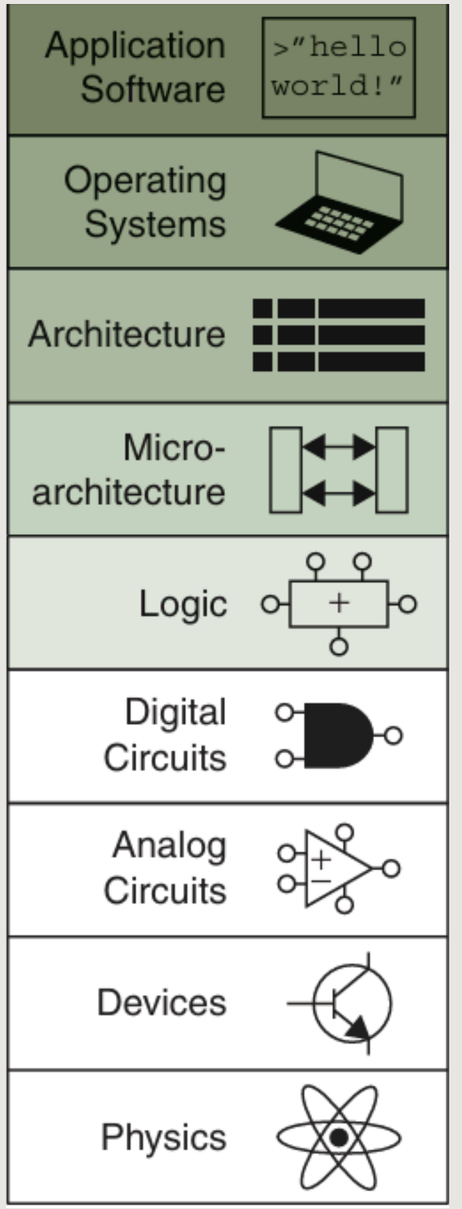
# BACON

*Building embedded Applications with Cloud, MQTT, and Node-RED*

# *THE ART OF ABSTRACTION*

The critical technique for managing complexity is abstraction: hiding details when they are unimportant.

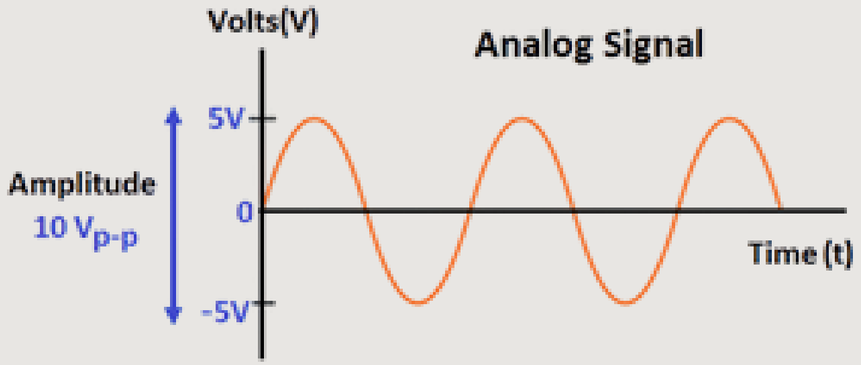
Understanding the major hardware elements of an embedded board is critical to understanding the entire system's architecture, because ultimately, the capabilities of an embedded device are limited or enhanced by what the hardware is capable of.



# *ANALOG AND DIGITAL SIGNALS*

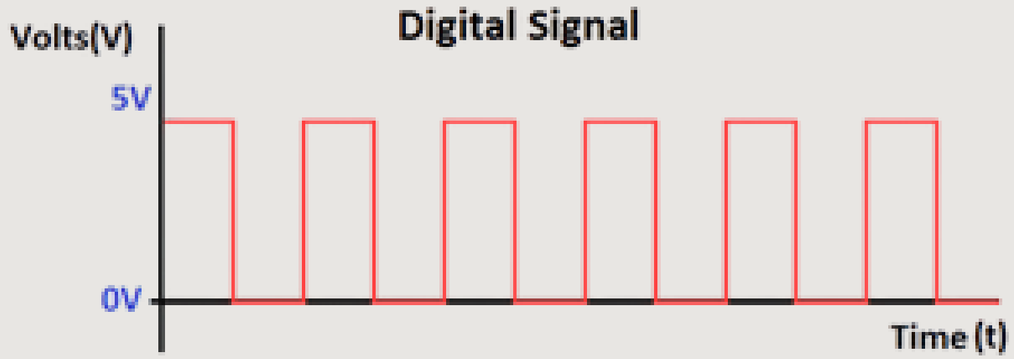
## ANALOG SIGNALS

- The signal can assume an infinite number of amplitude values.
- Real-world signals (e.g., audio, temperature)
- Used in sensors, microphones, and analog meters




## DIGITAL SIGNALS

- The signal can assume only a finite number of amplitude values.
- Used in computers, processors, digital circuits



**Understanding the hardware elements of an embedded board**

# *LEVEL 1: ANALOG*

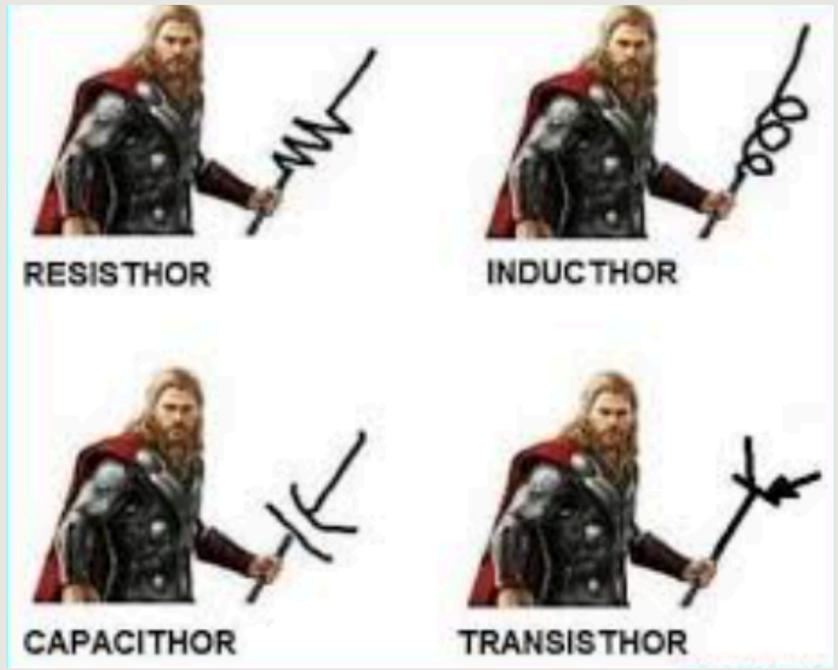
- Concerned with designing circuits that process or interact with real-world signals
- Examples include audio signals, sensors, and power management
- Analog circuitry is integrated with digital logic in chips and embedded systems
- Analog components are needed for signal conditioning and the power-related stuff.

**Understanding the hardware elements of an embedded board**

# *LEVEL 1: ANALOG*

Analog components used :

- Resistors
- capacitors
- inductors
- diodes
- transistors
- mosfets
- operational amplifiers



**ANALOG STUFF**

# RESISTORS, CAPACITORS, INDUCTORS AND DIODES

## Resistors

A resistor limits the flow of electric current. It is a passive component

- Pull-up / Pull-down resistors
- Current Limiting
- Voltage Divider


## Capacitors

Capacitors store and release electrical energy. They oppose changes in voltage

- Filtering
- Timing Circuits


## inductors

Inductors store energy in a magnetic field when current flows through them. They resist changes in current.

- DC-DC Converters
- Power Filtering
- RF Circuits


## Diodes

A diode allows current to flow in only one direction.

- Reverse Polarity Protection
- Clamping and Protection
- Rectification

**ANALOG STUFF**

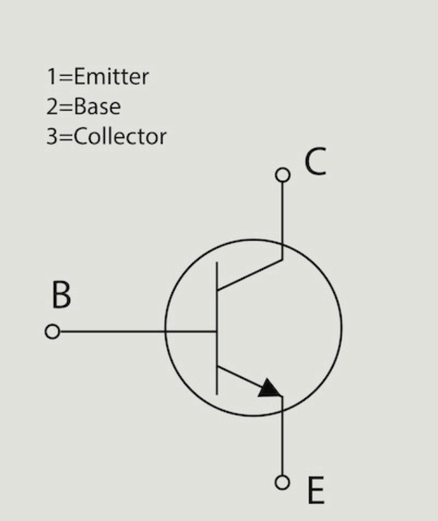
# TRANSITORS AND OPERATIONAL AMPLFIERS

## Transisitor

A transistor is a semiconductor switch or amplifier with three terminals: Base (B), Collector (C), and Emitter (E). It controls current flow using a small input signal.
Types:

- BJT (Bipolar Junction Transistor) – current-controlled
- MOSFET (Metal-Oxide-Semiconductor Field-Effect Transistor) – voltage-controlled

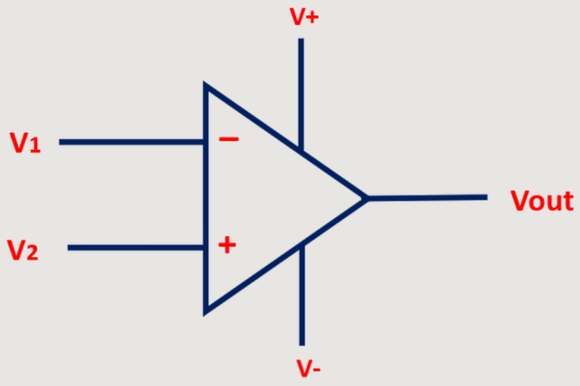
- Switching Loads
- Signal Amplification
- Voltage Regulation & Protection




## Operational Amplifier

An Op-Amp is a high-gain voltage amplifier with differential input (+ and – terminals) and a single-ended output. It is mainly used in analog signal processing.

- Sensor Signal Conditioning
- Comparator Circuits
- Active Filters
- Integrators / Differentiators
- Oscillators and Timers



**ANALOG STUFF**

# MOSFETS

Metal-Oxide-Semiconductor Field-Effect Transistor
It's a type of transistor — basically an electronic switch that can turn things ON or OFF, or control how much current flows.

Gate (G) – Like a button or switch you press.
Drain (D) – Where current flows into the MOSFET.
Source (S) – Where current leaves the MOSFET.

You turn the MOSFET ON by applying a voltage to the Gate.
You turn it OFF by removing that voltage.

**uses:**

- Control Motors, LEDs, Buzzers
- Switching Circuits
- PWM Applications
- Efficient Power Use

**Types of MOSFETs :**
nMOS : Turns ON when Gate voltage is HIGH
pMOS : Turns ON when Gate voltage is LOW

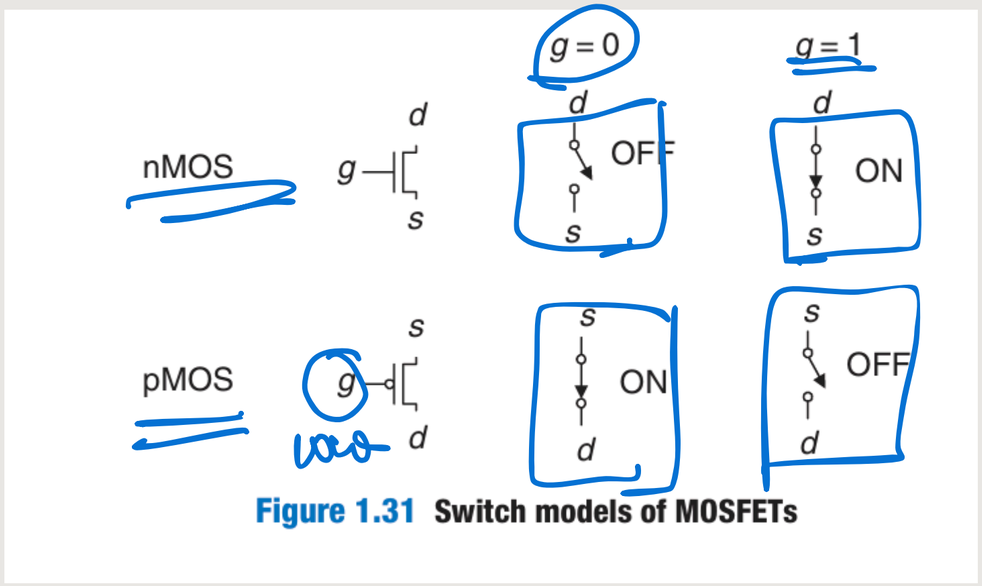



**Figure 1.31** Switch models of MOSFETs

**Understanding the hardware elements of an embedded board**

# *LEVEL 2: DIGITAL ELECTRONICS*

- Only two levels: 0 (LOW) and 1 (HIGH)
- All the digital components are built using analog circuits.
- They perform all the major processing and computation in a system.
- All the storage and memory components are built using digital logic.

**Key components:**

- Clocks
- Combinational
- Sequential

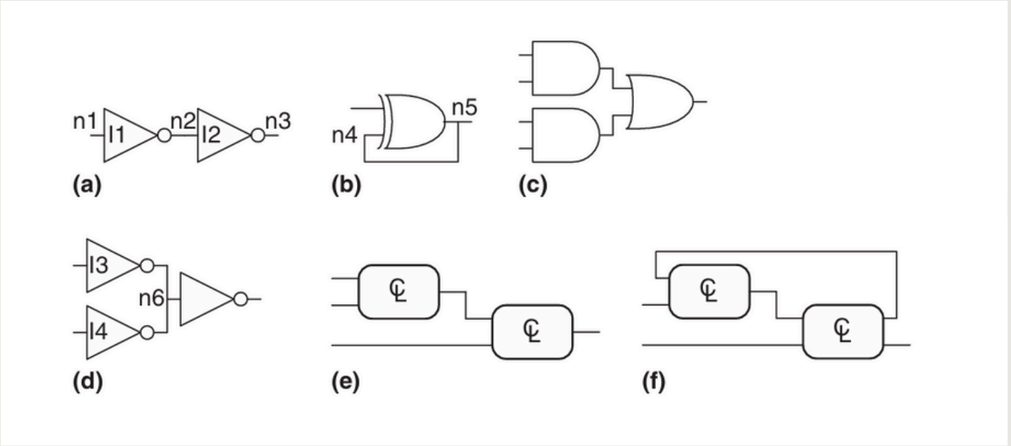
# *combinational and sequential logic*

## COMBINATIONAL

- Combinational Logic: No memory, output depends on inputs only.
- No feedback.
- Required for the general data processing.
- Examples: Adders, multiplexers, decoders.

## SEQUENTIAL

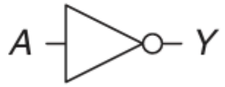
- Sequential Logic: Includes memory, depends on current input and past states.
- Includes feedback.
- Required to build memory and storage elements.
- Examples: Registers, counters, flip-flops.



**DIGITAL**

# LOGIC GATES

## NOT




$Y = \overline{A}$

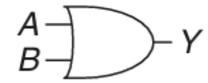
| A | Y |
|---|---|
| 0 | 1 |
| 1 | 0 |

## AND


$Y = AB$

| A | B | Y |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

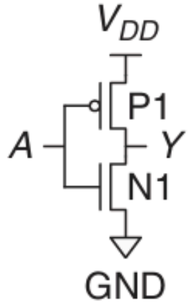
## OR

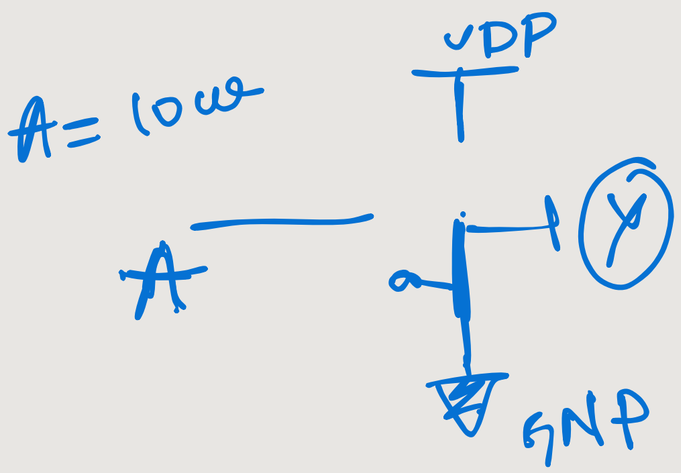



$Y = A + B$

| A | B | Y |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

CMOS IMPLEMENTATION:



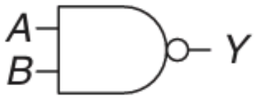

A=low → PMOS → Y → VDD (high)

A=high → NMOS → Y → GND (low)

**DIGITAL**

# LOGIC GATES

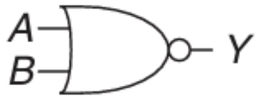
## NAND




$Y = \overline{AB}$

| A | B | Y |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

## NOR




$Y = \overline{A + B}$

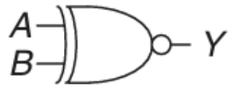
| A | B | Y |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 0 |

## XOR


$Y = A \oplus B$

| A | B | Y |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

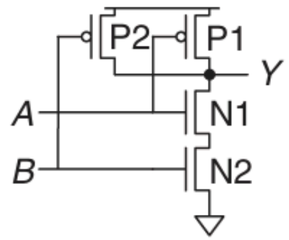
## XNOR

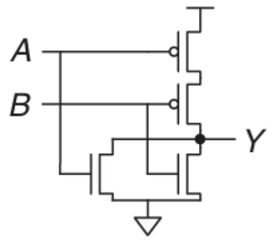



$Y = \overline{A \oplus B}$

| A | B | Y |
|---|---|---|
| 0 | 0 | |
| 0 | 1 | |
| 1 | 0 | |
| 1 | 1 | |

CMOS IMPLEMENTATION:



DIGITAL

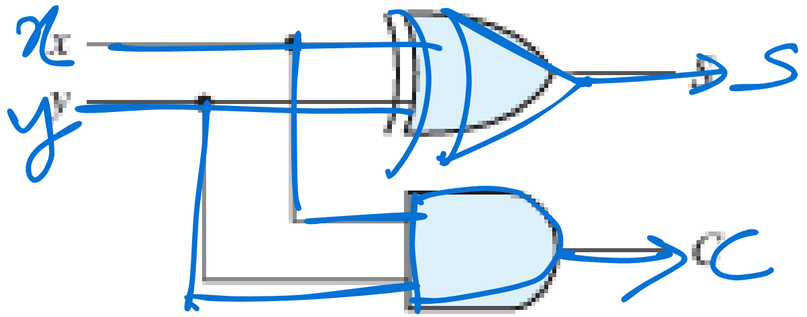
# Combinational building blocks

## HALF ADDER

| x | y | C | S |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 |
| 1 | 1 | 1 | 0 |

10

S → XOR
C → AND



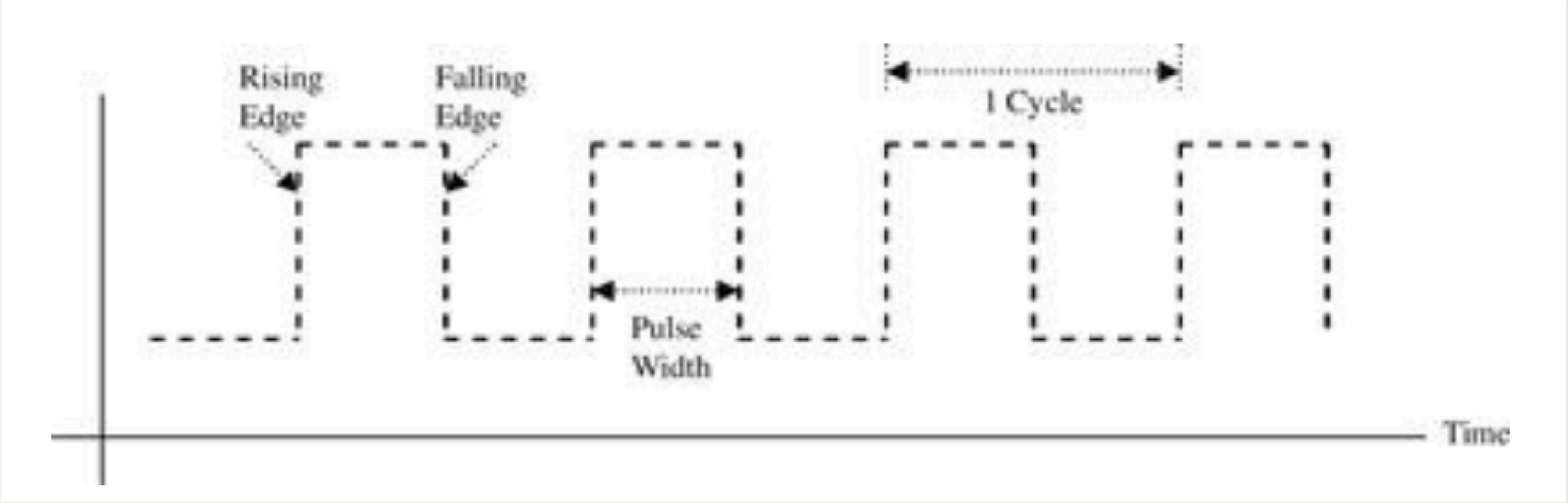

(b) $S = x \oplus y$
$C = xy$

OR y C

**DIGITAL**

# CLOCKS

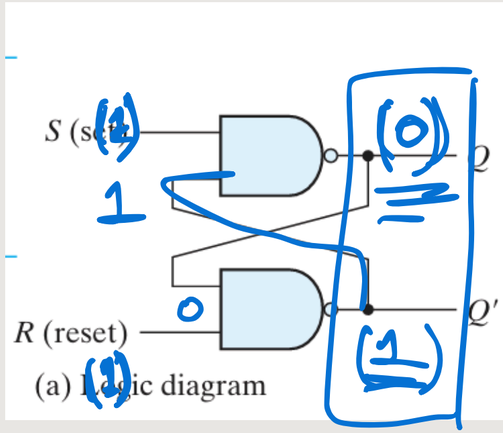
- Provides timing
- Defines when data is read or written



# SEQUENTIAL CIRCUITS

## Latches

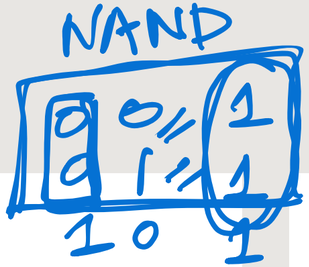
A latch is a basic memory device that can store 1 bit of data.
It is level-sensitive, meaning it responds to the input only when the control (enable) signal is active.

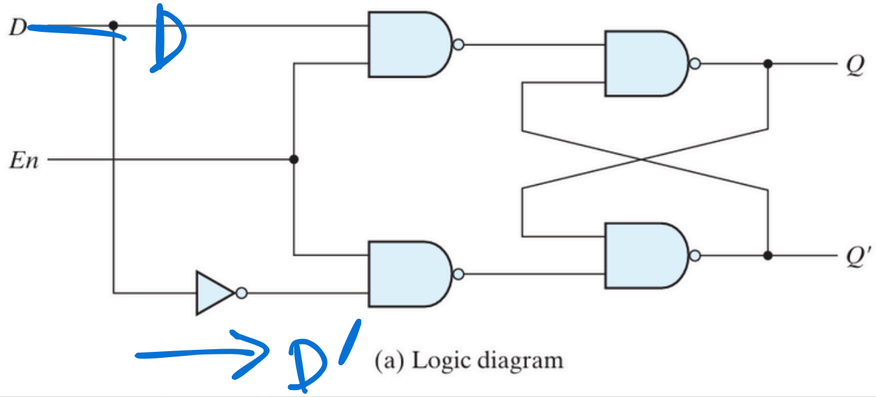



(a) Logic diagram

| S | R | Q | Q′ | |
|---|---|---|---|---|
| 1 | 0 | 0 | 1 | |
| 1 | 1 | 0 | 1 | (after $S = 1, R = 0$) |
| 0 | 1 | 1 | 0 | |
| 1 | 1 | 1 | 0 | (after $S = 0, R = 1$) |
| 0 | 0 | 1 | 1 | (forbidden) |

(b) Function table




(a) Logic diagram

| En | D | Next state of Q |
|---|---|---|
| 0 | X | No change |
| 1 | 0 | $Q = 0$; reset state |
| 1 | 1 | $Q = 1$; set state |

(b) Function table

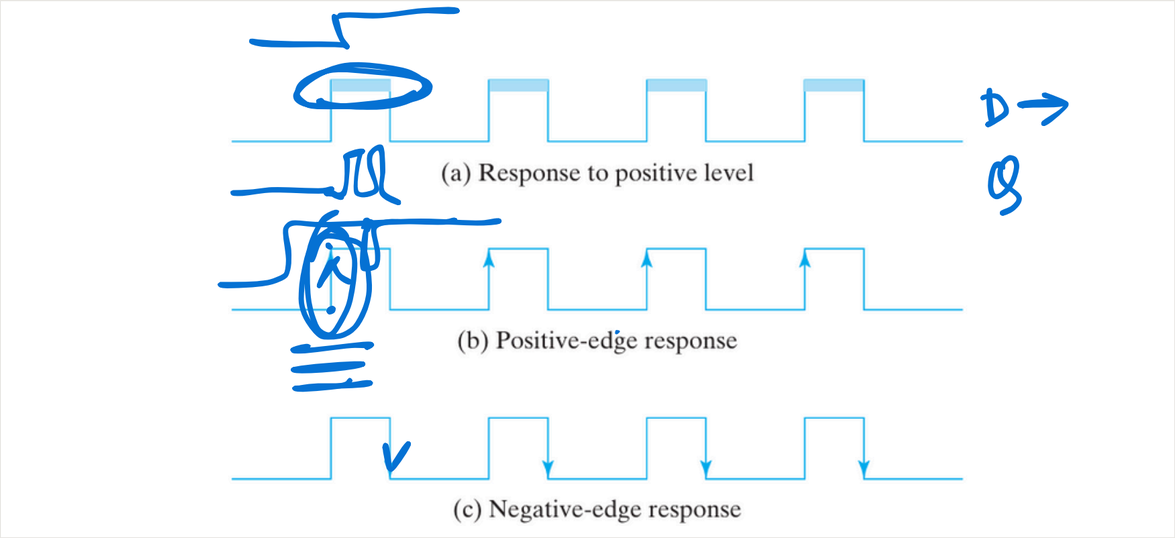
DIGITAL ELECTRONICS

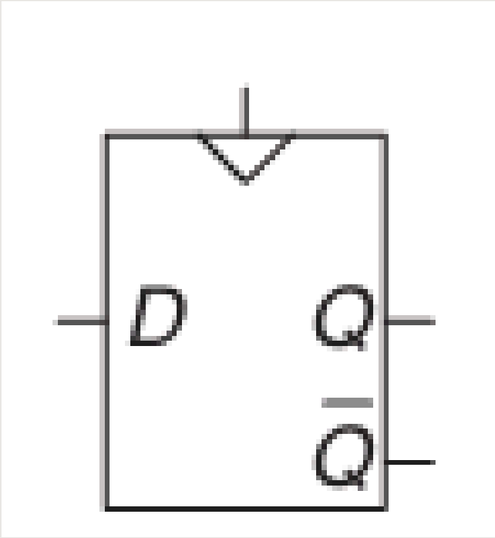
# SEQUENTIAL CIRCUITS

## Flip-Flops

A flip-flop is like an improved latch — it also stores 1 bit, but it's edge-triggered, meaning it only updates its value at the rising or falling edge of a clock signal.

Building block of registers, counters, and memory.



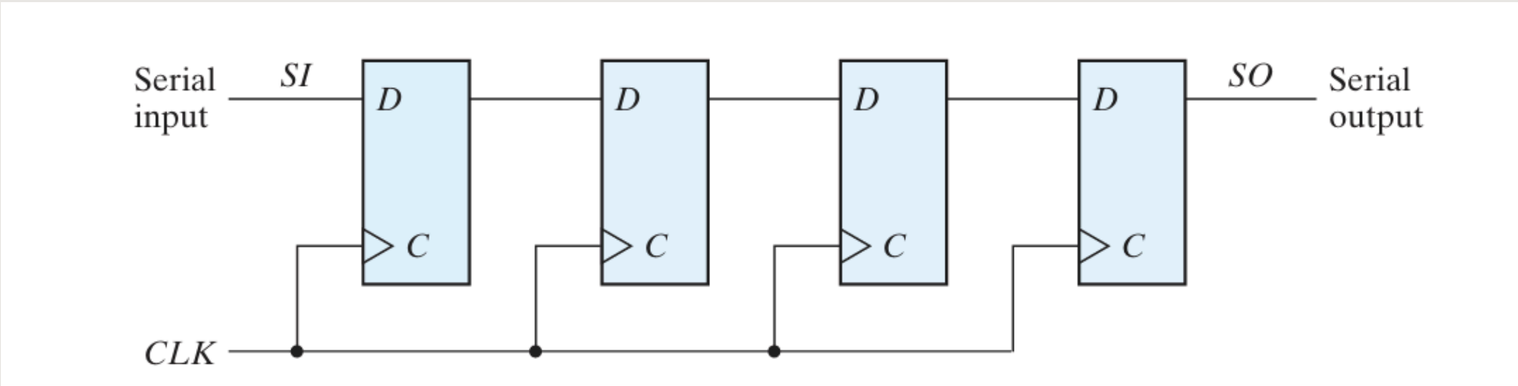
**DIGITAL ELECTRONICS**

# SEQUENTIAL CIRCUITS

## Registers

A register is a group of flip-flops connected together to store multiple bits (e.g., 8-bit, 16-bit).
Registers often include control logic for loading, clearing, and shifting data.

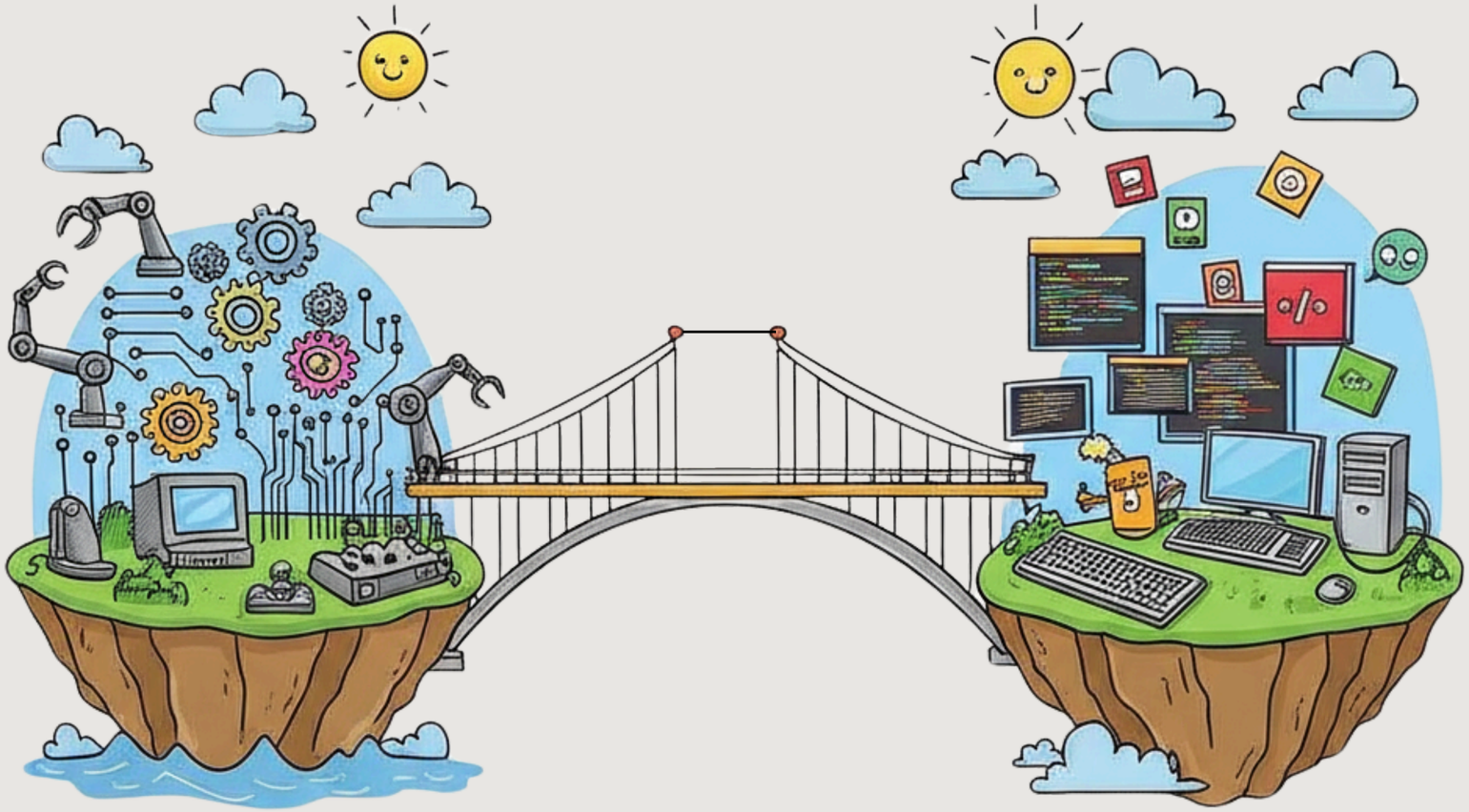
Store data, addresses, flags, or instructions in microcontrollers.



# How analog and digital work together

- The current chips are a combination of analog and digital circuits
- Analog reads and conditions real-world signals
- Digital processes, stores, and transmits information either as digital or analog signals
- Sensors give analog signals.
- Signals are then conditioned and digitized (ADC).
- Digital logic processes and stores the data or transmits the output.
- This is how real-world chipsets and embedded systems function

# *Intro to Embedded Systems*



# *Embedded Systems*

It's a combination of hardware and software intended to perform specific task.

It is generally a part of a larger system.

Embedded system has computer hardware (such as CPU, ROM, RAM etc.) with the software embedded in it .

# *A combination of three aspects*

1

## Sense

Collection of data or input from the device’s surroundings using sensors.

Sensors detect real-world conditions like temperature, motion, light, or pressure.

2

## Process

Microprocessors or microcontrollers analyze the input and makes decisions based on logic.

Runs embedded code to process the sensor data.

May apply conditions, filters, or algorithms.

3

## Actuate

System performs a physical action or communicates results based on the processed data.

Controls motors, LEDs, buzzers, displays, or sends data wirelessly.


Is this an embedded system?

## IS THIS AN EMBEDDED SYSTEM?

**Yes**

It senses motion using an optic sensor, processes it in real-time, and sends precise signals to the computer.

## IS THIS AN EMBEDDED SYSTEM?

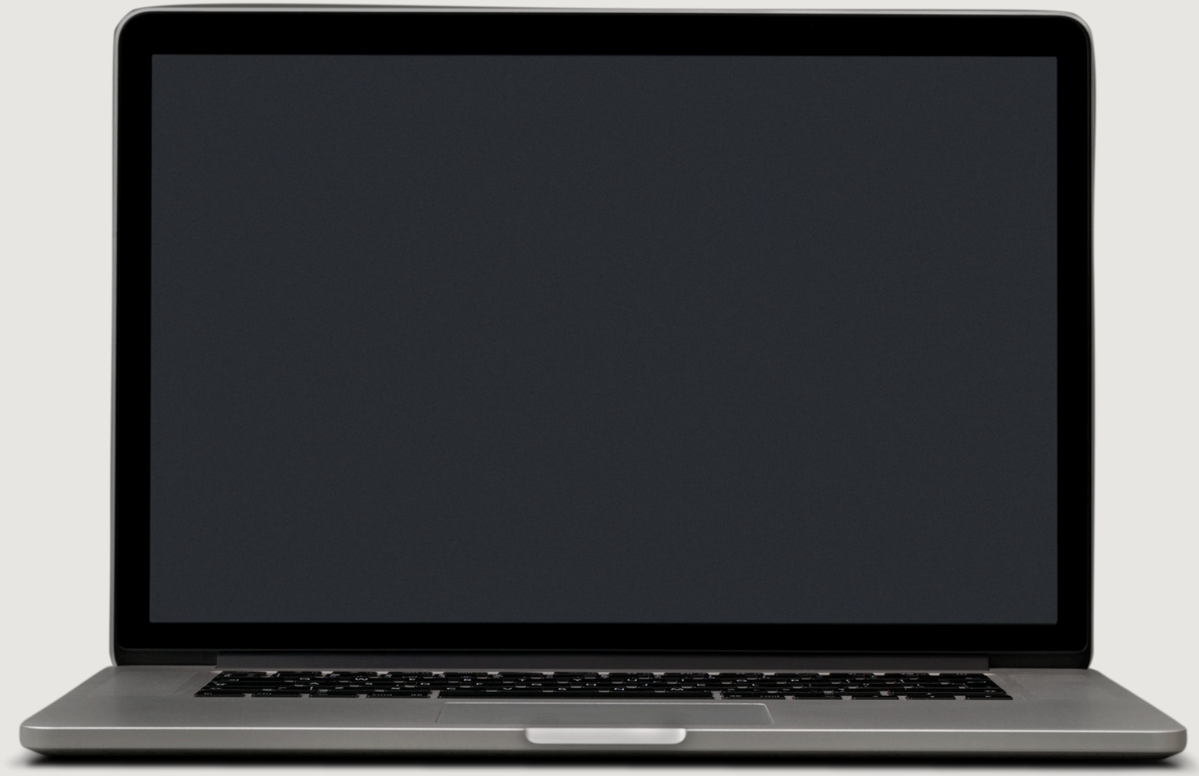
# No

No microcontroller, no software or logic.
Purely electrical — not programmable.

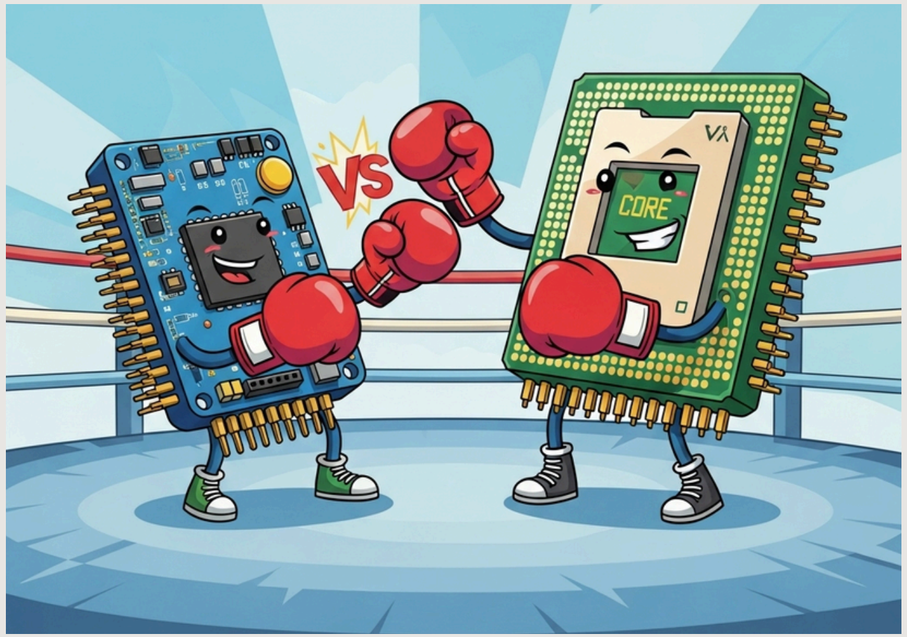
## IS THIS AN EMBEDDED SYSTEM?

**No**

It is a general-purpose computer, not a dedicated-function device. However, it does contain embedded systems like the touchpad or keyboard controller.



# *Microcontrollers vs Microprocessors*



# Microcontrollers

- Includes a CPU, memory (RAM and ROM), and (I/O) peripherals all on a single chip.
- Used in embedded systems like appliances
- Generally lower power consumption
- Typically less expensive than microprocessors due to their simpler architecture.
- Limited memory capacity; often comes with onboard memory
- Designed for specific tasks; not as versatile as microprocessors

# Microprocessors

- Contains a CPU; requires external components for memory and I/O (e.g., RAM, ROM).
- Used in personal computers, servers
- Generally higher power consumption.
- Typically more expensive due to their advanced capabilities
- Relies on external memory.
- More versatile and capable of handling multiple tasks simultaneously

*How do we write code for embedded systems?*

# *C for Embedded Systems*

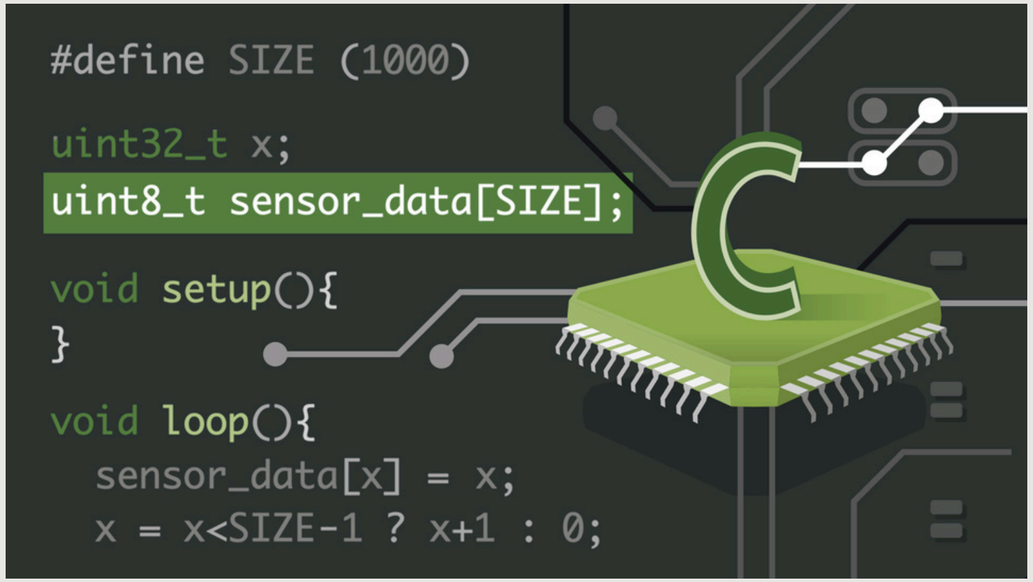
- Targets embedded hardware directly, such as microcontrollers or microprocessors with limited resources

- Does not have standard libraries like stdio.h. Instead, it uses hardware-specific libraries to interact directly with peripherals, such as GPIOs, timers, and UART

- Programs run directly without an operating system or with minimal OS support

- Involves direct hardware manipulation (manipulation of registers directly)

- Generally embedded systems does not support dynamic memory allocation due to fragmentation



**Microcontrollers are everywhere**

# *Examples*

| | |
|---|---|
| Washing Machines | Water level, timing, spin speed |
| Microwave Oven | Keypad input, cooking time, display |
| Remote Control | Button detection, IR signal transmission |
| Smartwatch/Fitness Band | Step counting, heart rate sensing, display |
| Car | Engine control, airbags, ABS, window motors |
| Bluetooth Speakers | Audio control, Bluetooth communication |
| Printers | Paper feed, ink levels, user interface |

How do we actually use the data collected by these devices?

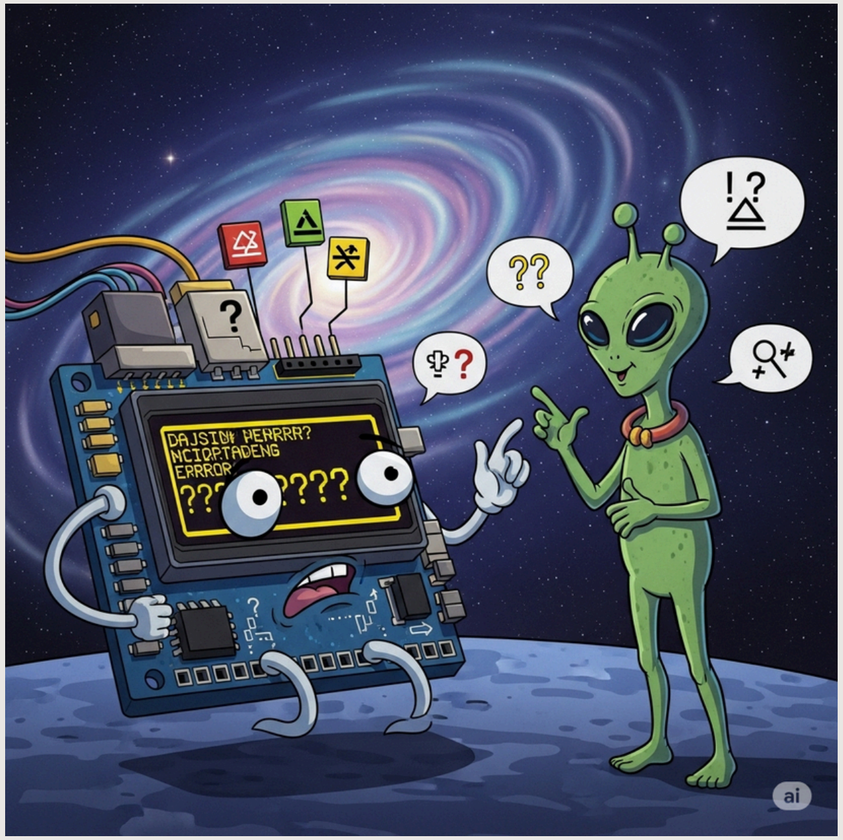
# *Communicating with Other Systems*

Microcontrollers need to talk to other devices — sensors, displays, other microcontrollers, or computers.
They also need to share information with higher-level systems — like:

- Smartphones (apps)
- PCs (data visualization/logging)
- Cloud servers (IoT)

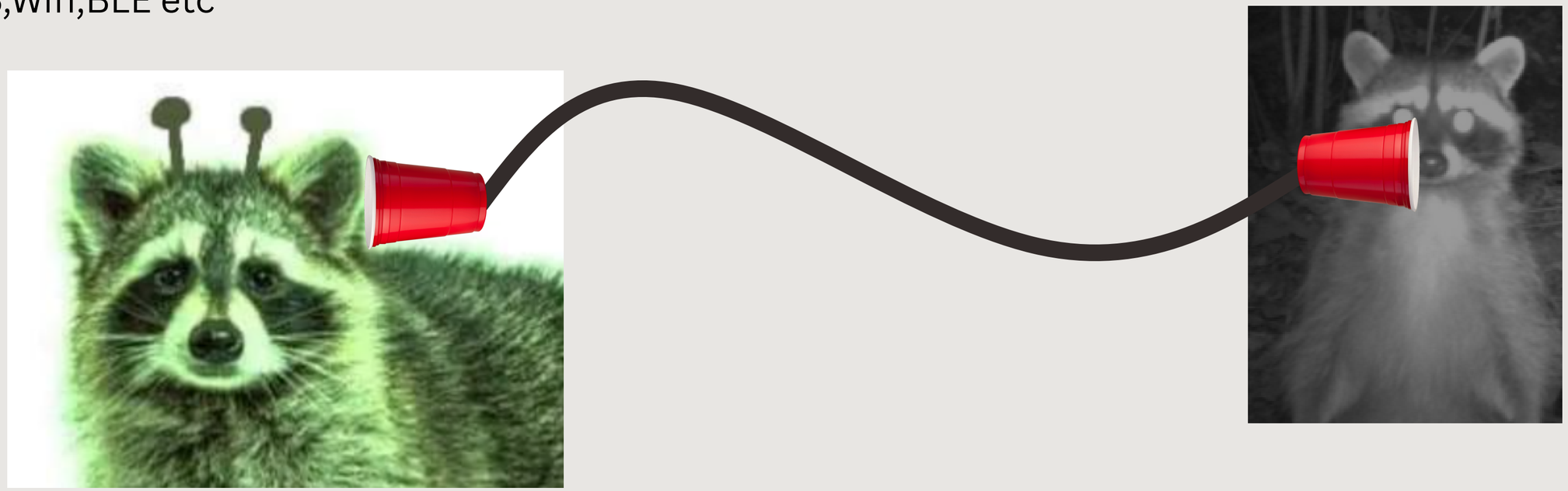
**Protocols** are used to enable this communication.
Why Communication Protocols Are Needed:

- Devices come from different manufacturers, with different capabilities
- They must understand each other's language to exchange data reliably
- Protocols define the rules, format, speed, and structure of this exchange



# What exactly is a communication protocol ?

- Any common language/encoding accepted by two or more systems is technically a communication protocol
- the medium that facilitates this communication is the communication channel and can be as simple as a wire
- most common daily occurences of a device to device communication is USB,Wifi,BLE etc

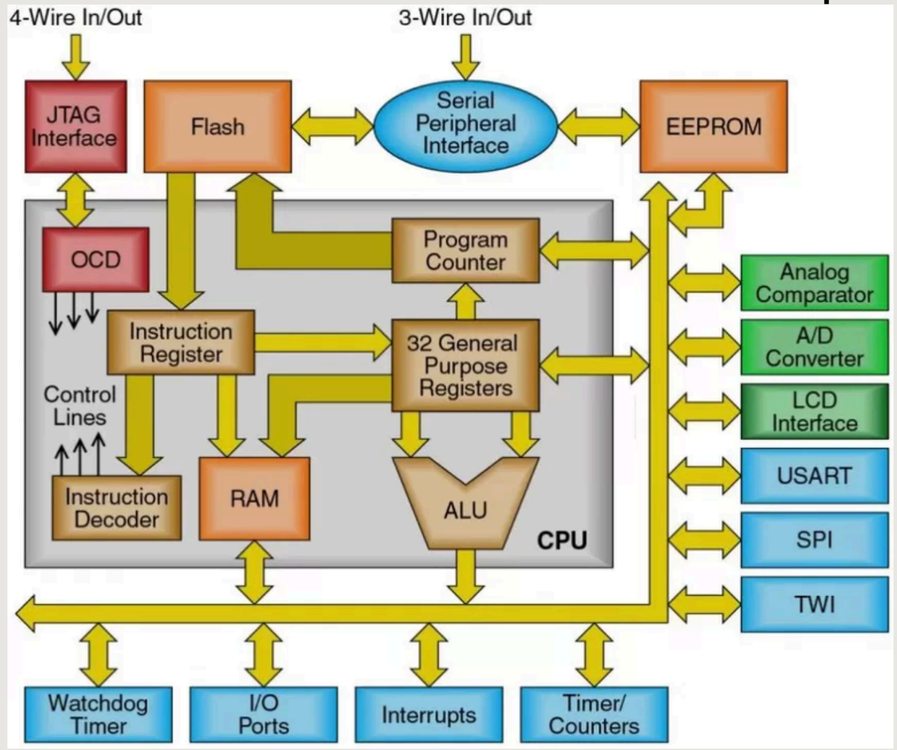


# How do we choose an appropriate communication protocol?

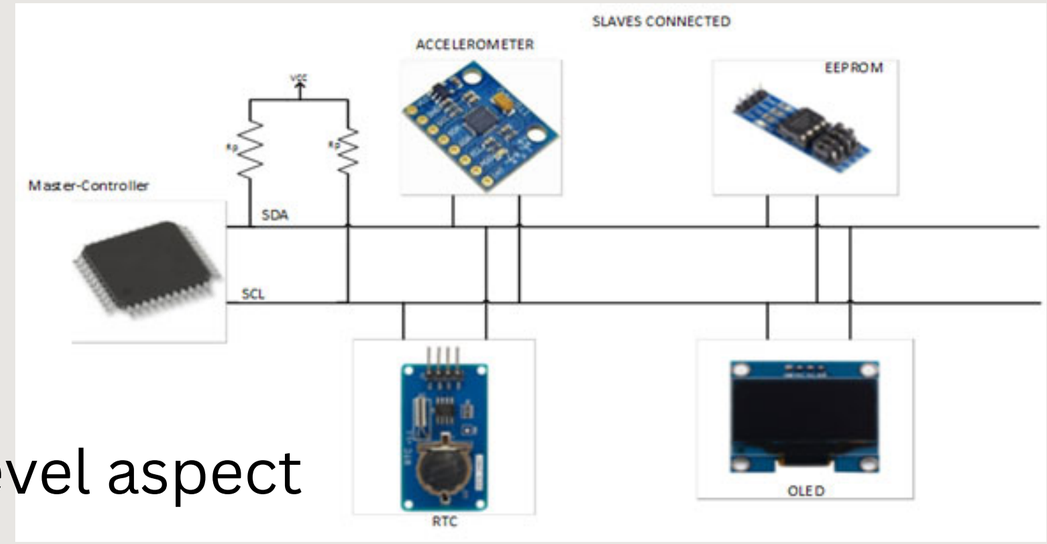
# How a system is designed?

- Whenever a high level system is designed it usually has an internal architecture and external ports
- But what this system does boils down to its core functionality and the design choices taken by the developer
- Different design require different choices based on
- usually a communication network has a receiver and a transmitter but there can be cases where multiple such modules can exist or they might be a dual featured system that can communicate in both directions (bi-directional)
- Based on what purpose a channel serves and its location withing a system block it can be classified as **inter** and **intra** communication protocol

low level aspect

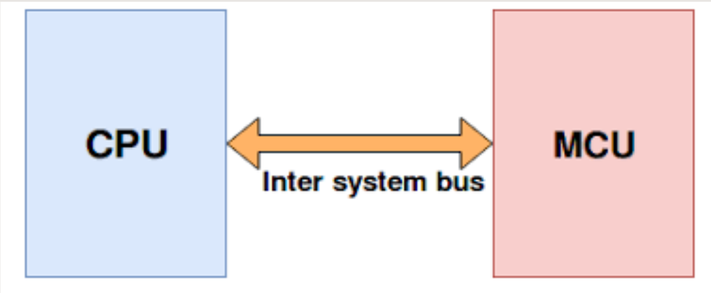



high level aspect



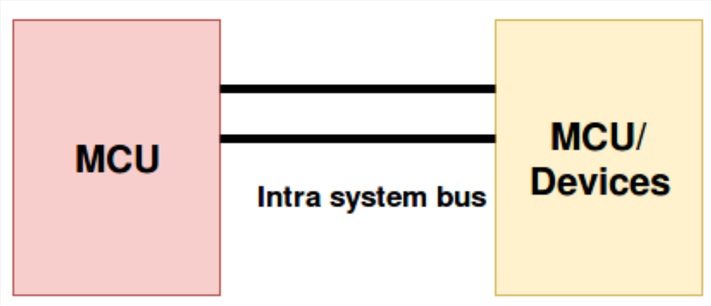
# Inter System Protocol:

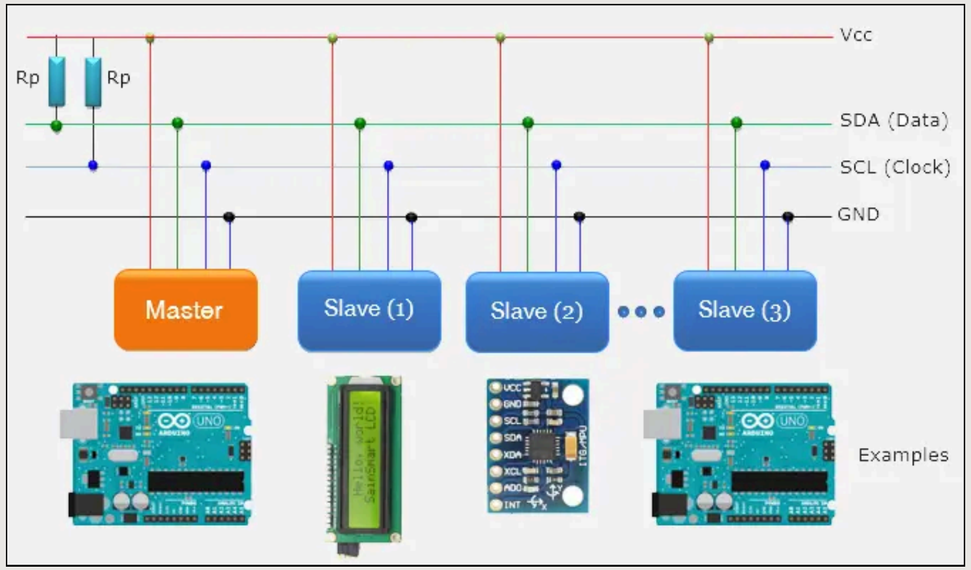
- The inter system protocol using to communicate the two different devices. Like communication between computer to microcontroller kit. The communication is done through a inter bus system
- Between two completely unknown devices like a keyboard to a laptop, a cpu to a microcontroller




# Intra System Protocol:

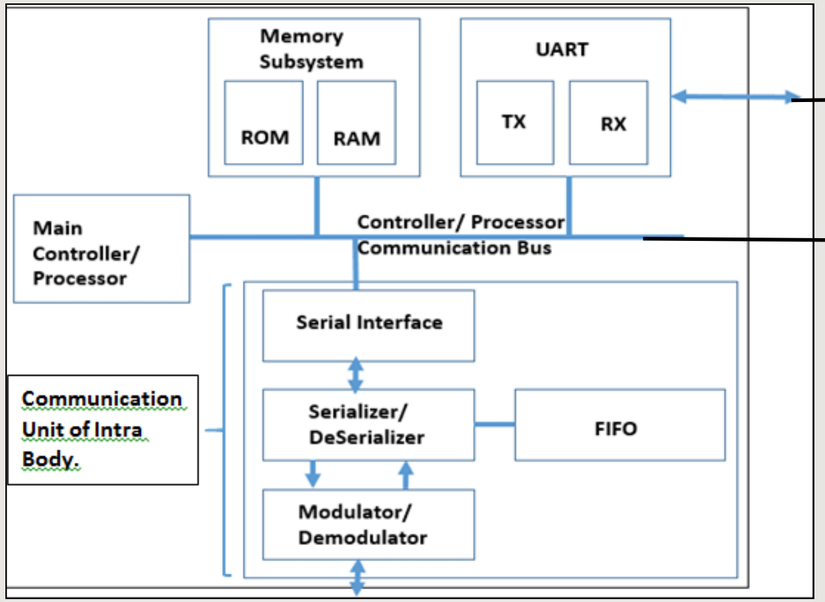
- "Intra-system communication protocols" refer to the set of rules and guidelines that enable different components within the same system or circuit board to communicate with each other.
- within the same device or device environment, much higher speeds for shorter lengths






# Typical intra device floormap

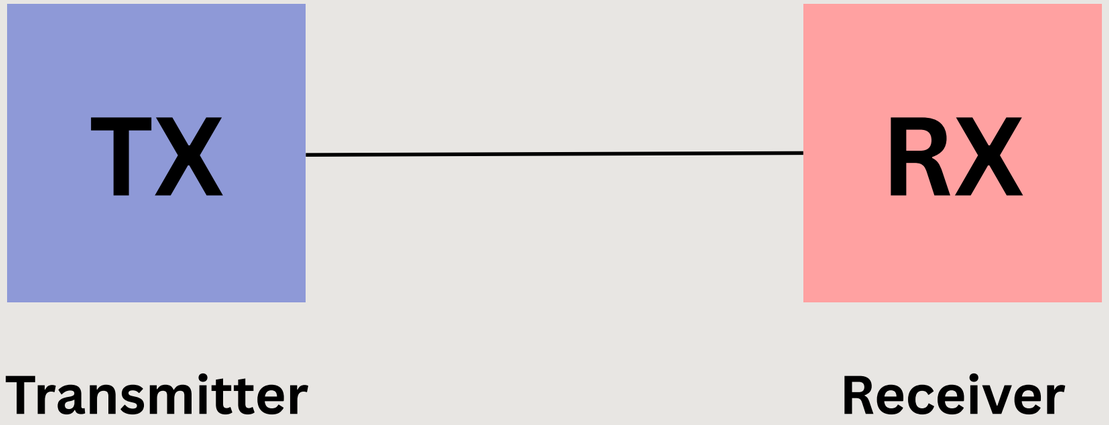
# A CPU architecture



# Lets delve into other aspects that define a design choice using examples

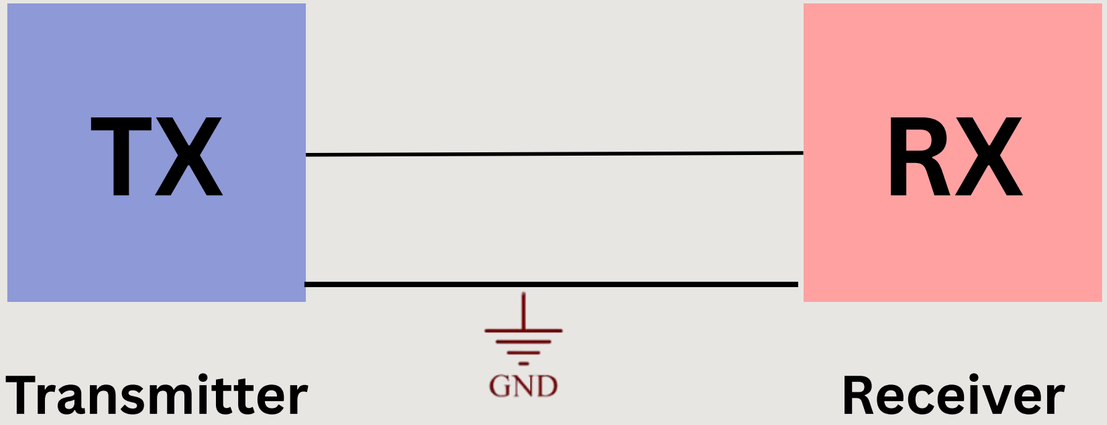
# Basic communication

- lets assume two devices that need to communicate with each other
- the line is uni-directional




- with the current configuration the device doesnt understand what voltage is what level as there is no complete circuit, a voltage is a potential difference and with no common ground there is no reference to measure this difference all voltages are mapped randomly
- Hence the system will misbehave
- lets say we connect a wire, do the systems really know? what is happening
- can it distinguish a high or a low?
- They are two different devices with no common standpoint, ie a single wire doesnt complete a circuit, a ground is needed

- Okay lets say we connect a ground




## Analog Signals

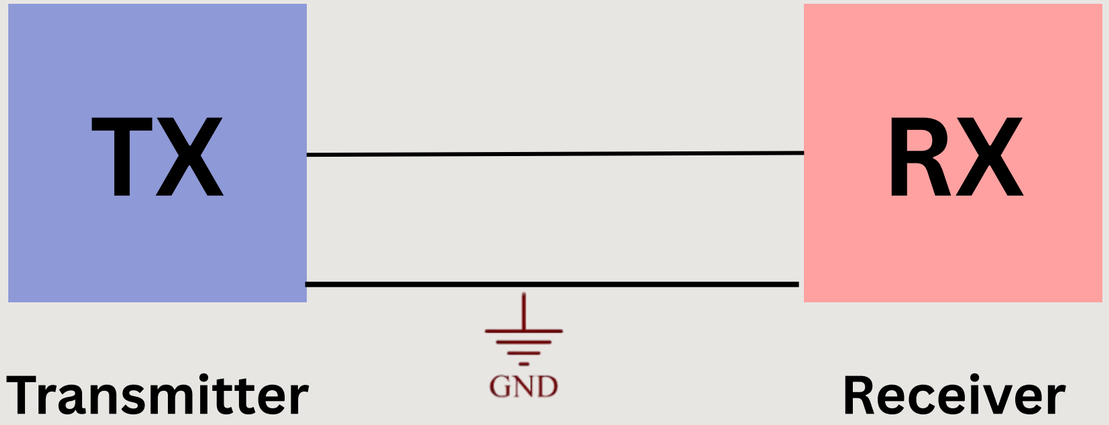
- now there are two ways, either we transmit an analog signal which can go between different voltage levels, we transmit an analog signal that has different voltage levels and each levels refers to a different data
- these are much harder to implement as we need a sampler to sample each voltage and a transmitter that can represent each data accuractely as a voltage ie ADC and DAC resp

## Digital Signals

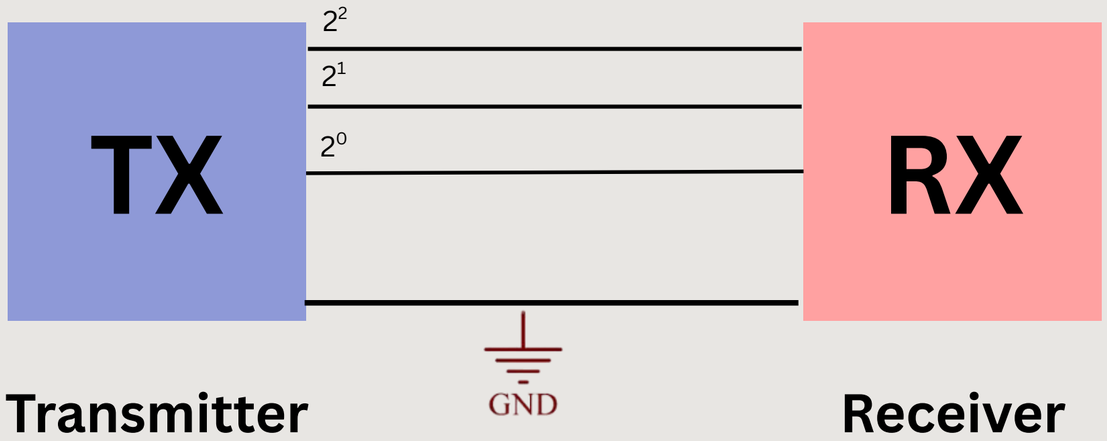
- this means that each wire represents one  bit either 0 or 1, HIGH or LOW
- much easier to implement

- Okay lets say we choose the most basic encoding scheme, Binary

- **now one wire represents one bit, is this really satifactory?**
- we need more combinations

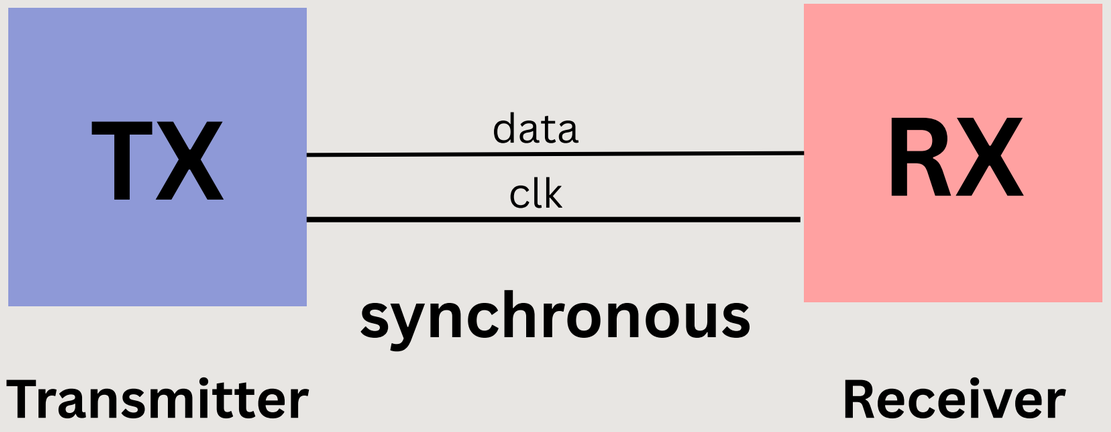



- lets add more wires




- this gives us total $2^n$ combinations where n is the no. of channels
- this limits us hardware wise, although its a simple encoding it requires more channels and becomes more expensive to implement practically
- this does mean it isnt used, it used on really low levels aspects such as registers, memoryetv

- Now there are different coding mehanism such as gray code, but they too face similar challenges
- **hence we try to use a much more complex scheme using bit toggling, ie by referencing a time frame and knowing which bit to expect during which time frame we can effectively communicate over a standardised protocol**

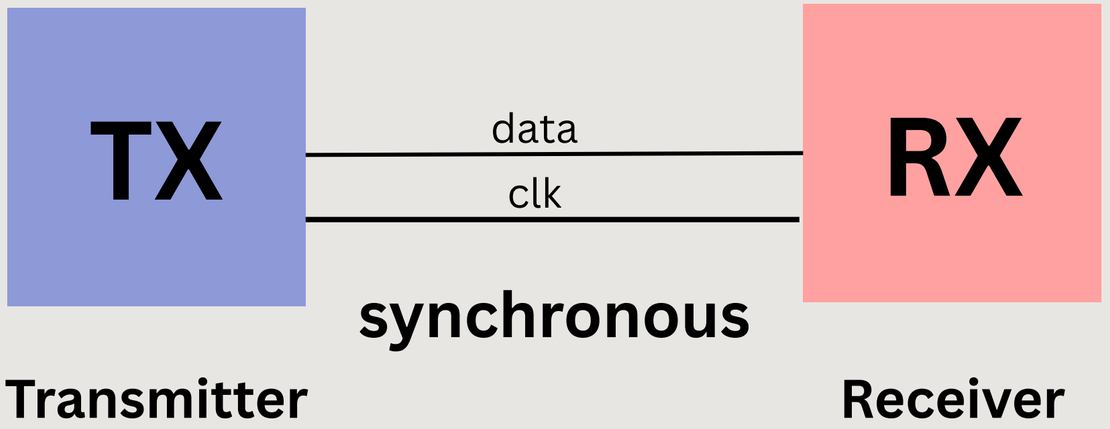


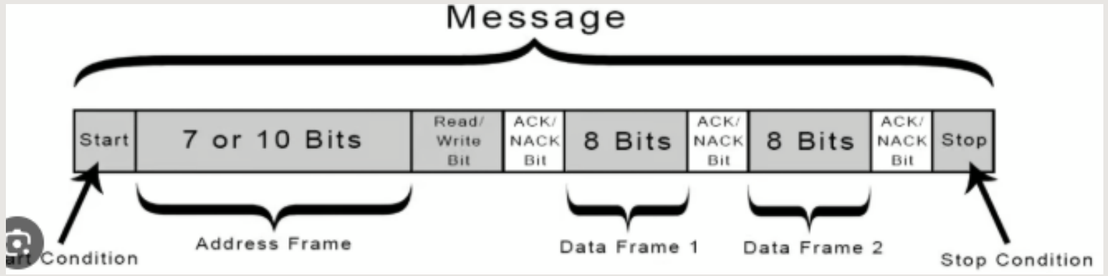

- time frame here refers to a clock, whereas we can have protocols that can implement asynchronous communication that are independent of clocks
- we shall look at how they work than different existing protocols and why they have been built that way


- **there are different ways different protocols implement their encodings but this is the most general concept behind each of these**

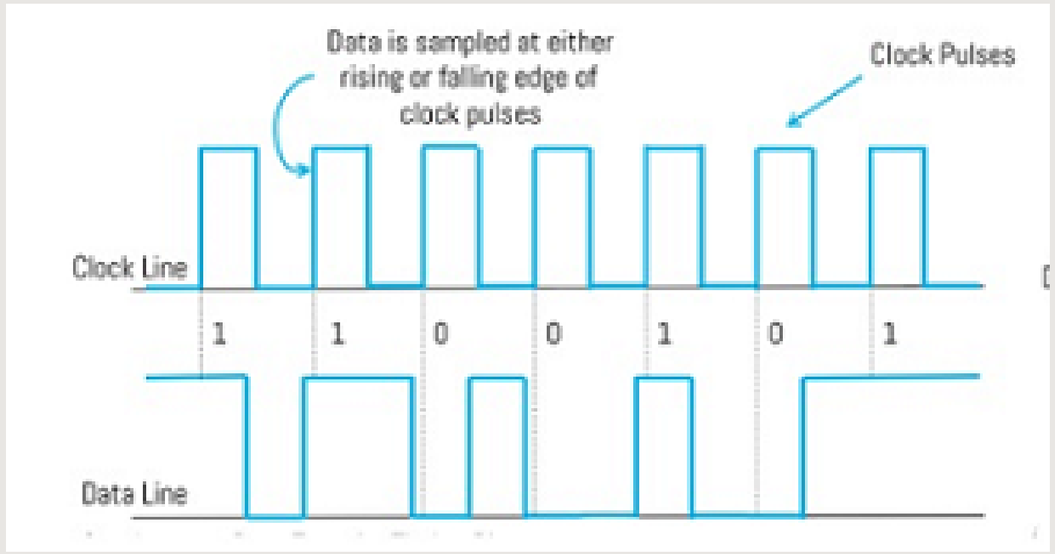
# Synchronous

- **a timing clock is referenced and the receiver understands data based on the timing, as each protocol has a well defined standard to send or receive a bit stream**



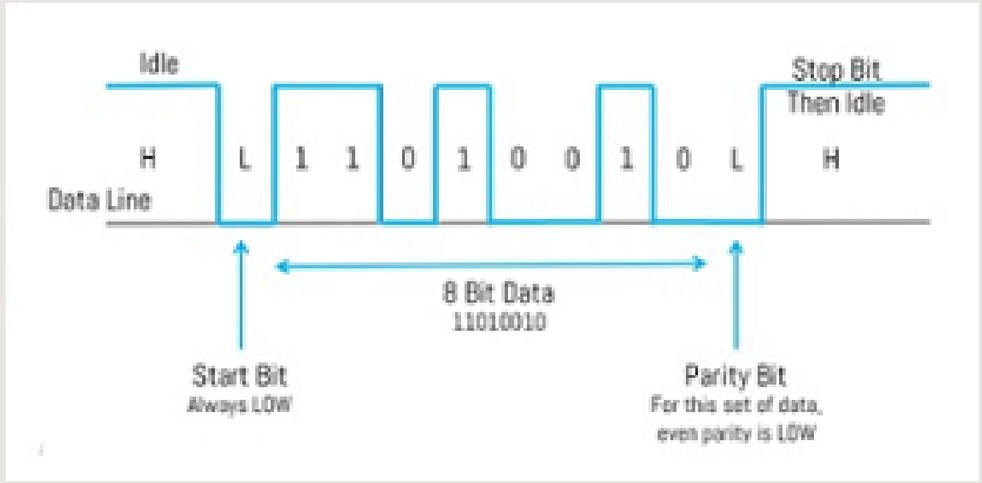

- the receiver looks for the start bit and processes data based on the standard, after the set size, a stop bit arrives then the reciver looks forward to the next start data bit



# Asynchronous

- **sometimes due to length constraints designers can use an asynchronous ciruit**
- **how is it that data can be transfered without a ground or a clock to define a reference**

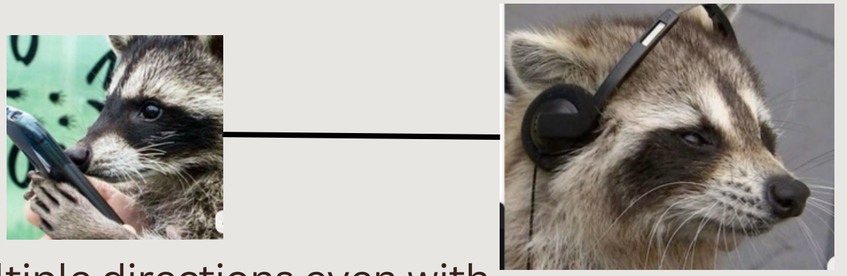

- in this case the receiver goes through something known as a pairing phase where it maps each and every signal voltage level, it tries to lock onto the phase of this signal wire
- this is fairly complex and can be much slower sometimes
- they utilise PLLs(phase locked loops) to undergo this and they can be expensive on the device end but cheaper over long distance communication as wire length is halved
- most common example is a wireless connection




- each protocol is a design choice of its own and they look for the start and stop bits based on these choices

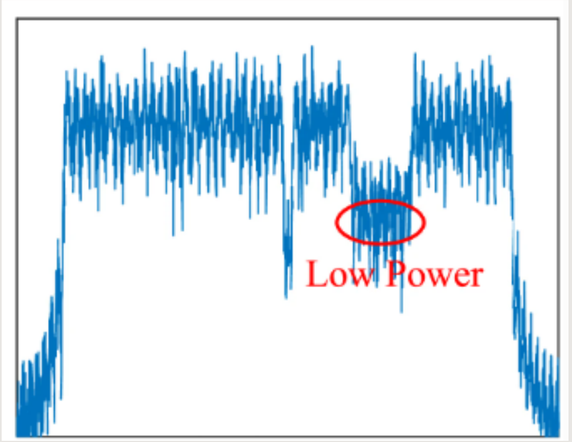
# Wired

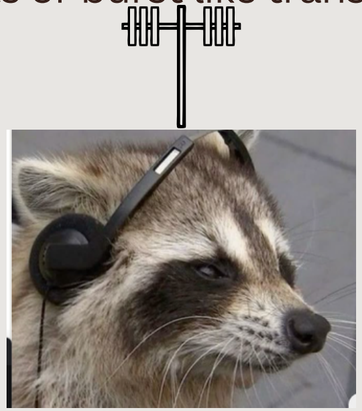
- wired circuits are simple to understand and like whatever we have seen here before, there is a physical conducting media that is high speed and more reliable and loseless

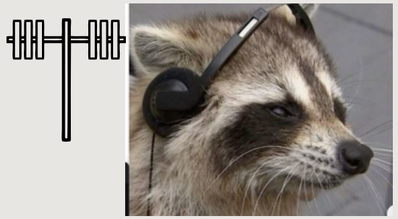


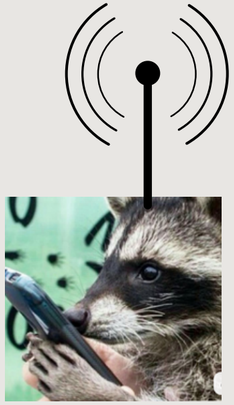
# Wireless

- wireless data transfer is much harder to implement as the signal can go in multiple directions even with complex waveguides, there is only so much we can expect from the system
- transmission occurs over air as its non conducting, cant happen in water as water shorts our signals
- more loseless as receiver can be anywhere within the device environment, and noise can be easily introduced
- hence to save power we use packets or burst like transmission



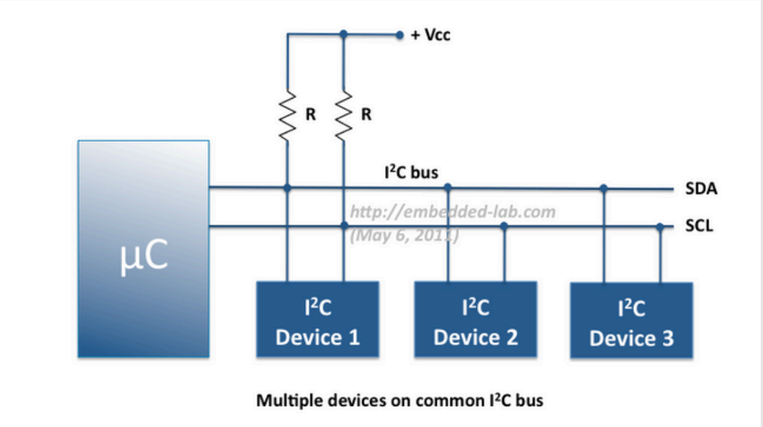




# Intra System Protocol:

## I2C Protocol

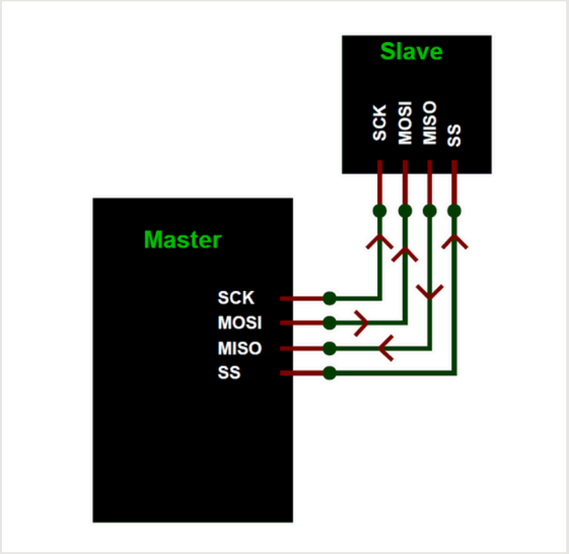
- Two-Wire Simplicity: I2C dramatically simplifies wiring by using only two shared lines (SDA for data, SCL for clock) for communication with multiple peripheral devices, significantly reducing pin count on the microcontroller and simplifying PCB layout.
- Address-Based Slave Selection: Each slave device on the I2C bus has a unique address. The master precisely targets a specific slave by sending its address, ensuring that only the intended device participates in the current data exchange while others remain inactive.
- Master-Controlled Communication: As a master-to-slave protocol, the microcontroller (master) dictates the flow of communication, initiating data transfers and controlling whether it's reading from or writing to a selected slave, making it suitable for managing various sensors, memory, and display units.



Multiple devices on common I²C bus

## SPI Protocol

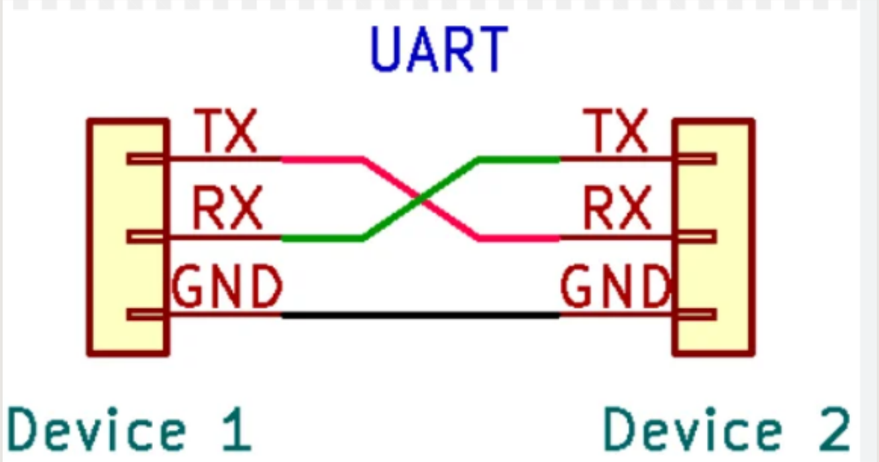
- A SPI has a master/Slave communication by using four lines. A SPI can have only one master and can have multiple slaves. A master is usually a microcontroller and the slaves can be a microcontroller, sensors, ADC, DAC, LCD etc.
- MISO (Master in Slave Out) - The Slave line for sending data to the master.
- MOSI (Master Out Slave In) - The Master line for sending data to the peripherals.
- SCK (Serial Clock) - The clock pulses which synchronize data transmission generated by the master.
- SS (Slave Select) –Master can use this pin to enable and disable specific devices.



# Inter System Protocol:

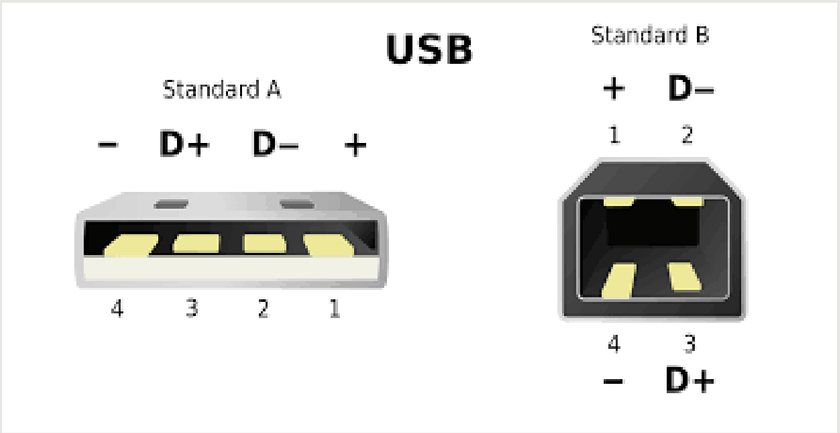
## UART Protocol

- Two-Wire Asynchronous Serial: UART is a two-wire (Tx for transmit, Rx for receive) serial communication protocol that operates asynchronously, meaning it does not use a shared clock line; instead, it relies on start and stop bits for synchronization of individual data bits.
- Bit-by-Bit Data Transfer: Data is transmitted and received serially, one bit at a time. Each data packet is framed with a start bit (to signal the beginning of data) and stop bits (to signal the end), allowing the receiver to synchronize with the incoming data stream.
- Point-to-Point Inter-System Communication: UART is commonly used as an "inter-system protocol" for direct, point-to-point communication between two different devices, such as an Arduino board and a computer, or two microcontrollers, often via direct wire connections without a shared bus system.




## USB Protocol

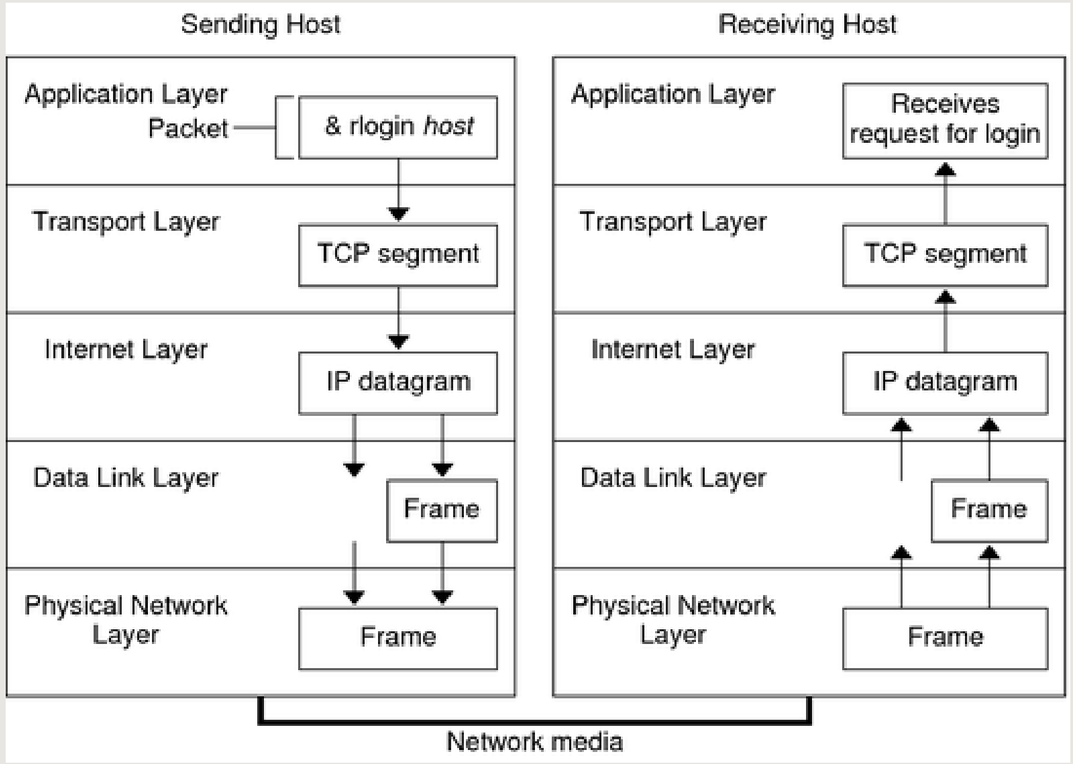
- Versatile Inter-System Connection: USB is a widely adopted "inter-system protocol" designed for communication between a host computer and various peripheral devices, serving as a universal standard for connecting system peripherals like mice, keyboards, and flash drives.
- Two-Wire Serial with Data & Power: It uses a two-wire serial communication approach for data transfer (D+ and D- lines) and also provides power to connected devices, simplifying connectivity and reducing the need for separate power adapters for many peripherals.
- Host-Controlled Plug-and-Play: USB communication is host-controlled (typically by a computer), but devices can initiate data transfer requests. It supports plug-and-play functionality, allowing devices to be connected and recognized without requiring manual configuration, often necessitating specific driver software for their functionality.



# TCP/IP Protocol

- most commonly used wireless protocol

- Foundational Internet Protocol Suite: TCP/IP is not a single protocol but a suite of communication protocols (including TCP and IP as its core components) that defines how data is exchanged over the internet and most modern computer networks.
- Reliable (TCP) and Unreliable (IP) Data Delivery: The suite combines IP (Internet Protocol) for addressing and routing data packets across networks (an unreliable, best-effort delivery service) with TCP (Transmission Control Protocol) which adds reliability, ensuring ordered, error-checked, and guaranteed delivery of data streams between applications.
- Layered Architecture for Network Communication: TCP/IP operates through a layered model (often simplified into Application, Transport, Internet, and Network Access layers), allowing different protocols to handle specific aspects of communication, from physical transmission to application-level data exchange, enabling diverse devices to communicate seamlessly across vast networks.



# Is our data really safe??

no

# Noise

- Data Corruption and Errors:
  - Bit Errors: Noise can cause a transmitted '0' to be interpreted as a '1' or vice versa, leading to bit errors in digital communication.
  - Data Loss/Alteration: In severe cases, noise can completely obscure parts of the signal, resulting in lost data packets or corrupted information, making the original message unrecoverable without retransmission.
  - Misinterpretation: Even subtle noise can alter the message enough to cause the receiver to misinterpret the sender's intent, leading to incorrect actions or misunderstandings.
- Reduced Signal Quality (Signal-to-Noise Ratio - SNR):
  - Lower SNR: Noise directly lowers the Signal-to-Noise Ratio (SNR), which is the ratio of desired signal power to noise power. A lower SNR indicates that the signal is less distinguishable from the background noise.
  - Impact on Clarity: As SNR decreases, the clarity and intelligibility of the received signal diminish. For audio, this means static or garbled speech; for video, it means grainy or distorted images; and for data, it means higher error rates.
  - System Performance Limit: Noise is a fundamental limiting factor on the maximum data rate that can be reliably transmitted over a channel (Shannon's Channel Capacity Theorem).
- Increased Resource Consumption:
  - Retransmissions: To combat errors introduced by noise, communication protocols often implement error detection and correction mechanisms. When errors are detected, the receiver requests retransmission of corrupted data, leading to increased bandwidth usage and latency.
  - Processing Overhead: Error correction codes (ECC) add redundant information to the data, which requires additional processing power at both the transmitter (encoding) and receiver (decoding and error correction).
  - Power Consumption: Transmitting at higher power to overcome noise can increase energy consumption, especially in wireless systems.
  - Reduced Communication Range and Reliability:
  - Decreased Range: A noisy environment effectively reduces the effective range of a communication system. The signal must be strong enough to rise above the noise floor at the receiver for reliable detection.
- Intermittent Connectivity: In wireless channels, fluctuations in noise levels (e.g., due to interference) can cause intermittent connection drops or poor performance.
- Unpredictable Behavior: Random and unpredictable noise can make system behavior erratic and harder to diagnose.

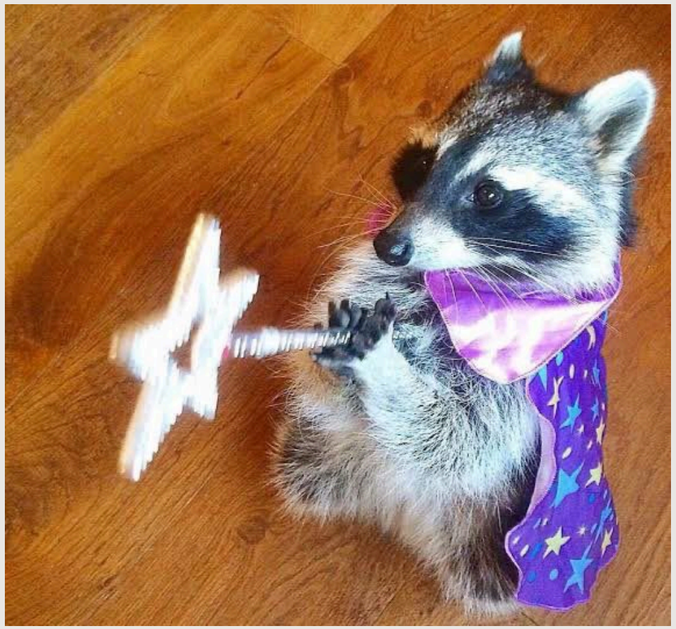
# Noise

- **Types of Noise (Examples):**
  - **Thermal Noise (Johnson-Nyquist noise):** Inherent in all electronic components due to random motion of electrons, present even in ideal conditions.
  - **Shot Noise**: Caused by the discrete nature of charge carriers (e.g., electrons, photons) flowing through a device, often significant in semiconductors.
  - **Interference (Electromagnetic Interference - EMI/RFI):** Unwanted signals from other electronic devices, power lines, radio transmitters, etc., that couple into the communication channel.
  - **Crosstalk**: Signal leakage from adjacent communication channels or wires within the same cable.
  - **Impulse Noise:** Short-duration, high-energy bursts of noise, often caused by lightning, motor ignitions, or switching transients.
  - **Quantization Noise**: Introduced during the analog-to-digital conversion process when a continuous analog signal is represented by discrete digital levels.

# Summary

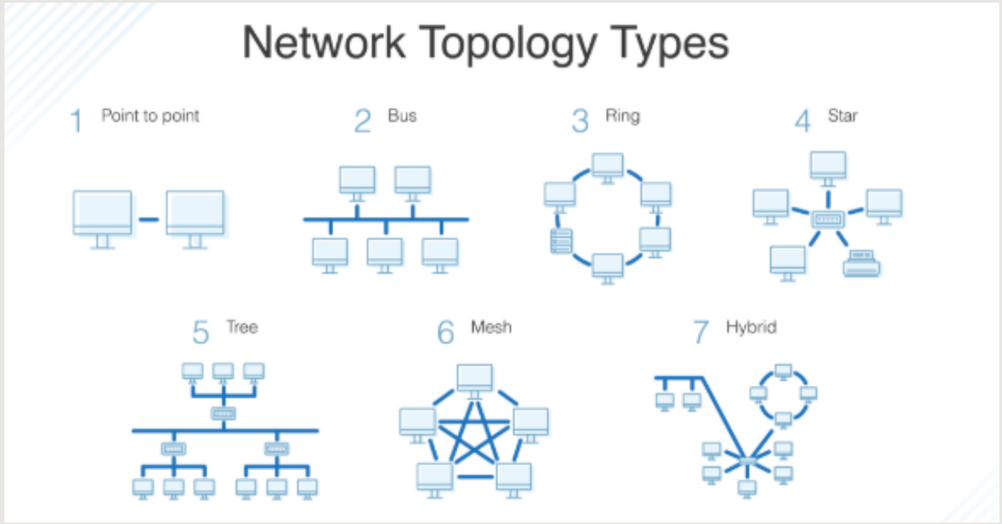
**To really decide a communication protocol these are the main key factors**

- **Communication type**: Is it real-time, where speed is critical, or asynchronous, where delivery time is less important?
- **Data volume and frequency**: How much data needs to be transmitted, and how often?
- **Distance**: How far apart are the communicating devices?
- **Security requirements**: Is encryption or authentication needed?
- **System configuration**: What type of devices are involved (microcontrollers, servers, etc.) and how are they connected (wired, wireless)?
- **Performance requirements:** What are the acceptable latency, bandwidth, and throughput?
- **Scalability:** Will the system need to handle more devices or data in the future?
- **Cost and complexity**: What is the budget and level of expertise available?



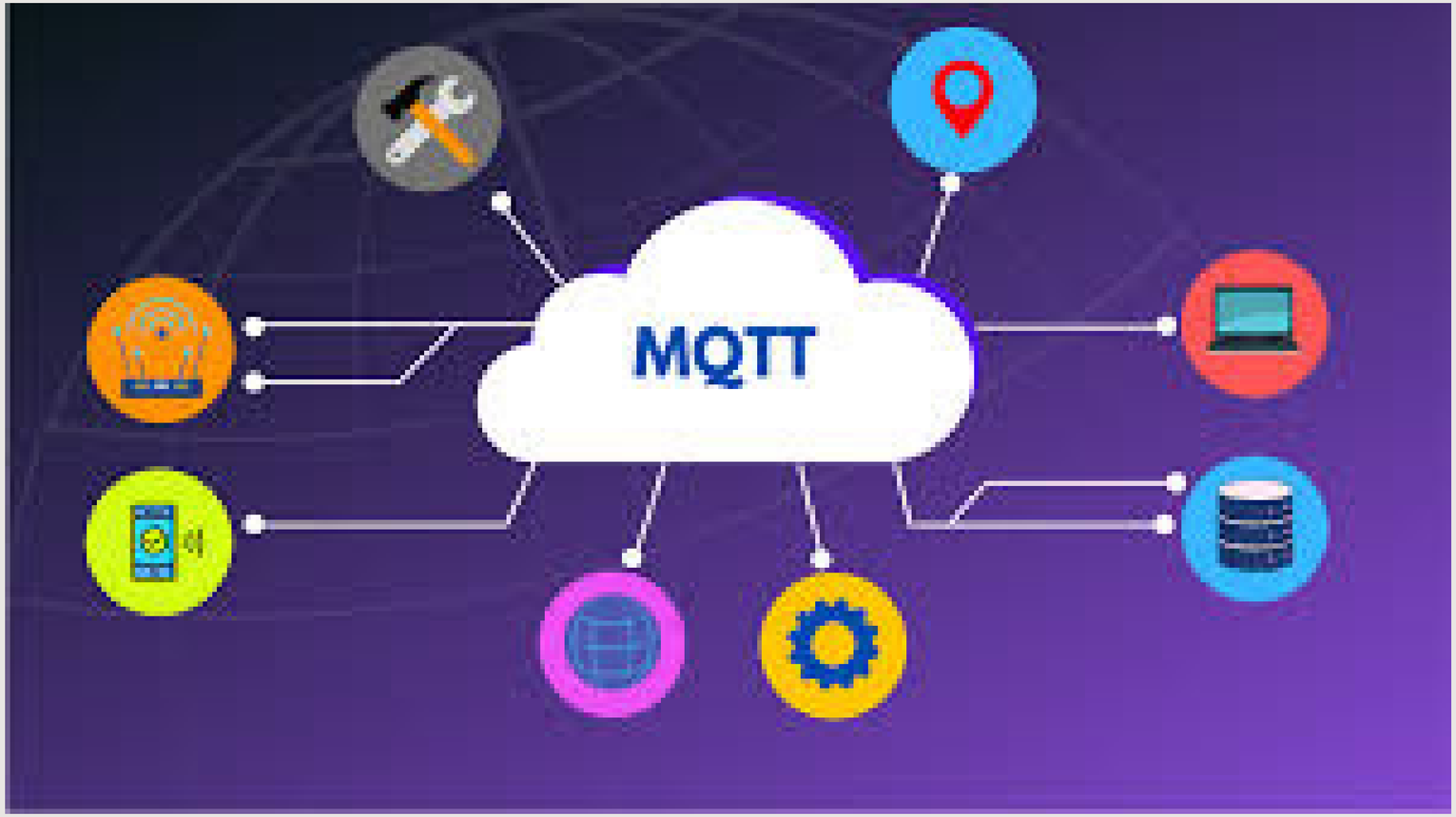
# Multi Device communication

- Now when multiple devices connect, how do we really know if data from system A goes to system B without being interpretted by system C
- there are different topologies for this particular scenario




- even though we do have the devices connected and many protocols at hand, we need a much more composed architecture to better connect and manage these devices one such is MQTT

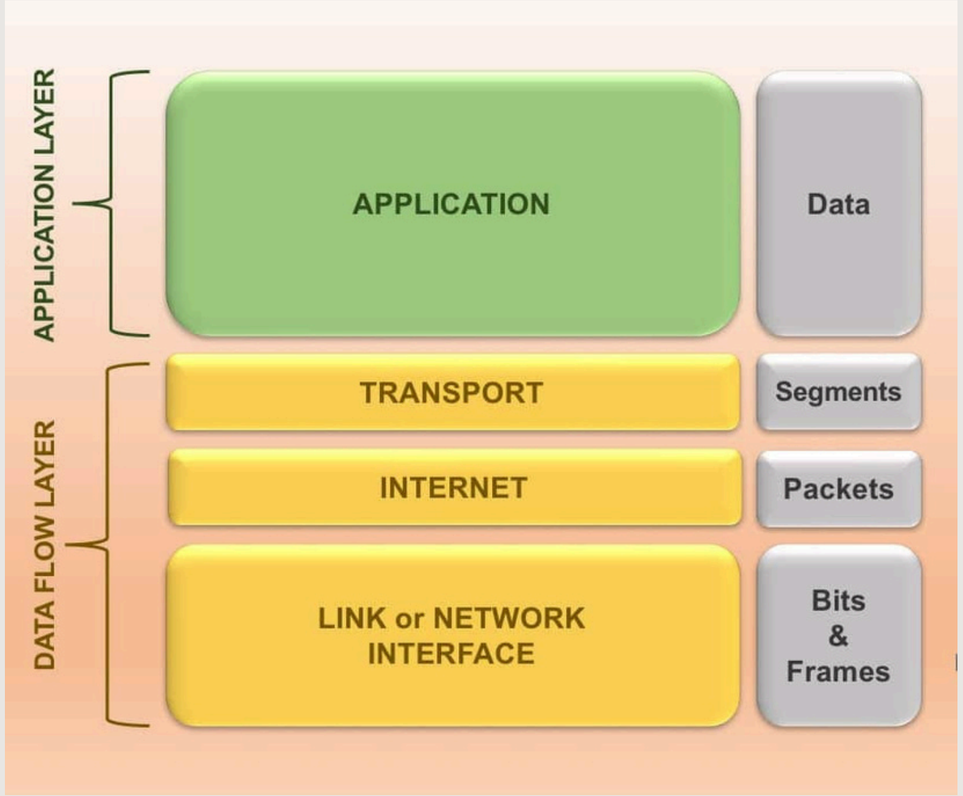
# *Introduction to MQTT!*



# What is MQTT

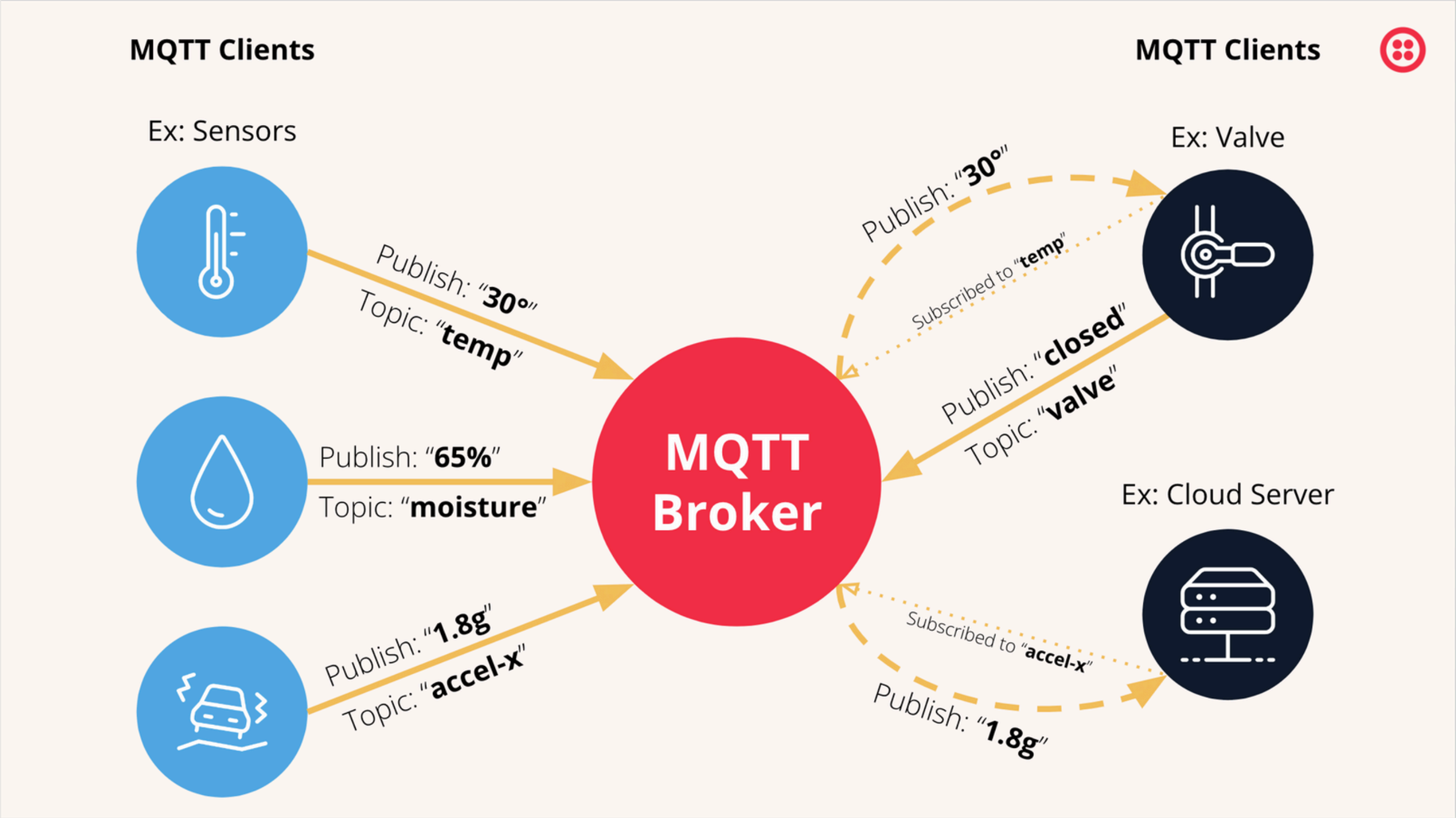
- MQTT (Message Queuing Telemetry Transport) is a lightweight messaging protocol designed for devices with limited resources and unreliable networks.
- It uses a publish-subscribe model rather than traditional client-server.
- MQTT is an application-layer protocol that operates over TCP/IP.

# Why MQTT?




- Designed for low-bandwidth, high-latency, or unreliable networks — perfect for IoT and embedded systems.
- Decouples producers and consumers of data, simplifying communication in large, distributed systems.
- Supports Quality of Service (QoS) levels to guarantee message delivery based on application needs.

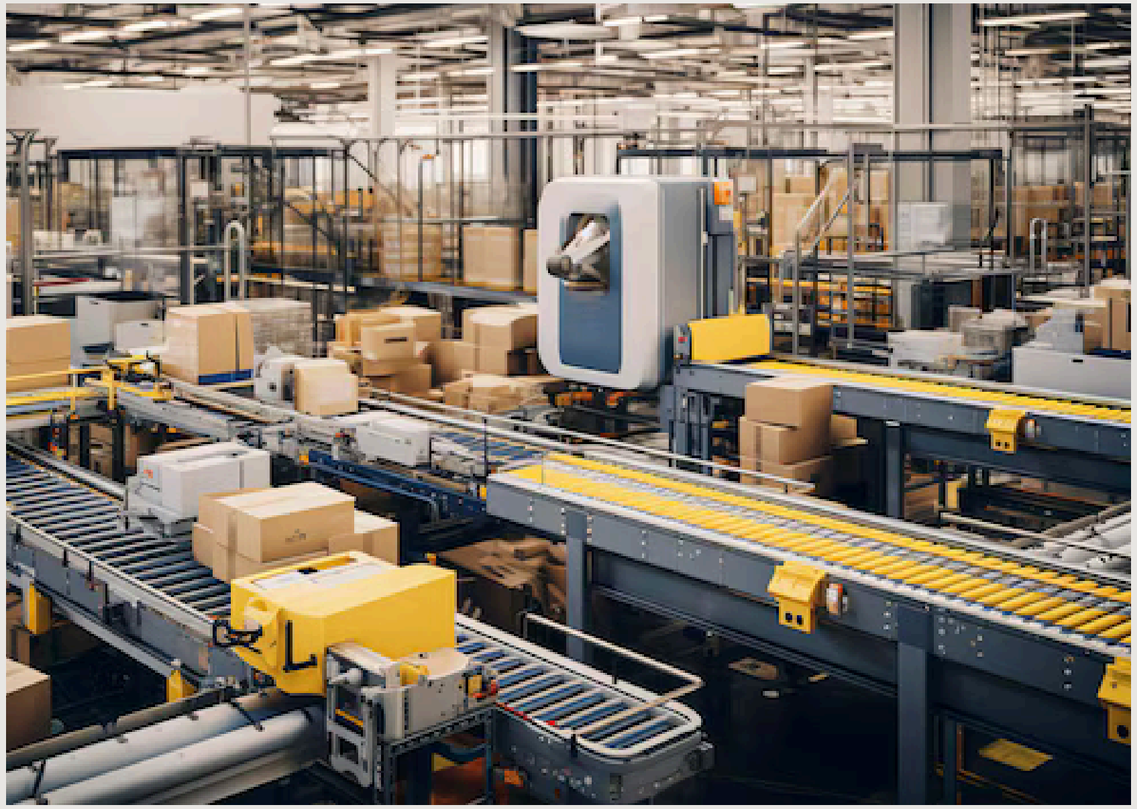
# Example MQTT network



# Clients, Messages and Topics

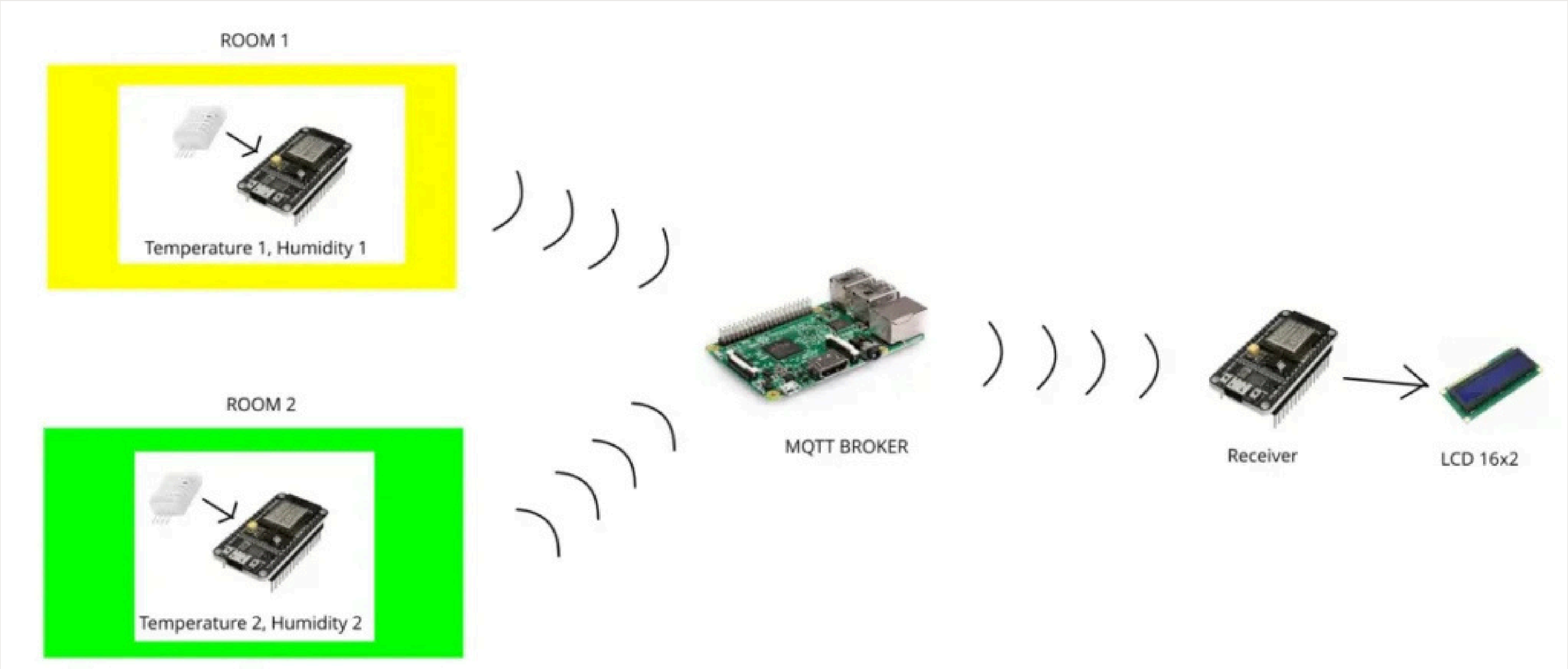
- An MQTT client can act as a publisher, subscriber, or both.
- Topics are hierarchical strings (e.g., sensor/room1/temperature) that organize messages.
- Clients subscribe to topics of interest and receive only relevant messages.
- Wildcards (e.g., sensor/+/temperature) allow flexible subscriptions.

# Publisher, Subscriber, and Broker

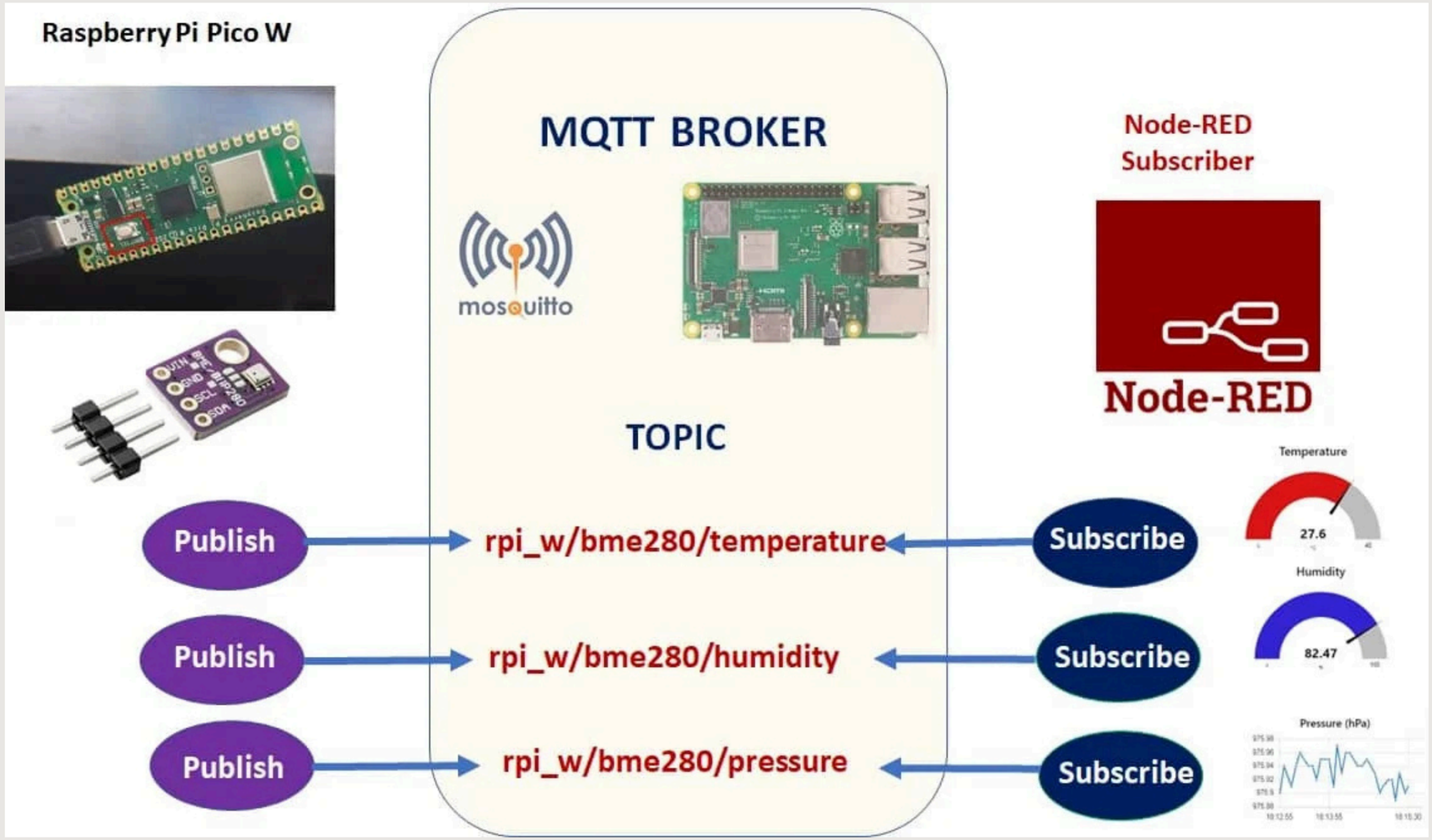


- Broker: Core of the system — receives messages, filters them, and forwards to subscribers. Handles QoS, session persistence, and security.
- Subscriber: Registers interest in a topic; receives matching messages from the broker.
- Publisher: Sends messages to a topic on the broker.

# Example Network - 1



# Example Network - 2



# Question Time!

1. Can you think of a real-world scenario where MQTT’s small message size and low bandwidth use are critical?

2. Where might MQTT not be the best choice?

3. When would you choose QoS 0 instead of QoS 2?

# Summary

- MQTT is a lightweight, efficient protocol ideal for IoT, machine-to-machine, and remote sensor communication.
- It uses a publish-subscribe model that decouples senders and receivers, simplifying large-scale communication.
- Runs on TCP/IP, providing reliable delivery even over unstable networks.
- Supports different Quality of Service (QoS) levels to balance reliability and network resource use.
- Key components: Publisher, Subscriber, Broker — working together to ensure data gets where it’s needed.
- Easily scales from small local networks to global cloud-based systems.

*Thank You!*



Sahasra, Ashmita, Ashmit, Ranjit