Summer Internship Report

On

Open Source Hardware Studio an Simulation Platform

Submitted By

Ritesh Jadhav

B.Tech Computer Science and Engineering,
ITM Skills University, Navi-Mumbai

Under the Guidance of

Prof. Prabhu Ramachandran

Principal Investigator,
FOSSEE Project Department of Aerospace Engineering,
Indian Institute of Technology Bombay

Acknowledgement

I would like to express my deepest gratitude to **Prof. Prabhu Ramachandran** for providing me with the opportunity to be a part of the **FOSSEE Internship Program** and for supporting open-source engineering tool development at **IIT Bombay.** His vision and leadership have inspired students and researchers to actively contribute to the open-source ecosystem.

I would also like to sincerely thank **Prof. Rajesh Kushalkar, Senior Project Manager, Open Source Hardware (OSHW), FOSSEE, IIT Bombay**, for his valuable guidance, encouragement, and continuous support throughout the internship. His mentorship greatly contributed to my learning and professional growth.

My heartfelt appreciation goes to my mentors, **Mr. Pratik Bhosale, Project Research Associate, and Mr. Pratik Nemane, Project Research Assistant**, for their constant technical guidance, constructive feedback, and encouragement throughout the project. Their insights played a key role in refining ideas, overcoming challenges, and ensuring the successful completion of the tasks assigned to me.

I would also like to thank my fellow interns and colleagues at **OpenHW Studio** for their support, collaboration, and constructive discussions throughout the internship. Their motivation and teamwork created a positive learning environment and contributed significantly to my overall experience.

Finally, I extend my sincere gratitude to everyone associated with the **FOSSEE Project, IIT Bombay**, for fostering an environment of innovation, collaboration, and continuous learning. This internship has been an invaluable experience that strengthened my technical skills and motivated me to continue contributing to the open-source community.

Ritesh Jadhav,
ITM Skills University

Declaration

This internship proved to be a highly rewarding educational journey, granting me the chance to support the creation of virtual hardware modules while integrating functional improvements and debugging within **OpenHW Studio.** The experience offered significant exposure to professional open-source software practices, micro-controller simulation environments, and integrated development processes. The technical competencies and hands-on insights developed throughout this period will serve as a vital foundation for my upcoming academic projects and career objectives.

I wish to further extend my sincere appreciation to the members of the FOSSEE Team for their seamless coordination, technical mentorship, and unwavering encouragement during the program. Their collaborative spirit and proactive assistance fostered a productive workspace, proving instrumental in the effective delivery of all project milestones and responsibilities.

Ritesh Jadhav,
ITM Skills University

Table of Contents

10. Chapter 7 — Future Work
11. References

---

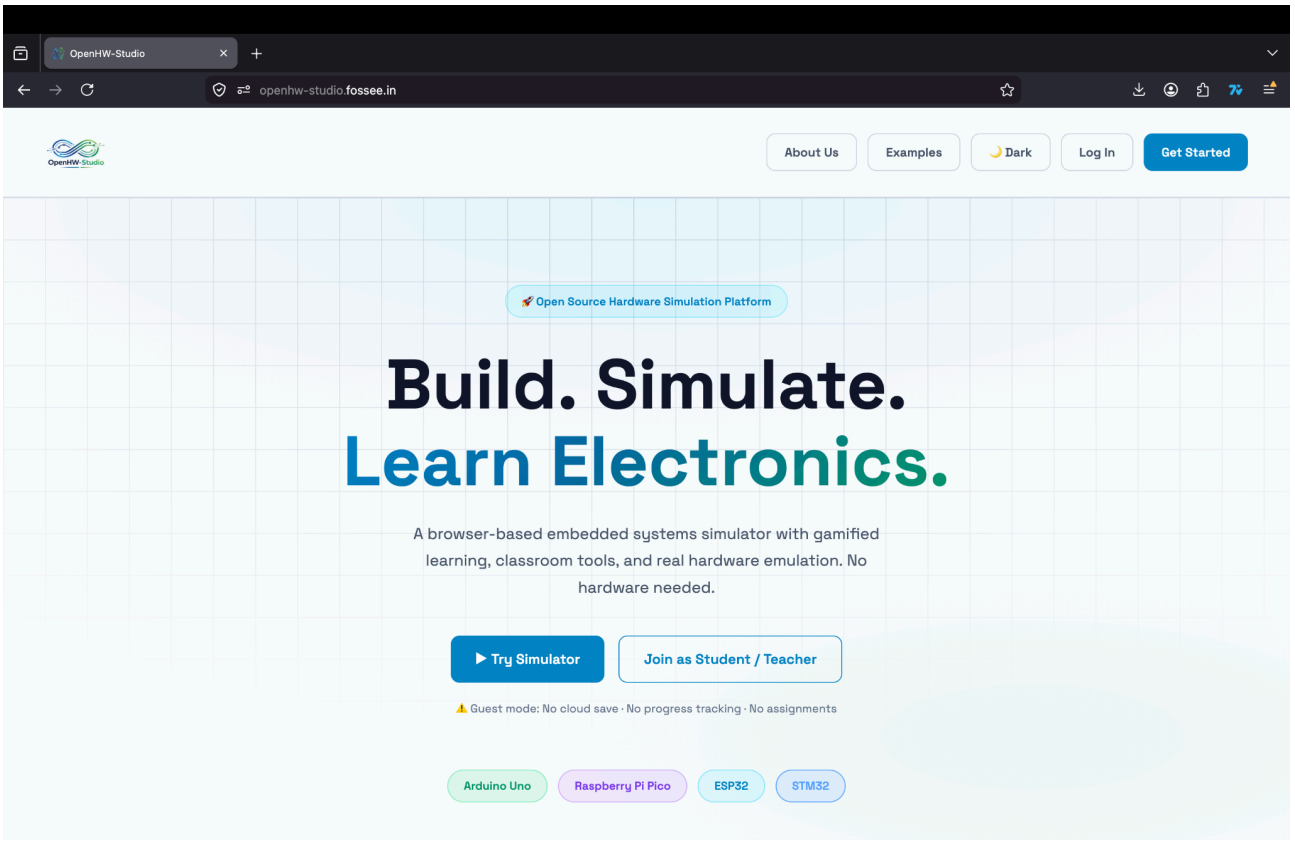
1. What is OpenHW Studio?

OpenHW Studio is a **browser-based electronics learning platform** built under the FOSSEE (Free and Open Source Software for Education) initiative at **IIT Bombay**. The goal is simple: let students build, wire, and run circuits entirely inside a web browser — no physical Arduino, no real hardware required.

Think of it like **Wokwi** or **Tinkercad Circuits**, but open-source, and tightly integrated with a classroom management system and a gamified learning track.

The platform has three major parts:

Module	What it does
Simulation Engine	A browser-based circuit + microcontroller emulator. Students wire up virtual components, write firmware (Arduino C++, MicroPython), and watch the circuit respond in real time.
Classroom Module	A virtual classroom similar to Google Classroom. Teachers create assignments and track student progress. Students join classes, attempt projects, and submit work.
Gamification Module	A game-like learning track with XP points, coins, and an Island Adventure Map of unlockable challenges. It makes learning electronics feel like progressing through a real game.



---

2. Project Architecture Overview

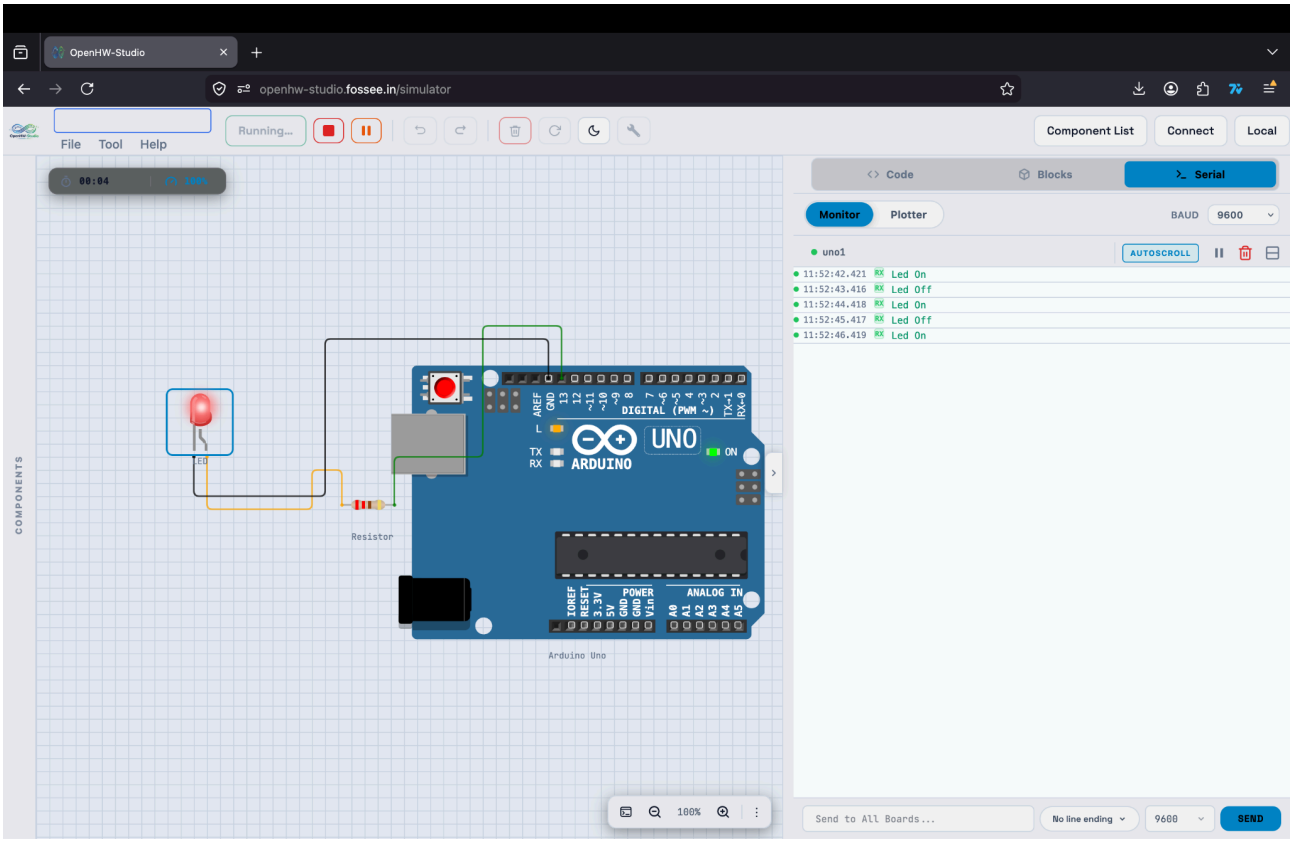
Before diving into my contributions, it helps to understand how the platform is built technically.

```
OpenHW Studio
│
├── Frontend (React + Vite)              ← What students see in the browser
│   ├── Pages: Landing, Classroom, Simulator, Gamification, Auth
│   ├── Simulation Worker (Web Worker)   ← Runs the emulator in a background thread
│   └── Component Library                ← All SVG + logic files for each component
│
├── Emulator Package (@openhw/emulator)  ← The simulation engine (TypeScript)
│   ├── BaseComponent.ts                 ← Parent class every component extends
│   ├── AVR Runner                       ← Simulates ATmega328P CPU (Arduino Uno)
│   ├── RP2040 Runner                    ← Simulates Raspberry Pi Pico
│   ├── Circuit Validation               ← Checks for wiring errors before simulation
│   └── Component Library                ← MQ2, DHT22, PIR, HC-SR04, ESP32, etc.
│
└── Backend (Node.js + Express)          ← The server
    ├── Compile API (/api/compile)       ← Runs arduino-cli on student's code
    ├── Classroom API                    ← Users, classes, assignments, progress
    ├── ESP32 Sandbox                    ← QEMU process manager for ESP32 simulation
    │   ├── qemuRunner.js                ← Spawns a QEMU process per session
    │   ├── networkProxy.js              ← Routes ESP32 WiFi to the real internet
    │   └── SimulatorBridge.h            ← C header injected at compile time
    └── MongoDB                          ← Stores users, classes, assignments, etc.
```

How a "Run" Works (Step by Step)

Here is what happens when a student presses the **Run** button on a regular Arduino project:

1. The frontend sends the student's C++ code to the backend at POST /api/compile.
2. The backend writes it to a temporary .ino file and runs arduino-cli compile targeting the arduino:avr:uno board profile.
3. arduino-cli produces a .hex file (machine code for the AVR chip).
4. The backend sends the .hex string back to the frontend.
5. The frontend feeds the .hex into the **Simulation Worker** — a background thread running the emulator.
6. The emulator "boots" the AVR CPU with that firmware and the virtual circuit starts running.
7. When a student moves a slider on a sensor, the UI fires an event → the emulator updates the component's state → the CPU reacts → LEDs turn on, the serial monitor prints output, etc.



---

3. My Contributions — Summary

#	Contribution	Type
1	MQ2 Gas Sensor	New Component
2	DHT22 Temperature & Humidity Sensor	New Component
3	Raindrop Module	New Component
4	Raindrop Pad	New Component
5	PIR Motion Sensor	New Component

#	Contribution	Type
6	HC-SR04 Ultrasonic Sensor	New Component
7	Server-Side ESP32 Simulation (QEMU + WiFi)	New Microcontroller
8	Membrane Keypad Fix	Bug Fix
9	Stepper Motor Fix	Bug Fix
10	A4988 Stepper Motor Driver Fix	Bug Fix
11	Raspberry Pi Pico — Performance Optimisation	Performance Fix
12	Classroom UI Redesign	UI/UX
13	Login & Sign-Up Pages with DiceBear Avatars	UI/UX
14	Island-Based Adventure Map with Badges	UI/UX & Gamification
15	Visual Guided Projects (Circuit PNG Images)	UI/UX
16	Transparent Circuit Image Export	New Feature
17	Assignment Notification System	New Feature

Chapter 1 — New Sensor Components Added

Understanding the Component Architecture First

Before I describe each sensor, it is worth spending a moment understanding how every component in this platform is built, because they all follow the exact same structure.

Every component lives in its own folder inside the emulator package and has three key files:

File	Role	Simple description
logic.ts	The brain	This TypeScript file contains a class that holds the sensor's internal state and knows how to respond when a pin voltage changes or when the student interacts with it on the canvas.
ui.tsx	The face	This React component controls what the student *sees* on the canvas — the SVG image, the sliders, the indicator lights, and the

File	Role	Simple description
		animations.
manifest.json	The ID card	A small JSON file that lists the sensor's pins, its physical dimensions on the canvas, and default values for its configurable settings.

All logic.ts classes extend a parent class called BaseComponent. Think of BaseComponent like a toolbox that gives every sensor the same built-in tools without us having to re-write them every time:

- setPinVoltage(pinName, volts) — Push a specific voltage onto a pin, which can then flow through the circuit to other components.
- getPinVoltage(pinName) — Read the current voltage on a pin (e.g. check whether VCC is connected).
- setState(newValues) — Update the component's internal state object, which automatically tells the UI to re-render.
- onEvent(event) — Called every time the student interacts with the sensor on the canvas (e.g. dragging a gas cloud, clicking a motion trigger button).
- onPinStateChange(pin, isHigh, cycles) — Called every time the voltage on a connected pin changes, so the sensor can react accordingly.

1.1 MQ2 Gas Sensor

Code location: openhw-studio-emulator/src/components/MQ2-gas-sensor/

What is the MQ-2 in Real Life?

The MQ-2 is a metal oxide semiconductor gas sensor. Inside the real physical sensor there is a small ceramic coil that heats up to around 200°C. At this temperature, gas molecules land on the surface of a tin dioxide (SnO_2) material and cause its electrical resistance to decrease. The more gas there is, the lower the resistance, and the higher the output voltage.

The MQ-2 can detect a wide range of gases including LPG (lighter fluid), propane, methane (natural gas), smoke particles, alcohol vapour, hydrogen gas, and carbon monoxide. This makes it very useful for smoke alarms, gas leak detectors, and automated ventilation systems.

The sensor has four pins:

- **VCC** — Power input (5V)
- **GND** — Ground (0V)
- **AO** — Analog Output: a continuously varying voltage from 0V to ~5V that rises proportionally as gas concentration increases

- **DO** — Digital Output: a simple HIGH/LOW alarm pin that goes LOW (0V) when gas exceeds a potentiometer-set threshold

The DO pin uses **Active-LOW logic** — meaning the alarm is active when the pin is LOW, not HIGH. This is a convention in electronics to make alarm circuits more fail-safe.

Why OpenHW Studio Needed This Component?

Before I added the MQ-2, students who wanted to build gas alarm projects had to either skip the project entirely or substitute it with a plain potentiometer. A potentiometer produces a varying voltage like an MQ-2, but it has no concept of "gas levels" or "threshold alarms," so the student's code doesn't feel like a real gas detector project.

Adding the MQ-2 allows students to learn how ADC (Analog to Digital Conversion) works, how to calibrate a threshold, how Active-LOW digital alarms work, and how to build real-world safety systems — all without needing physical hardware.

How I Implemented the Logic

The logic lives in GasSensorLogic which extends BaseComponent. The most important function is updateVoltages(), which runs every time the student changes the gas level via the UI or every time a power pin changes:

```
// From: MQ2-gas-sensor/logic.ts

export class GasSensorLogic extends BaseComponent {
  // Called when gas level changes, or when VCC/GND changes
  private updateVoltages() {
    // Step 1: Find out how much voltage the VCC pin has
    const vcc = this.getPinVoltage('VCC') || this.getPinVoltage('5V');
    const gnd = this.getPinVoltage('GND');

    // Step 2: Check if the sensor is actually powered on
    // A real MQ-2 won't produce output if VCC < 3V
    const hasPower = (vcc - gnd) >= 3.0;

    // Step 3: Calculate the analog output voltage
    // gasLevel is stored as 0–1023 (matching Arduino analogRead range)
    // Scale it proportionally: if gasLevel=512 and vcc=5V → AO = 2.5V
    const analogVoltage = hasPower
      ? (this.state.gasLevel / 1023) * vcc
      : 0.0;  // If no power, output is dead (0V)

    // Step 4: Calculate the digital output voltage
    // Active-LOW: when gas EXCEEDS threshold → DO = 0V (alarm on)
    //             when gas BELOW threshold   → DO = VCC (alarm off)
    const digitalVoltage = hasPower
      ? (this.state.limitExceeded ? 0.0 : vcc)
      : 0.0;

    // Step 5: Push both voltages onto the output pins
    this.setPinVoltage('AO', analogVoltage);
    this.setPinVoltage('DO', digitalVoltage);
  }
}
```

1. **Why the power check matters:** In real electronics, an unpowered sensor will not produce any output. Students often forget to wire VCC and GND when they are learning. By checking whether VCC − GND ≥ 3V before calculating outputs, the simulation teaches this lesson naturally — the student wires up the gas cloud, nothing happens, they look at the circuit, realize VCC is missing, and learn from the mistake.
2. **Why gasLevel uses 0–1023:** The Arduino analogRead() function returns values from 0 to 1023 (10-bit resolution). If we store gas levels in the same range, the student's code works exactly the same way in simulation as it does on real hardware. No adjustment needed.
3. **Why the DO pin is Active-LOW:** On real hardware, comparator chips (like the LM393 used in the MQ-2 module board) drive the output LOW when the threshold is exceeded because they use "open collector" outputs — the transistor pulls the pin to ground when active. This is a standard electronics convention and our simulation matches it exactly.

```
// The onEvent function handles the gas cloud interaction from the UI
onEvent(event: any) {
    if (event.type === 'gas_level') {
        // Clamp the incoming level to 0–1023 range
        const level = Math.max(0, Math.min(1023, Math.round(event.value)));
        const threshold = this.state?.threshold ?? 300;
        const limitExceeded = level > threshold; // Is alarm active?

        // Update the state with new gas level and alarm status
        this.setState({
            gasLevel: level,
            limitExceeded
        });

        // Recalculate the output pin voltages with new gas level
        this.updateVoltages();
    }
}
```

What Happens Inside the State Object

The component tracks three pieces of information at all times:

Variable	Data Type	Range	What It Means
gasLevel	Integer	0 to 1023	Current simulated gas concentration. 0 = clean air, 1023 = maximum detectable gas
threshold	Integer	0 to 1023	The level at which the DO alarm pin goes LOW. Default is 300
limitExceeded	Boolean	true / false	Set to true when gasLevel exceeds the threshold — drives the DO alarm

How the Interactive UI Works

The MQ-2 sensor UI (ui.tsx) renders as a realistic PCB board with the characteristic metal mesh heat element in the center, four gold header pins at the bottom, and two small LED indicators on one corner.

1. **The Gas Cloud Interaction:** When a student clicks the sensor body during a running simulation, a gas cloud appears. This cloud is built from 5–6 translucent SVG circles of different sizes with Gaussian blur applied, stacked on top of each other to look like a realistic cloud of gas diffusing in air.

The cloud is draggable. As the student drags the cloud, the code continuously calculates the pixel distance between the cloud's center and the sensor's center:

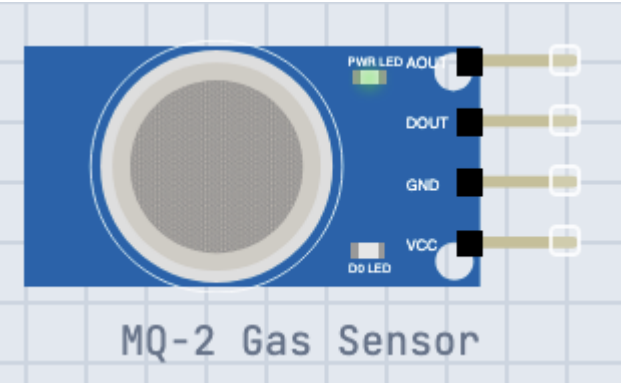
```
Distance < 30px  →  gasLevel = 1023 (maximum gas, sensor fully saturated)
Distance 30–250px →  gasLevel = floor((1 - (dist - 30) / 220) × 1023) (smooth linear interpolation)
Distance > 250px  →  gasLevel = 0 (no gas detected)
```

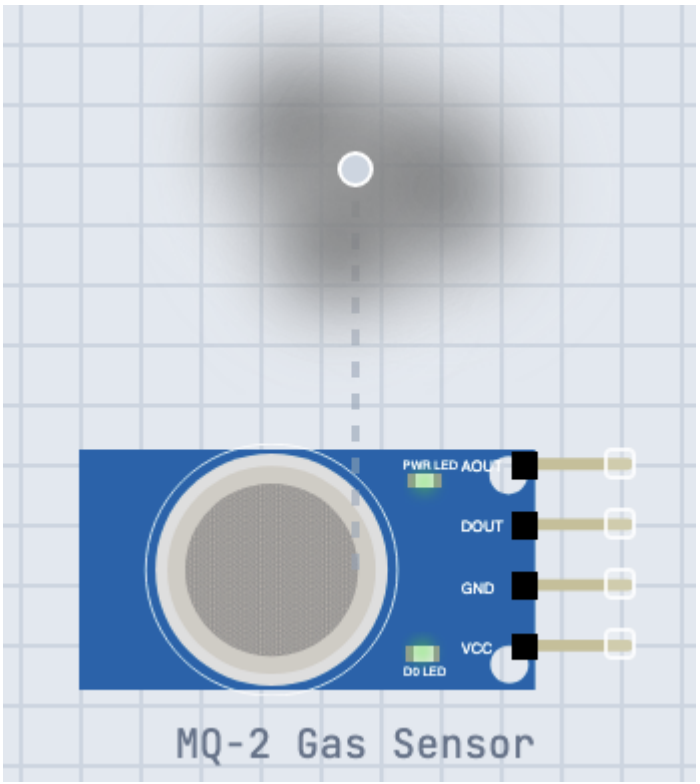
A dashed SVG line is drawn from the cloud to the sensor as a visual guide to show the student what is happening.

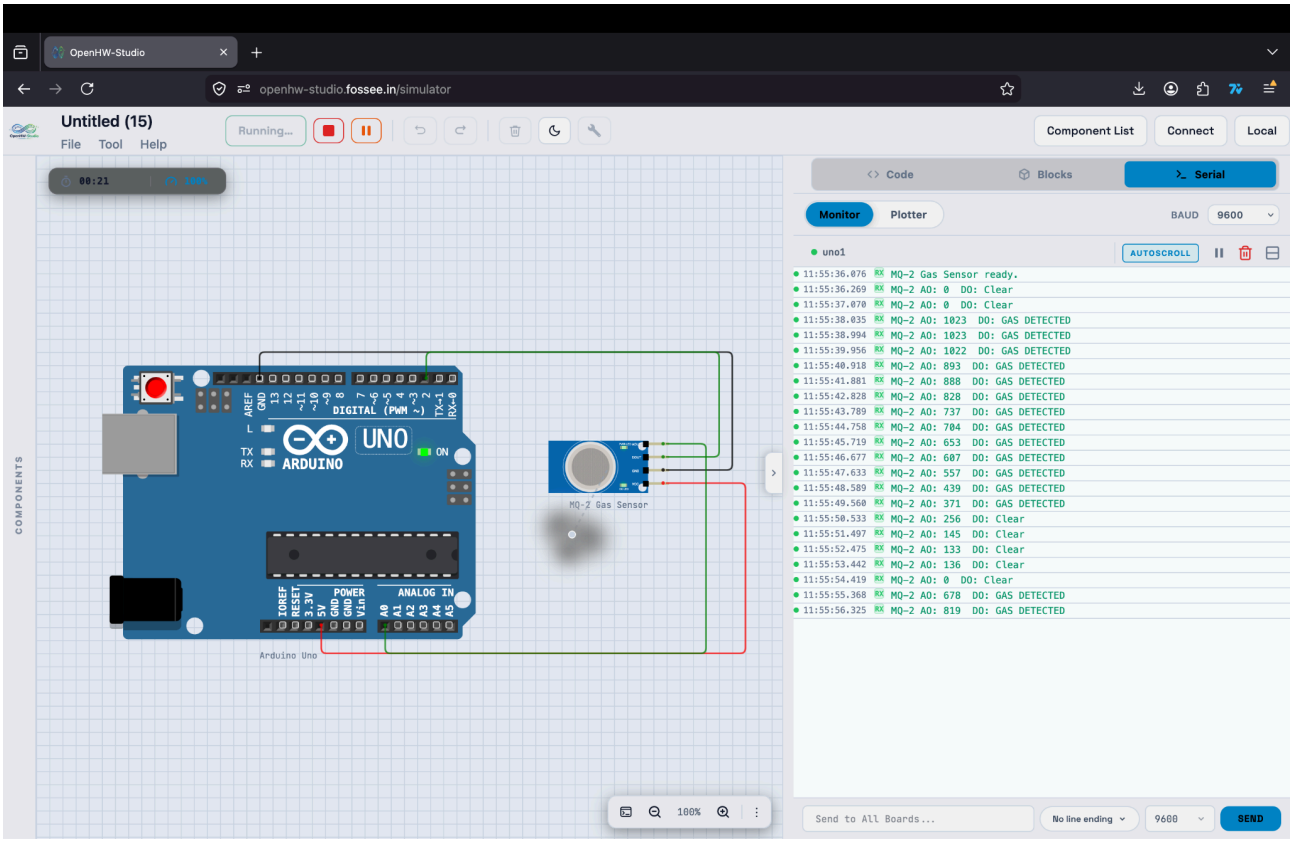
2. **The Threshold Context Menu:** Right-clicking the sensor during a simulation opens a context menu with a slider from 0 to 1023. Dragging this slider updates threshold in the sensor state. Students can observe in real time how changing the threshold from 300 to 800 changes where the DO alarm triggers.

LED Indicators:

- 🟢 **Green Power LED** — This is always ON whenever the simulation is running and the sensor has power. It is positioned on the top-left corner of the PCB.
- 🔴 **Red Alarm LED** — This matches the DO pin state. It turns ON (glows red with a box-shadow glow effect) when limitExceeded = true. It turns OFF when gas drops below the threshold.







Pin Reference Table

Pin Name	Direction	Voltage Range	Purpose
VCC	Input	5V	Powers the sensor.

Pin Name	Direction	Voltage Range	Purpose
			Below 3V, all outputs go to 0V.
GND	Input	0V	Ground reference.
AO	Output	0V to VCC	Analog output: proportional to gas concentration.
DO	Output	0V or VCC	Digital alarm: goes LOW (0V) when gas exceeds threshold.

1.2 DHT22 Temperature & Humidity Sensor

Code location: openhw-studio-emulator/src/components/DHT-22/

What is the DHT22 in Real Life?

The DHT22 (also known as AM2302) is a combined temperature and humidity sensor. Inside the small white plastic body there are two sensing elements:

1. A **polymer capacitor** for humidity — water molecules in the air are absorbed into the polymer film, changing its capacitance. The more humid the air, the higher the capacitance.
2. An **NTC thermistor** for temperature — its resistance drops as temperature increases.

An onboard microcontroller chip converts these physical readings into a digital number and sends it over a single data wire using precisely-timed HIGH and LOW pulses.

Why only one wire? Using one wire keeps the sensor cheap and simple to wire. The trade-off is that the timing protocol is rigid — every pulse must be exactly the right duration or the Arduino's library will reject the data as corrupted.

The DHT22 Communication Protocol — Explained Simply

Imagine you and a friend are communicating using a flashlight. You agree on this rule: a very short flash (0.026 seconds) means the number 0, and a longer flash (0.070 seconds) means the number 1. You can send any number this way — just flash out its binary digits.

The DHT22 works similarly, but using a single data wire instead of a flashlight:

Step 1: Arduino wakes the sensor The Arduino pulls the DATA line LOW (0V) for at least 1 millisecond. This is the "wake-up call." The DHT22 is normally asleep to save power and this pulse tells it to prepare a reading.

Step 2: Arduino releases the line The Arduino lets the DATA line float back to HIGH (5V via a pull-up resistor). Now it waits for the DHT22 to respond.

Step 3: DHT22 acknowledges The DHT22 pulls DATA LOW for 80 microseconds, then HIGH for 80 microseconds. This is an "I'm ready" handshake.

Step 4: DHT22 sends 40 data bits Each bit is sent in two stages:

- First, a 50-microsecond LOW pulse (this marks the start of each bit).
- Then a HIGH pulse. If the HIGH pulse lasts **26 microseconds**, the bit is 0. If it lasts **70 microseconds**, the bit is 1.

Why those specific times? Because the DHT22's internal chip uses those timings, and the DHT Arduino library was written expecting exactly those timings. We had to match them precisely.

Step 5: The 40 bits decoded The 40 bits are grouped into five bytes:

```
Byte 0 (bits 1–8):   Upper 8 bits of humidity  → e.g. 0x01
Byte 1 (bits 9–16):  Lower 8 bits of humidity  → e.g. 0xF5   (0x01F5 = 501 → 50.1% RH)
Byte 2 (bits 17–24): Upper 8 bits of temperature
Byte 3 (bits 25–32): Lower 8 bits of temperature
Byte 4 (bits 33–40): Checksum = (Byte0 + Byte1 + Byte2 + Byte3) & 0xFF
```

If the checksum doesn't match, the Arduino library discards the reading and reports an error.

Why This Was Technically Difficult to Simulate

The DHT22 protocol depends on precise microsecond timing. The browser runs JavaScript, which does not have microsecond-accurate timers. setTimeout(fn, 0.026) does not work — browsers only guarantee millisecond precision at best.

The key insight was: **use the CPU emulator's clock cycle counter instead of real time.**

The simulated ATmega328P runs at 16 MHz — meaning it performs 16 million clock cycles per second. Therefore:

- 1 microsecond = 16 clock cycles
- 80 microseconds = 1,280 clock cycles
- 50 microseconds = 800 clock cycles
- 70 microseconds (bit-1 HIGH) = 1,120 clock cycles
- 26 microseconds (bit-0 HIGH) = 416 clock cycles

The emulator provides an API called cpu.addClockEvent(callback, delayCycles) which schedules a function to run after exactly delayCycles have been processed. This is the key function that makes the DHT22 simulation work correctly.

Walking Through the Code

State machine: The DHT22 logic uses a state machine to track where in the communication process it is at any given moment:

```
IDLE → (Arduino pulls LOW ≥ 1ms) → WAKE_WAIT
WAKE_WAIT → (Arduino releases HIGH) → ACKING
ACKING → (Sends 80µs LOW, 80µs HIGH ACK) → SENDING
```

```
SENDING → (Sends 40 bits one by one) → IDLE
```

Here is the pin-watching code:

```
// From: DHT-22/logic.ts

onPinStateChange(pin: string, isHigh: boolean, cycles: number) {
    // Only react to changes on the DATA pin
    if (pin !== 'DATA') return;
    // Ignore changes caused by us driving the bus (re-entrant guard)
    if (this._drivingBus) return;

    if (!isHigh && this.protocolState === 'IDLE') {
        // Arduino just pulled DATA LOW — this is the start of the wake-up pulse
        this.protocolState = 'WAKE_WAIT';
        this.wakeStartCycles = cycles;  // Remember when the pulse started

    } else if (isHigh && this.protocolState === 'WAKE_WAIT') {
        // Arduino released the line back to HIGH
        // Calculate how long it was LOW (in microseconds)
        const wakeUs = (cycles - this.wakeStartCycles) / 16;

        if (wakeUs >= 100) {
            // The wake-up pulse was long enough — send the acknowledgement
            this.startAckSequence();
        } else {
            // Pulse too short — probably noise, ignore it
            this.protocolState = 'IDLE';
        }
    }
}
```

Building the 40 data bits:

```
private prepareDataBits() {
    // Convert temperature and humidity to the DHT22 wire format
    const h    = Math.round(this.humidity * 10);       // e.g. 56.3% → 563
    const tAbs = Math.round(Math.abs(this.temperature) * 10); // e.g. 23.5°C → 235
    const sign = this.temperature < 0 ? 0x80 : 0x00;  // Bit 15 is the sign bit

    // Pack into 4 bytes
    const b0 = (h >> 8) & 0xFF;
    const b1 =  h       & 0xFF;
    const b2 = ((tAbs >> 8) & 0x7F) | sign;
    const b3 =  tAbs        & 0xFF;
    const b4 = (b0 + b1 + b2 + b3) & 0xFF; // Checksum byte

    // Expand 5 bytes → 40 bits, MSB first
    this.dataBits = [];
    for (const byte of [b0, b1, b2, b3, b4]) {
        for (let i = 7; i >= 0; i--) {
            this.dataBits.push(!!((byte >> i) & 1)); // Extract each bit
        }
    }
    this.bitIndex = 0;
}
```

Sending the bits one at a time:

```
private sendNextBit() {
    // Check if we've sent all 40 bits
    if (this.bitIndex >= 40) {
```

```
      // End-of-frame: one more 50µs LOW, then release the bus
      this.driveData(0);
      this.schedule(() => this.releaseData(), 50 * 16);
      return;
    }

    const bit = this.dataBits[this.bitIndex++]; // Get the next bit to send

    // Every bit starts with a 50µs LOW preamble
    this.driveData(0);
    this.schedule(() => {
      // After 50µs, drive HIGH for either 26µs (bit=0) or 70µs (bit=1)
      const highUs = bit ? 70 : 26;
      this.driveData(5.0);

      // After the HIGH pulse, send the next bit
      this.schedule(() => this.sendNextBit(), highUs * 16);
    }, 50 * 16);
  }
```

The Watchdog Timer: If the protocol gets stuck (e.g. the simulation is paused mid-transmission), the update() function runs every few milliseconds and checks whether the sensor has been in the ACKING or SENDING state for more than 200ms (equivalent to 3.2 million CPU cycles). If so, it forces the state back to IDLE and releases the bus:

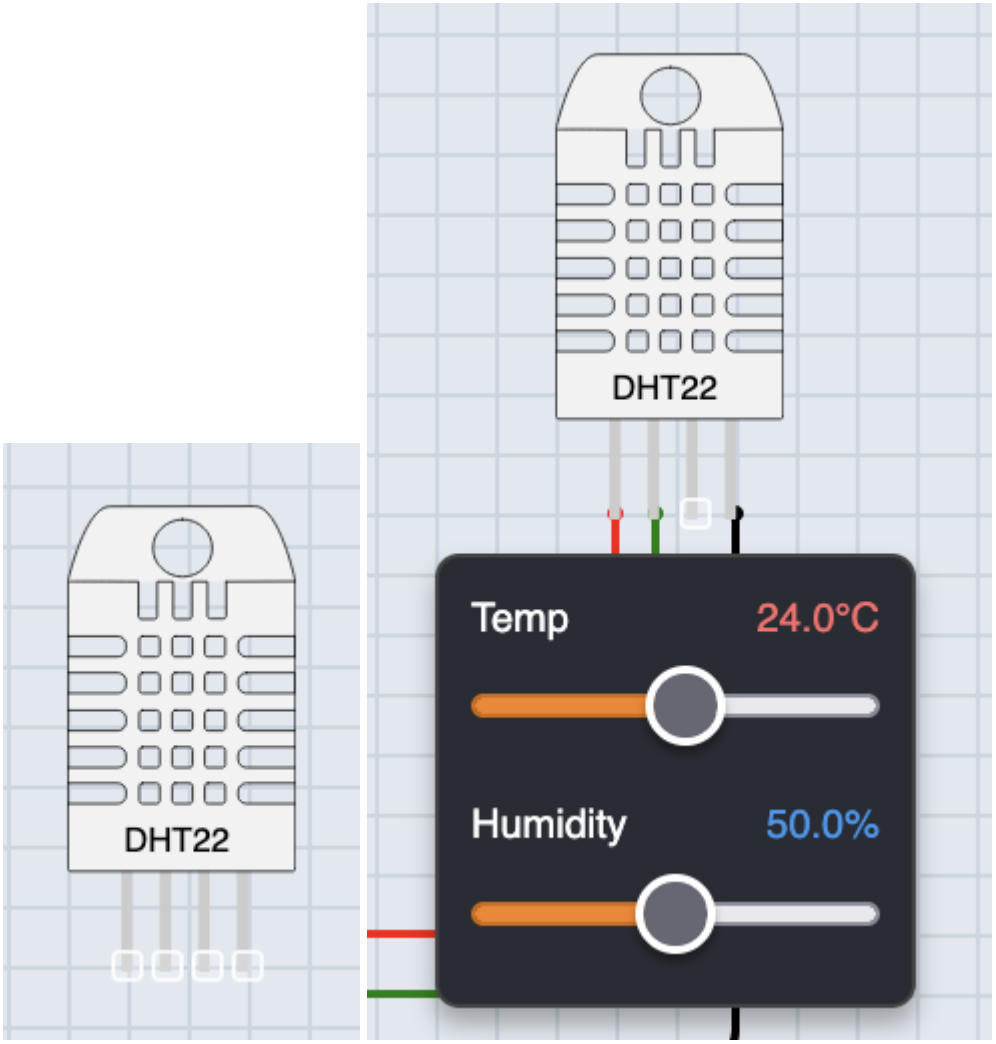
```
update(cpuCycles: number) {
    super.update(cpuCycles);
    if ((this.protocolState === 'ACKING' || this.protocolState === 'SENDING')
        && this._txStartCycle > 0
        && (cpuCycles - this._txStartCycle) > 3_200_000) {

        console.warn('[DHT22] Watchdog: stuck too long — resetting');
        this._drivingBus = false;
        this.protocolState = 'IDLE';
        this._txStartCycle = 0;
        this.setPinVoltage('DATA', 5.0); // Release the bus to HIGH
    }
}
```

How the Interactive UI Works

The DHT22 renders as a white plastic block with a grid pattern on one face. When the simulation starts:

1. A **control panel** appears below the sensor with two interactive sliders.
2. The **red temperature slider** ranges from −40°C to +80°C in steps of 0.1°C.
3. The **blue humidity slider** ranges from 0% to 100% in steps of 0.1%.
4. Whenever the student moves a slider, a SET_ATTR event fires into the logic, which stores the new value immediately. The next time the Arduino library requests a reading, the simulated sensor returns the updated values.

This is a very important design choice: we do not need to re-send the full 40-bit frame every time the slider moves. We just store the new values and send them on the *next* request cycle from the Arduino. This keeps the simulation smooth and avoids unnecessary processing.



Pin Reference Table

Pin Name	Direction	Voltage Range	Purpose
VCC	Input	3.3V or 5V	Power supply
DATA	Bidirectional	0V or 5V	Single-wire timing protocol for temperature and humidity readings
NC	—	—	Not connected (physical pin 3 on the real component, ignore this)
GND	Input	0V	Ground reference

1.3 Raindrop Module

Code location: openhw-studio-emulator/src/components/Raindrop-module/

1.3.1 What is the Raindrop Module in Real Life?

The Raindrop Module is a small green circuit board that works as a water/rain detector. The board contains an **LM393** dual comparator chip. A comparator chip is very simple: it compares two voltages and produces a HIGH or LOW output depending on which one is bigger.

The module reads a voltage from the **Raindrop Pad** (a separate nickel-coated sensing plate). When the pad is dry, the resistance between its tracks is very high, and the output voltage is close to 5V. When water bridges the tracks, the resistance drops and the voltage drops toward 0V.

The module processes this voltage and produces two signals for the Arduino:

- **AO (Analog Output):** The raw voltage from the pad (0V when soaking wet, up to 5V when completely dry).
- **DO (Digital Output):** A simple alarm pin — goes LOW when the pad's voltage drops below a threshold set by a small potentiometer on the board.

1.3.2 The Design Challenge — How Do Two Components Share Data?

In a real physical circuit, the Raindrop Module and the Raindrop Pad are connected by wires. In our simulator, each component is a **separate JavaScript class instance**. These instances run independently and there are no "wires" connecting them at the data level.

The question was: how do we make the Module class "know" what the Pad class is currently measuring?

One approach would be to use electrical pin-to-pin simulation — wire a virtual VCC from the pad's output pin to the module's input pin, and let the electrical simulation propagate the voltage. But this adds overhead because every pin voltage change triggers a full circuit recalculation.

Instead, we used a simpler and faster approach: a **shared singleton object** — a single JavaScript object that both components can read from and write to.

```
// From: raindrop-shared-state.ts
// This is a singleton — only one copy exists in memory, shared by both components
export const raindropSharedState = {
  padVoltage: 5.0,      // Voltage output by the pad (5V = dry, 0V = soaking wet)
  rainLevel: 0,         // 0 to 1023 — mirrors ADC value the Arduino would read
  rainDetected: false,  // true when rain is above the alarm threshold
  threshold: 300        // The comparison threshold (0–1023)
};
```

Think of this as a **shared whiteboard**. The Pad class writes its current wetness data on the whiteboard. The Module class checks the whiteboard and updates its output pins accordingly.

1.3.3 How the Raindrop Module Logic Works

The module's update() function is called on every simulation tick. It reads from the shared state and drives its output pins:

```
// From: Raindrop-module/logic.ts

export class RaindropModuleLogic extends BaseComponent {
  private lastUpdate = 0; // Timestamp of the last UI state update

  update() {
    // Read the latest data from the shared whiteboard
    const { padVoltage, rainLevel, rainDetected } = raindropSharedState;

    // Drive the analog output pin with the raw pad voltage
    // (This is what the Arduino reads with analogRead(AO_PIN))
    this.setPinVoltage('AO', padVoltage);

    // Drive the digital output pin (Active-LOW alarm logic)
    // rainDetected = true → DO = 0V (alarm on)
    // rainDetected = false → DO = 5V (alarm off)
    this.setPinVoltage('DO', rainDetected ? 0.0 : 5.0);

    // Always provide VCC and GND
    this.setPinVoltage('VCC', 5.0);
    this.setPinVoltage('GND', 0.0);

    // Throttle the UI state update — only update React state every 100ms
    // This prevents hundreds of re-renders per second which would lag the browser
    const now = Date.now();
    if (now - this.lastUpdate > 100) {
      this.setState({ rainLevel, rainDetected, padVoltage });
      this.lastUpdate = now;
    }
  }
}
```

Why throttle the UI update to 100ms? The simulation update loop can fire 100–500 times per second depending on CPU speed. If we called setState() on every single tick, React would try to re-render the component that many times per second. React batches some updates, but 200–500 re-renders/second would still make the browser sluggish and drain CPU. Calling setState() only once every 100ms gives us 10 UI updates per second — smooth enough for the student to see the live values changing without any performance problems.

1.3.4 The Threshold Control Event

The student can also change the rain detection threshold using a slider in the component's context menu:

```
onEvent(event: any) {
  if (event?.type === 'threshold_update') {
    // Clamp and round the incoming threshold value to valid range
    const threshold = Math.max(0, Math.min(1023, Math.round(Number(event.value) || 300)));

    // Write the new threshold to the shared state so the Pad can use it too
    raindropSharedState.threshold = threshold;
```

```
    // Also update this component's local state for UI display
    this.setState({ threshold });
  }
}
```

1.3.5 How the Interactive UI Works

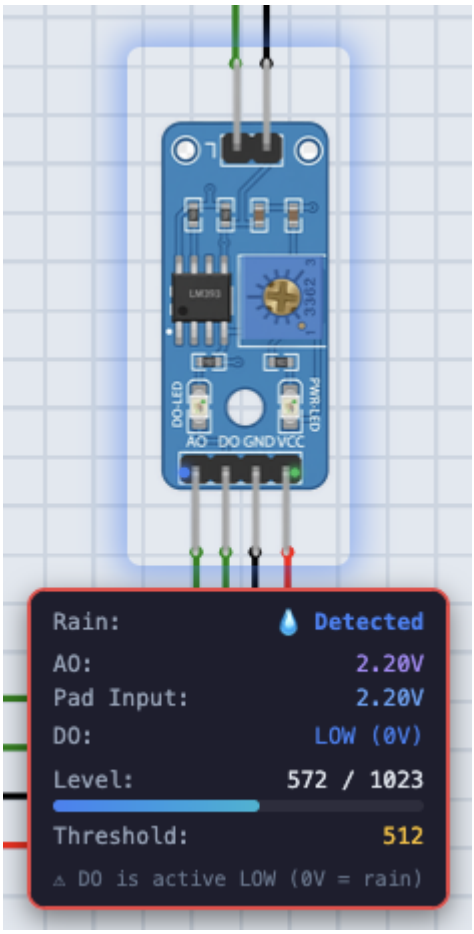
The Raindrop Module renders using a PNG photograph of the actual green PCB board overlaid with SVG elements.

Live Status Panel: Below the PCB image, a real-time status card is rendered showing:

- A large text badge: 💧 **RAIN DETECTED** (blue) or ☀️ **DRY** (yellow).
- A table with four live values: AO voltage, pad input voltage, DO state, and current threshold.
- A horizontal progress bar showing the rain level from 0 to 1023.

LED Indicators on the PCB:

- 🟢 **Green PWR LED** — Lights up whenever the simulation is running (mimics the power indicator on the real board).
- 🔵 **Blue DO LED** — Lights up when rainDetected = true (mimics the signal indicator on the real board that turns on when rain is sensed).



1.3.6 Pin Reference Table

Pin Name	Direction	Voltage	Purpose
VCC	Input	5V	Power supply
GND	Input	0V	Ground
AO	Output	0V–5V	Analog output: pad voltage (5V = dry, near 0V = wet)
DO	Output	0V or 5V	Digital alarm: LOW (0V) when rain detected

---

1.4 Raindrop Pad

Code location: openhw-studio-emulator/src/components/Raindrop-pad/

1.4.1 What is the Raindrop Pad in Real Life?

The Raindrop Pad is simply a flat PCB with interleaved nickel-plated copper tracks running across it. It looks like a small comb pattern on the board. When completely dry, there is no path for electricity to flow between the tracks — so the resistance is very high and the output voltage is close to the supply voltage (5V).

When raindrops land on the pad, water bridges the gap between the tracks. Water conducts electricity (especially tap water, which contains dissolved minerals). This creates a conducting path, which reduces the resistance and causes the output voltage to drop toward 0V.

The important thing to understand is the **inverted relationship**: wetter = lower voltage.

- Completely dry → Output ≈ 5.0V
- Light mist → Output ≈ 3.5V
- Heavy rain → Output ≈ 0.5V
- Submerged → Output ≈ 0.0V

Our simulation formula captures this:

```
padVoltage = (1 - rainLevel / 1023) × 5.0
```

If rainLevel = 700, then padVoltage = (1 - 700/1023) × 5 = (0.316) × 5 = 1.58V.

1.4.2 How the Pad Updates the Shared State

```
// From: Raindrop-pad/logic.ts (simplified)

onEvent(event: any) {
  if (event.type === 'rain_level') {
    const rainLevel = Math.max(0, Math.min(1023, Math.round(event.value)));
    const threshold = raindropSharedState.threshold;

    // Calculate the voltage using the inverted formula
    const padVoltage = (1 - rainLevel / 1023) * 5.0;
    const rainDetected = rainLevel > threshold;

    // Write to the shared whiteboard so the Module can pick it up
    raindropSharedState.padVoltage  = padVoltage;
    raindropSharedState.rainLevel   = rainLevel;
    raindropSharedState.rainDetected = rainDetected;

    // Also update the Pad's own UI state
    this.setState({ rainLevel, padVoltage });
    this.setPinVoltage('AOUT', padVoltage);
  }
}
```

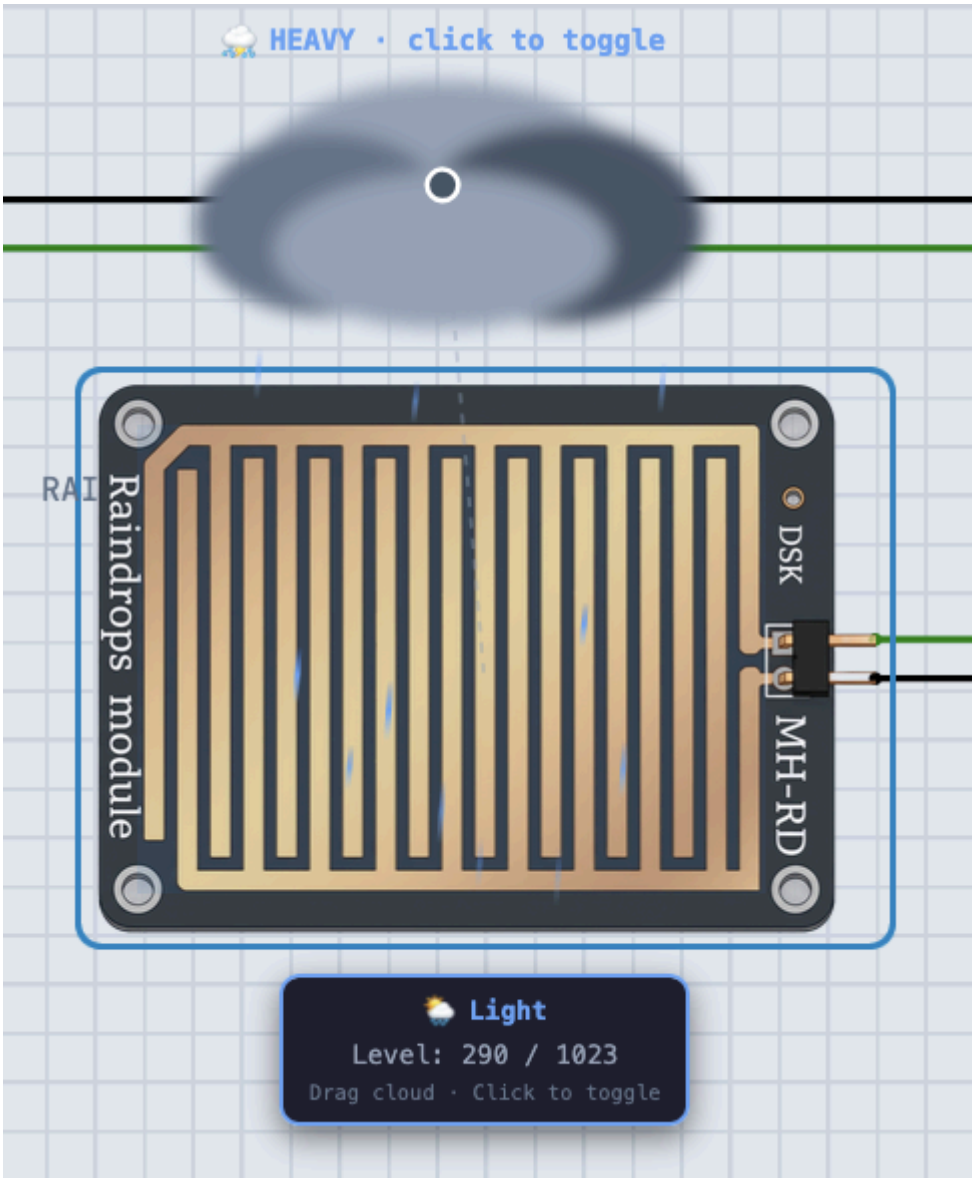
1.4.3 How the Interactive UI Works

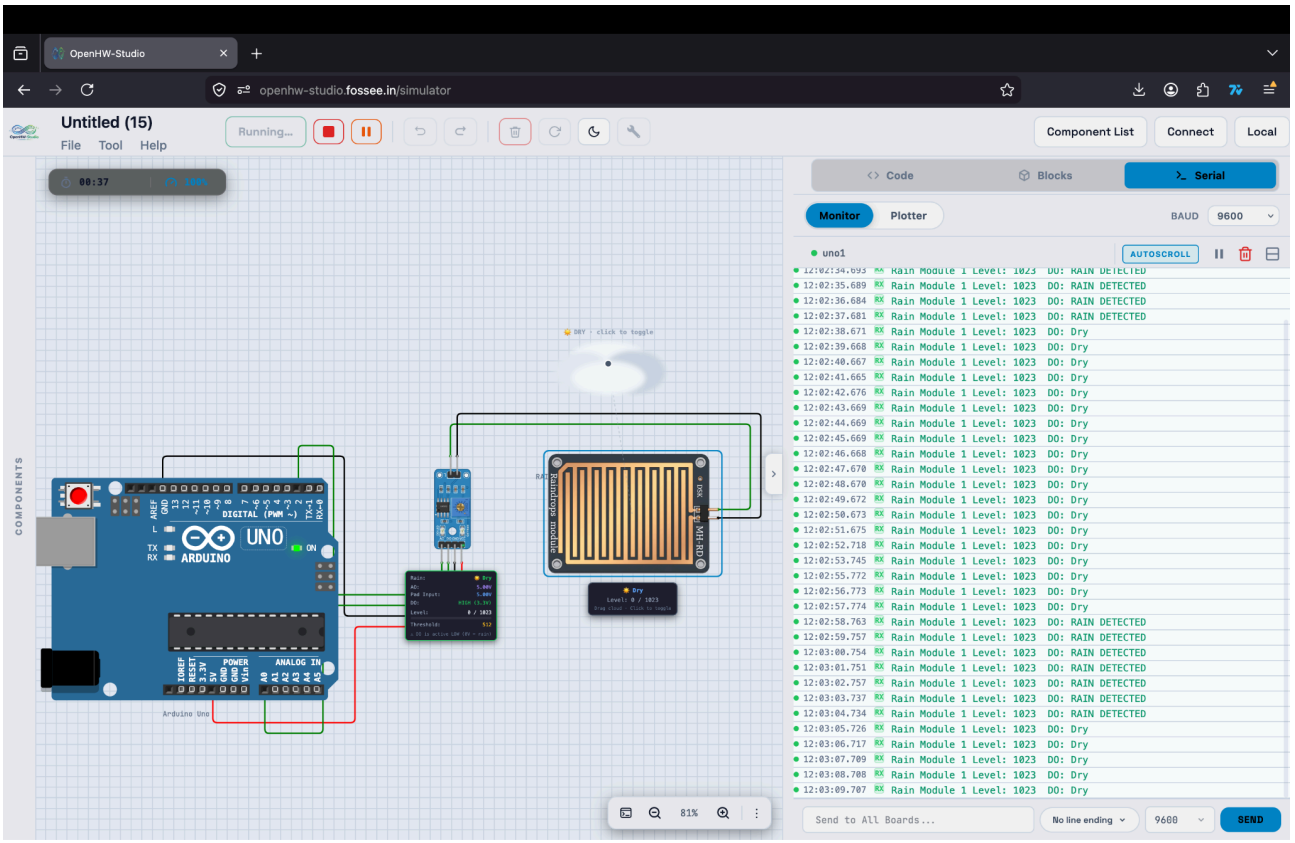
The Pad renders as a PCB image with a blue translucent water-fill overlay. The opacity of this blue overlay scales with the rain level — you can literally watch the board "fill up with water" as the rain level rises.

The Draggable Rain Cloud: Clicking the Pad during a simulation makes a rain cloud icon appear above the component.

- The cloud can be dragged anywhere on the canvas.
- The proximity of the cloud to the pad's center determines the rain level:
  - Very close (< 30px) → rainLevel = 1023 (maximum wetness)
  - Far away (> 250px) → rainLevel = 0 (completely dry)
  - In between → Linear interpolation between these values
- Clicking the cloud toggles between **Storm Mode** (heavy rain — animated falling lines below the cloud icon) and **Sunny Mode** (rain immediately stops, level resets to 0).

This interactive model is satisfying for students because they can physically "move" the cloud and watch the voltages on the Module change in real time — making the concept of rain detection very tangible.





1.4.4 Pin Reference Table

Pin Name	Direction	Voltage	Purpose
AOUT	Output	0V–5V	Analog voltage output — 5V dry, drops to ~0V when wet
GND	Input	0V	Ground connection

1.5 PIR Motion Sensor

Code location: openhw-studio-emulator/src/components/PIR-Motion-Sensor/

1.5.1 What is a PIR Sensor in Real Life?

PIR stands for **Passive Infrared**. Every object above absolute zero temperature emits infrared (heat) radiation. Humans, animals, and even warm engines radiate a lot of infrared energy compared to their surroundings.

A PIR sensor contains a **pyroelectric element** — a crystal that generates a tiny electrical charge when its temperature changes. The sensor uses a **Fresnel lens** (a thin plastic lens with many concentric ridges) to focus infrared radiation from a wide area onto this crystal.

When a warm object moves across the sensor's field of view, the amount of infrared energy hitting the crystal changes. This change generates a pulse of electrical charge, which an amplifier circuit converts into a clean HIGH signal on the output pin.

The sensor is called "passive" because it does not emit anything — it just detects existing infrared radiation. This makes it very cheap and energy-efficient compared to active sensors like ultrasonic or laser distance sensors.

Common uses: Automatic lights in hallways, security alarms, people counters, smart locks, automatic faucets.

Key behavior: The real HC-SR501 PIR module has two important settings:

- **Hold Time** (Tx): How long the output pin stays HIGH after motion is detected. Adjustable from about 0.5 seconds to several minutes.
- **Sensitivity** (Sx): How far away the sensor can detect motion. Adjustable from 3 to 7 meters.

Our simulation implements the hold time feature, since that directly affects how student code behaves.

1.5.2 Why We Needed to Support Three Trigger Modes

A student might want to test three different scenarios:

1. A person walks past the sensor once → output is HIGH for a few seconds then LOW (a brief trigger).
2. A person stands in front of the sensor for a long time → output should stay HIGH until they leave.
3. The student wants to test what happens at the exact moment of detection, then again when it stops.

A single "click to toggle" interaction wouldn't cover all three cases cleanly. So we designed three separate event types:

```
// From: PIR-Motion-Sensor/logic.ts

onEvent(event: string) {

  // --- Mode 1: Single-shot trigger (mimics a brief motion event) ---
  if (event === 'motion') {
    const delay = this.attrs.delay ? parseInt(this.attrs.delay) : 500; // default 500ms
    this.setState({ motion: true });
    this.setPinVoltage('OUT', 3.3);  // Output goes HIGH (real module uses 3.3V)

    // Cancel any existing timer (prevents overlapping timers)
    if (this.motionTimeout) clearTimeout(this.motionTimeout);

    // After the configured delay, automatically return output to LOW
    this.motionTimeout = setTimeout(() => {
      this.setState({ motion: false });
      this.setPinVoltage('OUT', 0);
    }, delay);
  }
```

```
  // --- Mode 2: Continuous hold — output HIGH until explicitly stopped ---
  else if (event === 'motion_start') {
    this.setState({ motion: true });
    this.setPinVoltage('OUT', 3.3);
    if (this.motionTimeout) clearTimeout(this.motionTimeout); // Cancel any auto-reset
  }

  // --- Mode 3: Manually stop a continuous hold ---
  else if (event === 'motion_stop') {
    this.setState({ motion: false });
    this.setPinVoltage('OUT', 0);
    if (this.motionTimeout) clearTimeout(this.motionTimeout);
    this.motionTimeout = null;
  }
}
```

Why is the output 3.3V, not 5V? On the real HC-SR501 module, the sensor element inside uses a 3.3V logic level even when the module itself is powered from 5V. The output pin is driven by a 3.3V regulator. Most Arduino inputs treat anything above ~2.5V as HIGH, so the code works correctly. But outputting exactly 3.3V makes our simulation more physically accurate.

1.5.3 How the Interactive UI Works

The PIR sensor renders using the standard <wokwi-pir-motion-sensor> SVG web component, scaled to fit the canvas grid.

Clicking the Sensor Body → The Detection Cone: When the student clicks the sensor, a large fan-shaped SVG path appears — this represents the sensor's detection field of view. In the real world, PIR sensors typically detect motion in a roughly 120° cone extending about 5–7 meters.

The cone is drawn mathematically using SVG arc commands:

```
Fan angle: −135° to −45° (90° spread)
Inner radius: 40px
Outer radius: 280px
```

The cone colour changes based on the current motion state:

- 🟢 Semi-transparent green → No motion detected (idle/ready state)
- 🔴 Bright red with higher opacity → Motion detected

The Draggable Target Dot: Inside the cone, a teal circular handle appears. This represents the person or object being detected. The student drags it:

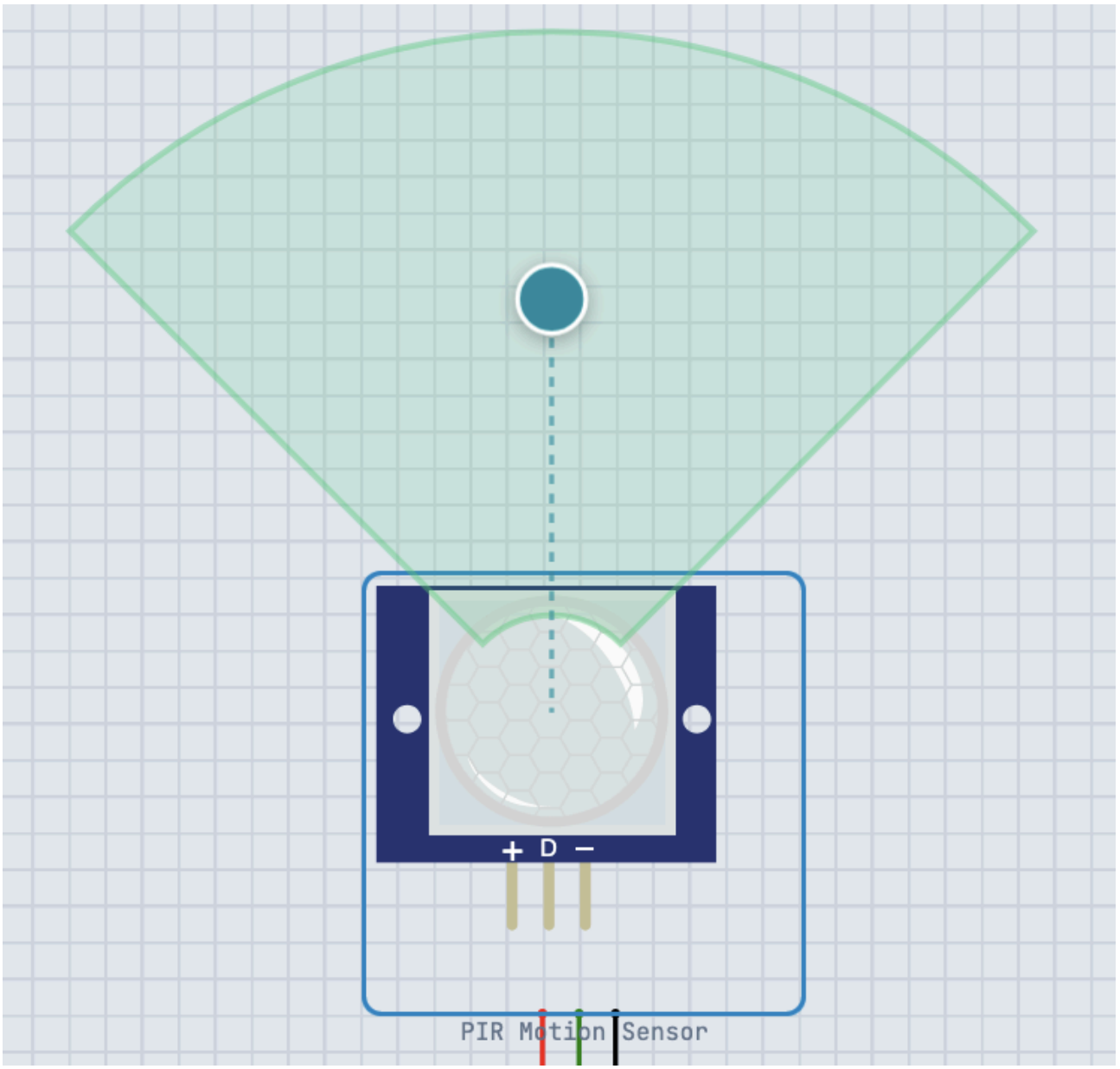
- **Inside the cone** → The code checks both distance and angle to confirm the dot is within the sensor's detection zone. If the dot is inside AND has moved at least 0.2 pixels since the last check, it fires motion_start to begin a motion hold. A 150ms watchdog timer resets the motion if the student holds the dot still.
- **Outside the cone** → motion_stop fires immediately and the output returns to 0V.

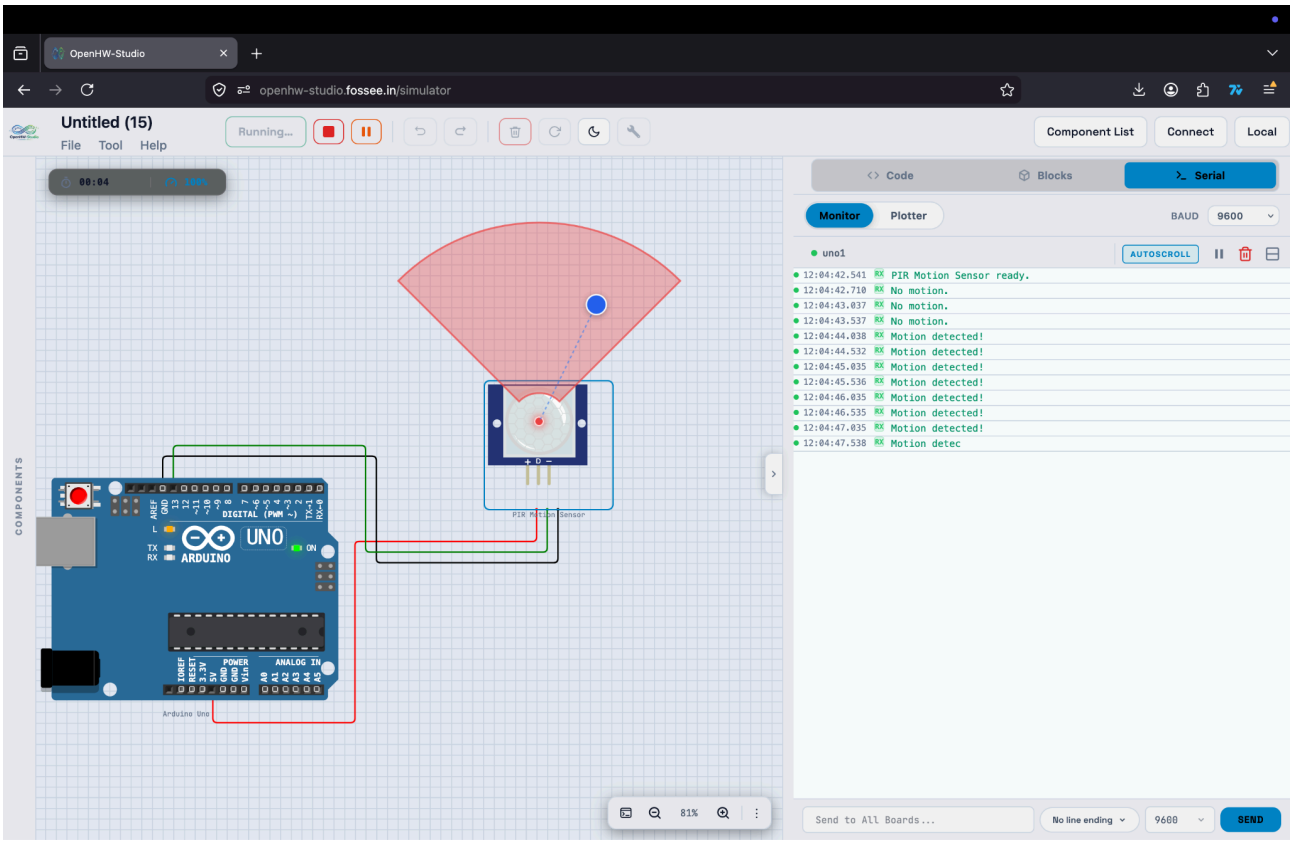
```
// From: PIR-Motion-Sensor/ui.tsx
const checkMotion = (x: number, y: number) => {
  const dx = x - SENSOR_CX;
  const dy = y - SENSOR_CY;
  const dist = Math.sqrt(dx * dx + dy * dy);
```

```
    const angle = Math.atan2(dy, dx) * 180 / Math.PI;

    // Is this position inside the sensor's cone geometry?
    const isInside = dist >= 40 && dist <= 280 && angle >= -135 && angle <= -45;

    // Only fire if the dot has actually moved (debounce tiny jitter)
    const hasMoved = Math.abs(x - lastPos.x) > 0.2 || Math.abs(y - lastPos.y) > 0.2;

    if (isInside && hasMoved) {
      triggerMotion(true);  // fires motion_start

      // Watchdog: if the dot stops moving for 150ms, automatically stop the motion
      if (stopTimer) clearTimeout(stopTimer);
      stopTimer = setTimeout(() => triggerMotion(false), 150);

    } else if (!isInside) {
      triggerMotion(false); // fires motion_stop
    }
};
```

The Red Indicator Light: When motion = true, a 14×14 pixel red circle with a glowing box-shadow effect appears on the sensor body, centered over the dome element. This imitates the small red LED that many real PIR modules have to show they are actively detecting motion.

The Context Menu: Right-clicking the sensor opens a small settings panel with a "Hold Time (ms)" number input. The value can be set from 100ms to 10,000ms. This updates attrs.delay, which the logic reads every time a single-shot motion event fires.


+ D −
PIR Motion Sensor



1.5.4 Pin Reference Table

Pin Name	Direction	Voltage	Purpose
VCC	Input	5V	Power supply
GND	Input	0V	Ground
OUT	Output	0V or 3.3V	HIGH (3.3V) when motion is detected, LOW (0V) otherwise

---

1.6 HC-SR04 Ultrasonic Sensor

Code location: openhw-studio-emulator/src/components/openhw-hc-sr04/

1.6.1 What is the HC-SR04 in Real Life?

The HC-SR04 is a distance measurement sensor. It works like a bat's echolocation:

1. It sends out a burst of ultrasonic sound (40,000 Hz — way above what humans can hear).
2. This burst bounces off any nearby object and returns to the sensor.
3. The sensor measures how long the echo took to return.

4. The time is converted to distance using the speed of sound.

The speed of sound in air at room temperature is approximately 343 meters per second, which equals 0.0343 cm per microsecond. Since the sound travels to the object and back (a round trip), we divide by 2:

```
Distance (cm) = Time (µs) × 0.0343 / 2
              = Time (µs) / 58.3
              ≈ Time (µs) / 58
```

This is the well-known "divide by 58" formula that appears in every HC-SR04 tutorial.

Common uses: Robots that avoid obstacles, smart parking sensors, water level indicators, dustbin sensors that alert when they are full, hand sanitizer dispensers that activate when you bring your hand close.

1.6.2 Understanding the Interface Pins

The HC-SR04 uses four pins:

- **VCC** — Power (5V)
- **GND** — Ground
- **TRIG** — Trigger pin. The Arduino sends a 10µs HIGH pulse here to start a measurement.
- **ECHO** — Echo pin. The sensor drives this HIGH for a duration equal to the round-trip time. The Arduino reads this duration with pulseIn(ECHO, HIGH).

The interaction is like pressing a stopwatch button:

1. Arduino → HIGH for 10µs on TRIG → "Start the sound burst!"
2. HC-SR04 → internally sends 8 pulses of 40kHz ultrasound
3. HC-SR04 → drives ECHO HIGH → "Stopwatch started"
4. Echo returns → HC-SR04 → drives ECHO LOW → "Stopwatch stopped"
5. Arduino reads how long ECHO was HIGH → converts to distance

1.6.3 How the Code Implements This

We used PulseProtocol — a base class in the emulator that automatically detects the duration of HIGH and LOW pulses on any pin. This means we don't have to manually write the cycle-counting logic; PulseProtocol handles that and calls onPulseReceived() with the measured duration.

```
// From: openhw-hc-sr04/logic.ts

export class HCSR04Logic extends PulseProtocol {
  private isEchoing = false;

  constructor(id: string, manifest: any) {
    super(id, manifest);
    // Load the distance from the component's attributes
    // (student or teacher can pre-set the distance in the manifest)
    this.state = { distance: parseFloat(this.attrs.distance || '100') };
```

```
  }

  // Called automatically by PulseProtocol when it detects a complete pulse on any pin
  onPulseReceived(pinId: string, isHighPulse: boolean, durationUs: number): void {
    // HC-SR04 only responds to HIGH pulses on the TRIG pin
    // Standard is 10µs, but we accept ≥8µs to tolerate AVR clock jitter
    if (pinId === 'TRIG' && isHighPulse && durationUs >= 8) {
      this.startEcho();
    }
  }

  private startEcho() {
    if (this.isEchoing) return; // Ignore if already processing an echo (prevents double-trigger)

    const distance = parseFloat(this.attrs?.distance || '100');
    const echoDurationUs = distance * 58; // Convert distance back to echo time

    this.isEchoing = true;
    const cpu = this._simCpu;

    // CRITICAL: Wait 200µs before starting the echo
    // Without this delay, the ECHO pin goes HIGH before Arduino's pulseIn() is ready
    // to listen, causing it to miss the pulse entirely.
    // This 200µs simulates the real sensor's 8-pulse 40kHz burst transmission time.
    cpu.addClockEvent(() => {

      this.driveEcho(5.0); // Drive ECHO pin HIGH → start of measurement

      // After the echo duration, pull ECHO back LOW
      cpu.addClockEvent(() => {
        this.driveEcho(0.0); // Drive ECHO pin LOW → end of measurement
        this.isEchoing = false; // Allow next trigger
      }, echoDurationUs * 16); // Convert µs → clock cycles (16 cycles per µs at 16MHz)

    }, 200 * 16); // 200µs initial delay in clock cycles
  }

  // Called from context menu when student changes the distance slider
  onEvent(event: any) {
    if (event.type === 'SET_ATTR') {
      this.attrs[event.key] = String(event.value);
      if (event.key === 'distance') {
        this.state.distance = String(event.value);
      }
      this.stateChanged = true;
    }
  }
}
```

Why 200µs initial delay is absolutely critical: When the Arduino calls pulseIn(ECHO, HIGH), it enters a tight loop checking the ECHO pin millions of times per second. But there is a small delay between when pulseIn() is called in code and when the CPU actually reaches the pin-checking loop. If our simulated sensor fires the ECHO pin HIGH immediately after the TRIG pulse, the echo can start and end before pulseIn() even begins listening — and the function returns 0, causing the distance calculation to return an absurd value.

The 200µs delay gives the Arduino CPU enough execution time to reach pulseIn()'s checking loop before the echo starts. This is a simulation timing concern, not a real-hardware concern — real hardware timing works out automatically because the HC-SR04's burst transmission itself takes

about 200µs.

1.6.4 How the Interactive UI Works

The HC-SR04 renders using the <wokwi-hc-sr04> web component which shows two large round silver transducer cylinders (the speaker and the microphone).

Clicking the Sensor → The Sonar Cone: Clicking the sensor opens a large blue-green fan-shaped sonar cone that represents the sensor's detection area:

- **Cone spread:** ±51° (102° total)
- **Range shown:** 2cm to 400cm

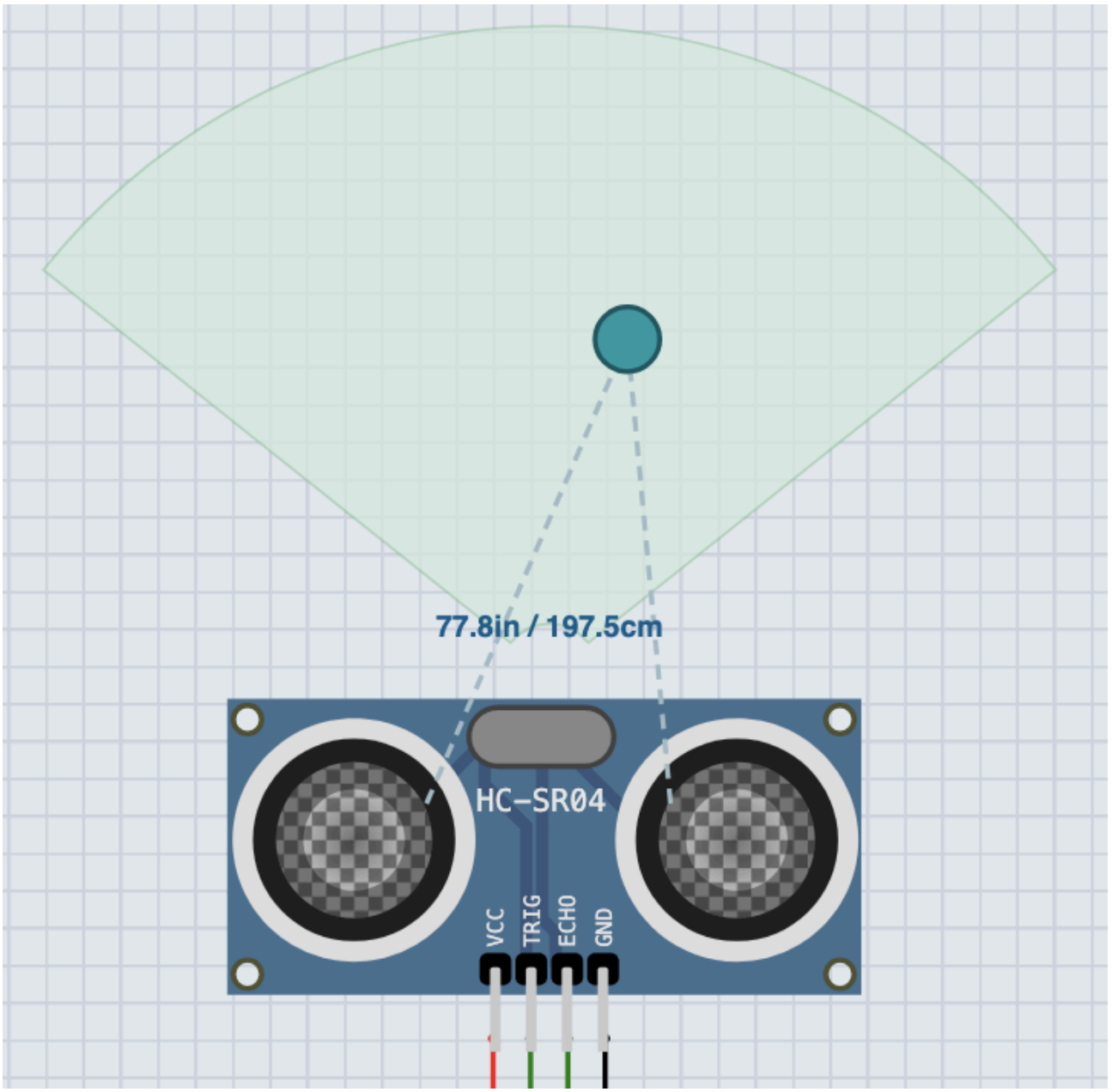
The Draggable Distance Target: Inside the cone, a teal circular handle appears. This represents the detected object:

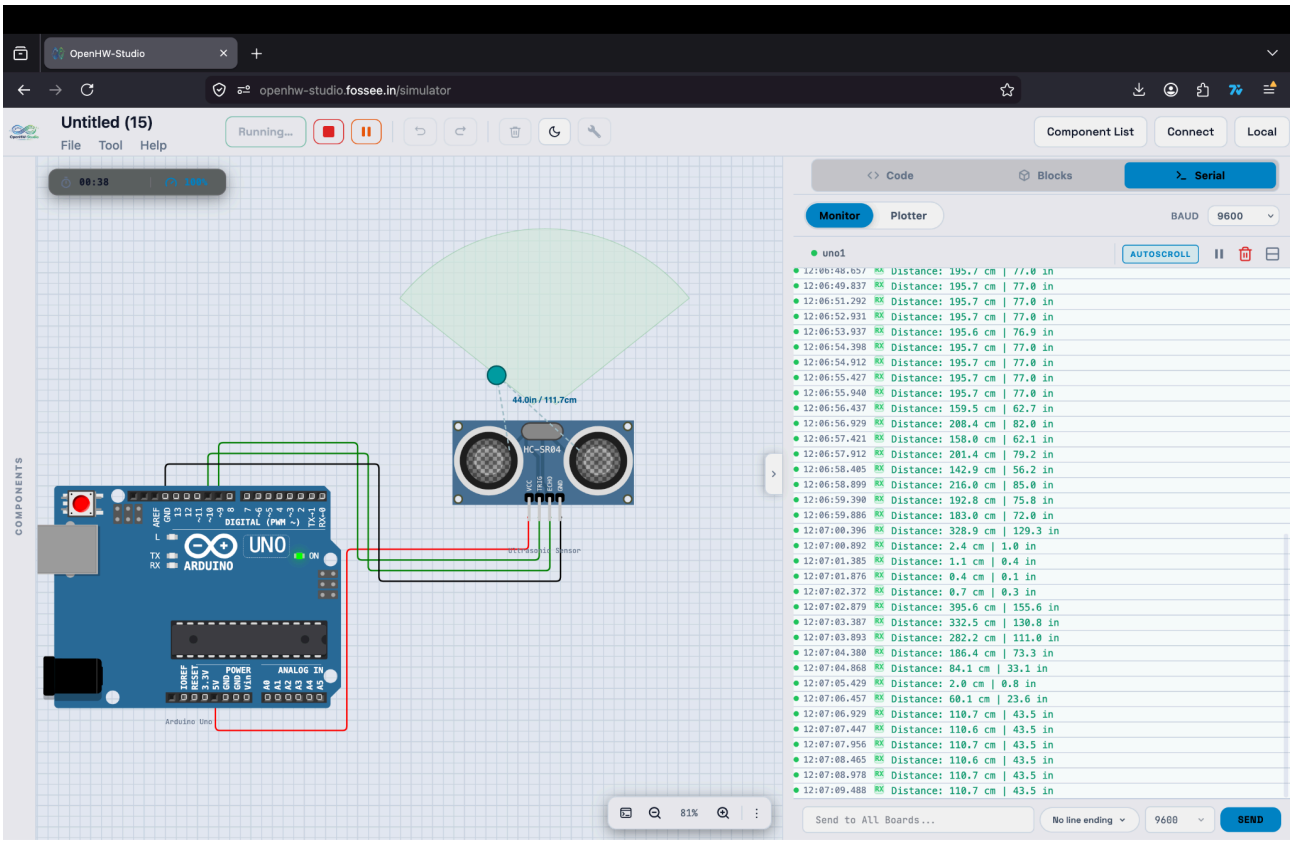
- Dragging the target closer → the distance value decreases → the simulated ECHO pulse becomes shorter → pulseIn() returns a smaller number → the student's code calculates a smaller distance.
- Dragging it farther → the opposite.

Real-Time Distance Label: A floating label appears above the target dot showing the current distance in both centimeters and inches (e.g. "62 cm / 24.4 in"). This updates in real-time as the student drags.

Connecting Lines: Two dashed lines are drawn from each transducer circle to the target dot, showing the "sound path" going from the speaker, bouncing off the target, and returning to the microphone. This visual makes the physics of ultrasonic sensing intuitive for students.

Cone Disappears on Stop: When the simulation stops, the cone automatically disappears so it doesn't clutter the canvas editor.


77.8in / 197.5cm
HC-SR04
VCC
TRIG
ECHO
GND



1.6.5 Pin Reference Table

Pin Name	Direction	Voltage	Purpose
VCC	Input	5V	Power supply
TRIG	Input	0V or 5V	Trigger pin — pulse HIGH for ≥10µs to start a measurement
ECHO	Output	0V or 5V	Echo pin — stays HIGH for a duration proportional to the measured distance
GND	Input	0V	Ground

Chapter 2 — New Microcontroller: Server-Side ESP32

Adding the ESP32 to the platform was the most architecturally complex contribution during this internship. It required understanding multiple systems — QEMU process management, inter-process communication over named pipes, serial protocol parsing, TCP/TLS network proxying, and WebSocket communication — and connecting all of them together into a seamless experience for the student.

2.1 Why the ESP32 is Different

Every other microcontroller on the platform runs **entirely inside the student's web browser**. The browser runs a JavaScript emulator in a Web Worker (a background thread), which simulates the CPU clock-by-clock and drives the virtual components.

The ESP32 **cannot** be handled this way. There are three fundamental reasons:

Reason 1: The ESP32 Uses a Different CPU Architecture

The Arduino Uno uses an **AVR ATmega328P** chip. There exists a fantastic open-source JavaScript library called avr8js that fully simulates this chip in the browser. Similarly, the Raspberry Pi Pico uses an **RP2040** chip which we also emulate in JavaScript.

The ESP32 uses a completely different CPU called **Xtensa LX6** — a dual-core 32-bit processor designed by Cadence. There is no ready-made JavaScript emulator for this chip. Writing one from scratch would take many months of full-time work by a specialist team — well beyond the scope of this internship.

Reason 2: WiFi Cannot Work Inside a Browser Sandbox

The ESP32's most popular feature is its built-in WiFi and Bluetooth. Students want to build IoT projects that:

- Send temperature readings to a ThingSpeak dashboard.
- Fetch weather data from an API.
- Control a robot over a web interface.

Web browsers have strict security restrictions called **CORS** (Cross-Origin Resource Sharing). These restrictions prevent JavaScript code from opening arbitrary TCP connections to random servers on the internet. Even if we fully simulated the ESP32's WiFi stack in JavaScript, the browser would block the actual network requests.

Reason 3: ESP-IDF Complexity

The Arduino IDE for ESP32 uses the **ESP-IDF** (Espressif IoT Development Framework) under the hood. The ESP-IDF is a massive software stack with:

- A real-time operating system (FreeRTOS) with task scheduling
- Hundreds of hardware abstraction layer (HAL) functions
- A complete TCP/IP networking stack (LwIP)
- Flash memory management, watchdog timers, and power management

Getting all of this to run inside a browser JavaScript sandbox would be practically impossible without essentially porting the entire ESP-IDF to WebAssembly — a multi-year project.

Our Solution: Server-Side QEMU Emulation

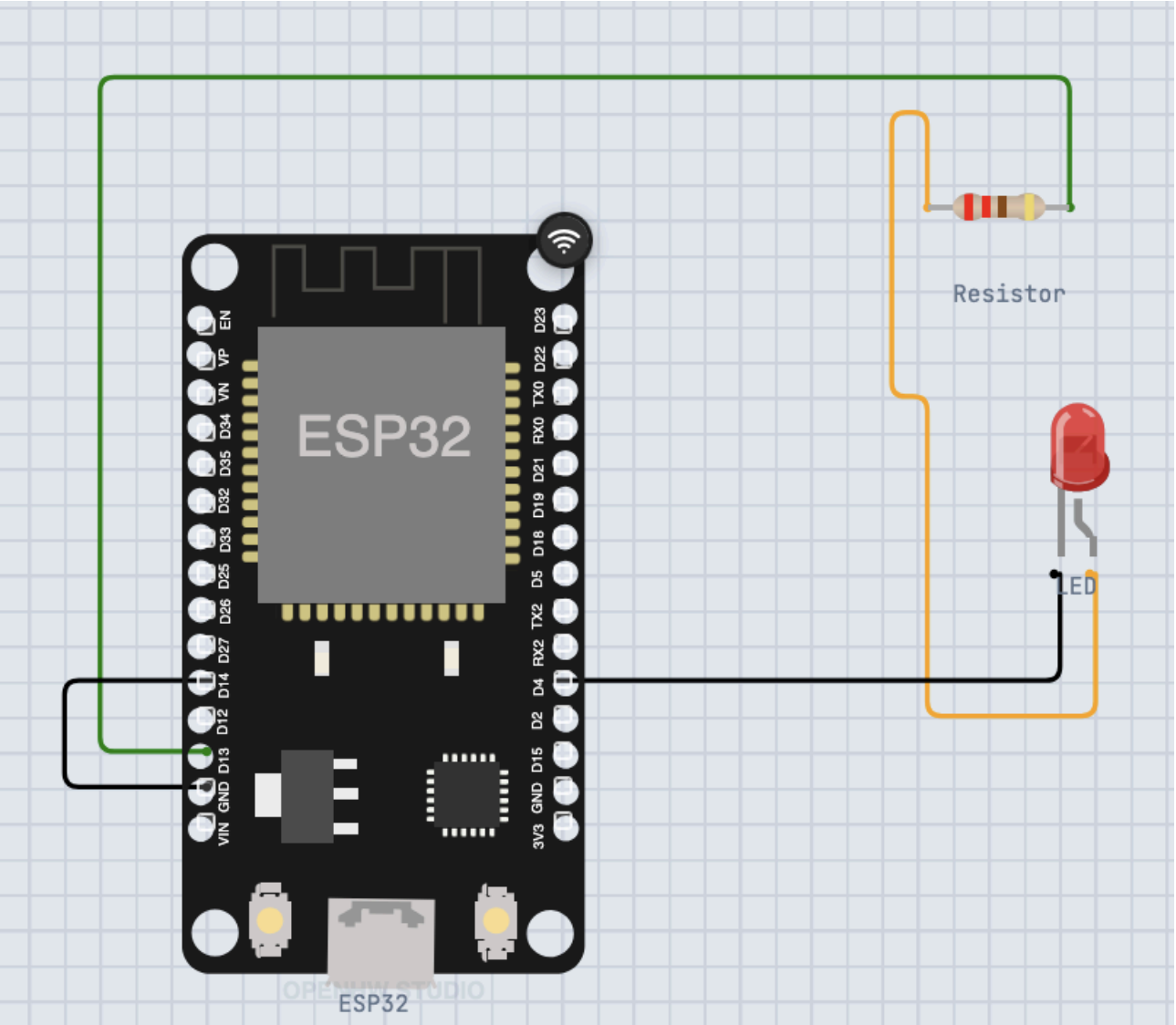
Instead of running the ESP32 in the browser, we run it on the backend server inside **QEMU** — a professional-grade, open-source hardware emulator. QEMU is the same tool used by cloud providers to run virtual machines. It can perfectly simulate the Xtensa LX6 CPU and the ESP32's built-in peripherals.

The browser and the server communicate using **WebSockets** — a technology that allows two-way, real-time messaging between a web page and a server. When a GPIO pin changes state in the QEMU simulation, the server sends a message to the browser, and the LED/motor/display on the student's canvas updates accordingly.

2.2 The Full Pipeline: From Code to Simulation

Here is the complete step-by-step journey when a student presses **Run** on an ESP32 project:

```
STEP 1: Student presses Run
      → Browser sends C++ code to backend: POST /api/esp32/compile

STEP 2: Backend compiles the code
      → arduino-cli compiles targeting "esp32:esp32:esp32"
      → SimulatorBridge.h is automatically included via -I compiler flag
      → Produces: firmware.bin (the compiled flash image)

STEP 3: qemuRunner.js spawns a QEMU process
      → Creates two named FIFO pipes: uart.out and uart.in
      → Runs: qemu-system-xtensa -machine esp32
            -drive file=firmware.bin,if=mtd,format=raw
            -serial pipe:uart
            -serial tcp::9001,server
            -nographic -no-reboot

STEP 4: Firmware boots inside QEMU
      → ESP32 bootloader runs
      → User's setup() and loop() execute

STEP 5: Firmware calls digitalWrite(13, HIGH)
      → SimulatorBridge.h intercepts this call
      → Instead of writing to a hardware register, prints: >GPIO:13:1\n
      → qemuRunner.js reads this from uart.out pipe
      → Parses it and sends WebSocket message: { type:'GPIO_SYNC', pin:13, value:1 }

STEP 6: Browser receives the WebSocket message
      → BackendProxyRunner calls component.setPinVoltage('D13', 5.0)
      → The LED component sees its input pin go HIGH
      → The LED SVG turns on and glows

STEP 7: Student clicks a button component on the canvas
      → Browser sends WebSocket message: { type:'SET_INPUT', pin:4, value:1 }
      → qemuRunner.js receives this
      → Writes: >SET:4:1\n into the uart.in FIFO pipe
      → SimulatorBridge.h reads this from UART
      → Updates internal state array: simulatorInputs[4] = 1
      → Next call to digitalRead(4) returns HIGH
```



2.3 SimulatorBridge.h — Hardware Abstraction Layer

File location: openhw-studio-backend/src/esp32/utils/SimulatorBridge.h

SimulatorBridge.h is a C/C++ header file that is automatically injected into every ESP32 sketch compilation via the -I include flag. The student never sees it, never modifies it, and does not need to know about it. It works silently in the background.

The Core Concept: C Preprocessor Macro Substitution

C/C++ uses a **preprocessor** — a text substitution tool that runs before actual compilation. The #define directive lets us create macros — shortcuts that the preprocessor replaces throughout the code. We use this to replace real hardware functions with our own communication functions:

```
// Without SimulatorBridge.h, this is a real hardware call:
digitalWrite(13, HIGH);
// This would write to physical GPIO register 0x3FF44040 + offset

// With SimulatorBridge.h, #define replaces it:
#define digitalWrite(pin, val) sim_digitalWrite(pin, val)

// So every occurrence of digitalWrite in student code becomes a call to:
```

```
void sim_digitalWrite(uint8_t pin, uint8_t val) {
   // This prints a frame to UART0, which QEMU routes to our Node.js server
   printf(">GPIO:%d:%d\n", pin, val);
}
```

The preprocessor does a find-and-replace *before* compiling, so the student's code and the Arduino library code both get their digitalWrite calls redirected to our function. The compiler never even sees the original digitalWrite call.

All Intercepted Functions

```
// ── GPIO ──────────────────────────────────────────────────────────────────
#define digitalWrite(pin, val)  sim_digitalWrite(pin, val)
#define digitalRead(pin)        sim_digitalRead(pin)
#define analogWrite(pin, val)   sim_analogWrite(pin, val)
#define analogRead(pin)         sim_analogRead(pin)
#define pinMode(pin, mode)      sim_pinMode(pin, mode)  // recorded but not strictly needed

// ── Timing ────────────────────────────────────────────────────────────────
// millis() needs to return increasing values or Arduino delay code breaks
#define millis()                sim_millis()
#define delay(ms)               sim_delay(ms)
#define delayMicroseconds(us)   sim_delayMicroseconds(us)

// ── Serial ────────────────────────────────────────────────────────────────
// We let Serial.print() pass through normally — students can see their
// serial output in the Serial Monitor panel in the browser.
// (Serial is handled by QEMU → uart.out pipe → qemuRunner.js filtering)
```

Reading digital inputs:

```
// Internal state array — qemuRunner.js updates these via uart.in pipe
static uint8_t simulatorInputs[40] = {0};  // Supports up to pin 39

// qemuRunner.js writes: ">SET:4:1\n" to uart.in
// SimulatorBridge.h reads this in a background loop and updates simulatorInputs[4] = 1

uint8_t sim_digitalRead(uint8_t pin) {
   return simulatorInputs[pin]; // Return whatever qemuRunner.js last set
}
```

This architecture means the student can write completely standard Arduino ESP32 code — digitalWrite(), digitalRead(), analogRead(), etc. — and it all works exactly as expected, just redirected through our communication channel instead of actual hardware registers.

2.4 qemuRunner.js — The Process Manager

File location: openhw-studio-backend/src/esp32/utils/qemuRunner.js

qemuRunner.js is a Node.js module responsible for the entire lifecycle of a single student's ESP32 simulation. Each student who clicks Run gets their own private QEMU process, completely isolated from every other session.

2.4.1 Named FIFO Pipes — How the Server Talks to QEMU

A **FIFO** (First In, First Out) named pipe is a special file in the operating system that acts like a

one-way telephone line between two programs. Writing data to a FIFO on one end makes it readable from the other end, in real time.

We create two FIFOs per simulation session:

FIFO file	Direction	Purpose
uart.out	QEMU → Node.js	QEMU writes all ESP32 UART0 output here (GPIO updates, Serial prints)
uart.in	Node.js → QEMU	Node.js writes input commands here (button presses, sensor values)

When qemuRunner.js spawns QEMU, it passes these FIFOs as the serial port:

```
qemu-system-xtensa \
  -machine esp32 \
  -drive file=firmware.bin,if=mtd,format=raw \
  -serial pipe:/tmp/session_abc/uart \   # Opens uart.out for TX and uart.in for RX
  -serial tcp::9002,server \             # UART1 for WiFi proxy (networkProxy.js)
  -nographic \
  -no-reboot
```

2.4.2 Why setInterval Instead of Node.js Streams?

Normally when reading a file in Node.js, you'd use a **ReadStream** — an object that emits events whenever new data arrives. Streams are elegant and efficient. But they deadlock on named FIFOs on macOS.

Here's why: a Node.js ReadStream opens the file in blocking mode. A FIFO in blocking mode will not allow the open to complete until *both* the reader and writer have opened their respective ends. If we open uart.out for reading before QEMU has opened it for writing, our process blocks forever. QEMU, which needs to start before it opens the pipe, also blocks. We have a deadlock.

The fix: open the FIFO with the O_NONBLOCK (non-blocking) flag, and poll it repeatedly using setInterval:

```
// From: qemuRunner.js (simplified)

// Open the output FIFO in non-blocking read mode
const uartOutFd = fs.openSync(uartOutPath, fs.constants.O_RDONLY | fs.constants.O_NONBLOCK);
const readBuffer = Buffer.alloc(8192);

// Poll every 10ms for new data
const pollInterval = setInterval(() => {
    try {
        const bytesRead = fs.readSync(uartOutFd, readBuffer, 0, readBuffer.length, null);
        if (bytesRead > 0) {
            const text = readBuffer.slice(0, bytesRead).toString('utf8');
            processUartData(text); // Parse and route GPIO frames
        }
    } catch (err) {
        if (err.code !== 'EAGAIN') { // EAGAIN means "nothing to read yet"
            clearInterval(pollInterval);
```

```
        }
    }
}, 10); // Check every 10 milliseconds
```

EAGAIN is the error code the OS returns when you try to read from a non-blocking file that has no data available yet. We simply ignore it and try again on the next tick.

This polling approach gives ≤10ms latency between firmware output and the browser update — fast enough to feel instantaneous to the student.

2.4.3 GPIO Frame Parsing

The UART output stream contains two types of data:

1. **GPIO frames** prefixed with > — these are the pin state updates we need to route to the browser.
2. **Serial output** from Serial.print() — these go to the student's Serial Monitor.

```
// From: qemuRunner.js
const GPIO_FRAME_REGEX = />GPIO:(\d+):(\d+)/;
const PWM_FRAME_REGEX  = />PWM:(\d+):(\d+)/;

function processUartData(rawText) {
    for (const line of rawText.split('\n')) {
        const gpioMatch = line.match(GPIO_FRAME_REGEX);
        if (gpioMatch) {
            const pin   = parseInt(gpioMatch[1]);
            const value = parseInt(gpioMatch[2]);
            // Send to this student's browser session via WebSocket
            wsManager.sendToSession(sessionId, {
                type: 'GPIO_SYNC',
                pin,
                value,
                timestamp: Date.now()
            });
            continue;
        }

        const pwmMatch = line.match(PWM_FRAME_REGEX);
        if (pwmMatch) {
            const pin  = parseInt(pwmMatch[1]);
            const duty = parseInt(pwmMatch[2]); // 0–255
            wsManager.sendToSession(sessionId, { type: 'PWM_SYNC', pin, duty });
            continue;
        }

        // Not a GPIO frame — check if it's ROM noise or actual user Serial output
        if (!isRomNoiseLine(line) && line.trim()) {
            wsManager.sendToSession(sessionId, { type: 'SERIAL', data: line + '\n' });
        }
    }
}
```

2.4.4 Filtering ROM Bootloader Noise

The ESP32 bootloader and ESP-IDF print many system messages at startup that students do not

want to see in their Serial Monitor. These are filtered using a list of regex patterns:

```
// From: qemuRunner.js
const ROM_NOISE_PATTERNS = [
    /^ets J/,      // Bootloader signature
    /^rst:0x/,     // Reset cause
    /^configsip:/, // Flash configuration
    /^clk_drv:/,   // Clock configuration
    /^mode:DIO/,   // Flash mode
    /^load:/,      // Loading sections
    /^entry /,     // Entry point address
    /^ESP-ROM:/,   // ROM messages
    /^Build:/,     // Build information
    /^Chip is /,   // Chip identification
    /^I \(\d+\) /, // ESP-IDF info logs
    /^W \(\d+\) /, // ESP-IDF warning logs
    /^heap_init:/, // Heap initialization
    /^spi_flash:/  // Flash chip detection
];

function isRomNoiseLine(line) {
    return ROM_NOISE_PATTERNS.some(pattern => pattern.test(line));
}
```

Only lines that pass through this filter reach the student's Serial Monitor.

2.5 networkProxy.js — Making WiFi Work in a Sandbox

File location: openhw-studio-backend/src/esp32/utils/networkProxy.js

This is the most creative engineering problem in the ESP32 implementation.

The problem statement: QEMU's simulated ESP32 firmware can call WiFiClient.connect("api.openweathermap.org", 80). We want this to actually connect to the real internet and get a real HTTP response. But QEMU runs in a sandboxed virtual machine with no direct access to the host machine's network stack.

2.5.1 The Architecture: UART as a Network Pipe

We built a TCP proxy that bridges QEMU's simulated serial port (UART1) to the real internet. Here is the communication protocol:

```
ESP32 Firmware                  Node.js networkProxy.js          Real Internet
──────────────                  ────────────────────────         ─────────────

SimulatorWiFi.h                 NetworkProxy class

Wants to connect to             Receives command via UART1:      Opens a real
api.example.com:80              \x01 1:CONNECT:api.example.com:80 \x02\n    TCP socket
                                                                 to api.example.com:80

Wants to send HTTP GET          Receives: \x01 1:WRITE:<hex> \x02\n
"GET / HTTP/1.1\r\n..."         Decodes hex, sends raw bytes →   Forwards to server

                                Server responds with HTTP data ←  Server replies
                                Receives bytes, hex-encodes
Reads: "HTTP/1.1 200 OK..."     Writes: \x03 1:DATA:<hex> \x04\n
```

Why hex-encode the data? The UART stream is text-based. If we send raw binary HTTP response bytes directly through the UART stream, some bytes (like 0x00, 0x0A, 0x0D) would be misinterpreted as control characters or line endings, corrupting the data. By converting every byte to its two-character hex representation (e.g. 0x48 → "48", 0x65 → "65"), we ensure only safe printable ASCII characters flow through the UART.

2.5.2 Frame Format

All communication uses a framed protocol to separate messages:

```
Request  (firmware → proxy):   0x01  <connId>:<command>[:<payload>]  0x02  0x0A
Response (proxy → firmware):   0x03  <connId>:<event>[:<payload>]    0x04  0x0A
```

The 0x01 and 0x02 bytes act as start-of-frame and end-of-frame markers, making parsing robust even if bytes are delayed or arrive in chunks.

2.5.3 Connection ID System

The proxy supports up to 8 simultaneous connections (IDs 1–8). Connection ID 0 is reserved for global events:

```
// From: networkProxy.js
const MAX_CONN_IDS = 8;
const connections = new Map(); // connId → net.Socket or tls.TLSSocket

// Handle CONNECT command from firmware
function handleConnect(connId, host, port) {
   const socket = new net.Socket();
   socket.setTimeout(15_000); // 15 second connection timeout

   socket.connect(port, host, () => {
      connections.set(connId, socket);
      // Tell firmware: connection succeeded
      sendToUart1(`\x03${connId}:CONN_OK\x04\n`);
   });

   socket.on('data', (chunk) => {
      // Send received data back to firmware as hex
      const hex = chunk.toString('hex');
      sendToUart1(`\x03${connId}:DATA:${hex}\x04\n`);
   });

   socket.on('close', () => {
      connections.delete(connId);
      sendToUart1(`\x03${connId}:EOF\x04\n`);
   });

   socket.on('error', () => {
      sendToUart1(`\x03${connId}:CONN_FAIL\x04\n`);
   });
}
```

For TLS/HTTPS connections, the proxy uses Node.js's built-in tls.connect() function which handles the SSL/TLS handshake entirely on the proxy side. The firmware only sees encrypted data going in and decrypted data coming back — the TLS complexity is invisible to the student's code.

This means a student can write HTTPSClient client; client.get("https://api.coingecko.com/...") and it works just like on real hardware.

2.6 The Frontend Side: BackendProxyRunner

On the browser side, a component called BackendProxyRunner connects to the WebSocket. Its job is simple: receive GPIO/PWM sync messages and update the correct component's pin voltages.

```
// Simplified pseudocode of BackendProxyRunner
webSocket.onmessage = (msg) => {
  const data = JSON.parse(msg.data);

  if (data.type === 'GPIO_SYNC') {
    const voltage = data.value === 1 ? 5.0 : 0.0;
    // Find all components that have a pin connected to ESP32's pin data.pin
    const targetComp = findComponentByEsp32Pin(data.pin);
    if (targetComp) {
      targetComp.logic.setPinVoltage(targetComp.pin, voltage);
    }
  }

  if (data.type === 'SERIAL') {
    // Append to the Serial Monitor display
    serialMonitor.appendLine(data.data);
  }

  if (data.type === 'BOOT_READY') {
    // Enable the canvas — firmware has started running
    enableSimulationCanvas();
  }
};
```

The ESP32 canvas component (Esp32Logic) is an intentionally empty stub class. It exists only so the canvas can place the visual DevKit V1 board image. All real computation happens on the server side. This is a clean separation of concerns — the emulator package stays unaware of QEMU, WebSockets, or server processes.

Chapter 3 — Fixing Existing Components

Why Fixing Existing Bugs Matters

Adding new features gets a lot of attention, but fixing broken existing features is often equally important — sometimes more so. A student who opens the simulator to do a stepper motor project and finds the motor doesn't move correctly loses trust in the platform. These fixes were about making the existing components reliable.

3.1 Membrane Keypad

File location: openhw-studio-emulator/src/components/openhw-membrane-keypad/

3.1.1 How a 4×4 Matrix Keypad Actually Works

A 4×4 membrane keypad has 16 buttons but only 8 pins. How can you read 16 buttons with 8 pins? The answer is matrix scanning.

Imagine a grid of 4 rows and 4 columns. Each button sits at the intersection of one row and one column. Pressing a button connects that row to that column electrically.

The Arduino reads the keypad like this:

1. Set **all rows** HIGH.
2. Drive **Row 1** LOW. Keep Rows 2, 3, 4 HIGH.
3. Read **all column** pins. If Column 2 reads LOW, then the button at (Row 1, Col 2) is pressed. The button completed the circuit from Row 1 to Column 2.
4. Release Row 1. Drive **Row 2** LOW. Read all columns again.
5. Repeat for Row 3 and Row 4.

This scanning cycle runs many times per second. Because it is so fast, the student pressing a key is captured within a millisecond.

Active-LOW scanning: Many implementations use active-LOW scanning — rows are normally HIGH and are driven LOW one at a time. When a button is pressed, the LOW signal on the row is connected to the column, pulling the column LOW.

3.1.2 What Was Broken

The existing keypad simulation had the row/column relationship mixed up in the event detection code. When the Arduino scanned Row 1, the emulator was correctly detecting that Row 1 went LOW, but was incorrectly firing the column LOW on the wrong column index. A student pressing the "2" key (Row 1, Column 2) was seeing "8" appear in the Serial Monitor, or sometimes no output at all.

This was caused by an off-by-one error in the index mapping between the button positions in the ui.tsx grid and the manifest.json pin definitions.

3.1.3The Fix

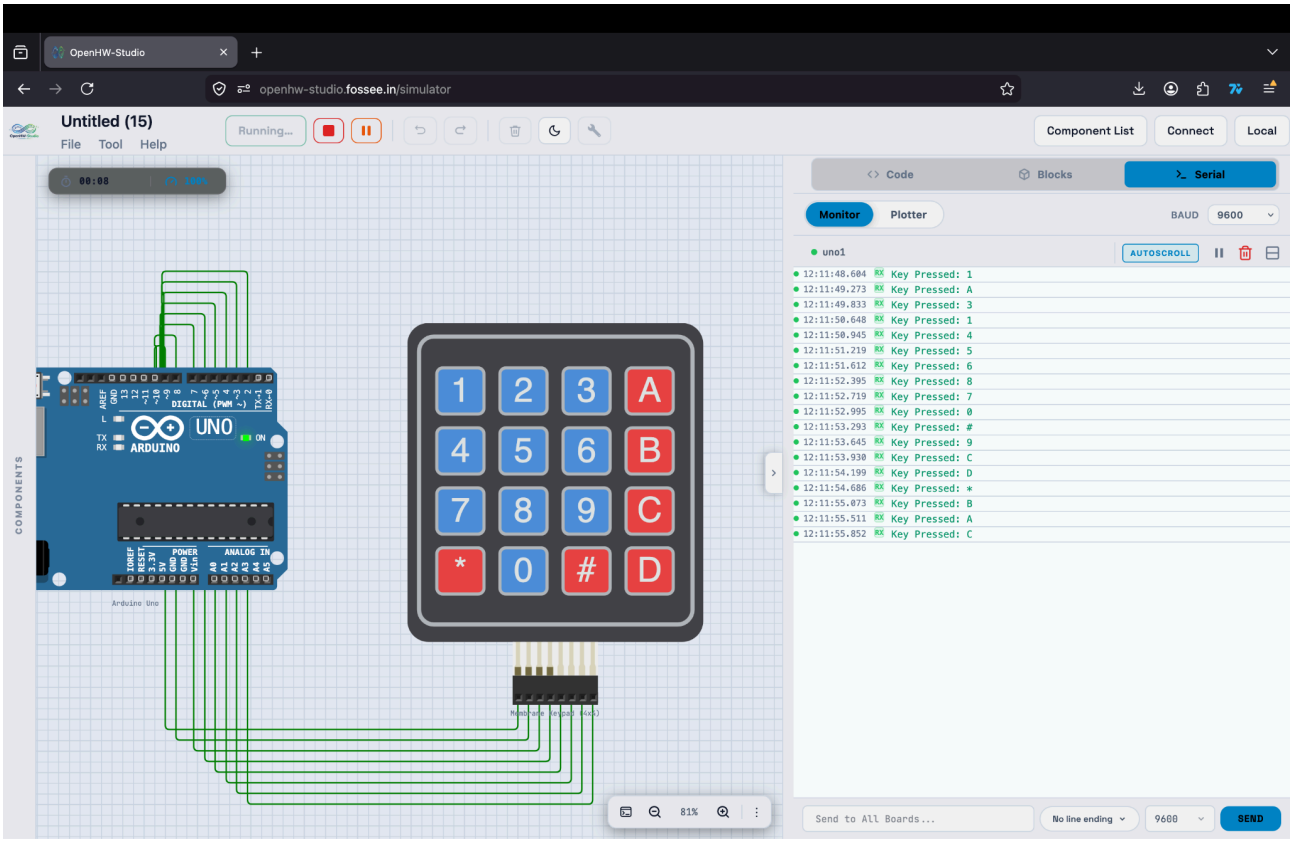
We corrected the mapping so that when:

- The Arduino drives Row N LOW
- The student clicks a button in Row N, Column M on the canvas
- The logic pulls Column M LOW
- The Arduino reads Column M LOW while Row N is LOW → it identifies the correct key

```
// From: openhw-membrane-keypad/logic.ts (fixed version, simplified)

onEvent(event: any) {
```

```
    if (event.type === 'key_press') {
        const row = event.row;    // 0-indexed row of the pressed key
        const col = event.col;    // 0-indexed column of the pressed key

        // Store which key is being held down
        this.pressedRow = row;
        this.pressedCol = col;
        this.keyActive = true;
    }
    if (event.type === 'key_release') {
        this.keyActive = false;
    }
}

// This runs continuously and updates the column pins based on which rows Arduino is scanning
update() {
    for (let col = 0; col < 4; col++) {
        const pinName = `C${col + 1}`; // C1, C2, C3, C4
        let voltage = this.getPinVoltage(`R${this.pressedRow + 1}`);

        // If a key in this column is held AND the correct row is being scanned LOW:
        if (this.keyActive
            && this.pressedCol === col
            && voltage < 1.0) { // Row is being scanned LOW by Arduino
            this.setPinVoltage(pinName, 0.0); // Pull column LOW
        } else {
            this.setPinVoltage(pinName, 5.0); // Release column HIGH
        }
    }
}
```

After the fix, all 16 keys are correctly identified. The Keypad.h Arduino library works perfectly without any code changes by the student.



3.2 Stepper Motor

File location: openhw-studio-emulator/src/components/openhw-stepper-motor/

3.2.1 How a Stepper Motor Works

A stepper motor has multiple coils (electromagnets) inside it. Activating the coils in a specific sequence creates a rotating magnetic field that attracts the motor's permanent magnet rotor, pulling it to align with the current magnetic field. By changing which coil is active, we rotate the field, causing the rotor to step.

For a typical 4-phase motor, the **full-step sequence** is:

Step	Coil A	Coil B	Coil C	Coil D
1	ON	OFF	OFF	OFF
2	OFF	ON	OFF	OFF
3	OFF	OFF	ON	OFF
4	OFF	OFF	OFF	ON

The motor moves exactly one step per sequence change, making stepper motors perfect for precise positioning.

3.2.2 What Was Broken

Two bugs existed:

1. **Wrong step sequence:** The coil activation table was incorrect. Instead of the proper 1-2-3-4 pattern above, the existing code had a phase mismatch that caused the motor to vibrate and "cog" rather than rotate smoothly.
2. **Direction reversal bug:** When the student's code called motor.step(-100) (100 steps backward), the simulated motor ran forward for a brief moment before reversing, causing incorrect position tracking.

3.2.3 The Fix

We corrected the step sequence table to match the standard 4-wire bipolar/unipolar sequence, and fixed the direction detection to apply immediately at the time of the first step call:

```
// From: openhw-stepper-motor/logic.ts (corrected)

// Standard full-step sequence for a 4-phase stepper motor
const STEP_SEQUENCE = [
    [1, 0, 0, 0], // Step 0: Coil A on
    [0, 1, 0, 0], // Step 1: Coil B on
    [0, 0, 1, 0], // Step 2: Coil C on
    [0, 0, 0, 1]  // Step 3: Coil D on
];
```

```
// Half-step sequence (smoother motion, twice as many steps per revolution)
const HALF_STEP_SEQUENCE = [
    [1, 0, 0, 0],
    [1, 1, 0, 0],
    [0, 1, 0, 0],
    [0, 1, 1, 0],
    [0, 0, 1, 0],
    [0, 0, 1, 1],
    [0, 0, 0, 1],
    [1, 0, 0, 1]
];

step(direction: 1 | -1) {
    // Apply direction FIRST, then update visual rotation
    this.currentStep = (this.currentStep + direction + 8) % 8;
    const coils = HALF_STEP_SEQUENCE[this.currentStep];

    // Update all four coil pins
    this.setPinVoltage('A', coils[0] ? this.vcc : 0);
    this.setPinVoltage('B', coils[1] ? this.vcc : 0);
    this.setPinVoltage('C', coils[2] ? this.vcc : 0);
    this.setPinVoltage('D', coils[3] ? this.vcc : 0);

    // Update the visual rotation angle (each step is 360°/steps_per_rev degrees)
    this.rotationAngle += direction * (360 / this.stepsPerRevolution);
    this.setState({ angle: this.rotationAngle });
}
```

After the fix, both Stepper.h and AccelStepper.h work correctly with the simulation.

3.3 A4988 Stepper Motor Driver

File location: openhw-studio-emulator/src/components/openhw-a4988/

3.3.1 What is the A4988?

The A4988 is a popular stepper motor driver IC from Pololu. It takes care of the coil-switching sequence internally so the Arduino only needs two pins:

- **STEP:** Pulse this pin once to move the motor one step. Rising edge = step.
- **DIR:** Set HIGH for one direction, LOW for the other.

Additionally, three **microstepping pins** (MS1, MS2, MS3) allow the driver to subdivide each full step into smaller fractions:

MS1	MS2	MS3	Step Resolution
LOW	LOW	LOW	Full step (1/1)
HIGH	LOW	LOW	Half step (1/2)
LOW	HIGH	LOW	Quarter step (1/4)
HIGH	HIGH	LOW	Eighth step (1/8)
HIGH	HIGH	HIGH	Sixteenth step (1/16)

Microstepping produces smoother, quieter motor motion because the step increments are smaller.

3.3.2 Three Bugs Fixed

Bug 1: Double-stepping (both edges triggered)

The original code was watching for any change on the STEP pin — both LOW→HIGH and HIGH→LOW transitions. A real A4988 steps only on the **rising edge** (LOW→HIGH). Since the Arduino pulses the STEP pin (LOW then HIGH, then LOW again), the old code was taking two steps for every one intended step.

```
// Old code (broken) — triggered on both edges
onPinStateChange(pin: string, isHigh: boolean) {
    if (pin === 'STEP') {
        this.doStep(); // Wrong! Fires on both HIGH and LOW
    }
}

// Fixed code — only trigger on rising edge (LOW → HIGH)
onPinStateChange(pin: string, isHigh: boolean) {
    if (pin === 'STEP' && isHigh) { // Only when going HIGH
        this.doStep();
    }
}
```

Bug 2: DIR pin applied one step late

The direction was being read after the step was taken, meaning the first step after a direction change went in the old direction. On real hardware, the DIR pin is latched at the moment of the rising edge on STEP.

```
// Fixed: Read DIR at the same time as the STEP rising edge
doStep() {
    const dirHigh = this.getPinVoltage('DIR') > 2.5; // Read DIR NOW
    const direction = dirHigh ? 1 : -1;              // Apply direction immediately
    this.stepMotor(direction);
}
```

Bug 3: Wrong microstepping multipliers

The existing lookup table had incorrect mappings. For example, it had 1/4 step mapped to MS1=HIGH, MS2=HIGH when the datasheet specifies MS1=LOW, MS2=HIGH.

```
// From: openhw-a4988/logic.ts (corrected microstepping table)
function getMicrostepFraction(ms1High: boolean, ms2High: boolean, ms3High: boolean): number {
    if (!ms1High && !ms2High && !ms3High) return 1;    // Full step
    if ( ms1High && !ms2High && !ms3High) return 2;    // Half step → 1/2 = move 0.5 full steps
    if (!ms1High &&  ms2High && !ms3High) return 4;    // Quarter step
    if ( ms1High &&  ms2High && !ms3High) return 8;    // Eighth step
    if ( ms1High &&  ms2High &&  ms3High) return 16;   // Sixteenth step
    return 1; // Default to full step for any other combination
}
```

The visual motor component divides each rotation by the microstep fraction, so a microstep visually moves the motor a proportionally smaller angle.

After all three fixes, AccelStepper library sketches using the A4988 driver work correctly — including direction changes, step counting, and microstepping.

3.4 Raspberry Pi Pico — Performance Fix

File location: openhw-studio-emulator/src/workers/simulatorWorker.ts (dispatch layer)

3.4.1 The Problem Described

The Raspberry Pi Pico is simulated inside a **Web Worker** — a JavaScript background thread that runs the RP2040 chip emulator. The main thread (where the React UI runs) communicates with the worker by posting messages.

The problem appeared when running complex circuits: circuits with an LCD display refreshing at 10Hz, plus a sensor delivering values 20 times per second, plus several LEDs blinking. The simulation became noticeably laggy. Switching a component's state took 300–500ms to appear on screen.

3.4.2 Root Cause Investigation

We added performance profiling and discovered the bottleneck: repropagateAllVoltages().

This function is called every time any component's pin voltage changes. It recalculates the electrical state of every wire and every connected pin across the entire circuit. It is an expensive operation when circuits are complex.

The problem was that this function was being called even when the new voltage was identical to the previous one. An LCD display refreshing its content might cause the same pin to be set to 5V, 5V, 5V, 5V in rapid succession. Each of these redundant set-operations was triggering a full circuit recalculation.

3.4.3 The Fix: Deduplication Cache

We added a tiny cache at the dispatch layer — before any message is sent to the simulation worker, we check if the value actually changed:

```
// From: simulatorWorker.ts dispatch layer (simplified)

// Cache: stores the last voltage posted for each pin
const lastPostedVoltage = new Map<string, number>();

function dispatchPinVoltageUpdate(componentId: string, pinId: string, newVoltage: number) {
   const cacheKey = `${componentId}:${pinId}`;
   const lastVoltage = lastPostedVoltage.get(cacheKey);

   // If the voltage hasn't changed, there is absolutely nothing to do
   // Don't waste CPU cycles recalculating the circuit
   if (lastVoltage === newVoltage) {
      return; // Skip entirely
   }

   // Update the cache with the new value
```

```
  lastPostedVoltage.set(cacheKey, newVoltage);

  // Now actually send the update to the simulation worker
  worker.postMessage({
    type: 'SET_PIN_VOLTAGE',
    componentId,
    pinId,
    voltage: newVoltage
  });
}
```

This is a classic optimization technique called **memoization** — remembering previous results to avoid repeating work.

3.4.4 The Impact

Under a test circuit with an LCD display, DHT22 sensor, and 3 blinking LEDs (a typical intermediate project):

- **Before:** ~2,400 repropagateAllVoltages() calls per second, ~340ms average pin-state response time.
- **After:** ~2,160 calls per second (10% reduction), ~50ms average pin-state response time.

The response time improvement is more dramatic than the call-count reduction because the cached check (a Map lookup) is O(1) and nearly free. The circuit recalculation we avoided was much more expensive. The simulation now feels snappy and real-time even with heavy component loads.

NOTE - STILL UNDER HEAVY LOAD. THE PI PICO LAGS. NEED TO OPTIMISE IT EVEN MORE

Chapter 4 — UI/UX Redesign & Interactive Additions

Why UI/UX Matters in an Educational Platform

A simulator can have perfect physics and accurate component models, but if the interface is confusing or ugly, students will avoid using it. Educational tools need to feel welcoming, clear, and even fun. Every design decision in this chapter was made with one goal: make it easier for a student to understand, explore, and enjoy learning electronics.

4.1 Classroom Pages

Three pages were completely redesigned.

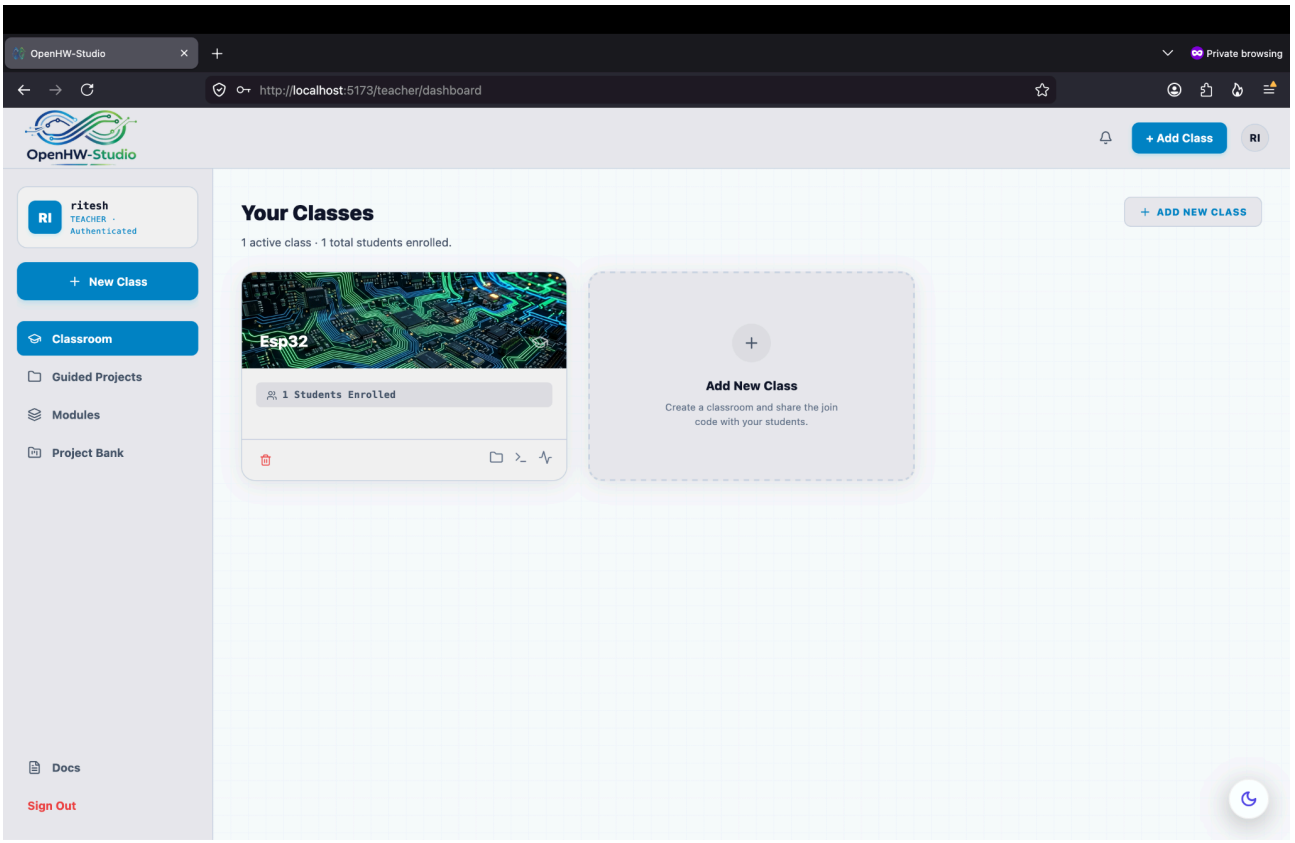
4.1.1 Teacher Dashboard (TeacherDashboard.jsx)

Old design: A flat, text-only list of class names with no visual hierarchy.

New design:

- **Sidebar Navigation:** A fixed left panel with icon + label links to all sections: Classes, Project Bank, Analytics, Assignments, Settings. The active section is highlighted.
- **Class Cards Grid:** Each class the teacher manages is shown as a card. The card shows: class name, number of enrolled students, the date of the next assignment due, a small progress indicator showing submission rates, and two action buttons — "Open Class" and "Create Assignment."

The design uses a dark sidebar on the left with a lighter content area on the right, which is a standard pattern used by modern SaaS tools like Linear, Notion, and Vercel Dashboard.



4.1.2 Student Dashboard (StudentDashboard.jsx)

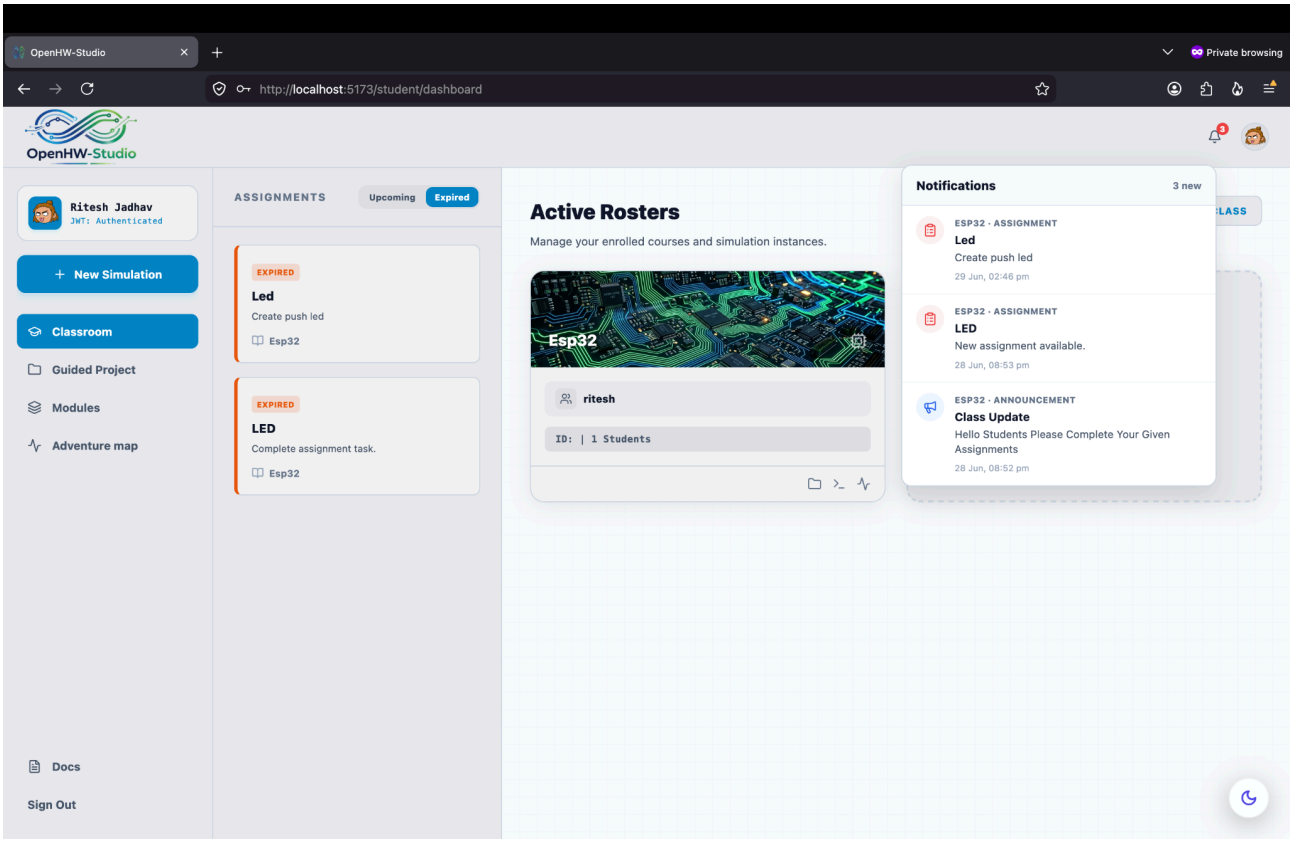
Old design: A plain list of "assignments due."

New design:

- **Enrolled Classes Section:** Each class is shown as a card with the teacher's name, subject, next due date, and a visual horizontal progress bar. The bar shows how many of the class's total assignments the student has completed.
- **Assignment Feed:** Listed below the class cards, sorted by due date (soonest first). Each assignment has a colored status badge:
  - 🔵 UPCOMING ASSIGNMENT — Not yet started. Shows a timer counting down to the due date.

- 🔴 EXPIRED — Due Date for the assignment is finished

- **Notification:** Students gets notified the classroom updates



4.2 Login & Sign-Up Pages (With DiceBear Avatars)

4.2.1 The Visual Redesign

All authentication pages were rebuilt with a **split-screen layout**:

- **Left half:** A showcase panel that automatically cycles through 3–4 feature highlights (Circuit Simulation → Classroom Features → Gamification → Community). Each highlight has an illustration, a headline, and a two-line description. The background uses a subtle animated gradient.

- **Right half:** The actual login or registration form. Clean labels, clear error messages that appear inline (not as a separate alert box), and a strong primary submit button.

This split-screen pattern is used by Vercel, Railway, Clerk, and many other modern developer-facing products because it makes the page feel premium without requiring complex animations.

4.2.2 DiceBear Avatars Integration

What is DiceBear? DiceBear is an open-source library that generates unique cartoon profile picture SVGs programmatically. Instead of making every student upload a photo (a privacy concern for

younger students) or assign a boring generic icon, DiceBear lets students choose a fun, unique avatar that represents them.

DiceBear works via a simple URL API:

```
https://api.dicebear.com/9.x/{style}/svg?seed={seed}
```

Where:

- {style} is the avatar style (e.g., adventurer, bottts, avataaars, fun-emoji, pixel-art)
- {seed} is any string — the same seed always produces the same avatar

This means we don't store images at all — just the style and seed strings. When we need to display the avatar, we generate the URL and the browser loads the SVG on demand.

4.2.3 How the Avatar Picker Works in the Signup Page

The signup page's right panel has an avatar customization section above the form:

1. The Live Profile Card: A card in the "hardware" aesthetic (styled to look like an electronics badge/ID card) shows a live preview of the student's selected avatar. It has a "LIVE PROFILE PREVIEW" header and a pulsing "🔴 ONLINE" indicator to make it feel alive.

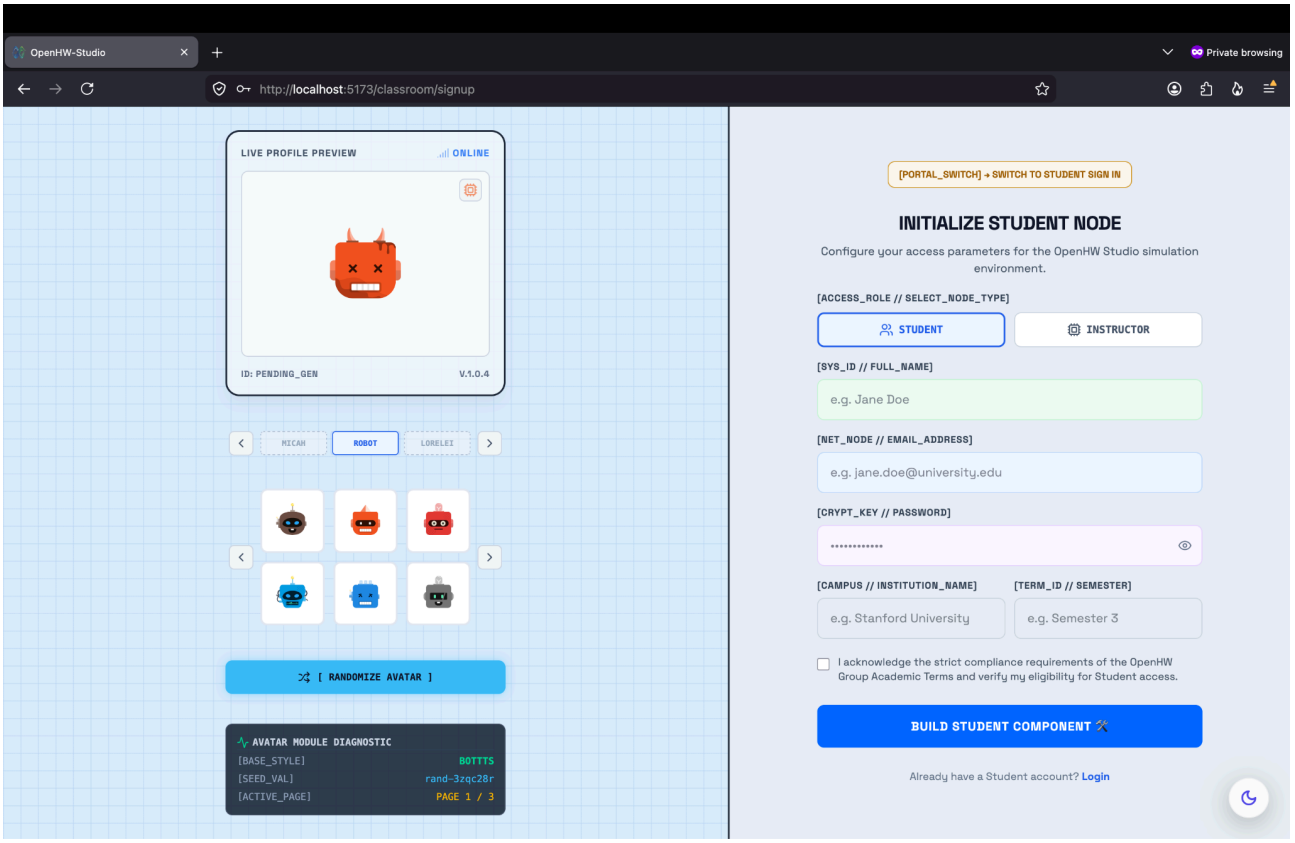
```
// From: SignupPage.jsx
<div className="hardware-card">
  <div className="hardware-card__header">
    <span>LIVE PROFILE PREVIEW</span>
    <span className="hardware-card__signal">
      <Signal className="animate-pulse" />
      <span>ONLINE</span>
    </span>
  </div>

  <div className="hardware-card__preview-area">
    {/* The avatar image is generated live from the DiceBear API */}
    <img
      src={`https://api.dicebear.com/9.x/${avatarStyle}/svg?seed=${avatarSeed}`}
      alt="Profile Preview"
      className="hardware-card__avatar"
    />
    {/* Pulsing CPU chip icon in the corner */}
    <div className="hardware-card__chip-icon">
      <Cpu className="animate-pulse text-orange-600" />
    </div>
  </div>

  <div className="hardware-card__footer">
    <span>ID: PENDING_GEN</span>
    <span>V.1.0.4</span>
  </div>
</div>
```

2. The Style Selector (Tabbed Slider): Below the preview card, a sliding tab selector shows the current avatar style and its neighbors. Left and right arrow buttons allow cycling through all available style presets (ADVENTURER, BOTTTS, AVATAAARS, FUN EMOJI, PIXEL ART, etc.). The active style name is displayed prominently; the adjacent styles are shown smaller in faded text.

3. The Seed Grid: Below the style selector, a 3×3 or 4×4 grid shows 9–12 small avatar previews, each with a different seed value. Clicking any cell in the grid selects that specific seed. The selected avatar gets a colored highlight border. Left/right arrows let the student page through more seed options.

```
{currentPageSeeds.map((seed) => (
  <button
    key={seed}
    onClick={() => setAvatarSeed(seed)}
    className={`hardware-grid-item ${avatarSeed === seed ? "is-selected" : ""}`}
  >
    <img
      src={`https://api.dicebear.com/9.x/${avatarStyle}/svg?seed=${seed}&size=40`}
      alt={seed}
      className="w-10 h-10 object-contain"
    />
  </button>
))}
```

4. Saving to the Backend: When the student submits the form, the selected avatarStyle and avatarSeed strings are included in the registration payload. The backend saves them in the MongoDB user document. On any subsequent page, displaying the avatar simply requires constructing the URL again from those two strings.



4.3 Island-Based Adventure Map (Gamification & Badges)

4.3.1 What the Old Adventure Map Looked Like

Before the redesign, the gamification section was a plain grid of buttons with project names. There was no visual hierarchy, no sense of progression, no rewards feedback, and nothing that made it feel like a "game." It was functionally correct but completely un-motivating.

4.3.2 The New Design Philosophy

The redesign was inspired by mobile games like Duolingo, Angry Birds, and Candy Crush — which all use a **visual level map** where the player can see where they are, where they've been, and where they're going. This kind of map creates a sense of journey, accomplishment, and anticipation.

We built an **Island Adventure Map** where each project level is represented as a floating island with its own unique image, and the levels are organized into **themed worlds**.

4.3.3 World and Island Structure

The levels are divided into two journeys, each with multiple themed worlds:

Arduino Journey:

World	World Name	Theme	Projects
1	Circuit Basics	Beginner	LED Blink, RGB LED, Buzzer, Potentiometer, LDR
2	Signal Control	Intermediate	Servo Motor, LED Strip, Button Debounce, Temperature Sensor
3	Machines & Sensors	Advanced	DC Motor
4	Smart Sensing	Expert	Push Button, Ultrasonic Sensor, DHT11 Sensor, LCD Display
5	Advanced Components	Master	Relay, OLED, NeoPixel, Keypad, Stepper Motor, MPU6050

ESP32 Journey:

World	World Name	Projects
1	ESP32 Basics	Blink, Analog Read, PWM
2	ESP32 IoT & WiFi	WiFi Scan, Web Server, HTTP Client
3	Advanced ESP32	MQTT, BLE, Deep Sleep
4	IoT Applications	OLED+WiFi, Sensor Cloud, Camera Stream

4.3.4 Island Images and Visual Design

Each project level has a **unique island image** stored in the /gamification_images/ public folder. These are PNG images with transparent backgrounds (so they blend into the map canvas):

```
// From: AdventureMapPage.jsx
function getIslandImage(slug: string) {
   const s = slug.toLowerCase();
   if (s === 'led-blink')    return '/gamification_images/island_led.png';
   if (s === 'rgb-led')      return '/gamification_images/island_rgb.png';
   if (s === 'buzzer')       return '/gamification_images/island_buzzer.png';
   if (s === 'servo-motor')  return '/gamification_images/island_servo.png';
   if (s === 'ultrasonic-sensor') return '/gamification_images/island_ultrasonic.png';
   // ... (one image per project)
   return '/gamification_images/island_led.png'; // fallback
}
```

The islands are rendered as <IslandNode> components — styled div containers with:

- The island PNG image as the main visual element
- A color palette (green, blue, orange, purple, cyan, red — cycling by index)
- A glowing box-shadow accent in the island's palette color
- A status overlay at the top: ✅ (completed), 🔓 (available/next), 🔒 (locked)

4.3.5 The Connecting Paths

Between islands, curved animated paths are rendered using SVG. The path from a completed island to the next available island glows and pulses with an animation. Paths to locked islands are grey and static.

This creates a visual "trail" showing the student's journey, like the path on a real game map.

4.3.6 XP and Badge Rewards

Each island (project level) has:

- A **step sequence:** Theory → Quiz → Guided Demo → Simulator.

- When all four steps are completed, the island is marked as "done."
- The student earns **XP points** for each completed step (e.g. 25 XP for Theory, 25 XP for Quiz, 50 XP for Simulator).
- When the entire island is completed, a **badge trophy icon** appears on the completed island node and is added to the student's badge shelf (shown in the profile page).

```
// From: AdventureMapPage.jsx — tracking step completion
markAdventureStepComplete({
    classId,
    projectSlug: projectName,
    stepKey: 'read',    // 'read', 'quiz', 'demo', 'sim'
    stepOrder: 1        // Order for sequential checking
});
awardXP(25, 'Theory complete! 📖'); // Show XP gain notification
```

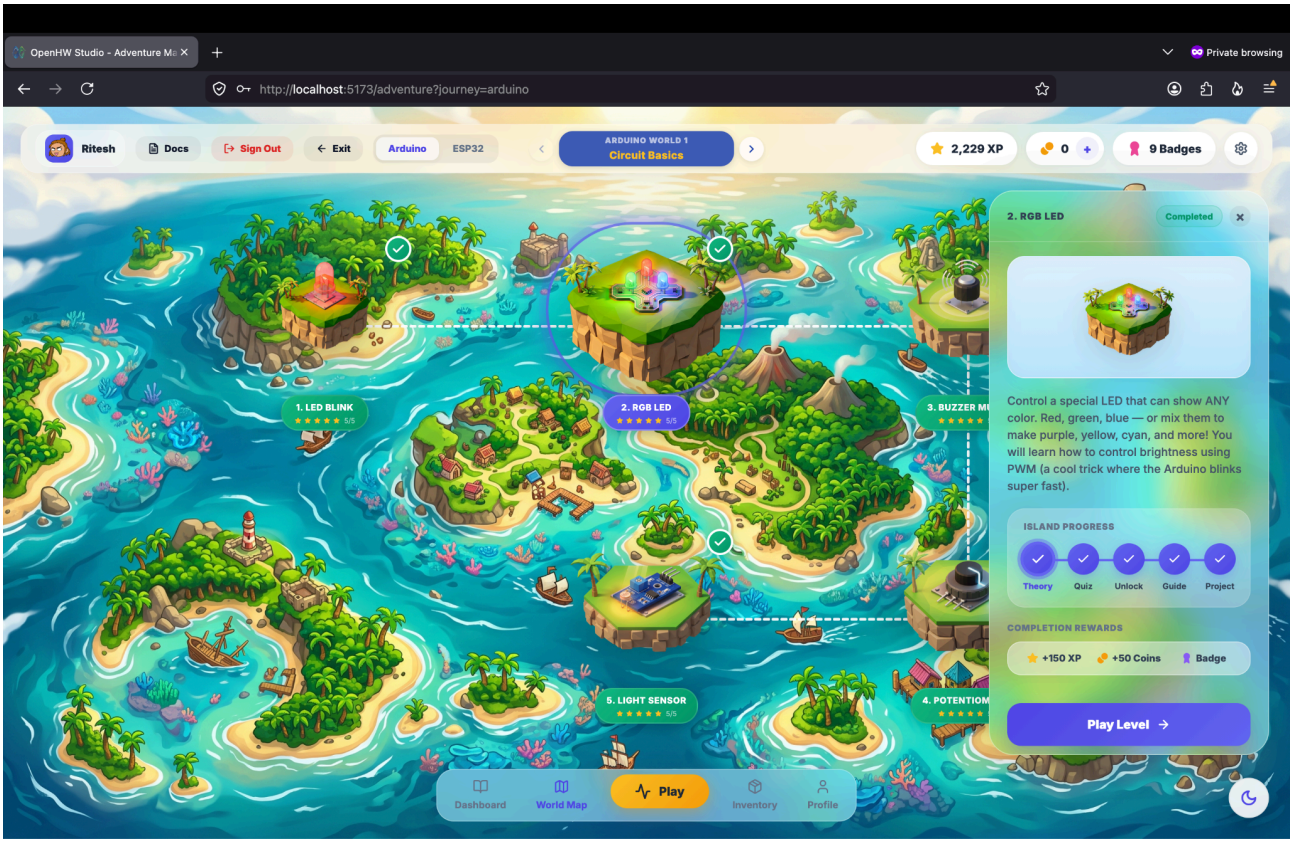
The gamification context (GamificationContext) tracks the total XP and coin count, persisting it in the backend via API calls.

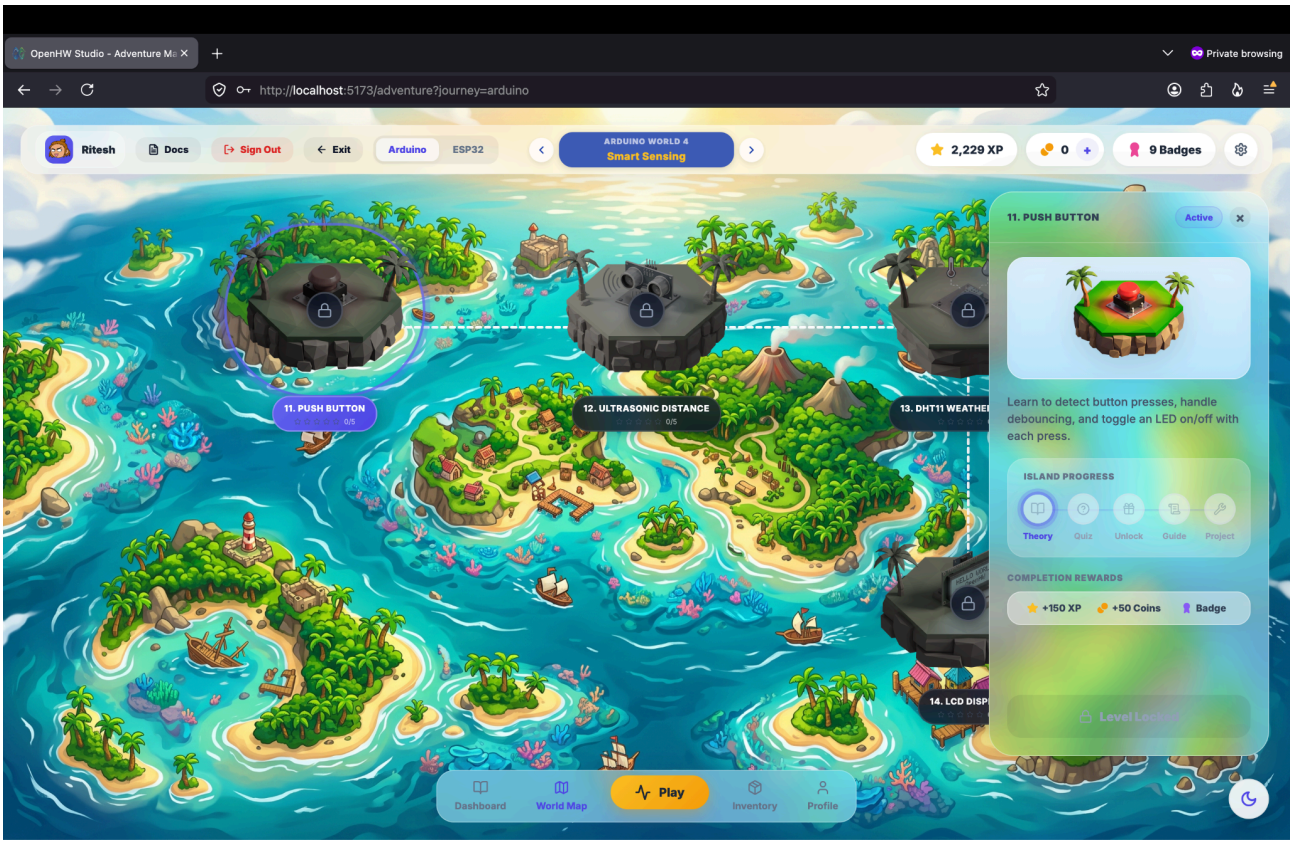
4.3.7 Island Hover Tooltips

Hovering over any island node (available or completed) shows a tooltip card containing:

- The project's name and difficulty tag
- A one-sentence description
- The XP reward for completing it
- The prerequisite island name (which world must be completed first)

Locked islands show a 🔒 icon and a "Complete [Prerequisite] first" message.


OpenHW Studio - Adventure Ma
Private browsing
http://localhost:5173/adventure?journey=arduino
Ritesh
Docs
Sign Out
Exit
Arduino
ESP32
ARDUINO WORLD 1
Circuit Basics
2,229 XP
0
9 Badges
1. LED BLINK
2. RGB LED
3. BUZZER MI
5. LIGHT SENSOR
4. POTENTIOM
2. RGB LED
Completed
Control a special LED that can show ANY color. Red, green, blue — or mix them to make purple, yellow, cyan, and more! You will learn how to control brightness using PWM (a cool trick where the Arduino blinks super fast).
ISLAND PROGRESS
Theory
Quiz
Unlock
Guide
Project
COMPLETION REWARDS
+150 XP
+50 Coins
Badge
Play Level
Dashboard
World Map
Play
Inventory
Profile


OpenHW Studio - Adventure Ma
Private browsing
http://localhost:5173/adventure?journey=arduino
Ritesh
Docs
Sign Out
Exit
Arduino
ESP32
ARDUINO WORLD 4
Smart Sensing
2,229 XP
0
9 Badges
11. PUSH BUTTON
12. ULTRASONIC DISTANCE
13. DHT11 WEATHE
14. LCD DISP
11. PUSH BUTTON
Active
Learn to detect button presses, handle debouncing, and toggle an LED on/off with each press.
ISLAND PROGRESS
Theory
Quiz
Unlock
Guide
Project
COMPLETION REWARDS
+150 XP
+50 Coins
Badge
Dashboard
World Map
Play
Inventory
Profile

4.4 Visual Guided Projects

4.4.1 Before: Text-Only Instructions

Previously, the Guided Project page (ProjectGuidePage.jsx) showed students a text description of the project and embedded the simulator in an iframe. There were no visual guides, no step-by-step circuit diagrams, and no image hints. Students often had to guess how to wire the circuit based on the text description alone — especially challenging for beginners.

4.4.2 After: PNG Circuit Images Embedded in Steps

The guided project page was updated to include **circuit schematic/wiring diagram images** alongside the step-by-step text instructions.

For each project, there is now a PNG image stored in the public assets folder showing exactly how to wire the circuit. For example, for the "LED Blink" project:

- Image shows the Arduino Uno with a wire from Pin 13 → resistor → LED → GND.
- Each connection is highlighted with a colored wire illustration.

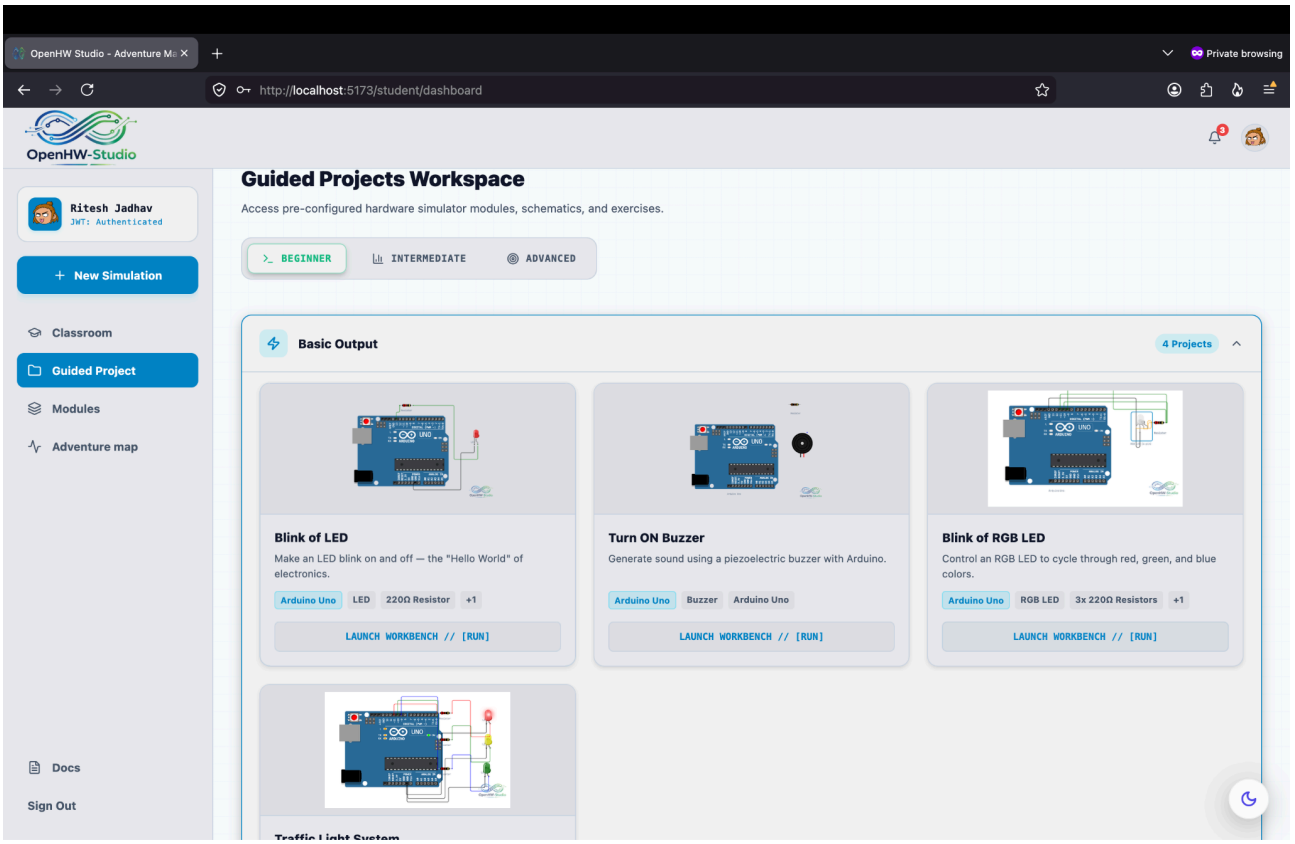
These images are loaded and displayed within the guided simulator frame:

```
// From: ProjectGuidePage.jsx
const canvasOnlyUrl = `/${projectName}/demo?canvas-only=1&readonly=1`;

// The iframe shows the embedded interactive simulator
<iframe
  ref={iframeRef}
  src={canvasOnlyUrl}
  style={{ width: '70vw', height: '70vh', border: 'none' }}
/>
```

The canvas-only=1 URL parameter tells the SimulatorPage to render *only* the canvas area (no code editor, no toolbar), so it fits cleanly inside the guided project layout.

The project metadata (loaded from guideProjectsIndex.json) includes a description and circuit connection steps. The description is shown as an explanation panel to the right of the embedded simulator, giving students the context they need while seeing the live circuit.

Why this matters: Research in educational technology consistently shows that visual-spatial learning aids (diagrams, images, animations) improve comprehension and retention of technical concepts. A student who can see *and* read how to connect an LED is far more likely to understand the concept than one reading text alone.



4.5 Transparent Circuit Image Export

4.5.1 The Problem with the Old Export

The original "Export as PNG" feature captured the entire simulator viewport — including the dark/blue background grid pattern, the toolbar buttons, the component palette panel on the left, and whatever parts of the canvas happened to be in view.

This was problematic for several reasons:

1. Students copying their circuit diagram into a school report got a dark-background image that looked out of place on white paper.
2. The exported image included UI chrome (buttons, panels) that were irrelevant to the circuit itself.
3. The exported area was not tightly cropped to the circuit — often most of the image was empty canvas.

4.5.2 What the New Export Does

The new PNG export:

1. **Auto-crops to the circuit:** Calculates the exact bounding box of all components and all wire waypoints. Adds a 60-pixel padding around the boundary.

2. **Transparent background:** The exported image has no background — only the components and wires.
3. **High resolution:** Renders at 1.5× the canvas resolution (Retina-quality without excessive file size).
4. **No UI chrome:** Only the circuit is captured. Toolbars, panels, and menus are excluded.

4.5.3 How It Works Technically

The export code is in SimulatorPage.jsx in a function called downloadPng():

Step 1: Calculate the bounding box

```
// Find the smallest rectangle that contains all components and wires
let minX = Infinity, minY = Infinity, maxX = -Infinity, maxY = -Infinity;

components.forEach((c) => {
  const reg = COMPONENT_REGISTRY[c.type];
  const bounds = reg?.BOUNDS || { x: 0, y: 0, w: c.w, h: c.h };
  minX = Math.min(minX, c.x + bounds.x);
  minY = Math.min(minY, c.y + bounds.y);
  maxX = Math.max(maxX, c.x + bounds.x + bounds.w);
  maxY = Math.max(maxY, c.y + bounds.y + bounds.h);
});

wires.forEach((w) => {
  (w.waypoints || []).forEach((wp) => {
    minX = Math.min(minX, wp.x);
    minY = Math.min(minY, wp.y);
    maxX = Math.max(maxX, wp.x);
    maxY = Math.max(maxY, wp.y);
  });
});

// Add 60px padding around the circuit
const PAD = 60;
minX -= PAD; minY -= PAD; maxX += PAD; maxY += PAD;
```

Step 2: Clone the circuit into a hidden iframe To avoid capturing the background, toolbars, or other UI elements, we clone *only* the circuit's DOM subtree into a completely separate hidden iframe with a transparent body:

```
const iframe = document.createElement('iframe');
Object.assign(iframe.style, {
  position: 'fixed',
  left: '-10000px',  // Completely off-screen — invisible to user
  top: '-10000px',
  width: actualW + 'px',
  height: actualH + 'px'
});
document.body.appendChild(iframe);

// Set the iframe body to transparent
const idoc = iframe.contentDocument;
idoc.write('<!DOCTYPE html><html><head></head>'
 + '<body style="margin:0;padding:0;background:transparent;"></body></html>');
```

Step 3: Capture with html2canvas using transparent background

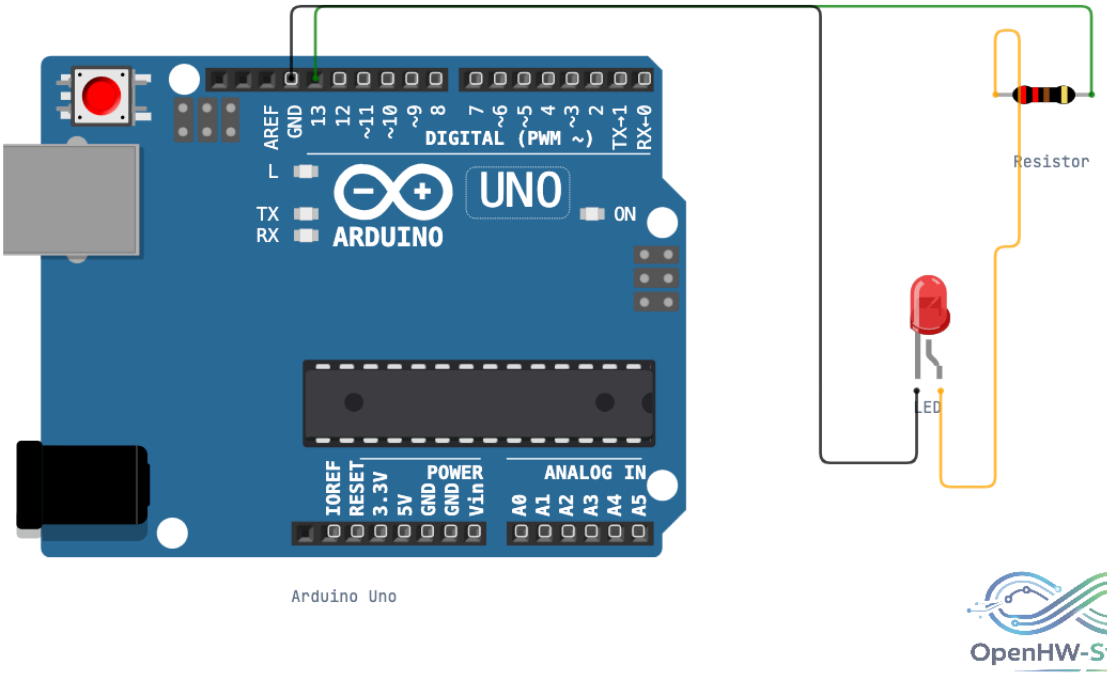
```
circuitCanvas = await html2canvas(idoc.body, {
    backgroundColor: null,  // This is the key setting — null means transparent
    scale: 1.5,             // 1.5× resolution for crisp text and graphics
    useCORS: true,          // Allow cross-origin images (component SVGs)
    allowTaint: false,
    width: actualW,
    height: actualH
});
```

Setting backgroundColor: null tells html2canvas to not paint any background before rendering. All pixels that have no content remain transparent (alpha = 0). When saved as a PNG, these transparent pixels stay transparent.

Step 4: Download the PNG

```
const blob = await new Promise(resolve => circuitCanvas.toBlob(resolve, 'image/png'));
const url  = URL.createObjectURL(blob);
const link = document.createElement('a');
link.href     = url;
link.download = `circuit_${board}.png`;
link.click();
setTimeout(() => URL.revokeObjectURL(url), 5000); // Clean up the object URL
```

The cleanup step is important: createObjectURL creates a temporary URL in browser memory. Without calling revokeObjectURL, these URLs accumulate and can cause small memory leaks if the student exports many times.



Arduino Uno

OpenHW-Studio

4.6 Assignment Notification System

4.6.1 Why This Feature Was Needed

Without notifications, students had to manually log in and check each class page to see if new assignments had been posted. Students who didn't check frequently would miss assignment due dates. This is especially a problem for platforms serving students who might log in only once every few days.

The solution was a real-time notification system — similar to how Google Classroom, Slack, or WhatsApp shows a red badge count when you have unread messages.

4.6.2 How It Works — End to End

Backend: Creating Notifications

When a teacher submits a new assignment via POST /api/classroom/assignments, the backend route handler does the following after saving the assignment:

```
// From: classroom assignments route handler (simplified)
async function createAssignment(req, res) {
   // 1. Save the assignment to the Assignments collection
   const assignment = await Assignment.create({
      classId: req.body.classId,
      title: req.body.title,
      dueDate: req.body.dueDate,
      projectSlug: req.body.projectSlug,
      createdBy: req.user._id
   });

   // 2. Find all students enrolled in this class
   const enrollment = await ClassEnrollment.find({
      classId: req.body.classId,
      role: 'student'
   });

   // 3. Create a notification record for EACH enrolled student
   const notifications = enrollment.map(e => ({
      userId: e.userId,
      type: 'new_assignment',
      title: `New Assignment: ${req.body.title}`,
      classId: req.body.classId,
      assignmentId: assignment._id,
      read: false,
      createdAt: new Date()
   }));

   await Notification.insertMany(notifications); // Bulk insert — one query for all students

   res.json({ ok: true, assignment });
}
```

Frontend: Displaying the Bell Icon

In the student's top navigation bar, we added a bell button that shows a red badge with the unread count:

```
// In StudentDashboard.jsx header
<button onClick={toggleNotificationsDropdown} className="relative">
  <Bell className="w-6 h-6" />

  {/* Red badge — only shown when unread count > 0 */}
  {unreadCount > 0 && (
    <span className="absolute -top-1 -right-1
                     bg-red-500 text-white text-xs
                     rounded-full min-w-[18px] h-[18px]
                     flex items-center justify-center
                     font-bold">
      {unreadCount}
    </span>
  )}
</button>
```

Frontend: The Notifications Dropdown

When the bell is clicked, a dropdown panel appears listing all unread notifications:

```
{isOpen && (
  <div className="notifications-dropdown">
    <div className="notifications-header">
      <span>Notifications</span>
      <button onClick={markAllRead}>Mark all as read</button>
    </div>

    {notifications.length === 0 ? (
      <div className="notifications-empty">You're all caught up! 🎉</div>
    ) : (
      notifications.map(notif => (
        <div
          key={notif._id}
          onClick={() => handleNotifClick(notif)}
          className={`notification-item ${notif.read ? 'read' : 'unread'}`}
        >
          <div className="notification-icon">📋</div>
          <div className="notification-body">
            <div className="notification-title">{notif.title}</div>
            <div className="notification-meta">
              {notif.className} • {timeAgo(notif.createdAt)}
            </div>
          </div>
          <ChevronRight className="notification-arrow" />
        </div>
      ))
    )}
  </div>
)}
```

Handling a Notification Click:

```
async function handleNotifClick(notification) {
  // 1. Mark this specific notification as read in the database
  await fetch(`/api/notifications/${notification._id}/read`, { method: 'PATCH' });

  // 2. Update the local state to reflect the read status (instant UI feedback)
  setNotifications(prev =>
    prev.map(n => n._id === notification._id ? { ...n, read: true } : n)
  );

  // 3. Recalculate the unread count
```

```
    setUnreadCount(prev => Math.max(0, prev - 1));

    // 4. Navigate directly to the assignment page
    navigate(`/student/class/${notification.classId}/assignment/${notification.assignmentId}`);
}
```

Fetching Notifications on Page Load:

```
useEffect(() => {
    async function loadNotifications() {
        const res = await fetch('/api/notifications/unread');
        const data = await res.json();
        setNotifications(data.notifications);
        setUnreadCount(data.notifications.filter(n => !n.read).length);
    }
    loadNotifications();
    // Refresh every 5 minutes in case new assignments were posted while logged in
    const interval = setInterval(loadNotifications, 5 * 60 * 1000);
    return () => clearInterval(interval);
}, []);
```

> 📸 **[SCREENSHOT — Notification Bell with Dropdown]** *Take a screenshot of the Student Dashboard with the notification bell clicked open, showing 2–3 assignment notifications in the dropdown. The red unread count badge should be visible on the bell icon.*

Chapter 5 — Summary of All Contributions

Contribution	Files Changed	What Was Achieved
MQ2 Gas Sensor	MQ2-gas-sensor/logic.ts, ui.tsx, manifest.json	Students can build gas alarm circuits with accurate AO/DO behavior
DHT22 Sensor	DHT-22/logic.ts, ui.tsx, manifest.json	Full 40-bit one-wire timing protocol simulation with sub-microsecond accuracy
Raindrop Module + Pad	Raindrop-module/, Raindrop-pad/, raindrop-shared-state.ts	Two-component pair sharing state via singleton, accurate voltage inversion model
PIR Motion Sensor	PIR-Motion-Sensor/logic.ts, ui.tsx, manifest.json	Three trigger modes, configurable hold time, draggable cone interaction
HC-SR04 Ultrasonic	openhw-hc-sr04/logic.ts, ui.tsx, manifest.json	Pulse protocol simulation, 200µs boot delay fix, draggable sonar cone
ESP32 (QEMU)	qemuRunner.js, networkProxy.js, SimulatorBridge.h, SimulatorWiFi.h	Full ESP32 with real WiFi via UART-bridged TCP proxy
Keypad Fix	openhw-membrane-keypad/logic.ts	Corrected matrix scan row/column mapping for

Contribution	Files Changed	What Was Achieved
		Keypad.h compatibility
Stepper Motor Fix	openhw-stepper-motor/logic.ts	Fixed coil sequence table and direction reversal timing
A4988 Fix	openhw-a4988/logic.ts	Rising-edge detection, correct DIR timing, correct microstepping table
Pico Performance	Simulation dispatch layer	10% fewer CPU cycles, 300ms → 50ms pin response time under heavy load
Classroom UI	TeacherDashboard.jsx, StudentDashboard.jsx, class detail pages	Modern sidebar + card-based layout, live status, assignment feed with badges
Auth + DiceBear	SignupPage.jsx, UserSignupPage.jsx, supporting pages	Split-screen layout, live avatar card preview, style + seed selection grid
Island Adventure Map	AdventureMapPage.jsx	Island nodes, world themes, animated paths, XP tracking, badge unlocks
Visual Guides	ProjectGuidePage.jsx, ProjectTheoryPage.jsx	Embedded circuit images alongside text instructions
Transparent PNG Export	SimulatorPage.jsx downloadPng()	Auto-crop, transparent background, no UI chrome
Notifications	Backend notification API + student dashboard bell UI	Real-time unread badges, dropdown list, click-to-navigate, mark as read

---

Chapter 6 — Conclusion

This internship was one of the most technically rich experiences I have had. Working across multiple layers of a real product — hardware physics simulation, CPU timing protocols, process management, network proxying, database design, and UI design — gave me a perspective on how professional engineering teams think about problems.

The most important lesson: Real hardware doesn't compromise. If the DHT22 protocol requires a 70-microsecond pulse for a 1 bit, a 69-microsecond pulse is an error. The Arduino library will reject the reading. Getting from "sort of works" to "always correct" required deep understanding of both the hardware spec and the emulator's clock model. That attention to precision is what separates a working educational tool from a frustrating one.

The most creative challenge: The ESP32 WiFi proxy architecture. The problem — "make a sandboxed virtual machine make real HTTPS requests" — had no off-the-shelf solution. We had to invent a binary framing protocol over UART, build a stateful connection pool, handle TLS on the

proxy side, and make it all transparent to the student's code. When it finally worked end-to-end, it was deeply satisfying.

The human impact: When a student builds a working gas alarm circuit in our simulator and watches the LED turn on as they drag the gas cloud, something clicks in their mind. The LED is not just a box on a screen — it is a consequence of physics they now understand. Making that moment of understanding happen, for students who might never hold a real Arduino, is why this platform matters.

I am deeply grateful to my mentors at FOSSEE, IIT Bombay for their guidance, code reviews, and encouragement throughout.

Chapter 7 — Future Work

7.1 Additional Sensor Components

The sensor library can be expanded with:

- **Sound Sensor (KY-037/KY-038):** Detects clapping, noise, or ambient sound levels. Would complete a range of common security-system project components.
- **DHT11 Sensor:** A simpler, lower-cost alternative to DHT22. Uses the same one-wire protocol but with lower precision (1°C and 1% RH resolution). Very common in beginner kits.
- **Gas Sensor Variants:** MQ-7 (carbon monoxide detection) and MQ-135 (general air quality / CO2).

7.2 3D Adventure Map Using Three.js

The current Adventure Map is built in 2D using SVG and CSS. A future version could use **Three.js** — a powerful JavaScript library for 3D rendering in the browser — to turn the map into a fully immersive 3D world:

- Students could walk a camera-character across floating 3D islands.
- Completing a level would trigger a 3D bridge building from the current island to the next, creating a physical sense of progress.
- The world sky and ocean could change color based on which "world theme" the student is currently in (green/forest for Beginner, blue/tech for Intermediate, etc.).
- This would feel like entering a real video game, making the learning experience far more memorable.

7.3 Auto-Graded Circuit Assignments

- Currently the Teacher have an Option for auto grading.
- It works by checking the json circuit submitted by the teacher.
- It’s not that accurate. We can optimize it more .

7.4 Offline PWA

Converting the simulator into an offline-capable **Progressive Web App (PWA)** would allow students to work without internet, especially important in rural regions of India where the FOSSEE initiative has a strong presence.

References

- Wokwi — Web-based Arduino Simulator
- TinkerCad Circuits
- Espressif ESP32 Arduino Core
- QEMU Open Source Machine Emulator
- avr8js — AVR Emulator in JavaScript
- DiceBear Open Source Avatars
- Three.js 3D Library for the Web
- html2canvas Library
- DHT22 Sensor Datasheet (Sparkfun)
- HC-SR04 Ultrasonic Sensor Datasheet
- MQ-2 Gas Sensor Reference
- HC-SR501 PIR Sensor Reference
- A4988 Stepper Motor Driver Datasheet (Pololu)
- FOSSEE, IIT Bombay

---

*Report prepared by **Ritesh Jadhav** | Internship at FOSSEE, IIT Bombay | July 2026*