

EMBEDAGENT: Benchmarking Large Language Models in Embedded System Development

Anonymous Author(s)

Abstract

Large Language Models (LLMs) have shown promise in various tasks, yet few benchmarks assess their capabilities in embedded system development. In this paper, we introduce EMBEDAGENT, a paradigm designed to simulate real-world roles in embedded system development, such as *Embedded System Programmer*, *Architect*, and *Integrator*. This paradigm enables LLMs to be tested in tasks that bridge the gap between digital and physical systems, allowing for a more comprehensive assessment of their capabilities. To evaluate LLMs on these tasks, we propose EMBEDBENCH, the first comprehensive benchmark for embedded system programming, circuit design, and cross-platform migration. EMBEDBENCH consists of 126 cases, covering 9 electronic components across 3 hardware platforms. Through extensive experiments on 10 mainstream LLMs, we uncover several key findings. Surprisingly, despite the simplicity of the cases, DeepSeek-R1 achieves only a 55.6% pass@1 rate when provided with schematic information, and 50.0% when tasked with generating the schematics itself. In the cross-platform migration tasks, LLMs show relatively strong performance with MicroPython on the Raspberry Pi Pico (with the top model achieving 73.8% pass@1), but perform poorly on ESP-IDF, where the best model reaches only 29.4% pass@1. Interestingly, we observe that general-purpose chat LLMs like DeepSeek-V3 often fail to utilize relevant pre-trained knowledge in this domain, while reasoning LLMs tend to overthink and overlook efficient knowledge during pretraining. Based on these insights, we propose two strategies—retrieval augmented generation and compiler feedback—to enhance LLM performance. These strategies result in significant improvements, with Deepseek-R1 reaching a 65.1% pass@1 with correct schematics, and 53.1% without. Additionally, the accuracy of the Arduino to ESP32 migration task improves from 21.4% to 27.8%.

CCS Concepts

- Computer systems organization → Embedded systems.

Keywords

Embedded System, Large Language Models

ACM Reference Format:

Anonymous Author(s). 2025. EMBEDAGENT: Benchmarking Large Language Models in Embedded System Development. In *Proceedings of Symposium on Operating Systems Principles (SOSP #460)*. ACM, New York, NY, USA, 18 pages. <https://doi.org/XXXXXXXX.XXXXXXXX>

1 Introduction

Large Language Models (LLMs) have demonstrated remarkable expertise across various software engineering tasks, such as code generation [8, 17], defect detection [32], program repair [12, 29, 34] and code translation [30]. To assess the these abilities of LLMs, benchmarks like HumanEval [4] and SWE-bench [13] have been

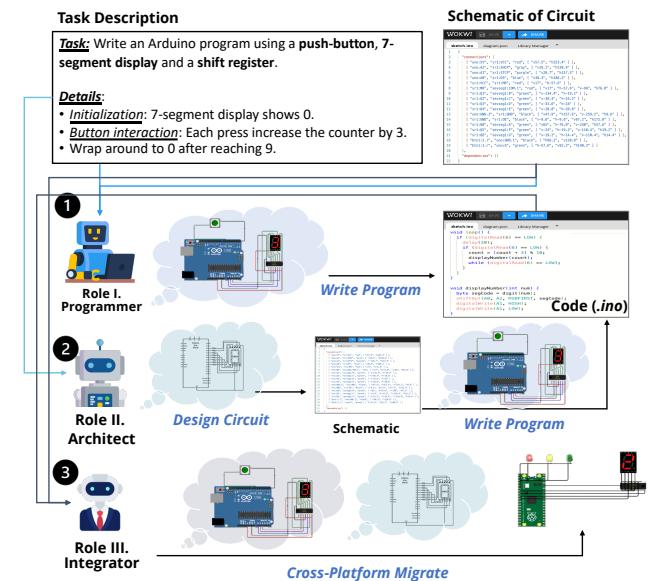


Figure 1: Three Settings of EMBEDAGENT: ① **Embedded System Programmer**: Given the task description and schematic of circuit, LLMs are expected to write Arduino code. ② **Embedded System Architect**: Given the task description, LLMs are expected to design the circuit and write the code. ③ **Embedded System Integrator**: Given the schematic of circuit and code of one hardware, LLMs are expected to migrate the design of circuit and code to another hardware platform.

introduced, offering valuable insights into the strengths and limitations of LLMs.

Research gap – However, as a key carrier connecting the digital LLMs to the physical world [14, 28], there are few benchmarks to evaluate the capabilities of LLMs in the development of embedded systems. Prior studies [9, 33] have primarily focused on code generation for embedding application, leaving their abilities in circuit design and cross-platform migration largely unexplored.

EMBEDBENCH – To address the research gap, we construct EMBEDBENCH, a new benchmark designed to evaluate the capabilities of LLMs on fundamental embedded system tasks. To facilitate the evaluation, we introduce EMBEDAGENT, which operates under three distinct settings that simulate read-world roles, as shown in Figure 1:

- ❶ **Embedded System Programmer**: Given a task description and a circuit schematic, LLMs are expected to write embedded system code (e.g., Arduino code).
- ❷ **Embedded System Architect**: Given a task description, LLMs are expected to design the circuit and write the corresponding embedded code.
- ❸ **Embedded System Integrator**: Given the code and the circuit schematic for one hardware platform, LLMs are expected to

117 migrate both the circuit design and code to another platform
 118 (Arduino, ESP32, and Raspberry Pi Pico).

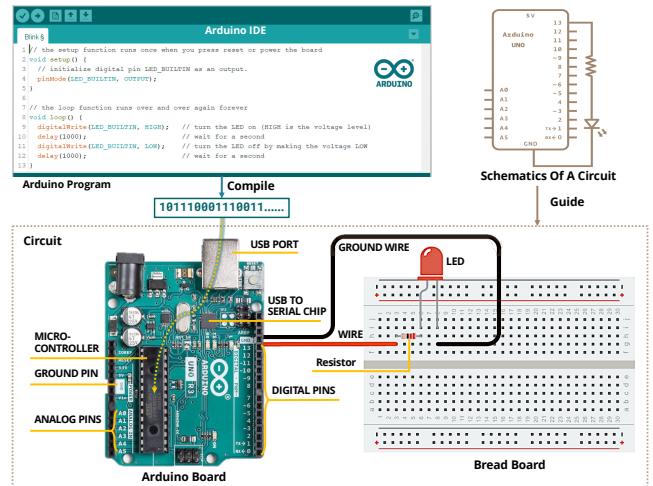
119 For each setting, there are manually constructed 126 cases in EMBEDBENCH, including task description, reference solution and automated correctness validation.

120 However, a comprehensive evaluation of LLMs’ capabilities in
 121 embedded system development presents several challenges. First,
 122 representing circuit schematics in a format that LLMs can under-
 123 stand and generate is challenging, since LLMs typically interact
 124 solely through natural language inputs and outputs. Second, verifying
 125 the correctness of the generated code is difficult. Previous work
 126 often evaluates correctness through serial output [9, 33], but this
 127 approach can be misleading, as LLMs may produce plausible out-
 128 puts without ensuring correct hardware behavior. Finally, verifying
 129 LLM-generated solutions at scale remains costly and inefficient,
 130 as manual assembly and physical testing are time-consuming and
 131 resource-intensive.

132 **Design for End-to-End Automated Evaluation** – To over-
 133 come the above challenges, we design a comprehensive evaluation
 134 from **representation**, **evaluation**, and **pipeline** aspects. First, we
 135 design an *interpretable circuit representation* that allows LLMs
 136 to understand and generate circuit structures effectively, bridging
 137 the gap between natural language inputs and hardware representa-
 138 tion. Second, to overcome the limitations of traditional evaluation
 139 based on serial output, we introduce a **hardware-driven evalua-
 140 tion approach**, where the behavior of virtual hardware in the
 141 Wokwi¹ environment is monitored in real-time to verify the cor-
 142 rectness of the generated code. Third, to tackle the inefficiency of
 143 manual validation, we develop an **end-to-end automated evalua-
 144 tion pipeline**, which maps LLM-provided schematics to simulated
 145 connections within the virtual environment, ensuring an efficient
 146 and scalable evaluation process.

147 **Study** – Through extensive experiments on 10 mainstream LLMs,
 148 we uncover several key findings. First, although our benchmark
 149 focuses on relatively basic tasks in embedded system development,
 150 these problems remain challenging for state-of-the-art LLMs. For
 151 example, even when provided with correct circuit schematics, the
 152 best-performing model, Deepseek-R1, achieves only 55.6% pass@1.
 153 When the correct schematics are not provided, its performance
 154 drops to 50% pass@1. In the cross-platform migration task (from
 155 Arduino to ESP32), the best result is achieved by Claude 3.7 Sonnet
 156 (Thinking), with a pass@1 score of only 29.4%. Second, we observed
 157 that different types of LLMs exhibit different weaknesses in this
 158 domain. For example, chat LLMs often struggle to flexibly apply
 159 their pretrained knowledge. In contrast, reasoning LLMs tend to
 160 approach problems from a very low-level perspective and often
 161 fail to effectively leverage the higher-level knowledge encoded
 162 during pretraining. Surprisingly, in some cases, a reasoning LLM
 163 that designs the circuit by itself can outperform the same model
 164 when given a predefined schematic.

165 Building on these insights, we explore two strategies to improve
 166 LLM performance in embedded system development tasks: 1) *Re-
 167 trieval augmented generation*, which enhances LLMs by incorporat-
 168 ing previous experience; 2) *Compiler feedback*, which helps correct



169 **Figure 2: Arduino Workflow.** The workflow requires a com-
 170 bination of hardware and software. On the hardware side,
 171 an Arduino board, electronic components, and a circuit
 172 schematic are needed. On the software side, code must be
 173 written based on the schematic and then uploaded to the
 174 board via USB port and a USB-to-serial chip, finally processed
 175 in the microcontroller.

176 syntax errors in the generated code. With these methods, Deepseek-
 177 R1 improves to 65.1% pass@1 when provided with correct circuit
 178 schematics, and 53.1% without them. For the Arduino to ESP32
 179 migration task, accuracy improves from 21.4% pass@1 to 27.8%.

180 The contribution of this paper includes:

181 ✓ **Benchmark EMBEDBENCH:** We introduce EMBEDBENCH, a
 182 benchmark for embedded system development. To the best of our
 183 knowledge, this is the first comprehensive benchmark designed
 184 to assess LLMs’ capability in embedded system programming, cir-
 185 cuit design, and cross-platform migration. It includes of 126 cases,
 186 covering 9 electronic components across 3 hardware platforms.

187 ✓ **Executable Context and automated validation mech-
 188 anism:** We propose an efficient and automated evaluation pipeline
 189 for assessing embedded systems developed by LLMs. This frame-
 190 work streamlines the testing process, ensuring consistency and
 191 reliability.

192 ✓ **Comprehensive Study and Vision:** We conduct a com-
 193 prehensive evaluation of 10 mainstream LLMs on our benchmark and
 194 uncover several insightful findings regarding their strengths and
 195 limitations. Based on these findings, we propose two effective meth-
 196 ods to improve LLM performance in embedded system tasks.

2 Background

197 In this section, we provide background information on the work-
 198 flow of embedded systems and introduce Wokwi, a virtual circuit
 199 simulation platform.

2.1 Embedded System Workflow

200 Embedded systems are specialized computing systems that are
 201 based on hardware and driven by software. They are widely used

¹<https://wokwi.com/>

in areas such as smart homes, robotics, and the Internet of Things (IoT). These systems serve as an important bridge between the virtual world and the physical world. Building a complete embedded system typically requires a combination of both hardware and software. Figure 2 illustrates the process of building a simple embedded system using an Arduino development board and an LED.

On the hardware side, it is necessary to prepare the development board and the required components. In this case, the components are an Arduino board and an LED. Then, based on the components and the serial port information of the board, the circuit schematic is constructed.

On the software side, code is written according to the schematic. In the figure, the Arduino IDE is used to write code that makes the LED blink every second. This code is compiled into a binary file and uploaded to the development board, enabling interaction between the virtual world and the physical world.

2.2 Virtual Circuit Simulation Platform Wokwi

Wokwi[27] is an online simulation platform for electronic circuits and embedded systems. It is primarily used to simulate the behavior of various microcontrollers such as Arduino, ESP32, and Raspberry Pi Pico, along with peripheral electronic components like sensors, LEDs, and displays. It provides a convenient virtual environment that allows developers to quickly build, test, and debug projects directly in the browser, enabling code and circuit design verification without the need for physical hardware.

Figure 3 shows an example of developing an embedded system using the Wokwi platform. Similar to real-world development, creating an embedded system in Wokwi also involves both software and hardware. On the hardware side, appropriate development boards and necessary components are selected, followed by schematics. The schematics are represented in the Diagram (.json) shown in Figure 3, where a structured file records the connections.

On the software side, the corresponding Arduino code is written, as shown in the Code (.ino) section of the figure. Once the code is complete, clicking the “Activate” button on the website compiles the program. Users can interact with the embedded system through their computer to determine whether the expected functionality has been achieved.

3 Benchmark Construction

In this section, we provide a detailed introduction to the construction process of our benchmark. It can be divided into three main steps. First, determining the components required for each problem. Second, formulating appropriate problems based on the hardware usage. Finally, constructing test cases according to the problems. We also present statistics of our benchmark and analyze its quality.

3.1 Electronic Component Selection

To ensure balanced utilization of components across problems, we selected seven primary hardware types as seed components: LED, RGB LED, Push Button, 7-Segment Display, LED Bar Graph, Slide Potentiometer, and Servo. Detailed descriptions of each component are provided in Appendix A. As illustrated in Figure 4 Step I, each problem is assigned a distinct set of electronic components. For each problem, we randomly select 1 to 3 components from the

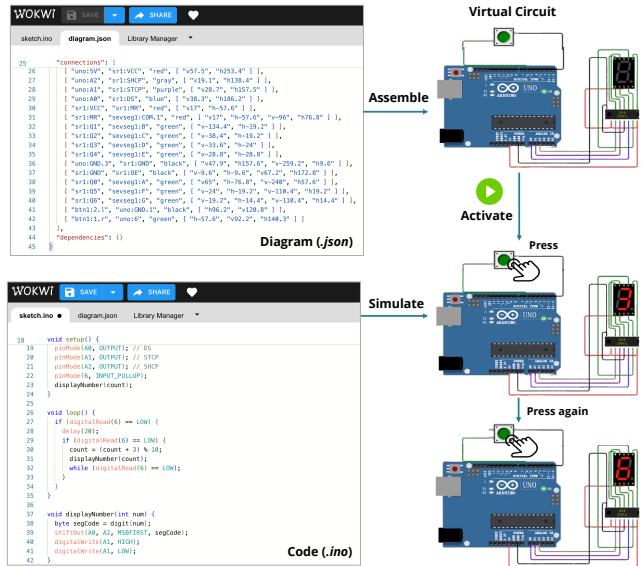


Figure 3: Illustration of Wokwi [27] - A Virtual Circuit Simulation. The upper part (i.e., Diagram) shows how the virtual circuit is denoted in Wokwi, the lower part (i.e., Code) shows how the Arduino code simulates the virtual circuit. Once the virtual circuit is activated, it will be virtually powered. According to the Arduino code, the number shows in 7-segment display increase once the button has been pressed.

seven primary types. Each selected component is then randomly assigned a usage count between 1 and 3. We sample each unique hardware combination twice, resulting in a total of 126 cases, each with a unique distribution of components. An example problem is shown in the Task Description of Figure 4 step II, where the sampled configuration includes two types of components (7-Segment Display and Push Button), each hardware use one instance.

3.2 Problem Construction

After determining the required electronic components for each problem, we formulate the problem descriptions through human annotation based on these components. Each problem must satisfy two key conditions. First, all selected electronic components must be actively utilized within the problem. Second, to ensure the stability of the evaluation results, any given system state must be maintained for at least 2 seconds.

Each problem description is composed of two parts. The first part provides a high-level overview, specifying the electronic components involved and the task's general objective. The second part outlines the specific operations and the expected states of each electronic component following those operations.

A simplified version is shown in the Task Description of Figure 4, Step II. The problem description begins by stating that the electronic components include a button and a 7-segment display. The task is to implement a counter. It then specifies the detailed behavior: The initial state of the 7-segment display should show the number 0, and each button press should increment the counter by 3.

Based on the formulated problems, we develop corresponding test cases, with each problem containing 3 to 5 test cases. These test cases are designed to verify whether the code generated by the LLMs satisfies all the requirements specified in the problem description. Additionally, we provide correct reference implementations to ensure that each problem is solvable.

For example, in Figure 4, Step II, to verify the correctness of the code, we need to check whether the output for the n th button click is equal to $(n*3)mod10$. Accordingly, the first three test cases check the behavior for 1 to 3 button clicks. A fourth test case simulates 20 button presses to verify whether the code handles extended sequences and edge conditions correctly, ensuring comprehensive evaluation.

3.3 Executable Scripts Preparation

Unlike previous work that relies on physical hardware and determines correctness through serial output, we use Wokwi simulation platform to observe the real-time state of each electronic component at any given moment. This capability is essential for evaluating embedded systems, as the primary goal of embedded systems is to ensure that the electronic components perform the expected operations.

We use an automated submission bot to submit the code generated by LLMs and monitor the electronic components to ensure the behavior aligns with the specified requirements. Each problem is associated with a unique script containing all the test cases defined in Step II, which are used to verify correctness. For instance, in Step III of Figure 4, the validation process involves checking whether the 7-segment display shows the expected number after the push button has been pressed a specified number of times.

3.4 Quality Analysis

Each problem, along with its corresponding solutions and test cases, is created by two volunteers holding bachelor's degrees. After the initial development, another two volunteers with bachelor's degrees are responsible for reviewing the content to ensure its quality and accuracy. Annotators should follow these main guidelines: (1) The problem description must fully utilize all the specified electronic components. (2) The provided test cases must be sufficient to verify the correctness of the problem, ensuring that the reference code passes all of them. (3) Any states that require verification must be maintained for at least two seconds to avoid the impact of potential network delays. This multi-stage review process helps ensure the correctness, reliability, and overall quality of the questions. Further details regarding annotation rules can be found in Appendix C. [And the detail statistical of our work is in appendix](#)

4 Experiments

4.1 Experiment Setup

4.1.1 Evaluation Tasks. To systematically evaluate embedded systems, we propose three task settings designed to assess LLMs' embedded system development capabilities under different conditions: Settings ①, ② and ③ which is shown in Figure 1. For the i -th problem, we define the problem description as q_i , the schematic configuration as w_i , the hardware specifications as h_i , the code solution as c_i ,

the test cases as u_i , the programming language as L , the development board as B , and the LLMs used for code generation as M . The LLMs' predicted probability distribution is denoted by $P(\cdot)$, while $I(\cdot)$ represents a validation function to verify the correctness of the LLM's output. The evaluation result r_i is an n -dimensional binary matrix, where n is the number of test cases, and each element is either 1 (the test case is passed) or 0 (the test case is failed).

| Setting ① Embedded System Programmer Prompt | |
|--|--|
| ### Problem Description | |
| You are tasked with programming an Arduino (mega) to create an interactive game using three LEDs ... | |
| ### Components Usage | |
| {'type': 'arduino-mega', 'id': 'mega'} | |
| ... | |
| ### Components Information | |
| type: led | |
| pins: | |
| 'A': The positive pin of the LED. | |
| 'C': The negative pin of the LED. ... | |
| ### Connections | |
| ['mega:GND.1', 'led_red:C'] | |
| ['led_red:A', 'mega:2'] | |
| ['mega:GND.1', 'led_yellow:C'] | |
| ... | |

Table 1: The prompt of Setting ① task

Setting ① As illustrated in Figure 1, this task requires LLMs to generate embedded system code based on a given problem description and a schematic configuration. Successfully completing the task demands a comprehensive understanding of the problem requirements, the functionality of each hardware component, and the roles of individual pins. This task evaluates the LLM's capability as a programmer. The process can be formalized as:

$$r_i = I(P(c_i | q_i, w_i, h_i, L, B; M); u_i) \quad (1)$$

The prompt for the “Setting ①” tasks is shown in Tabel 1. First, the LLM is provided with a problem description. Next, a list of components is given, where each component is represented as a dictionary containing two keys: the component's type and its unique identifier. Following this, detailed specifications for each component type are presented, including the available pins and their respective functions. Lastly, the schematics are described. Each connection is specified as a pair of strings in the format “id:pin”, denoting the connection between specific pins of two components. For example, [“uno:GND.1”, “led1:COM”] represents a connection between pin GND.1 of the uno component and the COM pin of the led1 component. The LLM is expected to generate Arduino code that utilizes the given components and schematics configuration to solve the specified problem.

Setting ② As shown in Figure 1, in this task, in addition to generating code for the embedded system, LLMs are also required to design circuits. This requires a deeper understanding of hardware components, familiarity with the functionalities of various serial ports, and a certain level of logical reasoning and spatial capabilities.

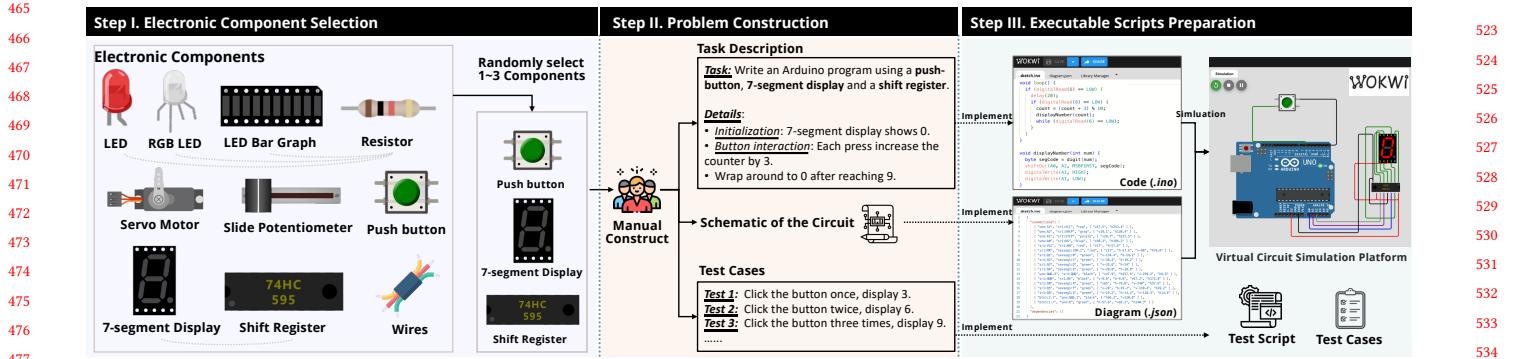


Figure 4: Data Preparation Pipeline

| Setting ❷ Embedded System Architect Prompt | |
|---|--|
| <p>### Problem Description</p> <p>You are tasked with programming an Arduino (mega) ...</p> <p>### Components Usage</p> <pre>{'type': 'arduino-mega', 'id': 'mega'} {'type': 'rgb-led', 'id': 'rgb1'} ...</pre> <p>### Components Information</p> <pre>type: rgb-led pins: 'R': Controls the brightness of the red LED. ... #Example Connections</pre> <p>Here is an example of a connection format:</p> <p>[CONNECTIONS]</p> <pre>["uno:A2", "sr1:SHCP"] ["sr1:MR", "sevseg1:COM.1"] ... [CONNECTIONS]</pre> | |

Table 2: The prompt of Setting ❷ task

This task evaluates the LLM’s capability as an expert in embedded system development. The process can be formulated as:

$$r_i = I(P(w_i, c_i | q_i, h_i, L, B; M); u_i) \quad (2)$$

As shown in Table 2, the prompt is constructed similarly to the “Setting ❶” task. LLMs are given a task description without schematics of circuits. To format the output, we include example connections to guide the LLM in constructing an appropriate schematic in the required format. The LLM is expected to return both the schematic for the given components and the code to control the components and complete the task.

Setting ❸ As shown in Figure 1, the “Setting ❸” task requires LLMs to migrate both the schematics and code from a given source platform to a target platform. The LLM must consider the similarities and differences between platforms, including serial ports, programming languages, and the syntax of various function calls. This process can be formalized as:

$$r_i = I(P(w'_i, c'_i | h'_i, L', B', w_i, c_i; M); u_i) \quad (3)$$

| Setting ❸ Embedded System Integrator Prompt | |
|--|--|
| <p>### Arduino Code</p> <pre>void setup() { ... }</pre> <p>### Components Usage</p> <p>...</p> <p>### Components Information</p> <pre>type: pi-pico pins: 'GP0' to 'GP22': Digital GPIO pins for input/output operations. ...</pre> <p>### Arduino Connection</p> <p>...</p> | |

Table 3: The prompt of Setting ❸ task

Here, the variables with superscripts (w'_i, c'_i, h'_i, L', B') denote the corresponding elements in the target platform. Detailed information of each platform used in this task is provided in Appendix B. Unlike the “Setting ❶” and “Setting ❷” tasks. As shown in Table 3, the “Setting ❸” task does not include an explicit problem description. Instead, because it involves platform migration, the input includes the Arduino code and schematics connections. The component usage and specifications remain consistent with the previous two tasks, with the addition of the target platform’s details. The LLM must first translate the schematics from Arduino to the target platform and then generate equivalent code that replicates the same functionality.

4.1.2 Evaluation Method. We use pass@1 as the evaluation metric following prior work [4, 16]. The temperature is set to 0 and employ greedy decoding for generation. A problem is considered successfully solved if the LLM passes all test cases.

4.1.3 LLMs for evaluation. We select 10 state-of-the-art LLMs spanning 6 different LLM families, and categorize them into three types for evaluation: **Reasoning LLMs** (O3-mini[23], Claude 3.7 Sonnet (Thinking)[5], DeepSeek-R1[10], QWQ[26]), **SFT-Distilled**

| Models | Size | Task | | | |
|------------------------------------|----------|----------------|----------------|------------------|--------------------|
| | | Setting ① | Setting ② | Setting ③ | |
| | | Arduino C++ | Arduino C++ | ESP32 ESP-IDF | Pi Pico MicroPy |
| <i>Reasoning LLM</i> | | | | | |
| Claude 3.7 | \ | 48.4 | 48.4 | 29.4 | 73.8 |
| O3-mini | \ | 47.6 | 38.9 | 16.7 | 61.9 |
| DS-R1 | 671B/31B | 55.6 | 50.0 | 21.4 | 60.3 |
| QWQ | 32B | 39.7 | 50.0 | 4.0 | 52.4 |
| <i>SFT-Distilled Reasoning LLM</i> | | | | | |
| DS-R1-Dist. | 70B | 35.7 | 23.8 | 0.0 | 11.1 |
| DS-R1-Dist. | 32B | 27.0 | 11.1 | 3.2 | 27.8 |
| <i>Chat LLM</i> | | | | | |
| DS-V3 | 671B/31B | 37.3 | 25.4 | 2.4 | 46.8 |
| Llama-3.3-Ins. | 70B | 32.5 | 21.4 | 0.0 | 23.0 |
| Qwen2.5-Ins. | 32B | 25.4 | 23.8 | 3.2 | 33.3 |
| Starcoder2 | 15B | 11.1 | 2.4 | 0.0 | 4.0 |

Table 4: pass@1(%) results of each LLMs. Abbreviations: sch (schematics), DS(DeepSeek), Claude 3.7(Claude 3.7 Sonnet (thinking)), DS-R1-Dist. 70B (DeepSeek-R1-Distill-Llama), DS-R1-Dist. 32B (DeepSeek-R1-Distill-Qwen), Ins. (Instruct). Ds-R1 and DS-V3 are MOE LLMs with 32B parameters activated during inference.

Reasoning LLMs (DeepSeek-R1-Distill-Llama[10], DeepSeek-R1-Distill-Qwen[10]), and **Chat LLMs** (DeepSeek-V3[6], Llama-3.3-Instruct[1], Starcoder2[19], Qwen2.5-Instruct[31]).

4.2 Overall Result

Table 4 presents the pass@1 evaluation results for different types of LLMs, arranged in descending order based on their parameter size. The key observations are as follows:

1) EMBEDBENCH is challenging for studied LLMs. In both the “Setting ①” and “Setting ②” tasks, the highest-performing LLM, DeepSeek-R1, achieves only 55.6 and 50.0 pass@1, respectively. For the “cross-platform” task, most LLMs struggle to transfer Arduino code to ESP32’s ESP-IDF effectively. Even the best-performing LLM, Claude 3.7 Sonnet (thinking), reaches only 29.4 pass@1. Transferring code to the Raspberry Pi Pico using MicroPython is slightly more successful, with a pass@1 of 73.8 at best.

2) Reasoning LLMs outperform Chat LLMs. However, SFT-distilled reasoning LLMs show mixed results compared to their base versions. For example, QwQ surpasses Qwen2.5-Instruct, across all tasks. However, DeepSeek-R1-Distill-Qwen does not show a significant improvement over its base LLM. Additionally, DeepSeek-R1 consistently outperforms DeepSeek-V3 in all tasks.

5 Analysis

We first analyze why LLMs struggle to construct embedded systems by investigating concrete cases. Next, we unpack the chain of

thought generated by different reasoning LLMs when tackling embedded system tasks. Finally, we try some methods to explore LLMs’ upper bound of capabilities in constructing embedded systems.

5.1 Case Study

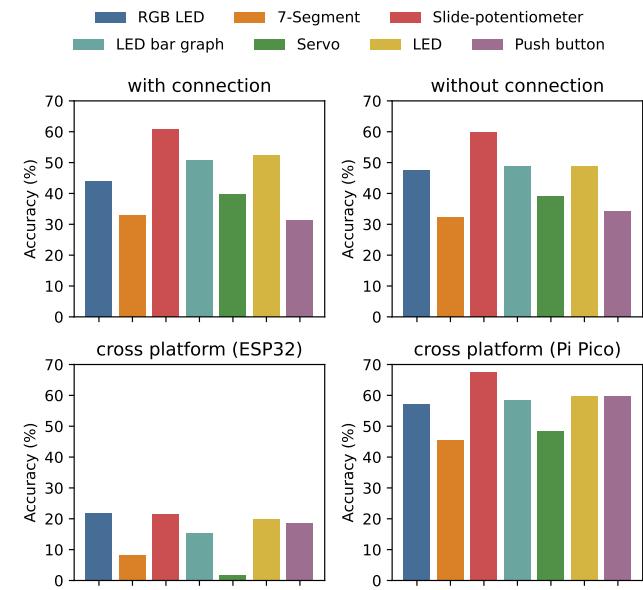


Figure 5: The average accuracy of reasoning LLMs (QwQ, DeepSeek-R1, O3-mini, Claude 3.7 Sonnet (Thinking)) on problems involving specific electronic components.

To investigate why LLMs struggle with constructing embedded systems, we categorize problems based on the specific electronic components they involve. For instance, if a problem requires the use of both a “Servo” and an “LED”, we classify it under both “Servo” and “LED” categories. We then measure the average accuracy of various reasoning LLMs (QwQ, DeepSeek-R1, O3-mini, Claude 3.7 Sonnet (Thinking)) across seven major electronic components in four different tasks. The results are presented in Figure 5. From the figure, we observe that LLMs struggle particularly with problems involving “7-Segment displays” and “Push Buttons” in both the “Setting ①” and “Setting ②” tasks. In the “Setting ② (Pi Pico)” task, LLMs generally perform well. However, in the “Setting ② (ESP32)” task, accuracy is consistently low across all electronic components. In the following content, we provide a detailed analysis of these challenges, focusing on the “7-Segment”, “Push Button” and the ESP32 platform.

7-Segment Display LLMs struggle to determine the correct voltage levels for each port when displaying numbers (0 to 9) or characters such as “A” and “P”.

A significant portion of the errors in the “Setting ①” task related to 7-segment displays comes from this issue, accounting for 48 out of 116 errors (approximately 41.4%). Similarly, in the “Setting ②” task, 45 out of 103 errors (approximately 43.7%) fall into the same category.

Listing 1 Subject-101 (QWQ)

```

697 // Lookup table for 7-segment display
698 // (common anode)
699 byte seg_code[10] = {
700     0x90, // 0
701     0xF9, // 1
702     0xA2, // 2
703     0xB0, // 3
704     0x99, // 4
705     0x92, // 5
706     0x82, // 6
707     0xF8, // 7
708     0x80, // 8
709     0x90 // 9
710 };
711
712
713
714

```

An example is shown in Listing 1. The LLM uses hexadecimal values to represent the voltage levels of each segment in the 7-segment display. For instance, to represent the number 1, it uses the value `0xF9`, which corresponds to the binary pattern `0b11111001`. This means that two segments are set to low voltage (i.e., they are activated in a common-anode display), correctly displaying the digit 1. However, the LLM struggles to consistently generate correct values for each digit. In this example, both the digits 0 and 9 are represented using the same value `0x90`, which indicates an error, as they should have different segment configurations.

Push Button For push buttons, LLMs often struggle with handling button debounce.

In the “Setting ❶” task, nearly half of the errors related to push buttons, 43 out of 108 (approximately 39.8%), are caused by improper debounce handling. Similarly, in the “Setting ❷” task, 51 out of 106 errors (approximately 48.1%) result from the same issue.

Listing 2 Subject-76 (Claude 3.7 sonnet (thinking))

```

732 // debounceDelay = 150ms
733 if (digitalRead(btn1Pin) == LOW && (currentTime -
734     ↪ lastBtn1Press) > debounceDelay) {
735     lastBtn1Press = currentTime;
736     incrementNumber(1);
737 }
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754

```

As shown in Listing 2, the LLM implements a basic debounce mechanism by checking whether the button is pressed (LOW) and ensuring that at least 150 ms have passed since the last press. However, even though the button press duration is intended to be 150 ms, various factors can make it difficult to maintain this timing precisely. As a result, this approach may lead to false triggers, where a single press is misinterpreted as multiple presses. A more reliable method would involve tracking the previous button state and updating the debounce timer only when a state change occurs (for example, from HIGH to LOW). This helps reduce the likelihood of erroneous detections.

ESP32 The primary issue observed in ESP32 code generated by LLMs is the high frequency of syntax errors.

A substantial portion of the generated code, 211 out of 504 cases (approximately 41.9%), contains such errors. Common problems include missing or incorrect usage of relevant header files, as well as the use of functions that are incompatible with the specified version of the ESP-IDF, even when the version is clearly provided in the prompt. When comparing Python and ESP-IDF code generation, it becomes evident that current LLMs lack sufficient training in domain-specific embedded programming languages such as ESP-IDF.

Finding 1: The main challenges for LLMs are: (1) Using a 7-Segment Display to display numbers or letters. (2) Debouncing button inputs. (3) Syntax of ESP-IDF.

5.2 Deep thinking for Embedded System

After conducting case studies on hardwares and platforms where LLMs are prone to make errors, we further investigate the underlying causes of these errors. Specifically, we analyze the reasoning chains produced by the reasoning LLMs to better understand how and why such errors occur in the development of embedded system.

5.2.1 The thinking Format of each LLMs. Given that current LLMs are trained on limited data related to embedded system development, we begin by analyzing the differences in the chains of thought produced by each reasoning LLM.

Claude 3.7 Sonnet (Thinking) devotes a significant portion of its reasoning process to debugging. This frequent and thorough debugging may help explain why Claude 3.7 Sonnet (Thinking) achieves the best performance on Setting ❸ tasks (e.g., ESP32), where LLMs are particularly prone to syntax errors. The reasoning process of Claude 3.7 Sonnet (Thinking) begins with a brief preparation phase that involves understanding the question, analyzing schematics diagrams, and outlining a plan for code generation. Following this initial stage, the model spends an average of 4,440 tokens (out of a total 5,703, or approximately 77.8% of its reasoning chain) identifying and resolving issues in the generated code. As illustrated in Table 5, the model first uses 955 tokens to generate an initial answer, then proceeds to debug the code multiple times

Claude 3.7 Sonnet Thinking Process

Generate the anwser use 955 tokens

Let's analyze what we have and what we need to do to migrate from Arduino Mega to a Pi Pico running MicroPython ...

Here's the original Arduino code: ...

Repair the code use 2025 tokens

Wait, I need to refine this a bit ...

Let me further check and correct this code ...

Let me fix the code ...

This logic is a bit confusing. Let me simplify ...

Wait, I need to clarify the LED logic once more ...

Table 5: An example of Claude 3.7 Sonnet Thinking Process

755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812

813 **DeepSeek-R1 and QWQ.** exhibit similar reasoning patterns,
 814 marked by frequent reflection throughout each stage of the problem-
 815 solving process. A typical indicator of this reflective behavior is the
 816 word “wait”[10, 22], which often signals moments of uncertainty or
 817 perceived inconsistency. This tendency toward continual self-evaluation
 818 may contribute to their strong performance on both code generation
 819 tasks which require advanced reasoning skills.

820 Compared to LLMs like Claude 3.7 Sonnet, DeepSeek-R1, and
 821 QWQ demonstrate a more introspective approach. From under-
 822 standing the initial problem to analyzing potential flaws in the
 823 generated code, these LLMs repeatedly assess and revise their rea-
 824 soning. The keyword “wait” frequently appears in their outputs
 825 as a marker of this process. On average, DeepSeek-R1 uses “wait”
 826 approximately 20.4 times per problem, while QWQ uses it about
 827 40.7 times. An example is shown in 6. From the example, we can see
 828 that Deepseek-R1 reflects on all stages of problem understanding,
 829 code structure construction, and code testing in a balanced manner.

Finding 2: (1) Claude 3.7 Sonnet (Thinking) tends to focus
 871 more on code repair, which gives it an advantage in ensuring
 872 syntactic correctness. (2) DeepSeek-R1 and QWQ engage in
 873 reflective thinking throughout the reasoning process, which
 874 helps ensure the accuracy of their logical reasoning. (3) SFT-
 875 distilled reasoning LLMs exhibit significant instability in their
 876 thought processes when faced with this entirely new task.

877 **5.2.2 Why struggle to display 7-Segment?** To further explore why
 878 LLMs encounter difficulties in embedded system development, we
 879 analyze their performance on a specific hardware task: controlling
 880 a 7-segment display. This task is widely used in embedded systems
 881 and is likely represented in the training data of many LLMs. How-
 882 ever, it also requires a multi-step reasoning process that includes
 883 hardware comprehension, pin mapping, and display logic. Based
 884 on the case study, we select three LLMs with varying capabilities,
 885 ranging from weak to strong, and evaluate how each performs
 886 when solving problems related to 7-segment displays:

887 ① **Starcoder2** is a code LLM trained on Stack v2 (67.5TB of GitHub
 888 code), which is publicly available. Using the provided tool [20], we
 889 verified that the digit table shown in Listing 1 is indeed present in
 890 the Stack v2 training set. Despite this, evaluation results reveal that
 891 StarCoder2 achieved 0% accuracy (0/45) on 7-segment display tasks,
 892 even in “Setting ①” task where the schematics are included. More-
 893 over, only 22.2% (10/45) of its generated solutions incorporated
 894 a digit table similar to the one in Listing 1. These findings indicate
 895 that **LLMs with small parameter sizes struggle to retrieve**
 896 and apply pretrained knowledge effectively without sufficient
 897 exposure to task-specific training.

898 ② **DeepSeek-V3** is a state-of-the-art chat LLM that performs well
 899 on a variety of general tasks. However, in the “Setting ①” task,
 900 which uses the 7-segment display, it achieves only 6.7% accuracy
 901 (3/45), despite including a digit table in 93.3% (42/45) of its outputs.
 902 While DeepSeek-V3 successfully activates knowledge related to
 903 7-segment display data present in its pretraining corpus, only 15.5%
 904 (7/45) of its generated solutions correctly adapted the digit table
 905 to a common anode configuration as required by the task. Even
 906 fewer solutions incorporated more complex changes in response
 907 to the problem’s requirements. These results indicate that while
 908 larger models trained on more extensive datasets **demonstrate an**
 909 **increased ability to recall relevant pretraining knowledge**, they
 910 still face challenges in flexibly applying that knowledge based on
 911 the task-specific contexts.

912 ③ **DeepSeek-R1** is a reasoning LLM that differs from chat LLMs
 913 by employing a longer chain-of-thought before output. This reason-
 914 ing ability enables it to outperform the previous models, achieving
 915 44.4% accuracy (20/45) on the “Setting ①” task. However, reasoning-
 916 based LLMs also face challenges in 7-segment tasks. Unlike its base
 917 model, DeepSeek-V3, which only reuses pretraining knowledge
 918 with minimal adaptation, DeepSeek-R1 tends to over-analyze. On
 919 average, it uses approximately 4429 tokens ((4429/10342) 42.8%
 920 of the total chain of thought) focusing on binary representations
 921 of digits—an unnecessary detour for this relatively simple task.
 922 This excessive reasoning often leads to errors, with some chains
 923 of thought even stating, “But this is tedious.” (See Appendix E). A
 924 more effective approach would be to retrieve a known digit table

Deepseek-R1 Thinking Process

Okay, let’s tackle this problem step by step. First, I need to understand all the requirements and how the components are connected.
 Wait , the connections say that ...
 Wait , let me think again. Let me list the segments: ...
 ...
 Now, moving on to the code structure.
 Wait , but the LEDs should alternate: ...
 Wait , the STCP is the latch pin. ...
 ...
Testing the code:
 Wait , looking back at the connections:
 Wait , the connections say:...
 The only possible issue is ...

Table 6: An example of Deepseek-R1 Thinking Process

SFT-distilled reasoning LLMs show significant instability when tackling embedded system development tasks. This instability suggest that SFT-distilled reasoning LLMs struggle with unfamiliar tasks, and may explain that in some cases these LLMs underperform compared to their base LLMs.

Compared to their teacher LLMs, both DeepSeek-R1-Distill-Llama and DeepSeek-R1-Distill-Qwen exhibit significantly higher variance in token usage during reasoning. For the “Setting ①” and “Setting ②” tasks, DeepSeek-R1-Distill-Llama shows variances of 40,552,828 and 35,707,662 tokens, respectively, while DeepSeek-R1-Distill-Qwen reaches 118,677,362 and 89,303,221 tokens—over an order of magnitude greater than the teacher model’s 11,435,790 and 7,748,574.

On Setting ③ tasks (ESP32 and Pi Pico), DeepSeek-R1-Distill-Llama averages only 696 and 569 thinking tokens, and DeepSeek-R1-Distill-Qwen averages 540 and 849, compared to DeepSeek-R1’s 3,760 and 3,848. These findings reinforce the idea that SFT-distilled reasoning models are more brittle and less consistent when faced with unfamiliar problem domains.

(e.g., Listing 1), and then adapt it based on the task requirements. These results show that reasoning LLMs may *entangle in unnecessarily complex thought processes*, limiting their ability to apply pretrained knowledge efficiently in domain-specific tasks.

Finding 3: For 7-segment display tasks, different type of LLMs face different difficulties: (1) For chat LLMs, the primary challenge lies in retrieving relevant pretraining knowledge and adapting it appropriately to the task-specific constraints. (2) For reasoning LLMs, the issue stems from overcomplicating the problem, leading to inefficient use of known information and reduced task performance.

5.2.3 *The gap between Setting ① and Setting ②.* From the results in Table 4, we observe an interesting phenomenon. For chat LLMs and SFT-distilled reasoning LLMs, performance on the “Setting ①” task is significantly better than on the “Setting ②” task. However, for reasoning LLMs, the “Setting ①” task does not show a clear advantage over the “Setting ②” task. In fact, for the QWQ model, the “Setting ②” performance is even better than the “Setting ①” result. In this section, we aim to analyze why the performance gap between these two tasks varies across different types of LLMs.

| QwQ Thinking Process in Different Tasks | |
|--|--|
| Setting ① thinking process | |
| Okay, let's tackle this problem step by step ... | |
| Looking at the connections: | |
| - The RGB LED's R, G, B pins are connected to Mega's digital pins 2,3,4. Since these are PWM pins (pins 2-13 have PWM)... | |
| - The 7-segment is driven via the shift register (sr1). The shift register's DS, SHCP, STCP are connected to Mega's A0, A1, A2. Wait, no, checking the connections again: the connections list says: ... | |
| Looking at the connections for the 7-segment: ... | |
| Think 9276 tokens | |
| So in the code: ... | |
| The bar graph and RGB LED parts should work as intended. | |
| Setting ② thinking process | |
| Okay, let's tackle this problem step by step. | |
| ... | |
| Now, let's outline the connections: | |
| For the shift register (sr1): | |
| - DS (data) to Arduino pin 2 | |
| - SHCP (shift clock) to pin 3... | |
| For the 7-segment display, | |
| Think 1565 tokens | |
| Now, compiling all the connections into the required format. | |
| Now, the code: ... | |

Table 7: An example of QWQ Thinking Process

As discussed in Section 4.2.1, reasoning LLMs such as DeepSeek-R1 and QwQ often use the keyword “wait” to indicate moments of

uncertainty or inconsistencies in their reasoning process. Therefore, the frequency of the word “wait” can serve as a rough proxy for the LLM’s perceived difficulty in handling a task. Interestingly, DeepSeek-R1 used the word “wait” 3,369 times in the “Setting ②” task, but this number increased to 4,503 in the “Setting ①” task. The discrepancy is even more pronounced for QwQ, which used “wait” 5,779 times in the “Setting ②” setting and 9,782 times in the “Setting ①” setting. Contrary to expectations, these results suggest that for LLMs with strong reasoning capabilities, it may actually be more effective to allow them to perform the schematics themselves, rather than supplying the correct schematics in advance.

To better understand this phenomenon, we examined the cases that QWQ successfully solved in the “Setting ②” task but failed to solve in the “Setting ①” setting. We found that the model often discovered alternative schematic schemes that were easier for it to reason about than the official, correct schematics. A representative example is shown in Table 7. In this example, the main difference lies in how the 74HC595’s DS, SHCP, and STCP pins are connected. In the correct schematics, since some tasks involve more complex hardware setups with limited digital ports, we standardize the pin connections to A0, A1, and A2 on the Arduino. However, for this specific task, connecting to pins 2, 3, and 4 would be a more natural and efficient choice. The LLM selected this alternative schematics in the “Setting ②” setting, leading to a smoother reasoning process. This small change had a significant impact. In the “Setting ②” task, QWQ used only 1,565 tokens to construct a digital logic table. In contrast, for the “Setting ①” task—despite being given the correct schematics—it used 9,276 tokens, indicating a much more complex and less efficient reasoning path.

Finding 4: For reasoning LLMs, rigidly enforcing a predefined reasoning path Instead, allowing LLMs the flexibility to determine their own schematic strategy can result in more effective problem-solving.

5.3 LLMs’ Upper-bound for Embedded Systems

Given the subpar performance of LLMs in our benchmark, this section explores methods to unlock their potential for embedded system development based on the analysis above.

R1-Retrieval As discussed in Section 4.2.2, we observe that LLMs encounter different challenges in the code generation task (Setting ① and ②). Chat LLMs can leverage pretrained knowledge but often struggle to adapt flexibly to specific problem contexts. In contrast, reasoning LLMs tend to approach tasks from an overly low-level perspective, missing higher-level abstractions. Ideally, an effective LLM should combine pretrained knowledge with adaptive reasoning capabilities, enabling it to adjust and apply its understanding to the specific problem dynamically.

To address this, we propose a method called R1-Retrieval, which combines the strengths of both DeepSeek-V3 and DeepSeek-R1. The approach is as follows: we first use LLM DeepSeek-V3 to generate code for embedded system tasks that can be addressed using pretrained knowledge alone. This generated code is then provided to DeepSeek-R1 as a form of retrieval-augmented input within its prompt. With this setup, DeepSeek-R1 can leverage the pretrained

| Methods | Setting ① | Setting ② | Setting ③ (ESP32) |
|--------------|-----------|-----------|-------------------|
| DeepSeek-R1 | 55.6 | 50.0 | 21.4 |
| R1-Retrieval | 65.1 | 53.1 | \ |
| R1-Compiler | \ | \ | 27.8 |

Table 8: The result of R1-Retrieval and R1-Compiler

output of DeepSeek-V3 as contextual knowledge, enabling it to reason more effectively and solve problems that require both learned knowledge and adaptive reasoning ability.

For evaluation, we selected a simple task for each of the seven main electronic components—tasks that DeepSeek-V3 can handle with ease. For each target task, if it involves a particular electronic component, the corresponding code from DeepSeek-V3 is included in the retrieval input. These code and the associated prompts are detailed in the Appendix F. The results are summarized in Table 8. With the R1-Retrieval method, the pass@1 score of DeepSeek-R1 improved from 55.6 to 65.1 on the “Setting ①” task, and from 50.0 to 53.1 on the “Setting ②” task.

R1-Compiler. For Setting ③ task development using the ESP32 and the ESP-IDF framework, LLMs often generate code with syntax errors, making deployment infeasible. To mitigate this, we introduce R1-Compiler, a method that incorporates the compiler as an external feedback mechanism to guide code repair. Specifically, we capture compiler error messages and feed them back into DeepSeek-R1, prompting it to revise its output accordingly. The detailed prompting strategy is provided in the Appendix G.

The incorporation of compiler feedback significantly reduces syntax errors from 34.1% (43/126) to 3.2% (4/126) after just one repair iteration. The final result is shown in Table 8, the pass@1 score increases from 21.4 to 27.8.

5.4 Implications

Although LLMs perform poorly on our fundamental embedded systems design benchmark, they exhibit remarkable potential. For example, even without being given the correct schematics, they are able to rely on their inherent reasoning abilities to find better solutions. When activated with pre-trained knowledge and provided with the correct schematics, the reasoning LLM can achieve a pass@1 score of 65.1%.

We believe that with more data and improved training methods, such as reinforcement learning, multi-agent collaboration, or multimodal LLM, it is possible to enable interaction between the virtual and physical worlds solely through LLMs. This could represent the first step toward achieving embodied intelligence.

6 Related Work

Benchmark for Embedded System. There are some benchmarks which related to embedded systems [9, 18, 33], but they mainly focus on evaluating LLMs from a coding perspective rather than assessing their ability to design embedded systems from scratch. Also, these evaluation methods often rely on serial output embedded within the generated code, which introduces a clear gap between the simulated results and the actual performance on hardware. Evaluating LLMs

on real hardware is also significantly more costly and resource-intensive. Our benchmark leverages the Wokwi simulation platform, enabling automated evaluation that closely reflects real hardware behavior, without the time and resource costs associated with using physical hardware components.

Large Language Models for Embedded Systems. Existing approaches that use LLMs for hardware platform programming primarily rely on integrated development environments or programming frameworks. However, they rarely enable genuine interaction between the LLM and the hardware[2, 3, 7, 15, 24]. Some studies investigated the use of LLMs for hardware-related work. These include generating code for Field Programmable Gate Arrays (FPGAs)[11, 21, 25], using LLMs to write Arduino code but focusing on human-in-the-loop testing[9], or leveraging serial output to provide additional information for LLMs debugging[33]. Despite these advancements, most of the work remains limited to code generation and does not involve complete embedded system design. Our benchmark is the first to allow an LLM to independently implement the full process of embedded system development.

7 Threat to Validity

Our study identifies three potential threats to validity, along with the steps we have taken to mitigate them.

First, the manual construction of our benchmark dataset introduces the possibility of subjective bias in labeling. To reduce this risk, two independent annotators labeled the data separately, and any differences were discussed and resolved through consensus.

Second, the LLMs used in our experiments inherently involve randomness, which can cause variability in the observed results. To mitigate this issue, we set the decoding temperature to 0 and employed greedy decoding during inference, ensuring deterministic and consistent results across multiple runs.

Third, our evaluation relies on the correctness and stability of the Wokwi simulation platform. Wokwi supports a large and active user community, hosting more than one million Arduino projects, 800,000 ESP32 projects, and 100,000 Raspberry Pi Pico projects. During the process of creating our benchmark and associated test cases, we encountered no unexpected behaviors, which enhances our confidence in the platform’s reliability.

8 Conclusion

In this work, we propose a benchmark to evaluate the ability of LLMs in embedded system design. Our benchmark consists of four types of tasks, each with 126 cases. These tasks assess the LLMs’ capabilities in embedded programming with given wiring, autonomous circuit wiring, end-to-end embedded system design, and cross-platform migration. We also provide an automated and accurate evaluation method and conduct experiments on 10 state-of-the-art LLMs. Through our analysis, we find that although the LLMs face challenges in solving these tasks, they also demonstrate significant potential. For instance, discovering wiring schemes that are even better than the reference solutions. Based on our findings, we propose several simple strategies that can effectively improve the performance of LLMs in embedded system development.

1161

References

- [1] Meta AI. 2024. Introducing meta llama 3: The most capable openly available llm to date. Blog. <https://ai.meta.com/blog/meta-llama-3/> Online; accessed 15-January-2024.
- [2] Thomas Ball, Abhijith Chatra, Peli de Halleux, Steve Hodges, Michal Moskal, and Jacqueline Russell. 2019. Microsoft MakeCode: embedded programming for education, in blocks and TypeScript. In *Proceedings of the 2019 ACM SIGPLAN symposium on SPLASH-E*. 7–12.
- [3] Robert W Brennan and Jonathan Lesage. 2022. Exploring the implications of OpenAI codex on education for industry 4.0. In *International Workshop on Service Orientation in Holonic and Multi-Agent Manufacturing*. Springer, 254–266.
- [4] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374* (2021).
- [5] Claude. 2025. Claude 3.7 Sonnet. <https://www.anthropic.com/clause/sonnet>
- [6] DeepSeek-AI. 2024. DeepSeek-V3 Technical Report. *arXiv:2412.19437 [cs, CL]* <https://arxiv.org/abs/2412.19437>
- [7] James Devine, Joe Finney, Peli de Halleux, Michal Moskal, Thomas Ball, and Steve Hodges. 2019. MakeCode and CODAL: Intuitive and efficient embedded systems programming for education. *Journal of Systems Architecture* 98 (2019), 468–483.
- [8] Xueying Du, Mingwei Liu, Kaixin Wang, Hanlin Wang, Junwei Liu, Yixuan Chen, Jiayi Feng, Chaofeng Sha, Xin Peng, and Yiling Lou. 2024. Evaluating large language models in class-level code generation. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 1–13.
- [9] Zachary Englhardt, Richard Li, Dilini Nissanka, Zhihan Zhang, Girish Narayanswamy, Joseph Breda, Xin Liu, Shwetak Patel, and Vikram Iyer. 2024. Exploring and characterizing large language models for embedded system development and debugging. In *Extended Abstracts of the CHI Conference on Human Factors in Computing Systems*. 1–9.
- [10] Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, et al. 2025. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948* (2025).
- [11] Zhi Guo, Walid Najjar, and Betul Buyukkurt. 2008. Efficient hardware code generation for FPGAs. *ACM Transactions on Architecture and Code Optimization (TACO)* 5, 1 (2008), 1–26.
- [12] Xueyu Hu, Kun Kuang, Jiankai Sun, Hongxia Yang, and Fei Wu. 2024. Leveraging print debugging to improve code generation in large language models. *arXiv preprint arXiv:2401.05319* (2024).
- [13] Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. 2023. Swe-bench: Can language models resolve real-world github issues? *arXiv preprint arXiv:2310.06770* (2023).
- [14] Manuel Jiménez, Rogelio Palomera, and Isidoro Couvertier. 2013. *Introduction to embedded systems*. Springer.
- [15] Philip Koopman, Howie Choset, Rajeev Gandhi, Bruce Krogh, Diana Marculescu, Priya Narasimhan, Joann M Paul, Ragunathan Rajkumar, Daniel Siewiorek, Asim Smailagic, et al. 2005. Undergraduate embedded system education at Carnegie Mellon. *ACM Transactions on Embedded Computing Systems (TECS)* 4, 3 (2005), 500–528.
- [16] Sumith Kulal, Panupong Pasupat, Kartik Chandra, Mina Lee, Oded Padon, Alex Aiken, and Percy S Liang. 2019. Spoc: Search-based pseudocode to code. *Advances in Neural Information Processing Systems* 32 (2019).
- [17] Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. 2024. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. *Advances in Neural Information Processing Systems* 36 (2024).
- [18] Mingjie Liu, Nathaniel Pinckney, Brucek Khailany, and Haoxing Ren. 2023. Verilogeval: Evaluating large language models for verilog code generation. In *2023 IEEE/ACM International Conference on Computer Aided Design (ICCAD)*. IEEE, 1–8.
- [19] Anton Lozhkov, Raymond Li, Loubna Ben Allal, Federico Cassano, Joel Lamy-Poirier, Nouamane Tazi, Ao Tang, Dmytro Pykhtiar, Jiawei Liu, Yuxiang Wei, Tianyang Liu, Max Tian, Deniz Koçetković, Arthur Zucker, Younes Belkada, Zijian Wang, Qian Liu, Dmitry Abulkhanov, Indraneil Paul, Zhuang Li, Wen-Ding Li, Megan Risdal, Jia Li, Jian Zhu, Terry Yue Zhuo, Evgenii Zheltonozhskii, Ni Osae Osae Dade, Wenhao Yu, Lucas Krauß, Naman Jain, Yixuan Su, Xuanli He, Manan Dey, Edoardo Abati, Yekun Chai, Niklas Muenninghoff, Xiangru Tang, Muh-tasham Oblokulov, Christopher Aiki, Marc Marone, Chenghao Mou, Mayank Mishra, Alex Gu, Binyuan Hui, Tri Dao, Armel Zebazé, Olivier Dehaene, Nicolas Patry, Canwen Xu, Julian McAuley, Han Hu, Torsten Scholak, Sébastien Paquet, Jennifer Robinson, Carolyn Jane Anderson, Nicolas Chapados, Mostofa Patwary, Nima Tajbakhsh, Yacine Jernite, Carlos Muñoz Ferrandis, Lingming Zhang, Sean Hughes, Thomas Wolf, Arjun Guha, Leandro von Werra, and Harm de Vries. 2024. StarCoder 2 and The Stack v2: The Next Generation. *arXiv:2402.19173 [cs, SE]*
- [20] Marc Marone and Benjamin Van Durme. 2023. DataPortraits. <https://dataportraits.org/>
- [21] Tomas G Moreira, Marco A Wehrmeister, Carlos E Pereira, Jean-François Petin, and Eric Levrat. 2010. Automatic code generation for embedded systems: From UML specifications to VHDL code. In *2010 8th IEEE International Conference on Industrial Informatics*. IEEE, 1085–1090.
- [22] Niklas Muenninghoff, Zitong Yang, Weijia Shi, Xiang Lisa Li, Li Fei-Fei, Hannaneh Hajishirzi, Luke Zettlemoyer, Percy Liang, Emmanuel Candes, and Tatsunori Hashimoto. 2025. s1: Simple test-time scaling. *arXiv preprint arXiv:2501.19393* (2025).
- [23] OpenAI. 2025. OpenAI o3-mini. <https://openai.com/index/openai-o3-mini/>
- [24] Sudeep Pasricha. 2022. Embedded systems education in the 2020s: Challenges, reflections, and future directions. In *Proceedings of the Great Lakes Symposium on VLSI 2022*. 519–524.
- [25] Abhinav Podili, Chi Zhang, and Viktor Prasanna. 2017. Fast and efficient implementation of convolutional neural networks on FPGA. In *2017 IEEE 28Th international conference on application-specific systems, architectures and processors (ASAP)*. IEEE, 11–18.
- [26] Qwen Team. 2025. QwQ-32B: Embracing the Power of Reinforcement Learning. <https://qwenlm.github.io/blog/qwq-32b/>
- [27] Wokwi. 2019. Wokwi: World's most advanced ESP32 simulator. <https://wokwi.com/>
- [28] Wayne Wolf. 2010. *High-performance embedded computing: architectures, applications, and methodologies*. Elsevier.
- [29] Chunqiu Steven Xia and Lingming Zhang. 2023. Keep the Conversation Going: Fixing 162 out of 337 bugs for \$0.42 each using ChatGPT. *arXiv preprint arXiv:2304.00385* (2023).
- [30] Ruiyang Xu, Jialun Cao, Yaojie Lu, Hongyu Lin, Xianpei Han, Ben He, Shing-Chi Cheung, and Le Sun. 2024. CRUXEval-X: A Benchmark for Multilingual Code Reasoning, Understanding and Execution. *arXiv preprint arXiv:2408.13001* (2024).
- [31] An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoran Wei, Huan Lin, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jingren Zhou, Junyang Lin, Kai Dang, Keming Lu, Keping Bao, Kexin Yang, Le Yu, Mei Li, Mingfeng Xue, Pei Zhang, Qin Zhu, Rui Men, Runji Lin, Tianhao Li, Tianyi Tang, Tingyu Xia, Xingzhang Ren, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yu Wan, Yuqiong Liu, Zeyu Cui, Zhenru Zhang, and Zihai Qiu. 2024. Qwen2.5 Technical Report. *arXiv preprint arXiv:2412.15115* (2024).
- [32] Aidan ZH Yang, Claire Le Goues, Ruben Martins, and Vincent Hellendoorn. 2024. Large language models for test-free fault localization. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*. 1–12.
- [33] Huanqi Yang, Mingzhe Li, Mingda Han, Zhenjiang Li, and Weitao Xu. 2024. EmbedGenius: Towards Automated Software Development for Generic Embedded IoT Systems. *arXiv preprint arXiv:2412.09058* (2024).
- [34] Li Zhong, Zilong Wang, and Jingbo Shang. 2024. Ldb: A large language model debugger via verifying runtime execution step-by-step. *arXiv preprint arXiv:2402.16906* (2024).

1209
1210
1211
1212
1213
1214
1215
1216
1217
1218
1219
1220
1221
1222
1223
1224
1225
1226
1227
1228
1229
1230
1231
1232
1233
1234
1235
1236
1237
1238
1239
1240
1241
1242
1243
1244
1245
1246
1247
1248
1249
1250
1251
1252
1253
1254
1255
1256
1257
1258
1259
1260
1261
1262
1263
1264
1265
1266
1267
1268
1269
1270
1271
1272
1273
1274
1275
1276

1277 A Hardware Descriptions

This section provides an overview of the basic hardware components used in our benchmark, including their functions and pin configurations.

1281

1282 LED

A Light Emitting Diode (LED) is a basic component used for visual indication

1285

1286 . Pins:

- 1287 • A – Anode (positive terminal)
- 1288 • C – Cathode (negative terminal)

1289

1290 Pushbutton

A pushbutton is a momentary switch used for digital input.

1292

1293 Pins:

- 1294 • 1.1, 1.r – First contact terminals
- 1295 • 2.1, 2.r – Second contact terminals

1296

1297 RGB LED

An RGB LED combines red, green, and blue LEDs in a single package to produce different colors.

1300

1301 Pins:

- 1302 • R, G, B – Controls red, green, and blue channels respectively
- 1303 • COM – Common cathode (connected to ground)

1304

1305 Seven-Segment Display

A seven-segment display is used for numerical output.

1306

1307 Pins:

- 1308 • A–G – Individual segment controls
- 1309 • DP – Decimal point
- 1310 • COM.1, COM.2 – Common anode pins

1311

1312 LED Bar Graph

An LED bar graph displays signal levels or statuses with multiple LEDs in a row.

1315

1316 Pins:

- 1317 • A1–A10 – Anodes of each LED
- 1318 • C1–C10 – Cathodes of each LED

1319

1320 Slide Potentiometer

A slide potentiometer provides analog output based on position.

1321

1322 Pins:

- 1323 • GND – Ground
- 1324 • SIG – Analog output
- 1325 • VCC – Supply voltage

1326

Attribute: The output value ranges from 0 to 1023.

1327

1328 Resistor

A resistor limits current flow in a circuit.

1330

1331 Pins:

- 1332 • 1, 2 – Connection terminals

1333

Attribute: Resistance value: $1000\Omega(1k\Omega)$

1334

1335 74HC595 Shift Register

A shift register expands the number of digital outputs using serial data input.

1336 Pins:

- 1337 • DS – Serial data input
- 1338 • SHCP – Shift register clock input
- 1339 • STCP – Storage register clock input (latch)
- 1340 • OE – Output enable (active low)
- 1341 • Q0–Q7 – Parallel outputs
- 1342 • MR – Master reset (active low)
- 1343 • GND – Ground
- 1344 • VCC – Supply voltage (typically 5V)

1348 Servo Motor

A servo motor allows precise angular positioning.

1349 Pins:

- 1350 • PWM – Control signal
- 1351 • V+ – Power supply (5V)
- 1352 • GND – Ground

1356 B Platform Descriptions

This section outlines the hardware platforms used in our benchmark, including general-purpose input/output (GPIO) pins and power configurations.

1357 Raspberry Pi Pico

A microcontroller board based on the RP2040 chip.

1363 Pins:

- 1364 • GP0–GP22 – Digital GPIO pins
- 1365 • GP26–GP28 – Analog input pins
- 1366 • GND.1–GND.8 – Ground connections
- 1367 • VSYS, VBUS, 3V3 – Power supply inputs and outputs
- 1368
- 1369

1370 Arduino Uno Rev3

A popular microcontroller board based on the ATmega328P.

1371 Pins:

- 1372 • 0–13 – Digital GPIO pins
- 1373 • A0–A5 – Analog input pins
- 1374 • GND.1–GND.3 – Ground pins
- 1375 • VIN – External power input (7–12V)
- 1376 • 5V – 5V power output
- 1377
- 1378
- 1379

1380 Arduino Mega

A microcontroller board with expanded I/O capabilities, based on the ATmega2560.

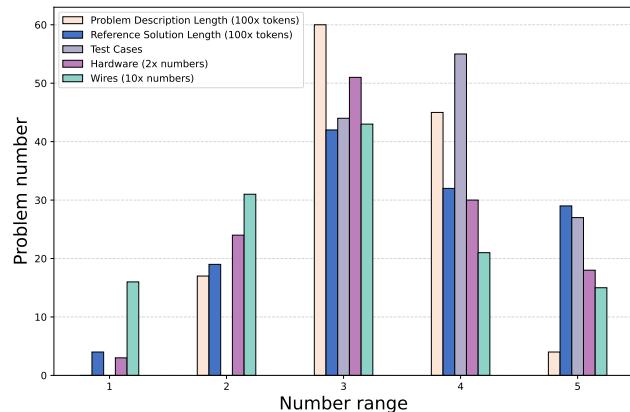
1382 Pins:

- 1383 • 0–53 – Digital GPIO pins
- 1384 • A0–A15 – Analog input pins
- 1385 • GND.1–GND.5 – Ground pins
- 1386 • VIN – External power input (7–12V)
- 1387 • 5V – 5V power output
- 1388
- 1389

Note: Pins 2–13, 44–46 support hardware PWM output (15 channels total).

1393 **ESP32-DevKitC V4**1394 A development board based on the ESP32 microcontroller with
1395 integrated Wi-Fi and Bluetooth.1396 **Pins:**

- 1397
- 0, 2, 4, 5, 12–19, 21–23, 25–27, 32, 33, 34, 35
1398 – GPIO pins
 - GND. 1–GND. 3 – Ground pins
 - 3V3 – 3.3V power output

1403 **C Data Annotation and Quality Control**1404 **Human Annotation**1405 In this section, we provide the detailed process for human annotation
1406 of our benchmark.1407 Initially, we recruited two bachelor’s degree holders as annotators.
1408 After undergoing rigorous training, each annotator independently performed the annotations. To ensure quality control,
1409 an additional pair of bachelor’s degree holders reviewed all annotated content. The guidelines for our annotation primarily cover
1410 the following aspects:1411 **(1) Optimal Hardware Resource Utilization Principle:** the
1412 problem description must ensure the complete and efficient utilization
1413 of all available hardware resources.1414 **(2) State Duration Stability Specification:** with the exception
1415 of instantaneous interaction scenarios (e.g., button-triggered interactions),
1416 all device states must maintain a stable duration of no less
1417 than two seconds.1418 **(3) The Test Validity Criteria:** test cases must satisfy com-
1419 pleteness criteria, meaning they should comprehensively verify
1420 the functional characteristics and correctness of the target prob-
1421 lem. Each test case should be designed with a specific validation
1422 objective in mind.1423 These guidelines aim to ensure the quality and effectiveness of
1424 the test case repository while enhancing the stability and repro-
1425 ducibility of the testing process.1426 **Quality Control**1427 To ensure the correctness of the benchmark, each example was
1428 independently annotated by two annotators to minimize subjective
1429 bias. Upon completing their annotations, the two annotators
1430 cross-verified each other’s results. Once both annotators reached a
1431 consensus, the annotated content was reviewed by two additional
1432 volunteers for further validation.1433 Volunteers meticulously examined the annotations and corrected
1434 errors. Through discussion, they selected the most accurate ver-
1435 sions. The finalized annotations include the problem statements,
1436 corresponding solutions, and test cases. During the review phase,
1437 volunteers adhered to the following standardized quality assess-
1438 ment protocol:1439 **(1) Problem Description Evaluation:** Assess the completeness
1440 and clarity of the problem formulation. Problem Description Evalu-
1441 ation: The problem formulation must be complete and clear, with
1442 unambiguous requirements covering all specifications.1443 **(2) Test Case Validation:** Verify the logical soundness of the test
1444 case design, ensuring alignment with the problem’s functional and1445 edge-case requirements. Evaluate whether the test suite provides
1446 sufficient validation for algorithmic correctness and robustness.1447 **(3) Stability Testing:** Perform a minimum of five independent
1448 test executions under controlled conditions. We mitigated external
1449 interference (e.g., computational resource contention) to ensure
1450 reproducible results. Document response times and output consis-
1451 tency for all runs to assess system reliability.1452 **D Dataset Statistics**1453 **Figure 6: The dataset statistics. The horizontal axis number i**
1454 **means $[(i-1)^{*}interval, i^{*}interval]$, for example (100x tokens)**
1455 **means interval is 100.**1456 In this section, we present statistical insights into our dataset,
1457 covering Problem Description Length, Reference Solution Length,
1458 the number of electronic components, and the number of Wires
1459 used. As shown in Figure 6, most problem descriptions and reference
1460 solutions fall within the 200–400 tokens range. The distribution
1461 of test cases is relatively uniform, with a majority of problems
1462 containing four test cases. Additionally, the number of electronic
1463 components is mostly concentrated between 4 and 8, while the
1464 number of wires predominantly falls in the 10–40 range.

1466 1467 1468 1469 1470 1471 1472 1473 1474 1475 1476 1477 1478 1479 1480 1481 1482 1483 1484 1485 1486 1487 1488 1489 1490 1491 1492 1493 1494 1495 1496 1497 1498 1499 1500 1501 1502 1503 1504 1505 1506 1507 1508

1509 E Thinking Process of 7-Segment Display

1510 Part of Thinking Process in 7-Segment Display

1511 ...
 1512 For digit 1, the bits would be:
 1513 Bit7 (Q7): 0 Bit6 (Q6): 1 Bit5 (Q5): 1 Bit4 (Q4): 1 Bit3 (Q3): 1 Bit2 (Q2): 0
 1514 Bit1 (Q1): 0 Bit0 (Q0): 1
 1515 So binary 0 1 1 1 1 0 0 1 → which is 0b01111001 (0x79). Wait, but that
 1516 would set Q0 to 1 (A off), Q1 to 0 (B on), Q2 to 0 (C on), and the rest (Q3-6
 1517 to 1, Q7 0). So segments B and C are on. That's correct. So the byte for
 1518 digit 1 is 0x79.
 1519 **This process is tedious**, but necessary. Let's do this for each digit:
 1520 Digit 0: 0x40 (binary 01000000) → A-F on, G off.
 1521 Digit 1: 0x79 (0b01111001) → B and C on.
 1522 Digit 2: Segments A, B, G, E, D → A (Q0=0), B (Q1=0), G (Q6=0), E (Q4=0),
 1523 D (Q3=0). So which bits are 0? Q0, Q1, Q3, Q4, Q6. So the byte bits:
 1524 Q7:0 Q6:0 (G on) Q5:1 (F off) Q4:0 (E on) Q3:0 (D on) Q2:1 (C off) Q1:0 (B
 1525 on) Q0:0 (A on)
 1526 So bits: 0 0 1 0 0 1 0 0 → binary 00100100 → 0x24. Wait, let's write it as
 1527 8 bits: 0 0 1 0 0 1 0 0 → that's 0x24 (36). Wait:
 1528 Q7:0 → bit7:0 Q6:0 → bit6:0 Q5:1 → bit5:1 Q4:0 → bit4:0 Q3:0 → bit3:0
 1529 (Wait, D is on, so Q3 is 0 → bit3 is 0) Wait, no. Wait, for digit 2, D is on
 1530 (Q3=0). So Q3 is 0 → bit3 is 0. Let me re-calculate:
 1531 Segments for digit 2: A, B, G, E, D → A (Q0=0), B (Q1=0), G (Q6=0), E
 1532 (Q4=0), D (Q3=0). Other segments (C and F) are off → Q2=1, Q5=1.
 1533 So the bits:
 1534 Q7:0 Q6:0 Q5:1 Q4:0 Q3:0 Q2:1 Q1:0 Q0:0
 1535 Which is 0 0 1 0 0 1 0 0 → binary 00100100 → 0x24. So yes, 0x24 for digit
 1536 2.
 1537 Hmm, **this is time-consuming**. Maybe there's a better way to approach
 1538 this. Alternatively, perhaps we can create a lookup table using a helper
 1539 function that for each digit, defines which segments are on (LOW), and
 1540 then build the byte accordingly.
 1541 Alternatively, perhaps using a standard 7-segment code table for common
 1542 anode displays. For example, the standard byte codes for a common anode
 1543 7-segment display connected to a shift register.
 1544 Wait, the standard segment codes for common anode (assuming segments
 1545 a to g are connected to Q0 to Q6):
 1546 For example, the standard code for 0 is 0x03 (binary 00000011) but that
 1547 depends on the wiring. Wait, perhaps this approach is error-prone. Let
 1548 me look for a standard 7-segment code table.
 1549 Upon checking, the standard byte patterns for a common anode 7-
 1550 segment display (assuming segments are a, b, c, d, e, f, g, dp connected to
 1551 Q0-Q7), but in our case, the segments are connected as:
 1552 Q0 → A Q1 → B Q2 → C Q3 → D Q4 → E Q5 → F Q6 → G
 1553 So each segment corresponds to a specific Q output. The standard codes
 1554 for the digits would then need to be mapped to these Q outputs. Let's list
 1555 the segments for each digit and then map to the corresponding bits.
 1556 ...

1557 F Prompt and Example of R1-Retrieval

1558 R1-Retrieval Prompt

1559 { prompt same as "w/ schematics" or "w/o schematics" }
 1560
 1561 **### Retrieve Code**
 1562 Here are some retrieved code snippets that may be helpful for solving
 1563
 1564
 1565

1566 the problem:

1567 LED

```
1568 // Define the pin where the LED is connected
1569 const int ledPin = 2;
1570
1571 void setup() {
1572     // Initialize the LED pin as an output
1573     pinMode(ledPin, OUTPUT);
1574     // Ensure the LED is off initially
1575     digitalWrite(ledPin, LOW);
1576 }
1577
1578 void loop() {
1579     // Turn the LED on for 1 second
1580     digitalWrite(ledPin, HIGH);
1581     delay(1000);
1582
1583     // Turn the LED off for 2 seconds
1584     digitalWrite(ledPin, LOW);
1585     delay(2000);
1586
1587     // Turn the LED on for 3 seconds
1588     digitalWrite(ledPin, HIGH);
1589     delay(3000);
1590
1591     // Turn the LED off for 3 seconds
1592     digitalWrite(ledPin, LOW);
1593     delay(3000);
1594
1595     // The sequence will repeat indefinitely due to the
1596     // loop() function
1597 }
```

1598 PushButton

```
1599 // Define the pins for the RGB LED and the push button
1600 const int redPin = 2;      // Red pin of the RGB LED
1601 const int greenPin = 3;    // Green pin of the RGB LED
1602 const int bluePin = 4;     // Blue pin of the RGB LED
1603 const int buttonPin = 5;   // Push button pin
1604
1605
1606 // Variables to manage the button state and debouncing
1607 int buttonState = HIGH;    // Current state of the button
1608 int lastButtonState = HIGH; // Previous state of the
1609 // button
1610 unsigned long lastDebounceTime = 0; // Timestamp of the
1611 // last button state change
1612 const unsigned long debounceDelay = 50; // Debounce delay
1613 // in milliseconds
1614
1615 // Variable to track the current color state
1616 int colorState = 0; // 0: Off, 1: Red, 2: Green, 3: Blue
1617
1618 void setup() {
1619     // Set the RGB LED pins as OUTPUT
1620     pinMode(redPin, OUTPUT);
1621     pinMode(greenPin, OUTPUT);
1622     pinMode(bluePin, OUTPUT);
```

```

1625
1626
1627 // Set the button pin as INPUT_PULLUP
1628 pinMode(buttonPin, INPUT_PULLUP);
1629
1630 // Initialize the RGB LED to be off
1631 digitalWrite(redPin, LOW);
1632 digitalWrite(greenPin, LOW);
1633 digitalWrite(bluePin, LOW);
1634 }
1635
1636 void loop() {
1637     // Read the current state of the button
1638     int reading = digitalRead(buttonPin);
1639
1640     // Check if the button state has changed (due to noise
1641     // or pressing)
1642     if (reading != lastButtonState) {
1643         // Reset the debounce timer
1644         lastDebounceTime = millis();
1645     }
1646
1647     // Check if the debounce delay has passed
1648     if ((millis() - lastDebounceTime) > debounceDelay) {
1649         // If the button state has changed
1650         if (reading != buttonState) {
1651             buttonState = reading;
1652
1653             // If the button is pressed (LOW state)
1654             if (buttonState == LOW) {
1655                 // Cycle through the color states
1656                 colorState = (colorState + 1) % 4;
1657
1658                 // Update the RGB LED based on the current color
1659                 // state
1660                 switch (colorState) {
1661                     case 0: // Off
1662                         digitalWrite(redPin, LOW);
1663                         digitalWrite(greenPin, LOW);
1664                         digitalWrite(bluePin, LOW);
1665                         break;
1666                     case 1: // Red
1667                         digitalWrite(redPin, HIGH);
1668                         digitalWrite(greenPin, LOW);
1669                         digitalWrite(bluePin, LOW);
1670                         break;
1671                     case 2: // Green
1672                         digitalWrite(redPin, LOW);
1673                         digitalWrite(greenPin, HIGH);
1674                         digitalWrite(bluePin, LOW);
1675                         break;
1676                     case 3: // Blue
1677                         digitalWrite(redPin, LOW);
1678                         digitalWrite(greenPin, LOW);
1679                         digitalWrite(bluePin, HIGH);
1680                         break;
1681                 }
1682             }
1683         }
1684     }
1685 }
```

```

1683     // Save the current reading as the last button state
1684     lastButtonState = reading;
1685 }
1686
1687 RGB LED
1688
1689
1690 // Define the pins for the RGB LED
1691 const int redPin = 2;    // Red LED connected to pin 2
1692 const int greenPin = 3;  // Green LED connected to pin 3
1693 const int bluePin = 4;   // Blue LED connected to pin 4
1694
1695 void setup() {
1696     // Set the RGB LED pins as output
1697     pinMode(redPin, OUTPUT);
1698     pinMode(greenPin, OUTPUT);
1699     pinMode(bluePin, OUTPUT);
1700
1701     // Initially turn off the RGB LED
1702     digitalWrite(redPin, LOW);
1703     digitalWrite(greenPin, LOW);
1704     digitalWrite(bluePin, LOW);
1705 }
1706
1707 void loop() {
1708     // Red for 3 seconds
1709     digitalWrite(redPin, HIGH);
1710     digitalWrite(greenPin, LOW);
1711     digitalWrite(bluePin, LOW);
1712     delay(3000);
1713
1714     // Yellow (Red + Green) for 3 seconds
1715     digitalWrite(redPin, HIGH);
1716     digitalWrite(greenPin, HIGH);
1717     digitalWrite(bluePin, LOW);
1718     delay(3000);
1719
1720     // White (Red + Green + Blue) for 3 seconds
1721     digitalWrite(redPin, HIGH);
1722     digitalWrite(greenPin, HIGH);
1723     digitalWrite(bluePin, HIGH);
1724     delay(3000);
1725
1726     // Turn off the LED for 3 seconds
1727     digitalWrite(redPin, LOW);
1728     digitalWrite(greenPin, LOW);
1729     digitalWrite(bluePin, LOW);
1730     delay(3000);
1731 }
1732
1733 Slide Potentiometer
1734
1735
1736 // Define the pins for the RGB LED
1737 const int redPin = 2;    // Red LED connected to digital
1738     // pin 2
1739 const int greenPin = 3;  // Green LED connected to digital
1740     // pin 3
1741 const int bluePin = 4;   // Blue LED connected to digital
1742     // pin 4
```

```

1741 // Define the pin for the slide potentiometer
1742 const int potPin = A3; // Slide potentiometer connected
1743 ← to analog pin A3
1744
1745 void setup() {
1746     // Set the RGB LED pins as output
1747     pinMode(redPin, OUTPUT);
1748     pinMode(greenPin, OUTPUT);
1749     pinMode(bluePin, OUTPUT);
1750
1751     // Initialize the RGB LED to be off
1752     digitalWrite(redPin, LOW);
1753     digitalWrite(greenPin, LOW);
1754     digitalWrite(bluePin, LOW);
1755 }
1756
1757 void loop() {
1758     // Read the value from the slide potentiometer (0 to
1759     ← 1023)
1760     int potValue = analogRead(potPin);
1761
1762     // Determine the color based on the potentiometer value
1763     if (potValue < 341) {
1764         // First third of the range: Red
1765         digitalWrite(redPin, HIGH);
1766         digitalWrite(greenPin, LOW);
1767         digitalWrite(bluePin, LOW);
1768     } else if (potValue >= 341 && potValue < 682) {
1769         // Second third of the range: Green
1770         digitalWrite(redPin, LOW);
1771         digitalWrite(greenPin, HIGH);
1772         digitalWrite(bluePin, LOW);
1773     } else {
1774         // Final third of the range: Blue
1775         digitalWrite(redPin, LOW);
1776         digitalWrite(greenPin, LOW);
1777         digitalWrite(bluePin, HIGH);
1778     }
1779
1780 7 Segment Display
1781
1782 #include <Servo.h>
1783
1784 // Pin definitions
1785 const int dataPin = A0; // DS pin of 74HC595
1786 const int latchPin = A1; // STCP pin of 74HC595
1787 const int clockPin = A2; // SHCP pin of 74HC595
1788
1789 // 7-segment display digit patterns (common anode)
1790 const byte digitPatterns[10] = {
1791     B11000000, // 0
1792     B1111001, // 1
1793     B10100100, // 2
1794     B10110000, // 3
1795     B10011001, // 4
1796     B10010010, // 5
1797     B10000010, // 6
1798     B11111000, // 7
1799     B10000000, // 8
1799
1799

```

```

1799 B10010000 // 9
1800 }
1801
1802 unsigned long previousMillis = 0; // Stores the last time
1803 ← the timer was updated
1804 const long interval = 1000; // Interval at which to
1805 ← update the timer (1 second)
1806 int timer; // Timer value
1807
1808 void setup() {
1809     // Set the pin modes
1810     pinMode(dataPin, OUTPUT);
1811     pinMode(latchPin, OUTPUT);
1812     pinMode(clockPin, OUTPUT);
1813
1814     // Initialize the 7-segment display to be off
1815     digitalWrite(latchPin, LOW);
1816     shiftOut(dataPin, clockPin, MSBFIRST, B11111111); // /
1817     ← Turn off all segments
1818     digitalWrite(latchPin, HIGH);
1819
1820 void loop() {
1821     unsigned long currentMillis = millis();
1822
1823     // Check if 1 second has passed
1824     if (currentMillis - previousMillis >= interval) {
1825         previousMillis = currentMillis;
1826
1827         // Increment the timer by 2 seconds
1828         timer += 2;
1829
1830         // Determine what to display on the 7-segment display
1831         int displayValue;
1832         if (timer > 10) {
1833             displayValue = timer % 10; // Display only the last
1834             ← digit
1835         } else {
1836             displayValue = timer; // Display the full timer value
1837         }
1838
1839         // Display the value on the 7-segment display
1840         digitalWrite(latchPin, LOW);
1841         shiftOut(dataPin, clockPin, MSBFIRST,
1842             ← digitPatterns[displayValue]);
1843         digitalWrite(latchPin, HIGH);
1844     }
1845
1846 Servo
1847
1848 #include <Servo.h>
1849
1850 // Define the pins
1851 const int btn1Pin = 2; // Button connected to digital pin
1852 ← 2
1853 const int servo1Pin = 3; // Servo1 connected to digital
1854 ← pin 3
1855 const int servo2Pin = 4; // Servo2 connected to digital
1856 ← pin 4
1856
1856

```

```

1857
1858
1859 // Create servo objects
1860 Servo servo1;
1861 Servo servo2;
1862
1863 // Variables to store the servo positions
1864 int servo1Pos = 0;
1865 int servo2Pos = 180;
1866
1867 // Variables for button debouncing
1868 int buttonState = HIGH;
1869 int lastButtonState = HIGH;
1870 unsigned long lastDebounceTime = 0;
1871 const unsigned long debounceDelay = 50; // Debounce delay
1872     ↪ in milliseconds
1873
1874 void setup() {
1875     // Initialize the button pin as input with pull-up
1876     ↪ resistor
1877     pinMode(btn1Pin, INPUT_PULLUP);
1878
1879     // Attach the servos to their respective pins
1880     servo1.attach(servo1Pin);
1881     servo2.attach(servo2Pin);
1882
1883     // Set initial positions of the servos
1884     servo1.write(servo1Pos);
1885     servo2.write(servo2Pos);
1886 }
1887
1888 void loop() {
1889     // Read the current state of the button
1890     int reading = digitalRead(btn1Pin);
1891
1892     // Check if the button state has changed (due to noise
1893     ↪ or pressing)
1894     if (reading != lastButtonState) {
1895         // Reset the debouncing timer
1896         lastDebounceTime = millis();
1897     }
1898
1899     // Check if the debounce delay has passed
1900     if ((millis() - lastDebounceTime) > debounceDelay) {
1901         // If the button state has changed
1902         if (reading != buttonState) {
1903             buttonState = reading;
1904
1905             // Only toggle the servo positions if the button is
1906             ↪ pressed (LOW)
1907             if (buttonState == LOW) {
1908                 // Toggle the servo positions
1909                 if (servo1Pos == 0) {
1910                     servo1Pos = 180;
1911                     servo2Pos = 0;
1912                 } else {
1913                     servo1Pos = 0;
1914                     servo2Pos = 180;
1915                 }
1916
1917             // Move the servos to their new positions
1918
1919 }
```

```

1920
1921     servo1.write(servo1Pos);
1922     servo2.write(servo2Pos);
1923 }
1924
1925 // Save the current reading as the last state for the
1926     ↪ next loop
1927     lastButtonState = reading;
1928 }
1929
1930 LED Bar Graph
1931
1932
1933 // Define the pins for bargraph1 and bargraph2
1934 int bargraph1Pins[] = {2, 3, 4, 5, 6, 7, 8, 9, 10, 11};
1935 int bargraph2Pins[] = {3, 4, 5, 6, 7, 8, 9, 10, 11, 12};
1936
1937 void setup() {
1938     // Initialize all LED pins as OUTPUT
1939     for (int i = 0; i < 10; i++) {
1940         pinMode(bargraph1Pins[i], OUTPUT);
1941         pinMode(bargraph2Pins[i], OUTPUT);
1942     }
1943
1944     // Turn off all LEDs initially
1945     for (int i = 0; i < 10; i++) {
1946         digitalWrite(bargraph1Pins[i], LOW);
1947         digitalWrite(bargraph2Pins[i], LOW);
1948     }
1949
1950 void loop() {
1951     // Left to right sequence
1952     for (int i = 0; i < 10; i++) {
1953         // Turn on the current LED in bargraph1
1954         digitalWrite(bargraph1Pins[i], HIGH);
1955
1956         // Turn on the previous LED in bargraph2 (if not the
1957         ↪ first LED)
1958         if (i > 0) {
1959             digitalWrite(bargraph2Pins[i - 1], HIGH);
1960         }
1961
1962         // Turn off the LED in bargraph2 if bargraph1 is at the
1963         ↪ first LED
1964         if (i == 0) {
1965             digitalWrite(bargraph2Pins[9], LOW);
1966         }
1967
1968         // Wait for 2 seconds
1969         delay(2000);
1970
1971         // Turn off the current LED in bargraph1
1972         digitalWrite(bargraph1Pins[i], LOW);
1973
1974         // Turn off the previous LED in bargraph2 (if not the
1975         ↪ first LED)
1976         if (i > 0) {
1977             digitalWrite(bargraph2Pins[i - 1], LOW);
1978         }
1979     }
1980 }
```

```

1973 }
1974 }
1975
1976 // Right to left sequence
1977 for (int i = 9; i >= 0; i--) {
1978   // Turn on the current LED in bargraph1
1979   digitalWrite(bargraph1Pins[i], HIGH);
1980
1981   // Turn on the previous LED in bargraph2 (if not the
1982   // first LED)
1983   if (i > 0) {
1984     digitalWrite(bargraph2Pins[i - 1], HIGH);
1985   }
1986
1987   // Turn off the LED in bargraph2 if bargraph1 is at the
1988   // first LED
1989   if (i == 0) {
1990     digitalWrite(bargraph2Pins[9], LOW);
1991   }
1992
1993   // Wait for 2 seconds
1994   delay(2000);
1995
1996   // Turn off the current LED in bargraph1
1997   digitalWrite(bargraph1Pins[i], LOW);
1998
1999   // Turn off the previous LED in bargraph2 (if not the
2000   // first LED)
2001   if (i > 0) {
2002     digitalWrite(bargraph2Pins[i - 1], LOW);
2003   }
2004 }

2005
2006 G Prompt of R1-Compiler
2007
2008 R1-Compiler Prompt
2009 The code is build failed because of the following error:
2010
2011
2012 { the error message from compiler }
2013
2014 Please fix the error
2015 Note:
2016 1. Your connections should in [CONNECTIONS] and [/CONNECTIONS]
2017 2. Your code should in [ESP32 Code] and [/ESP32 Code]
2018 3. Even if the origin connection or code is correct, you should still
2019 rewrite it.
2020 4. First give the connection and then give the code.
2021
2022
2023 Received 20 February 2007; revised 12 March 2009; accepted 5 June 2009
2024
2025
2026
2027
2028
2029
2030
2031
2032
2033
2034
2035
2036
2037
2038
2039
2040
2041
2042
2043
2044
2045
2046
2047
2048
2049
2050
2051
2052
2053
2054
2055
2056
2057
2058
2059
2060
2061
2062
2063
2064
2065
2066
2067
2068
2069
2070
2071
2072
2073
2074
2075
2076
2077
2078
2079
2080
2081
2082
2083
2084
2085
2086
2087
2088

```