جامعة الملك عبدالله للعلوم والتقنية
King Abdullah University of Science and Technology

أكاديمية كاوست
KAUST ACADEMY

Internet of Things (IoT)

DAY 2 From analog signals to a board your phone can control

August 2 to 6, 2026

KGSP Summer Enrichment Program 2026
KAUST Academy, KAUST

Salah Abdeljabar
salah.abdeljabar@kaust.edu.sa

🗺️ On Today's Agenda

1. **Analog vs digital sensors**: the ADC, and sampling a continuous world
2. 🔧 **Project 2**: read a potentiometer as a number from 0 to 4095
3. **Controlling actuators**: the DAC, and faking analog with PWM
4. 🔧 **Project 3**: fade an LED smoothly instead of switching it
5. **Anatomy of an IoT application**: the thing, the internet, the decision
6. **How things talk**: publish/subscribe, HTTP, and the protocol landscape
7. 🔧 **Project 5**: control your board from a web page on your phone

By the end your board will be on the network, taking orders from your phone.

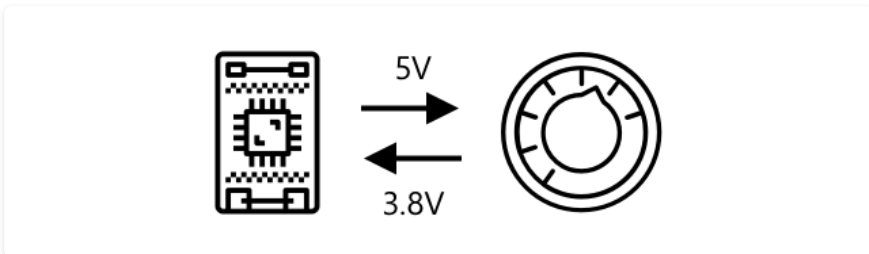
Analog vs Digital Sensors 

...and how a digital device reads an analog world

Analog Sensors

An **analog** sensor returns a **variable voltage** that maps to what it measures.

- A **potentiometer**: 5 V in → 0–5 V out as the dial turns
- A thermistor's voltage → converted to °C in code
- The value can be **anything** in a range



The ADC: Analog → Digital

IoT devices are **digital**: they only understand 0s and 1s. An **analog-to-digital converter (ADC)** turns a voltage into a number.

- A light sensor at 3.3 V outputs **1 V** → ADC value **300**
- Resolution sets the range (e.g. **10-bit** = 0–1023)
- Built into the device, or on a connector **HAT**

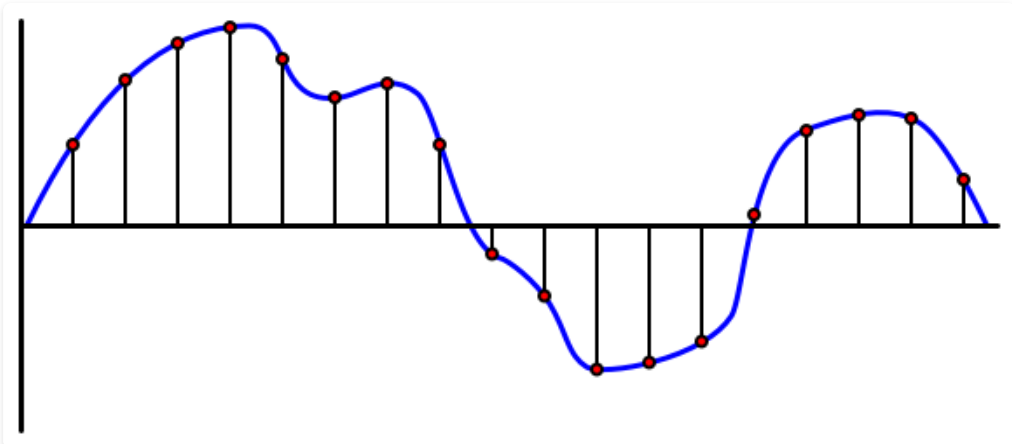
Libraries hide the math

Arduino: `analogRead(pin)` → **300** · Python: the `light` property → **300**

Sampling: Snapshots of a Continuous World

The ADC can’t capture every instant, so it **samples**: reading the voltage at a fixed rate and keeping the dots, not the curve.

- **Sample rate** = readings per second
- Sample **too slowly** and you miss what happened in between
- Sample **faster** and you get a truer signal, at the cost of power, memory, and bandwidth

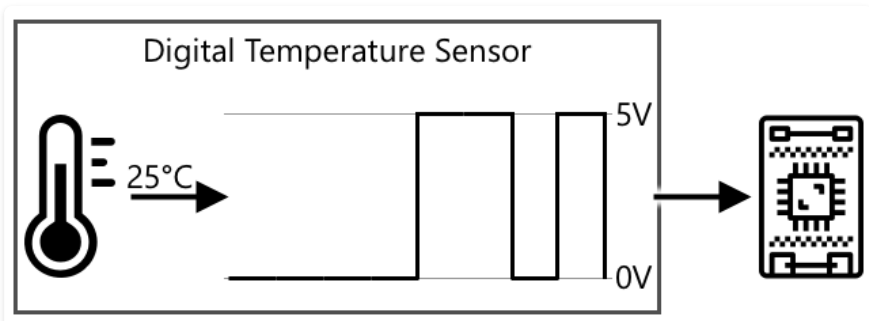


A few readings a minute is plenty for soil moisture. Audio needs thousands per second.

Digital Sensors

A **digital** sensor sends its signal as 0s and 1s directly: no external ADC needed.

- Simplest: a **button**, 3.3 V = **1**, 0 V = **0**
- Complex ones have a **built-in ADC** and send serialized data
- Enables richer, even **encrypted** data (e.g. a camera)



PROJECT 2 · ANALOG INPUT

Analog Inputs: Reading a Potentiometer

Measure a *range* of voltage (0–3.3 V) as a number 0–4095 with the ESP32's ADC.

Builds on: Project 1's `digitalRead()`, now reading a range of voltage instead of just two states.

⏱️ ~20 min 📊 Beginner 🔖 Prerequisite: Project 1

⬇ Download the sketch

Save `Project_2_ESP32_Analog_Inputs.ino`, then open it in the Arduino IDE (setup: **Foundation 3**).

Project 2 · Analog Inputs

Read a potentiometer as a number from **0 to 4095** with the ESP32's ADC.

- Tell digital signals from analog ones
- Read a variable resistor with `analogRead()`
- Watch live readings on the Serial Monitor
- Convert a raw ADC count into a voltage

Key pin: potentiometer `4` (ADC)

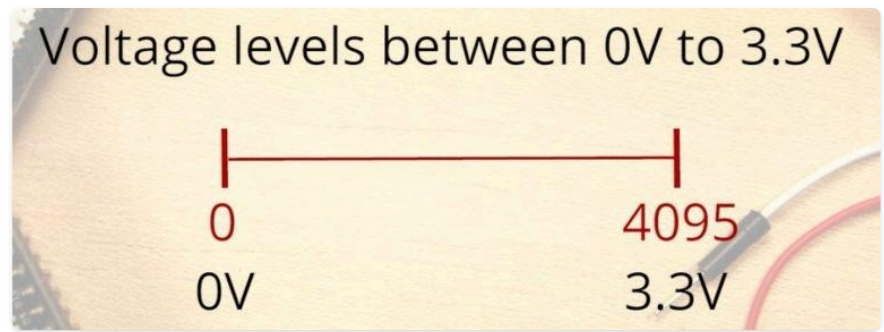
Open the guide ↗

Analog vs Digital: the ADC

Project 1 saw only HIGH or LOW. An **analog** input measures a **continuous** voltage between 0 and 3.3 V and turns it into a number.

- The ESP32 ADC is **12-bit**, so readings run **0 to 4095**
- 0 V → **0**, 3.3 V → **4095**, everything in between is proportional
- Read it with `analogRead()`

> ⚠ **Warning**
>
> We use **GPIO 4** (ADC2). ADC2 pins cannot be read while **Wi-Fi** is on, so in later Wi-Fi projects prefer an **ADC1** pin (32 to 39).

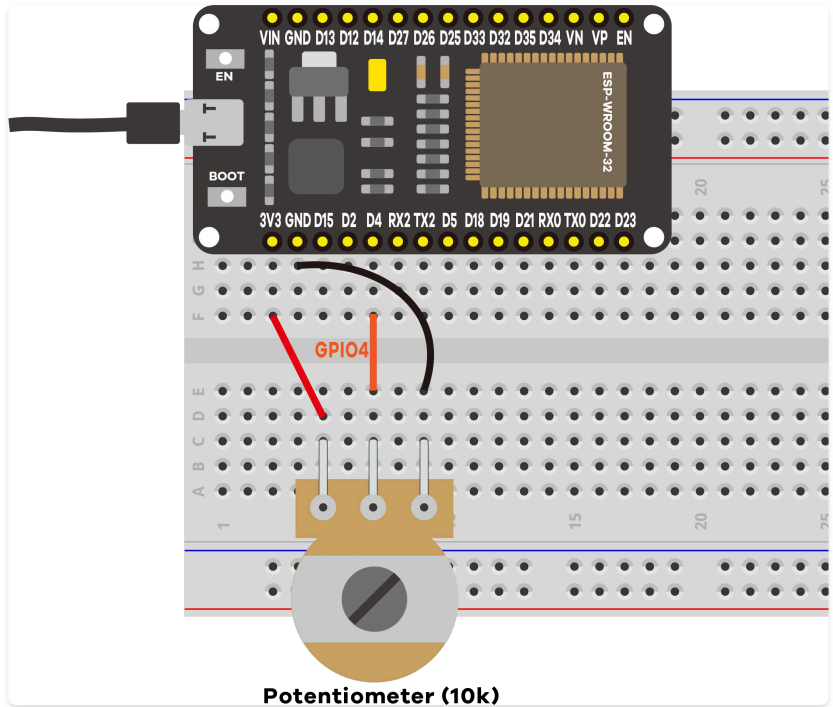


Wiring the Potentiometer

A potentiometer is a **variable resistor** with three legs:

1. One outer leg → **3.3 V**
2. The other outer leg → **GND**
3. The middle leg (**wiper**) → **GPIO 4**

Turning the knob moves the wiper, tapping off anything from 0 V to 3.3 V, and the ADC reports the matching 0 to 4095 value. Any of the board's 15 ADC pins would work.



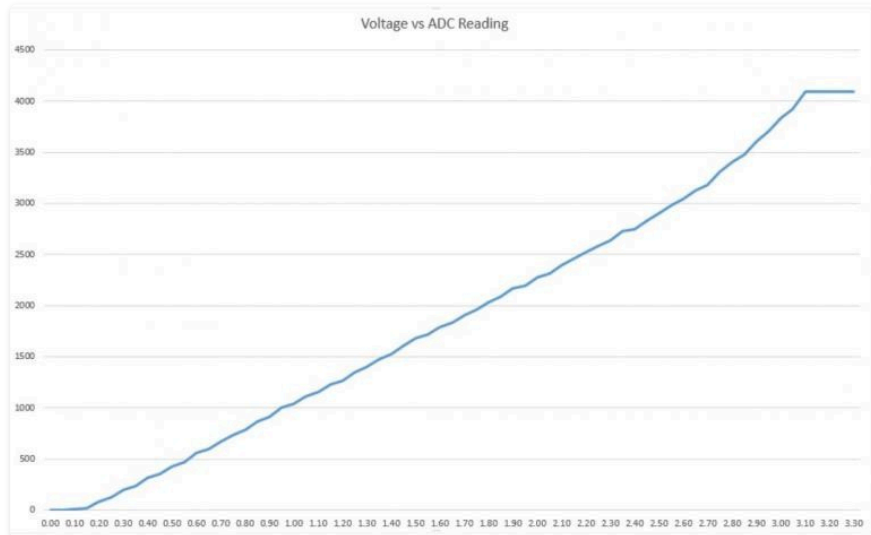
Code: Read → Convert → Print

```
void loop() {
  potValue = analogRead(potPin);            // 1. READ: returns 0 .. 4095
  float voltage = potValue * 3.3 / 4095.0;  // 2. CONVERT to volts
  Serial.printf("Pot: %4d / 4095  (~%.2f V)\n", potValue, voltage);
  delay(500);                               // 3. slow down so it is readable
}
```

- `analogRead(pin)` compares the input against the 3.3 V reference and returns a **0 to 4095** integer.
- The conversion is a plain **proportion**: `raw × 3.3 / 4095`. A reading of 2048 is about **1.65 V**, half of 3.3 V.
- **Tools → Serial Plotter** (115200) draws a live graph as you turn the knob, a great way to *see* the analog signal.

The ADC Is Not Perfectly Linear

Ideally the value tracks the voltage in a straight line. In reality the readings **flatten near the ends**, so you cannot tell ~0 V from ~0.1 V, or ~3.2 V from ~3.3 V.



- Fine for a knob or a light sensor
- Matters for **precise** measurement

What you learned:

- The board has **15 ADC pins**, each 12-bit (0 to 4095)
- `analogRead()` is all it takes to read one
- The same GPIO 4 was a digital input in Project 1

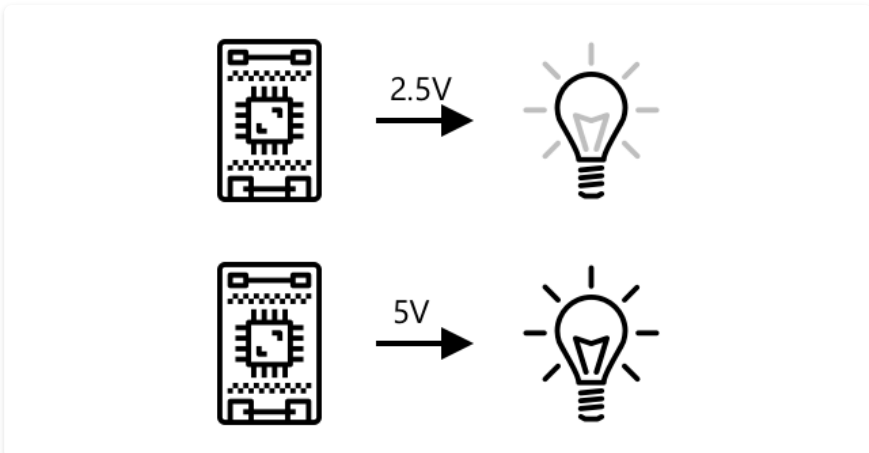
Controlling Actuators 

Analog voltages, PWM, and simple on/off

Analog Actuators & the DAC

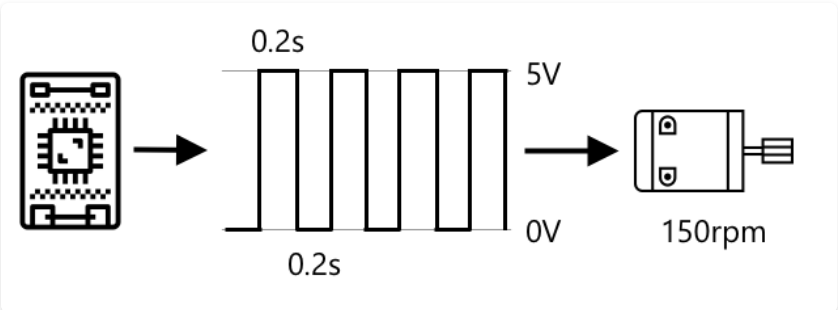
An **analog** actuator responds to the **voltage** supplied, like a **dimmable light**.

- The device is digital, so it needs a **DAC** (digital-to-analog converter)
- The DAC turns 0s and 1s into a real voltage the actuator can use

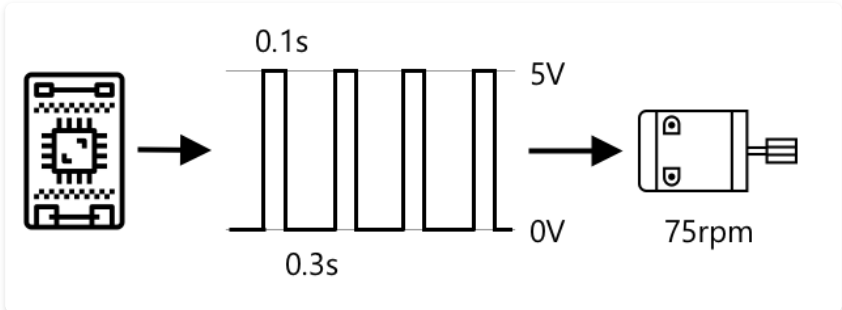


PWM: Faking Analog with Digital

Pulse-Width Modulation sends rapid on/off pulses. The **duty cycle** (the % of time it's on) sets the effective power.



50% duty → motor at **150 RPM**



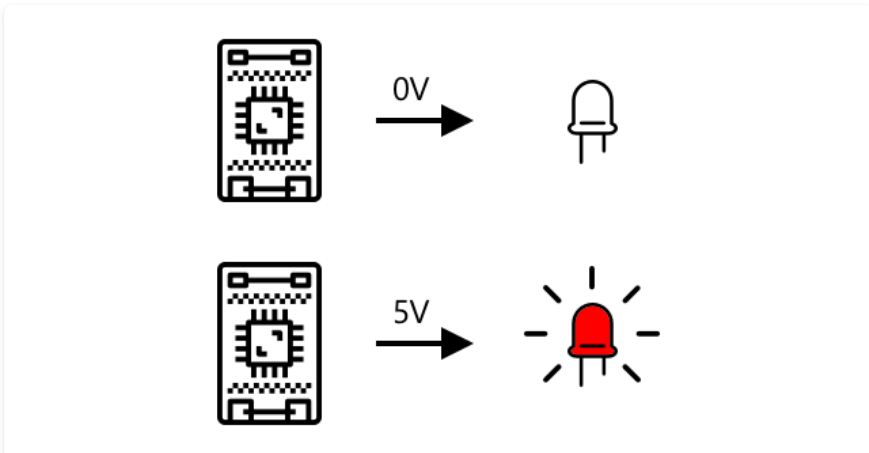
25% duty → motor at **75 RPM**

Digital Actuators

The simplest actuators have just **two states**: on or off.

- An **LED**: 0 V = off, 5 V = on
- A **relay**: a small voltage switches a larger circuit
- Complex ones (screens) need **libraries**

```
digitalWrite(LED_PIN, HIGH); // on
digitalWrite(LED_PIN, LOW);  // off
```



PROJECT 3 · PWM OUTPUT

PWM (Analog Output): Fading an LED

Produce an analog-looking output from a digital pin to smoothly fade an LED.

Builds on: Project 1's `digitalWrite()`, now switching so fast the eye blurs it into a brightness level instead of a blink.

⏱️ ~20 min 📊 Beginner 🔖 Prerequisite: Project 1

⬇ Download the sketch

Save `Project_3_ESP32_PWM.ino`, then open it in the

Project 3 · PWM Analog Output 🌗

Fade an LED smoothly using the ESP32's built-in LED PWM controller.

- Understand pulse width modulation and duty cycle
- Use the 16-channel LED PWM controller, configurable frequency and resolution
- Drive a pin with `ledcAttach()` and `ledcWrite()` (Core v3.x+ API)
- Compare a digital on/off output with an analog-style dimming one

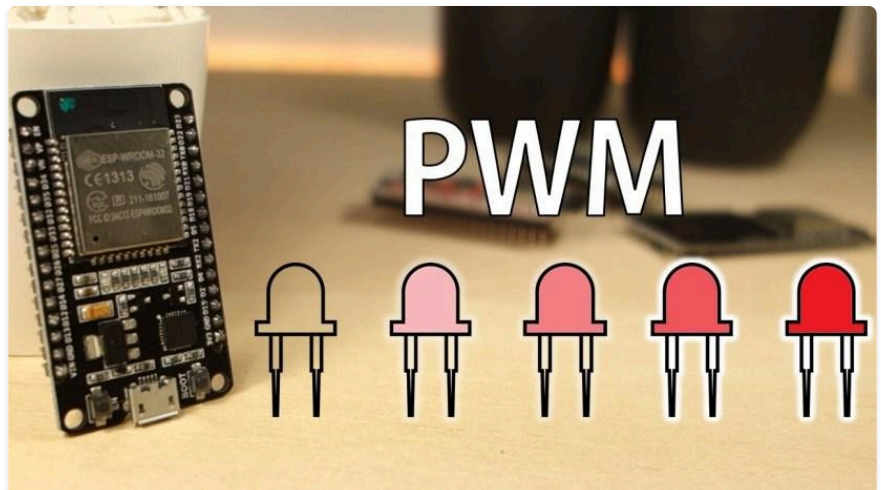
Key pin: LED `4` (PWM)

What Is PWM?

A digital pin is only ever fully **ON** (3.3 V) or **OFF** (0 V), it cannot output 1.6 V. But switch it on and off **very fast** and the LED's average brightness follows the **duty cycle**, the fraction of time it is ON.

- **0%** duty → dark
- **50%** duty → medium
- **100%** duty → full brightness

At **5000 Hz** the switching is far faster than your eye, so you see a steady brightness, not flicker. That is **Pulse Width Modulation**.



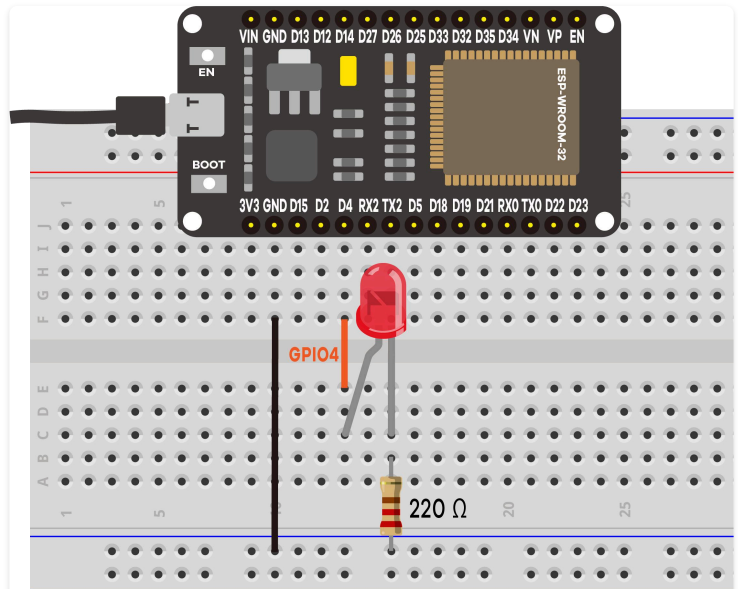
Two Functions Run the Controller

The ESP32 has a dedicated **LED PWM controller**. With Core **v3.x+** it takes just two calls:

```
// setup(): enable PWM on the pin
ledcAttach(ledPin, 5000, 8); // pin, 5 kHz, 8-bit
// anytime: set the brightness
ledcWrite(ledPin, dutyCycle); // 0 = off .. 255 = full
```

- One call picks the pin, frequency, and resolution and allocates a channel for you
- **8-bit** resolution gives 256 levels (0 to 255)
- v3.x+ `ledcWrite()` takes the **pin**, not a channel

Wiring is three connections: GPIO 4 → **220 Ω** → LED anode, cathode → **GND**.



The Fade Loop

```
void loop() {
  for (int dutyCycle = 0; dutyCycle <= 255; dutyCycle++) {   // fade in
    ledcWrite(ledPin, dutyCycle);
    delay(15);
  }
  for (int dutyCycle = 255; dutyCycle >= 0; dutyCycle--) {   // fade out
    ledcWrite(ledPin, dutyCycle);
    delay(15);
  }
}
```

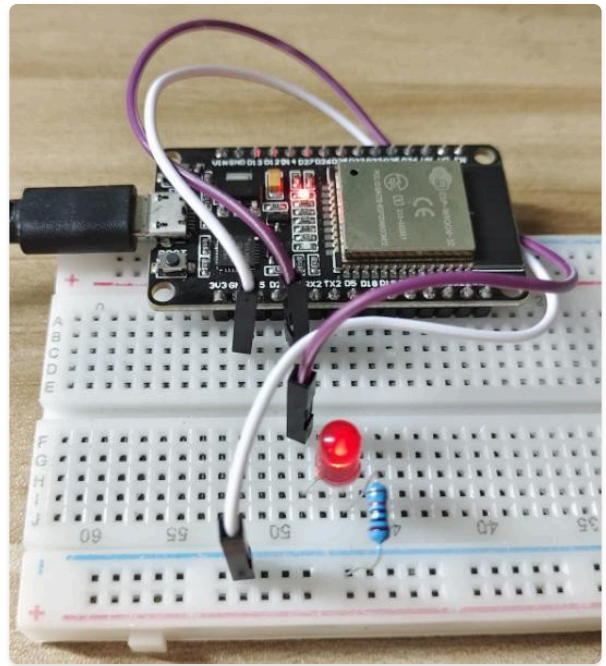
- Each step nudges the duty cycle by one; `delay(15)` spaces them, so a full fade is 256 × 15 ms ≈ **3.8 s**.
- The hardware does the fast switching; the code only says *how bright*.
- The sketch prints one line per **direction change**, not per step, so the monitor stays readable.

It Breathes

The LED breathes smoothly brighter and dimmer, over and over.

What you learned:

- Fast digital switching plus a duty cycle gives an **analog-looking** output
- Any output-capable pin can be a PWM pin
- Low PWM frequencies flicker; 5000 Hz is safe for LEDs
- The same trick sets motor speed and mixes RGB color (Project 6)



Components of an IoT Application

The Thing, the Internet, and Decisions

The Thing + The Internet

🔌 The Thing

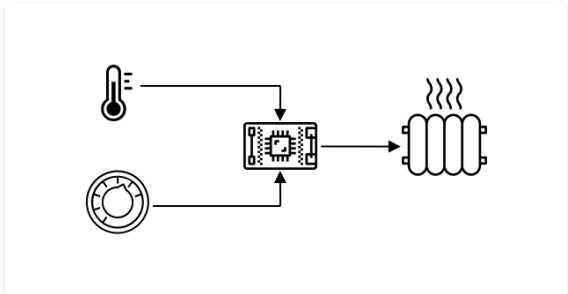
A device that interacts with the physical world, usually a small, low-power computer with **sensors** and **actuators**.

☁️ The Internet

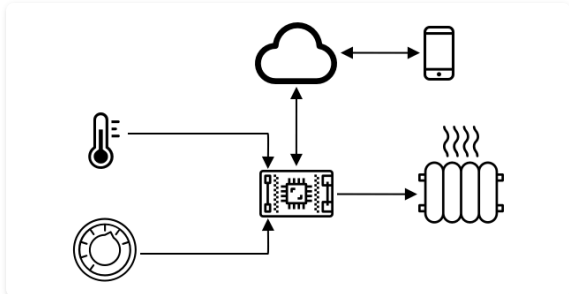
Cloud services the device connects to, in order to **receive** telemetry, **store & process** data, and **send commands** back.

> ⓘ **Note**
>
> Add **decision-making** (databases, AI, other data sources) and the loop is complete: **sense** → **send** → **decide** → **act**.

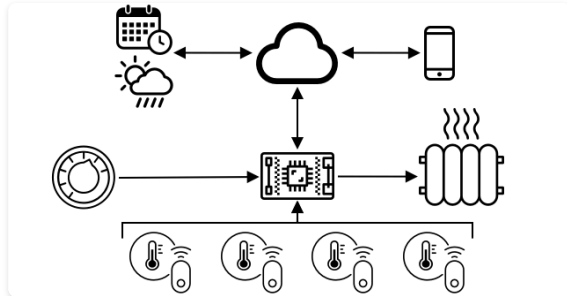
A Thermostat Gets Smarter



Basic: temperature + dial → heater



Connected: cloud + phone app



Smart: AI + calendar + weather

Each step adds more **data** and more **decision-making** in the cloud.

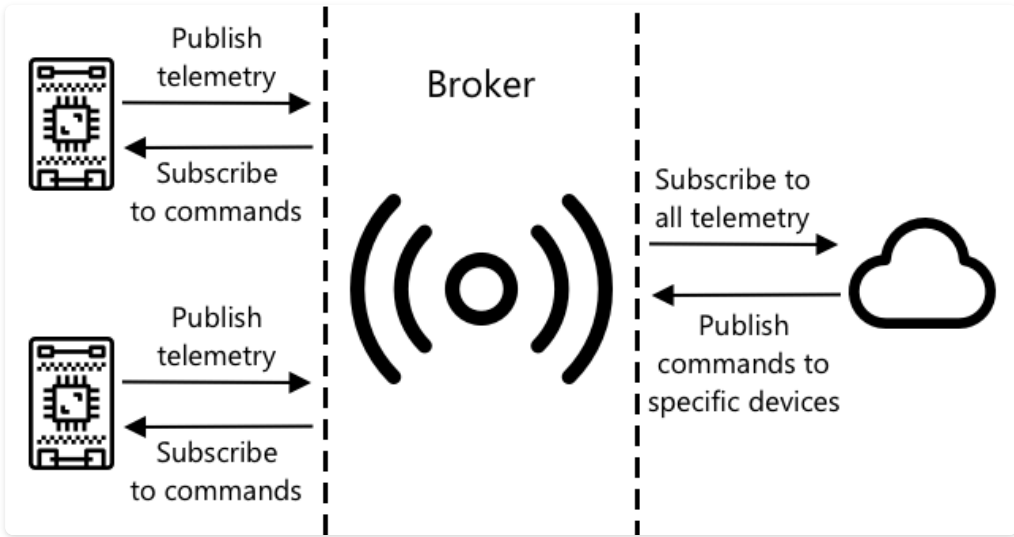
IoT Communication 

How devices and the cloud talk

Publish / Subscribe

Most IoT messaging uses **pub/sub** through a **broker**:

- Devices **publish** telemetry to topics
- Devices **subscribe** to command topics
- The **broker** routes messages between them
- Cloud services subscribe to telemetry & publish commands



Popular Protocols

MQTT

Lightweight, designed for IoT, pub/sub. *(Our focus.)*

AMQP

Robust enterprise messaging.

HTTP / HTTPS

The web standard: simple, but higher overhead.

> **Important**
>
> **MQTT** is the most popular protocol for IoT: small, efficient, and built for constrained devices.

PROJECT 5 · WI-FI / WEB

LED Switch Web Server

The ESP32 hosts a web page with ON/OFF buttons that control two LEDs from your phone or laptop.

Builds on: Project 1's `digitalWrite()`, now triggered by a browser over Wi-Fi instead of a wire.

⏱️ ~30 min 📊 Intermediate

🔖 Prerequisite: Project 1, Foundation 3

⬇ Download the sketch

Save `Project_5_ESP32_LED_Switch_Web_Server.ino`, then open it in the Arduino IDE (setup: **Foundation 3**).

Remember to set your Wi-Fi credentials

Project 5 · LED Switch Web Server 🌐

Toggle two LEDs from a web page hosted directly on the board, no cloud involved.

- Connect to Wi-Fi and find the board's IP address
- Host a web page from the ESP32 itself
- Read an HTTP request and route it to an action
- Keep the page in sync with each output's state

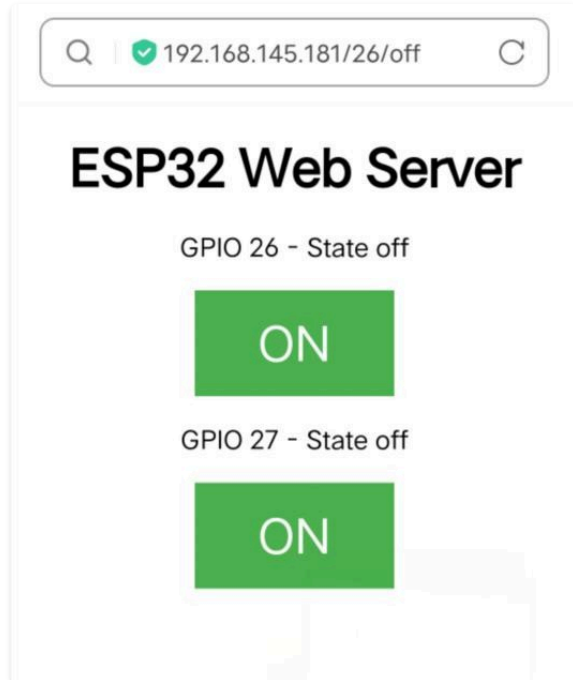
Key pins: LEDs `26`, `27`

Open the guide ↗

How an ESP32 Web Server Works

1. The ESP32 joins your Wi-Fi and gets an **IP address**
2. It **listens** on port 80 for browsers
3. Visit `http://<ESP_IP>/` and it sends back an HTML page
4. Each button is a link to a URL like `/26/on`; the board sees that path, switches the LED, and re-sends the page with the new state

That request-then-respond loop is the heart of **every** web-server project in this kit (5 to 9).

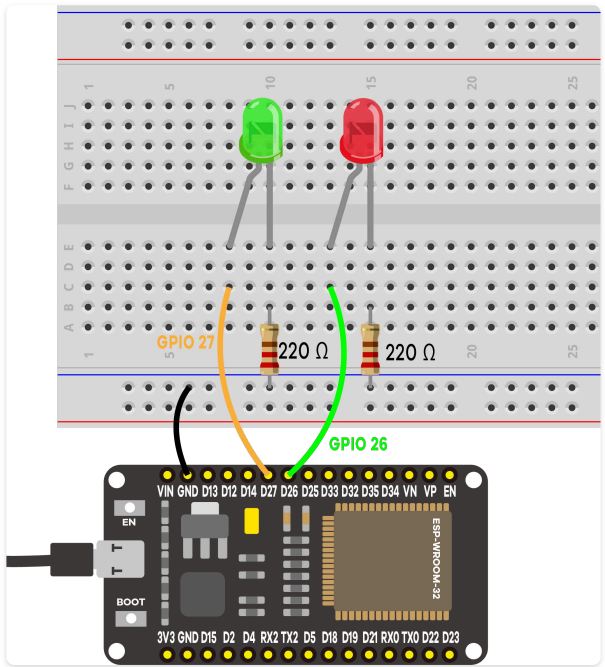


Connecting to Wi-Fi

Set your network name and password at the top of the sketch, then it joins and prints a clickable address:

```
WiFi.begin(ssid, password);   // returns at once
while (WiFi.status() != WL_CONNECTED) {
  delay(500); Serial.print("."); // a dot per 0.5 s
}
Serial.println(WiFi.localIP()); // type this in a browser
server.begin();                 // listen on port 80
```

The ESP32 radio is **2.4 GHz only**. Endless dots mean wrong credentials or a 5 GHz network. Wiring: two LEDs, each GPIO → 220 Ω → anode, cathode → GND.



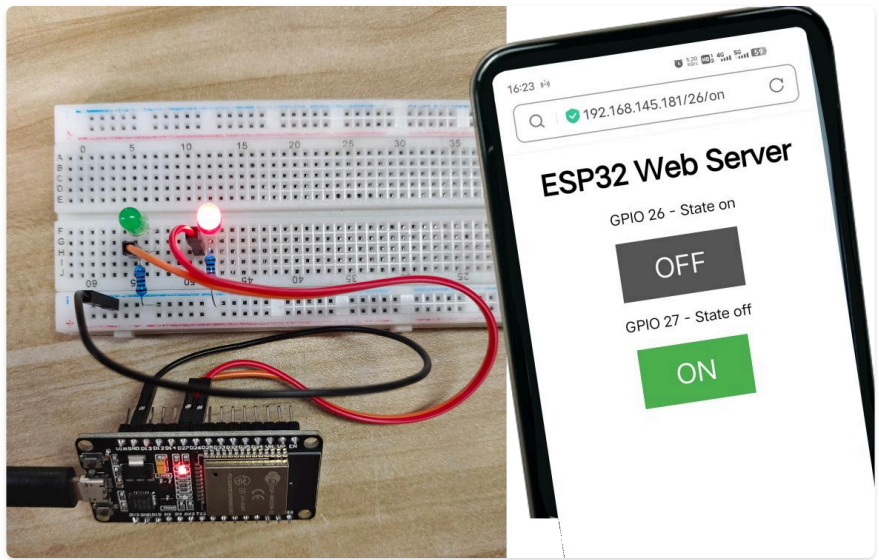
HTTP Is Just Text

```
WiFiClient client = server.available();   // did a browser connect?
// ... read bytes into `header` until the BLANK line that ends the request ...
client.println("HTTP/1.1 200 OK");        // status line: it worked
client.println("Content-type:text/html"); // render the reply as HTML
client.println();                         // blank line ends our header
if (header.indexOf("GET /26/on") >= 0) {  // the URL carries the command
  digitalWrite(output26, HIGH); output26State = "on";
}
client.println("<!DOCTYPE html>...");     // build the page, with current state
```

- A request is text lines ended by a **blank line**; a response is a status line, headers, a blank line, then the page.
- The **URL carries the command**: links like `/26/on`, and `indexOf()` returns `>= 0` when the browser asked for one.
- Port **80** is the default for `http://`, so the typed address needs no `:port`.

The Board Tracks Its Own State

The ESP32 cannot read back what an output pin is doing, so it keeps `output26State` and `output27State` and writes the current value into every page. The browser always shows what the board **believes**.



What you learned:

- Your board is now a **server**: the step that makes it IoT
- It joins 2.4 GHz Wi-Fi and the router hands it an IP
- HTTP is text, and the URL carries the command
- Tracked state matters when a button changes it too (Project 8)

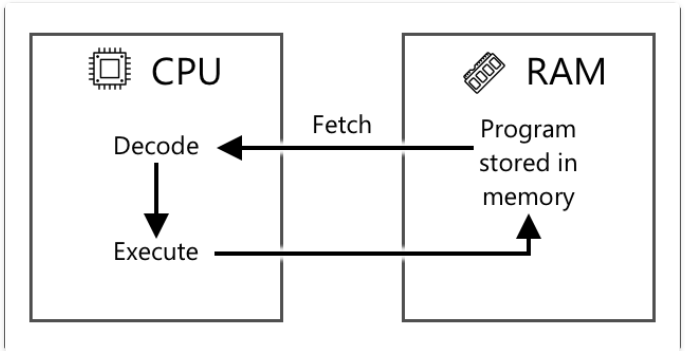
Inside a Microcontroller 

CPU · Memory · I/O

🧠 The CPU

The **brain** that runs your code, driven by a **clock** that ticks millions or billions of times a second.

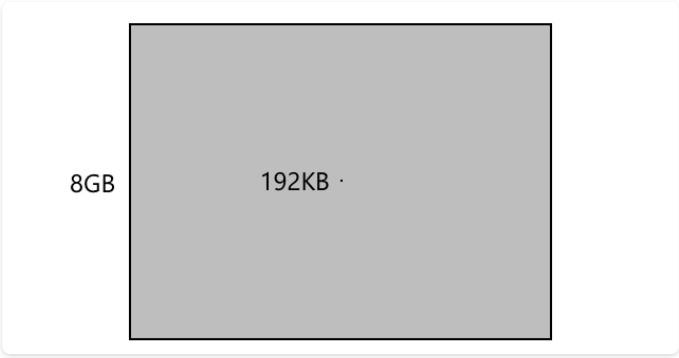
- Each tick = one step of the fetch–decode–execute cycle
- Speed in **Hz**: 1 MHz = 1M ticks/s, 1 GHz = 1B ticks/s
- An MCU runs at ~**120 MHz**; a PC at several **GHz**
- Slower = **cooler & lower power** → months on a battery



💾 Memory

Two kinds, both **tiny** next to a PC:

- **Program memory**: non-volatile, keeps code without power
- **RAM**: volatile, holds running data; lost on power-off
- An MCU: ~**192 KB** RAM & **4 MB** storage
- A PC: **8 GB** RAM & **500 GB** storage (*~40,000× more RAM*)



The dot in the center is 192 KB vs 8 GB

🔌 Input / Output

Microcontrollers talk to the world through **general-purpose I/O (GPIO)** pins, each configurable in software:

🧠⬅️ Input

Read values **from sensors**

🧠➡️ Output

Send signals **to actuators**

Thank you!

For spending the day building the Internet of Things with us.

Questions? Ask now, or write to me any time.

Slides, labs, and resources

KGSP Summer Enrichment Program 2026
KAUST Academy, KAUST

Salah Abdeljabar
salah.abdeljabar@kaust.edu.sa