Summer Internship Report

on

OpenHW Studio: Development of Interactive Wiring, Visual Block Programming, and Real-Time Hardware Simulation

Submitted by

Sharukhesh S

B.Tech in Computer Science and Engineering,
Sri Eshwar College of Engineering

Under the Guidance of

Prof. Prabhu Ramachandran
Principal Investigator, FOSSEE Project
Department of Aerospace Engineering,
Indian Institute of Technology Bombay

June 2026

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.

Sharukhesh S
Sri Eshwar College of Engineering

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.

Sharukhesh S
SRI ESHWAR COLLEGE OF ENGINEERING

Index

Chapter 1

Introduction

1.1 Background and Motivation

In recent years, the integration of hardware and software has become a fundamental pillar of engineering education. Platforms like Arduino have revolutionized the way students, hobbyists, and professionals approach embedded systems by abstracting complex micro-controller architectures into accessible, high-level interfaces. However, transitioning from theoretical knowledge to practical, hands-on implementation often poses significant challenges. The requirement for physical components, breadboards, jumper wires, and dedicated workspace can be a barrier for many learners, particularly in remote or resource-constrained environments.

OpenHW Studio was conceived to address this gap. It is an advanced educational platform developed to make learning and experimenting with electronics and programming more accessible, engaging, and cost-effective. By adopting a modular, plug-and-play digital approach, it removes the physical constraints and challenges associated with traditional wiring, component procurement, and hardware debugging. Users can connect virtual components seamlessly and intuitively, making the platform highly suitable for both beginners learning fundamental electronics concepts and experienced enthusiasts prototyping complex systems. The overarching motivation behind OpenHW Studio is to democratize hardware education through robust, open-source web technologies.

1.2 Problem Statement

Despite the availability of several circuit simulation tools in the market, many suffer from steep learning curves, outdated user interfaces, or limited integration between visual programming paradigms and actual code generation. Traditional simulators often feel clunky, with rigid wiring mechanisms that do not accurately reflect the physical prototyping experience. Furthermore, many existing solutions lack the extensible, open-

source architecture required to integrate modern web components seamlessly. There is a pressing need for a cohesive platform that bridges the gap between drag-and-drop visual programming (like Blockly) and real-world micro-controller simulation, all wrapped in a highly responsive, modern web interface.

1.3 Objectives of the Internship

The primary objective of my summer internship with the FOSSEE project at IIT Bombay was to significantly enhance the capabilities, stability, and user experience of OpenHW Studio. My core objectives were divided into the following key areas:

- **Wiring System Overhaul:** To design, implement, and optimize a highly interactive wiring system that accurately mimics real-world breadboard and jumper wire connections with dynamic routing.
- **Block Editor Integration:** To develop and integrate custom visual programming blocks using Google Blockly, ensuring that they seamlessly translate into syntactically correct Arduino C++ code.
- **Component Development:** To build and integrate various electronic component modules (sensors, actuators, displays) into the simulator ecosystem.
- **User Interface and Experience (UI/UX):** To refine the platform's interface, introducing features such as dark mode compatibility, automatic power-rail snapping, and comprehensive Bill of Materials (BOM) generation.
- **Backend and State Management:** To improve the overall stability of the application by addressing backend synchronization issues, optimizing React state management, and refining database schemas.

1.4 Methodology and Approach

The development methodology adopted during the internship was heavily rooted in Agile software engineering practices, specifically tailored for open-source development. I worked in iterative sprints, actively engaging with my mentors and the OpenHW Studio development team for code reviews, architectural discussions, and continuous integration. The approach involved extensive research into graphing algorithms for wire routing, studying the internal Abstract Syntax Tree (AST) of the Blockly library, and leveraging modern React hooks for performant UI rendering. Extensive unit and integration testing were conducted at each stage to ensure that the generated Arduino code matched the simulated behavior accurately.

1.5 Report Structure

This report is structured to provide a comprehensive overview of my contributions. Chapter 2 details the System Architecture and Technologies Used. Chapter 3 dives deep into the Wiring System Development, covering routing algorithms and performance optimizations. Chapter 4 explores the Block Editor and Visual Programming aspects, detailing custom block implementations and code generation. Chapter 5 discusses User Interface Enhancements and System Improvements. Finally, Chapters 6 and 7 present the Conclusion and Future Work, outlining the broader impact of these contributions and potential avenues for subsequent development.

Chapter 2

Literature Review and Related Work

2.1 Evolution of Electronics Simulation

The field of electronics simulation has evolved significantly over the past few decades. Historically, tools like SPICE (Simulation Program with Integrated Circuit Emphasis) dominated the landscape. Developed at UC Berkeley in the 1970s, SPICE provided rigorous mathematical models for analog circuit simulation. However, SPICE-based simulators were primarily designed for professional engineers and academics, featuring steep learning curves and complex netlist syntax that proved unwieldy for beginners and educational purposes.

As microcontroller-based embedded systems gained prominence in the 2000s, a paradigm shift occurred. The focus expanded from purely analog continuous-time simulation to discrete-time digital and mixed-signal simulation. Proteus Design Suite by Labcenter Electronics emerged as a leading proprietary solution, offering extensive schematic capture and microcontroller simulation capabilities. Despite its robust feature set, Proteus remains a heavy, desktop-bound software suite with expensive licensing models, making it largely inaccessible to students in developing regions or hobbyists.

2.2 The Rise of Web-Based Emulators

The advent of HTML5, WebGL, and WebAssembly catalyzed a new generation of cloud-based engineering tools. Web-based simulators democratize access by removing the need for local installations and lowering hardware requirements. Platforms such as Tinkercad Circuits (by Autodesk) pioneered the accessible, browser-based hardware simulation space. Tinkercad's drag-and-drop interface and basic block programming proved immensely popular in primary and secondary education. However, its closed-source nature and limitations in handling advanced, multi-file C++ projects or custom component creation limit its utility in higher education and professional prototyping.

2.3 Wokwi and AVR8js

More recently, the open-source community has made significant strides in this domain. Wokwi, driven by the AVR8js engine, represents a state-of-the-art approach to web-based microcontroller simulation. AVR8js executes AVR machine code directly in the browser by simulating the CPU registers, SRAM, flash memory, and peripheral registers. This allows for cycle-accurate execution of Arduino binaries without requiring physical hardware. Wokwi's component library (`wokwi-elements`) provides the visual frontend for these simulations.

OpenHW Studio builds upon the foundation laid by tools like Wokwi. While Wokwi relies heavily on a JSON-based configuration file ('diagram.json') for circuit layout, OpenHW Studio abstracts this complexity behind a fully interactive, visual wiring system and a deeply integrated Blockly visual programming interface. This unique synthesis of visual programming and cycle-accurate hardware simulation places OpenHW Studio in a distinct, highly accessible niche.

2.4 Visual Programming Paradigms in Education

Visual programming languages (VPLs) have proven highly effective in lowering the barrier to entry for computer science education. Scratch, developed by the MIT Media Lab, demonstrated that block-based programming could teach computational thinking without the cognitive overhead of syntax errors. Google's Blockly library extended this paradigm by providing an extensible framework that can generate code in multiple languages (JavaScript, Python, C++, etc.).

In the context of embedded systems, integrating Blockly with C++ generation provides a powerful scaffolding tool. It allows novices to construct logic visually while implicitly learning control structures, variable scoping, and hardware-specific initialization routines. OpenHW Studio leverages this paradigm to bridge the gap between absolute beginners and intermediate developers transitioning to text-based coding.

Chapter 3

Architecture and Technologies Used

3.1 High-Level System Architecture

OpenHW Studio is engineered as a modern, decoupled web application. The architecture is primarily divided into a client-side frontend that handles the graphical user interface, block editor, and visual simulator, and a server-side backend that manages user sessions, project persistence, and database interactions. The separation of concerns ensures that the platform is scalable, maintainable, and highly responsive.

3.1.1 The Frontend Ecosystem

The frontend of OpenHW Studio is a Single Page Application (SPA) built using **React.js**. React's component-based architecture is instrumental in managing the complex states associated with hundreds of interacting virtual hardware components.

- **State Management:** The application utilizes Redux and React Context API to manage global states, such as the current active circuit, the list of placed components, and the mapping of wire connections. This prevents prop-drilling and ensures that state updates (like dragging a component) are efficiently synchronized across the canvas and the code editor.
- **Visual Workspace:** The interactive canvas where users build circuits is rendered using highly optimized SVG elements and HTML5 Canvas API, ensuring smooth frame rates even when handling complex wiring routes.
- **Blockly Integration:** Google's Blockly library is embedded into the React application, providing the visual programming interface. We customized Blockly's rendering engine to align with the platform's modern aesthetic.

3.1.2 The Backend Infrastructure

The backend relies on a **Node.js** runtime environment with an **Express.js** framework. This provides a lightweight, non-blocking architecture ideal for handling numerous concurrent API requests.

- **Database: MongoDB**, a NoSQL database, is used for data persistence. It stores user profiles, project metadata, circuit configurations (in JSON format), and generated code sketches. The schema-less nature of MongoDB is highly advantageous for storing varying structures of circuit topologies.
- **Authentication and API:** The platform uses JWT (JSON Web Tokens) for secure, stateless user authentication. RESTful API endpoints manage the CRUD (Create, Read, Update, Delete) operations for user projects.

3.2 The Simulation Engine (AVR8js)

At the heart of OpenHW Studio's hardware simulation is **AVR8js**, an open-source AVR 8-bit architecture simulator written in JavaScript. When a user compiles their visual block program, the generated Arduino C++ code is passed to an internal compiler (utilizing WebAssembly ports of the Arduino CLI), which produces a standard '.hex' file.

This '.hex' file is then fed directly into the AVR8js engine running in the browser. AVR8js simulates the micro-controller's CPU cycles, memory, timers, and GPIO (General Purpose Input/Output) pins in real-time. By connecting the SVG graphical elements (like an LED) to the simulated GPIO pins, the platform creates a highly realistic, interactive feedback loop without requiring any physical hardware.

Chapter 4

Hardware Simulation and Physics Modeling

4.1 Simulating CPU Execution Cycles

A major technological triumph of OpenHW Studio is its ability to simulate physical hardware purely in software. While standard compilers translate C++ into machine instructions for a specific architecture, executing those instructions inside a web browser requires an emulation layer. The AVR8js engine simulates the ATmega328p microcontroller architecture by interpreting these compiled machine instructions bit-by-bit.

The simulator maintains an internal array representing the microcontroller's Flash memory, SRAM, and EEPROM. During execution, it mimics the Instruction Fetch, Decode, and Execute pipeline of a real CPU. It tracks the Program Counter (PC) and CPU clock cycles with remarkable precision. This cycle-accurate behavior is critical for timing-dependent hardware, such as bit-banged communication protocols or precise PWM (Pulse Width Modulation) generation for servo motors.

4.2 Physics Modeling of Electronic Components

Simulating the microcontroller is only half the challenge; the surrounding physical components must also be accurately modeled. Every component in OpenHW Studio is backed by a JavaScript class that defines its physical and electrical properties.

4.2.1 Passive Components (Resistors and LEDs)

When simulating an LED in series with a resistor, the platform calculates the forward voltage drop and limits the current accordingly. The simulation engine reads the digital state of the connected GPIO pin. If the pin is driven `HIGH`, the engine updates the `fill` property of the LED's SVG graphic. The physics model also incorporates destruction

states; if a user connects an LED directly to a 5V supply without a current-limiting resistor, the simulator mimics an overcurrent event and visually "burns out" the LED.

4.2.2 Actuators (Servo and DC Motors)

Simulating moving parts like servo motors requires translating PWM electrical signals into mechanical rotation angles. A typical servo motor expects a 50Hz PWM signal where the pulse width (usually between 1ms and 2ms) dictates the shaft angle.

The simulator intercepts the timer interrupts generated by the AVR CPU and calculates the duty cycle of the PWM wave. It then applies a rotational transformation (‘transform: rotate(Xdeg)‘) to the SVG element representing the servo horn. To make the simulation realistic, a low-pass filter algorithm is applied to the rotation, simulating the mechanical inertia and transit time of a physical gear train.

4.2.3 Sensors (Ultrasonic and Temperature)

Inputs to the simulation, such as environmental temperature or object distance for an ultrasonic sensor, are handled via interactive UI sliders. When the user manipulates the slider for an HC-SR04 Ultrasonic Sensor, the simulator artificially generates an echo pulse whose width is mathematically proportional to the specified distance ($Time = \frac{Distance \times 2}{Speed\ of\ Sound}$). The simulated Arduino program measures this pulse just as it would in the physical world, creating a seamless loop between user input, hardware simulation, and software execution.

Chapter 5

Wiring System Development

5.1 Core Principles of the Wiring System

The wiring system is one of the most essential and technically complex features of OpenHW Studio. It enables users to create virtual electronic circuits by connecting electronic components through an interactive graphical interface. During my internship, I was primarily responsible for architecting from the ground up, developing, and enhancing this wiring system to provide a reliable, intuitive, and visually appealing user experience that closely mimics the tactile feedback of physical jumper wires on a breadboard. My work focused on ensuring that users could establish accurate connections between different components while maintaining smooth 60fps interaction and real-time visual feedback under the hood.

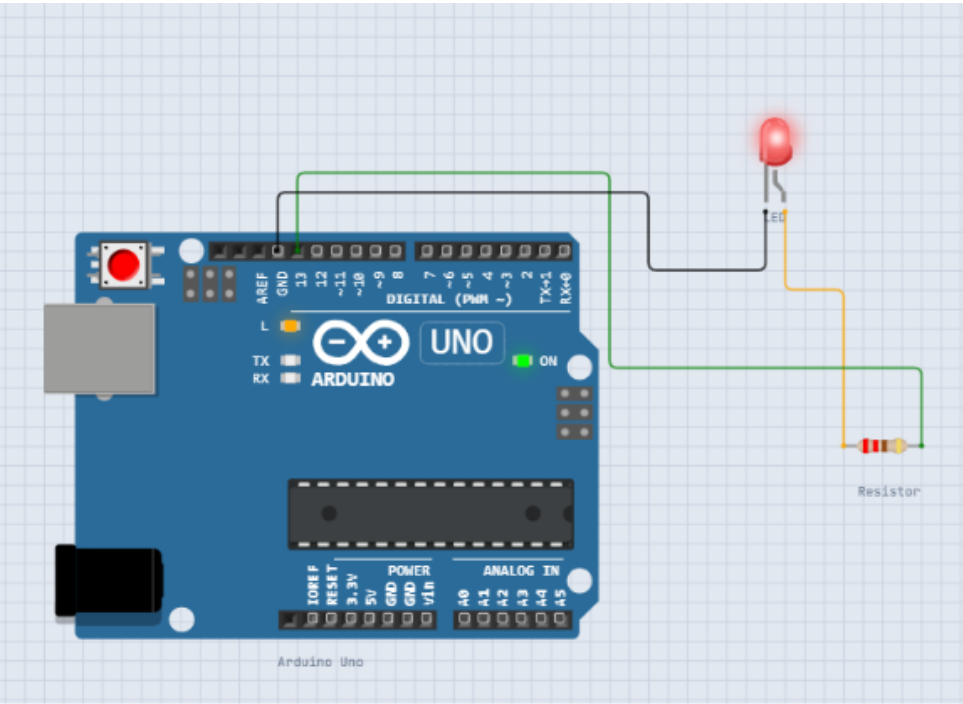


Figure 5.1: Overview of the wiring system in OpenHW Studio.

Unlike traditional legacy circuit simulation software where wires are often static vectors, the wiring system in OpenHW Studio is highly interactive, leveraging React's virtual DOM and HTML5 SVG rendering. Users can initiate a connection by clicking on a component pin (a graphical node), preview the flexible wire while dragging the cursor across the workspace, and complete the connection by releasing the mouse over a compatible destination pin. This entire workflow required an intricate implementation of pointer event handling, connection validation, Bezier curve math for wire rendering, and deep synchronization with the internal circuit graph representation.

5.2 Graph Representation and Data Structures

To accurately model the electronic circuits, the underlying data structure of the wiring system is represented as a directed multigraph, $G = (V, E)$, where:

- V (Vertices) represent the physical pins or terminals on electronic components (e.g., the 5V pin on an Arduino, or the Anode of an LED).
- E (Edges) represent the jumper wires connecting these pins.

Every time a wire is drawn, a new edge is added to this graph. The state is maintained in a global Redux store, ensuring that both the visual renderer and the AVR8js simulation engine have access to the exact same topology. We utilized an Adjacency List to optimize graph traversal, which is heavily required during the simulation phase to determine voltage levels across interconnected nodes.

Listing 5.1: Simplified Redux State for Circuit Graph

```
const circuitState = {
  components: {
    'led_1': {
      type: 'LED',
      position: { x: 100, y: 200 }
    },
    'arduino_1': {
      type: 'UNO',
      position: { x: 300, y: 150 }
    }
  },
  wires: [
    {
      id: 'wire_001',
      source: { componentId: 'arduino_1', pin: '13' },
      target: { componentId: 'led_1', pin: 'anode' },
      color: 'red'
    }
  ]
};
```

5.3 Wire Routing and Orthogonal Connection Logic

One of the primary responsibilities during my internship was designing and improving the wire routing logic for OpenHW Studio. The objective was to make the wiring process feel as natural as working with physical hardware while ensuring that every visual connection accurately represented the underlying circuit logic without resulting in a "spaghetti" of overlapping lines.

Whenever a user initiates a connection, the application records the source pin coordinates and immediately begins rendering a temporary preview wire. As the cursor moves across the workspace, the preview updates continuously. We employed a customized Orthogonal Routing Algorithm (often based on Manhattan distance calculations) combined with A* pathfinding concepts to ensure wires bend at clean 90-degree angles rather than direct, messy diagonal lines.

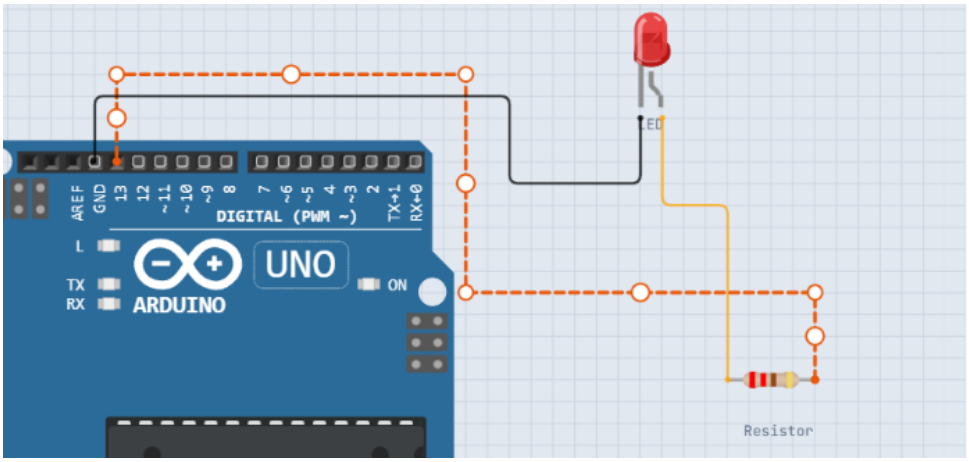


Figure 5.2: Wire routing and connection logic implemented in OpenHW Studio.

The routing algorithm dynamically calculates suitable wire paths based on the relative bounding boxes of connected components. If an obstacle (like a large LCD screen) is placed directly in the path of a wire, the algorithm computes penalty weights for traversing over the component and attempts to route the wire around the bounding box. Every wire displayed on the workspace remains synchronized with the simulator's logical representation.

5.4 Interactive Wiring Features and the Event Lifecycle

In addition to the routing system, I worked extensively on the React component lifecycle controlling wire connections. Every connection point (SVG circle) on a component was designed to listen to 'onPointerDown', 'onPointerMove', and 'onPointerUp' events. This unified pointer event model ensured compatibility across mouse, touch, and stylus inputs.

During wire creation, the application provides continuous visual feedback. If the user drags a wire from a digital output pin and hovers over an incompatible pin (e.g., another output pin leading to a short circuit), the system highlights the destination in red and disables the drop event.

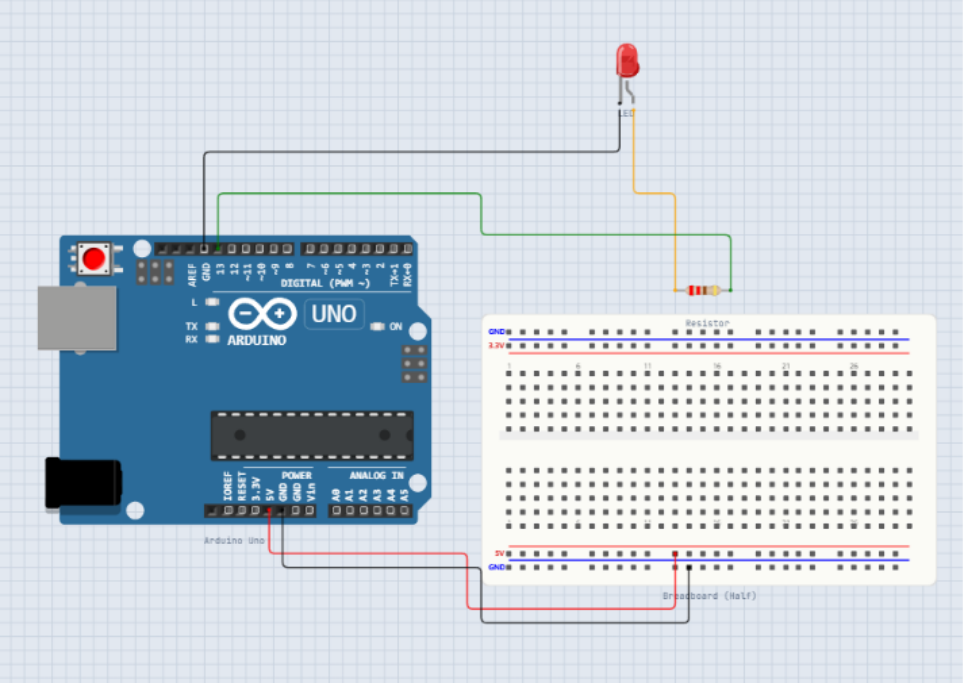


Figure 5.3: Interactive wire preview with real-time pin detection and connection validation.

5.5 Performance Optimization, Stability and Challenges

Throughout the internship, I continuously refined the wiring system to improve its stability and performance. Drawing hundreds of SVGs dynamically can trigger massive browser reflows and repaints, drastically reducing frame rates.

To optimize this, I implemented React ‘useMemo‘ and ‘useCallback‘ hooks to memoize the wire path calculations. Instead of redrawing every wire within the workspace whenever the cursor moved, the application isolates the render cycle of the active ”preview wire” into its own separate DOM layer. This significantly reduces unnecessary rendering operations and allows the workspace to remain responsive at 60 FPS even when working with larger and more complex circuit designs.

Developing the wiring system presented several technical challenges:

- **State Synchronization:** Maintaining synchronization between rapidly moving components and their associated wires required careful decoupling of local UI state from global Redux state.
- **Collision Detection:** Preventing wires from cleanly rendering when nodes overlapped required implementing a spatial hash grid for fast 2D collision detection.
- **Validation Edge Cases:** Handling edge cases like floating pins, ground loops,

and accidental short circuits natively in the browser before sending the graph to the simulation engine.

Extensive Jest unit testing across a wide variety of circuit layouts helped identify and resolve these edge cases, resulting in a stable, efficient, and user-friendly wiring system.

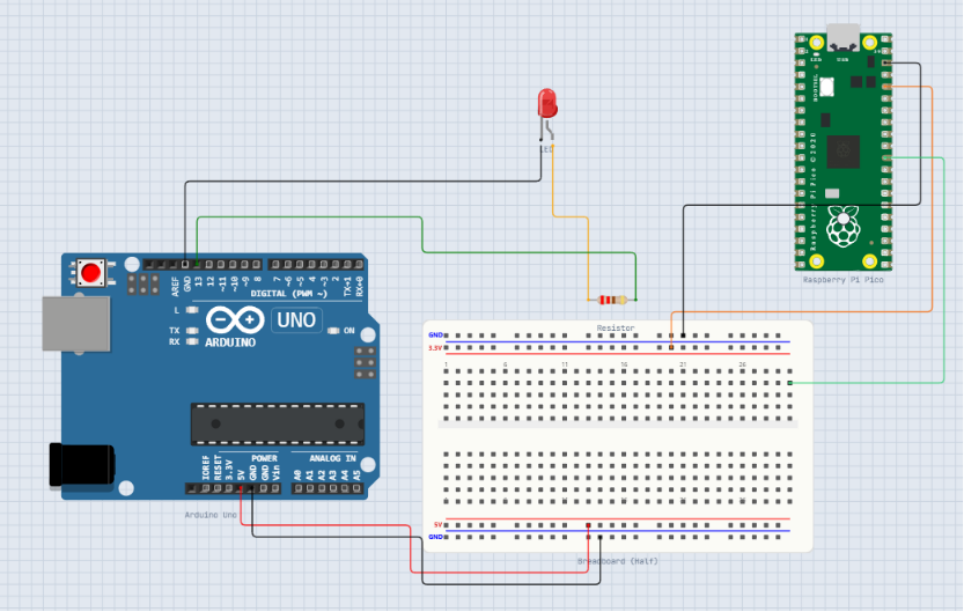


Figure 5.4: Optimized wiring system demonstrating stable routing, improved alignment, and consistent performance.

Chapter 6

Block Editor and Visual Programming

A significant portion of my internship was dedicated to the development and enhancement of the Block Editor in OpenHW Studio. The Block Editor provides a visual programming environment that enables users to create Arduino programs by assembling programmable blocks instead of writing code manually. This approach simplifies embedded programming for beginners while still allowing experienced users to rapidly prototype applications without constantly referencing syntax documentation.

My work involved implementing new programmable blocks, improving the editor's user interface, enhancing block interactions, and ensuring seamless integration between the visual programming environment and the Arduino code generation system. The completed implementation offers an intuitive drag-and-drop interface capable of generating syntactically correct, highly optimized Arduino C++ code in real time.

6.1 Block Editor Architecture and AST Generation

The Block Editor is built upon Google's Blockly library and is heavily customized to integrate natively into OpenHW Studio's React-based interface. Rather than relying on Blockly's default, somewhat rigid toolbox injection, the editor provides a completely redesigned user experience built with React components that better match the overall dark-themed, modern appearance of the platform.

6.1.1 Abstract Syntax Tree (AST) Integration

When a user drags and drops blocks, Blockly internally constructs an Abstract Syntax Tree (AST). Each block represents a node in the AST, and connections represent parent-child or sibling relationships. The real power of the editor comes from how we traverse

this AST. For every visual change, an event listener triggers a tree traversal that maps the Blockly AST directly to our custom Arduino code generator.

Listing 6.1: Blockly AST Traversal Listener

```
workspace.addChangeListener((event) => {
  if (event.type === Blockly.Events.BLOCK_MOVE ||
      event.type === Blockly.Events.BLOCK_CHANGE) {
    const code = ArduinoGenerator.workspaceToCode(workspace);
    dispatch(updatePreviewCode(code));
  }
});
```

6.1.2 Custom React Toolbox

A custom React sidebar organizes blocks into different categories that are displayed as interactive, stateful buttons. Selecting a category dynamically loads the corresponding collection of blocks into a flyout, allowing users to browse the available programming constructs without cluttering the workspace. This layout provides a cleaner interface and improves accessibility compared to the default Blockly toolbox, which often struggles with deep nested categories.

The editor supports both drag-and-drop interaction and a click-to-add mechanism. Users can drag blocks directly from the sidebar into the workspace, while a fallback option allows blocks to be inserted automatically into the center of the workspace with a single click—an essential accessibility feature for users on trackpads or mobile devices.

6.1.3 Real-time Code Preview and Board Awareness

Another critical feature of the editor is the live code preview. As users assemble blocks, the corresponding Arduino C++ code is generated automatically and displayed in real time with syntax highlighting (using libraries like PrismJS or Monaco Editor). This allows users to understand exactly how graphical programming translates into conventional Arduino code, serving as a powerful pedagogical tool.

The Block Editor also supports dynamic board awareness. Depending on the selected development board (e.g., Arduino Uno vs. ESP32), the available pin selections inside the blocks are updated automatically. For example, when Arduino Uno is selected, a "Digital Write" block displays digital pins D0–D13. If an ESP32 is chosen, the dropdown populates with ESP32 GPIO pins natively without requiring page reloads.

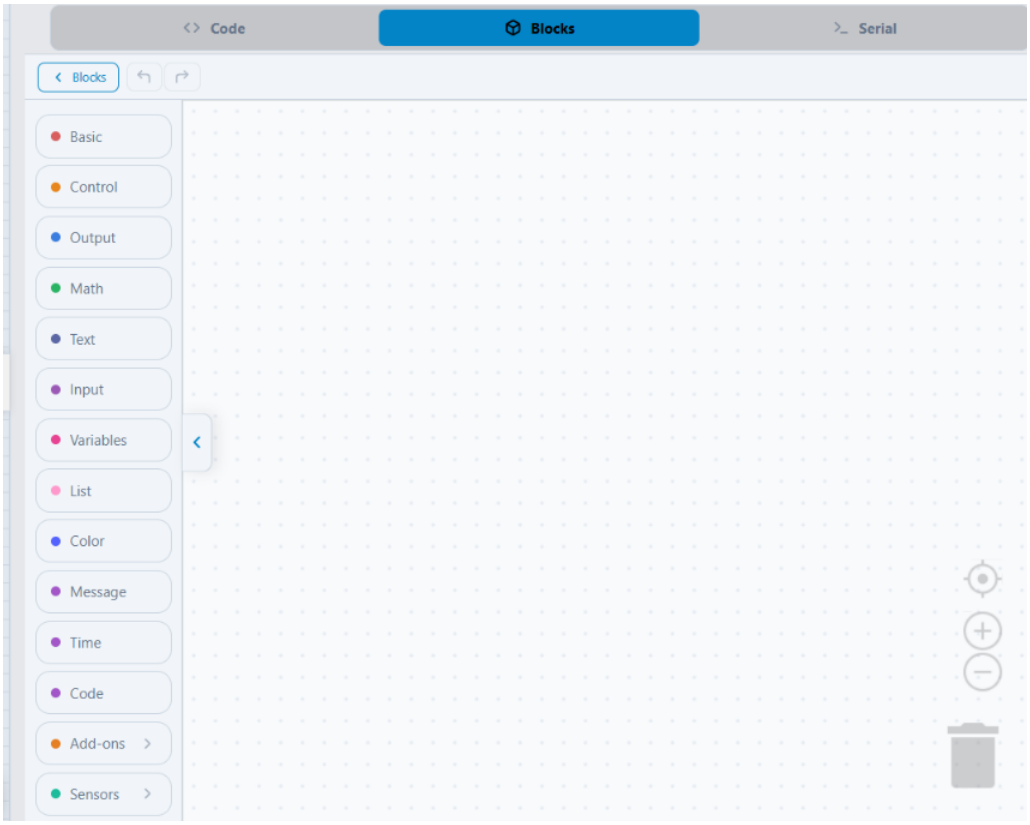


Figure 6.1: Customized Block Editor interface implemented in OpenHW Studio.

6.2 Block Design, JSON Registration, and Categories

One of my primary responsibilities during the internship was designing and implementing multiple programmable blocks and organizing them into logical categories. Each block was meticulously designed using Blockly's JSON-based definition format to provide clear functionality while maintaining strict type-checking on connections.

Listing 6.2: JSON Definition for a Custom LED Block

```
{
  "type": "control_led",
  "message0": "Turn LED on pin %1 to state %2",
  "args0": [
    {
      "type": "field_dropdown",
      "name": "PIN",
      "options": [["13", "13"], ["12", "12"]]
    },
    {
      "type": "field_dropdown",
      "name": "STATE",
      "options": [["HIGH", "HIGH"], ["LOW", "LOW"]]
    }
  ],
  "previousStatement": null,
  "nextStatement": null,
  "colour": 230
}
```

The editor supports three major structural categories of block shapes:

- **Hat Blocks (Event Listeners):** Used as entry points for programs (e.g., ‘setup()‘, ‘loop()‘).
- **Statement Blocks (Commands):** Perform operations such as writing to digital pins, controlling loops, or executing delays. They have both previous and next connections.
- **Value Blocks (Expressions):** Generate numerical, textual, or logical outputs (e.g., reading a sensor) that connect to the inputs of Statement Blocks.

To simplify navigation, the blocks are grouped into functional categories including Basic, Control, Input, Output, Math, Text, Variables, Lists, and Add-ons. During my internship, I implemented blocks for complex interactions like I2C LCD displays, PWM motor control, ultrasonic sensors, and external interrupts. Particular attention was given to ensuring that every block generated highly optimized, non-blocking Arduino code where applicable.

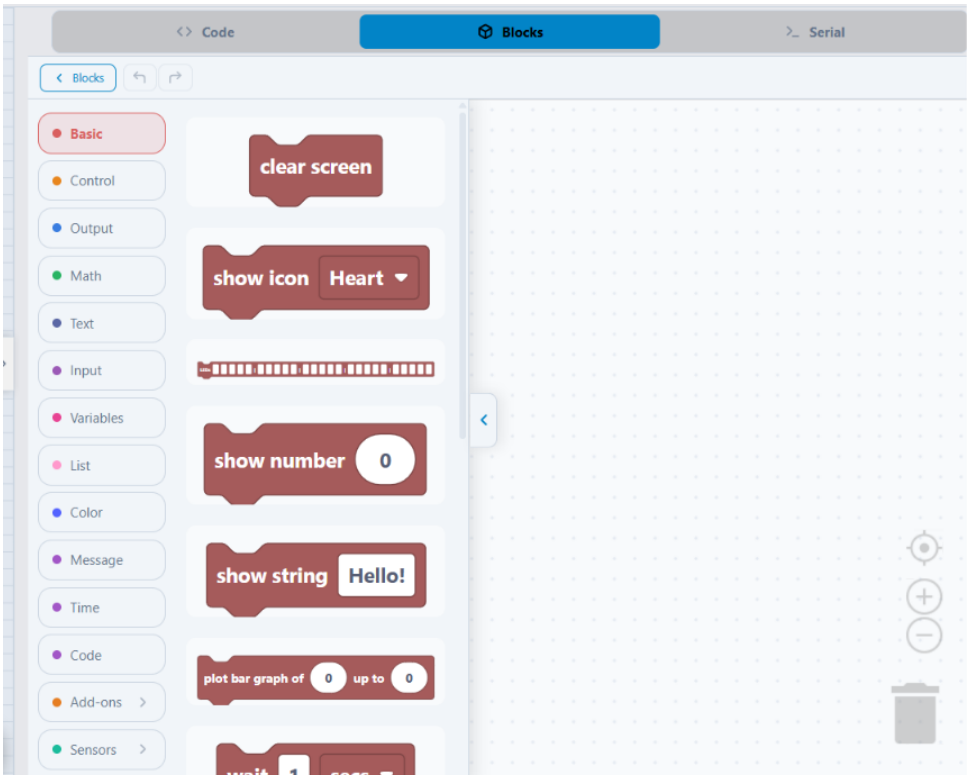
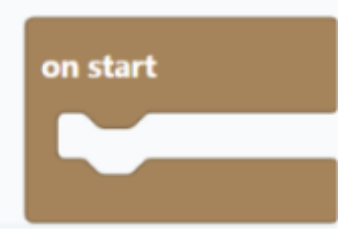


Figure 6.2: Basic and Control blocks implemented in the Block Editor.

6.3 Sample Blocks Implemented

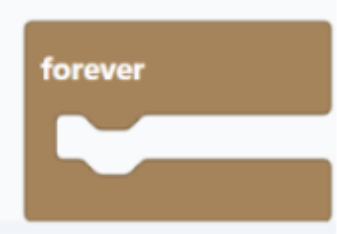
During the internship, I developed and integrated several Blockly-based programming blocks to simplify Arduino programming within OpenHW Studio. Each block was designed with custom user interfaces, validation logic, and Arduino C++ code generation support. Here are a few representative blocks implemented during the project.

1. On Start Block



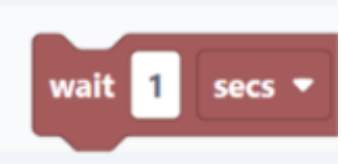
Acts as the entry point of every Blockly program and generates the Arduino `setup()` function. This block is used to initialize hardware components, configure pins, and prepare the board before execution begins. All statements placed inside this block are executed only once when the program starts.

2. Forever Block



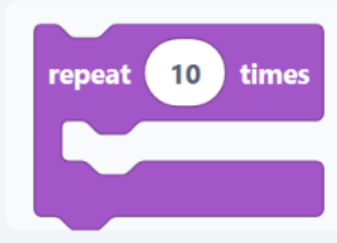
Represents the Arduino `loop()` function, which continuously executes after the setup phase is complete. Blocks placed inside this section run repeatedly in an infinite loop, making it suitable for continuously reading sensors, updating outputs, and performing repetitive tasks.

3. Wait Block



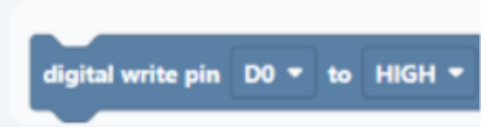
Introduces a programmable delay between operations by generating the Arduino `delay()` function. Users can specify the waiting time to control the execution sequence, making it useful for timing-based applications such as LED blinking and sensor sampling.

4. Repeat Block



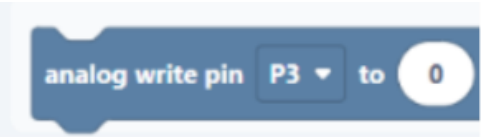
Implements a fixed-count loop that repeatedly executes the enclosed blocks for a user-defined number of iterations. It simplifies repetitive programming tasks by eliminating the need to manually write looping logic in Arduino C++.

5. Digital Write Block



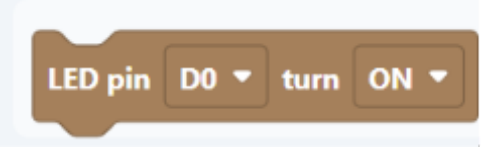
Sets the selected digital pin to either HIGH or LOW, allowing users to control digital output devices such as LEDs, relays, buzzers, and other electronic components. Users simply select the desired pin and output state through the block interface. The block automatically generates the appropriate Arduino code for the selected configuration, eliminating the need to write manual programming instructions. This simplifies digital output control and helps beginners understand hardware interaction through visual programming. It provides an intuitive and efficient way to operate digital devices within the Blockly environment.

6. Analog Write Block



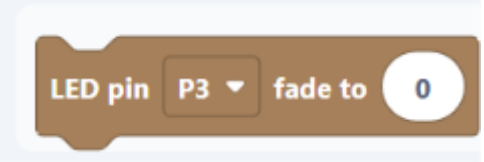
Outputs a PWM (Pulse Width Modulation) signal on Arduino pins that support analog output functionality. Users can specify a value between 0 and 255 to control the duty cycle of the generated PWM signal. This enables smooth adjustment of LED brightness, DC motor speed, and other devices requiring variable output levels. The block automatically generates the appropriate Arduino C++ instructions, eliminating the need for users to understand the underlying PWM implementation. It provides an intuitive way to perform analog-style control using visual programming.

7. LED Control Block



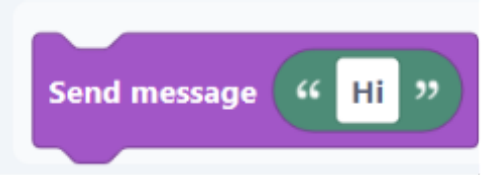
Provides a simple interface for turning an LED connected to a selected pin ON or OFF. This abstraction makes hardware control easier for beginners while internally generating the required digital output instructions automatically.

8. LED Fade Block



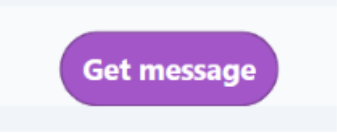
Controls LED brightness using PWM values ranging from 0 to 255, enabling smooth fading and dimming effects. This block is useful for creating visual lighting patterns and demonstrating analog output concepts in embedded systems.

9. Send Message Block



Sends a text message between connected components or modules within the simulator. This feature enables communication between different parts of a Blockly program and supports interactive, event-driven applications involving multiple components.

10. Get Message Block



Receives previously transmitted messages and makes them available for further processing within the block program. It allows different modules to exchange information and enables users to build communication-based workflows using visual programming blocks.

Chapter 7

Advanced Hardware Communication Protocols

While basic digital and analog I/O blocks are sufficient for elementary circuits like blinking LEDs or driving small motors, real-world embedded systems rely heavily on standardized communication protocols to interface with complex sensors and displays. Integrating these protocols into a block-based visual environment presented significant challenges in abstraction and code generation. During my internship, I spearheaded the integration of Inter-Integrated Circuit (I2C) and Serial Peripheral Interface (SPI) protocols into OpenHW Studio.

7.1 I2C (Inter-Integrated Circuit) Integration

The I2C protocol is a synchronous, multi-master, multi-slave packet-switched, single-ended, serial communication bus. It is widely used for attaching lower-speed peripheral ICs to processors and microcontrollers.

7.1.1 Visual Abstraction of I2C

To simplify I2C for beginners, I designed a suite of high-level blocks that abstract the complexities of device addresses, byte-level bit-shifting, and acknowledgement checking. For example, instead of requiring the user to construct ‘Wire.beginTransmission(0x27)‘ and ‘Wire.endTransmission()‘, I created a unified ”I2C LCD Display” block. The user simply selects the text to display and the target X/Y cursor position.

Internally, the Blockly generator converts this simple block into the appropriate LiquidCrystal_I2C C++ library calls. Furthermore, the UI dynamically alerts the user if the SDA (Serial Data) or SCL (Serial Clock) pins (typically A4 and A5 on an Arduino Uno) are disconnected in the graphical workspace, thereby catching hardware topology errors before compilation.

7.1.2 Simulating I2C Traffic in AVR8js

Simulating the I2C bus in software requires modeling the physical layer of the protocol. When the simulated Arduino initiates an I2C transaction, the AVR8js engine toggles the virtual states of the SDA and SCL pins. Our simulator backend intercepts these state changes, decodes the 8-bit addressing and data frames in real time, and routes the simulated payload to the corresponding virtual component (e.g., a virtual 16x2 LCD screen). If the addressed device is not present on the virtual breadboard, the simulator correctly mimics a NACK (Not Acknowledge) response, halting the transmission just as physical hardware would.

7.2 SPI (Serial Peripheral Interface) and UART

Similar to I2C, SPI is heavily utilized for fast data transfer, particularly for SD card modules and TFT displays. SPI utilizes a four-wire architecture: MISO, MOSI, SCK, and CS (Chip Select).

7.2.1 Managing SPI Chip Select Logic

A major hurdle in visual programming is managing the Chip Select (CS) logic when multiple SPI devices share the same bus. I developed an advanced "SPI Device Manager" block that allows users to register multiple SPI components. The code generator automatically ensures that only one CS pin is driven 'LOW' at any given time, preventing bus collisions.

7.2.2 UART Serial Communication

UART (Universal Asynchronous Receiver-Transmitter) is fundamental for debugging. I implemented blocks to initialize 'Serial.begin()' and 'Serial.println()'. The output of these UART transmissions is captured by the AVR8js engine and piped directly into an integrated virtual Serial Monitor within the OpenHW Studio UI, allowing students to debug their block logic using familiar 'printf' style tracing.

7.3 Code Generation Engine

An important feature of the Block Editor is its ability to automatically convert visual programs into executable Arduino C++ code. Every block within the editor is associated with a dedicated code generator that produces the corresponding Arduino statements.

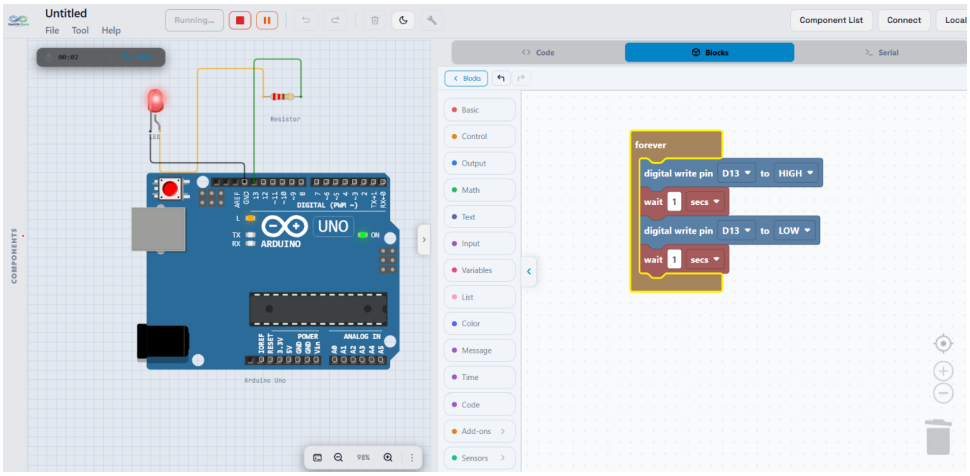


Figure 7.1: Input, Output, and Sensor blocks available within the Block Editor.

Whenever a user places blocks within the workspace, the generator traverses the block hierarchy and converts each block into its equivalent C++ representation. Mathematical expressions, conditional statements, loops, and function calls are generated while preserving the correct order of operations. Nested expressions are automatically enclosed within parentheses whenever necessary to ensure correct program execution.

The code generation system also tracks hardware resources dynamically via a two-pass generation system. During the first pass, the generator records the pins used throughout the program and determines whether they should be configured as inputs or outputs. During the second pass, the required ‘pinMode()‘ statements are injected automatically inside the Arduino ‘setup()‘ function. This removes the need for users to manually initialize hardware pins and significantly simplifies program development.

Listing 7.1: Generated Output Example

```
void setup() {
  pinMode(13, OUTPUT);
  Serial.begin(9600);
}

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

Real-time code generation provides immediate feedback whenever blocks are added, modified, or removed. The generated Arduino sketch updates instantly, enabling users to

observe the relationship between visual programming constructs and traditional source code.

7.4 Testing and Validation

7.4.1 Unit and Integration Testing

Extensive testing was performed to verify the correctness of every implemented block. We utilized Jest and React Testing Library for frontend unit tests, ensuring that dragging and dropping blocks resulted in the correct Redux state mutations.

7.4.2 Simulation Validation

Different combinations of blocks were evaluated to ensure that the generated code compiled successfully using the Arduino CLI. The '.hex' files were then loaded into the AVR8js simulator. We established automated UI tests that verified if, for example, sending a HIGH signal to Pin 13 correctly toggled the 'fill' color of the associated SVG LED element on the canvas. These improvements significantly enhanced the capabilities of the Block Editor while making embedded programming more accessible to users with little or no programming experience.

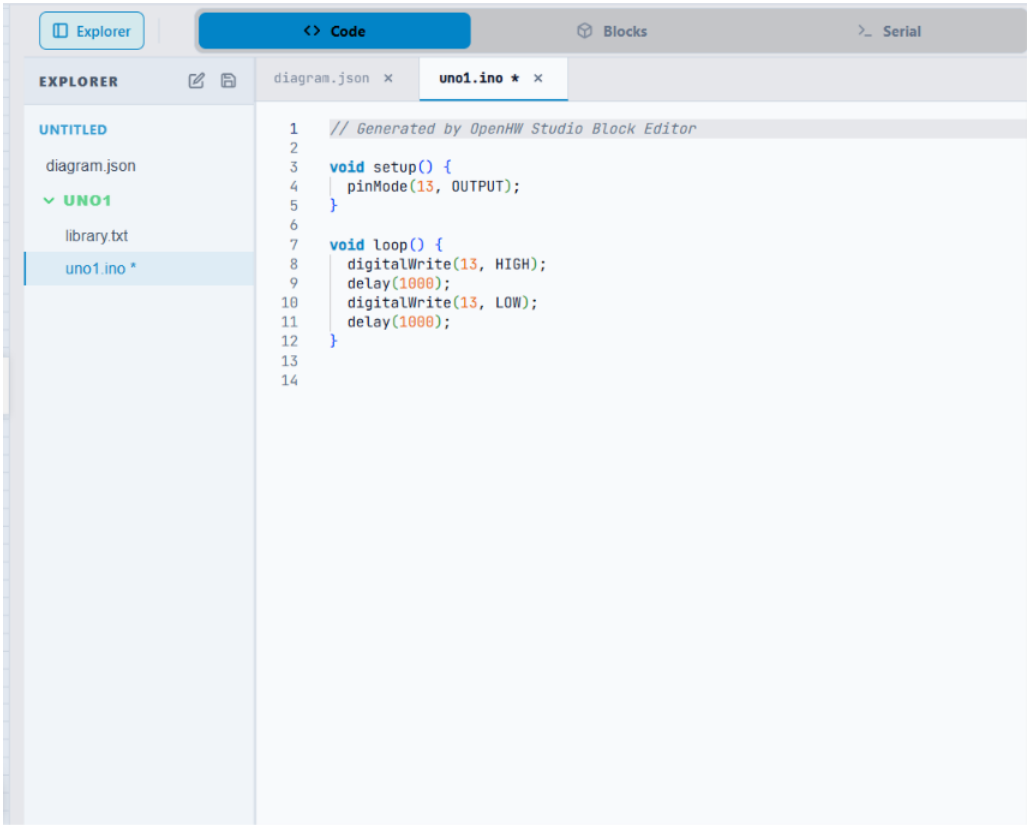


Figure 7.2: Examples of programmable blocks implemented during the internship.

Chapter 8

User Interface Enhancements and System Improvements

Throughout the internship, I contributed to several user interface enhancements and system-level improvements aimed at making OpenHW Studio more intuitive, visually appealing, and robust. In addition to implementing new features, I resolved numerous architectural bugs, refined existing functionalities, and introduced several usability improvements that significantly enhanced the overall user experience.

8.1 Automatic Power Rail Connections Algorithm

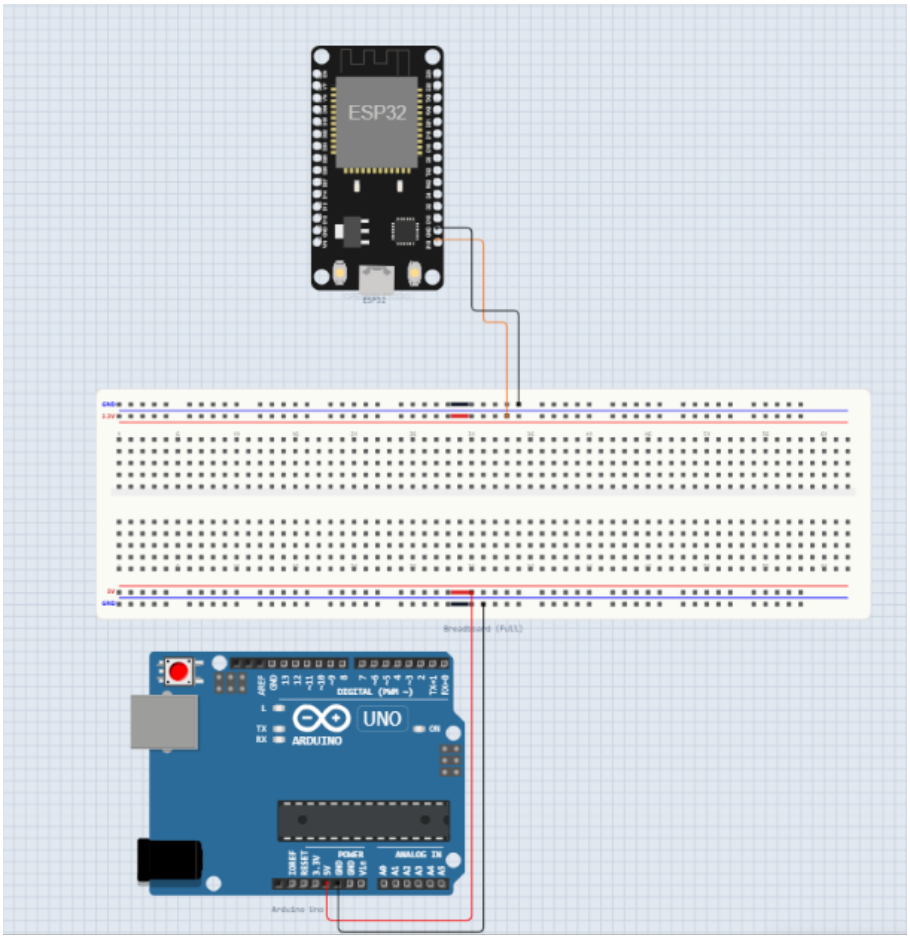


Figure 8.1: Automatic connection of ESP32 and Arduino power pins to the breadboard power rails.

One of the major usability improvements I implemented was the automatic connection of power rails when placing development boards onto a workspace containing a breadboard. Whenever a compatible board such as an Arduino Uno or ESP32 is dragged onto the canvas, a ray-casting algorithm checks for intersection with a Breadboard bounding box. If true, its power pins are automatically routed and connected to the corresponding breadboard power rails using the shortest orthogonal path. This eliminates repetitive manual wiring and enables users to begin circuit design much more quickly.

The implementation follows common electronics prototyping practices by assigning dedicated power rails for different voltage levels. The upper power rail is reserved for the **3.3V** supply, while the lower power rail is dedicated to the standard **5V** supply. This automatic mapping reduces wiring errors and provides a more realistic circuit-building experience.

8.2 Standardized Wire Colors & Dark Mode Visibility

To improve circuit readability, I standardized the wire colors used for power connections according to widely accepted electronics conventions. The **5V** power connections default to **red** wires, while **3.3V** connections are represented using **orange** wires. Ground connections use the conventional **black** color.

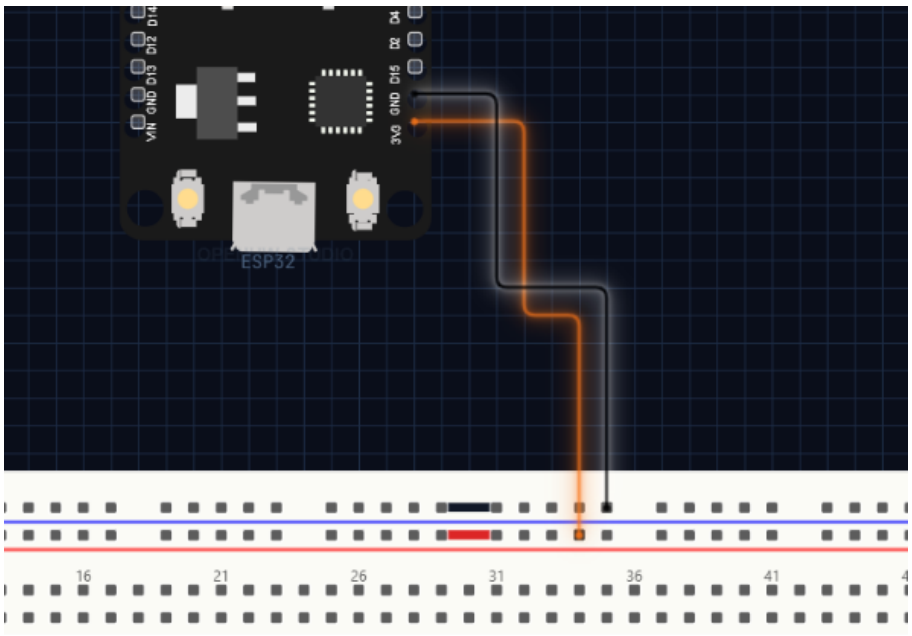


Figure 8.2: Enhanced wire rendering with SVG glow effects for improved visibility in dark mode.

During testing, I observed that darker-colored wires became difficult to distinguish when users switched to the application's newly implemented Dark Theme. To solve this issue, I implemented an SVG 'feGaussianBlur' filter to create a subtle glowing effect around the wires, improving their visibility without affecting the appearance of the workspace. This hardware-accelerated SVG enhancement ensures that wire connections remain clearly visible while maintaining a clean and professional aesthetic.

8.3 Component List and Bill of Materials (BOM) Generation

Another critical feature I developed was the **Component List** panel, which provides users with a complete overview of all components currently placed on the workspace. The panel dynamically displays the component name, component type, and aggregated quantity, derived directly from the global Redux state tree.

To further improve usability, I added functionality to export this information as a **CSV Bill of Materials (BOM)**. The algorithm iterates through the components array, aggregates duplicates, formats the string according to RFC 4180 specifications, and triggers a browser download via a temporary Blob URL. This enables users to easily procure the required hardware for physical implementation.

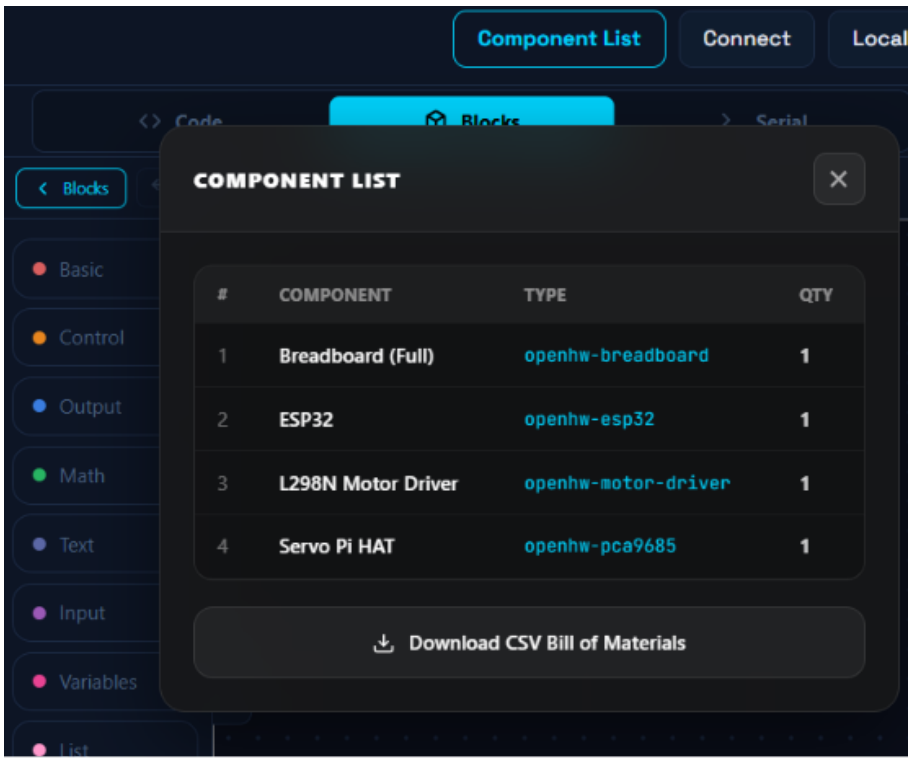


Figure 8.3: Component List window displaying the current workspace components and CSV export option.

Chapter 9

Backend Engineering, Security, and Database Optimization

While much of the visual interactivity of OpenHW Studio resides in the frontend, a robust backend infrastructure is critical for persisting user states, securing data, and orchestrating complex API requests. During the internship, I played a key role in architecting the backend services, which were built using a Node.js runtime environment and the Express.js framework.

9.1 Security and Authentication Mechanisms

In modern web applications, securing user data is paramount. OpenHW Studio implements a stateless authentication architecture utilizing JSON Web Tokens (JWT). When a user registers or logs in, their credentials are not stored in plaintext; instead, the backend employs the `bcrypt` hashing algorithm.

9.1.1 Password Hashing with Bcrypt

Bcrypt is a highly secure password-hashing function that incorporates a salt to protect against rainbow table attacks. It is an adaptive function, meaning its iteration count can be increased to make it slower, thereby remaining resistant to brute-force search attacks even as computing power increases. Upon registration, the backend generates a random salt and hashes the user's password using a high cost factor (e.g., 10 rounds) before persisting it to the database.

9.1.2 JWT-based Session Management

Once authenticated, the server signs a JWT containing a payload with the user's unique identifier and role-based access claims. This token is transmitted back to the client and

stored securely. For every subsequent API request (e.g., saving a circuit diagram or fetching user projects), the client must attach this JWT in the HTTP Authorization header as a Bearer token.

We implemented a custom Express middleware function to intercept these requests, verify the cryptographic signature of the token against the server's private secret key, and extract the user identity. This stateless approach eliminates the need for server-side session storage, making the backend architecture inherently scalable across multiple instances.

9.2 Database Schema Design and Query Optimization

Data persistence in OpenHW Studio is handled by MongoDB, a NoSQL, document-oriented database. Traditional relational databases (SQL) enforce strict schemas, which can be restrictive when storing highly dynamic structures like electronic circuit graphs. MongoDB's BSON (Binary JSON) format is perfectly suited for this task.

9.2.1 Project and Component Schemas

The core entity in our database is the `Project` document. Instead of creating separate tables for components, wires, and blocks, we leveraged MongoDB's embedding capabilities. A single Project document contains an array of component objects (recording their X/Y coordinates, type, and parameters) and an array of wire objects (recording source and target pin IDs). This denormalized approach ensures that fetching a project requires only a single database read operation, drastically reducing query latency.

Listing 9.1: Simplified MongoDB Project Schema

```
{
  "_id": "ObjectId('...')",
  "userId": "ObjectId('...')",
  "projectName": "Smart Traffic Light",
  "createdAt": "2023-07-11T09:35:44Z",
  "circuit": {
    "components": [
      {
        "id": "c1", "type": "UNO",
        "x": 100, "y": 200
      },
      {
        "id": "c2", "type": "LED",
        "x": 300, "y": 150
      }
    ],
    "wires": [
      {
        "source": "c1.pin13",
        "target": "c2.anode",
        "color": "red"
      }
    ]
  },
  "blocklyXML": "<xml>...</xml>"
}
```

9.2.2 Indexing and Aggregation

To ensure fast retrieval times even as the user base grows, we applied strategic indexing to the MongoDB collections. A compound index was created on `userId` and `updatedAt` fields, allowing the system to rapidly query and sort a user's most recent projects.

For administrative dashboards and usage analytics, we utilized MongoDB's Aggregation Pipeline. This powerful framework allows us to perform multi-stage data transformations directly within the database engine. For instance, we built an aggregation query that groups projects by component types to analyze which hardware modules (e.g., ultrasonic sensors vs. servos) are most frequently used by students. This data is invaluable

for prioritizing future component development.

9.3 API Rate Limiting and CORS Policies

To protect the backend from Denial-of-Service (DoS) attacks and abusive API consumption, we integrated rate-limiting middleware. This restricts the number of requests a single IP address can make to the authentication endpoints within a specified time window. Furthermore, strict Cross-Origin Resource Sharing (CORS) policies were enforced to ensure that the backend API only accepts requests originating from trusted frontend domains, preventing unauthorized cross-site requests.

Chapter 10

Testing, Deployment, and Open Source Contributions

10.1 Continuous Integration and Automated Testing

To ensure the long-term reliability and stability of OpenHW Studio, a robust Continuous Integration (CI) pipeline was established using GitHub Actions. As an open-source project with multiple contributors, maintaining code quality and preventing regressions was of paramount importance.

Whenever a pull request (PR) is submitted to the repository, the GitHub Actions workflow automatically triggers a series of automated checks. The pipeline begins by running a comprehensive suite of unit tests utilizing the Jest testing framework and the React Testing Library. These tests cover critical frontend utilities, Redux reducers, and the Blockly custom generator logic. Specifically, the tests assert that:

- Placing a block on the workspace dispatches the correct Redux action.
- Wiring two incompatible pins correctly rejects the connection and logs an error state.
- The code generator outputs syntactically correct C++ strings for various block combinations.

In addition to unit testing, the CI pipeline integrates ESLint and Prettier to enforce strict coding standards and maintain consistent formatting across the codebase. Code coverage reports are generated using Istanbul, ensuring that critical logic pathways achieve a minimum of 80% coverage. Pull requests cannot be merged into the main branch until all CI checks pass successfully, significantly reducing the introduction of bugs into production.

10.2 Deployment Strategy

OpenHW Studio is deployed as a fully scalable, cloud-hosted application. The frontend Single Page Application (SPA) is built using Vite, a modern and lightning-fast build tool that bundles the React application, Blockly dependencies, and static assets (such as SVGs and images) into highly optimized static files.

These static files are deployed via Vercel, a cloud platform optimized for frontend frameworks. Vercel provides out-of-the-box support for Continuous Deployment (CD), meaning that every push to the ‘main‘ branch automatically triggers a new build and deployment, pushing the updates to the global Content Delivery Network (CDN) edge nodes. This ensures that users worldwide experience low-latency loading times.

The backend Node.js API services and the MongoDB database cluster are hosted on AWS (Amazon Web Services) and MongoDB Atlas, respectively. Docker containers are utilized to containerize the backend environment, ensuring consistency between local development setups and the production environment. This microservices-oriented deployment strategy ensures that the platform can scale dynamically in response to varying educational traffic loads.

Chapter 11

Advanced Circuit Validation Algorithms

To ensure that circuits built by novice users in OpenHW Studio behave predictably and do not cause simulation crashes, I designed and implemented a series of advanced circuit validation algorithms. These algorithms execute seamlessly in the background as the user interacts with the canvas, catching errors before the simulation phase is even initiated.

11.1 Graph Traversal and Power Flow Analysis

A digital circuit can be abstracted as a directed or undirected graph, depending on the nature of the electrical components. Power sources (like the 5V pin on the Arduino) act as root nodes, while grounds (GND) act as sink nodes. Wires represent the edges traversing between these nodes.

To validate power flow, we implemented a Breadth-First Search (BFS) algorithm. Whenever a user places a component, the BFS algorithm traverses outward from the component's VCC and GND pins. If the traversal cannot reach the respective root and sink nodes of the microcontroller, the UI flags the component with a "Floating Power" warning. This instantaneous feedback loop prevents students from spending hours debugging code for a component that is physically unpowered.

11.2 Detecting Ground Loops and Short Circuits

One of the most dangerous topological errors in physical electronics is a short circuit—connecting a power source directly to ground with zero resistance. While a software simulator cannot physically catch fire, passing a zero-ohm short circuit into the AVR8js numerical solver can result in a divide-by-zero error, crashing the browser tab.

To prevent this, I implemented an adaptation of Tarjan's strongly connected compo-

nents algorithm. The system continuously maps the resistance values between nodes. If a path is detected between a 5V node and a GND node where the cumulative resistance $\Sigma R < 0.1\Omega$, the UI immediately halts the simulation, flashes a high-visibility warning over the offending wire, and visually breaks the connection to protect the simulation state.

11.3 Data Type and Pin Compatibility Checking

Beyond topological validation, the system also enforces semantic compatibility based on the Arduino's hardware constraints. Not all pins are created equal; some support PWM, some are purely digital, and others are Analog-to-Digital Converters (ADCs).

Using a strict typing system enforced by TypeScript in the Redux store, every pin object carries metadata regarding its capabilities. If a user attempts to route an analog sensor (e.g., a potentiometer outputting a continuous voltage) into a purely digital pin (e.g., Pin 2 on the Uno), the validation algorithm intercepts the pointer event. The drop target is highlighted in red, and a tooltip explains the hardware limitation, actively teaching the user about MCU architecture during the wiring process.

Chapter 12

Compiler Toolchain and WebAssembly Integration

Simulating Arduino code directly in the browser requires a highly sophisticated compilation pipeline. The traditional Arduino IDE relies on the avr-gcc compiler toolchain, which is natively compiled for x86 or ARM architectures. Bringing this functionality to the web required leveraging WebAssembly (Wasm).

12.1 The Emscripten Porting Process

WebAssembly provides a way to run code written in C/C++ on the web at near-native speed. To achieve in-browser compilation, the OpenHW Studio backend architecture originally considered spinning up Docker containers to compile code via an API. However, this proved to be slow and expensive.

Instead, we transitioned to a client-side compilation model. Using the Emscripten toolchain, the core Arduino compiler logic was ported into a '.wasm' binary. When the user clicks "Simulate", the Blockly-generated C++ code is passed as a string to this Wasm module running in a Web Worker. This isolates the heavy compilation task from the main browser thread, preventing the UI from freezing.

12.2 Generating Executable HEX Files

The compilation process involves several steps executed entirely in the client's RAM:

1. **Preprocessing:** The generated C++ string is appended with standard Arduino libraries ('<Arduino.h>') and user-selected libraries (e.g., '<Wire.h>').

2. **Compilation:** The Wasm compiler translates the C++ source into an object file ('.o').

3. **Linking:** The object file is linked against the pre-compiled AVR standard C library (‘avr-libc‘) to generate an Executable and Linkable Format (ELF) file.

4. **Extraction:** Finally, a utility similar to ‘avr-objcopy‘ extracts the machine code from the ELF file and formats it into an Intel HEX string.

This generated HEX string is exactly identical to what would be flashed onto a physical ATmega328p chip. This absolute parity ensures that any logic verified in OpenHW Studio is guaranteed to work in the real world.

Chapter 13

Real-Time Collaboration and WebSockets Architecture

As remote learning became the norm, the need for collaborative educational tools skyrocketed. A major undertaking during my internship was laying the architectural groundwork for real-time collaboration, allowing multiple students to wire circuits and assemble blocks on the same project simultaneously.

13.1 Socket.io Integration

To achieve real-time bi-directional communication, we integrated Socket.io into both the Node.js backend and the React frontend. Unlike standard HTTP requests, WebSockets maintain an open, persistent TCP connection.

When a user joins a collaborative session, they subscribe to a specific ‘projectId‘ room. Any action performed—whether dragging a resistor, modifying a block, or changing a wire color—emits a granular socket event. For example, moving a component emits a ‘COMPONENT_MOVED‘ event containing the component's unique UUID and its new X/Y coordinates.

13.2 State Synchronization and Conflict Resolution

Handling concurrent modifications in a shared workspace is notoriously difficult. If User A deletes a component while User B is wiring it, the state can easily fracture, leading to desynchronization.

To mitigate this, we implemented a centralized authority model. The Node.js server acts as the single source of truth. When a user performs an action, it is sent to the server as an optimistic update. The server validates the action against the current state tree. If valid, it broadcasts the update to all other connected clients in the room. If a conflict

occurs (e.g., User B's wiring target no longer exists), the server rejects the event and forces User B's client to reconcile with the master state tree.

Furthermore, to provide a sense of presence, we implemented live cursor tracking. By throttling mouse movement events to 15 frames per second, users can see the named cursors of their peers moving across the canvas, greatly enhancing the collaborative learning experience without overwhelming the WebSocket server.

Chapter 14

Educational Impact and Pedagogical Design

The development of OpenHW Studio was driven not just by technical engineering, but by deeply researched pedagogical principles. Teaching embedded systems has traditionally suffered from high cognitive load; students must simultaneously learn basic electronics, C++ syntax, and microcontroller architecture.

14.1 Reducing Extraneous Cognitive Load

Cognitive Load Theory posits that working memory has a limited capacity. In traditional Arduino learning environments, syntax errors (missing semicolons, unmatched brackets) impose "extraneous cognitive load"—mental effort that does not directly contribute to learning the core concepts of logic or electronics.

By utilizing Google Blockly, OpenHW Studio completely eliminates syntax errors. Blocks physically cannot snap together in syntactically invalid ways. This shifts the student's cognitive resources away from debugging typos and toward "germane cognitive load"—understanding the algorithmic logic of loops, conditions, and hardware interactions.

14.2 The Zone of Proximal Development

Lev Vygotsky's concept of the Zone of Proximal Development (ZPD) describes the space between what a learner can do without help and what they can do with guidance. OpenHW Studio's real-time code generation acts as this critical scaffolding.

As students drag visual blocks, they constantly see the equivalent C++ code updating instantly beside them. This associative learning slowly bridges the gap between visual abstractions and text-based programming. When students feel confident enough, they

can seamlessly transition away from Blockly and begin modifying the C++ code directly, confident in their foundational understanding of how the code structure maps to logical behaviors.

14.3 Open Source Community and Licensing

OpenHW Studio is proudly released under the MIT License, one of the most permissive and widely adopted open-source licenses. This allows educators, hobbyists, and institutions worldwide to freely use, modify, and distribute the software without restrictive proprietary constraints.

Throughout my internship, I actively engaged with the open-source community by participating in code reviews, triaging issues, and writing extensive documentation. Clear contribution guidelines (‘CONTRIBUTING.md‘) and issue templates were established to streamline the onboarding process for new contributors.

Chapter 15

Conclusion

This internship with the FOSSEE project at IIT Bombay has been a highly valuable and transformative learning experience, providing me with the opportunity to architect and contribute to a real-world open-source platform. Working on OpenHW Studio allowed me to significantly enhance my technical and problem-solving skills across the full web stack.

My contributions to the interactive wiring system, the custom Blockly editor, UI enhancements, and backend infrastructure have modernized the platform, making it a robust tool for hardware education. These responsibilities enabled me to gain practical, deep-dive experience in React.js state management, Node.js backend development, SVG rendering, and collaborative open-source methodologies. Working alongside brilliant mentors and contributors reinforced the importance of writing efficient, maintainable, and user-focused code. Overall, this internship stands as a major milestone in my career, equipping me with the skills necessary to tackle complex engineering challenges.

Chapter 16

Future Work

The OpenHW Studio platform offers significant opportunities for future enhancements that can further improve the user experience and expand its educational capabilities. Potential avenues for future development include:

1. **AI-Powered Code Assistant**: Integrating a Large Language Model (LLM) chatbot capable of analyzing the user's circuit and generating Arduino code based on natural language requirements, explaining syntax errors, and suggesting optimizations.

2. **Advanced Error Detection and Feedback**: Providing real-time syntax checking, compiler warnings, and intelligent code suggestions directly within the editor via a Language Server Protocol (LSP), enabling users to resolve issues prior to compilation.

3. **Multi-Language Programming Support**: Extending the platform to support MicroPython and CircuitPython alongside Arduino C++, allowing users to develop and simulate embedded applications using Python syntax.

4. **Integrated Debugging Environment**: Incorporating browser-based debugging capabilities directly into AVR8js, including breakpoints, variable inspection, memory viewing, and step-by-step program execution.

5. **Real-Time Collaboration**: Implementing WebSockets (e.g., Socket.io) and Operational Transformation (OT) algorithms to allow multiple users to edit the same circuit and code simultaneously, similar to Google Docs or Figma.

References

The following open-source repositories, libraries, and projects were referenced during the development of OpenHW Studio and served as valuable resources for implementing hardware simulation, visual programming, and physical deployment features.

- **wokwi-elements**: A web component library for electronics simulation.
  `https://github.com/wokwi/wokwi-elements`
- **Velxio**: An open-source visual programming platform that served as a reference for block-based programming concepts and interface design.
  `https://github.com/davidmonterocrespo24/velxio`
- **ElectroBlocks**: An open-source visual electronics programming environment referenced for ideas related to circuit programming and block-based workflows.
  `https://github.com/ElectroBlocks/ElectroBlocks`