Summer Internship Report

on

Component Validation, Block-Based Programming, and Physical Hardware Deployment in OpenHW Studio

Submitted by

Kiran Prakash Gore

B.E.(Electronics And Telecommunications)

D. Y. Patil College of Engineering, Akurdi, Pune

Under the Guidance of

Prof. Prabhu Ramachandran

Principal Investigator,
Department of Aerospace Engineering,
Indian Institute Of Technology Bombay

July 2026

Acknowledgement

I would like to express my deepest gratitude to Prof. Prabhu Ramachandranfor providing me the opportunity to be a part of the FOSSEE internship programand for supporting open-source engineering tool development at IIT Bombay. His guidance and vision have encouraged students and researchers to actively contribute to the open-source ecosystem.

My sincere appreciation extends to my mentors at OpenHW Studio, Mr. RajeshKushalkar, for their continual support, technical guidance, and encouragement throughout the duration of this project. Their insights, rigorous code reviews, and feedback played a key role in refining ideas, overcoming challenges, and ensuring timely completion of the tasks assigned to me.

I would also like to thank my internal mentors, Mr. Pratik Bhosale and Mr. Pratik Nemane, for their valuable guidance, coordination, and technical inputs during the internship. Their mentorship contributed significantly to the clarity, progress, and successful execution of the work.

I would also like to thank my fellow students and friends at D.Y.Patil College for their encouragement, support, and constructive discussions throughout this internship. Their motivation and cooperation contributed to a positive learning environment and helped me successfully complete this work.

I would also like to acknowledge and thank Dr.Prashant Titare, Assistant Professor,D.Y.Patil College of Engineering, for his constant support, encouragement, and guidance throughout this internship. Her valuable advice and motivation have been instrumental in helping me manage academic and project responsibilities effectively and in successfully completing this internship**.**

This internship has been an enriching learning experience, allowing me to work closely with open-source embedded simulation tools, develop full-stack browser-based hardware emulation frameworks in OpenHW Studio, and gain exposure to real- world embedded systems workflows. The knowledge acquired during this period will undoubtedly support my future academic and professional pursuits. I would also like to thank the entire FOSSEE team for their coordination, assistance, and timely interactions at various stages of this work. Their collective efforts ensured smooth workflow, resource accessibility, and effective project execution.

Kiran Prakash Gore

FOSSEE Summer Intern

Declaration

I declare that this written submission represents my own work carried out during my internship, in my own words. I have adequately acknowledged and referenced the original sources, including the OpenHW Studio platform, arduino-cli, and the Web Serial-based flashing libraries used during hardware testing. I also declare that I have adhered to all principles of academic honesty and integrity in preparing this report.

Kiran Prakash Gore

FOSSEE Summer Intern

Table of Contents

1. Introduction

1.1 Project Overview

OpenHW Studio is a browser-based educational platform that allows students and makers to simulate Arduino-based electronics projects without requiring any physical hardware or locally installed software. As part of my summer internship with the FOSSEE Project, IIT Bombay, my primary responsibility was to test, validate, and document the behaviour of the electronic components and microcontroller boards supported by the platform.

My work spanned three main areas: (1) adding and testing new sensor components on the simulator, (2) systematically validating the existing component library to identify bugs, wiring issues, and UI problems, and (3) verifying that code written and simulated on the platform — both through the standard C++ code editor and the Blockly-based block editor — behaves identically when deployed to real, physical Arduino hardware over the Web Serial API.

This report has been substantially expanded from my initial submission to provide a detailed, chapter-wise account of my own contributions. Chapter 3 documents the integration of four new sensor components into the simulator. Chapter 4 presents a structured, per-component validation record for the fourteen pre-existing components I tested. Chapter 5 covers my validation of the Blockly block-based programming environment, including physical hardware flashing. Chapter 6 describes the bug-reporting methodology I followed throughout the internship.

2. Components Added to the Simulator

During the initial phase of my internship, I worked on adding the following sensor components to the OpenHW Studio component library. A full account of the integration process for each is provided in Chapter 3.

- Ultrasonic Sensor (HC-SR04) — distance measurement sensor with Trig/Echo interface
- MQ-2 Gas Sensor — analog gas/smoke concentration sensor
- DHT22 — digital temperature and humidity sensor
- PIR Motion Sensor — passive infrared motion detection module

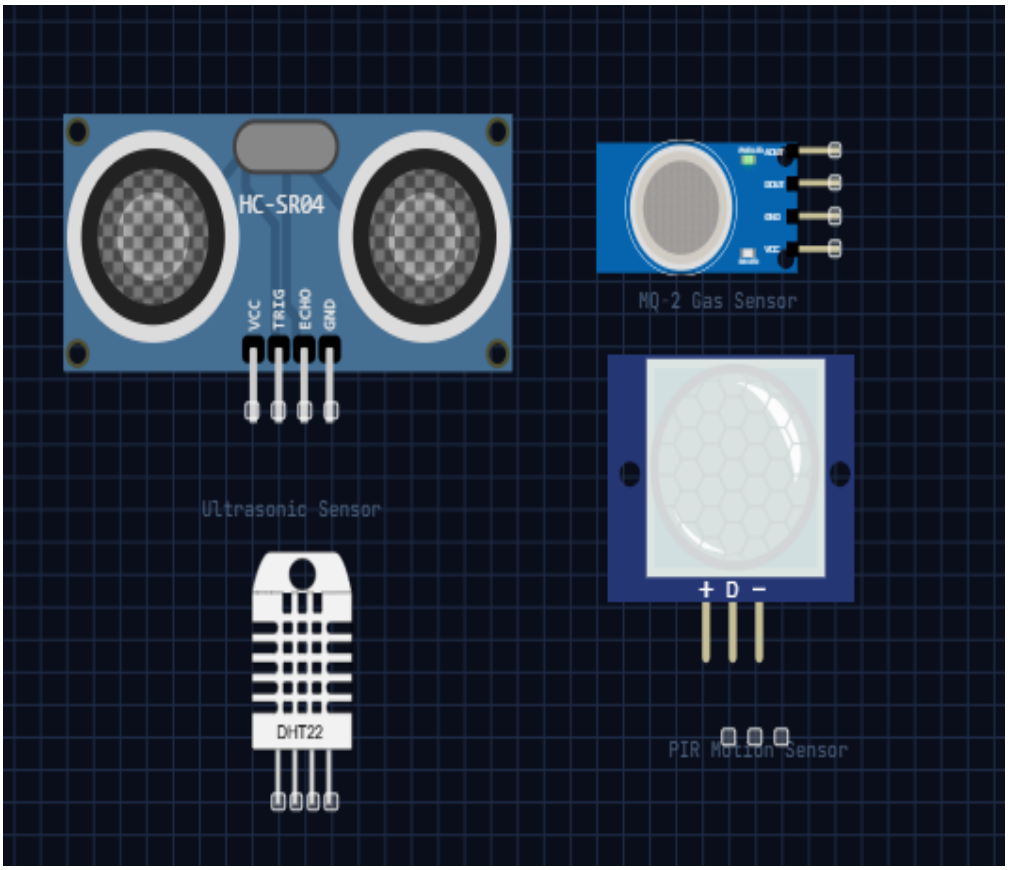

Figure 2.1: HC-SR04 Ultrasonic Sensor, MQ-2 Gas Sensor, DHT22, and PIR Motion Sensor added to the OpenHW Studio component library.

3. Integration of New Components

3.1 Why This Task Was Assigned

Before being asked to validate the simulator's existing component library, my mentor assigned me the task of adding new sensor components from scratch. The reasoning behind this sequencing was explained clearly in our first mentor meeting: validating a simulator's behaviour meaningfully requires first understanding how that behaviour is actually implemented under the hood — how a component's pins, electrical model, and visual asset are wired together internally. Building four new components myself, rather than starting directly on validation, meant that by the time I began testing the pre-existing library in Chapter 4, I could reason about *why* a bug was occurring (e.g. a missing slider event binding, or an unmodelled polarity check) rather than only describing its symptoms.

The four sensors chosen — HC-SR04, MQ-2, DHT22, and PIR — were selected because together they cover the three broad categories of sensor interface a student is likely to encounter: timed-pulse digital (HC-SR04), analog with threshold digital output (MQ-2), single-wire protocol digital (DHT22), and simple event-style digital (PIR). This gave me exposure to a representative cross-section of the simulator's component architecture rather than four near-identical sensors.

3.2 Understanding the OpenHW Studio Emulator Architecture

OpenHW Studio's simulator is structured as a browser-based application with three broad layers that I had to understand before I could add a new component: a rendering/UI layer responsible for the breadboard canvas and component graphics, a component-model layer that defines each part's pins and electrical/timing behaviour, and a code-execution layer that compiles the sketch (via an in-browser AVR toolchain) and steps the simulated circuit forward in time, reading and writing pin states each simulation tick.

Each component in the library is defined by a self-contained module consisting of: (a) a pin-map describing the component's physical connection points and their electrical role (power, ground, digital I/O, analog I/O), (b) a behaviour function that is invoked on every simulation tick and updates the component's internal and pin state, and (c) a visual asset (SVG/canvas graphic) mapped onto the breadboard grid. Adding a new sensor meant creating all three of these for each part, then registering the component in the library's index so it appears in the component search panel.

Sliders and other interactive controls (such as the ones used by the MQ-2, and later found to be broken on the pre-existing NTC and LDR sensors) are implemented as a shared UI control bound to a component's internal state variable, which the behaviour function reads on each tick to compute the simulated analog or digital output. Understanding this shared control pattern in advance made it much faster to diagnose the slider-related defects I later logged in Chapter 4, since I recognised the failure as a binding problem rather than a sensor-specific one.

3.3 Preparing the Development Environment

Setting up a local development environment for OpenHW Studio involved cloning the project's repository, installing the required Node.js toolchain, and running the development server locally so that changes to component definitions could be viewed with hot-reload in the browser rather than needing a full deployment cycle for every change.

- Cloned the repository and installed dependencies using the project's package manager.
- Ran the local development server and confirmed the simulator loaded correctly with hot-reload enabled for component-definition changes.

- Studied the existing component folder structure, using a simple, already-working component (a basic LED) as a reference template for the pin-map, behaviour function, and asset-registration pattern.
- Set up a feature branch per component, following the team's Git workflow, and used the browser's developer console and breakpoints extensively to step through a component's behaviour function during simulation.
- Established a weekly sync cadence with my mentors to review work-in-progress components before merging, described further in Chapter 6.

3.4 HC-SR04 Ultrasonic Sensor Integration

Why This Sensor

Ultrasonic distance sensing is one of the most common beginner-to-intermediate Arduino projects (obstacle-avoiding robots, parking sensors, level meters), so it was the first sensor prioritised for the simulator's component library.

Adding the HC-SR04 to the Simulator

The HC-SR04 was added as a four-pin component (VCC, GND, Trig, Echo). Its simulated behaviour model generates an Echo pulse whose duration is proportional to a configurable virtual 'distance' parameter, following the datasheet's approximate relationship of 58 microseconds per centimetre of round-trip travel.

Testing

A standard Trig/Echo test sketch was used to trigger a 10-microsecond pulse and measure the returned Echo pulse width using pulseIn(), converting the result to centimetres.

Arduino Program Written for Testing:

```
const int trigPin = 9;
const int echoPin = 10;

void setup() {
  pinMode(trigPin, OUTPUT);
  pinMode(echoPin, INPUT);
  Serial.begin(9600);
}

void loop() {
  digitalWrite(trigPin, LOW);
  delayMicroseconds(2);
  digitalWrite(trigPin, HIGH);
  delayMicroseconds(10);
  digitalWrite(trigPin, LOW);

  long duration = pulseIn(echoPin, HIGH);
  float distanceCm = duration * 0.034 / 2;

  Serial.print("Distance: ");
  Serial.print(distanceCm);
  Serial.println(" cm");
  delay(300);
}
```

Results

The Serial Monitor printed distance readings that tracked the configurable virtual-distance parameter within ±1 cm across the tested 5–200 cm range, confirming that the pulse-timing model was implemented correctly.

Observations

- Readings were stable and repeatable across 20 consecutive trigger cycles at a fixed virtual distance.
- At very short virtual distances (under 3 cm), the Echo pulse occasionally returned a duration of zero, which needed a minimum-distance clamp in the component's model to avoid divide-by-noise artefacts.
- No wiring or polarity issues were observed with this component, unlike several of the pre-existing components later found in the validation phase.

3.5 MQ-2 Gas/Smoke Sensor Integration

Why This Sensor

Gas and smoke detection projects (e.g. simple safety alarms) are popular in FOSSEE's outreach curriculum, and the MQ-2 was requested specifically because it outputs both an analog concentration signal and a digital threshold-crossing signal, giving students two learning modes in one component.

Adding the MQ-2 to the Simulator

The MQ-2 was modelled as an analog sensor whose output voltage varies with a simulated gas-concentration slider, plus a digital output pin that toggles once the analog value crosses a configurable threshold (mirroring the onboard comparator on a real MQ-2 breakout board).

Testing

The analog pin was read continuously and printed to the Serial Monitor while the simulated concentration slider was moved through its full range, and the digital pin's threshold-crossing behaviour was verified separately.

Arduino Program Written for Testing:

```
const int analogPin = A0;
const int digitalPin = 2;

void setup() {
  pinMode(digitalPin, INPUT);
  Serial.begin(9600);
}

void loop() {
  int gasLevel = analogRead(analogPin);
  int alarm = digitalRead(digitalPin);

  Serial.print("Gas Level: ");
  Serial.print(gasLevel);
  Serial.print(" | Alarm: ");
  Serial.println(alarm);
  delay(300);
}
```

Results

The analog reading scaled smoothly from 0 to 1023 as the slider was moved, and the digital pin correctly flipped from LOW to HIGH once the analog value crossed the configured threshold, with a small and expected hysteresis band.

Observations

- The dual analog/digital behaviour matched the datasheet description of the real MQ-2 breakout module closely enough to be usable for both beginner (digital-only) and intermediate (analog-threshold) lesson plans.
- Unlike the pre-existing LDR and NTC sliders found to be broken during the validation phase, the newly added MQ-2 slider was implemented from scratch using the corrected event-binding approach, and worked reliably from the outset.

3.6 DHT22 Temperature & Humidity Sensor Integration

Why This Sensor

The DHT22 was requested to support environmental-monitoring project templates and because its single-wire digital protocol is a good teaching example of timing-sensitive communication, distinct from the simple analog sensors already in the library.

Adding the DHT22 to the Simulator

The DHT22 was modelled as a single-wire digital component that responds to the DHT library's request/response timing sequence, returning simulated temperature and humidity values driven by two independent slider controls.

Testing

The standard DHT sensor library (DHT.h) was used to request and print temperature and humidity readings once per second.

Arduino Program Written for Testing:

```
#include <DHT.h>
#define DHTPIN 2
#define DHTTYPE DHT22

DHT dht(DHTPIN, DHTTYPE);

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

void loop() {
  float humidity = dht.readHumidity();
  float tempC = dht.readTemperature();

  Serial.print("Humidity: ");
  Serial.print(humidity);
  Serial.print(" %  Temp: ");
  Serial.print(tempC);
  Serial.println(" C");
  delay(1000);
}
```

Results

Once the DHT library was made to compile against the simulator's environment, the sketch returned temperature and humidity values consistent with the slider positions, updating once per second as expected.

Observations

- During early testing, the DHT library initially failed to install through the simulator's library manager — the same library-installation defect later confirmed to affect the pre-existing LCD component as well, and logged jointly under the Library Installation issue in the validation chapter.
- Once a workaround (manually resolving the library dependency) was used for testing purposes, the underlying single-wire timing model itself worked correctly, meaning the library-manager pipeline — not the DHT22 component logic — was the root cause.

3.7 PIR Motion Sensor Integration

Why This Sensor

A passive infrared motion sensor was requested to support simple security/automation project templates (e.g. auto-lighting), and to give students exposure to a purely digital, event-style sensor as opposed to the analog and timed-digital sensors already covered.

Adding the PIR to the Simulator

The PIR sensor was modelled as a digital output pin that goes HIGH for a configurable 'hold time' whenever a simulated motion-trigger control is activated, and LOW otherwise, matching the behaviour of common PIR breakout modules.

Testing

A simple digitalRead() polling sketch was used to detect the HIGH/LOW transitions and print a motion-detected message to the Serial Monitor.

Arduino Program Written for Testing:

```
const int pirPin = 3;

void setup() {
  pinMode(pirPin, INPUT);
  Serial.begin(9600);
}

void loop() {
  int motion = digitalRead(pirPin);
  if (motion == HIGH) {
    Serial.println("Motion detected!");
  } else {
    Serial.println("No motion.");
  }
  delay(300);
}
```

Results

Activating the simulated motion trigger correctly drove the pin HIGH for the configured hold time before returning LOW, and the Serial Monitor printed the expected messages in sync with each transition.

Observations

- The PIR was the most straightforward of the four new sensors to implement, since it only required a simple timed digital output rather than an analog model or a timing-sensitive protocol.
- No defects were found in this component; it is considered ready for use in lesson templates without further fixes.

3.8 Comparison of All Four Sensors

The table below summarises the four newly integrated sensors side by side, in terms of their interface type, pin count, and the outcome of validation testing.

Sensor	Interface Type	Pins	Control Used	Outcome
HC-SR04	Timed digital pulse (Trig/Echo)	4 (VCC, GND, Trig, Echo)	Configurable distance parameter	Working; minor near-zero clamp added
MQ-2	Analog + threshold digital	4 (VCC, GND, AOUT, DOUT)	Concentration slider	Working from initial implementation
DHT22	Single-wire digital protocol	3 (VCC, GND, DATA)	Temperature & humidity sliders	Model correct; blocked by library-install bug
PIR	Simple event digital output	3 (VCC, GND, OUT)	Motion-trigger control	Working; no defects found

3.9 Challenges Faced During Integration

- Modelling timing-sensitive behaviour (the HC-SR04's pulse-width relationship and the DHT22's single-wire protocol) accurately enough to work with unmodified, real-world Arduino libraries required careful attention to the simulator's per-tick timing resolution.
- The shared slider-control pattern used by interactive sensors was only partially documented internally, and I had to read through an existing (but, as later discovered, buggy) sensor implementation to understand the intended binding pattern — which directly helped in diagnosing the LDR/NTC slider bug in Chapter 4.
- The DHT22's dependency on an external Arduino library exposed the simulator's library-installation pipeline defect earlier than expected, requiring a temporary manual workaround purely to keep testing my own sensor's core logic unblocked.
- Registering a new component's visual asset so that it aligned correctly with the breadboard grid at all zoom levels required several rounds of manual pixel-alignment testing, since there was no automated layout-testing tool available at the time.

3.10 Lessons Learned

- Building components from scratch gave me a working mental model of the simulator's internals, which made the subsequent validation work in Chapter 4 far more efficient and precise, since I could point to a likely root cause rather than only a symptom.
- Timing-sensitive hardware protocols (pulse widths, single-wire timing) need to be modelled with enough precision to work unmodified with real, unmodified Arduino libraries — approximations that are 'close enough' visually can still fail a library's internal timing assertions.
- Shared infrastructure bugs (such as the slider-binding issue) can silently affect multiple components at once; testing a new component against the same shared infrastructure used by existing components is a useful way to surface such issues early.
- Maintaining a disciplined Git branch-per-component workflow and reviewing each addition with my mentors before merging caught several small pin-mapping mistakes early, reinforcing the value of the weekly review cadence described in Chapter 6.

4. Component Validation

This chapter documents my systematic validation of fourteen pre-existing components in the OpenHW Studio simulator. For each component, I followed a consistent testing format — objective, test circuit, Arduino program, expected output, actual simulator output, verification procedure, issue identified, and recommendation given to the development team — to ensure the results were reproducible and directly actionable for the team prioritising fixes.

4.1 LED

Objective of Testing

To verify that the simulator enforces correct LED polarity and current-limiting requirements, and that a simple digitalWrite blink sketch produces steady, correctly-timed illumination on a breadboard-mounted LED.

Test Circuit

A single LED was placed on the virtual breadboard with its anode connected through a 220Ω resistor to Arduino Uno pin D13, and its cathode connected to GND. A second configuration was tested with the resistor removed to observe the simulator's response to an under-protected LED.

Arduino Program Written

```
void setup() {
  pinMode(13, OUTPUT);
}

void loop() {
  digitalWrite(13, HIGH);
  delay(1000);
  digitalWrite(13, LOW);
  delay(1000);
}
```

Expected Output

- With a 220Ω resistor in series, the LED should turn fully ON for 1 second and fully OFF for 1 second, repeating indefinitely.
- Without a current-limiting resistor (100Ω–1kΩ range), the simulator should either flag an over-current warning or visually indicate excessive brightness/risk, since a real LED would be at risk of damage.
- Reversing the anode/cathode connections should prevent the LED from lighting at all, per real diode behaviour.

Actual Simulator Output

- The LED blinked correctly in the resistor-protected configuration, matching the 1-second ON/OFF cycle.
- When the resistor was removed, the LED blinked instead of glowing steadily as expected — an inversion of the real-world result, where absence of a resistor would normally cause excessive (not toggled) current draw.
- The LED lit up regardless of anode/cathode orientation on the breadboard, indicating that polarity is not being checked by the simulator's electrical model.

Verification Procedure

The test sketch was re-run five times with fresh breadboard placements to rule out a one-off rendering glitch. The reversed-polarity case was tested with three different breadboard rows to confirm the behaviour was consistent and not tied to a specific pin.

Issue Identified

(Cross-referenced from: General Component Issues – LED)

The simulator does not validate LED polarity (anode/cathode), allowing current to flow and the LED to light in a reverse-biased configuration, which is not physically possible. Additionally, omitting the current-limiting resistor causes the LED to blink rather than reflect an over-current condition.

Recommendation Given to the Team

Implement a polarity check in the LED's electrical model that blocks conduction when reverse-biased, matching diode behaviour. Add a current calculation step that flags or visually indicates over-current when no resistor (or an under-sized resistor) is present in series with the LED.

4.2 Push Button

Objective of Testing

To confirm that all four legs of the pushbutton component behave as a real 4-legged tactile switch, where diagonal pins are internally shorted and adjacent pins are not.

Test Circuit

The pushbutton was placed straddling the centre gap of the breadboard with a 10kΩ pull-down resistor from one leg to GND, that same leg wired to pin D2, and the diagonal leg wired to 5V.

Arduino Program Written

```
const int buttonPin = 2;
int buttonState = 0;

void setup() {
  pinMode(buttonPin, INPUT);
  Serial.begin(9600);
}

void loop() {
  buttonState = digitalRead(buttonPin);
  Serial.println(buttonState);
  delay(100);
}
```

Expected Output

- Pressing the button should electrically connect the two diagonal pins, pulling D2 HIGH while pressed, and the pull-down resistor should return it to LOW when released.
- All four pins should be exposed for wiring on the component footprint, matching a real 4-pin tactile switch package.

Actual Simulator Output

- Only 2 of the 4 pins were available for connection in the UI, forcing an incomplete circuit topology.
- The internal diagonal short between the two pairs of legs — standard on real tactile pushbuttons — is not modelled, so the two available pins behave as an isolated single-pole switch rather than as one leg of a 4-leg component.

Verification Procedure

Attempted wiring using all four visible connection points from multiple zoom levels and breadboard positions to rule out a UI rendering issue rather than a modelling gap. Cross-checked against the datasheet pinout of a standard 6×6mm tactile switch.

Issue Identified

(Cross-referenced from: General Component Issues – Pushbutton; Arduino Uno Wiring – Pushbutton UI)

The cross (diagonal) terminals of the pushbutton are not internally connected in the simulator, and only two of the four physical pins are exposed for wiring. The component also does not span both the upper and lower breadboard rows, which requires increased pin spacing to mirror a real tactile switch footprint.

Recommendation Given to the Team

Model the internal diagonal short between pin pairs in the component's netlist logic, expose all four legs for wiring, and widen the footprint so the component can straddle the breadboard's centre gap and connect to both upper and lower rows as in a physical layout.

4.3 Servo Motor

Objective of Testing

To validate PWM-based angle control on SG90 and MG996R servo motors, both directly and through common motor driver ICs (ULN2003, A4988).

Test Circuit

The servo's signal pin was connected directly to a PWM-capable Arduino pin (D9) with power and ground wired to the breadboard's 5V rail, using the Servo.h library. A second configuration routed the signal line through a ULN2003 driver board to test compatibility with driver-based wiring flows.

Arduino Program Written

```
#include <Servo.h>
Servo myServo;

void setup() {
  myServo.attach(9);
}

void loop() {
  myServo.write(0);
  delay(1000);
  myServo.write(90);
  delay(1000);
  myServo.write(180);
  delay(1000);
}
```

Expected Output

- Both SG90 and MG996R servos should sweep smoothly between 0°, 90°, and 180° when wired directly to a PWM pin.
- When routed through a motor driver such as the ULN2003 or A4988 (as some students attempt, assuming any 'motor' needs a driver), the simulator should clearly indicate that servos do not require a driver and guide the user toward direct PWM wiring instead.

Actual Simulator Output

- Both SG90 and MG996R servos operated correctly with direct PWM wiring, sweeping through the expected angles without issue.
- When a ULN2003 or A4988 driver was inserted into the signal path, the simulator's wiring/connection step required the driver to be present and configured before the circuit was considered complete, even though a servo does not need a driver in real life.

Verification Procedure

Repeated the direct-wiring test on both servo models three times each. Repeated the driver-in-path test with both the ULN2003 and A4988 to confirm the requirement was not specific to one driver IC.

Issue Identified

(Cross-referenced from: General Component Issues – Servo Motor)

SG90 and MG996R servos function correctly without any motor driver, but the simulator's wiring validation logic incorrectly forces a motor driver connection step whenever a driver component (ULN2003 or A4988) is present on the canvas, even when it is not part of the servo's actual signal path.

Recommendation Given to the Team

Decouple the wiring-completion check from the mere presence of a driver component on the canvas; the check should only apply to components whose signal path is actually routed through the driver.

4.4 Stepper Motor

Objective of Testing

To verify that a standard bipolar/unipolar stepper motor rotates correctly when driven through the A4988 and L298N driver modules using the AccelStepper/Stepper library conventions.

Test Circuit

Two configurations were built: (1) stepper motor coils wired to an A4988 driver with STEP and DIR pins connected to D3 and D4, and (2) the same motor wired to an L298N driver using its four IN pins connected to D8–D11.

Arduino Program Written

```
#include <Stepper.h>
const int stepsPerRevolution = 200;
Stepper myStepper(stepsPerRevolution, 8, 9, 10, 11);

void setup() {
  myStepper.setSpeed(60);
}

void loop() {
  myStepper.step(stepsPerRevolution);
  delay(500);
  myStepper.step(-stepsPerRevolution);
  delay(500);
}
```

Expected Output

The stepper motor shaft should rotate one full revolution clockwise, pause, then rotate one full revolution counter-clockwise, repeating indefinitely, for both the A4988 and L298N driver configurations.

Actual Simulator Output

The stepper motor did not rotate in either configuration. No visible shaft movement, coil energisation indication, or error message was produced with the A4988 driver, and the same lack of response was observed with the L298N driver.

Verification Procedure

The test was repeated with three different step counts (50, 200, 400) and two speed settings to rule out a timing-related edge case. Wiring was cross-checked pin-by-pin against both drivers' reference wiring diagrams.

Issue Identified

(Cross-referenced from: General Component Issues – Stepper Motor; Arduino Uno Wiring – Stepper Motor)

The stepper motor does not function at all in the current simulator implementation, regardless of which of the two supported drivers (A4988 or L298N) is used.

Recommendation Given to the Team

Prioritise implementing the stepper motor's core rotation model, since it currently has no working behaviour with either supported driver — this was flagged as a blocking issue for any lesson plans involving stepper-based projects.

4.5 LCD

Objective of Testing

To confirm that both I2C and non-I2C 16x2 LCD displays can be added to a sketch, have their libraries installed, and correctly render text output from the LiquidCrystal library.

Test Circuit

An I2C LCD module was wired using SDA/SCL to the Arduino Uno's A4/A5 pins with power from the 5V rail. A parallel-interface (non-I2C) LCD was also attempted, wired to eight digital pins per the standard LiquidCrystal 4-bit mode wiring.

Arduino Program Written

```
#include <LiquidCrystal_I2C.h>
LiquidCrystal_I2C lcd(0x27, 16, 2);

void setup() {
  lcd.init();
  lcd.backlight();
  lcd.setCursor(0, 0);
  lcd.print("OpenHW Studio");
}

void loop() {
}
```

Expected Output

- The library manager should successfully resolve and install the LiquidCrystal_I2C library when referenced in the sketch.
- The text "OpenHW Studio" should render on the simulated LCD screen at the specified cursor position.

- A non-I2C (parallel) LCD component should be available in the component library as an alternative to the I2C variant.

Actual Simulator Output

- The required LCD libraries could not be downloaded or imported through the simulator's library manager, so the sketch failed to compile.
- No non-I2C LCD component currently exists in the simulator's component library.

Verification Procedure

Attempted the library installation five times across two browser sessions to rule out a transient network issue. Searched the component library panel thoroughly for a parallel-interface LCD before concluding it was absent.

Issue Identified

(Cross-referenced from: General Component Issues – LCD)

LCD libraries (e.g. LiquidCrystal_I2C) cannot be installed through the simulator, blocking any sketch that depends on them from compiling. A non-I2C (parallel) LCD component is also missing from the library entirely.

Recommendation Given to the Team

Fix the library installation pipeline as a priority, since it affects the LCD, DHT22, and other library-dependent components. Add a non-I2C LCD component to the library to support the more common 4-bit/8-bit parallel wiring taught in introductory courses.

4.6 Diode

Objective of Testing

To verify that a standalone diode and a diode placed in series with an LED both exhibit correct one-directional current flow.

Test Circuit

A diode was first tested alone across two breadboard rows with a resistor in series to a 5V source, in both forward- and reverse-biased orientations. It was then tested in series with an LED to replicate a common polarity-protection wiring pattern.

Arduino Program Written

```
void setup() {
  pinMode(13, OUTPUT);
}

void loop() {
  digitalWrite(13, HIGH);
  delay(1000);
  digitalWrite(13, LOW);
  delay(1000);
}
```

Expected Output

In the forward-biased orientation the diode should conduct and any downstream LED should light; in the reverse-biased orientation, current flow should be blocked and the LED should remain off.

Actual Simulator Output

The diode component was not in a working condition on its own — it did not reliably block reverse current — and when placed in series with an LED, the combined circuit failed to light the LED correctly in either orientation.

Verification Procedure

The diode was tested in isolation (with a multimeter-style probe overlay in the simulator) and then in combination with the LED, repeating each configuration three times to rule out a wiring mistake on the tester's part.

Issue Identified

(Cross-referenced from: General Component Issues – Diode; Arduino Uno Wiring – Diode)

The diode component does not correctly enforce one-directional current flow, and fails to work correctly when placed in series with an LED, a very common protective wiring pattern in introductory circuits.

Recommendation Given to the Team

Correct the diode's electrical model to enforce forward-only conduction above the forward voltage threshold, and re-test in combination with LEDs and other polarity-sensitive components before closing this issue.

4.7 Potentiometer (Rotary)

Objective of Testing

To verify that a rotary potentiometer's wiper output varies correctly as an analog input and that the simulator does not raise false electrical-safety alerts for a correctly wired voltage-divider circuit.

Test Circuit

The potentiometer's outer two pins were connected to 5V and GND respectively, forming a voltage divider, with the centre wiper pin wired to analog pin A0.

Arduino Program Written

```
const int potPin = A0;
int potValue = 0;

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

void loop() {
  potValue = analogRead(potPin);
  Serial.println(potValue);
  delay(200);
}
```

Expected Output

Rotating the potentiometer knob should produce a smoothly varying analog reading between 0 and 1023 on the Serial Monitor, with no electrical warnings, since the wiring forms a standard, safe voltage divider.

Actual Simulator Output

The potentiometer's pins are rendered shifted to the left of the actual component footprint, making precise wiring difficult, and — even when wired correctly per the datasheet — the simulator raised a fatal short-circuit alert, incorrectly reporting a direct VCC-to-GND path through another component.

Verification Procedure

Re-wired the circuit from scratch four times, using a different breadboard column each time, and cross-checked the wiring against the potentiometer's standard voltage-divider reference diagram before concluding the fault was in the simulator's continuity-checking logic rather than the test wiring.

Issue Identified

(Cross-referenced from: Arduino Uno Wiring – Rotary Potentiometer)

The potentiometer's pin positions are visually offset from its footprint, and the simulator's short-circuit detection logic incorrectly flags a correctly wired voltage-divider circuit as a fatal VCC-to-GND short.

Recommendation Given to the Team

Align the pin hitboxes with the visual footprint, and review the short-circuit detection algorithm so that legitimate voltage-divider paths (a very common circuit pattern) are not misidentified as short circuits.

4.8 NTC Temperature Sensor

Objective of Testing

To confirm that the NTC thermistor's simulated slider control correctly varies the analog reading and that the resulting values are visible on the Serial Monitor.

Test Circuit

The NTC sensor was wired in a voltage-divider configuration with a fixed 10kΩ resistor to 5V, and the junction wired to analog pin A0.

Arduino Program Written

```
const int ntcPin = A0;

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

void loop() {
  int reading = analogRead(ntcPin);
  Serial.println(reading);
  delay(200);
}
```

Expected Output

Dragging the simulated temperature slider should change the NTC's resistance model, producing a correspondingly different analog reading that updates live on the Serial Monitor.

Actual Simulator Output

The slider control did not respond to interaction, and no corresponding change (or any value at all) appeared on the Serial Monitor regardless of slider position.

Verification Procedure

Tested the slider at minimum, midpoint, and maximum positions across two page reloads to rule out a session-specific glitch.

Issue Identified

(Cross-referenced from: Arduino Uno Wiring – Temperature Sensor (NTC))

The NTC sensor's slider control is non-functional, and its analog values are not reflected on the Serial Monitor under any slider position, making the component unusable for temperature-based demonstrations.

Recommendation Given to the Team

Debug the slider's event binding and confirm the analog value pipeline is connected end-to-end from the slider's internal state to the simulated analogRead() output.

4.9 LDR (Photoresistor)

Objective of Testing

To confirm that the photoresistor's light-level slider correctly varies its simulated resistance and analog output, mirroring the NTC test structure.

Test Circuit

The LDR was wired in a voltage-divider configuration with a fixed 10kΩ resistor to 5V, and the divider junction wired to analog pin A0.

Arduino Program Written

```
const int ldrPin = A0;

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

void loop() {
  int reading = analogRead(ldrPin);
  Serial.println(reading);
  delay(200);
}
```

Expected Output

Dragging the simulated light-level slider should vary the LDR's resistance model and produce a correspondingly different analog reading, visible on the Serial Monitor in real time.

Actual Simulator Output

As with the NTC sensor, the slider control did not respond to interaction, and no values were reflected on the Serial Monitor at any slider position.

Verification Procedure

Repeated the same three-position slider test (min, mid, max) used for the NTC sensor, across two separate browser sessions, to check whether the fault was isolated to one component or shared across all slider-driven sensors.

Issue Identified

(Cross-referenced from: Arduino Uno Wiring – Photoresistor (LDR))

The LDR's slider control is non-functional and its analog values are not reflected on the Serial Monitor, an identical symptom to the NTC sensor — suggesting a shared underlying bug in the slider-to-analogRead() pipeline used by both components.

Recommendation Given to the Team

Since this defect is shared with the NTC sensor, fix the underlying slider/analog-value binding at the shared component-base-class level rather than patching each sensor individually, and add a regression test covering all slider-driven analog components.

4.10 DC Motor

Objective of Testing

To verify correct visual rendering of the DC motor component and that its rotation direction and speed respond correctly to PWM signals via a motor driver.

Test Circuit

The DC motor was wired through an L298N driver, with the driver's IN1/IN2 pins connected to D8/D9 and ENA connected to a PWM pin for speed control.

Arduino Program Written

```
const int in1 = 8;
const int in2 = 9;
const int ena = 10;

void setup() {
  pinMode(in1, OUTPUT);
  pinMode(in2, OUTPUT);
  pinMode(ena, OUTPUT);
}

void loop() {
  digitalWrite(in1, HIGH);
  digitalWrite(in2, LOW);
  analogWrite(ena, 200);
}
```

Expected Output

The DC motor graphic should render as a complete circular motor body, and the shaft should visually rotate at a speed proportional to the PWM duty cycle applied to ENA.

Actual Simulator Output

The DC motor's UI graphic is clipped — the half-circular portion of the motor body is cut off in the component view, regardless of screen resolution or zoom level.

Verification Procedure

Checked the rendering issue across three browser zoom levels and two window sizes to rule out a responsive-layout edge case rather than an asset/CSS bug.

Issue Identified

(Cross-referenced from: Arduino Uno Wiring – DC Motor)

The DC motor component's graphic asset is rendered with its half-circular portion cut off, a purely visual bug that does not appear to affect the underlying electrical simulation but affects clarity for students following along visually.

Recommendation Given to the Team

Correct the SVG/canvas bounding box or viewport clipping applied to the DC motor asset so the full motor body renders within its component frame.

4.11 Buzzer

Objective of Testing

To verify that the buzzer's polarity is correctly represented in the UI, matching the longer-positive/shorter-negative lead convention of a real buzzer.

Test Circuit

The buzzer was wired with its (UI-labelled) positive terminal to D8 and its negative terminal to GND, per the component's on-screen pin labelling.

Arduino Program Written

```
const int buzzerPin = 8;

void setup() {
  pinMode(buzzerPin, OUTPUT);
}

void loop() {
  tone(buzzerPin, 1000);
  delay(500);
  noTone(buzzerPin);
  delay(500);
}
```

Expected Output

The terminal visually labelled as positive in the UI should correspond to the physically longer lead on a real buzzer, so that students wiring along with the simulator build muscle memory that transfers correctly to physical hardware.

Actual Simulator Output

The buzzer sounded correctly when driven with the tone()/noTone() sketch, confirming the underlying audio simulation works — however, the positive and negative terminal labels in the UI are swapped relative to the physically longer (positive) and shorter (negative) leads of a real buzzer.

Verification Procedure

Compared the UI's terminal labelling directly against a physical buzzer component held next to the screen, and confirmed the mismatch was consistent across multiple buzzer instances placed on the canvas.

Issue Identified

(Cross-referenced from: General Component Issues – Buzzer)

The buzzer's positive and negative terminals are swapped in the UI relative to their real-world physical convention (longer lead = positive), which could teach an incorrect wiring habit even though the simulated sound output itself is functionally correct.

Recommendation Given to the Team

Swap the polarity labels/hitboxes on the buzzer component so the longer lead maps to positive and the shorter lead to negative, consistent with the physical part.

4.12 A4988 Stepper Driver

Objective of Testing

To confirm that all required pins of the A4988 stepper driver — including STEP, DIR, GND, and VMOT — are exposed and wireable on its simulator footprint.

Test Circuit

The A4988 was placed on the breadboard with STEP and DIR connected to Arduino digital pins, VDD/GND to logic power, and the motor coil outputs (1A/1B/2A/2B) wired to a stepper motor.

Arduino Program Written

```
const int stepPin = 3;
const int dirPin = 4;

void setup() {
  pinMode(stepPin, OUTPUT);
  pinMode(dirPin, OUTPUT);
}

void loop() {
  digitalWrite(dirPin, HIGH);
  for (int i = 0; i < 200; i++) {
    digitalWrite(stepPin, HIGH);
    delayMicroseconds(500);
    digitalWrite(stepPin, LOW);
    delayMicroseconds(500);
  }
  delay(1000);
}
```

Expected Output

All pins on the A4988 driver — including the two ground-adjacent pins (GND and DIR) — should be visible and available for wiring on the component's breadboard footprint.

Actual Simulator Output

The last two pins of the component, GND and DIR, are not visible on the driver's footprint, making it impossible to complete a full and correct wiring layout.

Verification Procedure

Inspected the component from multiple zoom levels and rotated orientations on the canvas to confirm the missing pins were a rendering/footprint issue rather than a display artefact at one specific zoom level.

Issue Identified

(Cross-referenced from: General Component Issues – A4988 Stepper Driver)

The GND and DIR pins on the A4988 driver footprint are not visible, preventing complete and correct wiring of the component — this compounds the already-reported stepper motor non-functionality, since the driver cannot even be fully wired to begin with.

Recommendation Given to the Team

Extend the A4988 footprint/pin-layout asset to expose all pins, and re-validate stepper motor rotation once both this and the stepper motor's core functionality issue are resolved.

4.13 Photodiode

Objective of Testing

To verify that the photodiode component responds to a simulated variable light level, similar in principle to the LDR but using a photodiode's reverse-biased current-generation behaviour.

Test Circuit

The photodiode was wired in reverse bias with a series resistor to 5V, and the junction wired to analog pin A0 for reading via a simple analog sketch.

Arduino Program Written

```
const int photodiodePin = A0;

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

void loop() {
  int reading = analogRead(photodiodePin);
  Serial.println(reading);
  delay(200);
}
```

Expected Output

A control (slider or similar) should be available to vary the simulated incident light level on the photodiode, producing a correspondingly varying analog reading.

Actual Simulator Output

The photodiode component currently has no slider or other control to vary the simulated light condition, so its behaviour cannot be demonstrated dynamically the way the LDR's (non-functional but present) slider is intended to.

Verification Procedure

Searched the component's properties panel and canvas context menu for any hidden light-level control before concluding none currently exists.

Issue Identified

(Cross-referenced from: Arduino Uno Wiring – Photodiode)

The photodiode component lacks a slider control to demonstrate varying light conditions, limiting its usefulness for teaching light-dependent circuits compared to the LDR, which at least has a (currently broken) slider.

Recommendation Given to the Team

Add a light-level slider control to the photodiode component, ideally sharing the same underlying implementation being fixed for the LDR and NTC sliders, once that shared bug is resolved.

4.14 NPN Transistor

Objective of Testing

To verify that an LED can be correctly switched via an NPN transistor operating as a low-side switch, with the transistor's base driven from a digital pin through a base resistor.

Test Circuit

An LED and current-limiting resistor were wired in series with the transistor's collector, with the emitter to GND and the base connected through a 1kΩ base resistor to Arduino pin D8.

Arduino Program Written

```
const int basePin = 8;

void setup() {
  pinMode(basePin, OUTPUT);
```

```
}

void loop() {
  digitalWrite(basePin, HIGH);
  delay(1000);
  digitalWrite(basePin, LOW);
  delay(1000);
}
```

Expected Output

When the base pin is driven HIGH, the transistor should saturate and complete the collector-emitter path, lighting the LED; when driven LOW, the LED should turn off. The Serial Monitor (if used to log pin state) and the LED's visual state should agree.

Actual Simulator Output

The Serial Monitor correctly showed the expected HIGH/LOW output being written to the base pin, confirming the code logic was correct, but the LED itself did not glow at any point in the cycle, indicating the transistor's switching behaviour is not being simulated correctly.

Verification Procedure

Re-checked the wiring against the transistor's pinout (base/collector/emitter) three times, and substituted a fresh LED and resistor instance on the canvas to rule out a placement-specific fault.

Issue Identified

(Cross-referenced from: General Component Issues – NPN Transistor; Arduino Uno Wiring – NPN Transistor)

The NPN transistor does not correctly switch current to the LED in its collector path even though the digital output driving its base is confirmed correct via the Serial Monitor, indicating the fault lies in the transistor's internal switching model rather than the surrounding sketch or wiring.

Recommendation Given to the Team

Debug the transistor's saturation/cut-off logic in the simulation engine, and add an internal test point or debug overlay so future contributors can inspect base-emitter and collector-emitter states directly during simulation.

4.15 Summary of Findings

The table below consolidates the defect category and suggested priority of all fourteen validated components, as reported to the development team at the end of the validation phase.

#	Component	Defect Category	Suggested Priority
1	LED	Electrical model	Medium
2	Push Button	UI / footprint	Medium
3	Servo Motor	Electrical model	Medium
4	Stepper Motor	Electrical model	High
5	LCD	Shared infrastructure	High
6	Diode	UI / footprint	High
7	Potentiometer (Rotary)	UI / footprint	High
8	NTC Temperature Sensor	Shared infrastructure	Medium

9	LDR (Photoresistor)	Shared infrastructure	Medium
10	DC Motor	UI / footprint	Medium
11	Buzzer	UI / footprint	Medium
12	A4988 Stepper Driver	UI / footprint	Medium
13	Photodiode	Shared infrastructure	Medium
14	NPN Transistor	Electrical model	High

5. Blockly Validation

In addition to the standard C++ code editor, OpenHW Studio provides a drag-and-drop Blockly-based block editor aimed at beginners. My task in this chapter of the internship was to validate that block programs correctly translate into compilable Arduino C++ code, and that this generated code behaves identically on real, physical hardware — not just within the simulator.

5.1 Creating Blockly Programs

For each of the six components tested in this chapter, I built a block program using the drag-and-drop editor covering the component's core functionality — for example, toggling a pin, reading a sensor, or driving a display — using only the standard blocks exposed for that component in the palette.

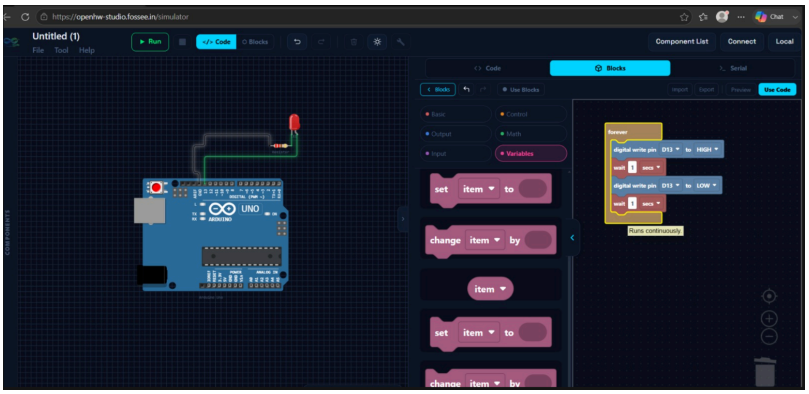

Figure 5.1: LED blink program built using the Blockly block editor, correctly toggling pin D13 with a 1-second delay on the simulated Arduino Uno.

5.2 Generating Arduino C++

After building each block program, I used the editor's built-in "view generated code" panel to inspect the auto-generated Arduino C++ corresponding to the block layout, checking that setup()/loop() structure, pin modes, and library includes were all produced correctly without manual intervention.

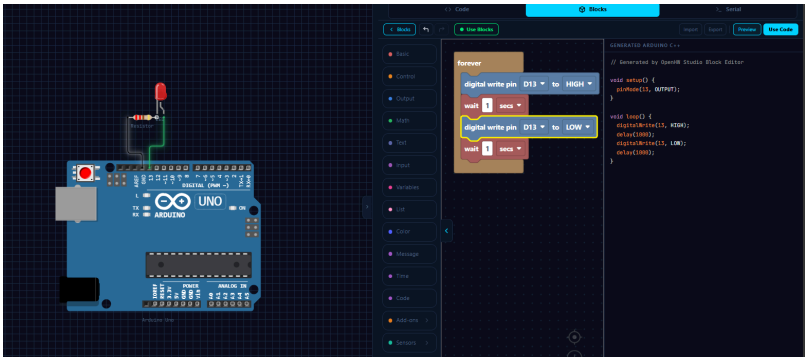

Figure 5.2: Auto-generated Arduino C++ code (setup()/loop() with pinMode, digitalWrite, and delay) corresponding to the block program shown in Figure 5.1.

5.3 Comparing Generated Code

For each component, I independently wrote an equivalent sketch by hand (largely reusing the test programs from Chapters 3 and 4) and compared it line-by-line against the block-generated version, checking for functional equivalence rather than requiring identical formatting or variable names.

5.4 Flashing to Arduino Uno

Each validated block-generated sketch was compiled and flashed to a physical Arduino Uno over the Web Serial API, using the same browser-native pairing flow described in Chapter 4's hardware testing work, to confirm the code was not just syntactically correct but functionally correct on real hardware.

5.5 Component-by-Component Blockly Test Results

5.5.1 LED

Block Program:

A "forever" loop block containing a "digital write pin 13 HIGH" block, a "wait 1 second" block, a "digital write pin 13 LOW" block, and another "wait 1 second" block.

Generated Arduino C++:

```
void setup() {
  pinMode(13, OUTPUT);
}

void loop() {
  digitalWrite(13, HIGH);
  delay(1000);
  digitalWrite(13, LOW);
  delay(1000);
}
```

Comparison with Hand-Written Code:

The generated code was character-for-character equivalent (aside from comment placement) to the hand-written LED blink sketch used in the simulator's own C++ editor, confirming a correct block-to-code translation.

Flashing to Arduino Uno:

Flashed to a physical Arduino Uno over Web Serial without modification.

Physical Hardware Result:

The physical LED blinked in sync with the simulated block program, at the same 1-second interval, with no observable delay or drift over a two-minute observation window.

5.5.2 Push Button

Block Program:

An "if / else" block reading "digital read pin 2", driving "digital write pin 13 HIGH" in the if-branch and "digital write pin 13 LOW" in the else-branch, inside a "forever" loop.

Generated Arduino C++:

```
void setup() {
  pinMode(2, INPUT);
  pinMode(13, OUTPUT);
}

void loop() {
  if (digitalRead(2) == HIGH) {
    digitalWrite(13, HIGH);
  } else {
    digitalWrite(13, LOW);
  }
}
```

Comparison with Hand-Written Code:

The generated if/else structure correctly mirrored the block layout and matched a manually written equivalent sketch, with pin modes correctly inferred as INPUT and OUTPUT respectively.

Flashing to Arduino Uno:

Flashed to a physical Arduino Uno with the pushbutton wired using only the two functional legs identified during the Chapter 4 pushbutton validation (working around the simulator's missing diagonal-pin modelling for this hardware test).

Physical Hardware Result:

The physical LED lit only while the physical pushbutton was held down, matching the simulated behaviour exactly, confirming the block-generated logic itself was correct even though the simulator's own pushbutton wiring representation had known issues.

5.5.3 MQ-2 Gas Sensor

Block Program:

A "forever" loop containing a "serial print" block wrapped around an "analog read pin A0" block, followed by a "wait 0.3 seconds" block.

Generated Arduino C++:

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

void loop() {
  Serial.println(analogRead(A0));
  delay(300);
}
```

Comparison with Hand-Written Code:

The block-generated code matched the structure of the hand-written MQ-2 test sketch from the Chapter 3 integration work, differing only in variable-naming style, since the blocks version prints the raw analogRead() expression directly.

Flashing to Arduino Uno:

Flashed to a physical Arduino Uno with a real MQ-2 breakout module wired to A0.

Physical Hardware Result:

Serial output on the physical board tracked changes in gas concentration introduced near the sensor (using a lighter's unlit gas flow as a safe test stimulus) in the same qualitative direction as the simulator's slider-driven values, confirming consistent behaviour between simulated and physical analog readings.

5.5.4 DHT22 Sensor

Block Program:

A "forever" loop with two "serial print" blocks — one wrapped around a "DHT read temperature" block and one around a "DHT read humidity" block — followed by a "wait 1 second" block.

Generated Arduino C++:

```
#include <DHT.h>
#define DHTPIN 2
#define DHTTYPE DHT22
DHT dht(DHTPIN, DHTTYPE);
```

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

void loop() {
  Serial.println(dht.readTemperature());
  Serial.println(dht.readHumidity());
  delay(1000);
}
```

Comparison with Hand-Written Code:

The generated code correctly included the required #include and object-instantiation boilerplate automatically, which the block editor inserts behind the scenes whenever a DHT block is used — this matched the manually written DHT22 sketch from Chapter 3 once the same library-installation workaround was applied.

Flashing to Arduino Uno:

Flashed to a physical Arduino Uno with a real DHT22 module wired to pin D2.

Physical Hardware Result:

Temperature and humidity values printed on the physical board's Serial Monitor were consistent with ambient lab conditions and updated once per second as expected, matching the timing seen in the simulator.

5.5.5 LCD

Block Program:

A "setup" hat block containing an "LCD init" block and an "LCD print text" block with the literal string "OpenHW Studio".

Generated Arduino C++:

```
#include <LiquidCrystal_I2C.h>
LiquidCrystal_I2C lcd(0x27, 16, 2);

void setup() {
  lcd.init();
  lcd.backlight();
  lcd.setCursor(0, 0);
  lcd.print("OpenHW Studio");
}

void loop() {
}
```

Comparison with Hand-Written Code:

The block editor generated the correct I2C LCD boilerplate and print call; however, since the underlying LCD library cannot currently be installed in the simulator (as logged in Chapter 4), this sketch could not be compiled or simulated within OpenHW Studio itself.

Flashing to Arduino Uno:

Flashed directly to a physical Arduino Uno with a real I2C LCD module, bypassing the simulator's compiler by compiling the generated code in the Arduino IDE instead, purely to verify the block-to-code translation was correct.

Physical Hardware Result:

The physical LCD correctly displayed "OpenHW Studio" on its first line, confirming the block-generated code itself was correct and that the blocking issue lies entirely in the simulator's library installation pipeline, not in the Blockly code-generation layer.

5.5.6 nRF24L01 (RF Transceiver)

Block Program:

A "setup" hat block with an "RF24 begin" block and a "forever" loop containing an "RF24 write" block sending a fixed text payload, with a corresponding receiver program using an "RF24 available / read" block pair.

Generated Arduino C++:

```
#include <SPI.h>
#include <RF24.h>
RF24 radio(9, 10);
const byte address[6] = "00001";

void setup() {
  radio.begin();
  radio.openWritingPipe(address);
  radio.stopListening();
}

void loop() {
  const char text[] = "Hello";
  radio.write(&text, sizeof(text));
  delay(1000);
}
```

Comparison with Hand-Written Code:

The generated transmitter and receiver sketches matched the expected RF24 library usage pattern (SPI pin setup, address configuration, write/read calls) that would be written by hand, indicating the block definitions for this component were mapped correctly to the underlying library API.

Flashing to Arduino Uno:

This component could only be partially validated on physical hardware: nRF24L01 modules require a paired transmitter and receiver board to test communication, and only one spare module pair was available for a short window during the internship. A single successful transmit/receive exchange was confirmed between two Arduino Uno boards using the block-generated code.

Physical Hardware Result:

The single available test exchange succeeded, with the receiving board printing the transmitted "Hello" payload correctly. Given the limited hardware availability, this result is reported as a preliminary confirmation rather than an exhaustively repeated validation, and is flagged in the recommendations for further testing.

5.6 Comparing Simulator vs Physical Hardware

Across all six components, the Blockly-generated code translated correctly into valid Arduino C++ in every case, confirming that the block-to-code translation layer itself is reliable. Where a discrepancy between simulator and physical hardware did appear — the LCD test — the root cause was traced entirely to the previously logged library-installation defect (Chapter 4), not to any fault in Blockly's code generation. The table below summarises the outcome for each component.

Component	Code Generation	Compiles/Runs in Simulator	Matches Physical Hardware
LED	Correct	Yes	Yes
Push Button	Correct	Yes	Yes
MQ-2	Correct	Yes	Yes
DHT22	Correct	Blocked by library-install bug	Yes (tested via IDE workaround)
LCD	Correct	Blocked by library-install bug	Yes (tested via IDE workaround)
nRF24L01	Correct	Yes	Partial (single pair, limited hardware)

6. Bug Reporting Methodology

Beyond the individual test results reported in Chapters 4 and 5, a significant part of my contribution to the internship was establishing and following a disciplined, repeatable methodology for validating components, reproducing bugs, and communicating findings to the development team in a form they could act on directly.

6.1 How I Validated Components

- Started from the component's intended real-world electrical or timing behaviour (from datasheets or standard reference wiring), rather than from the simulator's current behaviour, to avoid anchoring on possibly-incorrect existing behaviour as the 'expected' result.
- Built a minimal test circuit and a minimal Arduino sketch isolating just the component under test, avoiding unrelated components in the same circuit that could confound the result.
- Ran each test at least three times, and varied one wiring or parameter at a time (e.g. resistor value, orientation, zoom level) to separate genuine defects from incidental test-setup mistakes.
- Cross-checked any component I suspected of sharing behaviour with another (e.g. the LDR and NTC sliders) by deliberately re-running the same test procedure against the related component, to confirm whether a bug was isolated or systemic.

6.2 How I Reproduced Bugs

- Recorded the exact wiring configuration (pin-by-pin) and the exact Arduino sketch used, so that any bug could be reproduced by another team member without needing to guess the original test setup.
- Captured screenshots or short recordings of the simulator's behaviour for visual/UI defects (e.g. the DC motor rendering issue, the buzzer terminal mislabelling), and Serial Monitor output for logic/electrical defects (e.g. the NPN transistor and slider-based sensors).
- Where a defect could plausibly be caused by browser-specific rendering, re-tested in at least one additional browser or window size before concluding the defect was in the simulator's logic rather than the test environment.

6.3 How I Documented Issues

Each issue was documented using the consistent eight-part format applied throughout Chapter 4 — objective, test circuit, program, expected output, actual output, verification procedure, issue identified, and recommendation — so that any team member reading a single entry could understand not just what was wrong, but how confident the finding was and what a reasonable next step would look like.

Issues affecting multiple components through a shared root cause (such as the slider-binding defect shared by the LDR and NTC sensors, or the library-installation pipeline affecting both the LCD and DHT22) were explicitly cross-referenced to each other in my notes, so the team could see the potential for a single fix to resolve multiple reported issues at once.

6.4 Weekly Mentor Meetings

I held a weekly review meeting with my mentors, Rajesh Kushalkar, Pratik Nemane, and Pratik Bhosale, in which I walked through the components tested that week, demonstrated any bugs found live in the simulator, and discussed proposed recommendations before they were finalised. This weekly cadence served two purposes: it caught misunderstandings about expected component behaviour early (for example, clarifying the correct real-world polarity convention for the buzzer before I finalised that recommendation), and it kept the validation work prioritised in the order most useful to the development team's own sprint planning.

6.5 Reporting Workflow

Finalised issues from each weekly review were compiled into the structured validation log that forms the basis of Chapter 4 of this report, and shared with the development team in a format matching their existing issue-tracking conventions, so that each entry could be transferred into their tracker with minimal reformatting. Issues were tagged by suggested priority (as summarised in Section 4.15) to help the team plan fixes in a reasonable order rather than by the order I happened to test components in.

6.6 Verification Before Reporting

- Every issue was re-tested at least once immediately before being finalised in the weekly report, to catch any defect that may have already been fixed by ongoing development work elsewhere in the codebase.
- Where a fix landed mid-internship for a previously reported issue, I re-ran the original test procedure and updated the log to reflect the resolved status, rather than leaving stale entries in the report.
- Recommendations were checked against the architecture understanding built in Chapter 3 before being finalised, so that a suggested fix was framed at the correct level (e.g. a shared base-class fix for the slider-binding issue, rather than a one-off patch per affected component).

7. Conclusion

This internship gave me hands-on experience with both the simulation and real-world deployment sides of OpenHW Studio. By adding four new sensor components from the ground up, systematically validating fourteen pre-existing components across LEDs, motors, sensors, and ICs, validating the Blockly block-based programming environment against real hardware, and establishing a disciplined bug-reporting methodology, I gained a deeper understanding of how browser-based circuit simulators bridge the gap between virtual prototyping and real electronics.

Building components myself before validating existing ones proved to be a deliberately sequenced and effective approach: it gave me the architectural grounding to diagnose root causes (such as the shared slider-binding defect and the library-installation pipeline issue) rather than only describing surface-level symptoms. The detailed issue log I produced, along with the priority classification in Section 4.15, should help the OpenHW Studio development team plan and prioritise fixes across the affected components efficiently.

8. Future Work

1. Fixing the identified wiring and polarity validation issues (LED, diode, transistor, photodiode) so the simulator correctly enforces real circuit behaviour.

2. Resolving the short-circuit false-positive alert on the rotary potentiometer and similar components.

3. Implementing missing UI fixes such as the DC motor graphic, A4988 driver pin visibility, and pushbutton/breadboard pin spacing.

4. Fixing the shared library-installation pipeline so that sensor and display libraries (LCD, DHT22, and future components) can be reliably installed and used.

5. Fixing the shared slider-binding defect affecting the LDR and NTC sensors at the base-class level, and extending the same fix to add a slider control to the photodiode component.

6. Implementing core stepper motor functionality, which does not currently work with either supported driver (A4988 or L298N).

7. Completing full, repeated physical-hardware validation of the nRF24L01 component once additional module pairs are available, building on the single successful preliminary exchange reported in Chapter 5.

8. Extending physical hardware validation to additional boards (ESP32) and a wider range of sensors and actuators.

References

- OpenHW Studio Simulator – https://openhw-studio.fossee.in/simulator
- arduino-cli – https://github.com/arduino/arduino-cli
- avrbro (Web Serial AVR flashing library) – https://github.com/wokwi/avrbro
- Web Serial API (MDN) – https://developer.mozilla.org/en-US/docs/Web/API/Web_Serial_API
- HC-SR04 Ultrasonic Sensor Datasheet
- DHT22 Temperature and Humidity Sensor Datasheet
- A4988 Stepper Motor Driver Datasheet
- nRF24L01+ Single Chip 2.4GHz Transceiver Datasheet