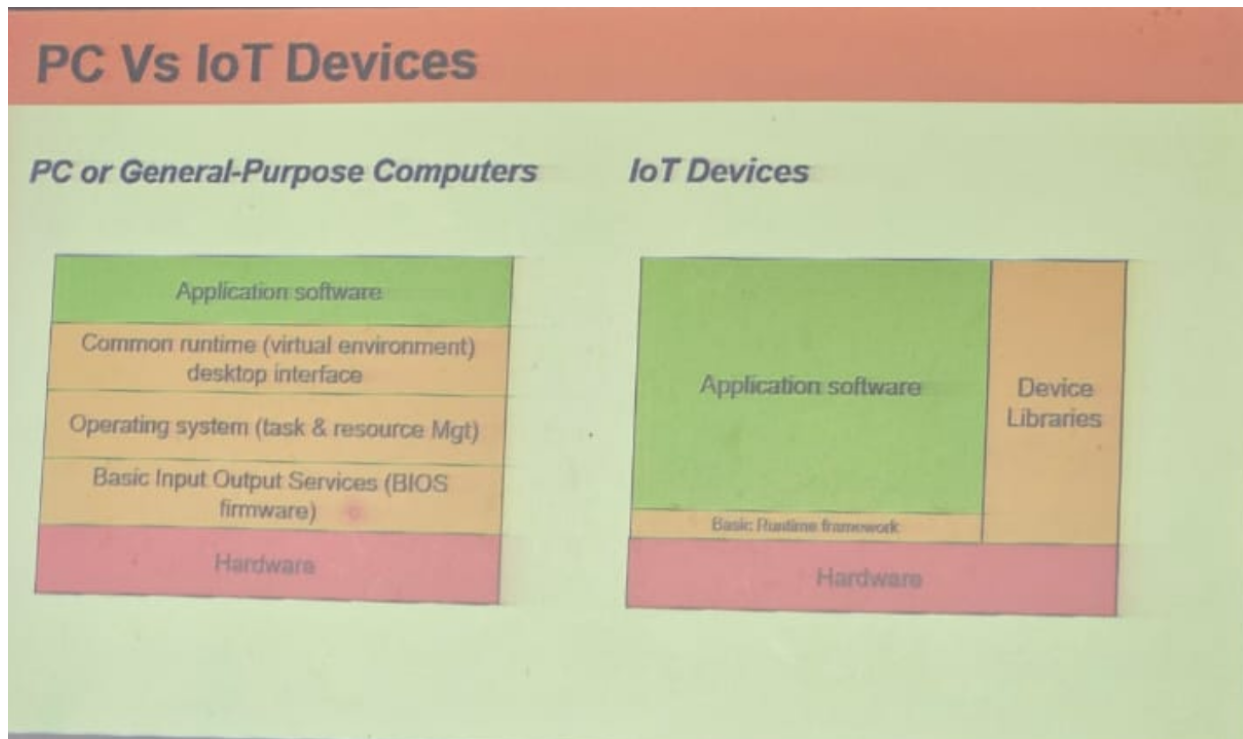


IOT Software

PC Vs IOT Devices



1. General-Purpose Computers (PCs)

PCs are designed for a wide range of tasks, such as software development, gaming, browsing, and multimedia processing. Their architecture is more complex to support multi-tasking and a variety of applications.

PC Software and System Components:

- **Application Software:** Runs user applications like browsers, word processors, games, and development tools.
- **Common Runtime Environment:** Supports virtual environments and desktop interfaces (e.g., Windows Explorer, macOS GUI).
- **Operating System (OS):** Handles task management, memory allocation, security, and hardware abstraction. Examples: Windows, Linux, macOS.
- **BIOS/Firmware (Basic Input/Output System):** Initializes hardware at startup and loads the OS. Helps to get rid of the difficulty of managing hardware.
- **Hardware:** Includes CPU, RAM, storage, GPU, motherboard, and peripherals.

Key Characteristics of PCs:

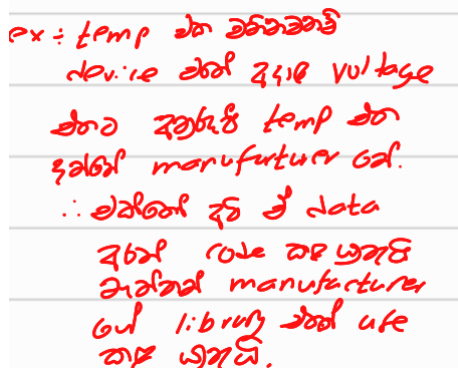
- ✓ Designed for versatility and high performance.
 - ✓ Multi-tasking and multi-user capabilities.
 - ✓ High processing power, memory, and storage.
 - ✓ Supports heavy applications like video editing and AI modeling.
-

2. IoT Devices (Internet of Things)

IoT devices are designed for specific tasks like sensor data collection, automation, and real-time operations. They operate with minimal resources and often rely on wireless communication.

IoT Software and System Components:

- **Application Software:** Runs lightweight programs for data collection, processing, and automation (e.g., temperature monitoring, smart locks, fitness trackers).
- **Device Libraries:** Pre-compiled code to interact with hardware (e.g., sensor drivers, communication protocols like MQTT).



ex: temp નો સંચાર
device નો 3.3v voltage
નો સંચાર temp નો
સંચાર manufacturer code.
∴ એકલો ફો ડાટા
સંચાર code નો ઉપયોગ
આવડે manufacturer
નો library નો ઉપયોગ
નો ઉપયોગ.

- **Basic Runtime Framework:** Instead of a full OS, many IoT devices use real-time operating systems (RTOS) or firmware-based execution to manage tasks efficiently. **This will not manage the hardware. This will only help to start the program.**

In IoT devices, a **Basic Runtime Framework** is a lightweight software layer responsible for managing the execution of programs, handling hardware interactions, and sometimes providing minimal scheduling and memory management. It is **much simpler than a full operating system** but still offers essential functionalities for embedded systems.

- **Hardware:** Typically includes microcontrollers (MCUs), sensors, network modules (Wi-Fi, Bluetooth), and actuators.

Key Characteristics of IoT Devices:

- ✓ Optimized for low power consumption and efficiency.
 - ✓ Designed for real-time operations with minimal latency.
 - ✓ Often lacks a full OS (uses firmware or RTOS).
 - ✓ Smaller storage and computing power compared to PCs.
 - ✓ Connects to the cloud or other devices for data exchange.
-

3. Key Differences Between PCs and IoT Devices

Feature	General-Purpose Computers (PCs)	IoT Devices
Operating System	Full OS (Windows, Linux, macOS)	Minimal OS, RTOS, or firmware
Task Handling	Multi-tasking, multi-user	Single-task or real-time processing
Processing Power	High (multi-core CPUs, GPUs)	Low (microcontrollers, embedded chips)
Power Consumption	High (requires external power)	Low (battery-operated or low power)
Storage	Large HDD/SSD (GBs to TBs)	Minimal (KBs to MBs, SD cards, flash memory)
Connectivity	Ethernet, Wi-Fi, Bluetooth, USB	Wireless (Wi-Fi, Zigbee, LoRa, NB-IoT)
User Interface	GUI-based (mouse, keyboard)	Often headless or simple interfaces

4. Why Do IoT Devices Use a Different Architecture?

- **Efficiency:** IoT devices must function with limited power and processing.
 - **Reliability:** Many IoT applications require real-time response, so they avoid full-fledged OS overhead.
 - **Security:** Lightweight frameworks reduce attack surfaces, making them more secure.
-

Hardware and software interactions

Hardware & Software Interactions

General-Purpose software

- Hardware resources are managed by the OS and firmware components
- Often presented to the programmer as a device independent virtual environment.
- Your program access the "virtual" device using high-level commands.
 - You can send a MQTT message without worrying about the type of network connection.
 - You can read the ASCII code from the keyboard without considering its layout.
 - You can print on screen without considering its resolution or size.
 - You can open a file without known where it is stored in the disk.

IoT Device software

- Often your program must manage hardware directly, using low level commands.
 - GPIO to control individual device ports
 - ADC/DAC and PWM to work with analog devices
 - I/O protocols like SPI, I2C, UART, etc. to transfer data to/from peripheral sensor modules
 - Memory management and Flash storage management
- Application software (sometimes with help from a pre-written library) must manage all resources.
 - You must establish / check connectivity before sending a MQTT message.
 - You must translate key-switch status to key-codes /ASCII code.
 - Your IO procedures often depend on the specific hardware being used.

1. General-Purpose Software (PCs)

- Hardware is **abstracted** by the **Operating System (OS)** and **firmware**.
- Provides a **virtual environment** for programmers, making hardware details invisible.
- Programs interact with "virtual" devices using high-level commands:
 - Sending an **MQTT message** without worrying about the network type.
 - Reading ASCII input from the keyboard **without knowing the keyboard layout**.
 - Printing text **without handling screen resolution**.
 - Opening files **without knowing their exact location on the disk**.

◆ **Key Benefit: Developers don't need to worry about hardware details—everything is managed by the OS.**

2. IoT Device Software

- The program must manage **hardware directly**, using **low-level commands** such as:
 - **GPIO (General-Purpose Input/Output):** To control specific device ports.
 - **ADC (Analog-to-Digital Converter) / DAC (Digital-to-Analog Converter):** For analog signal processing.
 - **PWM (Pulse Width Modulation):** Used to control motors, LEDs, etc.
 - **I/O Protocols (SPI, I2C, UART):** Used for **sensor communication**.
 - **Memory and Flash Management:** No virtual memory—programmers must manage storage efficiently.

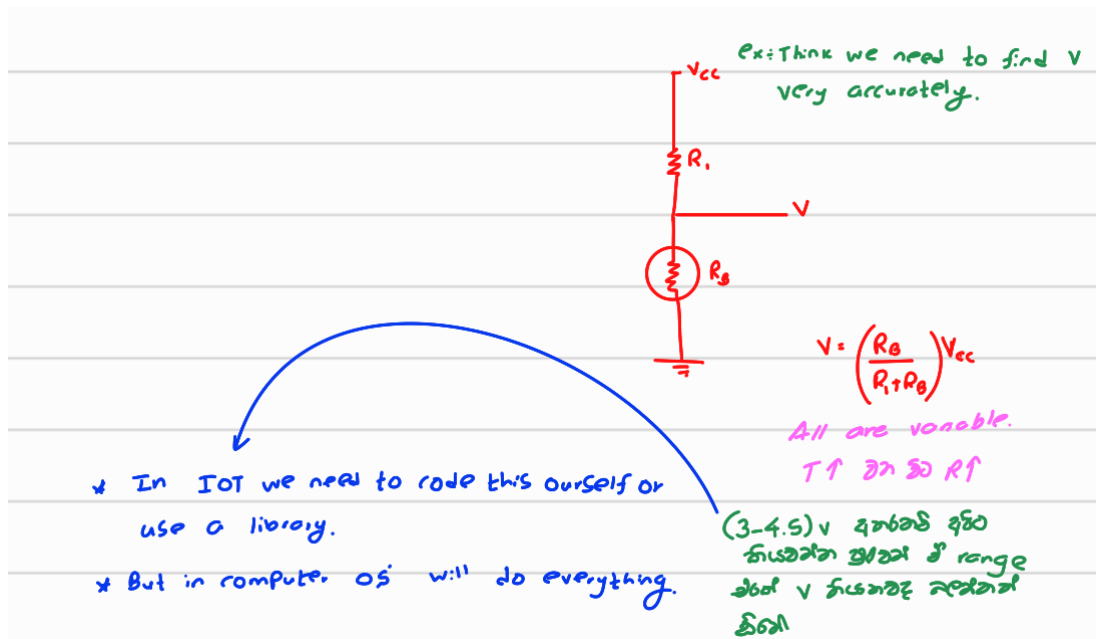
- **Application software must manually handle resources:** sometimes with the help from a pre-written library
 - **MQTT communication:** You need to **manually check connectivity** before sending messages.
 - **Keyboard input processing:** Raw key-switch data must be **translated into ASCII codes**.
 - **I/O operations depend on the hardware:** Unlike PCs, where printing text is abstracted, IoT devices require **specific configurations**.

◆ **Key Challenge: IoT developers must directly interact with hardware, requiring more low-level programming and manual resource management.**

3. Key differences

Feature	General-Purpose Software (PCs)	IoT Device Software
Hardware Abstraction	OS manages hardware	Programmer manages hardware
Programming Complexity	High-level APIs simplify development	Requires direct hardware control
Device Communication	Automatically handled	Needs manual configuration (SPI, I2C, UART)
Memory Management	Virtual memory and OS-level management	Manual memory and flash storage management
Ease of Use	Easier for developers	Requires embedded programming knowledge

- **PC software is user-friendly** because the OS abstracts hardware complexities.
- **IoT software requires low-level programming** because the developer must manually control hardware.
- If you're working with IoT, you'll often **write C or Python** for microcontrollers (ESP32, STM32, Raspberry Pi, etc.) and use **protocols like MQTT, I2C, and SPI**.



Cross platform development

1. Development on a PC (Traditional Software Development)

- Everything happens in **one environment** (coding, compiling, testing, and execution).
- Advanced debugging tools

available:

- Breakpoints & stepping through code:** Allows developers to pause execution and inspect variables.
- Debug outputs (print statements):** Easy to track program flow.
- Logging console:** Collects real-time logs for troubleshooting.
- Test scripts:** Automate testing.

✓ Why it's easy?

Because the **development environment and execution environment are the same**.

2. Development on IoT Devices (Embedded Systems)

- Development & execution occur in different environments :
 - Code is written & compiled on a PC (cross-compilation).
 - The compiled binary is deployed to the **IoT device** (like an ESP32, Raspberry Pi, or STM32).

Challenges in IoT Development

1. Limited debugging tools:

- No display** for direct debug output.
- No breakpoints or real-time stepping** (like in PC-based debugging).
- Debugging often requires **external hardware tools** (e.g., JTAG, logic probes).

2. Monitoring difficulties:

- Unlike a PC, where errors and logs appear in a console, IoT devices might require **serial debugging (UART/USB)** or logging to external files.
- Some IoT devices have **real-time constraints**, making debugging harder.

3. Hardware-dependent debugging:

- Debugging might need **oscilloscopes, logic analyzers, or JTAG programmers**.
- Debugging in low-power or networked IoT systems often requires **remote logging mechanisms**.

How to Debug IoT Devices Efficiently?

Here are **some workarounds** to make debugging easier in IoT projects:

1. Serial Debugging (UART)

- Most microcontrollers (ESP32, Arduino, STM32, etc.) support **UART-based debugging**.
- Example for ESP32 using Serial Monitor:

```
void setup() {  
    Serial.begin(115200); // Start serial communication  
    Serial.println("Device Started!");  
}  
  
void loop() {  
    Serial.println("Reading sensor data...");  
    delay(1000);  
}
```

- Output appears on the **serial monitor** in the Arduino IDE or PuTTY.

2. Remote Logging

- Instead of using a display, log data to a file or send it over **Wi-Fi, MQTT, or HTTP**.

3. JTAG Debugging

- JTAG (Joint Test Action Group) allows **hardware debugging** (used in ARM Cortex-based chips like STM32).
- Requires an **external debugger** like the **Segger J-Link**.

4. Use Emulators/Simulators

- Some IoT devices have emulators, such as:
 - **QEMU** (for ARM Cortex IoT devices).
 - **ESP-IDF simulator** (for ESP32).

- **Arduino Serial Plotter** (for sensor data visualization).

Simulation
through
debugg-
ing. we
need to
simulate the
environment
as well.
i.e. we need
to debug
(software +
hardware.)
2/5

Key Takeaways

Feature	PC Development	IoT Development
Coding & Compilation	Same machine	Cross-compilation (PC → IoT)
Debugging Tools	Breakpoints, stepping, console logs	Limited (requires UART, JTAG, external tools)
Execution Monitoring	Display output, logging	Often headless (no direct display)
Ease of Debugging	Easy with IDE tools	Harder, requires external probes

Features of IOT programming languages

1. Lightweight and Efficient

- IoT devices have limited processing power and memory, so programming languages must be optimized for small footprints.
- The code should be simple, fast, and consume minimal system resources to ensure smooth performance on embedded systems.

2. Embeddable

- IoT programming languages should integrate well with other environments, such as firmware or real-time operating systems.

- They should allow low-level hardware access for controlling sensors, actuators, and communication modules.

3. Event-Driven Focus

- IoT systems often rely on asynchronous event handling, where the system responds to external triggers like sensor inputs, network signals, or user commands.
- Features like timers, callbacks, and interrupt handling make the system responsive and efficient.

4. Extensible

- Modular programming allows flexibility in adding new features or integrating with third-party libraries.
- It supports different applications, from executing machine code on microcontrollers to running configuration scripts on cloud-connected devices.

5. High-Level Abstractions

- A good IoT language provides a simple and maintainable syntax while supporting multiple programming paradigms:
 - **Procedural** (structured step-by-step execution)
 - **Functional** (stateless, reusable functions)
 - **Data-flow** (event-based or stream processing)
 - **Object-oriented** (modular and reusable components)
 - These abstractions improve code readability and scalability, making development easier.
-

IoT - Popular runtimes and programming languages

A **programming language** defines how code is written, while a **runtime** is the environment that executes the code. The runtime provides the necessary system resources, libraries, and execution support.

For example, MicroPython (runtime) allows Python (language) to run on microcontrollers, and Java (language) requires a Java Virtual Machine (runtime) to execute its bytecode.

1. Arduino

- Based on "Wiring" framework, a C/C++ derivative with simpler syntax.
- Compiles into chip-native code.

2. NodeMCU

- Based on LUA programming language.
- Supports scripting and bytecode compilation.

3. NetCore NanoFramework

- Based on C# language with simplified syntax.
- Uses a small footprint .NetCore runtime library.

4. **MicroPython**

- A simplified version of Python.
- Requires Python firmware as runtime.

5. **Java**

- Based on popular Java and JavaScript (Node.js) languages.
- Compiles into bytecode or runs as a script.

6. **ESP**

- ESP native programming based on its assembly language.
- Runs directly on the chip.

7. **Ubuntu Core**

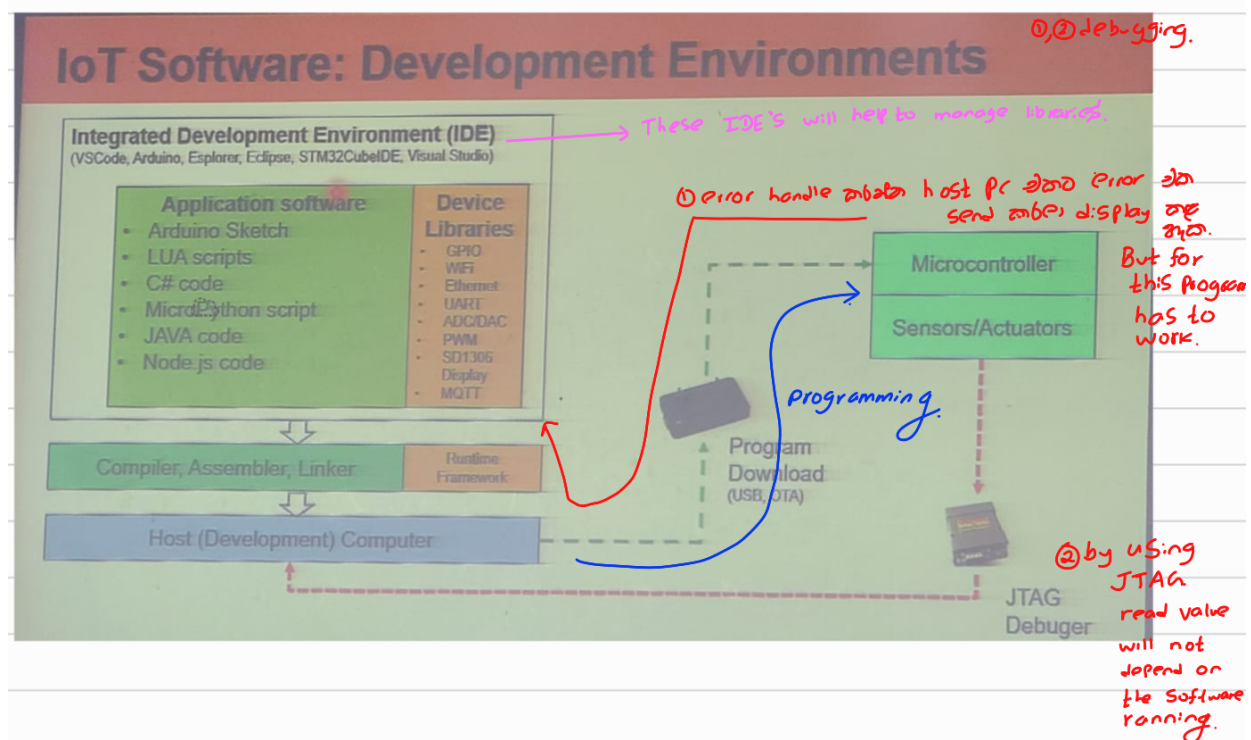
- Provides a scaled-down OS-like environment.
- Can be used with C/C++, Python, etc.

8. **Native Environments**

- Many chip manufacturers have their own assembly language compilers.
- Often runs directly on the chip.

Runtime	Programming Language
Arduino	C/C++ (Wiring framework)
NodeMCU	LUA
NetCore NanoFramework	C#
MicroPython	Python
Java	Java, JavaScript (Node.js)
ESP	Assembly Language
Ubuntu Core	C/C++, Python
Native Environments	Assembly Language (Chip-specific)

IoT - Software Development environment



1. Integrated Development Environment (IDE):

- Examples: VSCode, Arduino, Explorer, Eclipse, STM32CubeIDE, Visual Studio.
- Provides tools for writing, compiling, and debugging code.

2. Application Software:

- Common programming languages used:

3. Device Libraries:

- Software that comes from the manufacturer that is very specific to the device.

4. Compilation and Execution Process:

- Code is compiled, assembled, and linked on the **development computer**.
- A **runtime framework** is used to manage execution.

5. Program Deployment:

- Downloaded onto the microcontroller via **USB** or **Over-the-Air (OTA)** updates.

6. Microcontroller and Sensors/Actuators:

- The compiled code is executed on the microcontroller, interfacing with sensors and actuators.

7. Debugging:

- **JTAG Debugger** is used for hardware debugging and testing.

IoT software development : Key concerns

1. Writing Low-Level Code (C, C++)

- Keep the code simple and as short as possible.

- Avoid complex and time-consuming tasks.

2. Memory Constraints (RAM, ROM)

- Avoid using long arrays and large data structures.
- Use appropriate data types to optimize memory usage.

3. Real-Time Requirements and Scheduling

- Decide between RTOS (Real-Time Operating System) and bare-metal development.

- **RTOS (Real-Time Operating System):** An RTOS is beneficial if the application has multiple tasks with specific timing requirements. It can manage multiple threads, ensure that high-priority tasks are handled immediately, and manage resource allocation efficiently. However, it also introduces overhead.
- **Bare-Metal Development:** Without an RTOS, you need to manage tasks yourself, using interrupt-driven programming and timers. This requires more precise control over task scheduling and synchronization but can be more efficient if done correctly.

◦

* Time sharing task managements are not available.
ex: if you are waiting inside a loop for an input and meanwhile if another incident happen you may miss it.

- Manage tasks and synchronization efficiently if RTOS is not used.

ex: breaking system should be critically responsive more than acceleration.

4. Power Consumption Management

- Utilize sleep modes, wakeup timers, and low-power modes.
- Consider processor and peripheral capability differences across power modes.
- Optimize algorithms to reduce high-power mode usage.

5. Interrupt Handling and Low-Latency Response

- Ensure efficiency in handling interrupts.
- Critical for time-sensitive tasks in IoT applications.

Key Components of an IoT Software

Key Components of a IOT Software

