc.) to function. It is more suited for **general-purpose computing**, such as in computers or complex systems requiring high processing power. *Examples: Intel Core i7, ARM Cortex-A processors.*

2.1.2 Why Do We Need a Microcontroller?

1. **Cost-Effectiveness:** Microcontrollers are cheaper since they integrate everything needed for control tasks on a single chip. In contrast, microprocessors require additional components like external RAM, which increases cost and complexity.

2. **Compact and Power-Efficient:** MCUs are smaller and consume less power, making them ideal for **low-power, battery-operated devices** like remote controls or IoT devices.

3. **Real-Time Control:** Many embedded applications, like controlling motors or reading sensor data, require real-time performance. MCUs have **dedicated hardware (like timers and ADCs)** for these tasks, which microprocessors lack without additional components.

4. **Ease of Use:** MCUs are easier to set up for small, dedicated tasks since they do not require an external memory or peripheral interface.

2.1.3 Why Not Just Use a Microprocessor?

- **External Dependencies:** An MPU needs external RAM, ROM, and peripherals, making the system more **complex to design, build, and debug**.
- **Higher Power Consumption:** MPUs consume more power and are not ideal for portable or low-power applications.
- **Overkill for Simple Tasks:** Using a microprocessor for simple tasks like blinking an LED or reading a sensor is like using a bulldozer to plant a flower—**unnecessary and inefficient**.

2.1.4 When to Use a Microprocessor

Microprocessors are better suited for complex tasks requiring high computational power, such as:

- Running operating systems
- Performing heavy data processing
- Multitasking complex applications

2.1.5 Conclusion

We use microcontrollers in embedded systems because they are **cost-effective, compact, energy-efficient**, and tailored for specific tasks. In contrast, microprocessors are better suited for general-purpose, high-performance computing.

2.2 Introduction to Microcontrollers

A **microcontroller** is an integrated circuit designed for specific tasks by combining a processor, memory, and input/output peripherals on a single chip. It is commonly used in embedded systems for real-time applications.

2.3 Case Study: ESP32

Overview

The **ESP32** is a powerful, low-cost microcontroller developed by **Espressif Systems**. It is widely used in IoT and wireless applications due to its integrated Wi-Fi and Bluetooth capabilities.

Technical Specifications

Feature	Details
Processor	Dual-core Xtensa LX6, 240 MHz
Voltage	3.3V
Flash Memory	4 MB (varies by model)
RAM	520 KB SRAM
Wi-Fi	802.11 b/g/n
Bluetooth	v4.2 BR/EDR + BLE
GPIO Pins	34 (with ADC, PWM, SPI, I2C, etc.)

Variants in the ESP32 Family

Variant	Flash Memory	Additional Features
ESP32-WROOM	4 MB	Standard module
ESP32-S2	4 MB	Single-core, USB support
ESP32-C3	4 MB	RISC-V core, low power
ESP32-S3	16 MB	AI acceleration, USB OTG

Operating System on ESP32

- **Default OS: FreeRTOS**, pre-installed as part of the ESP-IDF SDK.
- **Other OS Options:**
  - **MicroPython:** Python-based scripting.
  - **Zephyr OS:** Open-source RTOS for IoT.
  - **Amazon FreeRTOS:** AWS integration for cloud IoT solutions.

2.4 Case Study: Arduino

Overview

Arduino is an open-source hardware and software platform designed for beginners and hobbyists. It offers a variety of microcontroller boards with an easy-to-use IDE.

Technical Specifications (Arduino Uno as Example)

Feature	Details
Processor	ATmega328P, 16 MHz
Voltage	5V (operates on 5V and 3.3V)
Flash Memory	32 KB
RAM	2 KB SRAM
EEPROM	1 KB
GPIO Pins	14 (6 PWM, 6 analog)

Variants in the Arduino Family

Variant	Processor	Voltage	Flash Memory	Special Features
Arduino Uno	ATmega328P	5V	32 KB	Basic development board
Arduino Mega	ATmega2560	5V	256 KB	More I/O pins (54 digital)
Arduino Nano	ATmega328P	5V	32 KB	Compact form factor
Arduino Due	SAM3X8E (ARM)	3.3V	512 KB	32-bit processor

Operating System on Arduino

- **Default:** No operating system (bare-metal programming).
- **Optional RTOS:**
  - **FreeRTOS:** Adds multitasking capabilities.
  - **ChibiOS:** A lightweight RTOS with Arduino support.

2.5 Case Study: Particle Photon

Overview

The **Particle Photon** is a Wi-Fi-enabled microcontroller optimized for IoT applications. It is part of the Particle ecosystem, which provides seamless cloud integration and a beginner-friendly environment.

Technical Specifications

Feature	Details
Processor	ARM Cortex-M3, 120 MHz
Voltage	3.3V
Flash Memory	1 MB
RAM	128 KB SRAM
Wi-Fi	802.11 b/g/n (Broadcom chipset)
GPIO Pins	18
Cloud Integration	Full Particle Cloud support

Operating System on Photon

- Runs on **Particle Device OS**, a lightweight RTOS for IoT applications.
- Provides built-in cloud functionality for remote monitoring and firmware updates.

2.6 Comparison: ESP32 vs Arduino vs Photon

Feature	ESP32	Arduino Uno	Photon
Processor	Dual-core Xtensa LX6, 240 MHz	ATmega328P, 16 MHz	ARM Cortex-M3, 120 M
Voltage	3.3V	5V	3.3V
Flash Memory	4 MB	32 KB	1 MB
RAM	520 KB	2 KB	128 KB
Connectivity	Wi-Fi, Bluetooth	None	Wi-Fi
GPIO Pins	34	14	18
Cloud Support	Limited (via libraries)	None	Full Particle Cloud
Ease of Use	Moderate	Beginner-friendly	Beginner-friendly

2.7 Conclusion

- **ESP32:** Ideal for high-performance IoT and wireless applications.
- **Arduino:** Best for beginners and simple projects.
- **Photon:** Optimized for cloud-connected IoT applications.
- Choose based on application complexity, connectivity, and processing power needs.

2.8 Purpose of SRAM vs Flash Memory on ESP32

1. SRAM (Static Random-Access Memory)

- **Purpose:** Temporary storage for **data** and **variables** during program execution.
- **Key Characteristics:**
  - **Volatile:** Data is lost when the power is turned off.
  - Fast read and write operations, essential for real-time tasks.
  - Used for storing:
    - **Runtime variables** (e.g., counters, flags).
    - **Function call stacks**.
    - **Buffers** for data transmission or sensor input (e.g., Wi-Fi, UART).
- **Example Use Cases:**
  - Storing intermediate data while processing sensor readings.
  - Holding packet data during network communication.

2. Flash Memory

- **Purpose:** Permanent storage for the **program code** and **static data**.
- **Key Characteristics:**
  - **Non-volatile:** Retains data even when power is off.

  - Slower than SRAM but provides much larger storage capacity.
  - Used for storing:
    * **Firmware** (compiled code).
    * **Configuration files** or static assets (e.g., images, HTML for web servers).
    * Non-volatile user data (e.g., Wi-Fi credentials, calibration data).

- **Example Use Cases:**
  - Storing the main firmware and libraries for the ESP32.
  - Saving web server files or sensor calibration data for future use.

Comparison of SRAM and Flash Memory on ESP32

Feature	**SRAM**	**Flash Memory**
Type	Volatile	Non-volatile
Purpose	Temporary data during execution	Permanent storage for firmware and data
Speed	Faster	Slower
Capacity	Limited (e.g., 520 KB on ESP32)	Larger (e.g., 4 MB or more)
Data Retention	Lost when powered off	Retains data after power-off
Examples	Variables, buffers, function stacks	Firmware, configuration, static files

Why Both Are Needed

- **SRAM:** Ensures fast, real-time operations by holding runtime data.
- **Flash:** Provides a large, reliable storage area for program code and persistent data.

Together, SRAM and Flash memory enable the ESP32 to function efficiently in a wide variety of applications, from running real-time tasks to storing IoT configurations and firmware.

Chapter 3

Compilation Flow: From Sketch to Board

The compilation and upload process in the Arduino ecosystem transforms your *sketch* (code) written in the Arduino IDE into machine code that runs on the microcontroller. Below is the step-by-step explanation of the process:

3.1 Writing the Sketch

A **sketch** is the program you write in the Arduino IDE, typically using a simplified version of C/C++. It consists of:

- `setup()` – Initializes settings and runs once.
- `loop()` – Repeats continuously after `setup()`.

3.2 Preprocessing

When you click `Verify` or `Upload`:

1. The IDE adds necessary includes like `#include <Arduino.h>`.
2. Combines multiple files (if any) into one.
3. Processes macros such as `#define`.

3.3 Compilation

The preprocessed sketch is compiled into machine code:

- **C++ Conversion:** The sketch is converted into a `.cpp` file.
- **Compiler Invocation:** The IDE uses the GCC toolchain to compile the code:
  - AVR GCC for Arduino Uno or Mega.
  - Xtensa GCC for ESP32.
  - ARM GCC for ARM-based boards.
- **Linking:** Combines the compiled files into an executable binary.

3.4 Generating the Binary

The compiler produces a binary file (e.g., `.hex` or `.bin`) for the microcontroller.

3.5 Uploading to the Board

The binary is uploaded via:

1. Opening a serial connection.
2. Sending a reset signal to enter bootloader mode.
3. Transferring the binary using a protocol like `avrdude` (AVR) or `esptool.py` (ESP32).

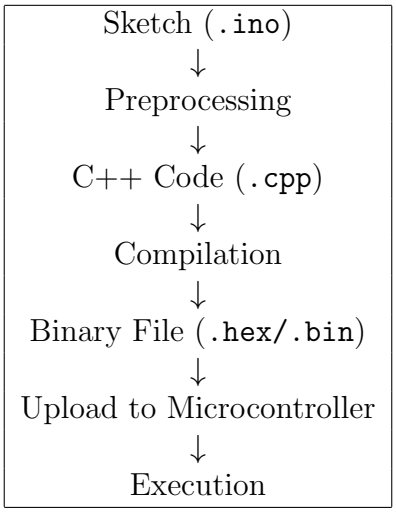
3.6 Program Execution

Once uploaded:

- The microcontroller executes the bootloader.
- Runs `setup()` once.
- Continuously loops through `loop()`.

3.7 Flow Summary

The following steps summarize the compilation process:



This flow ensures the Arduino board executes your intended program efficiently.

3.8 Native vs Cross-Compilation

Compilation can be broadly categorized into two types: **native compilation** and **cross-compilation**. The choice depends on the target platform for the compiled program.

3.8.1 Native Compilation

Native compilation refers to compiling code on the same platform where the program will eventually run.

Features:

1. The compiler and the target system architecture are the same.
2. The program can be directly executed after compilation.
3. No additional tools or configurations are needed for cross-platform compatibility.

Examples:

1. Compiling a program on an Intel processor based machine with x86-64 ISA for execution on an Intel machine with the same ISA.

3.8.2 Cross-Compilation

Cross-compilation involves compiling code on one platform to run on a different platform (the target).

Features:

1. The compiler generates machine code suitable for a target platform different from the host.
2. Requires a toolchain (cross-compiler) specific to the target architecture.
3. Used in embedded systems, IoT devices, and other cases where the target has limited resources.

Examples:

1. Compiling Arduino sketches on a computer and uploading them to an Arduino board (host: PC, target: microcontroller).
2. Cross-compiling for ARM-based systems (e.g., Raspberry Pi) on an x86 desktop.

3.8.3 Comparison

Aspect	**Native Compilation**	**Cross-Compilation**
Definition	Compilation for the same platform	Compilation for a different platform
Compiler	Native compiler	Cross-compiler
Target System	Same as the host	Different from the host
Ease of Setup	Easy, minimal configuration	Requires toolchains and setup
Use Cases	Local application development	Embedded systems, IoT, and cross-platform projects

3.8.4 Relevance to Arduino

The Arduino development process is an example of cross-compilation:

1. Sketches are written and compiled on a PC or laptop.
2. The compiled binary is uploaded to a microcontroller (the target platform).

This allows developers to use a powerful development environment to program devices with limited resources.

Chapter 4

FreeRTOS in Compiled ESP32 Binaries

4.1 Introduction

The ESP32 is a highly versatile microcontroller, widely used in embedded systems and IoT applications. One of its key strengths is its integration with **FreeRTOS**, a real-time operating system (RTOS) that provides multitasking and system management capabilities. This chapter explores the role of FreeRTOS in ESP32 development, its inclusion in compiled binaries, and the benefits it offers for embedded applications.

4.2 Why FreeRTOS Is Included in Compiled Binaries

1. FreeRTOS as Part of the ESP32 SDK

The default development frameworks for the ESP32, such as:

- **ESP-IDF (Espressif IoT Development Framework)**
- **Arduino Core for ESP32**

use FreeRTOS as their underlying operating system. It provides essential services such as task scheduling, timers, and inter-task communication.

2. Automatic Inclusion of FreeRTOS

When you compile a program for the ESP32, the compiled binary automatically includes:

- **Your application code**, such as the `setup()` and `loop()` functions or FreeRTOS tasks.
- **FreeRTOS kernel**, which handles multitasking and resource management.
- **ESP32-specific drivers and libraries**, such as the Wi-Fi and Bluetooth stacks.
- Any **external libraries** used in your program.

This seamless integration simplifies the development of complex applications.

4.3 Components Included in the Binary

The compiled binary typically includes:

Component	Description
Application Code	User-written functions like `setup()` and `loop()`.
FreeRTOS Kernel	Task scheduler, memory management, and synchronization primitives.
Hardware Abstraction Layer	Drivers for GPIO, UART, SPI, I2C, etc.
ESP32-Specific Libraries	Wi-Fi stack, Bluetooth stack, and system services.
External Libraries	Any additional libraries included in the program.

4.4 Bare-Metal Programming vs FreeRTOS

Although it is possible to program the ESP32 without FreeRTOS (bare-metal programming), this approach requires:

- Direct management of hardware resources.
- Handling interrupts and timers manually.
- Custom implementations of multitasking and synchronization.

In contrast, FreeRTOS simplifies these tasks by providing a robust, pre-built framework.

4.5 How to Confirm FreeRTOS in Your Binary

1. Build Logs (ESP-IDF)

If using **ESP-IDF**, the build process will show the inclusion of FreeRTOS-related files, such as:

```
freertos/port.c
freertos/queue.c
```

2. Runtime Task List

FreeRTOS provides a function, `vTaskList()`, which prints all running tasks, including system tasks, allowing you to confirm that FreeRTOS is managing the program.

4.6 Advantages of Including FreeRTOS

- **Multitasking:** Allows multiple tasks to run concurrently, improving efficiency.
- **Real-Time Scheduling:** Ensures tasks are executed within precise time constraints.
- **Simplified Resource Management:** Provides APIs for memory, synchronization, and task communication.
- **System-Level Features:** Manages system tasks like Wi-Fi, Bluetooth, and low-power modes seamlessly.

4.7 Conclusion

The inclusion of FreeRTOS in ESP32 binaries is crucial for leveraging the full potential of the microcontroller. It simplifies the development of real-time, multitasking applications, enabling developers to focus on application logic rather than low-level system management.

Chapter 5

PCBs

5.1 PCB Design with Microcontrollers: Why and When?

Why Move to PCB Design?

- **Prototyping Completes Successfully:** Once a circuit works as expected on a breadboard, moving to a PCB ensures reliability and scalability.
- **Compact and Durable Systems:** PCBs are more compact, durable, and professional compared to fragile breadboard setups.
- **Improved Signal Integrity:** PCBs reduce noise, interference, and loose connections, which are critical in sensitive or high-speed circuits.
- **Mass Production:** PCBs enable uniform and efficient manufacturing for large-scale production.
- **Complex Circuits:** Managing many components on a breadboard becomes impractical; PCBs organize and simplify connections.
- **Power and Heat Management:** PCBs provide efficient power distribution and allow for heat dissipation through thermal vias or heat sinks.
- **Custom Enclosure Fit:** PCBs can be designed to fit specific mechanical enclosures, improving aesthetics and functionality.
- **Long-Term Reliability:** PCBs are less prone to wear and tear, simplifying maintenance and troubleshooting with labeled traces.

ESP32-WROOM: A PCB Module

- The **ESP32-WROOM** is a small, pre-manufactured PCB module containing:
  - ESP32 microcontroller
  - Flash memory, antenna, crystal oscillator, and power regulation circuitry
- It simplifies using the ESP32 by providing solder pads or pin headers for easy integration.

Why Design a Custom PCB for ESP32-WROOM?

Even though ESP32-WROOM is a PCB module, designing a custom PCB may be necessary for:

- **Custom Circuit Integration:** Adding sensors, actuators, or power circuits.
- **Connector and Pin Optimization:** Arranging pins to match specific application needs.
- **Power Management:** Designing custom power supply or battery management circuits.
- **Mechanical Fit:** Ensuring proper fit within an enclosure.
- **Simplified Deployment:** Making mass production easier and more professional.

PCB-on-PCB: ESP32-WROOM on Custom PCB

- Using the ESP32-WROOM on a custom PCB effectively means placing a **PCB on top of another PCB**.
- This modular approach:
  - Simplifies development by abstracting complex microcontroller circuitry.
  - Saves space and reduces design complexity.
  - Enables modular upgrades or replacements.

Conclusion

Transitioning to PCB design from microcontroller modules like ESP32-WROOM ensures reliability, scalability, and a professional finish, especially in complex or mass-production applications.

Chapter 6

SoCs vs MCUs

6.1 Can a Microcontroller Be Considered an SoC?

What Is an SoC?

- A **System on Chip (SoC)** integrates a **CPU, memory, peripherals**, and other components required for a specific application onto a single chip.
- Designed to provide a **complete system** in a compact form factor.

Microcontroller Characteristics

- A microcontroller typically includes:
  - **CPU:** For processing tasks.
  - **RAM:** For temporary data storage.
  - **Flash memory or ROM:** For program storage.
  - **Peripherals:** Timers, I/O ports, ADC, UART, SPI, I2C, etc.
- Designed for **embedded systems** to perform specific tasks efficiently.

Why Microcontrollers Qualify as SoCs

- Like an SoC, a microcontroller integrates most of the components needed for a system to function onto a single chip.
- Example microcontrollers:
  - **ESP32:** Includes CPU, RAM, flash memory, Wi-Fi, and Bluetooth.
  - **STM32:** Combines a Cortex-M CPU, RAM, flash, and various peripherals.

Key Differences Between Microcontrollers and SoCs

- **Purpose and Complexity:**
  - **Microcontrollers:** Optimized for simpler, specific tasks (e.g., sensor control or motor management).

  - **SoCs:** More powerful, capable of running operating systems like Linux, used in complex systems (e.g., smartphones, IoT hubs).

- **Peripheral Integration:**
  - SoCs often include high-performance peripherals (e.g., GPUs, DSPs, wireless modules like 5G).
  - Microcontrollers have fewer peripherals and prioritize power efficiency.

Examples

- **Microcontroller as SoC:**
  - **ESP32:** Dual-core processor, RAM, flash, Wi-Fi, and Bluetooth in one chip.
  - **STM32F4:** Cortex-M CPU, RAM, flash, and peripherals.
- **Traditional SoC:**
  - **Raspberry Pi 4 SoC (Broadcom BCM2711):** Quad-core Cortex-A72, GPU, RAM (external), and high-speed peripherals.

Conclusion

- A microcontroller can be considered a **basic form of SoC** because it integrates most system components onto a single chip.
- However, SoCs generally refer to **more powerful and feature-rich chips** used in advanced systems, while microcontrollers are optimized for **simpler, low-power applications**.

Chapter 7

Connecting ESP32 to ThingSpeak Using HTTP POST

listings xcolor tcolorbox

7.1 Introduction

In IoT applications, microcontrollers like the **ESP32** need to send sensor data to cloud platforms for storage and analysis. **ThingSpeak** is a cloud-based IoT platform that enables:

- Real-time data storage and retrieval.
- Visualization through graphs and dashboards.
- Cloud-based analytics using MATLAB integration.
- Communication using HTTP, MQTT, and WebSockets.

This chapter explains how to connect an ESP32 to ThingSpeak and send sensor data using the **HTTP POST method**.

7.2 Setting Up ThingSpeak

Before programming the ESP32, follow these steps to configure ThingSpeak:

7.2.1 Step 1: Create a ThingSpeak Account

1. Visit `https://thingspeak.com/` and sign up.
2. Log in and navigate to "Channels".
3. Click on **New Channel**.

7.2.2 Step 2: Configure Your Channel

1. Enter a channel name (e.g., *ESP32 Sensor Data*).
2. Enable **Field 1** (Temperature) and **Field 2** (Humidity).
3. Click **Save Channel**.

7.2.3 Step 3: Get the API Key

1. Go to **API Keys** in your ThingSpeak channel.
2. Copy the **Write API Key** (needed for sending data).

7.3 ESP32 Code for Sending Data Using HTTP POST

The following ESP32 program:

- Connects to a Wi-Fi network.
- Reads sensor data (simulated in this example).
- Sends data to ThingSpeak using an **HTTP POST request**.

7.3.1 Code Implementation

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

// Wi-Fi Credentials
const char* ssid = "YourWiFiSSID";
const char* password = "YourWiFiPassword";

// ThingSpeak API
const char* server = "http://api.thingspeak.com/update";
const char* apiKey = "YOUR_THINGSPEAK_API_KEY";

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

    // Connect to Wi-Fi
    WiFi.begin(ssid, password);
    Serial.print("Connecting␣to␣Wi-Fi");
    while (WiFi.status() != WL_CONNECTED) {
        Serial.print(".");
        delay(500);
    }
    Serial.println("\nConnected␣to␣Wi-Fi!");
}

void loop() {
    if (WiFi.status() == WL_CONNECTED) {
        float temperature = readTemperature();
        float humidity = readHumidity();

        HTTPClient http;
        http.begin(server);
        http.addHeader("Content-Type", "application/x-www-form-
           urlencoded");

        // Create POST data
        String postData = "api_key=" + String(apiKey) + "&field1=" +
                          String(temperature) + "&field2=" + String(
                             humidity);

        int httpResponseCode = http.POST(postData);

        if (httpResponseCode > 0) {
            Serial.println("ThingSpeak␣Response:␣" + String(
               httpResponseCode));
        } else {
            Serial.println("Error␣in␣sending␣request:␣" + String(
               httpResponseCode));
        }

        http.end();
    }
    delay(15000); // Send data every 15 seconds
}

float readTemperature() { return random(25, 35); }
float readHumidity() { return random(40, 60); }
```

7.4 Understanding the Code

7.4.1 Connecting to Wi-Fi

The ESP32 connects to Wi-Fi using:

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

7.4.2 Creating an HTTP POST Request

The ESP32 sends sensor data using:

```
HTTPClient http;
http.begin(server);
http.addHeader("Content-Type", "application/x-www-form-urlencoded");
String postData = "api_key=" + String(apiKey) + "&field1=" +
                  String(temperature) + "&field2=" + String(humidity);
int httpResponseCode = http.POST(postData);
```

7.4.3 Handling the Response

If the request is successful, ThingSpeak responds with an HTTP status code:

```
if (httpResponseCode > 0) {
    Serial.println("ThingSpeak Response: " + String(httpResponseCode));
} else {
    Serial.println("Error in sending request");
}
```

7.5 Troubleshooting

Issue	Solution
Wi-Fi not connecting password, reboot ESP32.	Check SSID
HTTP Response -1	No internet access, check Wi-Fi.
No data on ThingSpeak	Verify API key and POST request format.
Data update delay	ThingSpeak limits free users to 15s intervals.

Table 7.1: Common Issues and Solutions

7.6 Conclusion

This chapter explained:

- How ESP32 sends data to ThingSpeak using **HTTP POST**.
- How to set up a **ThingSpeak channel**.
- How to troubleshoot common issues.

In the next chapter, we will explore **REST API** in detail, which will help us understand how ThingSpeak and other cloud services interact with IoT devices.

Chapter 8

REST API

8.1 Introduction to Internet Addresses and Ports

Every device on the internet is identified by an IP address, a numerical label assigned to each device. Additionally, services on a device use ports to distinguish different applications. For example:

- A web server typically listens on port 80 (HTTP) or 443 (HTTPS).
- A database service might be on port 3306 (MySQL) or 5432 (PostgreSQL).
- IP addresses are either IPv4 (e.g., `192.168.1.1`) or IPv6 (e.g., `2001:db8::ff00:42:8329`).

Communication over the internet happens via protocols like HTTP, which forms the basis of REST APIs.

8.2 JSON Files

JSON (JavaScript Object Notation) is a lightweight data-interchange format. It is easy for humans to read and write and easy for machines to parse and generate. Example:

Listing 8.1: Example JSON Object

```
{
    "name": "John Doe",
    "age": 30,
    "email": "johndoe@example.com",
    "skills": ["Python", "C++", "REST APIs"]
}
```

JSON is widely used in REST APIs to exchange data between client and server.

8.3 What is REST API?

REST (Representational State Transfer) API allows communication between systems over HTTP. The main HTTP methods used are:

- **GET** – Retrieve data

- **POST** – Send data
- **PUT** – Update data
- **DELETE** – Remove data

Example of using Python to make GET and POST requests:

Listing 8.2: Using requests module

```
import requests

# GET request
response = requests.get("https://jsonplaceholder.typicode.com/posts/1")
print(response.json())

# POST request
payload = {"title": "foo", "body": "bar", "userId": 1}
response = requests.post("https://jsonplaceholder.typicode.com/posts", json
print(response.json())
```

8.4 Watching Python HTTP Server via Wireshark

1. Start a simple Python HTTP server:

```
python3 -m http.server 8080
```

2. Open Wireshark and start capturing packets on the network interface. 3. Filter packets using HTTP:

```
http
```

4. Send a GET request using a browser or curl:

```
curl http://localhost:8080
```

5. Observe the request and response in Wireshark.

8.5 Using Postman to Talk to Python HTTP Server

1. Start the Python server. 2. Open Postman, create a GET request to `http://localhost:8080`.
3. Observe the response in Postman.

8.6 Using Postman with OpenLibrary API

To search for a book:

```
GET https://openlibrary.org/search.json?q=harry+potter
```

8.7 CRUD Operations

CRUD refers to:

- Create (POST)
- Read (GET)
- Update (PUT)
- Delete (DELETE)

Example of updating a post:

```
requests.put("https://jsonplaceholder.typicode.com/posts/1", json={"title":
```

8.8 Interfacing with an IoT Device via REST API

Assume an ESP32 sends sensor data:

```
import requests
response = requests.get("http://iot-device.local/sensor")
print(response.json())
```

8.9 Hosting a Python Web Server for IoT Data

The following is a simple Python Flask server to which IoT devices can POST their data.

8.10 Hosting a Python Web Server for IoT Data

Create a Python server:

```
from flask import Flask, request, jsonify

app = Flask(__name__)

data = []

@app.route("/sensor", methods=["POST"])
def receive_data():
    content = request.json
    data.append(content)
    return jsonify({"message": "Data received"})

@app.route("/dashboard", methods=["GET"])
def dashboard():
    return jsonify(data)

if __name__ == "__main__":
    app.run(port=5000)
```

Plot data in Jupyter Notebook:

```
import matplotlib.pyplot as plt
import requests

response = requests.get("http://localhost:5000/dashboard")
data = response.json()

timestamps = [entry["timestamp"] for entry in data]
temperatures = [entry["temperature"] for entry in data]

plt.plot(timestamps, temperatures)
plt.xlabel("Time")
plt.ylabel("Temperature")
plt.show()
```

8.11 ESP32 WebServer for REST API

The following code will run a http server on the IoT device supporting the API endpoint http://iot-ipaddress/sensor.

Listing 8.3: ESP32 Web Server

```
#include <WiFi.h>
#include <WebServer.h>
#include <ArduinoJson.h>
#define PASSWORD "xxxxxxx"

const char* ssid = "Your_SSID";
const char* password = PASSWORD;

WebServer server(80);

float getRandomTemperature() {
    return random(200, 350) / 10.0;
}

float getRandomHumidity() {
    return random(400, 700) / 10.0;
}

void handleSensorData() {
    DynamicJsonDocument doc(200);
    doc["temperature"] = getRandomTemperature();
    doc["humidity"] = getRandomHumidity();

    String jsonResponse;
    serializeJson(doc, jsonResponse);
```

```
    server.send(200, "application/json", jsonResponse);
}

void setup() {
    Serial.begin(115200);
    WiFi.begin(ssid, password);
    Serial.print("Connecting to WiFi...");
    while (WiFi.status() != WL_CONNECTED) {
        delay(1000);
        Serial.print(".");
    }
    Serial.println("\nConnected to WiFi!");
    Serial.print("ESP32 IP Address: ");
    Serial.println(WiFi.localIP());
    server.on("/sensor", HTTP_GET, handleSensorData);
    server.begin();
    Serial.println("Server started.");
}

void loop() {
    server.handleClient();
}
```

Chapter 9

Understanding URIs in REST APIs

9.1 Introduction to URIs

A **Uniform Resource Identifier (URI)** uniquely identifies a resource in a RESTful API. It generally follows the structure:

```
scheme://host:port/path?query_parameters
```

where:

- **Scheme**: Protocol used (e.g., `http`, `https`).
- **Host**: The domain or IP address of the server (e.g., `www.example.com`).
- **Port**: The network port (optional, default is `80` for HTTP and `443` for HTTPS).
- **Path**: The resource being accessed (e.g., `/users/123`).
- **Query Parameters**: Additional key-value pairs (e.g., `?sort=desc&limit=10`).

9.2 Route Parameters vs. Query Parameters

REST APIs use both **route parameters** and **query parameters** to pass information. Each serves a distinct purpose.

9.2.1 Route Parameters (Path Parameters)

- Used to uniquely identify a resource.
- Placed within the **URI path**.
- Typically required for RESTful API endpoints.

Example:

```
GET /users/{id}
```

For retrieving user `123`, the actual request would be:

`GET /users/123`

Another example with nested resources:

`GET /orders/{orderId}/items/{itemId}`

If `orderId = 45` and `itemId = 678`, the request would be:

`GET /orders/45/items/678`

9.2.2 Query Parameters

- Used for filtering, sorting, or paginating data.
- Passed after a `?` in key-value pairs.
- Multiple parameters are separated by `&`.
- Optional, providing additional functionality.

Example: Filtering and Sorting

`GET /users?age=25&sort=desc&limit=10`

This request:

- Filters users with `age = 25`.
- Sorts in **descending** order.
- Limits results to `10` users.

Example: Pagination

`GET /products?page=2&pageSize=20`

Retrieves the **second page** with **20 items per page**.

9.3 Comparison of Route and Query Parameters

Feature	Route Parameters	Query Parameters
Placement	In the URI path	After `?` in the URL
Purpose	Identifies a specific resource	Filters or modifies a resource request
Required?	Yes (usually)	No (optional)
Data Type	Typically IDs or identifiers	Additional filters like sort, pagination
Example	`/users/123`	`/users?age=25&sort=desc`

Table 9.1: Comparison of Route Parameters and Query Parameters

9.4 Best Practices for API Design

- Use **route parameters** for uniquely identifying resources.
- Use **query parameters** for filtering, sorting, and pagination.
- Keep URIs clean and readable, avoiding deeply nested structures.
- Design APIs to be intuitive and RESTful by following standard conventions.

Chapter 10

PetStore Application Backend - CRUD API

10.1 Introduction

This document provides a complete implementation of the PetStore backend API using Flask. Additionally, it demonstrates CRUD operations using cURL and Postman.

10.2 Backend Implementation

10.2.1 Flask API Code

Listing 10.1: PetStore Backend Implementation

```
from flask import Flask, jsonify, request
from flask_cors import CORS  # Allow cross-origin requests

# Initialize Flask app
app = Flask(__name__)
CORS(app)  # Enable CORS

# In-memory database
pets = [
    {"id": 1, "name": "Buddy", "type": "Dog"},
    {"id": 2, "name": "Mittens", "type": "Cat"},
    {"id": 3, "name": "Goldie", "type": "Fish"}
]

# Fetch all pets
@app.route('/pets', methods=['GET'])
def get_all_pets():
    return jsonify(pets)

# Fetch pet by ID
@app.route('/pets/<int:id>', methods=['GET'])
def get_pet_by_id(id):
```

```
    pet = next((p for p in pets if p["id"] == id), None)
    return jsonify(pet) if pet else ("Not Found", 404)

# Add a new pet
@app.route('/pets', methods=['POST'])
def add_pet():
    new_pet = request.json
    new_pet["id"] = len(pets) + 1
    pets.append(new_pet)
    return jsonify(new_pet), 201

# Update an existing pet
@app.route('/pets/<int:id>', methods=['PUT'])
def update_pet(id):
    pet = next((p for p in pets if p["id"] == id), None)
    if not pet:
        return "Not Found", 404
    pet.update(request.json)
    return jsonify(pet)

# Delete a pet
@app.route('/pets/<int:id>', methods=['DELETE'])
def delete_pet(id):
    global pets
    pets = [p for p in pets if p["id"] != id]
    return "", 204

if __name__ == '__main__':
    app.run(debug=True)
```

10.3 Using cURL to Test the API

10.3.1 Fetching All Pets

```
curl -X GET http://localhost:5000/pets
```

10.3.2 Fetching a Pet by ID

```
curl -X GET http://localhost:5000/pets/1
```

10.3.3 Adding a New Pet

```
curl -X POST http://localhost:5000/pets \
     -H "Content-Type: application/json" \
     -d '{"name": "Charlie", "type": "Dog"}'
```

10.3.4 Updating a Pet

```
curl -X PUT http://localhost:5000/pets/1 \
     -H "Content-Type: application/json" \
     -d '{"name": "Charlie Updated", "type": "Dog"}'
```

10.3.5 Deleting a Pet

```
curl -X DELETE http://localhost:5000/pets/1
```

10.4 Using Postman to Test the API

10.4.1 Fetching All Pets

1. Open Postman. 2. Select **GET** method. 3. Enter URL: `http://localhost:5000/pets`. 4. Click **Send**.

10.4.2 Fetching a Pet by ID

1. Select **GET** method. 2. Enter URL: `http://localhost:5000/pets/1`. 3. Click **Send**.

10.4.3 Adding a New Pet

1. Select **POST** method. 2. Enter URL: `http://localhost:5000/pets`. 3. Go to **Body** →Select **raw** →Choose **JSON**. 4. Enter the following JSON:

```
{
    "name": "Charlie",
    "type": "Dog"
}
```

5. Click **Send**.

10.4.4 Updating a Pet

1. Select **PUT** method. 2. Enter URL: `http://localhost:5000/pets/1`. 3. Go to **Body** →Select **raw** →Choose **JSON**. 4. Enter the following JSON:

```
{
    "name": "Charlie Updated",
    "type": "Dog"
}
```

5. Click **Send**.

10.4.5 Deleting a Pet

1. Select **DELETE** method. 2. Enter URL: `http://localhost:5000/pets/1`. 3. Click **Send**.

10.5 Conclusion

This document provides a full implementation of a PetStore backend API along with detailed instructions for testing CRUD operations using both cURL and Postman.

Chapter 11

Load Balancing a REST API with NGINX

11.1 Introduction to Load Balancing and Reverse Proxy

Load balancing is a technique used to distribute incoming network traffic across multiple servers, ensuring no single server bears too much load. This enhances the availability, fault tolerance, and scalability of web applications.

A **reverse proxy** is a server that sits between client requests and backend servers. It forwards client requests to multiple backend servers and returns the response to the client. A common use case of a reverse proxy is load balancing, where it distributes incoming requests among multiple API servers.

NGINX is a widely used web server that also functions as a powerful reverse proxy and load balancer. It efficiently routes requests to backend servers and supports multiple load-balancing strategies such as round-robin, least connections, and IP-hash.

11.2 Setting Up Load Balancing for a REST API

To demonstrate load balancing, we create two identical REST API servers using Flask and configure NGINX to distribute requests between them.

11.2.1 Creating Multiple REST API Servers (Flask)

Each REST API server runs on a different port and returns a JSON response indicating which server handled the request.

```
# File: server1.py
from flask import Flask, jsonify

app = Flask(__name__)

@app.route('/api/data', methods=['GET'])
def get_data():
    return jsonify({"server": "Server 1", "message": "Hello from Server 1!"})
```

```
if __name__ == '__main__':
    app.run(host='0.0.0.0', port=5001)  # Run on port 5001

# File: server2.py
from flask import Flask, jsonify

app = Flask(__name__)

@app.route('/api/data', methods=['GET'])
def get_data():
    return jsonify({"server": "Server 2", "message": "Hello from Server 2!"})

if __name__ == '__main__':
    app.run(host='0.0.0.0', port=5002)  # Run on port 5002
```

Configuring NGINX as a Load Balancer

The following NGINX configuration forwards incoming requests to both Flask servers using round-robin load balancing.

```
http {
    upstream backend_servers {
        server 127.0.0.1:5001;
        server 127.0.0.1:5002;
    }

    server {
        listen 8080;

        location /api/ {
            proxy_pass http://backend_servers;
        }
    }
}
```

11.2.2 Starting the Servers and Load Balancer

To start the Flask API servers, run the following commands in separate terminal windows:

```
python server1.py
python server2.py
```

To start NGINX, use:

```
nginx -c /path/to/nginx.conf
```

If using Docker, run:

```
docker run --name nginx-lb -v /path/to/nginx.conf:/etc/nginx/nginx.conf:ro -p 8080:80
```

11.2.3 Testing the Load Balancer

To verify the load balancer is working, make repeated requests using a browser or cURL:

```
curl http://localhost:8080/api/data
```

Each request alternates between the two servers, returning:

```
{"server": "Server 1", "message": "Hello from Server 1!"}
{"server": "Server 2", "message": "Hello from Server 2!"}
```

11.3 Conclusion

By using NGINX as a reverse proxy, we efficiently distribute REST API requests among multiple backend servers, ensuring high availability, scalability, and fault tolerance. This setup helps applications handle increased traffic loads while maintaining responsiveness and reliability.

Chapter 12

Why REST API is Better than RPC

12.1 What are RPCs?

Remote Procedure Call (RPC) is a protocol that allows a program to execute a function on a remote server as if it were a local function. Instead of making an HTTP request to a RESTful endpoint, the client calls a function directly, passing parameters, and receiving a return value.

RPCs can use various protocols such as:

- **gRPC** (Google Remote Procedure Call) – Uses Protocol Buffers for efficient communication.
- **JSON-RPC** – Uses JSON format over HTTP or WebSockets.
- **XML-RPC** – Uses XML for communication.

12.2 Why REST API is Better than RPC

While RPC can be efficient in some cases, REST APIs have several advantages that make them a better choice for many applications.

12.2.1 Simplicity and Readability

- **REST**: Uses standard HTTP methods (`GET`, `POST`, `PUT`, `DELETE`), making it easier to understand and use.
- **RPC**: Uses custom function names (e.g., `getUserData`, `updateUserInfo`), which can be less intuitive and require additional documentation.

12.2.2 Scalability and Statelessness

- **REST**: REST APIs are typically **stateless**, meaning each request is independent. This makes scaling easier as load balancers can distribute requests efficiently.
- **RPC**: Some RPC implementations require maintaining session state, which can lead to scaling issues.

12.2.3 Flexibility and Loose Coupling

- **REST**: Works with **resources** (e.g., `/users/123`), making it loosely coupled and easy to modify without breaking clients.

- **RPC**: Tightly coupled; changes to function signatures (e.g., `getUser(123)`) can break clients.

12.2.4 Standardization

- **REST**: Uses standard **HTTP status codes, headers, and methods**, making it easier to debug and integrate with other services.

- **RPC**: Lacks such standardization, often requiring custom error handling.

12.2.5 Better Caching Support

- **REST**: Supports **HTTP caching** with headers like `ETag`, `Last-Modified`, and `Cache-Control`, improving performance.

- **RPC**: Generally does not have built-in caching mechanisms.

12.2.6 Language Agnosticism

- **REST**: Works with any programming language since it uses **JSON, XML, or other text formats** over HTTP.

- **RPC**: Often requires **platform-specific formats** (e.g., gRPC with Protocol Buffers), making it harder for non-native clients to consume.

12.2.7 Security

- **REST**: Can use **standard security mechanisms** like OAuth, JWT, and API keys over HTTPS.

- **RPC**: Security depends on implementation; some RPC frameworks have authentication built-in, but others require custom solutions.

12.2.8 Human-Friendly

- **REST**: Uses **URL-based resource access** (e.g., `/users/123`), making it more readable and self-documenting.

- **RPC**: Function call-based (e.g., `getUser(123)`) can be harder to understand without extensive documentation.

12.3 When to Use RPC Instead?

While REST is generally preferred for **web-based, scalable, loosely coupled services**, RPC has some use cases where it excels:

- **High-performance internal APIs** – gRPC is optimized for low-latency, high-throughput communication.
- **Real-time communication** – gRPC supports bidirectional streaming, making it ideal for chat applications and IoT.
- **Strict contract enforcement** – gRPC with Protocol Buffers provides **strongly-typed contracts** between client and server.

12.4 Conclusion

REST APIs are better suited for most **web applications** because they are simple, scalable, standardized, and language-agnostic. However, RPC (especially **gRPC**) is a better choice for **high-performance microservices** and applications needing **low latency or real-time communication**.

Chapter 13

Introduction to MQTT

13.1 Introduction to MQTT

13.1.1 What is MQTT?

MQTT (Message Queuing Telemetry Transport) is a lightweight messaging protocol optimized for devices with limited resources and networks with low bandwidth or high latency. This makes it highly suitable for Internet of Things (IoT) applications, where devices often operate on constrained networks.

13.1.2 Why MQTT for IoT?

- **Low Bandwidth Usage:** Ideal for sensor data transmission.
- **Power Efficiency:** Suitable for battery-powered devices.
- **Easy Implementation:** Simple and effective protocol structure.
- **Reliable Communication:** Offers different Quality of Service (QoS) levels.

13.1.3 MQTT in Smart Homes

MQTT allows seamless communication between smart home devices like sensors, actuators, and control systems. For example:

- A temperature sensor **publishes** temperature data.
- A smart thermostat **subscribes** to the temperature data and adjusts HVAC settings.
- A smartphone app **subscribes** to topics to monitor devices remotely.

13.2 MQTT Architecture and Components

13.2.1 Key Components

- **Broker:** The central server that routes messages between publishers and subscribers. (e.g., Mosquitto)

- **Publisher:** Sends data to a topic (e.g., a temperature sensor publishing to `home/livingroom/temp`
- **Subscriber:** Receives messages from subscribed topics (e.g., a thermostat listening to temperature data).
- **Topics:** Hierarchical strings used to categorize messages (e.g., `home/kitchen/light`).

13.2.2 Topic Structure in a Smart Home

Example topic hierarchy:

- `home/livingroom/temperature`
- `home/kitchen/light`
- `home/bedroom/humidity`
- `home/garage/doorstatus`

Wildcards in Topics

- **+ (Single-level):** Subscribes to one level (e.g., `home/+/temperature`).
- **(Multi-level):** Subscribes to all levels (e.g., `home/`).

13.2.3 Quality of Service (QoS) Levels

- **QoS 0:** At most once (fire and forget).
- **QoS 1:** At least once (guaranteed delivery but possible duplicates).
- **QoS 2:** Exactly once (guarantees no duplicates).

13.3 Smart Home Case Study

13.3.1 Scenario Description

Consider a smart home setup with the following devices:

- **Temperature Sensors**: Publish data to topics like `home/livingroom/temperature`.
- **Smart Thermostat**: Subscribes to temperature topics and adjusts the HVAC system.
- **Motion Sensors**: Publish occupancy data.
- **Light Controllers**: Subscribe to occupancy data to manage lighting.
- **Door Sensors**: Publish door status for security notifications.

13.3.2 Data Flow Example

1. The temperature sensor detects 26°C and publishes to `home/livingroom/temperature`.
2. The smart thermostat receives the data and activates the air conditioning.
3. The mobile app displays real-time temperature data to the user.
4. Occupancy sensors publish data, prompting lights to turn on when someone enters a room.

13.3.3 Advantages of Using MQTT in Smart Homes

- **Scalability**: Easily add new devices without reconfiguring existing ones.
- **Real-time Communication**: Quick response to environmental changes.
- **Efficient Bandwidth Usage**: Devices receive only relevant messages.

13.4 Example Code

13.4.1 Publisher: Simulated Temperature Sensor

Listing 13.1: Temperature Sensor Publisher

```
import paho.mqtt.client as mqtt
import time
import random

broker = "test.mosquitto.org"
port = 1883
topic = "home/livingroom/temperature"

client = mqtt.Client("TemperatureSensor")
client.connect(broker, port, 60)

while True:
    temperature = round(random.uniform(18.0, 30.0), 2)
    result = client.publish(topic, temperature)
    if result[0] == 0:
        print(f"Sent {temperature} C  to topic {topic}")
    time.sleep(5)
```

13.4.2 Subscriber: Smart Thermostat

Listing 13.2: Smart Thermostat Subscriber

```
import paho.mqtt.client as mqtt

broker = "test.mosquitto.org"
port = 1883
topic = "home/livingroom/temperature"

def on_connect(client, userdata, flags, rc):
```

```
    print("Connected with result code", rc)
    client.subscribe(topic)

def on_message(client, userdata, msg):
    temperature = float(msg.payload.decode())
    print(f"Received {temperature} C  from {msg.topic}")
    if temperature > 25:
        print("Turning on AC.")
    elif temperature < 20:
        print("Turning on heater.")

client = mqtt.Client("SmartThermostat")
client.on_connect = on_connect
client.on_message = on_message

client.connect(broker, port, 60)
client.loop_forever()
```

13.5 Hands-On Activities

1. Modify the publisher to simulate humidity data and publish to `home/livingroom/humidity`.
2. Create a light controller that subscribes to occupancy data and automates lighting.
3. Use wildcard subscriptions like `home/+/temperature` to subscribe to multiple rooms.
4. Implement retained messages so new subscribers receive the last published value.
5. Experiment with different QoS levels and observe message delivery behavior.

13.6 Summary

- MQTT provides an efficient communication method for IoT devices.
- The publish/subscribe model facilitates scalable smart home applications.
- Real-world use cases include environmental monitoring, automated lighting, and security systems.

13.7 Resources

- MQTT Official Site
- Mosquitto Broker Documentation
- Paho MQTT Library
- MQTT Explorer Tool

Chapter 14

oneM2M and Resource Oriented Architecture (ROA)

14.1 Introduction to oneM2M

oneM2M is a global standards initiative that aims to develop technical specifications to ensure interoperability of Machine-to-Machine (M2M) and Internet of Things (IoT) systems. It provides a common M2M Service Layer that can be embedded within hardware and software to connect devices and applications across various sectors.

Key Objectives

- Provide a common service layer.
- Enable interoperability across different verticals and technologies.
- Support various IoT use cases (smart homes, health, transportation, etc.).

14.2 Resource Oriented Architecture (ROA)

ROA is the architectural model that oneM2M uses to manage and represent resources. It organizes system entities as resources that can be manipulated through standard CRUD (Create, Retrieve, Update, Delete) operations.

Core Concepts

- **Resource:** Any manageable entity (e.g., device, application entity, container).
- **URI (Uniform Resource Identifier):** Each resource is uniquely identifiable.
- **CRUDN Operations:** Extend standard CRUD with Notification (N).
- **Hierarchical Structure:** Resources are structured in a tree-like hierarchy.

14.3 oneM2M Resource Tree Structure

The resource tree begins with the Common Service Entity (CSE), branching into various child resources.

Resource Types

- **CSEBase:** Root resource for a CSE.
- **AE (Application Entity):** Represents an application.
- **Container:** Holds data instances.
- **ContentInstance:** Actual data payload.
- **Subscription:** Manages notifications.

Example Hierarchy

```
/CSEBase
 AE (TemperatureApp)
    Container (TemperatureData)
       ContentInstance (Data1)
       ContentInstance (Data2)
    Subscription (TempAlert)
```

14.4 Code Examples Using HTTP REST API

14.4.1 Create an Application Entity (AE)

```python
import requests

cse_url = "http://localhost:8080/~/in-cse/in-name"
headers = {
    'X-M2M-Origin': 'admin:admin',
    'Content-Type': 'application/json;ty=2'
}
data = {
    "m2m:ae": {
        "rn": "TemperatureApp",
        "api": "app-sensor",
        "rr": True
    }
}
response = requests.post(cse_url, headers=headers, json=data)
print(response.status_code, response.json())
```

Listing 14.1: Creating an AE

14.4.2 Create a Container

```
container_url = cse_url + "/TemperatureApp"
headers['Content-Type'] = 'application/json;ty=3'
data = {
    "m2m:cnt": {
        "rn": "TemperatureData"
    }
}
response = requests.post(container_url, headers=headers, json=data)
print(response.status_code, response.json())
```

Listing 14.2: Creating a Container

14.4.3 Add ContentInstance (Data)

```
content_url = container_url + "/TemperatureData"
headers['Content-Type'] = 'application/json;ty=4'
data = {
    "m2m:cin": {
        "con": "23.5"
    }
}
response = requests.post(content_url, headers=headers, json=data)
print(response.status_code, response.json())
```

Listing 14.3: Adding Data

14.4.4 Retrieve Latest Data

```
retrieve_url = content_url + "/la"  # 'la' refers to latest content
    instance
response = requests.get(retrieve_url, headers={'X-M2M-Origin': 'admin:
    admin'})
print(response.status_code, response.json())
```

Listing 14.4: Retrieving Latest Data

14.5 Notifications and Subscriptions

Applications can subscribe to resources to receive notifications when data changes.

14.5.1 Create a Subscription

```
subscription_url = container_url + "/TemperatureData"
headers['Content-Type'] = 'application/json;ty=23'
data = {
    "m2m:sub": {
        "rn": "TempAlert",
        "nu": ["http://my-server.com/notify"],
        "nct": 2
    }
```

```
}
response = requests.post(subscription_url, headers=headers, json=data)
print(response.status_code, response.json())
```

Listing 14.5: Creating a Subscription

14.6 Key Takeaways

- oneM2M leverages a Resource Oriented Architecture.
- CRUDN operations manage the lifecycle of resources.
- Hierarchical resource trees ensure structured data management.
- Subscriptions facilitate real-time notifications.

14.7 Further Reading and Resources

- oneM2M Official Website
- TS-0001 Functional Architecture
- GitHub Examples for oneM2M

Chapter 15

Polling vs Interrupt-Driven Approach in Embedded Systems

15.1 Introduction

In embedded systems, handling inputs such as push buttons or capacitive touch sensors can be achieved using either **polling** or **interrupt-driven** approaches. Understanding the difference between these two methods is crucial for optimizing system performance, responsiveness, and power consumption.

This lecture note covers the differences between **polling** and **interrupts**, their advantages and disadvantages, and how **light sleep mode** further enhances efficiency. Example codes are provided to demonstrate these concepts using an ESP32 and its built-in **capacitive touch sensor**.

—

15.2 Polling Approach

15.2.1 What is Polling?

Polling is a method where the microcontroller continuously checks the status of an input device at regular intervals. If a condition is met (e.g., a button is pressed), an action is performed.

15.2.2 Advantages of Polling

- Simple to implement.
- No need for additional hardware configurations.

15.2.3 Disadvantages of Polling

- Inefficient: Wastes CPU cycles checking the input constantly.
- Less responsive: Delays in response due to periodic checks.
- High power consumption: CPU is always active.

15.2.4 Example Code: Polling a Capacitive Touch Sensor

This example continuously checks the **capacitive touch sensor** on GPIO 4 and turns on the LED if a touch is detected.

```
#define TOUCH_PIN T0  // GPIO 4 (Capacitive touch sensor)
#define LED_PIN 2     // Built-in LED

void setup() {
  Serial.begin(115200);
  pinMode(LED_PIN, OUTPUT);
}

void loop() {
  int touchValue = touchRead(TOUCH_PIN);  // Read touch sensor
  Serial.println(touchValue);

  if (touchValue < 40) {  // Threshold for detecting touch
    digitalWrite(LED_PIN, HIGH);  // Turn LED ON
    Serial.println("Touched! LED ON");
  } else {
    digitalWrite(LED_PIN, LOW);   // Turn LED OFF
  }

  delay(100);  // Small delay for stability
}
```

15.2.5 How It Works:

- The `loop()` function continuously checks the touch sensor value.
- If the value is below a threshold, it turns on the LED and prints "Touched! LED ON".
- Otherwise, the LED remains off.
- **Downside:** The ESP32 wastes CPU cycles checking the sensor **all the time**.

15.3 Interrupt-Driven Approach

15.3.1 What is an Interrupt?

An interrupt is a signal that **immediately stops** the CPU's current task and forces it to execute a specific function (Interrupt Service Routine - ISR). Once the ISR completes, the CPU resumes its previous task.

15.3.2 Advantages of Interrupts

- **Efficient:** CPU is not constantly checking input; it runs other tasks and only reacts when needed.
- **Responsive:** Immediate response to events.

- **Lower power consumption** compared to polling.

15.3.3 Disadvantages of Interrupts

- Slightly more complex to implement.
- Improper handling of interrupts may cause unexpected behavior (e.g., multiple triggers due to switch bouncing).

15.3.4 Example Code: Using an Interrupt for the Touch Sensor

This code triggers an interrupt when the **capacitive touch sensor** is activated.

```
#define TOUCH_PIN T0  // GPIO 4 (Capacitive touch sensor)
#define LED_PIN 2     // Built-in LED

volatile bool touched = false;  // Flag to track touch state

void IRAM_ATTR handleTouch() {
  touched = true;  // Set flag when touch is detected
}

void setup() {
  Serial.begin(115200);
  pinMode(LED_PIN, OUTPUT);

  // Attach interrupt to the touch sensor
  touchAttachInterrupt(TOUCH_PIN, handleTouch, 40);  // Threshold = 40
}

void loop() {
  if (touched) {
    touched = false;  // Reset flag
    digitalWrite(LED_PIN, HIGH);  // Turn LED ON
    Serial.println("Interrupt: Touched! LED ON");

    delay(500);  // Small delay to avoid flickering
    digitalWrite(LED_PIN, LOW);   // Turn LED OFF
  }
}
```

15.3.5 How It Works:

- When the touch sensor detects a touch, the **interrupt** function (`handleTouch()`) is called.
- The flag `touched` is set to `true`.
- In the `loop()`, if `touched == true`, the LED is turned on, and a message is printed.
- The **CPU is free to do other tasks** while waiting for an interrupt.

—

15.4 Interrupt with Light Sleep Mode (Most Efficient)

15.4.1 What is Light Sleep Mode?

Light sleep mode reduces power consumption by **disabling the CPU while keeping peripherals active**. The CPU wakes up **only when an interrupt occurs** (e.g., a touch event).

15.4.2 Benefits of Light Sleep Mode

- **Saves power** while waiting for user input.
- **Wakes up instantly** when a touch is detected.
- Ideal for battery-powered devices.

15.4.3 Example Code: Interrupt with Light Sleep Mode

```
#include "esp_sleep.h"

#define TOUCH_PIN T0  // GPIO 4 (Capacitive touch sensor)
#define LED_PIN 2     // Built-in LED

volatile bool touched = false;  // Flag to track touch state

void IRAM_ATTR handleTouch() {
  touched = true;  // Set flag when touch is detected
}

void setup() {
  Serial.begin(115200);
  pinMode(LED_PIN, OUTPUT);

  // Attach interrupt to the touch sensor
  touchAttachInterrupt(TOUCH_PIN, handleTouch, 40);  // Threshold = 40

  // Enable touchpad wake-up from sleep
  esp_sleep_enable_touchpad_wakeup();
}

void loop() {
  if (touched) {
    touched = false;  // Reset flag
    digitalWrite(LED_PIN, HIGH);  // Turn LED ON
    Serial.println("Interrupt: Touched! LED ON");

    delay(500);  // Small delay for visibility
    digitalWrite(LED_PIN, LOW);   // Turn LED OFF
  }

  Serial.println("Entering Light Sleep Mode...");
  delay(100);  // Short delay before sleep

```

```
36   // Put ESP32 into light sleep mode
37   esp_light_sleep_start();
38
39   // Execution resumes here after wake-up
40   Serial.println("Woke up from sleep!");
41 }
```

15.4.4 How It Works:

- ESP32 enters **light sleep mode**.
- When a **touch is detected**, the CPU **wakes up** and turns on the LED.
- The LED turns off after a short delay, and the ESP32 **goes back to sleep**.

—

15.5 Comparison Table: Polling vs Interrupt vs Sleep Mode

Category	Polling	Interrupt	Interrupt + Sleep
CPU Usage	High	Low	Very Low
Power Efficiency	Inefficient	More efficient	Most efficient
Responsiveness	Instant but wasteful	Fast response	Fast, wakes only when needed
Best For	Simple applications	Real-time systems	Low-power applications

Table 15.1: Comparison of Polling, Interrupt, and Interrupt with Sleep Mode

15.6 Conclusion

- **Polling** is simple but inefficient.
- **Interrupts** improve efficiency and responsiveness.
- **Light Sleep Mode** maximizes power savings.

Chapter 16

OTA

16.1 Introduction to OTA

16.1.1 What is OTA?

Over-The-Air (OTA) updates allow you to update the firmware of an ESP32 device wirelessly, without the need for physical access. This is particularly useful for devices deployed in remote or hard-to-reach locations.

16.1.2 Why use OTA?

- **Convenience**: No need to physically connect the device to a computer.
- **Scalability**: Update multiple devices simultaneously.
- **Maintenance**: Fix bugs, add features, or patch security vulnerabilities remotely.

16.1.3 Types of OTA Updates

- **Basic OTA**: Updates via Arduino IDE or HTTP server.
- **Advanced OTA**: Updates via cloud services (e.g., AWS IoT, Firebase, or custom backend).

16.2 Underlying Mechanics of OTA

16.2.1 How OTA Works on ESP32

The ESP32 has two app partitions: `ota_0` and `ota_1`. During an OTA update, the new firmware is written to the inactive partition. After the update, the bootloader switches to the updated partition.

16.2.2 Key Components

- **Bootloader**: Manages which partition to boot from.
- **Partition Table**: Defines the layout of flash memory (e.g., app partitions, OTA data, SPIFFS).

- **OTA Data**: Stores metadata about the active partition.

16.2.3 OTA Process

1. The device downloads the new firmware (e.g., from an HTTP server or cloud).
2. The firmware is written to the inactive partition.
3. The bootloader updates the OTA data to point to the new partition.
4. The device reboots and runs the updated firmware.

16.3 Implementing OTA on ESP32

16.3.1 Prerequisites

- Arduino IDE or PlatformIO.
- ESP32 board support installed.
- Wi-Fi credentials for the ESP32 to connect to the internet.

16.3.2 Basic OTA via Arduino IDE

Enable OTA in Arduino IDE

1. Go to `Tools > Port` and select the network port (e.g., `ESP32 at 192.168.x.x`).
2. Upload the sketch using `Sketch > Upload Over The Air`.

Code Example

```
#include <WiFi.h>
#include <ESPmDNS.h>
#include <WiFiUdp.h>
#include <ArduinoOTA.h>

const char* ssid = "YOUR_WIFI_SSID";
const char* password = "YOUR_WIFI_PASSWORD";

void setup() {
  Serial.begin(115200);
  WiFi.begin(ssid, password);

  while (WiFi.status() != WL_CONNECTED) {
    delay(1000);
    Serial.println("Connecting to WiFi...");
  }

  Serial.println("Connected to WiFi");

  // Initialize OTA
  ArduinoOTA.begin();

```

```
  // OTA Event Handlers
  ArduinoOTA.onStart([]() {
    Serial.println("OTA Update Started");
  });

  ArduinoOTA.onEnd([]() {
    Serial.println("OTA Update Finished");
  });

  ArduinoOTA.onProgress([](unsigned int progress, unsigned int total) {
    Serial.printf("Progress: %u%%\r", (progress / (total / 100)));
  });

  ArduinoOTA.onError([](ota_error_t error) {
    Serial.printf("Error[%u]: ", error);
    if (error == OTA_AUTH_ERROR) Serial.println("Auth Failed");
    else if (error == OTA_BEGIN_ERROR) Serial.println("Begin Failed");
    else if (error == OTA_CONNECT_ERROR) Serial.println("Connect Failed
   ");
    else if (error == OTA_RECEIVE_ERROR) Serial.println("Receive Failed
   ");
    else if (error == OTA_END_ERROR) Serial.println("End Failed");
  });
}

void loop() {
  ArduinoOTA.handle(); // Handle OTA updates
}
```

Listing 16.1: Basic OTA Example

Steps to Upload OTA

1. Compile and upload the above code to the ESP32 via USB.
2. Disconnect the USB cable.
3. Select the OTA port in Arduino IDE and upload a new sketch wirelessly.

16.3.3 OTA via HTTP Server

Set Up an HTTP Server

Host the firmware binary (`.bin` file) on an HTTP server (e.g., Apache, Nginx, or a cloud storage service).

Code Example

```
#include <WiFi.h>
#include <HTTPClient.h>
#include <Update.h>

const char* ssid = "YOUR_WIFI_SSID";
const char* password = "YOUR_WIFI_PASSWORD";
const char* firmwareUrl = "http://your-server.com/firmware.bin";
```

```
8
9  void setup() {
10   Serial.begin(115200);
11   WiFi.begin(ssid, password);
12
13   while (WiFi.status() != WL_CONNECTED) {
14     delay(1000);
15     Serial.println("Connecting to WiFi...");
16   }
17
18   Serial.println("Connected to WiFi");
19
20   // Check for updates
21   checkForUpdates();
22 }
23
24 void loop() {
25   // Your main code here
26 }
27
28 void checkForUpdates() {
29   HTTPClient http;
30   http.begin(firmwareUrl);
31   int httpCode = http.GET();
32
33   if (httpCode == HTTP_CODE_OK) {
34     int contentLength = http.getSize();
35     if (contentLength > 0) {
36       if (Update.begin(contentLength)) {
37         Serial.println("Starting OTA Update...");
38         size_t written = Update.writeStream(http.getStream());
39
40         if (written == contentLength) {
41           Serial.println("Update Complete");
42           if (Update.end()) {
43             Serial.println("Rebooting...");
44             ESP.restart();
45           } else {
46             Serial.println("Update Failed");
47           }
48         } else {
49           Serial.println("Write Failed");
50         }
51       } else {
52         Serial.println("Not Enough Space");
53       }
54     } else {
55       Serial.println("Invalid Content Length");
56     }
57   } else {
58     Serial.println("Failed to Download Firmware");
59   }
60
61   http.end();
62 }
```

Listing 16.2: HTTP OTA Example

Steps

1. Compile and upload the code to the ESP32.
2. Place the new firmware binary on the HTTP server.
3. The ESP32 will check for updates and download/install the new firmware.

16.4 Advanced OTA with Cloud Services

16.4.1 Cloud-Based OTA

Use services like AWS IoT, Firebase, or Azure IoT for OTA updates. These services provide better security, scalability, and management capabilities.

16.4.2 Example: AWS IoT OTA

Set up an AWS IoT job to push firmware updates to the ESP32. Use the AWS IoT SDK for ESP32 to handle OTA updates.

16.5 Best Practices for OTA

16.5.1 Security

- Use HTTPS for firmware downloads.
- Sign firmware updates with a private key and verify the signature on the device.

16.5.2 Reliability

- Implement rollback mechanisms in case of failed updates.
- Use a stable Wi-Fi connection.

16.5.3 Testing

Test updates thoroughly before deploying to production devices.

16.6 Common Issues and Troubleshooting

16.6.1 OTA Fails

- Ensure the firmware binary is correct and matches the device.
- Check Wi-Fi connectivity and server availability.

16.6.2 Insufficient Space

Optimize the firmware size or use a larger flash memory ESP32 variant.

16.6.3 Bootloader Issues

Ensure the correct partition table is used.

16.7 Conclusion

OTA is a powerful feature for ESP32 devices, enabling remote updates and maintenance. Start with basic OTA via Arduino IDE, then explore HTTP-based and cloud-based OTA for advanced use cases. Always prioritize security and reliability when implementing OTA updates.

Additional Resources

- ESP32 OTA Documentation
- Arduino OTA Library
- AWS IoT OTA Documentation

Chapter 17

IoT Networking Protocols

17.1 Introduction to IoT Networking

17.1.1 What is IoT Networking?

IoT networking refers to the communication technologies and protocols that enable IoT devices to connect, share data, and interact with each other and the cloud. The choice of networking protocol depends on factors like range, power consumption, data rate, and use case.

17.1.2 Key Considerations for IoT Networking

- **Range**: How far the signal needs to travel.
- **Power Consumption**: Battery life and energy efficiency.
- **Data Rate**: Speed of data transmission.
- **Cost**: Hardware and operational costs.
- **Scalability**: Number of devices supported.
- **Latency**: Delay in data transmission.
- **Security**: Encryption and authentication mechanisms.

17.2 Common IoT Networking Protocols

17.2.1 Wi-Fi

Description: Wi-Fi is a widely used wireless networking protocol based on the IEEE 802.11 standard, operating in the 2.4 GHz and 5 GHz frequency bands.

Pros:

- High data rates (up to several Gbps with Wi-Fi 6).
- Wide availability and compatibility.
- Easy to set up and use.

Cons:

- High power consumption (not ideal for battery-operated devices).
- Limited range (typically up to 100 meters).
- Congestion in dense environments.

Use Cases: Smart homes, industrial automation, and applications requiring high data throughput.

17.2.2 Bluetooth (Classic and BLE)

Description: Bluetooth is a short-range wireless protocol operating in the 2.4 GHz band. Bluetooth Low Energy (BLE) is optimized for low-power applications.

Pros:

- Low power consumption (especially BLE).
- Low cost and widespread adoption.
- Suitable for short-range communication (up to 100 meters).

Cons:

- Limited data rate (up to 2 Mbps for Bluetooth 5.0).
- Limited range compared to other protocols.

Use Cases: Wearables, health monitoring, and smart home devices.

17.2.3 Zigbee

Description: Zigbee is a low-power, low-data-rate wireless protocol based on the IEEE 802.15.4 standard, operating in the 2.4 GHz band with mesh networking support.

Pros:

- Low power consumption.
- Supports mesh networking (extended range and reliability).
- Scalable for large networks.

Cons:

- Low data rate (up to 250 kbps).
- Limited range for individual nodes (10-100 meters).

Use Cases: Smart lighting, home automation, and industrial control systems.

17.2.4 LoRaWAN

Description: LoRaWAN (Long Range Wide Area Network) is a low-power, long-range protocol designed for IoT applications, operating in sub-GHz frequency bands.

Pros:

- Very long range (up to 10 km in rural areas).
- Low power consumption (battery life of several years).
- Supports large-scale networks.

Cons:

- Low data rate (up to 50 kbps).
- Limited bandwidth for high-data applications.

Use Cases: Smart agriculture, environmental monitoring, and asset tracking.

17.3 Factors to Consider When Choosing a Networking Option

- **Range:** Short range (Bluetooth, Zigbee, Z-Wave), Medium range (Wi-Fi), Long range (LoRaWAN, Cellular).
- **Power Consumption:** Low power (BLE, Zigbee, LoRaWAN, LTE-M, NB-IoT), High power (Wi-Fi, Cellular).
- **Data Rate:** Low (LoRaWAN, Zigbee, Z-Wave), High (Wi-Fi, Ethernet, 5G).
- **Cost:** Low (Bluetooth, Zigbee), High (Cellular, Ethernet).
- **Scalability:** Small networks (Bluetooth, Z-Wave), Large networks (Zigbee, LoRaWAN, Cellular).
- **Latency:** Low (Wi-Fi, Ethernet, 5G), High (LoRaWAN, LTE-M, NB-IoT).
- **Security:** High (Cellular, Wi-Fi with WPA3, Ethernet), Moderate (Zigbee, Z-Wave, LoRaWAN).

17.4 Comparison Table of IoT Networking Protocols

Protocol	Range (meters)	Power	Data Rate	Cost	Scalability	Latency	Use Cases
Wi-Fi	50-100	High	High	Medium	Medium	Low	Smart homes, industrial IoT
Bluetooth	1-100	Low (BLE)	Low	Low	Small	Low	Wearables, health monitoring
Zigbee	10-100	Low	Low	Low	Large	Medium	Smart lighting, home automation
LoRaWAN	1000-10000	Very Low	Very Low	Medium	Large	High	Smart agriculture, asset tracking
Cellular	1000-35000	High (except LTE-M/NB-IoT)	High (5G)	High	Large	Low (5G)	Smart cities, connected vehicles
Z-Wave	30-100	Low	Low	Medium	Small	Medium	Smart home devices
Ethernet	Wired	Medium	Very High	High	Large	Very Low	Industrial automation, data centers

Table 17.1: Comparison of IoT Networking Protocols

17.5 Conclusion

The choice of IoT networking protocol depends on the specific requirements of the application, such as range, power consumption, data rate, and cost. Wi-Fi and Bluetooth are ideal for short-range, high-data-rate applications, while LoRaWAN and cellular are better suited for long-range, low-power use cases. Scalability, latency, and security are critical considerations when selecting a networking option.

17.6 Additional Resources

- `https://www.wi-fi.org/`
- `https://www.bluetooth.com/`
- `https://lora-alliance.org/`
- `https://zigbeealliance.org/`
- `https://www.3gpp.org/`

s