Summer Internship Report
on
ElectroBlocks Meets Python: A Powerful Upgrade

Submitted by:

Utkarsh
Amity University, Lucknow

Zoha Faiyaz
Jamia Millia Islamia

M. Boobalan
Kongu Engineering College

V. Abinaya
Anna University, MIT Campus

V. Anushiya
Anna University, MIT Campus

Under the guidance of:

Prof. Kannan M. Moudgalya
Department of Chemical Engineering
IIT Bombay

Noah Glaser
Developer, ElectroBlocks

Mr. Rajesh Kushalkar
Senior Project Manager, FOSSEE Project
IIT Bombay

Mr. Pratik Bhosale
Project Research Associate
FOSSEE Project, IIT Bombay

June 2025

Acknowledgement

We would like to take this opportunity to express our heartfelt gratitude to everyone who played a pivotal role in making our summer internship with FOSSEE – ElectroBlocks (Arduino) an enriching and memorable experience.

First and foremost, we extend our deepest thanks to Noah Glaser, developer of the Electroblocks simulation platform, Mr. Rajesh Kushalkar, Senior Project Manager at FOSSEE, IIT Bombay, and Mr. Pratik Bhosale, Project Research Associate at FOSSEE, IIT Bombay, for believing in our abilities and giving us the opportunity to work on this invaluable project. Their unwavering support and constant encouragement were the driving force behind our progress.

We are also deeply grateful to our mentors, whose guidance, patience, and expertise were instrumental throughout the internship. Their timely advice not only helped us overcome challenges and complete our tasks successfully, but also strengthened both our technical and interpersonal skills.

Lastly, we would like to thank our fellow interns and colleagues working on various projects. Their collaboration, constructive feedback, and thoughtful insights significantly contributed to our growth and made the entire experience all the more meaningful.

Utkarsh, Amity University, Lucknow

Zoha Faiyaz, Jamia Millia Islamia

Abinaya V, Anna University, MIT Campus

Anushiya V, Anna University, MIT Campus

Boobalan M., Kongu Engineering College

Declaration

We declare that this written submission represents our ideas in our own words. Whenever the ideas or words of others have been included, we have properly cited and referenced the original sources. We affirm that all sources used in the preparation of this thesis have been accurately acknowledged.

We also declare that we have strictly followed the principles of academic honesty and integrity, and have not misrepresented, fabricated, or falsified any idea, data, fact, or source in this submission. We understand that any violation of these principles may result in disciplinary action by the Institute and could also lead to legal consequences from the original sources if proper citations or permissions have not been provided.

Utkarsh, Amity University, Lucknow

Zoha Faiyaz, Jamia Millia Islamia

Abinaya V, Anna University,MIT Campus

Anushiya V, Anna University,MIT Campus

Boobalan M, Kongu Engineering College

Index

List Of Figures

Chapter 1: Introduction

Electroblocks is a practical and innovative platform aimed at making electronics and programming with Arduino more accessible and engaging. It features modular, plug-and-play components that connect easily, removing the need for complex wiring setups. Fully compatible with the Arduino ecosystem, users can program their projects using the Arduino IDE. With a clean, user-friendly interface and clearly labeled modules, Electroblocks is suitable for learners of all experience levels. It's particularly well-suited for STEM education, supporting a variety of projects—from basic circuits to more advanced systems. By blending the flexibility of Arduino with a modular, simplified hardware design, Electroblocks encourages exploration, creativity, and easy prototyping for both beginners and seasoned makers.

1.1 Project Overview

During the internship, we collaborated on an existing Electroblocks project that was already in development but required new features and improvements. Our main focus was on extending existing functionalities and translating them into Python-based implementations. Alongside this, we also brainstormed and introduced additional features to enhance the overall usability of the platform. Debugging and testing were a continuous part of the process, helping us refine the code and improve stability.

Chapter 2: Feature Additions

During the Internship, we added a lot of features to the project, most of these features aimed at extending the already existing blocks to generate code in Python language on the ElectroBlocks platform.

2.1 DROPDOWN MENU FOR LANGUAGE SELECTION

When we access the platform, we require a means/option to switch between different languages offered by the platform. This dropdown also dynamically enables and disables specific blocks that don't work with the pyFirmata protocol.

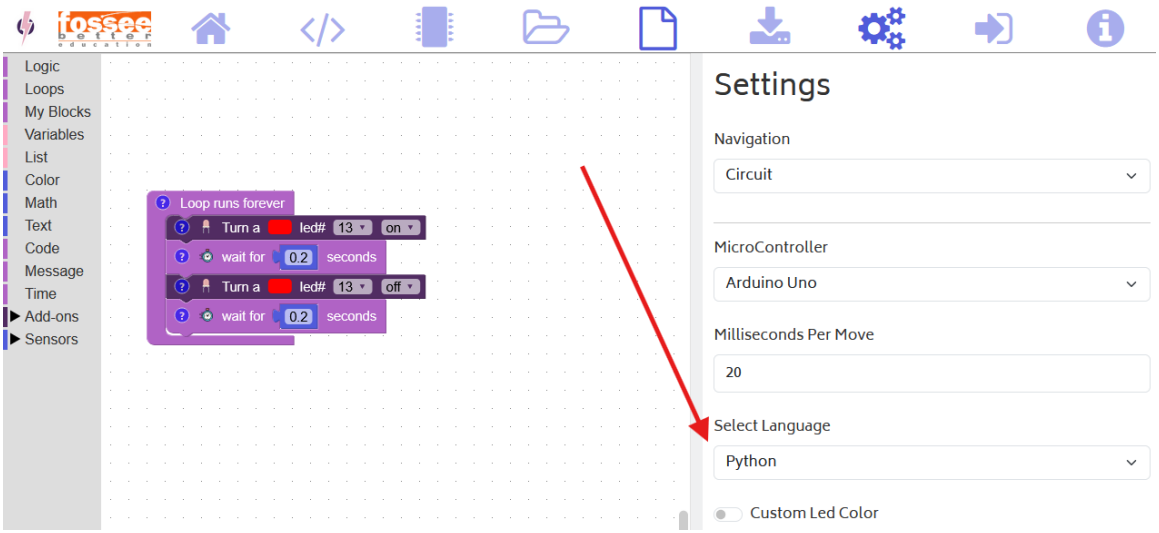


Fig 2. 1

Here's how the code handles the language change:

```
let settings = get(settingsStore);
  settingsStore.subscribe((newSettings) => {
    settings = newSettings;
  });
  if(settings.language === "C"){
    return Blockly["Arduino"].workspaceToCode(getWorkspace()) as string;
  }else{
  const resetPythonCode = `# Python Code Snippet
   print("Hello, World!")`;
   return resetPythonCode;
  }
```

Fig 2. 2 Changes to the `workspace.helper.ts`

Language Switching Process:

1. User Settings

- The language preference is stored in `settingsStore` and persists in the database.
- A dropdown in the settings UI allows the user to choose between C and Python.

2. Code Generation

```
if (getWorkspace()) {
      codeStore.set({
        cLang: getArduinoCode(),
        pythonLang: `# Python Code Snippet
    print("Hello, World!")`,
        boardType: settings.boardType,
      });
```

Fig. 2.1 1 Changes to the `register.ts`

- In workspace.helper.ts and registerEvents.ts, the settings.language value determines which language's code is generated:
    - C code is generated using Blockly["Arduino"].workspaceToCode.
    - Python code is hardcoded for invalid blocks print("Hello, World!")

3. Displaying the Code
    - In `code/+page.svelte`, the `settings.language` value is used to:
        - Highlight the correct code syntax using `highlight.js`.
        - Dynamically update the ` <code> ` block to match the selected language.
        - Ensure the correct code is displayed to the user.

4. Reactive Updates
    - Subscriptions to `settingsStore` ensure that any changes to the selected language immediately reflect in:
        - The displayed code.
        - The generated code.
        - The clipboard functionality.

2.2. MAIN PYTHON CODE GENERATOR

For generating the Python Code, we required a main Python generator to handle the conversion of Blockly blocks. It is achieved by defining reserved keywords, operator precedence and various utility methods for generating syntactically correct and semantically meaningful Python code.

Key features and functionality:

1. Initialization of Python Generator:
    - The file initializes a new Python generator through `Blockly["Python"]`.
    - Reserved Python keywords and built-in names are added to avoid naming conflicts in the generated code.
2. Operator Precedence:
    - Operator precedence levels specific to Python are defined. These levels help ensure that the generated code respects Python's precedence rules when combining multiple operations.
    - References:

        - https://neil.fraser.name/blockly/custom-blocks/operator-precedence
        - https://docs.python.org/3/reference/expressions.html#operator-precedence
3. Initialization of Workspace:
    - The `init` method prepares the generator for use in a Blockly workspace.
    - It:
        - Sets up variable mapping and initializes them based on types such as numbers, strings, booleans, and colors.
        - Prepares dictionaries to store imports, setup code, and function names.
4. Code Finalization:
    - The `finish` method is responsible for compiling the final Python code.
    - It combines:
        - Libraries (imports)
        - Setup code
        - Variable initializations
        - User-defined code
        - Functions
        - Special development variables are added if certain features like Bluetooth or Serial are used.
5. Utility Methods:
    - quote_: Properly escapes strings to create valid Python string literals.
    - scrub_: Handles comments and processes connected blocks to generate a seamless Python code sequence.
    - scrubNakedValue: Adds a newline to values that aren't part of a larger expression.
6. Dynamic Typing and Variable Initialization:
    - Variables are initialized dynamically based on their types (e.g., numbers default to 0, strings to "", booleans to False, and colors to (0, 0, 0)).
7. Comment Management:
    - The scrub_ method ensures that comments on Blockly blocks are appropriately translated into Python comments.

2.3 IMPLEMENTING PYTHON GENERATORS FOR VARIOUS BLOCKS

Once the main generator (`python.ts`) was completed, individual generators were to be implemented for individual blocks and categories for blockly, to generate their equivalent Python code.

2.3.1 Logic Block

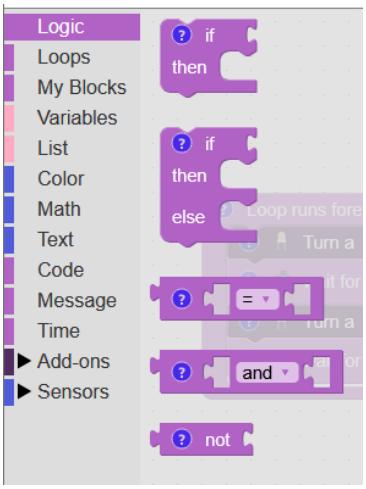


Fig. 2.3 1 Logic Block

1. logic_boolen
    - Before: Only Arduino code generation was defined for `logic_boolean` blocks.
    - Added: Python code generation for `logic_boolean`:

```
Blockly['Python']['logic_boolean'] = function(block: Block) {
  const code = block.getFieldValue('BOOL') == 'TRUE' ? 'True' : 'False';
  return [code, Blockly['Python'].ORDER_ATOMIC];
}
```

    - Python `True` and `False` values are returned based on the 'bool ' field value of the block.
2. logic_compare
    - Before: Only Arduino code generation was defined for `logic_compare` blocks.
    - Added: Python code generation for `logic_compare`:

```
Blockly['Python']['logic_compare'] = function(block: Block) {
  // Comparison operator.
  const OPERATORS = {
    EQ: '==',
    NEQ: '!=',
    LT: '<',
    LTE: '<=',
    GT: '>',
    GTE: '>='
  };
  const operator = OPERATORS[block.getFieldValue('OP')];
  const order =
    operator === '==' || operator === '!='
      ? Blockly['Python'].ORDER_COMPARISON
      : Blockly['Python'].ORDER_RELATIONAL;
  const argument0 = Blockly['Python'].valueToCode(block, 'A', order) || '0';
  const argument1 = Blockly['Python'].valueToCode(block, 'B', order) || '0';
  const code = "(" + argument0 + " " + operator + " " + argument1 + ")";
  return [code, order];
}
```

    - This generates Python comparison operators (==, !=, <, <=, etc.) for logical comparisons.

- It uses the `OP` field to determine the operator and constructs a Python-compliant comparison expression.

3. logic_operation
   - Before: Only Arduino code generation was defined for `logic_operation` blocks.
   - Added: Python code generation for `logic_operation`:

```
Blockly['Python']['logic_operation'] = function(block: Block) {
  // Operations 'and', 'or'.
  const operator = block.getFieldValue('OP') === 'AND' ? 'and' : 'or';
  const order =
    operator === 'and'
      ? Blockly['Python'].ORDER_LOGICAL_AND
      : Blockly['Python'].ORDER_LOGICAL_OR;
  let argument0 = Blockly['Python'].valueToCode(block, 'A', order);
  let argument1 = Blockly['Python'].valueToCode(block, 'B', order);
  if (!argument0 && !argument1) {
    // If there are no arguments, then the return value is False.
    argument0 = 'False';
    argument1 = 'False';
  } else {
    // Single missing arguments have no effect on the return value.
    const defaultArgument = operator === 'and' ? 'True' : 'False';
    if (!argument0) {
      argument0 = defaultArgument;
    }
    if (!argument1) {
      argument1 = defaultArgument;
    }
  }
  const code = argument0 + ' ' + operator + ' ' + argument1;
  return [code, order];
}
```

   - This generates Python comparison operators (==, !=, <, <=, etc.) for logical comparisons.

   - It uses the `OP` field to determine the operator and constructs a Python-compliant comparison expression

4. control_if
   - Before: Only Arduino code generation was defined for `control_if` blocks.
   - Added: Python code generation for `control_if`:

```
Blockly['Python']['control_if'] = function(block: Block) {
  let code = "",
  branchCode,
  conditionCode;
  // If/elseif/else condition.
  let n = 0;

  do{
    conditionCode =
      Blockly['Python'].valueToCode(block, 'IF' + n,
Blockly['Python'].ORDER_NONE) || 'False';
    branchCode = Blockly['Python'].statementToCode(block, 'DO' + n);
    code +=
      (n > 0 ? 'elif ' : 'if ') +
      conditionCode + ':\n' +
      branchCode;
    ++n;
  } while (block.getInput('IF' + n));

  if (block.getInput('ELSE')) {
    branchCode = Blockly['Python'].statementToCode(block, 'ELSE');
    code += 'else:\n' + branchCode;
  }
  return code + '\n';
};
```

- Handles Python `if`, `elif`, and `else` statements.
- The `IF` and `DO` fields are used to generate conditions and their corresponding code blocks.
- Supports multiple conditions (`if`, `elif`) and an optional `else`.

5. Similarly, we implement `control_ifelse`, `logic_negate`.

The changes ensure that Blockly can now generate Python code for these blocks in addition to Arduino code. Proper handling of operator precedence, default values, and conditional logic is included.

2.3.2 Variables Block

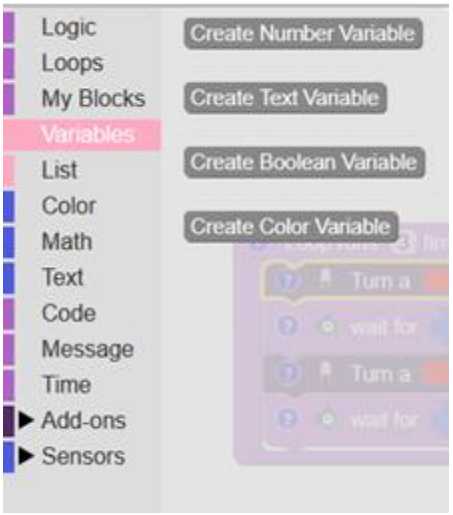


Fig. 2.3 2 Variables Block

This block defines how variable related blocks, such as those for setting and getting numbers, Booleans, strings and colours —are translated into executable code.

While creation of the blocks, the code ensures that when a user assigns a value to a variable in Blockly, it generates the corresponding code line (eg., `variable = value`; for Arduino or `variable = value` for Python), using a predefined value if no specific input is provided.

```
const setVariableFunctionPy = function (defaultValue) {
  return function (block, generator: CodeGenerator){
    const variableName =
generator.getVariableName(block.getFieldValue("VAR"));
    const variableValue = Blockly["Python"].valueToCode(
      block,
      "VALUE",
      Blockly["Python"].ORDER_ATOMIC
    );

    return variableName + " = " + (variableValue || defaultValue) + "\n";
  };
};
```

2.3.3 Text Block

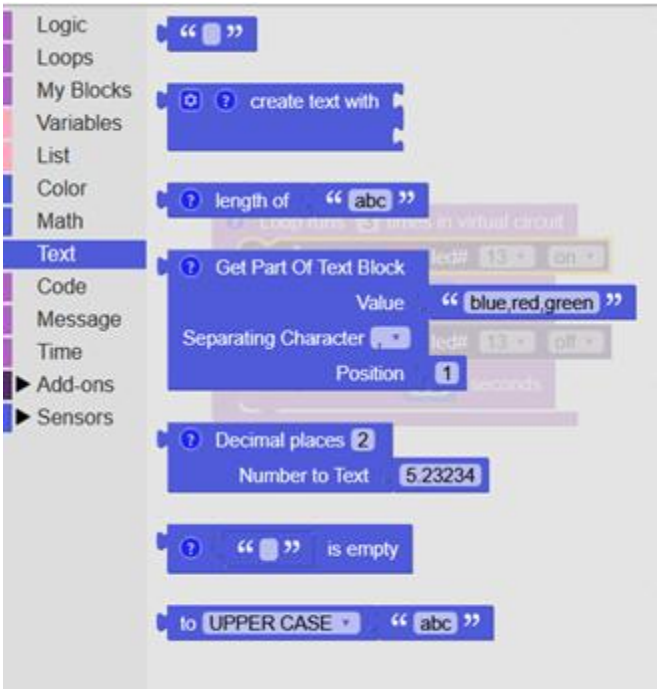


Fig. 2.3 3 Text Block

This block defines how text related functions are handled by Electroblocks, it handles various functions such as:

- Parsing input to text based on a regex expression, such as:
```
Blockly["Python"].text.forceString_.strRegExp = /^(['"])(?:\\.|[^\\])*?\1$/;
```
- Joining multiple text blocks or sentences,
- Retrieve text: based on multiple parameters such as the 'separating character' and position of a word.
- Conversion of text to numbers, like:

```
const code = `float(${string} if ${string}.replace('.','',1).isdigit() else
0)`;
```

- Finding out whether a string is empty or not.
- Converting a string to Uppercase or Lowercase.

It utilizes functions like `text_join`, 'text_length', 'text_isEmpty', 'number_to_string', 'text_changeCase', 'parse_string_block, which are defined at [1] along with all the code as well as several in-built functions from Python itself such as `.lower()`, `.upper()` etc.

In summary, these Python generators take the user's Blockly visual blocks and turn them into valid, readable Python code that mirrors the intended text operations. This allows non-programmers to build text-manipulating programs visually, while the system handles the translation into correct Python syntax behind the scenes.

2.3.4 Math Block

Fig. 2.3 4 Math Block

This block defines various mathematical operation functions that are supported by ElectroBlocks through Python's in-built `math` module, it handles functions such as:

- Setting a variable for an integer
- Finding out whether a number is even or odd.
- Handling basic arithmetic operations (add, subtract, multiply, divide power) between two inputs.
- Rounding a number to the nearest whole number using Python functions like: `round()`, `math.ceil()`, `math.floor()` from Python's `math` module.

2.3.5. Color Block

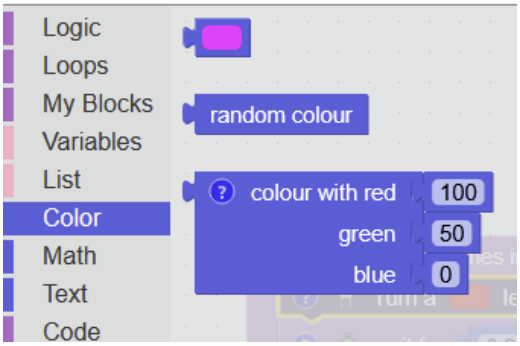


Fig. 2.3 5 Color Block

This block adds the ability for the system to work with colors in ElectroBlocks. It allows users to pick, create, and use random colors in their projects using Python code.

The Python Generators work by creating a special color object (called RGB) for you from the `RGB` dataclass.

```
function createColorStructPy() {
  Blockly["Python"].imports_["import_dataclass"] = `
from dataclasses import dataclass`
  Blockly["Python"].functionNames_["color_struct"] = `
@dataclass
class RGB:
  red: float
  green: float
  blue: float
  `;
}
```

- Picking Custom colors: When the user picks a color from the color palette it gets parsed by the `hexToRgb` function [2(a)], which then returns a valid numerical value and is then passed to the `RGB` dataclass, which returns the equivalent Python tuple for it.

```
Blockly["Python"]["color_picker_custom"] = function (block) {
  const rgb = hexToRgb(block.getFieldValue("COLOR"));
  createColorStructPy();
  return [
    `RGB(${rgb.r}, ${rgb.g}, ${rgb.b})`,
    Blockly["Python"].ORDER_ATOMIC,
  ];
};
```

- Random Colors: When the user wants a random color in ElectroBlocks. The Python generator used the `random` module to assign a random value from 0-255 for the R,G,B components of the color [2(b)].

2.3.6. Pins Block

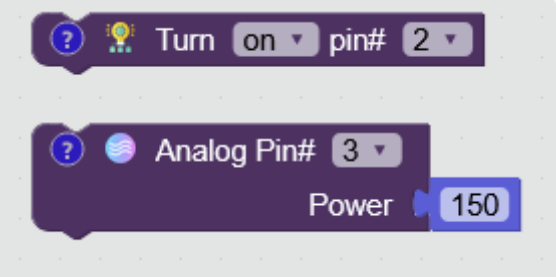


Fig. 2.3 6 Pins Block

This block controls an Arduino pin, letting you turn it on or off. It's like flipping a light switch, but for a piece of the Arduino.

The Python generators take this block and turn it into Python code that an Arduino understands. They look at what you set in the block (which pin, on or off) and write the correct Python commands so your Arduino does exactly what you want.

As we need pyFirmata, we first initialize pyFirmata for the board setup:

```
export function setup_code() {
  if(!Blockly["Python"].setupCode_["board_port"]) {
    Blockly["Python"].setupCode_["board_port"] =
      `PORT = "YOUR_PORT_HERE"\n`+
      `board = pyfirmata.Arduino(PORT)\n`+
      `\n`+
      `#Start the iterator thread to read inputs\n`+
      `it = pyfirmata.util.Iterator(board)\n`+
      `it.start()`+
      `\n`;
  }
}
```

- Digital Write: To write or give input to a specific pin we use pyFirmata's methods like `board.digital[pin].write(state)` or `board.get_pin('d:pin:p').write(value)`, so the Arduino can follow the commands.
- In Later parts, we utilize the ElectroBlocks library for digital and analog writes as we face some limitations with pyFirmata. [2]

2.3.7. Analog Sensor

Fig. 2.3 7 Analog Sensor Block

This code adds new features to a project that helps control electronics using blocks (like LEGO pieces, but for programming). Now, users can set up and read sensors not only for Arduino devices but also for Python-based boards. The update makes it easy to use both types of devices, making the system work for more people and different hardware.

- analog_read_setup: Generates Python setup code for analog input pins, creating a pin variable and enabling reporting (required for reading analog values). Uses the `board.get_pin(a:pin:mode)` from `pyFirmata`.

```
Blockly["Python"]["analog_read_setup"] = function (block: Block) {
  const pin = block.getFieldValue("PIN");
  firmata();
  const pinVar = `analog_pin_${pin}`;

  Blockly["Python"].setupCode_[pinVar] =
  `${pinVar} = board.get_pin('a:${pin}:i')
${pinVar}.enable_reporting()
`;
  return "";
};
```

- analog_read: Generates Python code to read the value from the specified analog pin variable.

```
Blockly["Python"]["analog_read"] = function (block: Block) {
  const pin = block.getFieldValue("PIN");
  const pinVar = `analog_pin_${pin}`;
  const code = `${pinVar}.read()`;
  return [code, Blockly["Python"].ORDER_ATOMIC];
};
```

2.3.8. Stepper Motor

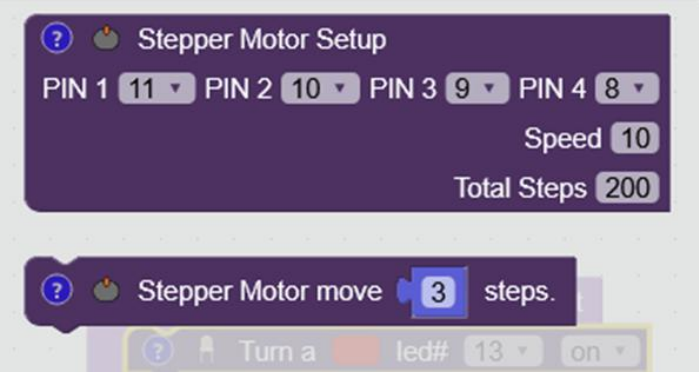


Fig. 2.3 8 Stepper Block

This code helps a visual programming tool (like Blockly) turn "blocks" that represent stepper motor actions into real code for Arduino and Python. It lets users set up and control a stepper motor by dragging blocks, and then automatically creates the code needed to run the motor in either Arduino or Python.

- stepper_motor_setup: it extracts user-specified settings (step count, speed, pin numbers) and generates borad initialization code:

```
Blockly["Python"]["stepper_motor_setup"] = function (block: Block) {
  const totalSteps = block.getFieldValue("TOTAL_STEPS");
  const speed = block.getFieldValue("SPEED");

  const pin1 = block.getFieldValue("PIN_1");
  const pin2 = block.getFieldValue("PIN_2");
  const pin3 = block.getFieldValue("PIN_3");
  const pin4 = block.getFieldValue("PIN_4");
  Blockly["Python"].imports_["pyfirmata"] = `
```

- stepper_motor_move: it generates code to move the stepper motor by a given number of steps that are specified by the user.

```
  const code = `
# Move the motor
_steps = ${steps}
if _steps >= 0:
  _direction = False
else:
  _direction = True
board.stepper_setup(abs(_steps), _direction)
`;
  return code;
};
```

2.3.9. Servo Motor

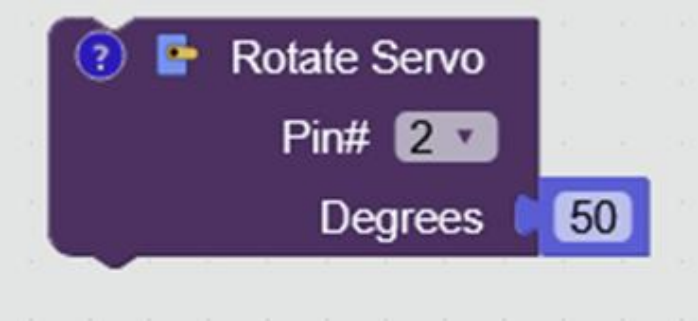


Fig. 2.3 9 Servo Block

This block defines how a servo motor connected to a specified pin is controlled by ElectroBlocks based on the degree of movement specified by the user.

- rotate_servo: It generates the code to move the motor itself based on the pin it's connected to and uses the degree block to rotate to the specified degree. It also records the most recent degree value so that other blocks can access the last known angle set by `rotate_servo`.

```
Blockly["Python"]["rotate_servo"] = function (block: Block) {
  const pin = block.getFieldValue("PIN");
  const pinVar = `servo_pin_${pin}`;
  const angleVar = `servo_angle_${pin}`;

  const degrees = Blockly["Python"].valueToCode(
    block,
    "DEGREE",
    Blockly["Python"].ORDER_ATOMIC
  );
  Blockly["Python"].setupCode_[pinVar] = `${pinVar} = board.get_pin('d:
${pin}:s')\n`;
  Blockly["Python"].setupCode_[`angle_var_${pin}`] = `${angleVar} = 0`;
  return `${pinVar}.write(${degrees})\n${angleVar} =
${degrees}\ntime.sleep(1)\n`;
};
```

- servo_read_degrees:It generates a variable that keeps track of the angle that was sent to the servo, as the physical servos don't usually support position feedback, i.e., allow us to read the value directly from it.
- Servo_move_adafruit_tico: This generates the code specifically for Adafruit servos, through `servo_adafruit(pin)` from pyFirmata.

6. RFID:

The RFID blocks allow Blockly to generate code (for Arduino or Python) that:

- rfid_setup: Initializes the RFID reader on specified TX/RX pins.
- rfid_scan: Checks if an RFID tag is present.
- rfid_tag: Reads the raw tag data as a string.
- rfid_card: Extracts the numeric card number from the tag.

7. Motion Sensor:

   ultra_sonic_sensor_setup

   - Sets up the Trig and Echo pins for an ultrasonic sensor.
   - Adds Python code to:
     - Import pyfirmata, time
     - Initialize the board and start iterator
     - Configure pin modes (OUTPUT for Trig, INPUT for Echo)
     - Define the ultra_sonic_distance() function that sends a pulse and measures the echo time to calculate distance.

   ultra_sonic_sensor_motion

   - Returns the result of calling the ultra_sonic_distance() function — i.e., the measured distance in cm.

8. Joystick:

   joystick_setup

   - Sets X and Y pins (analog) and button pin (digital).
   - Enables analog reporting for joystick movement.
   - Sets digital input for button.
   - Defines helper variables and a function set_joystick_values() that:
   - Calculates joystick angle and distance.
   - Updates:
     - internal_variable_degrees → direction (0–360°)
     - internal_variable_isJoyStickEngaged → joystick moved or not
     - internal_variable_isJoystickButtonPressed → button pressed (LOW) or not

   joystick_angle

   - Returns:
     The joystick angle in degrees (0–360°), or 0 if not engaged.

   joystick_button

- Returns:
  True if joystick button is pressed, otherwise False.

joystick_engaged

- Returns:
  True if joystick has been moved beyond a threshold, otherwise False.

9. Buttons:

   button_setup

   - Sets up a GPIO pin as a button input.
   - Uses INPUT or INPUT_PULLUP depending on user setting.
   - Adds appropriate pinMode() (Arduino) or btn_pin.mode = INPUT (Python using pyFirmata).

   is_button_pressed

   - Returns a boolean expression checking if the button is pressed.
   - Uses digitalRead(pin) == LOW if INPUT_PULLUP, or == HIGH otherwise (Arduino).
   - Uses .read() == 0 or .read() == 1 (Python) depending on pull-up.

   release_button

   - Simulation-only placeholder block. No actual code is generated.
   - Used for: GUI or logic simulation in Blockly, not hardware interaction.

10. Digital Sensor:

    digital_read_setup

    - Sets up a digital pin for input reading.
    - Imports pyfirmata and initializes the Arduino board and iterator.
    - Configures pin PIN as a digital input (for switches, sensors, etc.).
    - Stores the pin object as digital_read_pin_<PIN>.

The changes ensure that Blockly can now generate Python code for these blocks in addition to Arduino code. Proper handling of operator precedence, default values, and conditional logic is included.

2.4 UPLOADER TRANSITION: AVRGIRL TO UPLOADMULTITOOL

Changed from Avrgirl uploader to UploadMultiTool uploader for Arduino. By switching uploaders, we were able to upload Arduino code more easily, with support for all updated drivers and modules.

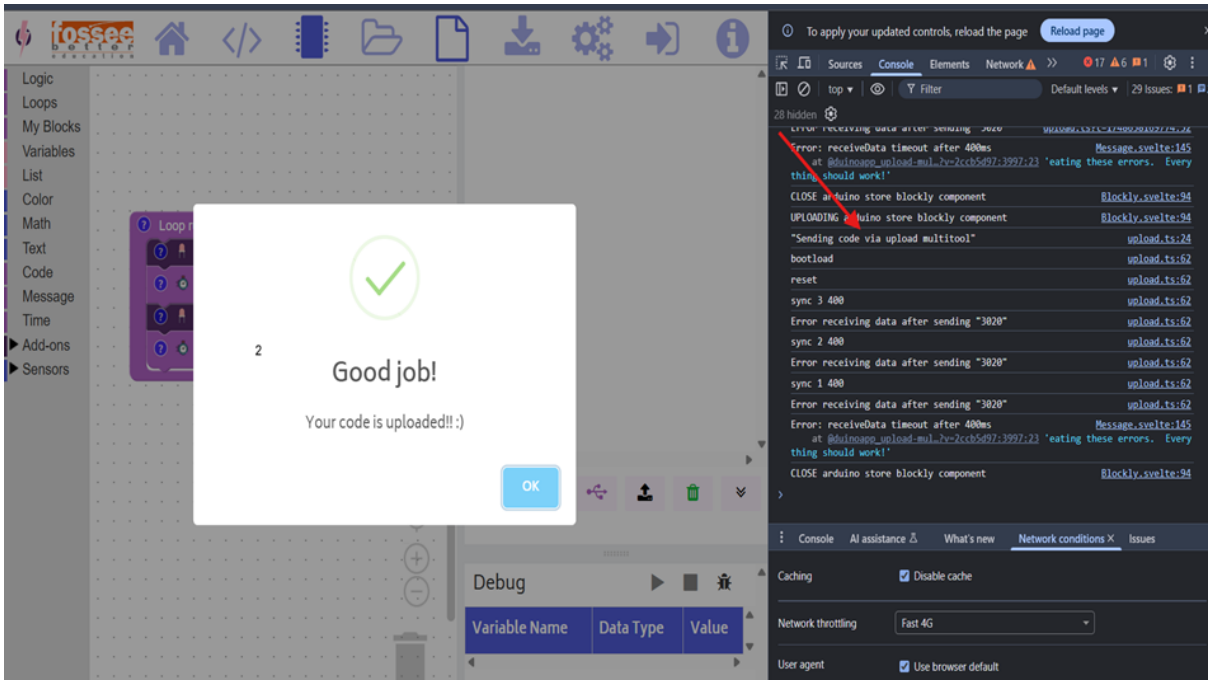


Fig 2.5 Uploader Transition: Avrgirl to UploadMultiTool

- Upload-MultiTool supports a wider range of microcontrollers, including ESP32 and newer Arduino boards, unlike Avrgirl which had limited board support
- Upload-MultiTool auto-detects and works with updated serial drivers across all platforms, reducing setup issues for users

Uploader Transition: Avrgirl → Upload-MultiTool:

- Avrgirl: Used for basic serial uploading to older Arduino boards.Anticlockwise: Rotates the entity to the left.
- Upload-MultiTool: A modern, flexible uploader supporting a wider range of microcontrollers with better serial support.

Steps to Implement the Uploader Change

1. Update Uploader Library:

- Replace avrgirl-arduino with @duinoapp/upload-multitool in your codebase.
- Ensure dependencies are updated in package.json and installed via npm.

2. Redesign Upload Logic:
   - Replace Avrgirl's upload syntax with Upload-MultiTool's upload() and WebSerialPortPromise.requestPort() API.
   - Implement new config parameters such as tool, cpu, speed, and bin for the specific microcontroller.

3. Enhance Cross-Platform Support:
   - Use Web Serial API to ensure compatibility across browsers (e.g., Chrome, Edge).
   - Check for support using WebSerialPortPromise.isSupported() and handle fallback gracefully.

4. Test and Validate Upload Flow:
   - Test upload on various boards like Arduino Uno, ESP32, and NodeMCU.
   - Log output using stdout.write() to verify hex writing, port detection, and board response.
   - Monitor upload stability, speed, and compatibility improvements over Avrgirl

2.5. SWITCHING COMPILATION BACKEND: ELECTROBLOCKS TO DUINO

Created visual flow to demonstrate both compilation methods. A visual representation to illustrate the transition from ElectroBlocks built-in compiler to the Duino cloud-based compiler helps users understand the new compilation flow effectively.

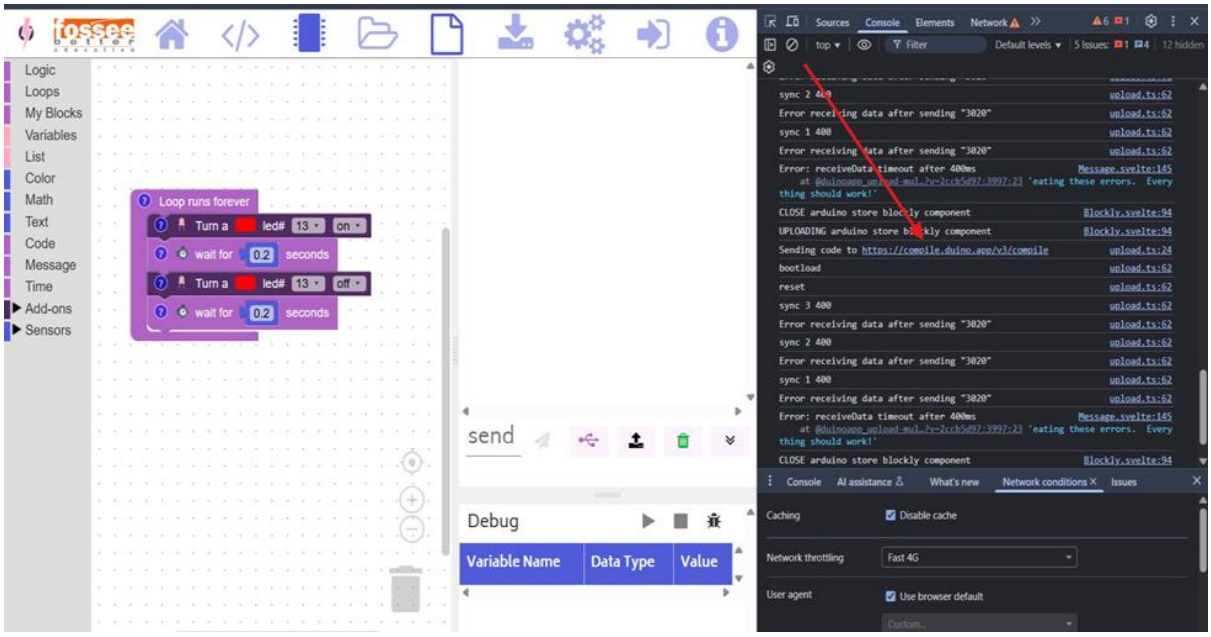


Fig 2.5 Implementation of Compilation Backend Transition Visualization

Objectives:

Visually demonstrate the difference between the legacy ElectroBlocks compiler and the new Duino cloud-based compiler..

Key Elements:

1. **Compilation Flow Diagram:** Use clear flowcharts or icons to represent each stage of compilation (input → processing → output).

2. **Before & After Comparison:** Create a split-view or timeline animation showing the compilation process using ElectroBlocks (offline/native) vs. Duino (cloud-based/API).

3. **Animation Phases:**
   - **Initial Phase:** Display code input from the user.
   - **ElectroBlocks Compilation**: Show local or simulated compilation with limited board support.
   - **Transition Phase:** Indicate changeover (e.g., arrow or switch icon).

- **Loop/Replay Option:** Allow looping to repeatedly

show the change in compilation flow.

Implementation Steps:

1. **Animation Planning:**
   - Visualize ElectroBlocks using an icon of a desktop or chip with limited outputs.
   - Visualize Duino using a cloud, API endpoint, and a code-to-hex arrow.
   - Highlight differences in speed, compatibility, and output clarity.
2. **User Interface:**
   - Place the animation in upload/compile button area or onboarding screen.
3. **Refinement:**
   - Make sure transitions between steps are smooth and visually engaging..
4. **Testing and Feedback:**
   - Share with users to verify if it improves understanding of the compiler transition.

By following these steps, you can create an effective and user-friendly mode that clearly communicates the transition from ElectroBlocks native compiler to Duino's cloud-based solution, enhancing both understanding and trust in the new system.

.

2.6. DEVELOPMENT OF CUSTOM UPLOAD LIBRARY FOR ELECTROBLOCKS

This feature allows ElectroBlocks to intelligently select and load the necessary libraries based on the structure and header content of the user's code. It enhances flexibility and supports a variety of microcontroller configurations and use cases without requiring user interaction on the frontend.

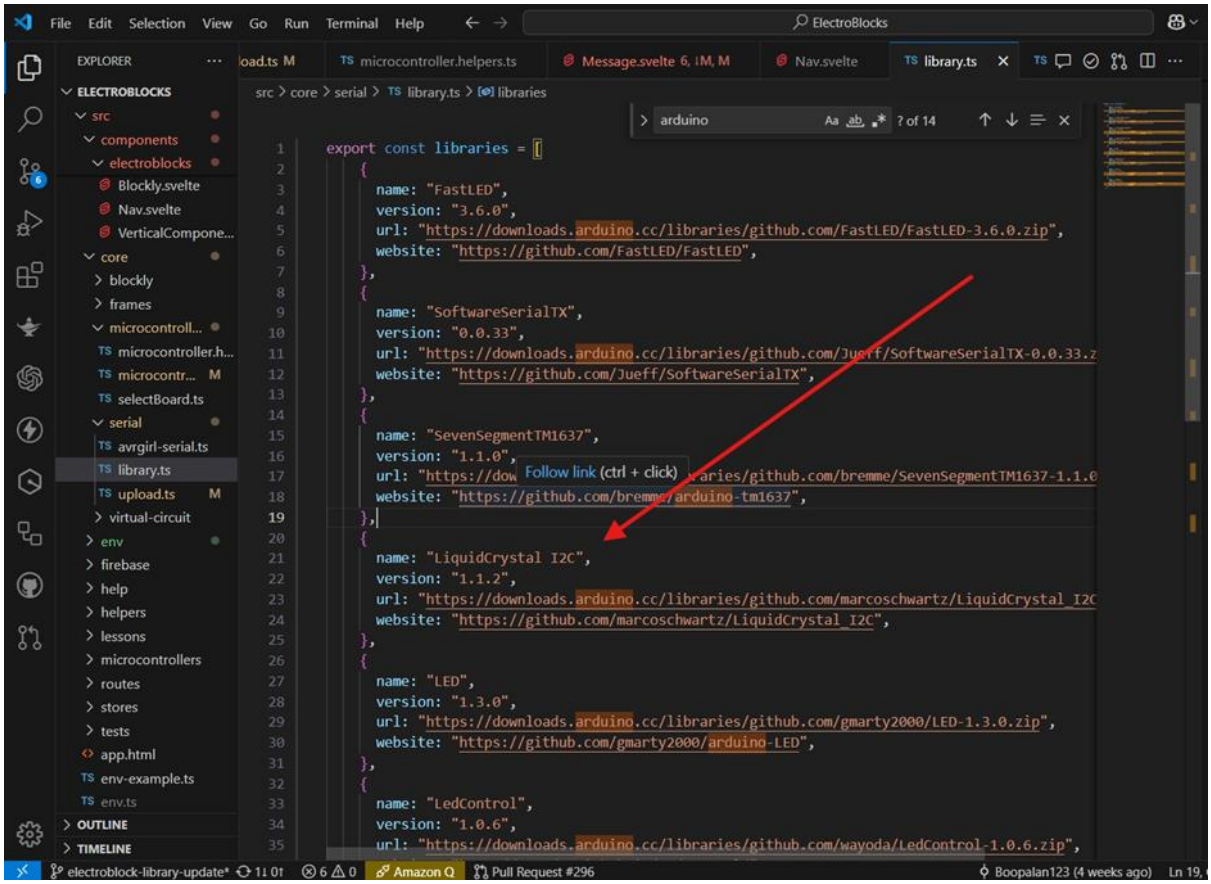


fig 2.6 Development of Custom Upload Library for ElectroBlocks

Steps to Implement the Library Auto-Selection

1. **Analyze Code Header:**

   - Extract key identifiers (e.g., #include, import, or specific syntax).
   - Detect the library dependencies directly from the header or keywords in the user's code.

2. **Map Header to Library:**

- Create a backend mapping system that links code headers to corresponding library files.
- Avoid redundancy by checking for already-included libraries.

3. **Load Library Programmatically:**

- Automatically inject the required library files into the code compilation process.
- Avoid redundancy by checking for already-included libraries.

4. **Update Compilation Flow:**

- Integrate this logic before passing the code to the compiler (Duino backend).
- Ensure compatibility with cloud-based compilation by structuring payloads correctly.

5. **Test and Validate:**

- Rigorously test the mapping logic with various code samples.
- Ensure compatibility with cloud-based compilation by structuring payloads correctly.

Conceptual Approach

1. **Header-Based Detection:**

   · The backend scans for specific keywords or structure in the code to infer which libraries are needed.

   · Examples: #include<LiquidCrysrtal.h> . It takes library which match “LiquidCrysrtal” like header.

2. **Backend-Driven Decision Logic:**

   · The user doesn’t need to choose libraries manually.

   · The backend ensures accuracy and prevents compilation errors due to missing dependencies..

3. **Modular and Scalable Design:**

   · Easily extendable to support more libraries in the future.

   · Structured to support Arduino, ESP32, and ESP8266 platforms.

User Experience Considerations

1. **No Manual Configuration:**

    - Users simply write code as usual. Library management happens automatically in the background.

2. **Error Handling:**

    - If no matching library is found, a descriptive error message is returned to the frontend or console.

3. **Consistency and Reliability:**

    - Ensures that users get a smooth experience across different boards and peripherals without extra setup.

By implementing backend-based automatic library selection, ElectroBlocks offers a seamless and intelligent experience for code compilation. This architecture removes the need for frontend dropdowns or manual intervention, ensuring higher reliability, less user confusion, and faster development cycles.

2.7 DROPBOX FOR ESP32 MICROCONTROLLER SELECTION

When users access the platform, it is essential to provide an option to switch between different supported microcontrollers, such as ESP32, Arduino MEGA, and Arduino UNO. This dropdown allows the platform to adapt its behavior based on the selected board.

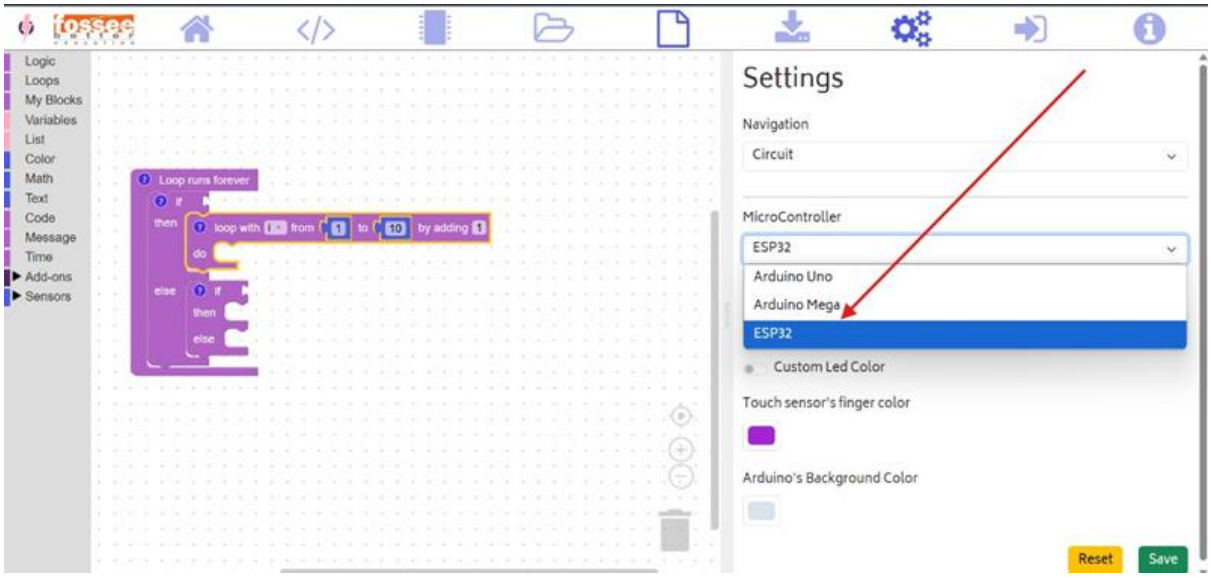


fig 2.7 Dropbox for ESP32 microcontroller selection

Key Functionalities:

·**Microcontroller Dropdown:**

Offers a list of supported boards (e.g., Arduino UNO, ESP32, Arduino MEGA) for user selection.

· **Dynamic Block Management:**

Upon selection, the system enables only those blocks or features compatible with the selected microcontroller.

For example, certain blocks not supported by ESP32 will be hidden or disabled.

· **Backend Integration:**

The selection updates the configuration used for compiling and uploading code.

· **Automatic Adaptation:**

Prevents users from mistakenly using incompatible blocks, ensuring a smooth and error-free development experience.

2.8 TOOLBOX VISIBILITY

One of the essential usability features implemented was Toolbox Visibility Management based on the selected board type and programming language. In a visual programming platform like ElectroBlocks, users rely on the toolbox to access available blocks for their target hardware and coding language. Displaying irrelevant blocks not only creates confusion but also increases the chances of code generation errors. Hence, implementing dynamic toolbox filtering was critical to enhancing the user experience.

2.8.1 Problem Identification

Initially, ElectroBlocks displayed all available blocks in the toolbox regardless of the selected board (Arduino Uno) Fig 2.5.1 or language (C or Python). This caused several issues:

- Users could drag blocks that were not supported by their chosen board or language.The visual environment became cluttered with unnecessary options.
- Certain blocks generated invalid or non-compilable code when used with unsupported board configurations.

This led to the need for a dynamic system that would filter and update the toolbox items in real-time based on the selected board and language.

2.8.2 Implementation Overview

To solve this, I worked on modifying the logic in multiple files across the ElectroBlocks codebase[4].

The primary changes were made in the following files:

- src/routes/blockly/settings/+page.svelte
- src/stores/code.store.ts

- src/firebase/model.ts

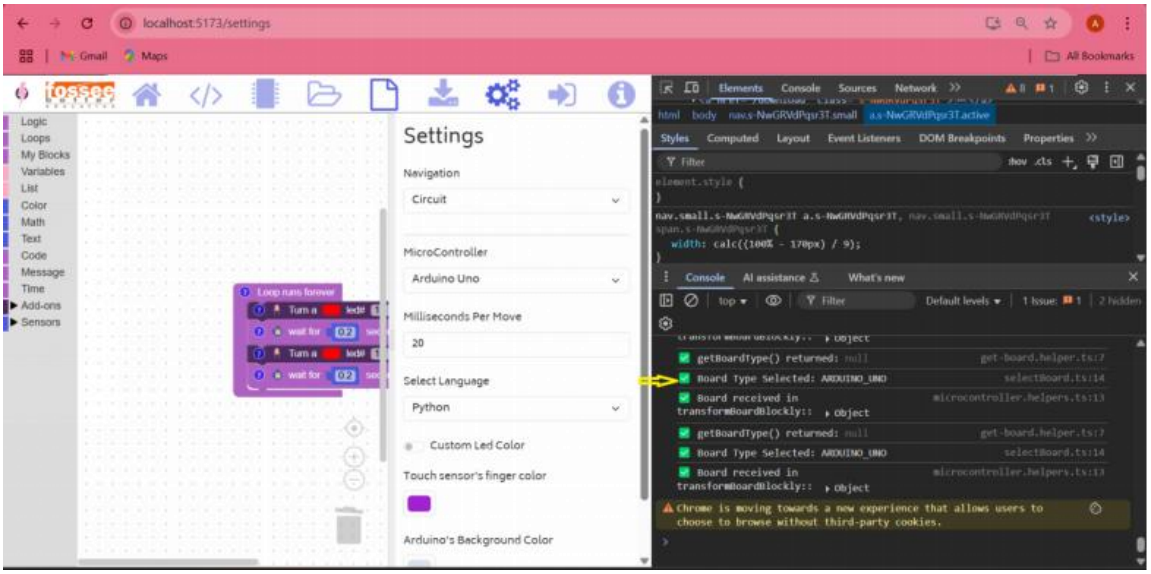


Fig. 2.8.1

2.8.3 Outcome

With these changes:

- The toolbox now automatically adapts to the selected board and language.
- Irrelevant blocks are hidden, improving usability and reducing confusion.
- Errors caused by unsupported blocks in the workspace were minimized.
- The UI became more intuitive and aligned with actual hardware capabilities.

2.9 RESTRICTION OF COMPILATION TO C

As part of the Python integration into ElectroBlocks, a critical requirement was to ensure that code compilation (upload) is permitted only for the C language, since ElectroBlocks' current upload mechanism is designed exclusively for C/C++ (Arduino) environments and does not support Python uploads.Allowing users to initiate uploads while in Python mode would result in failure, confusion, or unexpected behavior. Hence, a safeguard was implemented to restrict the upload feature to only when C is selected as the programming language, and to gracefully notify the user if they attempt otherwise.

2.9.1 Workflow

Here is how the logic now works after my contribution:

1. User selects a language in the UI.

2. settings.language is updated and synced across components.

3. When the user clicks "Upload":

4. Message.svelte checks selectedLanguage as shown Fig. 2.6.1

5. If "c": upload proceeds.

6. If "python": upload is blocked, and the user is alerted.

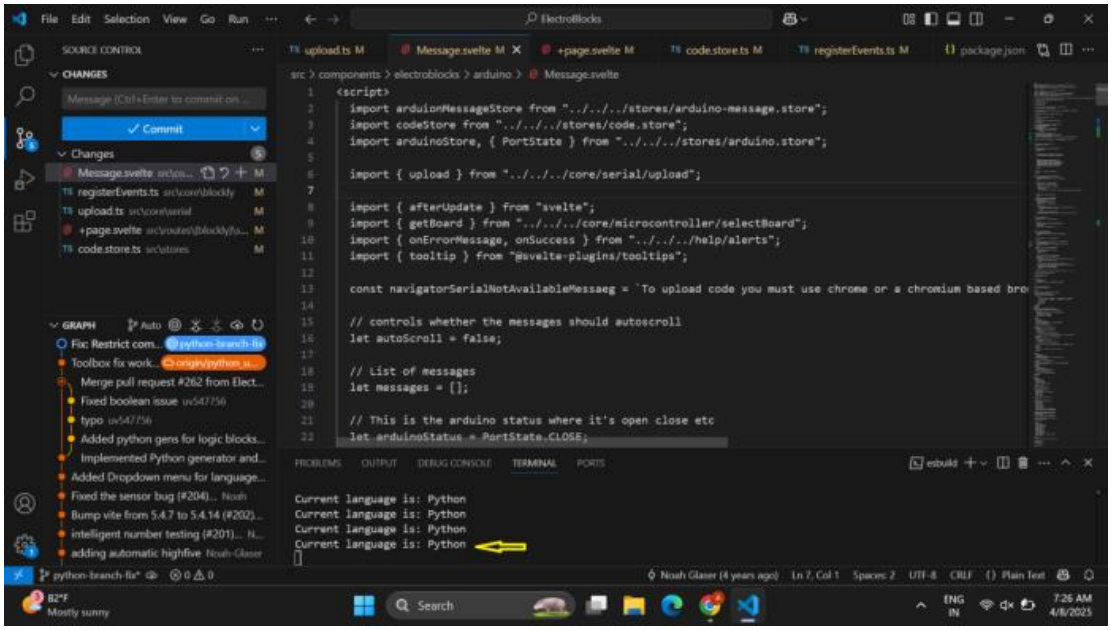


Fig. 2.9.1

2.9.2 Implementation overview

To solve this, I worked on modifying the logic in multiple files across the ElectroBlocks codebase[5]. The primary changes were made in the following files:

- src/components/electroblocks/arduino/Message.svelte
- src/routes/(blockly)/settings/+page.svelte
- src/stores/settings.store.ts
- src/core/blockly/registerEvents.ts
- src/core/blockly/toolbox.ts
- src/core/serial/upload.ts

2.9.3 Impact

- Prevents upload-related errors in unsupported Python mode as shown in Fig. 2.6.2.
- Provides clear user guidance and error handling.
- Establishes groundwork for future Python upload support.
- Aligns with ElectroBlocks' principle of beginner-friendliness by avoiding silent failures.

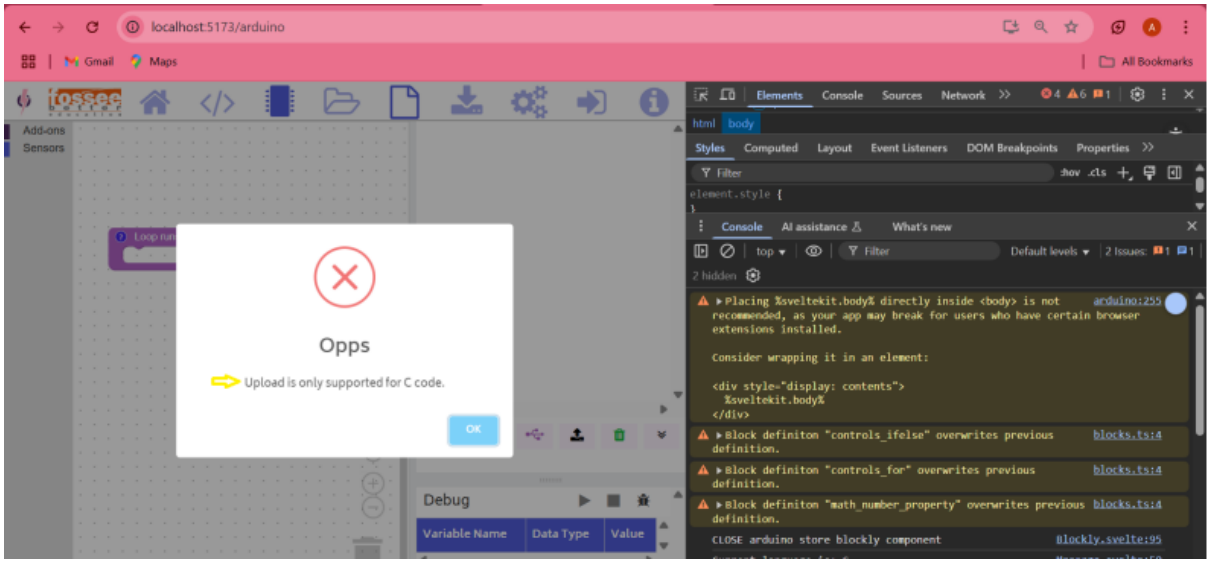


Fig. 2.9.2

2.10 JUPYTER NOTEBOOK FOR ELECTROBLOCK DEMO

As part of the Python integration effort, one of the important deliverables was to demonstrate ElectroBlocks Python functionality through Jupyter Notebook examples such as LEDs, RGB LEDs, and servo motors. These examples are intended to help users, especially beginners, understand how to interact with hardware using Python code generated via ElectroBlocks.. This is focused on creating a few well-structured, easy-to-understand hardware control examples using ElectroBlocks for educational and demonstrative purposes, hosted under the FLOSS Arduino Book repository by FOSSEE[6].

2.10.1 Implementation

To implement the Jupiter notebooks,the following prerequisites were to ensured

- Each notebook should begins with the installation command:

```
[1]: !pip install electroblocks==0.1.2
```

to install the ElectroBlocks Python Library.

- Firmware Updated
  The microcontroller Arduino Uno was flashed with the correct firmware as shown in the Fig 2.7.1,2.7.2 that allows ElectroBlocks Python commands to communicate with it.

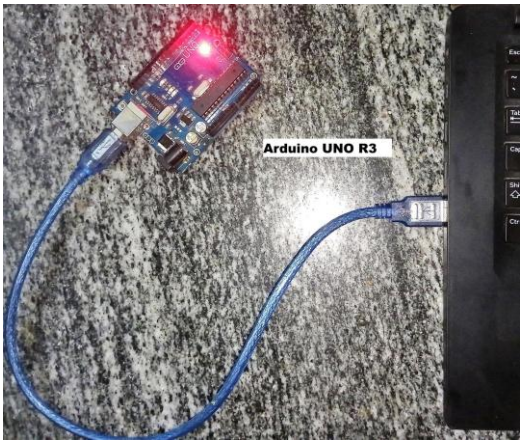


Fig 2.10.1

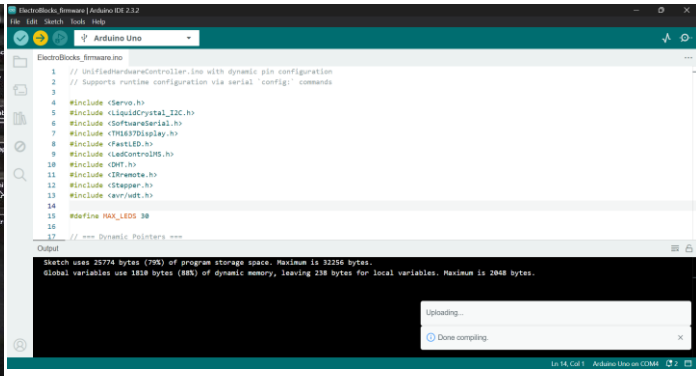


Fig 2.10.2

- Arduino Libraries Installed:
  While the notebooks focus on the Python code side, Arduino library installation is necessary for projects using both C and Python. The following libraries were recommended and noted in the documentation:

Library	GitHub Link
LiquidCrystal_I2C	https://github.com/johnrickman/LiquidCrystal_I2C
FastLED	https://github.com/FastLED/FastLED
LED Matrix	https://github.com/shaai/Arduino_LED_matrix_sketch
DHT Sensor	https://github.com/adafruit/DHT-sensor-library
IRremote	https://github.com/Arduino-IRremote/Arduino-IRremote
Stepper	https://github.com/arduino-libraries/Stepper

- Folder is created with three notebooks LED Blink,RGB LED Color Cycle,Servo Sweep as shown in Fig 2.7.3 Each notebook was created with detailed code cells and markdown instructions to help users set up and run hardware interaction projects.

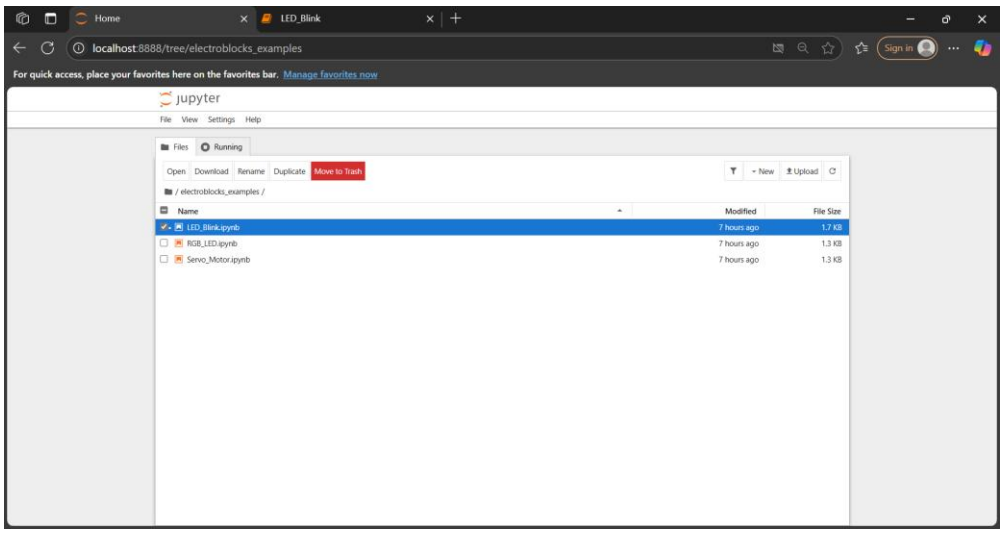


Fig 2.10.3

1. First notebook was created with markdown instructions to help users set up that is block formation as shown in Fig 2.7.4

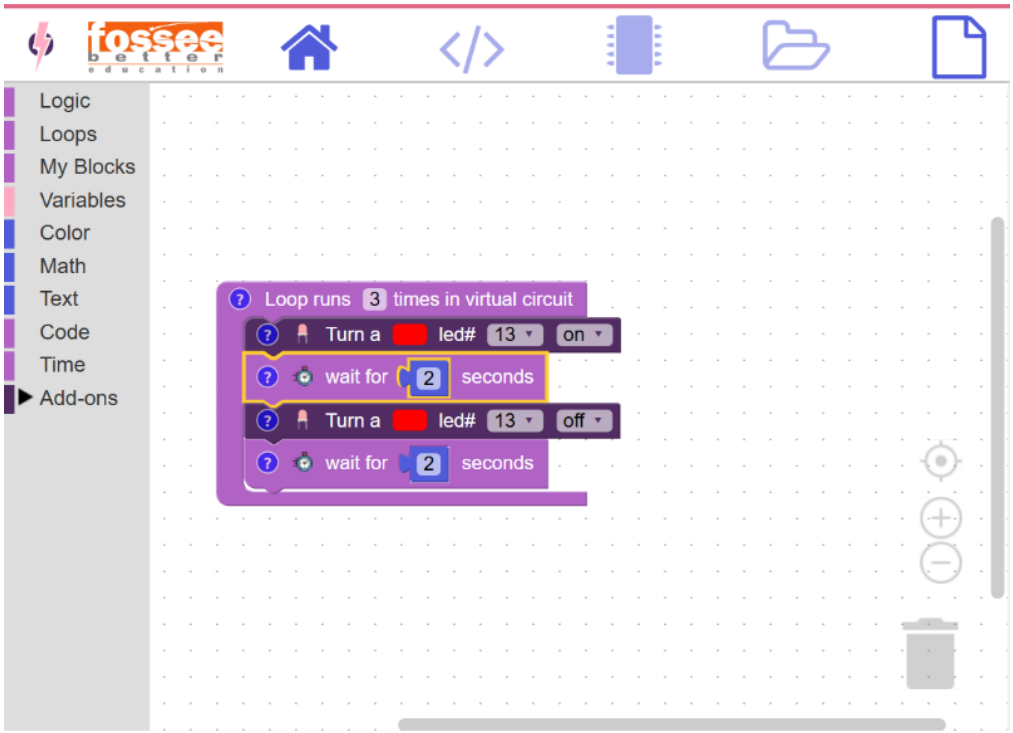


Fig 2.10.4

and detailed code cells where the code is copied from the Electroblock demo page for the code language is selected as python and respective led blink block formation as shown Fig 2.7.5 and run hardware interaction projects.

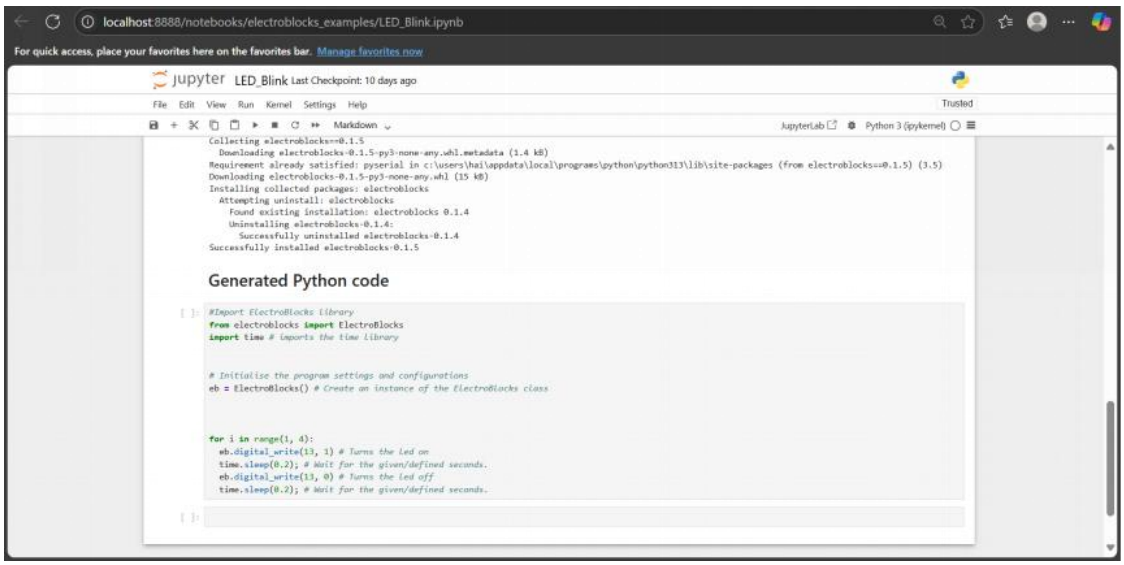


Fig 2.10.5

2. Second notebook was created with markdown instructions to help users set up that is block formation as shown in Fig 2.7.6

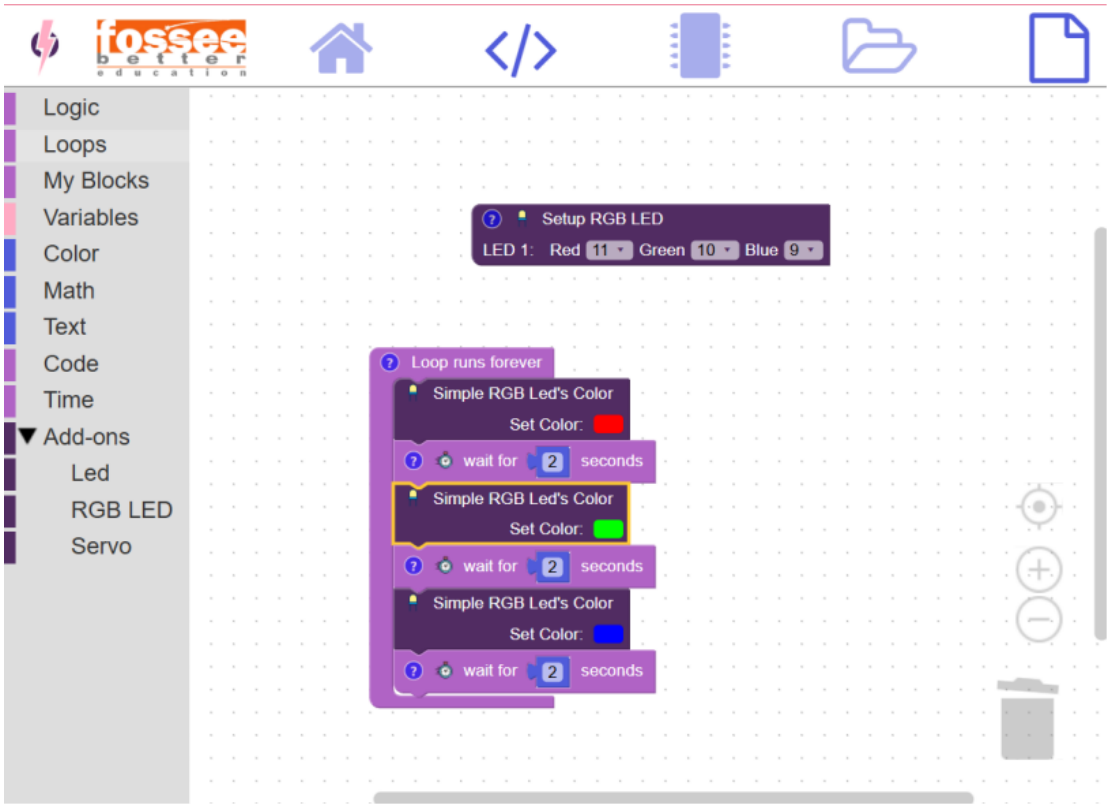


Fig 2.10.6

and detailed code cells where the code is copied from the Electroblock demo page for the code language is selected as python and respective RGB Led Cycle block formation as shown Fig 2.7.7 and run hardware interaction projects.

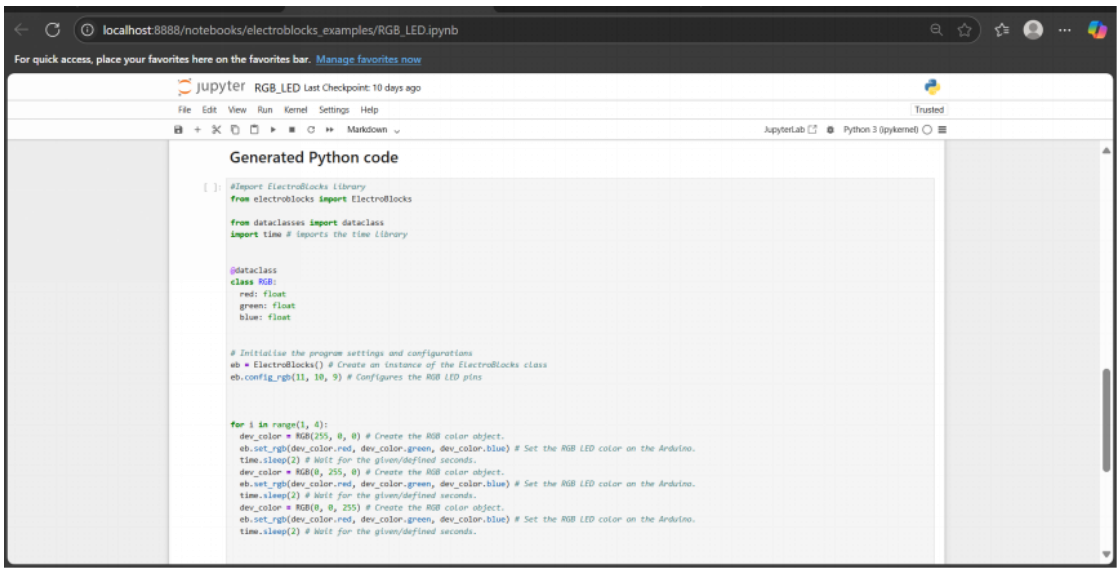


Fig 2.10.7

3. Each notebook was created with markdown instructions to help users set up that is block formation as shown in Fig 2.7.8

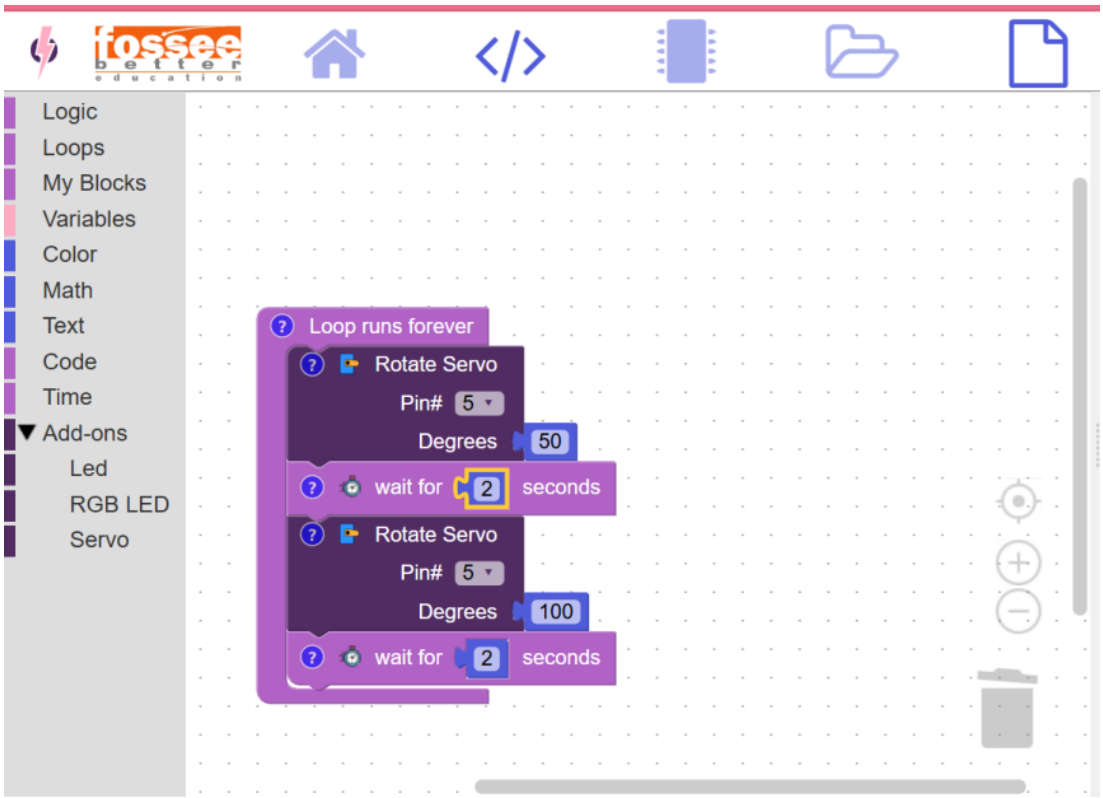


Fig 2.10.8

and detailed code cells where the code is copied from the Electroblock demo page for the code language is selected as python and respective Servo motor Cycle block formation as shown Fig 2.7.9 and run hardware interaction projects.

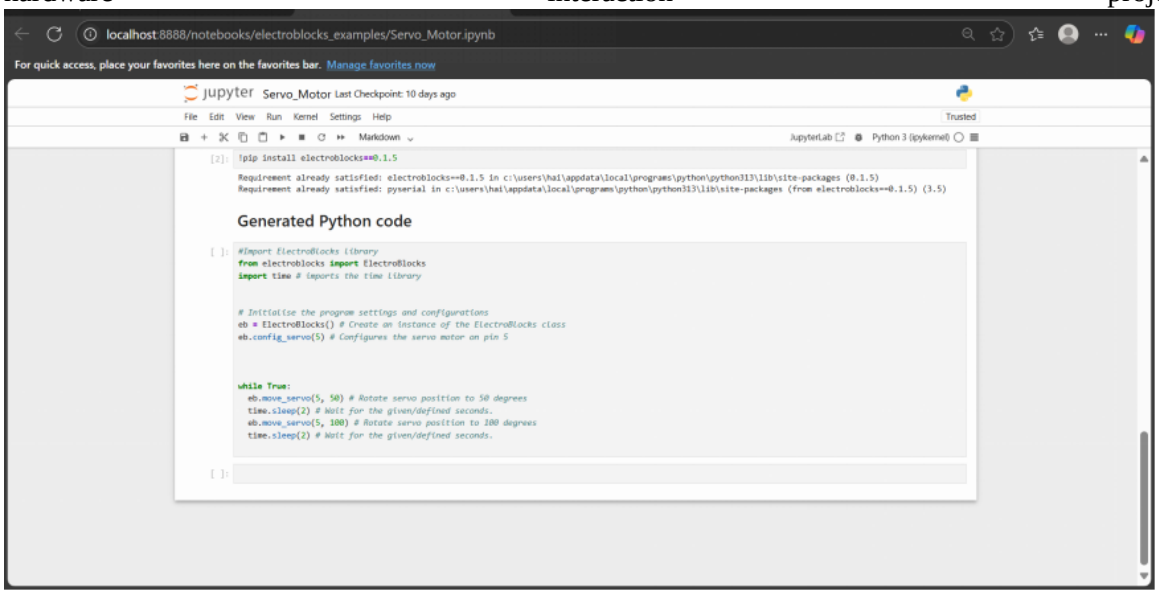


Fig 2.10.9

While running the jupyter notebook check whether the kernel status is idle and also check is it trusted in the top right corner as shown in the Fig 2.7.10.The final ElectroBlocks Python examples for Servo, RGB LED, and LED Blink is shown in the reference link[7].

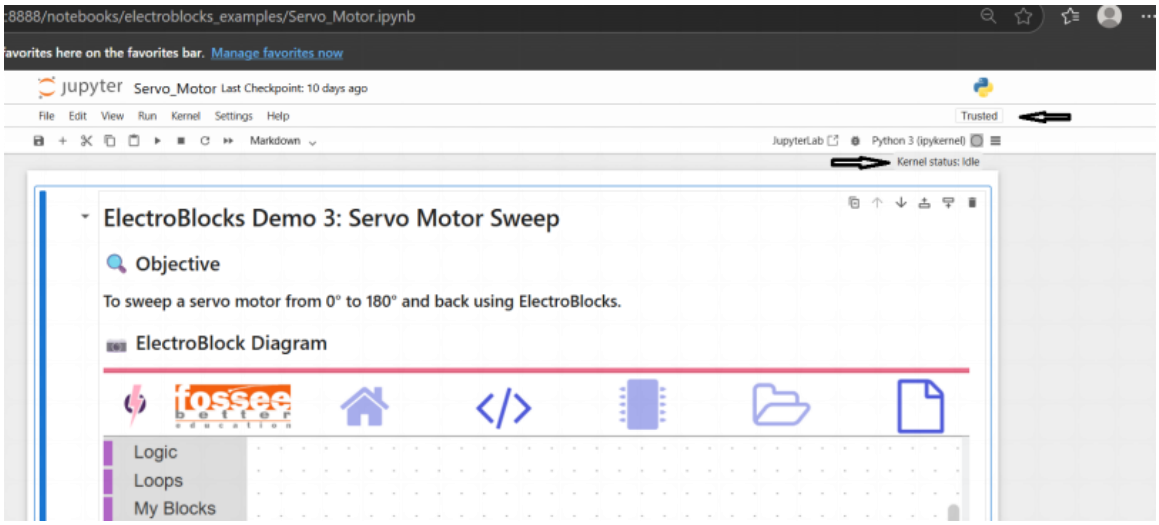


Fig 2.10.10

2.10.2 Impact

- Allows users to take their visually programmed Python code and run it in a Jupyter environment[6].
- Enhances documentation and experimentation by enabling markdown notes, cell-based debugging, and visual outputs.
- Makes ElectroBlocks suitable for educational institutions that rely heavily on Jupyter Notebooks for lab work and tutorials.

2.11 ELECTROBLOCKS COMPANION APP

2.11.1 Endpoint to show the available ports

This system consists of:

- A backend REST API endpoint for listing ports (/ports).
- A dynamic tray menu that reflects all connected ports.
- A refresh mechanism to update the port list at runtime.
- A detailed information dialog for each selected port.

1. Tray Menu Port Listing

The Electron tray menu dynamically shows a list of available serial ports (COM, ttyUSB, etc.).

- Each menu item displays the port path and manufacturer (e.g., /dev/ttyUSB0 (Arduino LLC)).
- Clicking on a port opens a dialog box showing detailed information such as:

Path Manufacturer Serial Number Vendor ID Product ID

2. Refresh Ports

- A dedicated "Refresh Ports" option is added below the port list.
- When clicked, it updates the tray menu with the current list of ports without restarting the app.

3. View JSON Endpoint

- A "View Ports (JSON)" option is also added to the tray.
- Opens the API response in the browser at “http://localhost:4000/ports” for debugging or advanced users.

4. Express /ports Endpoint

- Serves a list of all detected serial ports in JSON format.

```
const staticPath = path.join(__dirname, "build");

expressApp.use(express.static(staticPath));
expressApp.use(express.json({ limit: "10mb" }));

expressApp.get("/ports", async (req, res) => {
  try {
    const ports = await SerialPort.list();
    res.json(
      ports.map((port) => ({
        path: port.path,
        manufacturer: port.manufacturer || "Unknown",
        serialNumber: port.serialNumber || "N/A",
        vendorId: port.vendorId || "N/A",
        productId: port.productId || "N/A",
      }))
    );
  } catch (error) {
    res.status(500).json({ error: error.message });
  }
});
```

Fig.2.11.1

Endpoint using express.js and serialport function

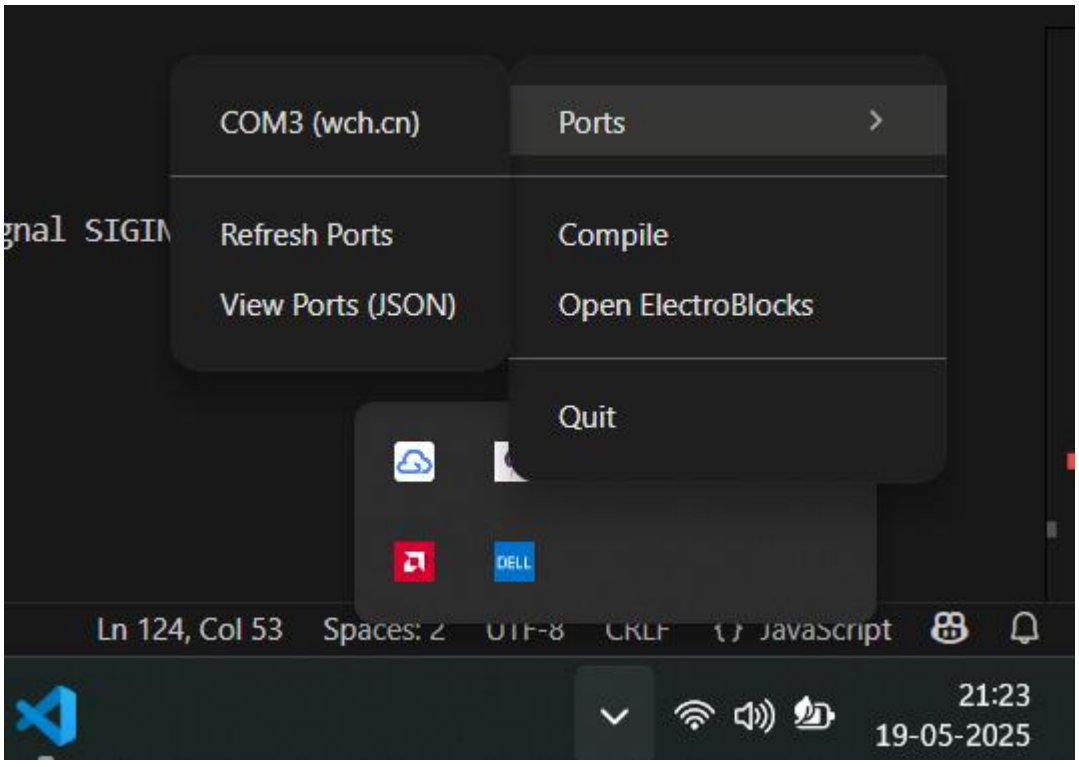


Fig.2.11.2

Tray-app that shows the ports endpoint

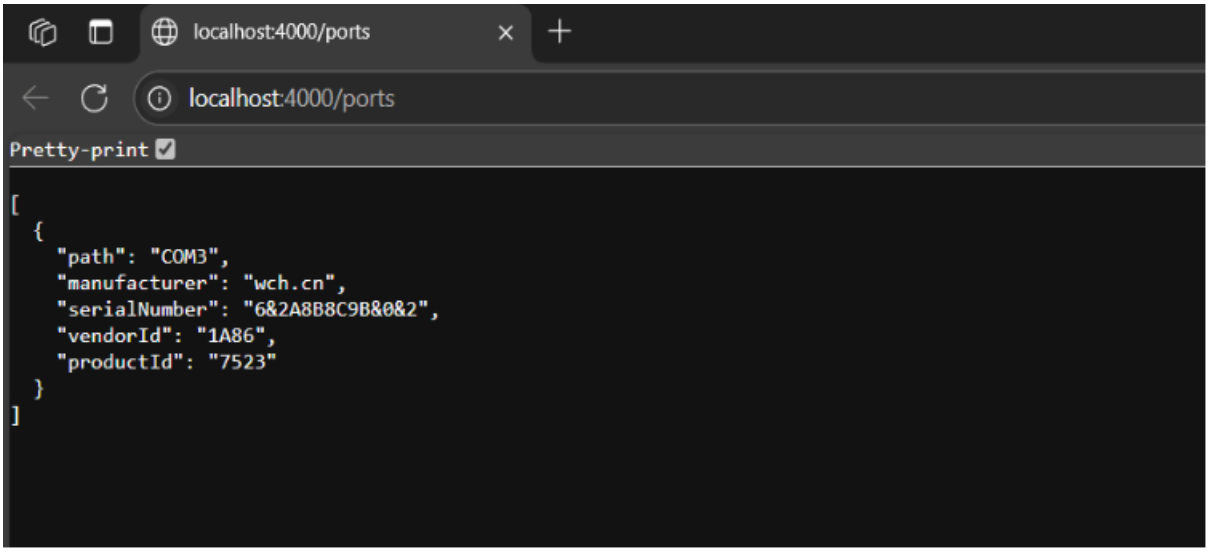


Fig.2.11.3

JSON format of the available ports

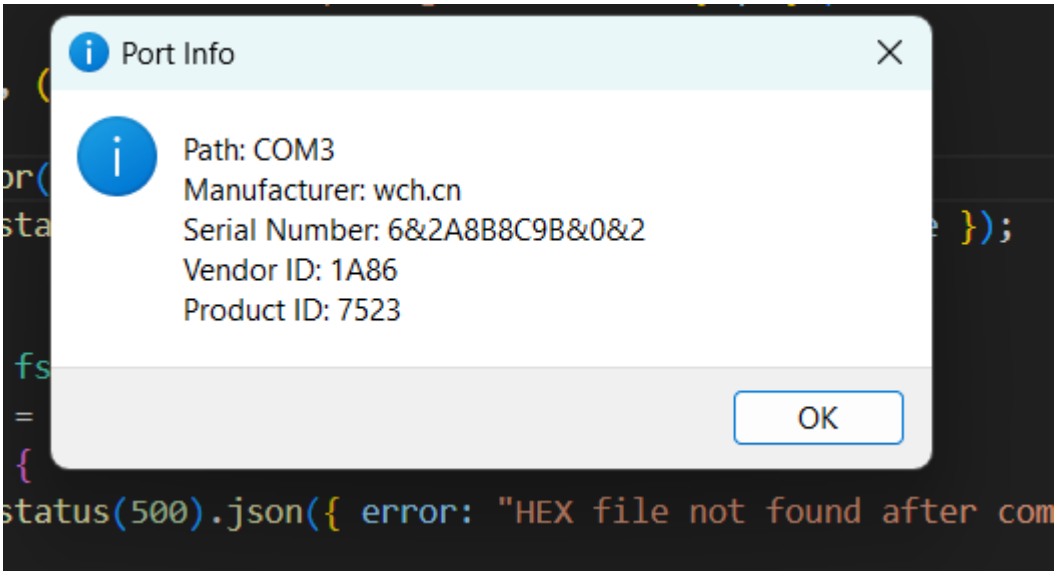


Fig.2.11.4

Port info by clicking the ports

2.11.2 Build a offline Electroblocks setup

This integration consists of:

- A customized build folder that embeds all fonts, styles, icons, and scripts.
- An automated unzip-and-cleanup system for build.zip.
- Static hosting of the interface using Express.

1. CDN Replacement and Build Folder Generation

- A complete audit of the ElectroBlocks front-end revealed external dependencies which were removed and replaced with local assets.

Original	Replacement
Google Fonts (via fonts.googleapis.com)	Local .woff2 files under /fonts/biryani/
Bootstrap CSS	Local bootstrap.min.css in /css/
Font Awesome CSS/Fonts	Local copies in /font-awesome/

- app.html and global.css were edited to reference local files using relative paths (/static/css/..., /fonts/...).
- After replacing these dependencies, the build was generated using:
  npm run build
- This produced a self-contained build/ folder, ready to be served locally.

2. Packaging and Deployment

To ensure the final app remained lightweight and easy to distribute:

- The build/ folder was zipped into build.zip.
- Only the zip was shipped with the Electron app binary.
- On the first launch of the app, this zip is:
  - Automatically extracted to a /build/ directory.
  - Deleted immediately after extraction to save disk space.

3. Express Server Integration

- The Electron app's built-in Express server was configured to host the extracted build folder:

“expressApp.use(express.static(path.join(__dirname, "build")));”

- When users click “Open ElectroBlocks” from the tray, it launches:”http://localhost:4000/”
- This opens the offline version of ElectroBlocks in their default browser, powered entirely by local files.

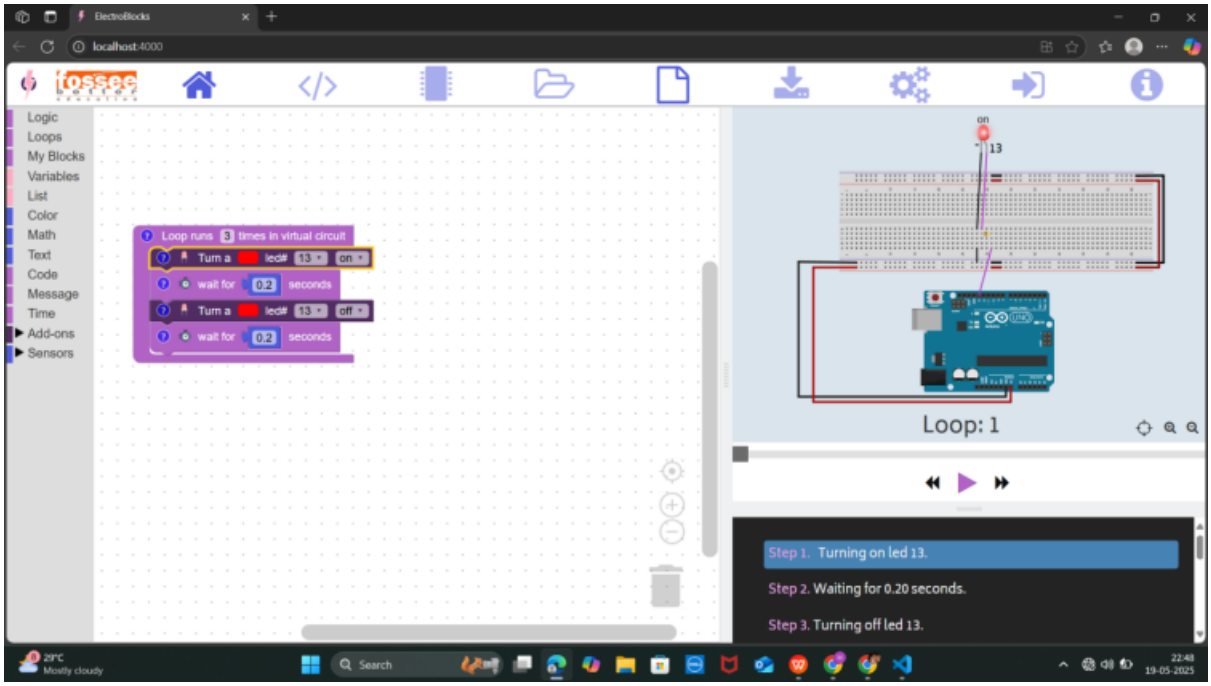


Fig.2.11.5

offline Electroblocks

2.11.3 Arduino compilation system for the Electroblocks companion app

1. File Watcher System (startInoWatcher)

- A periodic check (every 3 seconds) scans the user's Downloads folder for a file named electroblocks_code.ino.
- If found:A folder named downloads/electroblocks_code/ is created if it doesn't exist.
- The .ino file is moved to this folder.
- The tray menu is rebuilt to show the Compile option.
- Why this matters: Ensures users don’t need to perform any file management manually. The system maintains proper folder structure required by Arduino CLI (sketch folder name must match .ino file name).

```
// Auto move .ino file from system Downloads to project folder
const startInoWatcher = () => {
  setInterval(() => {
    if (fs.existsSync(downloadedInoPath)) {
      if (!fs.existsSync(inoFolder)) fs.mkdirSync(inoFolder, { recursive: true });
      try {
        fs.renameSync(downloadedInoPath, inoFilePath);
        console.log("Moved .ino file to project downloads folder.");
        buildTrayMenu();
      } catch (err) {
        console.error("Error moving .ino file:", err);
      }
    }
  }, 3000); // Every 3 seconds
};

const unzipBuildIfNeeded = () => {
  const zipPath = path.join(__dirname, "build.zip");
```

Fig.2.11.6

Arduino CLI: Installation Guide

- In order to compile .ino files headlessly, the Arduino CLI must be installed and available in your system PATH. Here's how to install it for your operating system.
- Go to the official Arduino CLI releases page: https://arduino.github.io/arduino-cli
- Download the appropriate ZIP for Windows (e.g., arduino-cli_latest_Windows_64bit.zip).
- Extract the ZIP to a folder of your choice (e.g., C:\arduino-cli).
- Add the folder to your system PATH
- Open Command Prompt, and run: arduino-cli version

Post-Installation Setup

- Initialize CLI configuration:
  - arduino-cli config init
- Install the Arduino AVR core (e.g., for Uno):
  - arduino-cli core update-index
  - arduino-cli core install arduino:avr

2. Arduino CLI Compilation Endpoint (/compile)

- The Express back-end exposes a REST API at /compile. When triggered:
- By default it is UNO if the different is connected it will automatically detect

```
// Detect board using arduino-cli
exec("arduino-cli board list", (err, stdout, stderr) => {
  if (err) {
    console.error("Board detection failed:", stderr || err.message);
    return res.status(500).json({ error: "Board detection failed." });
  }

  let fqbn = "arduino:avr:uno"; // Default
  const lines = stdout.split("\n");

  for (const line of lines) {
    if (line.includes("arduino:avr:uno")) {
      fqbn = "arduino:avr:uno";
      break;
    } else if (line.includes("arduino:avr:mega")) {
      fqbn = "arduino:avr:mega";
      break;
    } else if (line.includes("arduino:avr:nano")) {
      fqbn = "arduino:avr:nano";
      break;
    }
  }
```

Fig.2.11.7

- Next ,It runs the following command: (for now choose the default one) ”arduino-cli compile --fqbn arduino:avr:uno --output-dir downloads/electroblocks_code downloads/electroblocks_code”
- On success, it returns JSON including:

  hexFileName

  Full path to the .hex file

  - If it fails, it logs the error and returns a 500 response with a detailed message.

```
const compileCmd = `arduino-cli compile --fqbn ${fqbn} --output-dir ${inoFolder} ${inoFolder}`;
console.log(`Detected board. Compiling with FQBN: ${fqbn}`);

exec(compileCmd, (error, stdout, stderr) => {
  if (error) {
    console.error("Compile error:", stderr || error.message);
    return res.status(500).json({ error: stderr || error.message });
  }

  const files = fs.readdirSync(inoFolder);
  const hexFile = files.find((f) => f.endsWith(".hex"));
  if (!hexFile) {
    return res.status(500).json({ error: "HEX file not found after compilation." });
  }

  res.json({
    message: "Compilation successful",
    fqbn,
    hexFileName: hexFile,
    hexFilePath: path.join(inoFolder, hexFile),
  });
});
});
```

Fig.2.11.8

- Why this matters: Makes compilation one-click (via tray or REST), removes dependency on Arduino IDE, and supports integration with CI/CD or custom up-loaders.

3. Download Compiled HEX File (/download-hex)

- A dedicated endpoint allows users or the frontend to download the compiled .hex file:
- Ensures .hex exists before sending.
- Triggers a browser download with correct filename.

```
expressApp.get("/download-hex", (req, res) => {
  if (!fs.existsSync(hexFilePath)) {
    return res.status(404).send("HEX file not found. Please compile first.");
  }
  res.download(hexFilePath, "electroblocks_code.hex");
});

expressApp.get("*", (req, res) => {
  const indexPath = path.join(staticPath, "index.html");
  fs.readFile(indexPath, "utf8", (err, data) => {
    if (err) {
      res.status(500).send("Could not load app");
    } else {
      res.send(data);
    }
  });
});
```

Fig.2.11.9

- Why this matters: Enables integration with online firmware uploaders or bootloaders.

4. Tray Menu Integration with Local_Compile Option

- The Electron tray automatically adds a “Local_Compile” option only if the .ino file exists in the correct path:
- Clicking it opens the /local_compile endpoint in a browser.
- Prevents invalid states (e.g., trying to compile when no .ino is present).
- Why this matters: Maintains context-aware UX—shows relevant options only when meaningful.

5. Tray Menu Integration with Compile Option

- The Electron tray always includes a “Compile” option, regardless of whether the .ino file exists.
- Clicking it opens the /compile endpoint in the default browser.
- The /compile route itself handles file existence checks and responds with an appropriate message if the .ino file is missing.

```
const buildTrayMenu = () => {
  const compileOption = fs.existsSync(inoFilePath)
    ? {
        label: "Compile",
        click: () => {
          shell.openExternal("http://localhost:4000/compile");
        },
      }
    : null;

  const menuTemplate = [
```

Fig.2.11.10

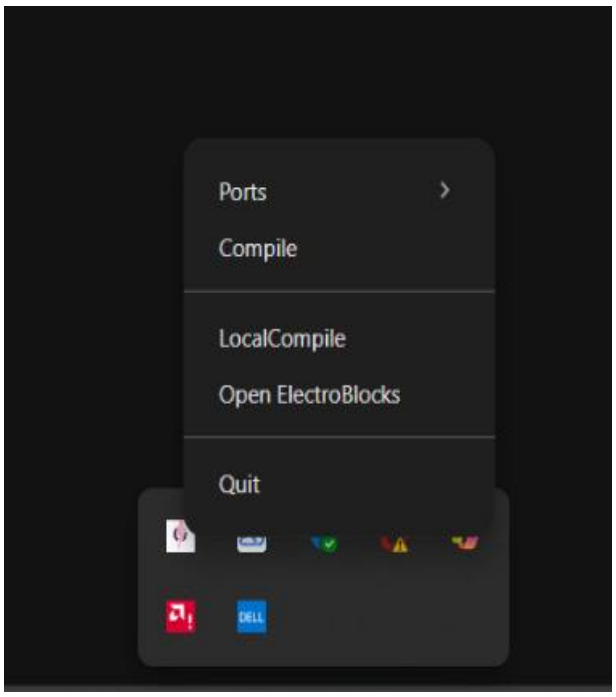


Fig.2.11.11

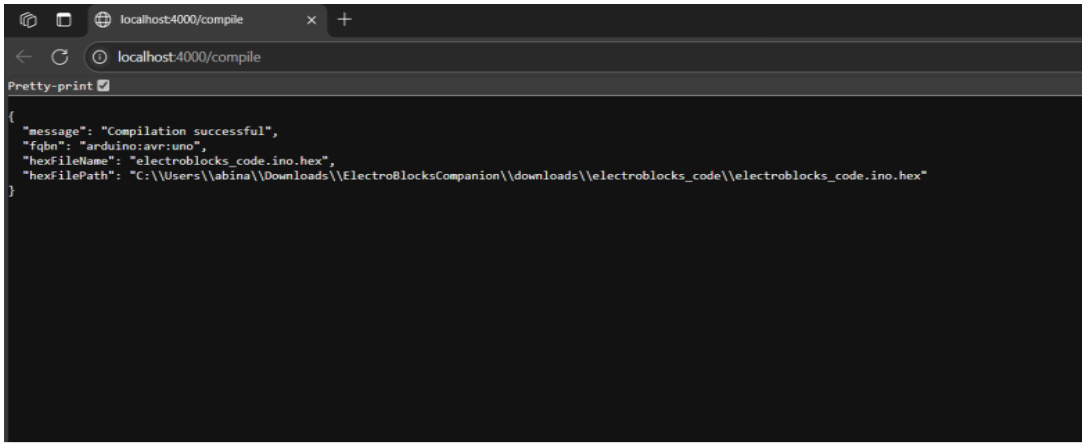


Fig.2.11.12

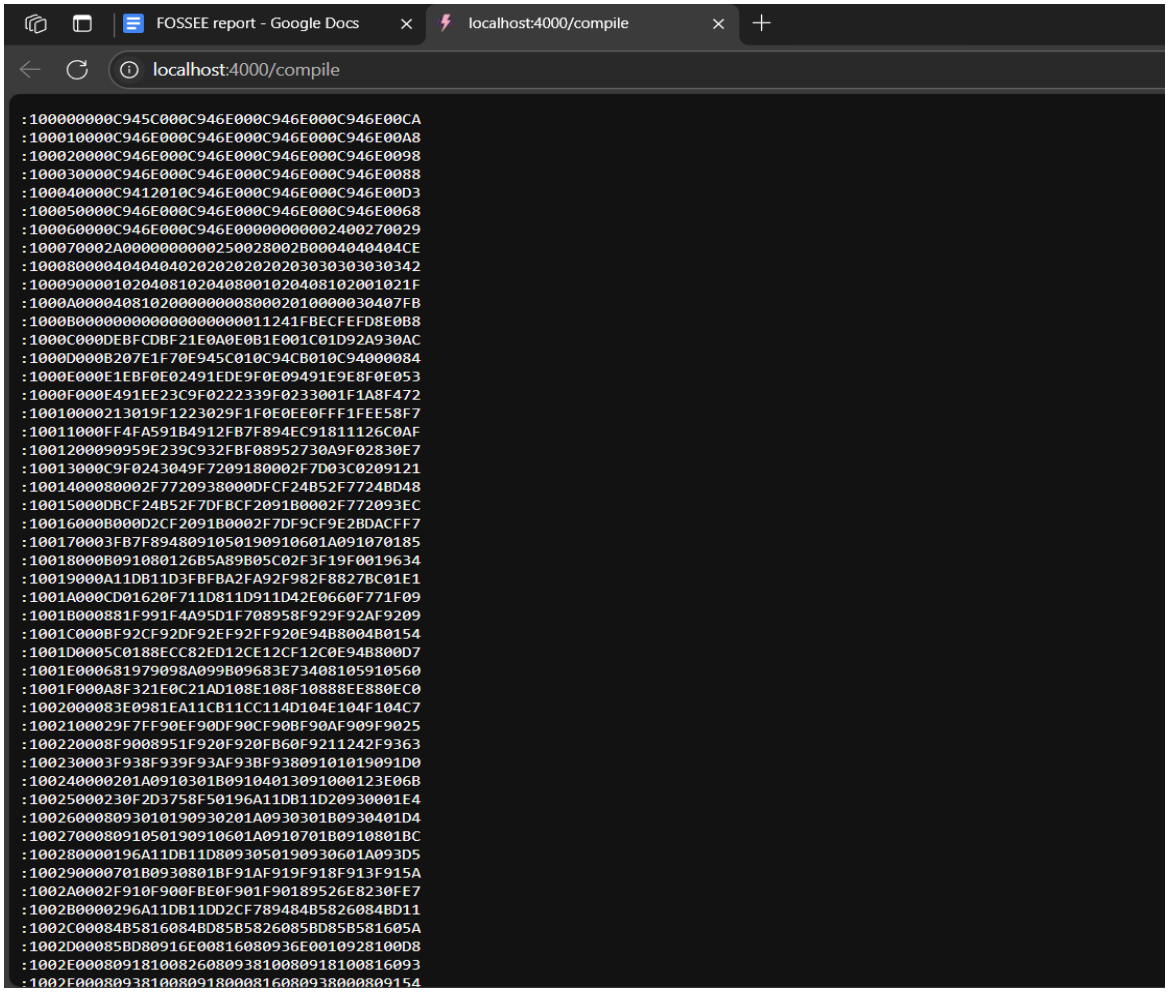


Fig.2.11.13

6. Functional Flow Overview

- User downloads electroblocks_code.ino from website
- App (running in background) detects new file in Downloads/
- Moves .ino to downloads/electroblocks_code/electroblocks_code.ino
- Tray menu updates to show "Compile"
- User clicks "Local_Compile" (via tray or browser endpoint)
- Arduino CLI compiles code → .hex file generated
- User (or frontend) accesses /download-hex to get the firmware.
- The "Compile" menu item in the tray is always shown, regardless of .ino file presence.
- On click, it opens /compile, which compiles the sketch using Arduino CLI and returns the .hex content.

2.12 UPDATE THE COLORS OF THE BLOCKS BASED ON SELECTED LANGUAGE

To improve the overall usability, visual clarity, and language contextualization of the platform, we implemented a robust dynamic color theme switching mechanism that responds to the selected programming language (Arduino/C++ or Python).

Implementation Details

1. Enum Definitions for Language-Specific Color Palettes

Two enums were defined to hold the color codes for each supported language:

```
export enum COLOR_THEME_C {
  SENSOR = '#505bda',
  ARDUINO_START_BLOCK = '#b063c5',
  COMPONENTS = '#512c62',
  ARDUINO = '#7b5184',
  DATA = '#57355d',
  VALUES = '#505bda',
  CONTROL = '#b063c5',
}

export enum COLOR_THEME_PYTHON {
  SENSOR = '#610C9F',
  ARDUINO_START_BLOCK = '#940B92',
  COMPONENTS = '#DA0C81',
  ARDUINO = '#E95793',
  DATA = '#7E2185',
  VALUES = '#A1126F',
  CONTROL = '#C41E75',
}
```

Each enum serves as a centralized, immutable theme definition. Block types such as SENSOR, DATA, CONTROL, etc., are mapped to specific hex color codes for consistency and clarity.

2. Reactive Store for Current Theme State

To enable real-time theme updates across the application, a writable Svelte store was created:

```
export const colorThemeStore = writable<typeof COLOR_THEME_C | typeof
COLOR_THEME_PYTHON>(COLOR_THEME_C);
```

This store holds the active color theme and is subscribed to by all components that require access to current color values. When the theme changes, all dependent components automatically re-render with the new color values—eliminating manual updates.

3. Initialization Based on Default Language Settings

At runtime, the theme is initialized using the user's saved language preference (from Firebase or local storage):

```
let COLOR_THEME: typeof COLOR_THEME_C | typeof COLOR_THEME_PYTHON =
  defaultSetting.language === 'Python' ? COLOR_THEME_PYTHON : COLOR_THEME_C;

colorThemeStore.set(COLOR_THEME);
```

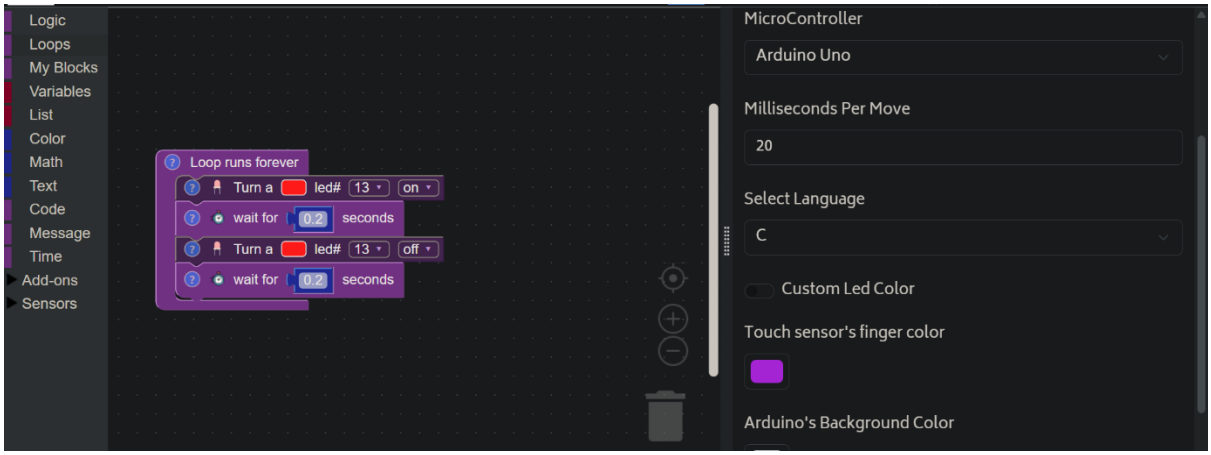
This ensures that when the app is loaded, the UI reflects the correct theme based on whether the user was last using Python or Arduino mode.

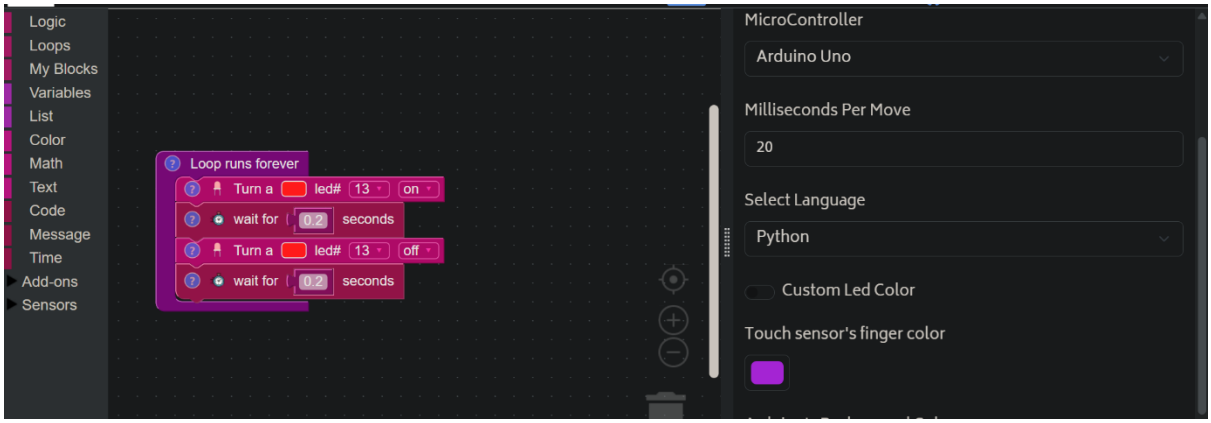
4. Dynamic Theme Switching via settingsStore Subscription

- A reactive subscription to settingsStore was added to monitor runtime changes to the programming language. When a change is detected, the updateTheme function updates the active theme
- This ensures that if a user switches the language setting mid-session, the color theme adapts immediately without requiring a page reload or manual refresh.

5. Usage Across the Application

Components that render visual blocks or UIs based on the current theme simply subscribe to colorThemeStore. This ensures full reactivity—any changes to the store instantly reflect in the component styling.





Chapter 3: Bug Fixes

During the Feature Additions, some features showed unintended results and/or duplication and were later fixed by us.

3.1 FIXED THE GENERATION OF DUPLICATE BOARD SETUP PYTHON CODE

While adding the Python Generators for various blocks [Chapter 2], we had added explicit imports of modules like `pyFirmata` and setup code for the `pyFirmata` module in every block we needed. This was inefficient and was later addressed by adding these imports and setup configuration to the main Python Generator itself, but later resulted in the Blocks from Addons category to generate duplicate code.

- Addressing duplicate board configurations:

```
this.setupCode_["firmata_board"] =
  'board = Arduino("YOUR_PORT_HERE")\n'+
  'it = util.Iterator(board)\n'+
  'it.start()'+
  'time.sleep(1)\n';
```

The following snippet was added to the main generator, which resulted in the dynamic configuration of the board, without needing to explicitly define a method in every generator that needed it. [3]

- Addressing duplicate module imports:

```
this.imports_["pyfirmata"] = `from pyfirmata import Arduino, util`;
```

The following snippet was also added to the main generator among others, which fixed the duplicate import issues. [3]

Chapter 4: Conclusion

This project was truly something special—like crafting a piece of art that we genuinely enjoyed building together. Adding new features, experimenting with ideas, and watching it evolve was not only fun but also a rich learning experience for all of us. We're deeply grateful to our mentors for their constant support and thoughtful guidance throughout the journey.

We genuinely believe this project holds a lot of promise. It has the potential to become a valuable resource for students and researchers exploring Electroblocks and Arduino. Every bit of effort and creativity we put into it came from our shared goal of making learning and prototyping more accessible and enjoyable. We're excited about the impact this work could have, and how it might inspire or support others in their own projects.

Chapter 5: Future Work

1. This project can be improved further by adding new features and fixing already present bugs.
2. Create a virtual environment that does not require the internet for people with bad internet.
3. It should work on a Windows machine as the server. You will need to get both the main Electro Blocks repo and the Electro Blocks server working.
4. Add support for other languages like Rust, Java for the ease of wider audience.
5. Enhanced data visualization tools.
6. User Interface Enhancements.
7. Add proper and detailed documentation of the entire codebase and how modules like interlinked to each other to improve understandability and easier debugging and feature additions.

References

1. Implemented variables, color, math, text and loop block generators for python (GitHub)[Text Block]: [https://github.com/ElectroBlocks/ElectroBlocks/pull/274/files#diff-d4ca0a399cf6a09f483724314e97316e4ccc3cba58260c4a649c6b65ed00fe92]

    a. [Color Block (rgbToHex)]: [https://github.com/ElectroBlocks/ElectroBlocks/blob/eec4afa5abdfe473f1657d5d4ac897f318772887/src/blocks/color/generators.ts#L3]

    b. [Color Block (random color)]: [https://github.com/ElectroBlocks/ElectroBlocks/blob/eec4afa5abdfe473f1657d5d4ac897f318772887/src/blocks/color/generators.ts#L54]

2. ElectroBlocks's Python Library (GitHub): [https://github.com/ElectroBlocks/python]

3. References for Python's Operator Precedence Table (World Wide Web)[Main Python Generator]:

    a. Reference for Python's operator precedence table (World Wide Web): [https://neil.fraser.name/blockly/custom-blocks/operator-precedence]

    b. Reference for Python's operator precedence table (World Wide Web): [https://docs.python.org/3/reference/expressions.html#operator-precedence]

4. https://github.com/ElectroBlocks/ElectroBlocks/tree/noah-toolbox-fix-import

5.https://github.com/ElectroBlocks/ElectroBlocks/pull/277/files

6. https://github.com/FOSSEE/FLOSS-Arduino-Book/tree/master

7.https://github.com/FOSSEE/FLOSS-Arduino-Book/tree/master/electroblocks_examples

8.https://github.com/ElectroBlocks/ElectroBlocksCompanion.git

9. Bug Fixes (GitHub): [https://github.com/ElectroBlocks/ElectroBlocks/blob/e119f2419c7a2749a6fa1c8e150d7d6dc24a4e65/src/core/blockly/generators/python.ts#L81]