TypeScript meets IoT: Safe programming for embedded devices

Soumaya Erradi
Senior Software Developer & IT and Electronics Instructor

Soumaya Erradi

- Senior software developer @ Atlantis
- Frontend and Web3 specialist
- International speaker
- IT and electronics divulger

We all have unused hardware lying around...

But why don't we use it?

- Requires C++ (which is not beginner-friendly)
- Debugging is painful 😖
- Different microcontrollers have different configurations

Wouldn’t it be great if we could just use Typescript?

Arduino

Arduino

- An **open-source electronics platform** for prototyping.
- Uses **microcontrollers** to control circuits, sensors, and motors.
- Programming is done via the **Arduino IDE (C++ based)**.

Arduino

- **Easy to use** for beginners and professionals.
- **Huge community** with lots of tutorials and libraries.
- Used in **robotics, IoT, home automation, and education**.

Arduino

- The first Arduino was created in 2005 in Italy.
- “Arduino” is named after a bar in Ivrea, Italy!
- Over **30 million Arduino boards** have been sold worldwide.

Why Typescript for IoT?

Typescript

- A superset of JavaScript with static typing
- Developed by Microsoft, adopted widely in the industry
- Enhances JavaScript without replacing it

Why Typescript?

- Type safety: Catch errors at compile-time
- Better tooling: IntelliSense, autocomplete, and refactoring
- Improved maintainability: Self-documenting code
- Large ecosystem and community support

Basic Typescript

Type Annotations & Inference

```
// Explicit type annotation

let age: number = 30;
let name: string = "Alice";



// Type inference

let isComplete = true; // inferred as boolean
```

Basic Types

- Primitives: string, number, boolean
- Arrays: number[] or Array<number>
- Tuples: [string, number]
- Enums: enum Direction { Up, Down, Left, Right }
- Any & Unknown: any vs unknown

Basic Types

Question: What is the *less general* type of x ?

```
const x = 42;
```

In other words, we are looking for a type T such that:

1. X has type T
2. There are no subtypes of T that satisfy the property 1

Basic Types

Answer: 42 is the *less general* type of 42

```
const x: 42 = 42

// This works too, but is more general
const y: number = 42
```

Basic Types

Question: What is the *less general* type of x ?

```
const x = "hello world"
```

Basic Types

Answer: "hello world" is the *less general* type of "hello world"

```
const x: "hello world" = "hello world"

// This works too, but is more general
const y: string = "hello world"
```

Basic Types

Note that Typescript performs type inference on variables

```
const x = 42 // inferred as `42`

let x = 42 // inferred as `number`
```

Basic Types

Question: What is the *less general* type of x ?

```
const x = [1, 2]
```

Basic Types

Answer: : The [1, 2] *tuple* is the *less general* type of [1, 2]

```
const x:[1, 2] = [1, 2]

// This works too, but is more general
const y:Array<number> = [1, 2]
```

Basic Types

Typescript only infers primitives as constants by default.

Use the *as const* keyword to narrow the inferred type:

```
// Inferred as Array<number>
const x = [1, 2]

// Inferred as the [1, 2] tuple
const y = [1, 2] as const
```

Basic Types

Spread in tuples.

Question: What is T defined as?

```
type Arr1 = [1, 2]
type Arr2 = [4, 5]

type T = [...Arr1, 3, ...Arr2]
```

Basic Types

Spread in tuples.

Answer: T is the [1, 2, 3, 4, 5] tuple

```
type Arr1 = [1, 2]
type Arr2 = [4, 5]

type T = [...Arr1, 3, ...Arr2] // => [1, 2, 3, 4, 5]
```

Basic Types

Casting values to types.

Question: what is T defined as?

```
const x = 42

type T = typeof x
```

Basic Types

Casting values to types.

Answer: T is defined as 42

```
const x = 42 // inferred as `42`

type T = typeof x // => `42`
```

Basic Types

The *unknown* type

u*nknown* is a supertype of all types

```
const x: unknown = 42
```

In other words, any value can be assigned to the type unknown.

Basic Types

The *never* type

never is an inhabited type

```
const x: never = // there is nothing I can assign to never
```

In other words, there are no values that have type never.

Intermediate Typescript

Union & Intersection Types

Union types (string | number) allow flexibility:

```
let id: string | number;
id = "123"; // OK
id = 456; // OK
```

Union

Question: What is the result of T ?

```
type T = number | never
```

Union

Answer: T is equivalent to number

```
type T = number | never // => number
```

Union

Question: What is the result of T ?

```
type T = 42 | number
```

Union

Answer: T is equivalent to number

```
type T = 42 | number // => number
```

Union

Question: What is the result of T ?

```
type T = number | unknown
```

Union

Answer: T is equivalent to unknown

```
type T = number | unknown // => unknown
```

Union & Intersection Types

Intersection types (A & B) combine multiple type definitions:

```
type Admin = { role: string };
type User = { name: string };
type AdminUser = Admin & User;
```

Intersection

Question: What is the result of T ?

```ts
type T = number & 42
```

Intersection

Answer: T is equivalent to 42

```
type T = number & 42 // => 42
```

Intersection

Question: What is the result of T ?

```
type T = number & unknown
```

Intersection

Answer: T is equivalent to number

```
type T = number & unknown // => number
```

Intersection

Question: What is the result of T ?

```
type T = 42 & never
```

Intersection

Answer: T is equivalent to never

```
type T = 42 & never // => never
```

Intersection

Question: What is the result of T ?

```
type T = number & string
```

Intersection

Answer: T is equivalent to never

```
type T = number & string // => never
```

There is no x such that x has both type *number* and *string*

Generics

```
// Making functions and types reusable

function identity<T>(arg: T): T {
  return arg;
}
```

Utility Types

- Partial<T>: Makes all properties optional
- Readonly<T>: Prevents mutation
- Record<K, T>: Defines an object type with specified keys

Utility Types

- Partial<T> example:

```
type User = { id: number; name: string };
let partialUser: Partial<User> = { name: "Alice" }; // id is optional
```

Utility Types

- Readonly<T> example:

```
type ReadonlyUser = Readonly<User>;
const user: ReadonlyUser = { id: 1, name: "Alice" };
user.name = "Bob"; // Error: Cannot assign to 'name' because it is a read-only property.
```

Utility Types

- Record<K, T> example:

```
type UserRole = "admin" | "editor" | "viewer";
type Permissions = Record<UserRole, string[]>;
const userPermissions: Permissions = {
  admin: ["create", "delete", "update"],
  editor: ["update"],
  viewer: ["read"]
};
```

Advanced Typescript

Mapped Types

```
type Optional<T> = { [K in keyof T]?: T[K] };

// Modifies properties dynamically
```

Branded Types (Nominal Typing)

- TypeScript uses structural typing, but we can enforce nominal typing using branded types.
- **Use Case**: Prevent accidental type mismatches
  - e.g., UserId vs. ProductId.

Branded Types (Nominal Typing)

```typescript
type UserId = string & { readonly brand: unique symbol };

function createUserId(id: string): UserId {
    return id as UserId;
}

let id: UserId = createUserId("abc123");
// let invalidId: UserId = "abc123"; // Error!
```

Discriminated Unions

- Best practice for handling multiple cases in a type-safe manner.
- **Use Case**: Type-safe handling of different object variants.

```ts
interface Circle {
    kind: "circle";
    radius: number;
}

interface Square {
    kind: "square";
    sideLength: number;
}

type Shape = Circle | Square;

function getArea(shape: Shape): number {
    switch (shape.kind) {
        case "circle": return Math.PI * shape.radius ** 2;
        case "square": return shape.sideLength ** 2;
    }
}
```

Mapped Types

Question: What is T defined as?

```ts
// The `K extends string` is just enforcing
// a constraint on the generic param
type Helper<K extends string, T> = {
  [k in K]: T
}

type T = Helper<"a" | "b", number>
```

Mapped Types

Answer: T is defined as {a: number, b: number}

```ts
// The `K extends string` is just enforcing
// a constraint on the generic param
type Helper<K extends string, T> = {
  [k in K]: T
}

type T = Helper<"a" | "b", number>
// => {a: number, b: number}
```

Mapped Types

This is actually the definition of the *Record* utility type

```
type Record<K extends keyof any, T> = {
  [k in K]: T
}
```

Mapped Types

Question: What is T defined as?

```ts
type Helper<T, K extends keyof T> = {
  [k in K]: T[k]
}
const o = {
  x: 0,
  y: 1,
  z: 2,
}
type T = Helper<typeof o, "x" | "y">
```

Mapped Types

Answer: T is defined as {x: number, y: number}

```ts
type Helper<T, K extends keyof T> = {
  [k in K]: T[k]
}
const o = {
  x: 0,
  y: 1,
  z: 2,
} // inferred as { x: number, y: number, z: number }
type T = Helper<typeof o, "x" | "y">
// => { x: number, y: number }
```

Mapped Types

This is actually the definition of the *Pick* utility type

```
type Pick<T, K extends keyof T> = {
  [k in K]: T[k]
}
```

Advanced Utility Types

- **Pick<T, K>**: Extracts specific properties from a type.
- **Use Case**: Useful for defining lightweight views or partial structures from complex types.

Pick Utility Type

```
type User = {
    id: number;
    name: string;
    email: string;
};

type UserPreview = Pick<User, "id" | "name">;

const user: UserPreview = {
    id: 1,
    name: "Alice"
    // email: "alice@example.com" // Error! Property 'email' does not
exist on type 'UserPreview'.
};
```


TS


ARDUINO

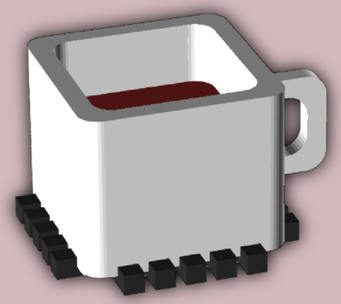
Typescript options for Arduino

Espruino

Espruino is a JavaScript interpreter that runs directly on microcontrollers.

No need for a PC connection after programming.

Works with ESP8266, ESP32, and other low-power boards.



Espruino

What can you do with Espruino?

- ✔ Run JavaScript directly on the board
- ✔ Control LEDs, motors, and sensors
- ✔ Build standalone IoT projects

```
const led = D13;
const servo = require("servo").connect(D9);

setInterval(function() {
    digitalWrite(led, !digitalRead(led)); // Blink LED
}, 1000);

const angle = 0;
setInterval(function() {
    angle = (angle === 0) ? 1 : 0;
    servo.move(angle);
}, 2000);
```

Espruino: Pros & Cons

- ✅ Runs directly on board
- ✅ Supports ESP32/ESP8266
- ❌ Limited TypeScript support
- ❌ Requires flashing firmware

Arduino IoT Cloud

The official cloud-based Arduino solution.

Allows remote control of Arduino.

Works with the Arduino IoT Cloud API.

What can you do with Arduino IoT SDK?

- ✔ Monitor and control Arduino remotely
- ✔ Send data to a cloud dashboard
- ✔ Control a servo or LED from a mobile app

```
import { ArduinoIoTCloud } from "arduino-iot-js";

const client = new ArduinoIoTCloud({
    clientId: "your_client_id",
    clientSecret: "your_client_secret",
});

const THING_ID = "your_thing_id";

async function main() {
    await client.connect();
    console.log("Connected to Arduino IoT Cloud!");

    let ledState = await client.getThingProperty(THING_ID, "ledState");
    await client.setThingProperty(THING_ID, "ledState", !ledState);
}
main();
```

Arduino IoT SDK: Pros & Cons

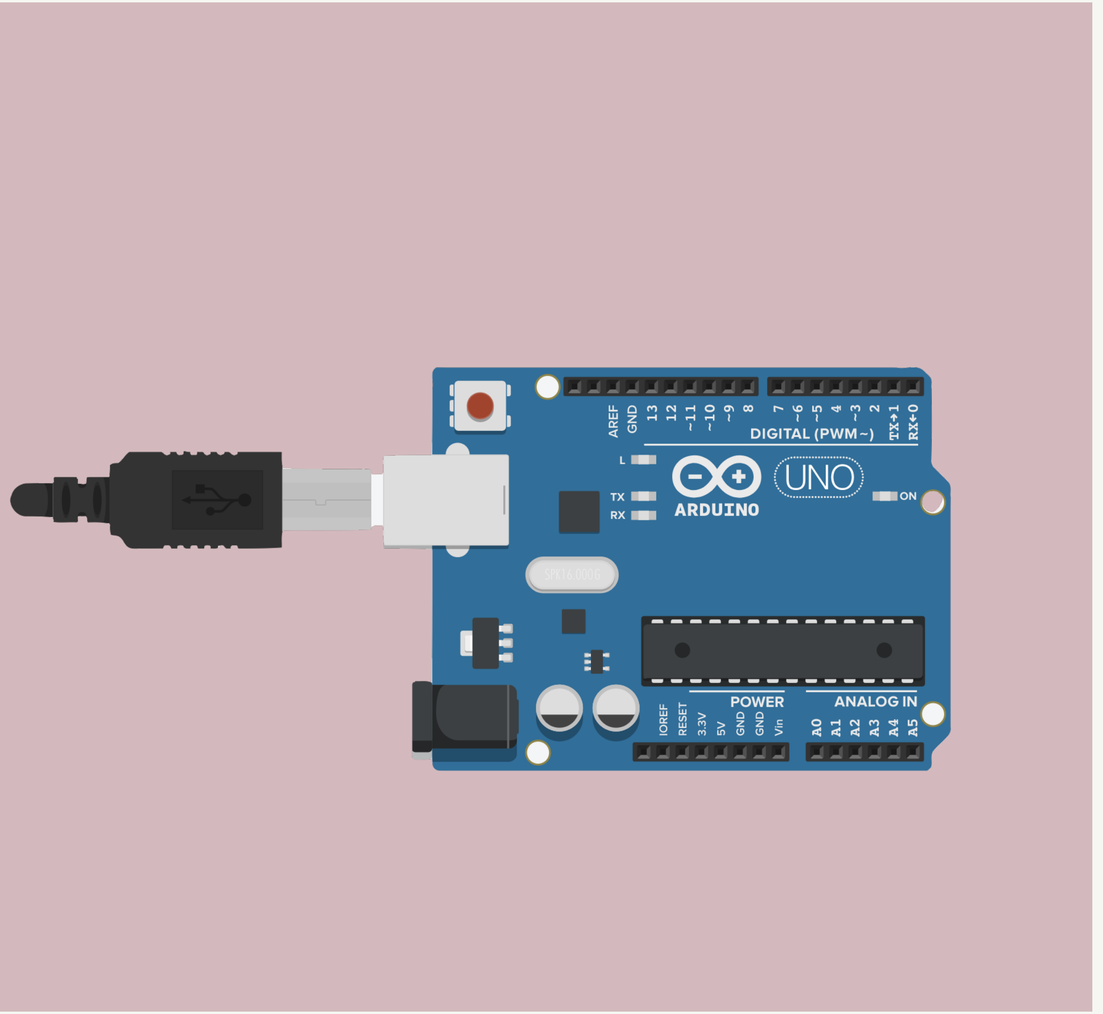
- ✅ Best for IoT applications
- ✅ Cloud-based remote control
- ❌ Requires internet access
- ❌ More setup required

Firmata

Protocol to communicate between a PC and Arduino.

Needs StandardFirmata uploaded to Arduino.

Used by Johnny-Five but can be used directly.



What can you do with Firmata?

- ✔ Real-time control from TypeScript
- ✔ Read/write digital and analog pins
- ✔ Automate motors, LEDs, and relays

```
import { Board } from "firmata";

const PORT = "/dev/ttyUSB0";

// Connect to Arduino
const board = new Board(PORT);

board.on("ready", () => {
    console.log("Arduino connected via Firmata!");

    // Blink LED on pin 13
    const ledPin = 13;
    board.pinMode(ledPin, board.MODES.OUTPUT);

    setInterval(() => {
        const currentValue = board.pins[ledPin].value || 0;
        board.digitalWrite(ledPin, currentValue ? 0 : 1);
        console.log(`LED is now ${currentValue ? "OFF" : "ON"}`);
    }, 1000);
});
```

Firmata: Pros & Cons

- ✅ More flexible than other libraries
- ✅ Works without library abstraction
- ❌ Harder to use than other libraries
- ❌ Needs a constant USB connection

Johnny-Five

A JavaScript robotics framework.

Works with TypeScript using Firmata.

Easiest way to control Arduino from TypeScript.

What can you do with Johnny-Five?

- ✔ Blink LEDs, control motors & sensors
- ✔ Integrate with web dashboards
- ✔ Automate robotics projects

```
import { Board, Led, Servo } from "johnny-five";

const board = new Board();

board.on("ready", () => {
    console.log("Arduino Ready!");

    const led = new Led(13);
    led.blink(1000);

    const servo = new Servo(9);
    let position = 0;

    setInterval(() => {
        position = position === 0 ? 180 : 0;
        servo.to(position);
    }, 2000);
});
```

Firmata: Pros & Cons

- ✅ Best TypeScript support
- ✅ Easiest to use
- ✅ Active community
- ❌ Requires USB connection

Winner: Johnny-Five 🏆

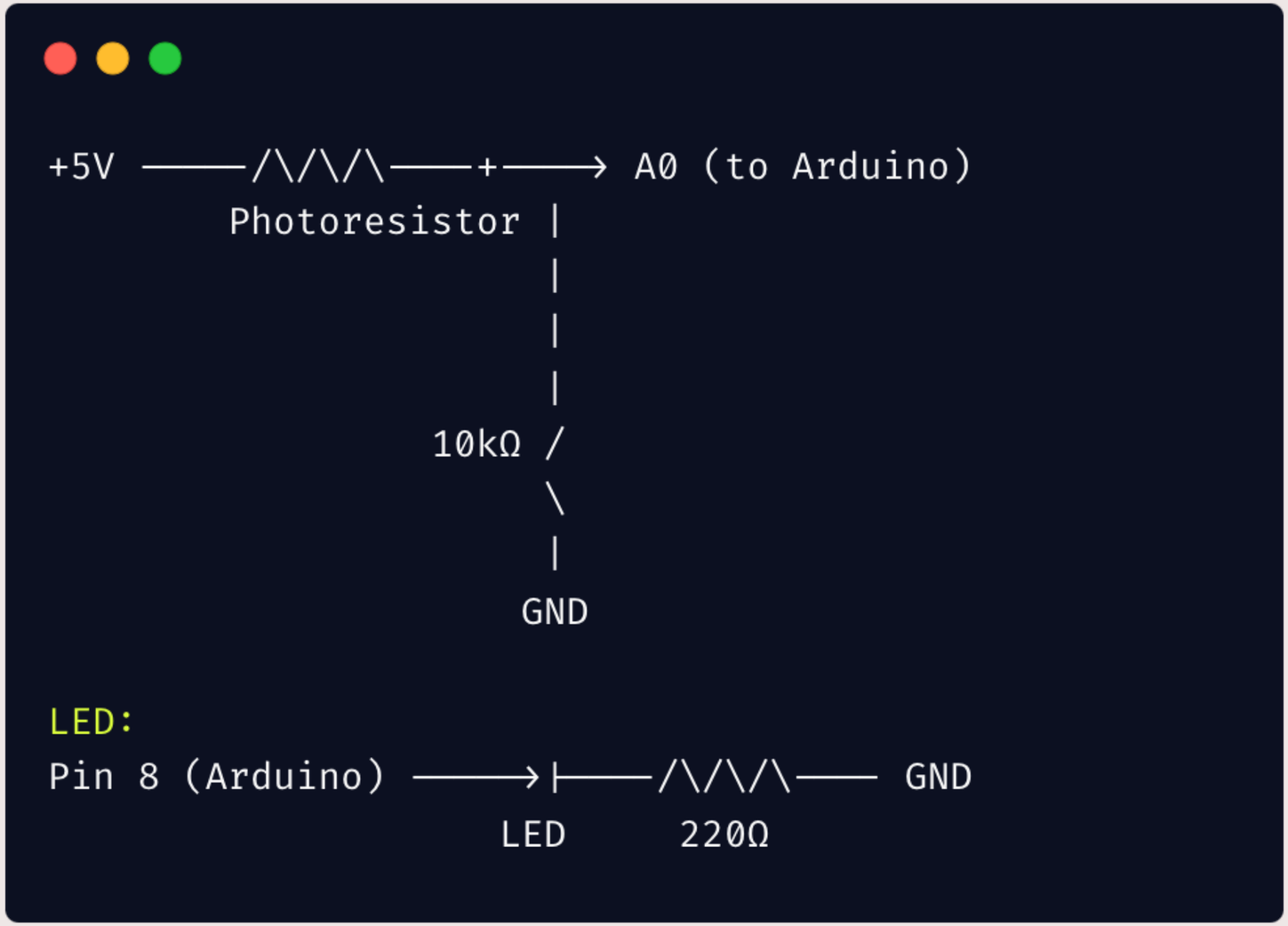
Johnny-Five

Johnny-Five is a **popular IoT framework** that lets us control embedded hardware using **JavaScript or TypeScript**.

How does it work under the hood?

- Uses Node.js + Firmata to communicate with hardware
- Works with Arduino, Raspberry Pi, ESP32, and more
- Now supports TypeScript definitions!

Demo


+5V ———/\/\/\———+———→ A0 (to Arduino)
Photoresistor |
|
|
|
10kΩ /
\
|
GND
LED:
Pin 8 (Arduino) ———→|———/\/\/\——— GND
LED
220Ω

```
void setup() {
  Serial.begin(9600);
}

void loop() {
  pinMode(8, OUTPUT);

  int lightLevel = analogRead(A0);
  Serial.println(lightLevel);

  if (lightLevel < 400) {
    digitalWrite(8, HIGH);
  } else {
    digitalWrite(8, LOW);
  }

  delay(500);
}
```

Dual-Photoresistor solar tracker with LED night mode

Real life use cases

- ⚡ **Dual photoresistors:**
  - The servo turns left or right based on which side has more light.
- 🌙 **Night mode:**
  - When both photoresistors detect low light, the LED turns ON.
- 💡 **Real-time adjustments:**
  - Servo constantly tracks the brighter light source.

Components needed

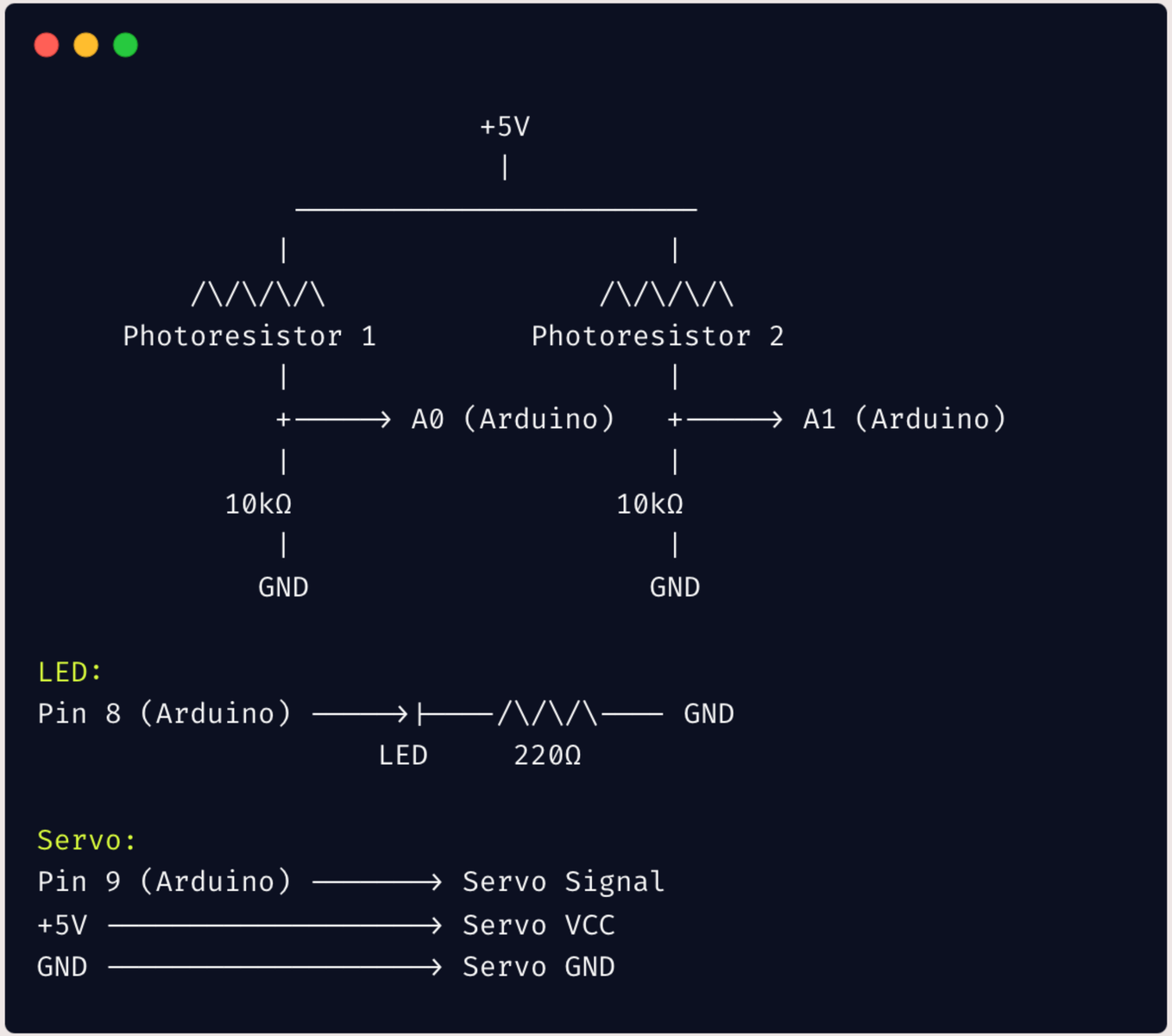
- **Arduino board:** main controller
- **2 Photoresistors**: Detect light intensity (left & right)
- **2 × 10kΩ Resistors**: Voltage divider for each LDR
- **Servo Motor**: adjusts to follow light
- **LED + 220Ω Resistor**: lights up in night mode

Circuit diagram

- **Left Photoresistor**: A0 (Analog Input)
- **Right Photoresistor**: A1 (Analog Input)
- **Servo**: Pin 9
- **LED**: Pin 8

Circuit diagram

Left Photoresistor	A0 (Analog Input)
Right Photoresistor	A1 (Analog Input)
Servo	Pin 9
LED	Pin 8


+5V
Photoresistor 1
Photoresistor 2
A0 (Arduino)
A1 (Arduino)
10kΩ
10kΩ
GND
GND
LED:
Pin 8 (Arduino)
LED
220Ω
GND
Servo:
Pin 9 (Arduino) ——→ Servo Signal
+5V ——→ Servo VCC
GND ——→ Servo GND

Real life use cases

- **Solar panel trackers** ➜ Align to sunlight for maximum efficiency. 🌞
- **Smart windows** ➜ Auto-adjust shades based on sunlight. 🏠
- **Security systems** ➜ Follow suspicious lights at night. 🚨

Wrapping Up 🚀

- TypeScript + IoT = ✅ Safer & Scalable
- Johnny-Five enables TS for embedded devices
- You don’t need C++ to build IoT projects!

serradi92@gmail.com

@sumyerradi

@sumy92

@soumaya-erradi

@soumayaerradi

@sumyerradi.bsky.social

Thank you!

Soumaya Erradi

www.soumayaerradi.it