Summer Internship Report

On

Integration of Arduino and MicroPython Evaluation Pipelines with Yaksh

Submitted by

Rishi Krishna S

School Of Engineering, CUSAT

Smita Takalkar

Savitribai Phule Pune University

Aayush Ankush

Vidyalankar Institute of Technology, Mumbai

Akshat Sharma

Maharaja Agrasen Institute Of Technology, Delhi

Under the guidance of

Prof. Prabhu Ramachandran

Department of Aerospace Engineering
IIT Bombay, Mumbai
Principal Invigilator, FOSSEE Project

Mr. Rajesh Kushalkar

Senior Project Manager, FOSSEE Project

IIT Bombay

Mr. Pratik Bhosale

Project Research Associate FOSSEE Project

IIT Bombay

Prathamesh Salunke

Research Assistants FOSSEE Project

IIT Bombay

December 2025

Acknowledgement

We would like to sincerely thank the **FOSSEE Project, Indian Institute of Technology Bombay**, for providing us with the opportunity to work on this project. The research-oriented environment and open-source culture fostered by FOSSEE enabled meaningful learning, hands-on experience, and practical exposure throughout the course of this work.

We are grateful to the FOSSEE team for their continued support and for creating a collaborative atmosphere that encouraged exploration and innovation.

We would like to express our sincere gratitude to **Prof. Prabhu Ramachandran**, Department of Aerospace Engineering, IIT Bombay, Mumbai, Principal Invigilator, FOSSEE Project, for his valuable guidance, support, and encouragement throughout the course of this project. We are also deeply thankful for his pivotal role in initiating and shaping Yaksh, which served as a strong foundation and inspiration for this work.

We would like to express our sincere gratitude to **Mr. Rajesh Kushalkar**, Senior Project Manager at the Indian Institute of Technology Bombay, for his guidance, encouragement, and valuable mentorship. His oversight and direction were instrumental in aligning the project with real-world requirements and research objectives.

We would also like to extend our heartfelt thanks to **Mr. Pratik Bhosale**, Project Research Associate at the FOSSEE Project, IIT Bombay, for his continuous guidance, technical insights, and constructive feedback throughout the project. His support played a crucial role in shaping the technical direction and successful completion of this work.

We are also thankful to **Mr. Prathamesh Salunke** of the FOSSEE Project, IIT Bombay, for his technical assistance, constructive discussions, and timely support during the development and integration phases of the project.

Rishi Krishna S, School Of Engineering, CUSAT

Smita Takalkar, Savitribai Phule Pune University

Aayush Ankush, Vidyalankar Institute of Technology, Mumbai

Akshat Sharma, Maharaja Agrasen Institute Of Technology, Delhi

Declaration

We hereby declare that this written submission represents our original work and ideas expressed in our own words. Wherever the ideas, data, or words of others have been included, they have been appropriately cited and referenced.

We affirm that all sources used in the preparation of this report have been accurately acknowledged. We further declare that we have strictly adhered to the principles of academic honesty and integrity and have not misrepresented, fabricated, or falsified any ideas, data, facts, or sources in this submission.

We understand that any violation of these principles may result in disciplinary action by the Institute and may also lead to legal consequences from the original authors or sources if proper citations or permissions have not been provided.

Rishi Krishna S, School Of Engineering, CUSAT

Smita Takalkar, Savitribai Phule Pune University

Aayush Ankush, Vidyalankar Institute of Technology, Mumbai

Akshat Sharma, Maharaja Agrasen Institute Of Technology, Delhi

Contents

Chapter 1: Introduction 6
1.1 Project Overview 6
Chapter 2: System Setup and Prerequisites 7
2.1 Operating System Requirements 7
2.2 Setting Up Linux Environment Using WSL (Windows Users) 7
2.2.1 Installing WSL 7
2.2.2 Activating WSL 7
2.3 System Package Prerequisites 7
2.3.1 Verifying Installation 8
2.4 Installing and Configuring ESP-IDF 8
2.4.1 Installing ESP-IDF Dependencies 8
2.4.2 Downloading and Installing ESP-IDF Tools 8
2.4.3 Setting Up ESP-IDF Environment Variables 9
2.5 Installing QEMU for ESP32 Emulation 9
2.5.1 Installing QEMU Dependencies 9
2.5.2 Installing Espressif QEMU Binaries 9
2.5.3 Verifying QEMU Installation 9
Chapter 3: Integrating Arduino .ino with yaksh 10
3.1 Design Approach; Arduino (.ino) to (.cpp) conversion 10
3.2 Conversion Script Implementation 10
3.2.1 Input Validation and File Handling 10
3.2.2 Auto-generated Header and Arduino Constants 12
3.2.3 Safe Copying of Student Code 12
3.2.4 Preservation of setup() and loop() 12
3.3 Hardware Simulation via test harness 12
3.4 Reusing Yaksh's existing cpp_evaluation 13
Chapter 4: ESP-IDF–Based Execution and Evaluation in Yaksh 14
4.1 Motivation for ESP-IDF and QEMU-based Evaluation 14
4.2 Overview of *EspIdfStdIOEvaluator* 14
4.3 Compilation and Execution Pipeline 15
4.4 Output Capture and Filtering 16
4.5 Error Handling and Diagnostics 17
4.6 Output Validation Strategy 18

4.7 Integration with Yaksh Grading System 19
Chapter 5: Arduino to ESP-IDF Build and QEMU Execution Script 20
5.1 Purpose of the Execution Script 20
5.2 Input Validation and File Preparation 20
5.3 ESP-IDF Environment Setup 21
5.4 Firmware Compilation Using ESP-IDF 21
5.5 Flash Image Generation 22
5.6 QEMU-based Execution 22
5.7 Watchdog Detection and Controlled Termination 23
5.8 Output Capture and Validation 24
5.9 Output Filtering 24
5.10 Role of the Script in Yaksh Integration 24
Chapter 6: Static Analysis & Instrumentation of .ino Using Tree-sitter 25
6.1 Motivation for Static Analysis in Arduino Evaluation 25
6.2 Overview of Tree-sitter 25
6.3 Parser Initialization and Language Setup 25
6.4 Detection of Arduino API Function Calls 26
6.4.1 Extracting Function Arguments 26
6.5 Variable Resolution and Assignment Tracking 27
6.5.1 Extracting Variable Assignments 27
6.6 Source Instrumentation for Runtime Observation 28
6.6.1 Instrumenting digitalWrite() Calls 28
6.7 Pin Validation Using CSV-Based Specifications 28
6.7.1 CSV Specification Format 28
6.7.2 Matching Function–Pin Usage 28
6.8 Pin Type Validation 29
6.9 Integration with the ESP-IDF Execution Pipeline 30
6.10 Advantages of the Tree-sitter–Based Approach 30
6.11 Summary 30
Chapter 7: Evaluation Outcomes with Examples 31
7.1 Evaluation Result States 31
7.1.1 Successful Execution 31
7.1.2 Build / Compilation Error 31
7.1.3 Output Mismatch 32
7.2 Example Evaluation Scenarios 33

7.2.1 Case C1: Correct API Usage and Expected Behavior 33
7.2.2 Case C2: Correct API with Incorrect Pin 34
7.2.3 Case C3: Missing Required API Usage 34
7.3 Combined Interpretation of Results 35
7.4 Summary 35
Chapter 8: MicroPython Evaluation Strategy in Yaksh 36
8.1 Observations from QEMU-Based ESP32 Execution for MicroPython 36
8.2 Introduction to the MicroPython Evaluation Track 36
8.3 Rationale for Using the MicroPython Unix Port 36
8.4 MicroPython Evaluation Using *QemuStdIOEvaluator* 37
8.4.1 Introduction 37
8.4.2 Purpose of QemuStdIOEvaluator 37
8.4.3 Class Structure 37
8.4.4 Initialization and Metadata Handling 37
8.4.5 Compilation and Environment Preparation 38
8.4.6 Execution Phase 38
8.4.7 Process Isolation and Control 38
8.4.8 Output Collection and Error Handling 39
8.4.9 Output Comparison and Grading 39
8.4.10 Limitations of the Script-Based MicroPython Evaluation Approach 39
8.5 Summary 39
Chapter 9: Need for a Hook-Based Evaluation Framework 40
Chapter 10: Architecture of the Hook-Based Evaluation System 41
10.1 Execution Model Overview 41
10.2 Runtime Module Injection Mechanism 41
10.3 Simulated Hardware Module (machine.py) 42
10.4 Simulated Time Module (time.py) 43
Chapter 11: Hook Evaluator Execution Flow and Integration Logic 44
11.1 Role of the Hook Evaluator 44
11.2 Injection of MicroPython Runtime Components 44
11.3 Execution Flow of check_answer() 45
Chapter 12: Writing Hook Test Cases 46
12.1 Structure of a Hook Test Case 46
12.2 Accessing Simulated Hardware Inside Tests 46
12.3 Typical Validation Pattern 46

12.4 Error Reporting and Partial Grading 47
Chapter 13: Execution and Validation Flow Inside a Hook Test 48
13.1 Executing Student Code in a Controlled Environment 48
13.2 Accessing Objects Created by the Student 48
13.3 Validating Object Configuration 48
13.4 Validating Functional Logic 49
Chapter 14: Writing Hook Test Cases: Design Principles and Execution Model 50
14.1 Why Objects Must Be Defined at the Global Level 50
14.2 How Hook Tests Interact with Global Objects 50
Chapter 15: Example: Temperature-Based LED Control 51
15.1 Problem Summary 51
15.2 Why Global Objects Are Required 52
15.3 How the Hook Executes the Student Code 52
15.4 Validating Hardware Configuration 53
15.5 Testing Functional Behavior 53
Chapter 16: Example: Temperature Measurement Using an NTC Thermistor 53
16.1 Problem Overview 54
16.2 Why Global Objects Are Required 55
16.3 How the Hook Executes the Student Code 55
16.4 Structural Validation 55
16.5 Functional Validation Logic 55
Chapter 17: Example: PWM-Based Motor Control 56
17.1 Problem Overview 56
17.2 Execution Model 57
17.3 Structural Validation 57
17.4 PWM Configuration Validation 57
Chapter 18: Example: Using a Timer Callback 58
18.1 Problem Overview 58
18.2 Structure of the Student Code 59
18.3 How the Hook Executes the Code 59
18.4 Conclusion 59

List Of Figures

Figure 2.1 Output of get_idf 9
Figure 3.1 C++ code evaluator modification 11
Figure 3.2ino to cpp converter script 11
Figure 4.1EspIdfStdIOEvaluator class initialisation 15
Figure 4.2Compling function inside evaluator class 16
Figure 4.3 function used for grading 17
Figure 4.4 function used for error handling 18
Figure 4.5 flexible output matching 19
Figure 4.6 Integration with Yaksh 19
Figure 5.1 Input validation script heading 20
Figure 5.2 Appending required headers 21
Figure 5.3 source the export.sh 21
Figure 5.4 Building the program via ESP-IDF 21
Figure 5.5 Building a flashable binary image 22
Figure 6.1 Analysis of C++ via tree sitter 25
Figure 6.2 finding digitalwrite() calls 26
Figure 6.3 Extracting arguments 27
Figure 6.4 convering digitalWrite into ESP_LOGI for better debugging 28
Figure 6.5 Checking for PIN correctness 29
Figure 6 Successful Build 31
Figure 7 Error on compiling 32
Figure 8 Error on output mismatch 33
Figure 9 Example Student program without errors 33
Figure 10 Expected output from the above porgram 34
Figure 11 The previous program with an induced error 34
Figure 12 The output for the above 34
Figure 13 Changed the analogRead into digitalRead 35
Figure 14Output of the Fig 27 35

Chapter 1: Introduction

Yaksh is an open source online assessment platform widely used for programming assignments in C, C++, Python, and other programming languages. However, it currently does not support Arduino based programming, where students can submit .ino files and rely on arduino specific functions such as setup(), loop(), pinMode(), analogRead() and PWM APIs.

This project extends Yaksh to evaluate Arduino programs automatically without requiring real hardware, by converting .ino files into standard cpp code and validation logic through simulated test cases.

Arduino-based programming allows students to write hardware-centric programs using .ino files and Arduino-specific abstractions such as setup(), loop(), pinMode(), digitalWrite(), analogRead(), and PWM APIs. Similarly, MicroPython enables programming microcontrollers using Python, with direct access to hardware peripherals through modules like machine, Pin, ADC, and PWM. These paradigms are widely adopted in introductory electronics, robotics, and IoT education, but are not directly compatible with Yaksh's existing evaluation pipeline.

1.1 Project Overview

This project extends Yaksh to support **Arduino and MicroPython program evaluation without requiring physical hardware**. For Arduino submissions, `.ino` files are automatically converted into standard C++ code, and Arduino-specific functions are mapped to a simulated execution environment. For MicroPython submissions, Python code written using MicroPython-specific APIs is executed and validated using a controlled simulation and mocking framework that emulates microcontroller behaviour.

By introducing support for both **ESP-IDF-based Arduino workflows** and **MicroPython**, the enhanced Yaksh platform enables automated assessment of embedded systems programming using simulated test cases. This significantly broadens Yaksh's applicability to hands-on electronics and embedded systems courses, while maintaining consistency, scalability, and reproducibility in evaluation.

Chapter 2: System Setup and Prerequisites

This chapter describes the system setup and prerequisite software required to enable ESP-IDF– and MicroPython–based program evaluation in Yaksh. Since embedded program compilation and emulation require a Linux-compatible environment, the setup process ensures consistency, reproducibility, and compatibility across development systems.

2.1 Operating System Requirements

The evaluation pipeline for ESP-IDF and MicroPython is designed to run on **GNU/Linux systems**, with Ubuntu or Debian-based distributions being preferred due to their official support and extensive documentation.

Users who already have a Linux system can skip the Windows Subsystem for Linux (WSL) setup and proceed directly to installing prerequisites. For Windows users, WSL provides a lightweight and reliable Linux environment without requiring dual booting or virtual machines.

2.2 Setting Up Linux Environment Using WSL (Windows Users)

Windows users must install and configure **Windows Subsystem for Linux (WSL)** to run a Linux distribution alongside Windows.

2.2.1 Installing WSL

Open PowerShell in administrator mode and execute:

wsl –install

This command enables the necessary Windows features and installs Ubuntu as the default Linux distribution. After installation, restart the system.

2.2.2 Activating WSL

After reboot, activate WSL by running:

wsl

To ensure operations occur within the Linux filesystem, switch to the home directory:

cd

This avoids working under */mnt*, which maps to the Windows filesystem and may cause permission or performance issues.

2.3 System Package Prerequisites

Once inside the Linux environment, system repositories should be updated to the latest versions:

sudo apt update && sudo apt upgrade -y

Install essential development tools required for compilation, scripting, and package management:

sudo apt install -y git python3 python3-pip cmake make gcc g++

2.3.1 Verifying Installation

Verify successful installation by checking version outputs:

git --version

python3 --version

pip3 --version

cmake --version

make --version

gcc --version

g++ --version

If each command prints a version number, the package is correctly installed.

2.4 Installing and Configuring ESP-IDF

ESP-IDF (Espressif IoT Development Framework) is the official development framework for ESP32-based systems. This project uses the **stable release** of ESP-IDF to ensure consistency and long-term compatibility.

2.4.1 Installing ESP-IDF Dependencies

ESP-IDF requires several additional packages for building and flashing applications:

*sudo apt-get install -y git wget flex bison gperf python3 python3-pip python3-venv *

cmake ninja-build ccache libffi-dev libssl-dev dfu-util libusb-1.0-0

2.4.2 Downloading and Installing ESP-IDF Tools

Create a directory for ESP-related tools and clone the ESP-IDF repository:

mkdir -p ~/esp

cd ~/esp

git clone -b v5.5.1 --recursive https://github.com/espressif/esp-idf.git

This project uses ESP-IDF version **v5.5.1** as the stable baseline, Install required ESP-IDF tools for ESP32 targets:

cd ~/esp/esp-idf

./install.sh esp32

2.4.3 Setting Up ESP-IDF Environment Variables

Before using ESP-IDF commands, environment variables must be configured

. $HOME/esp/esp-idf/export.sh

alias get_idf='. $HOME/esp/esp-idf/export.sh'

source ~/.bashrc

get_idf

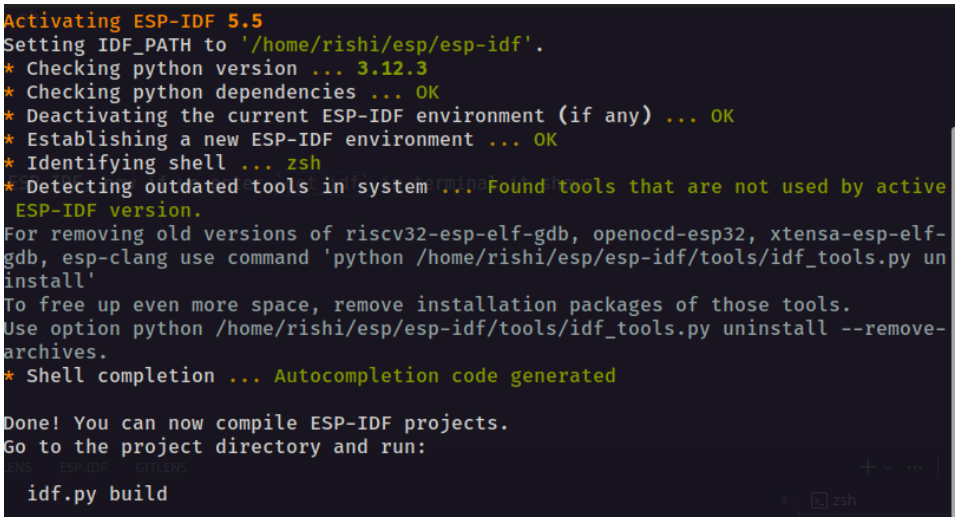


Figure 2.1 Output of get_idf

2.5 Installing QEMU for ESP32 Emulation

QEMU is used to emulate ESP32 firmware execution in a software-only environment. Espressif provides a fork of QEMU with support for ESP32 CPU, memory, and core peripherals.

2.5.1 Installing QEMU Dependencies

sudo apt-get install -y libgcrypt20 libglib2.0-0 libpixman-1-0 libsdl2-2.0-0 libslirp0

2.5.2 Installing Espressif QEMU Binaries

Activate ESP-IDF environment:

get_idf

python $IDF_PATH/tools/idf_tools.py install qemu-xtensa qemu-riscv32

cd ~/esp/esp-idf

. ./export.sh

2.5.3 Verifying QEMU Installation

python $IDF_PATH/tools/idf_tools.py list | grep qemu

Chapter 3: Integrating Arduino .ino with yaksh

Arduino programs use .ino files instead of .cpp, it depends on hardware APIs. Unlike .cpp files, arduino files do not contain main( ) function. Therefore, arduino files cannot be compiled using standard c++ compiler which Yaksh already have.

Yaksh's existing cpp_code_evaluator expects standard c++ source files, a test harness with main(). And a compilation and execution flow without hardware dependencies.

Thus the main goal of this project is to enable Yaksh to accept .ino submissions and evaluate them automatically using its existing c++ evaluator, without modifying Yaksh core logic heavily.

3.1 Design Approach; Arduino (.ino) to (.cpp) conversion

Arduino programs are written in .ino files and rely on the arduino runtime. These files cannot be compiled directly using a standard c++ compiler because they do not define main( ) function , depend on arduino specific headers and APIs and assumes hardware availability

To address this, the first step is automatic conversion of .ino files into valid .cpp files. This conversion process does not change the student's logic. Instead, it wraps the Arduino code in a form that a standard compiler understands.

This is done by:

A) Adding Arduino API stubs
B) Functions like pinMode(), digitalWrite(), analogRead(), and delay(), etc. are defined as empty or simulated functions so the compiler does not fail due to missing hardware libraries.

C) Preserving Student logic exactly as written

The original setup() and loop() functions remain unchanged. This ensures that the evaluation is based on the student's actual implementation and not a rewritten version.

B) Generating wrapper functions

Since arduino programs do not contain main() function wrapper functions such as run_setup() run_loop are automatically generated. These wrappers allows the test harness to explicitly invoke the student's setup() and loop() functions during evaluation.

This conversion step acts as a bridge between arduino programming model and standard cpp execution enabling further automated evaluation without modifying student submissions.

3.2 Conversion Script Implementation

The conversion is performed using a shell script that takes an Arduino .ino file as input and produces a corresponding .cpp file as output. This script acts as a preprocessing step before compilation.

3.2.1 Input Validation and File Handling

The script first validates that the input .ino file exists. If the file is missing, the conversion process is terminated to avoid invalid evaluation. The input .ino file path and output

.cpp file path are passed as command-line arguments, allowing seamless integration into Yaksh's submission pipeline.

```
converted_script = os.path.join(os.path.dirname(__file__),'..','scripts','converter_ino_cpp.sh')
for f in os.listdir(os.getcwd()):
    if f.endswith('.ino'):
        base = os.path.splittext(f)[0]
        cpp_file = base + '.cpp'

        subprocess.check_call([converter_script, f ,cpp_path])
        os.remove(f)
        self.submit_code_path = cpp_path
```

Figure 3.1 C++ code evaluator modification

```
#!/bin/sh
# Convert Arduino .ino → C++ for Yaksh evaluator
# Adds generic wrappers for setup() / loop()

set -e
INO_FILE="$1"
CPP_FILE="$2"

if [ ! -f "$INO_FILE" ]; then
    echo "Input file not found: $INO_FILE"
    exit 1
fi

cat > "$CPP_FILE" << 'HEADER'
// ===== Auto-generated from Arduino .ino =====
#include <stdio.h>
#define INPUT 0
#define OUTPUT 1
#define INPUT_PULLUP 2
#define HIGH 1
#define LOW 0
// Arduino API functions are provided by the test harness
HEADER

# Copy Arduino code safely (line by line)
while IFS= read -r line; do
    if [ "$line" = "#include <Arduino.h>" ]; then
        echo "// Arduino.h removed" >> "$CPP_FILE"
    else
        echo "$line" >> "$CPP_FILE"
    fi
done < "$INO_FILE"

cat >> "$CPP_FILE" << 'FOOTER'
// Yaksh evaluator wrappers
#ifdef __cplusplus
extern "C" {
#endif
void run_setup() { setup(); }
void run_loop()  { loop(); }
#ifdef __cplusplus
}
#endif
FOOTER
echo "Conversion complete: $CPP_FILE"
: $CPP_FILE"
```

Figure 3.2ino to cpp converter script

3.2.2 Auto-generated Header and Arduino Constants

At the beginning of the generated .cpp file, a header section is automatically inserted. This section defines commonly used Arduino constants such as INPUT, OUTPUT, HIGH, and LOW. These definitions ensure that references to standard Arduino macros do not cause compilation errors when using a standard C++ compiler.

Additionally, standard C headers such as <stdio.h> are included to support interaction with the test harness.

3.2.3 Safe Copying of Student Code

The student's Arduino code is copied line by line into the generated .cpp file. This approach ensures that, The original logic remains unchanged, Line structure is preserved for debugging and error reporting

If the script encounters #include <Arduino.h>, it removes the include directive, as the actual Arduino runtime is not available in the evaluation environment. This prevents dependency-related compilation failures while keeping the remainder of the code intact.

3.2.4 Preservation of setup() and loop()

The conversion process does not modify the setup() and loop() functions defined by the student. These functions remain exactly as written, ensuring that the evaluation is based solely on the student's implementation rather than a rewritten or adapted version of their code.

3.3 Hardware Simulation via test harness

Arduino programs are inherently hardware dependent. However, Yaksh operates in a sandbox software environment with no access to physical devices. To solve this, the project uses hardware simulation through test harness instead of real hardware.

Instructor can provide test harness in C that provides stub implementations of arduino API, records how these APIs are used by the student code and verifies the correctness using assertions. For example pinMode() can record which pin was configured and in which mode(input/output), analogRead() can return fixed value and record the pin accessed and digitalWrite() can track output usage and parameters.

After calling run_setup() and run_loop(), the test harness checks whether the student code behaved correctly by using standard C assertions. If expected behaviour is observed, the test passes, otherwise it fails.

This approach allows Yaksh to validate logical correctness, check pin configuration, ADC usage and PWM behaviour, also enforce correct API usage without needing hardware.

In essence, the test harness acts as a virtual arduino board, enabling meaningful evaluation of embedded logic in purely software environment.

3.4 Reusing Yaksh’s existing cpp_evaluation

A key design decision was **not to create a new evaluator** for Arduino programs.The project intentionally reuses Yaksh’s existing cpp_evaluator.

Once the .ino files converted to .cpp file and paired with test harness, Yaksh compiles the converted student code, linked it with the instructor-written test harness executes the resulting binary and determines pass/fail status based on the outcome.

By doing this no changes are required in Yaksh’s grading logic, existing compilation, execution and error handling mechanisms remain intact . arduino support becomes an extension and not a replacement.

This design approach ensures that the solution is minimally invasive, easy to maintain and consistent with Yaksh’s architecture.

Chapter 4: ESP-IDF–Based Execution and Evaluation in Yaksh

While source-level conversion enables Arduino programs to be compiled using a standard C++ toolchain, meaningful evaluation of embedded applications requires executing the code in an environment that closely resembles real microcontroller behavior. Simply compiling Arduino code without execution does not validate runtime logic, control flow, or peripheral interactions.

To address this, the project introduces an **ESP-IDF–based evaluator** for Yaksh that executes Arduino programs using **QEMU-based ESP32 emulation**, enabling realistic execution without requiring physical hardware.

4.1 Motivation for ESP-IDF and QEMU-based Evaluation

Arduino programs targeting ESP32 devices are ultimately compiled and executed using the ESP-IDF toolchain. ESP-IDF provides a robust build system, runtime environment, and compatibility with QEMU-based emulation.

Using QEMU allows Yaksh to:

- Execute embedded code in a sandboxed environment
- Capture runtime output via standard input/output
- Validate control flow and logic without physical boards

This approach bridges the gap between high-level Arduino sketches and low-level embedded execution.

4.2 Overview of *EspIdfStdIOEvaluator*

The *EspIdfStdIOEvaluator* is a custom evaluator class implemented by extending Yaksh's BaseEvaluator. It is responsible for:

- Accepting Arduino *.ino* submissions
- Converting and compiling them using ESP-IDF
- Executing the compiled firmware using QEMU
- Capturing and validating runtime output

The evaluator relies on an external build-and-run script that encapsulates Arduino-to-ESP-IDF conversion and QEMU execution, allowing the evaluator logic to remain clean and modular.

```python
class EspIdfStdIOEvaluator(BaseEvaluator):
    """
    Evaluator for ESP-IDF code using QEMU and arduino_to_esp/ino_to_running.sh
    """
    def __init__(self, metadata, test_case_data):
        self.files = []
        self.script_path = os.path.join(os.path.dirname(__file__), 'arduino_to_esp/ino_to_running.sh')
        self.output_file = os.path.join(os.path.dirname(__file__), 'arduino_to_esp/filtered_output.txt')
        self.raw_output_file = os.path.join(os.path.dirname(__file__), 'arduino_to_esp/output.txt')
        self.build_log_file = os.path.join(os.path.dirname(__file__), 'arduino_to_esp/build.log')
        self.timeout = 20  # seconds, can be set from settings if needed

        # Set metadata values
        self.user_answer = metadata.get('user_answer')
        self.file_paths = metadata.get('file_paths')
        self.partial_grading = metadata.get('partial_grading')

        # Set test case data values
        self.expected_input = test_case_data.get('expected_input')
        self.expected_output = test_case_data.get('expected_output')
        self.weight = test_case_data.get('weight')
        self.hidden = test_case_data.get('hidden')

        # For diagnostics
        self.diagnostic_info = ""
        self.compile_error = None  # Store compilation errors to show before output comparison
```

Figure 4.1EspIdfStdIOEvaluator class initialisation

During initialization, the evaluator extracts key information from Yaksh-provided metadata and test case configuration, including:

- Student-submitted source code
- Expected output for the test case
- Grading weight and visibility
- Partial grading configuration

Several file paths are also initialized to store:

- Raw QEMU output
- Filtered program output
- ESP-IDF build logs

A configurable timeout is defined to accommodate the longer execution time required by ESP-IDF builds and QEMU boot sequences.

4.3 Compilation and Execution Pipeline

The compilation process begins by writing the student's submission to a temporary `.ino` file within the ESP-IDF working directory. This ensures compatibility with the Arduino-to-ESP

conversion pipeline.Before execution, any previous output artifacts are removed to prevent stale data from affecting results.

Unlike standard Yaksh evaluators, the global signal-based timeout is disabled. ESP-IDF compilation and QEMU execution frequently exceed Yaksh's default timeout limits. Instead, a controlled subprocess timeout is applied during execution to prevent infinite hangs while allowing sufficient time for firmware boot and execution.

```
def compile_code(self):
    # Write user code to temp .ino file in arduino_to_esp
    arduino_to_esp_dir = os.path.abspath(os.path.join(os.path.dirname(__file__), 'arduino_to_esp'))
    ino_file = os.path.join(arduino_to_esp_dir, 'submission.ino')

    logger.info(f"[ESP-IDF Eval] Starting compilation for user code")
    logger.info(f"[ESP-IDF Eval] Arduino directory: {arduino_to_esp_dir}")
    logger.info(f"[ESP-IDF Eval] INO file path: {ino_file}")
    logger.info(f"[ESP-IDF Eval] Script path: {self.script_path}")

    # Write user answer to file
    try:
        with open(ino_file, 'w') as f:
            f.write(self.user_answer)
        logger.info(f"[ESP-IDF Eval] User code written to {ino_file}")
    except Exception as e:
        error_msg = f"Failed to write user code to file: {str(e)}"
        logger.error(f"[ESP-IDF Eval] {error_msg}")
        self.output_value = ''
        return False, error_msg

    # Remove previous output files
    for output_file in [self.output_file, self.raw_output_file]:
        if os.path.exists(output_file):
            try:
                os.remove(output_file)
                logger.info(f"[ESP-IDF Eval] Removed previous output file: {output_file}")
            except Exception as e:
                logger.warning(f"[ESP-IDF Eval] Failed to remove {output_file}: {str(e)}")

    # Disable the global signal alarm during QEMU execution
    # ESP-IDF build + QEMU execution can take 15+ seconds, which exceeds Yaksh's default 4s timeout
    # We use our own subprocess timeout instead
    logger.info(f"[ESP-IDF Eval] Disabling global signal alarm for long-running QEMU execution")
    signal.alarm(0)  # Cancel any pending alarm

    # Run the script from arduino_to_esp directory
    logger.info(f"[ESP-IDF Eval] Executing build script with timeout={self.timeout}s")
    try:
        proc = subprocess.run([
            'bash', self.script_path, ino_file
        ], cwd=arduino_to_esp_dir,
           stdout=subprocess.PIPE, stderr=subprocess.PIPE, timeout=self.timeout)

        stdout_text = proc.stdout.decode('utf-8') if proc.stdout else ""
        stderr_text = proc.stderr.decode('utf-8') if proc.stderr else ""

        logger.info(f"[ESP-IDF Eval] Script execution completed with return code: {proc.returncode}")
        if stdout_text:
            logger.info(f"[ESP-IDF Eval] Script STDOUT:\n{stdout_text}")
        if stderr_text:
            logger.warning(f"[ESP-IDF Eval] Script STDERR:\n{stderr_text}")

    except subprocess.TimeoutExpired:
        error_msg = f'Timeout: Compilation/QEMU execution exceeded {self.timeout}s time limit. QEMU may need more time to boot and execute. Consider increasing timeout.'
        logger.error(f"[ESP-IDF Eval] {error_msg}")
        self.output_value = ''
        self.diagnostic_info = error_msg
        return False, error_msg
```

Figure 4.2Compling function inside evaluator class

The evaluator then invokes a shell script that:

1. Converts the Arduino sketch to ESP-IDF–compatible code
2. Builds the firmware using ESP-IDF
3. Executes the firmware using QEMU

Both standard output and error streams are captured for diagnostics.

4.4 Output Capture and Filtering

QEMU execution produces verbose output, including:

- Bootloader messages
- ESP-IDF initialization logs
- Watchdog warnings

To ensure fair evaluation, the evaluator distinguishes between:

- **Raw output**, which contains all QEMU logs
- **Filtered output**, which contains only the student program's actual output

Only the filtered output is used for grading, while raw output is retained for debugging and diagnostic purposes.

```
def check_code(self):
    # Compare output using flexible sequence matching
    # Check if all expected lines appear in actual output in the same order
    # (but ignores extra lines like watchdog messages)

    def normalize(s):
        """Normalize by stripping whitespace from each line"""
        return '\n'.join(line.strip() for line in s.strip().splitlines() if line.strip())

    actual = normalize(self.output_value)
    expected = normalize(self.expected_output)

    # Split into lines for comparison
    expected_lines = expected.split('\n') if expected else []
    actual_lines = actual.split('\n') if actual else []

    # Flexible matching: check if all expected lines appear in actual output in sequence
    # Extra lines in actual output are ignored (e.g., watchdog messages)
    is_correct = True
    if expected_lines:
        expected_idx = 0
        for actual_line in actual_lines:
            if expected_idx < len(expected_lines) and expected_lines[expected_idx] == actual_line:
                expected_idx += 1
        # All expected lines must be found
        is_correct = (expected_idx == len(expected_lines))
    else:
        # If no expected output, consider it correct if there's any output
        is_correct = bool(actual.strip())

    mark_fraction = self.weight

    logger.info(f"[ESP-IDF Eval] Output comparison (flexible sequence matching):")
    logger.info(f"[ESP-IDF Eval] Expected output:\n{expected}")
    logger.info(f"[ESP-IDF Eval] Actual output (first 500 chars):\n{actual[:500]}")
    logger.info(f"[ESP-IDF Eval] Expected lines: {len(expected_lines)}")
    logger.info(f"[ESP-IDF Eval] Actual lines: {len(actual_lines)}")
    logger.info(f"[ESP-IDF Eval] Match result: {is_correct}")

    # Build detailed error message
    error_msg = None

    # If there was a compilation error, show that first instead of output mismatch
    if self.compile_error:
        return False, self.compile_error, mark_fraction

    if not is_correct:
        # Show which expected lines are missing
        logger.warning(f"[ESP-IDF Eval] Output mismatch detected!")
        expected_idx = 0
        missing_lines = []

        for i, actual_line in enumerate(actual_lines):
            if expected_idx < len(expected_lines) and expected_lines[expected_idx] == actual_line:
                logger.info(f"[ESP-IDF Eval] Line MATCHED: '{actual_line}'")
                expected_idx += 1
```

Figure 4.3 function used for grading

4.5 Error Handling and Diagnostics

If the build or execution process fails, the evaluator attempts to extract meaningful error messages from the ESP-IDF build logs.

A dedicated error extraction routine filters out build system noise and presents:

- Compiler error messages
- Line numbers and function context
- Relevant source code snippets

This significantly improves the quality of feedback shown to students, making debugging feasible even in a complex embedded toolchain, Compilation errors are stored separately and prioritized during evaluation, ensuring that users see build issues before output mismatch errors.

```
def _extract_compiler_errors(self, build_log_content):
    """Extract only the relevant compiler error messages from build.log"""
    if not build_log_content:
        return ""

    lines = build_log_content.split('\n')
    error_lines = []

    # Collect all lines that are part of compiler errors
    i = 0
    while i < len(lines):
        line = lines[i]
        # Look for error/warning messages from the compiler
        if 'error:' in line or 'In function' in line:
            # For "In function" lines, simplify the path
            if 'In function' in line:
                # Extract just the function part, remove the file path
                parts = line.split(': In function ')
                if len(parts) > 1:
                    error_lines.append(f"In function {parts[1]}")
                else:
                    error_lines.append(line)
            # For error lines, simplify to show just line:col and error message
            elif 'error:' in line:
                # Extract line number and error from: /path/to/file:13:22: error: message
                if ':' in line:
                    parts = line.split(':')
                    if len(parts) >= 4:
                        line_num = parts[1]
                        col_num = parts[2]
                        error_msg = ':'.join(parts[3:]).strip()
                        error_lines.append(f"Line {line_num}:{col_num}: {error_msg}")
                    else:
                        error_lines.append(line)
                else:
                    error_lines.append(line)

            # For error lines, also include the context (source code and pointer lines)
            if 'error:' in line:
                # Include the next few lines that show source code and error pointer
                j = i + 1
                while j < len(lines) and j < i + 10:
                    context_line = lines[j]
                    error_lines.append(context_line)
                    # Stop if we hit another compiler message or empty line followed by new error
                    if context_line.strip() == '' or context_line.startswith('/'):
                        break
                    # Also stop after showing the error pointer line (has spaces and ^ character)
                    if '^' in context_line and '|' in context_line:
                        j += 1
                        # Include one more line for the suggested fix (the ; line)
                        if j < len(lines) and '|' in lines[j]:
                            error_lines.append(lines[j])
                        break
                    j += 1
                i = j - 1
        i += 1
```

Figure 4.4 function used for error handling

4.6 Output Validation Strategy

Embedded program output may contain extra lines due to system logs or runtime warnings. Therefore, strict line-by-line comparison is unsuitable.

```
# Flexible matching: check if all expected lines appear in actual output in sequence
# Extra lines in actual output are ignored (e.g., watchdog messages)
is_correct = True
if expected_lines:
    expected_idx = 0
    for actual_line in actual_lines:
        if expected_idx < len(expected_lines) and expected_lines[expected_idx] == actual_line:
            expected_idx += 1
    # All expected lines must be found
    is_correct = (expected_idx == len(expected_lines))
else:
    # If no expected output, consider it correct if there's any output
    is_correct = bool(actual.strip())
```

Figure 4.5 flexible output matching

Instead, the evaluator uses **flexible sequence matching**, where:

- Expected output lines must appear in order
- Extra lines in actual output are ignored

This approach allows robust validation while tolerating unavoidable runtime noise.

If mismatches occur, the evaluator reports exactly which expected lines were not found, aiding student debugging.

4.7 Integration with Yaksh Grading System

Once output validation is complete, the evaluator returns:

- Pass/fail status
- Detailed error messages if applicable
- Assigned marks based on test case weight

```
# Filter out build system messages
filtered = []
for line in error_lines:
    if not any(x in line for x in ['ninja:', 'subcommand failed', 'ninja failed']):
        filtered.append(line)

return '\n'.join(filtered).strip()
```

Figure 4.6 Integration with Yaksh

Crucially, this evaluator integrates seamlessly with Yaksh's existing grading framework. No changes are required to Yaksh's core grading logic, database schema, or user interface.

ESP-IDF and QEMU support are implemented as an **extension**, preserving Yaksh's modular architecture while enabling embedded systems evaluation at scale.

Chapter 5: Arduino to ESP-IDF Build and QEMU Execution Script

This chapter explains the shell script used to convert Arduino .ino submissions into a runnable ESP-IDF firmware image and execute it using QEMU. The script acts as the **bridge between high-level Arduino code and low-level embedded execution**, enabling Yaksh to evaluate Arduino programs without physical hardware.

5.1 Purpose of the Execution Script

The *ino_to_running.sh* script performs the following end-to-end tasks:

1. Accepts an Arduino .ino file as input
2. Embeds the sketch into an ESP-IDF project structure
3. Builds the firmware using the ESP-IDF toolchain
4. Generates a flashable binary image
5. Executes the firmware using QEMU
6. Captures and filters runtime output for evaluation

This script encapsulates all ESP-IDF– and QEMU-specific complexity, allowing the Yaksh evaluator to invoke embedded execution through a single command.

5.2 Input Validation and File Preparation

The script begins by validating its usage and input:

- Ensures exactly one argument (the .ino file) is provided, Verifies that the input file exists

```
#!/usr/bin/env bash

# Usage: ./ino_to_running.sh <input.ino>
set -e

if [ $# -ne 1 ]; then
    echo "Usage: $0 <input.ino>"
    exit 1
fi

INO_FILE="$1"
TARGET_FILE="$(dirname "$0")/main/main.cpp"

if [ ! -f "$INO_FILE" ]; then
    echo "Input file $INO_FILE does not exist."
    exit 2
fi
```

Figure 5.1 Input validation script heading

The validated Arduino sketch is then injected into the ESP-IDF project's main.cpp file. During this step:

- Required ESP-IDF headers (Arduino.h, esp_log.h) are added
- A logging tag is defined for ESP-IDF compatibility
- The student's Arduino code is copied verbatim

```
{
    echo "//file: main.cpp"
    echo "#include \"Arduino.h\""
    echo "#include \"esp_log.h\""
    echo "static const char *TAG = \"APP\";"
    echo
    cat "$INO_FILE"
} > "$TARGET_FILE"
```

Figure 5.2 Appending required headers

This preserves student logic while embedding it into a structure compatible with ESP-IDF compilation.

5.3 ESP-IDF Environment Setup

ESP-IDF requires a properly configured build environment. The script automatically attempts to locate and source the *export.sh* file, which sets up required environment variables and toolchain paths.

```
# Import ESP-IDF environment
if [ -z "$IDF_PATH" ]; then
    if [ -f "$HOME/esp/esp-idf/export.sh" ]; then
        . "$HOME/esp/esp-idf/export.sh"
    elif [ -f "/opt/esp/idf/export.sh" ]; then
        . "/opt/esp/idf/export.sh"
    else
        echo "Could not find esp-idf export.sh. Please set up ESP-IDF environment."
        exit 3
    fi
fi
```

Figure 5.3 source the export.sh

Multiple common installation locations are checked to ensure portability across systems. If ESP-IDF is not found, the script exits with a clear error message, preventing ambiguous build failures.

5.4 Firmware Compilation Using ESP-IDF

Once the environment is configured, the script invokes:

```
# Build the project
idf.py build > build.log 2>&1
```

Figure 5.4 Building the program via ESP-IDF

This step:

- Compiles the Arduino-based application
- Builds the ESP-IDF runtime components
- Produces the application binary, bootloader, and partition table

All compilation output is redirected to a build.log file. This log is later parsed by the Yaksh evaluator to extract meaningful compiler errors and present them to students.

5.5 Flash Image Generation

ESP32 firmware consists of multiple binaries that must be placed at specific flash offsets. To prepare the firmware for QEMU execution, the script uses esptool.py to merge:

- Bootloader binary
- Partition table
- Application binary

These components are merged into a single **flashable binary image** with correct memory offsets.

```
# Now using esptools to merge the bootloader, partition table, and app binary to a flashable binary
BUILD_DIR="$(dirname "$0")/build"
APP_BIN="build/arduino_to_esp.bin"
BOOTLOADER_BIN="build/bootloader/bootloader.bin"
PARTITION_TABLE_BIN="build/partition_table/partition-table.bin"
FLASHABLE_BIN="build/flash_image.bin"

esptool.py --chip esp32 merge_bin -o "$FLASHABLE_BIN" \
    0x1000 "$BOOTLOADER_BIN" \
    0x8000 "$PARTITION_TABLE_BIN" \
    0x10000 "$APP_BIN" >> build.log 2>&1

# This is optional, but truncating the image into 4MB so it will run without any errors in QEMU
truncate -s 4M "$FLASHABLE_BIN"
echo "Created flashable binary at $FLASHABLE_BIN"


echo "Now emulating it via QEMU..."

# Finally running it in QEMU

set +e  # Don't exit on error, we need to handle QEMU cleanup properly

PATTERN="task_wdt: Task watchdog got triggered"
TIME_LIMIT=15  # Increased from 5 to 15 seconds for QEMU startup and execution

echo "[QEMU] Starting QEMU emulation with ${TIME_LIMIT}s timeout..."
echo "[QEMU] Flash image: $(pwd)/build/flash_image.bin"

# Clear output file before starting
> output.txt
```

Figure 5.5 Building a flashable binary image

To ensure compatibility with QEMU's ESP32 emulation, the final image is truncated to a fixed size (4 MB). This prevents flash size–related runtime errors during emulation.

5.6 QEMU-based Execution

After generating the flash image, the script launches QEMU to emulate an ESP32 system.

Key execution characteristics:

- QEMU runs in non-graphical mode

- The generated flash image is attached as a memory-mapped device
- Standard output and error streams are redirected to output.txt
- Execution is bounded by a configurable timeout to avoid infinite hangs

```
# Run QEMU with timeout, capturing both stdout and stderr
timeout "${TIME_LIMIT}s" qemu-system-xtensa -nographic -machine esp32 \
  -drive file=build/flash_image.bin,if=mtd,format=raw \
  > output.txt 2>&1 &

QEMU_PID=$!
echo "[QEMU] QEMU started with PID: $QEMU_PID"       You, 2 days ago • FI

# Wait a moment for QEMU to produce output
sleep 1

if [ -f output.txt ]; then
  OUTPUT_SIZE=$(wc -c < output.txt)
  echo "[QEMU] Output file size: $OUTPUT_SIZE bytes"
fi

# Monitor QEMU output in real-time
while read -r line; do
  if [[ "$line" == *"$PATTERN"* ]]; then
    echo "[QEMU] Watchdog pattern detected, stopping QEMU"
    kill "$QEMU_PID" 2>/dev/null
    break
  fi
done < <(tail --pid="$QEMU_PID" -Fn0 output.txt 2>/dev/null)

echo "[QEMU] Waiting for QEMU process to finish..."
wait "$QEMU_PID" 2>/dev/null

# Check final output
if [ -f output.txt ]; then
  FINAL_SIZE=$(wc -c < output.txt)
  echo "[QEMU] Final output size: $FINAL_SIZE bytes"
  if [ $FINAL_SIZE -eq 0 ]; then
    echo "[QEMU] WARNING: output.txt is empty!"
  fi
else
  echo "[QEMU] ERROR: output.txt not created!"
fi

set -e
```

QEMU is executed in the background, allowing the script to monitor its output in real time.

5.7 Watchdog Detection and Controlled Termination

Embedded programs running in QEMU may trigger watchdog warnings due to incomplete hardware emulation or idle loops. To handle this gracefully, the script continuously monitors QEMU output for known watchdog patterns.

When a watchdog trigger is detected:

- QEMU is terminated safely

- Execution ends early without crashing the evaluator
- Partial output remains available for analysis

This prevents unnecessary timeouts while still capturing meaningful program output.

5.8 Output Capture and Validation

Throughout execution:

- Output is written incrementally to output.txt
- File size is monitored to detect empty or stalled execution

Once QEMU terminates, the script performs final validation to ensure that output was generated. Missing or empty output files are flagged explicitly, enabling clear diagnostics.

5.9 Output Filtering

QEMU output contains a large amount of non-program-related data, including:

- UART buffer messages
- Bootloader logs
- System initialization traces

To isolate the student program's actual output, the script applies a filtering stage using sed. This removes known ESP-IDF and QEMU noise patterns and extracts only the relevant UART output.

The cleaned output is written to filtered_output.txt, which is later consumed by the Yaksh evaluator for grading.

5.10 Role of the Script in Yaksh Integration

This script serves as the **execution engine** for Arduino evaluation in Yaksh. By encapsulating:

- ESP-IDF setup
- Compilation
- Firmware packaging
- QEMU execution
- Output sanitization

it allows the Python-based evaluator to remain lightweight and platform-agnostic.

As a result, Yaksh can execute and assess embedded Arduino programs reliably, reproducibly, and without physical hardware, while maintaining compatibility with its existing evaluation framework.

Chapter 6: Static Analysis & Instrumentation of .ino Using Tree-sitter

This chapter describes the static analysis and source instrumentation mechanism used to inspect and validate Arduino programs before and during ESP-IDF–based execution. The approach leverages the Tree-sitter parsing framework to analyze Arduino (C++) source code structurally, enabling reliable detection of API usage patterns, pin configurations, and function calls without relying solely on runtime output.

6.1 Motivation for Static Analysis in Arduino Evaluation

While QEMU-based execution enables runtime validation of Arduino programs, certain aspects of embedded logic are more effectively verified through static analysis. These include:

- Whether specific Arduino APIs (e.g., digitalWrite, analogRead) are used
- Which pins are accessed by these APIs
- Whether required pins or functions appear in the student's code at all

Relying only on runtime behavior can be insufficient, especially when code paths are conditional or execution depends on timing. To address this, a static analysis layer was introduced to complement runtime evaluation.

6.2 Overview of Tree-sitter

Tree-sitter is an incremental parsing library that generates concrete syntax trees for source code. Unlike regex-based parsing, Tree-sitter provides:

- Language-aware parsing
- Precise node-level source locations
- Robust handling of nested and complex syntax

In this project, Tree-sitter is used with the C++ grammar to parse Arduino .ino files after conversion into valid C++ code.

6.3 Parser Initialization and Language Setup

The analysis pipeline begins by initializing the Tree-sitter parser with C++ language support:

- The C++ grammar is loaded using tree_sitter_cpp
- A Parser instance is created and configured with the C++ language
- The Arduino source code is read as raw bytes and parsed into a syntax tree

This syntax tree forms the foundation for all subsequent static analysis operations.

```
CPP_LANGUAGE = Language(tree_sitter_cpp.language())

parser = Parser()
parser.language = CPP_LANGUAGE
```

Figure 6.1 Analysis of C++ via tree sitter

6.4 Detection of Arduino API Function Calls

The syntax tree is traversed recursively to locate nodes of type call_expression. For each such node:

- The function name is extracted
- The name is compared against target Arduino APIs such as digitalWrite, digitalRead, and analogRead

```
# ------------------------------
# 2. Find digitalWrite() calls
# ------------------------------

def find_digital_write_calls(node, source, results):
    if node.type == "call_expression":
        fn = node.child_by_field_name("function")
        if fn:
            name = source[fn.start_byte:fn.end_byte].decode()
            if name == "digitalWrite":
                results.append(node)

    for child in node.children:
        find_digital_write_calls(child, source, results)
```

```
def find_function_calls(node, source, func_name, results):
    if node.type == "call_expression":
        fn = node.child_by_field_name("function")
        if fn:
            name = source[fn.start_byte:fn.end_byte].decode()
            if name == func_name:
                results.append(node)
    for child in node.children:
        find_function_calls(child, source, func_name, results)
```

Figure 6.2 finding digitalwrite() calls

Dedicated traversal functions are implemented to:

- Detect specific calls like digitalWrite()
- Detect arbitrary function calls specified dynamically (used later for CSV-based checks)

This approach ensures accurate detection even in deeply nested or complex code structures.

6.4.1 Extracting Function Arguments

Once a function call is identified, its argument list is extracted by:

- Locating the arguments child node
- Ignoring syntactic tokens such as parentheses and commas
- Collecting the actual argument expressions

For Arduino API calls, this typically results in:

- Pin number or pin variable
- Value or mode (e.g., HIGH, LOW)

- Enables API-level abstraction of ESP32 peripherals
- Integrates naturally with Yaksh's Python-based evaluation framework
- Simplifies validation of embedded logic and event-driven programs

```python
def extract_args(call_node, source):
    args = call_node.child_by_field_name("arguments")
    values = []

    for child in args.children:
        if child.type not in ("(", ")", ","):
            values.append(
                source[child.start_byte:child.end_byte].decode().strip()
            )

    return values  # [pin, value]


def extract_variable_assignments(tree, source):
    assignments = {}
    # Traverse the tree to find variable assignments
    def visit(node):
        if node.type == "declaration":
            # Look for int var = value;
            var_type = node.child_by_field_name("type")
            var_name = node.child_by_field_name("declarator")
            var_value = None
            if var_name:
                # Check for assignment
                for child in node.children:
                    if child.type == "init_declarator":
                        # Should have '=' and value
                        for subchild in child.children:
                            if subchild.type == "number_literal" or subchild.type == "identifier" or subchild.type == "field_identifier":
                                var_value = source[subchild.start_byte:subchild.end_byte].decode().strip()
                            elif subchild.type == "assignment_expression":
                                # assignment_expression: left, '=', right
                                right = subchild.child_by_field_name("right")
                                if right:
                                    var_value = source[right.start_byte:right.end_byte].decode().strip()
                var_name_str = source[var_name.start_byte:var_name.end_byte].decode().strip()
                if var_value:
                    assignments[var_name_str] = var_value
        for child in node.children:
            visit(child)
    visit(tree.root_node)
    return assignments
```

Figure 6.3 Extracting arguments

6.5 Variable Resolution and Assignment Tracking

Arduino code frequently uses variables to represent pin numbers instead of hardcoded literals. To correctly interpret such cases, the analysis includes a variable resolution step.

6.5.1 Extracting Variable Assignments

The syntax tree is traversed to identify variable declarations and initializations, such as:

int ledPin = 13;

For each declaration:

- The variable name is recorded
- The assigned value is extracted
- A mapping of variable → value is stored

This allows later resolution of pin numbers even when variables are used instead of literals.

6.6 Source Instrumentation for Runtime Observation

To enhance runtime observability, the analysis tool performs **source-level instrumentation**.

6.6.1 Instrumenting digitalWrite() Calls

Each detected digitalWrite() call is replaced with a logging statement:

```
# Instrument digitalWrite as before
calls = []
find_digital_write_calls(tree.root_node, source, calls)
edits = []
for call in calls:
    args = extract_args(call, source)
    pin, val = args[:2] if len(args) >= 2 else (None, None)
    # Resolve pin if it's a variable
    if pin in var_assignments:
        pin_resolved = var_assignments[pin]
    else:
        pin_resolved = pin
    log_stmt = (
        f'ESP_LOGI(TAG, "PIN {pin_resolved}, {val}");'
    ).encode()
    edits.append(
        (call.start_byte, call.end_byte, log_stmt)
    )
instrumented = bytearray(source)
```

Figure 6.4 convering digitalWrite into ESP_LOGI for better debugging

Key characteristics:

- The original call location is replaced with an ESP-IDF logging statement
- Pin variables are resolved to their assigned values when possible
- Instrumentation is applied using byte-level edits based on Tree-sitter node offsets

This enables precise runtime logging of GPIO activity during QEMU execution without modifying student logic manually.

6.7 Pin Validation Using CSV-Based Specifications

In addition to instrumentation, the tool supports **static validation against instructor-defined requirements** using a CSV file.

6.7.1 CSV Specification Format

The CSV file specifies expected API usage in the form:

function_name, pin_number

Example:

digitalWrite,13

analogRead,A0

6.7.2 Matching Function–Pin Usage

For each CSV entry:

- All calls to the specified function are located
- The pins used in those calls are extracted and resolved

- The actual pins used are compared against the expected pin

Three outcomes are reported:

- **FOUND** – function is used with the expected pin
- **PRESENT** – function is used, but with different pins
- **MISSING** – function is not used at all

This enables enforcement of assignment constraints such as “use pin 13” or “read from A0”.

```
# Check CSV for function/pin presence
with open(csv_file, newline='') as csvfile:
    reader = csv.reader(csvfile)
    # Skip header row (pin_type, pin_number)
    next(reader, None)
    for row in reader:
        if len(row) < 2:
            continue
        func, pin = row[0].strip(), row[1].strip()
        # Try to convert pin to int if possible, else keep as string
        try:
            pin_val = int(pin)
        except ValueError:
            pin_val = pin
        func_calls = []
        find_function_calls(tree.root_node, source, func, func_calls)
        pins_used = set()
        for call in func_calls:
            args = extract_args(call, source)
            arg_pin = args[0] if args else None
            # Resolve variable if needed
            if arg_pin in var_assignments:
                arg_pin_val = var_assignments[arg_pin]
                try:
                    arg_pin_val = int(arg_pin_val)
                except ValueError:
                    pass
            else:
                try:
                    arg_pin_val = int(arg_pin)
                except (ValueError, TypeError):
                    arg_pin_val = arg_pin
            pins_used.add(str(arg_pin_val))
        if str(pin_val) in pins_used:
            print(f"[FOUND] {func}({pin}) is present in the code.")
        elif pins_used:
            print(f"[PRESENT] {func} is used, but with different pin(s): {', '.join(sorted(pins_used))}")
        else:
            print(f"[MISSING] {func}({pin}) is NOT present in the code.")
```

Figure 6.5 Checking for PIN correctness

6.8 Pin Type Validation

The analysis includes predefined pin sets:

- Analog pins (A0–A5)
- Digital pins (D0–D13)

Based on the API being used:

- analogRead() is validated against analog pin sets
- digitalRead() and digitalWrite() are validated against digital pin sets

This prevents logically incorrect pin usage from passing evaluation silently.

6.9 Integration with the ESP-IDF Execution Pipeline

This static analysis and instrumentation stage integrates seamlessly with the ESP-IDF + QEMU pipeline:

1. Student .ino file is converted to C++
2. Tree-sitter analysis and instrumentation are applied
3. Instrumented source is compiled using ESP-IDF
4. QEMU execution produces structured GPIO logs
5. Yaksh evaluator validates behavior using runtime output and static checks

By combining **static analysis** and **dynamic execution**, the system achieves higher reliability and pedagogical correctness.

6.10 Advantages of the Tree-sitter–Based Approach

This approach offers several benefits:

- Robust parsing compared to regex-based methods
- Accurate detection of API usage and arguments
- Variable-aware pin resolution
- Safe and precise source instrumentation
- Improved feedback for students and instructors

6.11 Summary

This chapter presented the Tree-sitter–based static analysis and instrumentation framework used to analyze Arduino programs in Yaksh. By augmenting QEMU-based execution with structural code inspection and logging, the system enables precise verification of API usage, pin configuration, and GPIO behavior without relying solely on runtime output.

The next chapter builds on this foundation by describing how similar principles are applied to MicroPython evaluation using hook-based and logic-level simulation techniques.

Chapter 7: Evaluation Outcomes with Examples

This chapter presents the observable outcomes of the Arduino program evaluation pipeline after static analysis, instrumentation, compilation, and runtime execution. It explains how Yaksh classifies execution results and illustrates the behaviour using representative example cases.

The evaluation framework produces deterministic and user-readable feedback by categorizing each submission into one of three result states:

1. Successful Execution
2. Build / Compilation Error
3. Output Mismatch

7.1 Evaluation Result States

7.1.1 Successful Execution

A submission is marked as **successful** when:

- The instrumented Arduino program compiles successfully using ESP-IDF
- QEMU execution completes within the allowed timeout
- The filtered runtime output matches the expected output defined by the test case

In this case, Yaksh displays a success message and awards full marks for the question. The student is informed that the program has been evaluated correctly, and no further action is required.

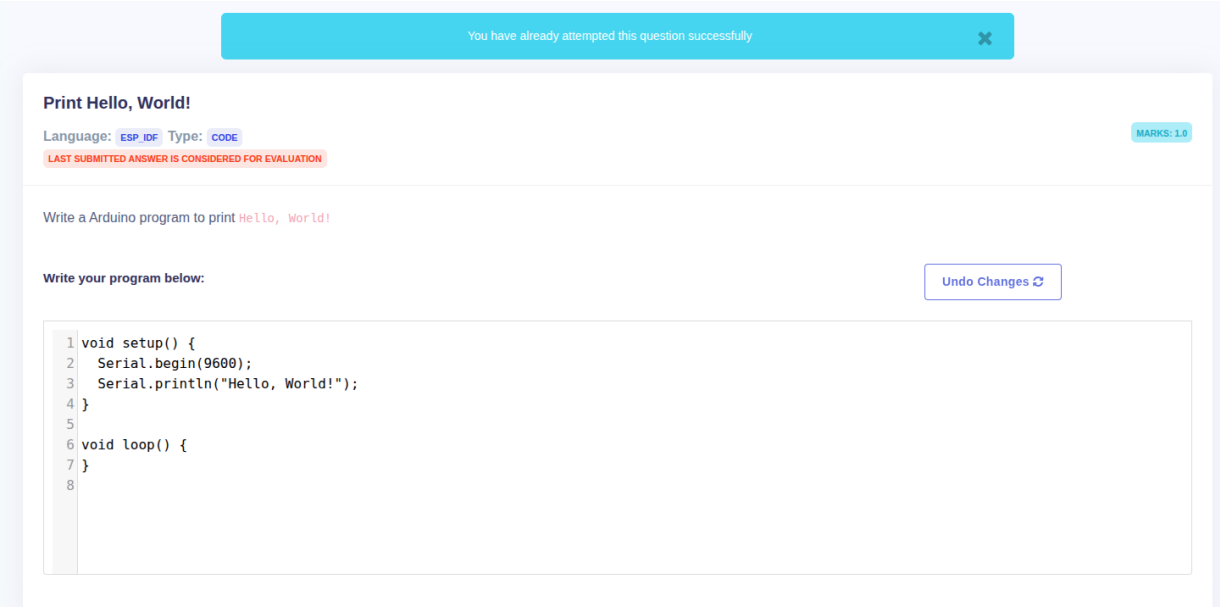


Figure 6 Successful Build

7.1.2 Build / Compilation Error

If the submission fails during the compilation stage, the evaluation terminates early and the result is classified as a **build error**.

Typical causes include:

- Syntax errors in the Arduino code
- Missing semicolons or incorrect function calls
- Unsupported or incorrectly used APIs

In such cases:

- The ESP-IDF build system reports compiler errors
- Yaksh extracts and displays relevant error messages
- Runtime execution is skipped entirely

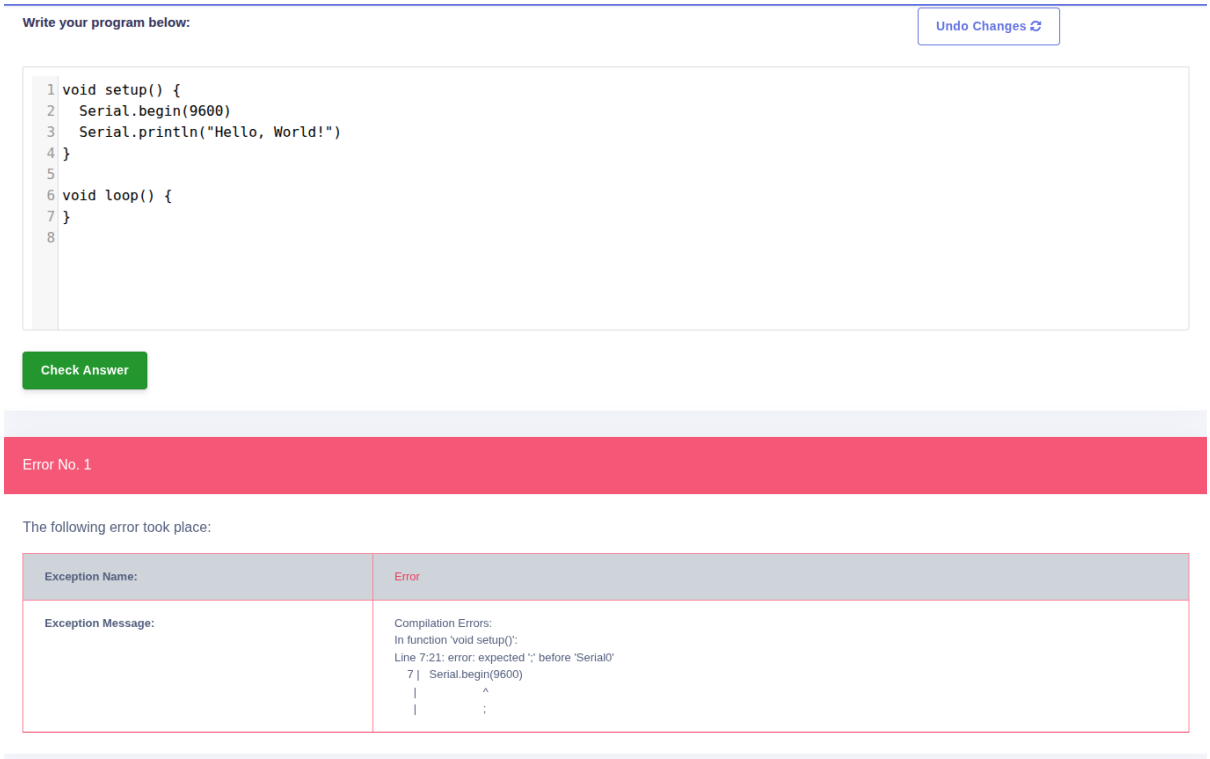


Figure 7 Error on compiling

This feedback allows students to correct compilation issues before reattempting the submission.

7.1.3 Output Mismatch

If compilation and execution succeed, but the program output does not match the expected output, the result is classified as an **output mismatch**.

This occurs when:

- The program prints incorrect text
- Output formatting differs from expectations
- Required output lines are missing or altered

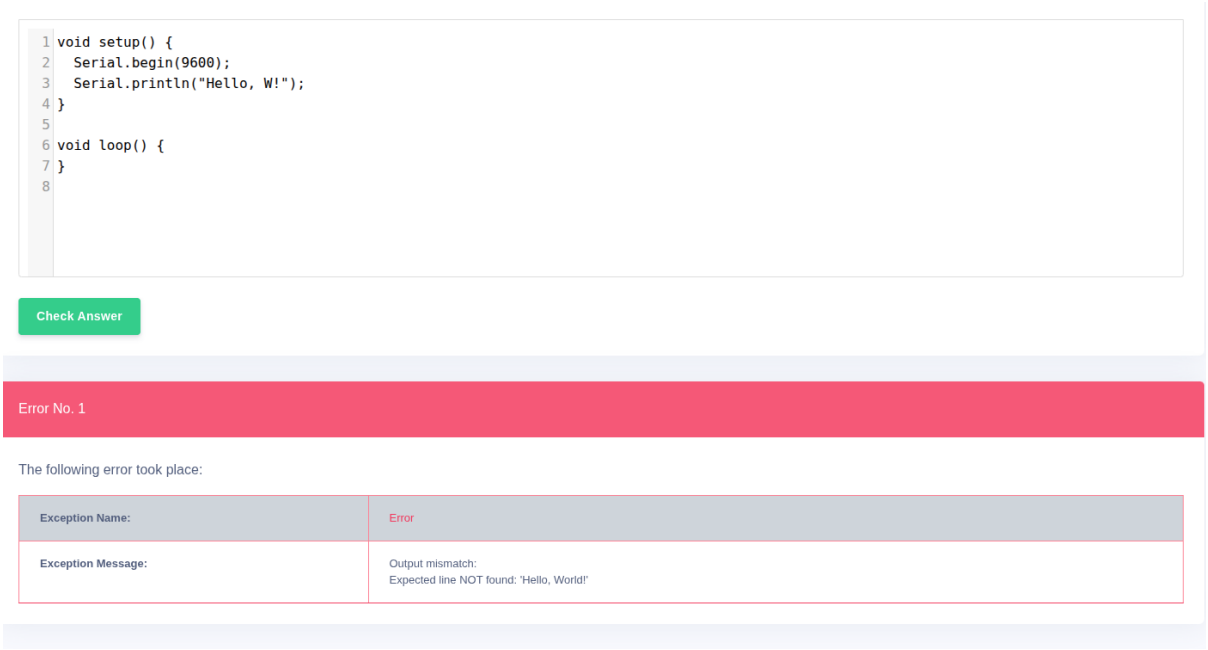


Figure 8 Error on output mismatch

Yaksh reports the exact expected output line that was not found, enabling precise debugging by the student.

7.2 Example Evaluation Scenarios

To demonstrate the behaviour of the evaluation framework, three representative cases—C1, C2, and C3—are discussed below.

7.2.1 Case C1: Correct API Usage and Expected Behavior

The student program correctly uses the required Arduino APIs and accesses the expected pin. Static analysis confirms correct API–pin mapping, and runtime execution produces the expected output.

Result

- Static analysis, Compilation: PASS
- Runtime, Output execution: PASS

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

void loop() {
  int sensorValue = analogRead(A0);
  Serial.print("Sensor Value: ");
  Serial.println(sensorValue);
  delay(500);
}
```

Figure 9 Example Student program without errors

```
❯ python main.py testing.ino out.ino pin_mngmt.csv
[OK] Written instrumented file: out.ino
[FOUND] analogRead(A0) is present in the code.
```

Figure 10 Expected output from the above porgram

7.2.2 Case C2: Correct API with Incorrect Pin

The student program uses the correct Arduino API but accesses a different pin than the one specified by the instructor.

Static Analysis Result

- Function detected correctly
- Pin mismatch detected and reported as **PRESENT with different pin**

Runtime Execution

- Program compiles and executes successfully
- Output may appear correct, but violates assignment constraints

Evaluation Outcome: Logical Constraint Violation (Reported via Static Analysis)

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

void loop() {
  int sensorValue = analogRead(A1);
  Serial.print("Sensor Value: ");
  Serial.println(sensorValue);
  delay(500);
}
```

Figure 11 The previous program with an induced error

```
→ python main.py testing.ino out.ino pin_mngmt.csv
[OK] Written instrumented file: out.ino
[PRESENT] analogRead is used, but with different pin(s): A1
```

Figure 12 The output for the above

7.2.3 Case C3: Missing Required API Usage

The student program does not use the required Arduino API (e.g., analogRead()), even though other logic executes correctly.

Static Analysis Result

- Required function–pin pair not found
- Reported as **MISSING**

Runtime Execution

- Program may compile and execute
- Fails logical requirements of the assignment

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

void loop() {
  int sensorValue = digitalRead(A0);
  Serial.print("Sensor Value: ");
  Serial.println(sensorValue);
  delay(500);
}
```

Figure 13 Changed the analogRead into digitalRead

```
→ python main.py testing.ino out.ino pin_mngmt.csv
[OK] Written instrumented file: out.ino
[MISSING] analogRead(A0) is NOT present in the code.
```

Figure 14Output of the Fig 27

7.3 Combined Interpretation of Results

By combining:

- Static analysis results (API usage, pin correctness)
- Compilation diagnostics
- Runtime output validation

the evaluation framework ensures that:

- Students write syntactically correct code
- Required hardware APIs are used correctly
- Program behaviour matches assignment expectations

This multi-layered evaluation approach provides more meaningful feedback than traditional output-only grading, especially for embedded systems assignments.

7.4 Summary

This chapter demonstrated how Yaksh classifies and reports evaluation outcomes using real execution and static analysis results. Through representative cases, it was shown that the system can accurately distinguish between successful executions, build errors, output mismatches, and logical constraint violations.

The next chapter extends these principles to MicroPython-based evaluation, where similar result classification is achieved through logic-level simulation and hook-based validation.

Chapter 8: MicroPython Evaluation Strategy in Yaksh

This chapter describes the evolution of MicroPython support in Yaksh, beginning with early experiments using QEMU-based ESP32 execution and progressing toward a logic-level simulation approach based on the MicroPython Unix port. It explains the design decisions, initial implementation, and limitations that motivated further refinement of the evaluation framework.

8.1 Observations from QEMU-Based ESP32 Execution for MicroPython

During initial experimentation, ESP32 applications were built using ESP-IDF and executed in QEMU using merged flash images. While this approach enabled basic execution of firmware images, several practical challenges were observed when applying it to MicroPython-based programs:

- Single application binaries were insufficient for successful QEMU execution
- QEMU required a fully structured flash image consisting of a bootloader, partition table, and application binary
- Monitoring and validating peripheral behaviour such as GPIO, ADC, and timers proved difficult and error-prone

These limitations highlighted that system-level hardware emulation was not well-suited for fine-grained, automated evaluation of MicroPython programs. As a result, an alternative approach was explored.

8.2 Introduction to the MicroPython Evaluation Track

To overcome the challenges of hardware-level emulation, the MicroPython evaluation strategy shifted away from ESP32 system emulation toward **logic-level execution and simulation**.

Instead of emulating the ESP32 hardware itself, the focus moved to:

- Executing MicroPython programs directly on the host system
- Simulating ESP32-specific APIs such as Pin, ADC, and Timer
- Ensuring deterministic and repeatable execution suitable for automated grading

This approach allowed greater control over peripheral behavior and simplified observation, validation, and grading of embedded logic.

8.3 Rationale for Using the MicroPython Unix Port

The **MicroPython Unix port** was selected as the foundation for MicroPython evaluation due to several key advantages:

- Provides fast and deterministic execution

This design decision marked a clear transition from **hardware emulation** to **logic-level simulation**, forming the basis for the MicroPython evaluation framework described in subsequent sections.

8.4 MicroPython Evaluation Using *QemuStdIOEvaluator*

8.4.1 Introduction

This section describes the design and working of the QemuStdIOEvaluator, which represents the **initial MicroPython evaluation strategy** implemented in Yaksh. The evaluator executes student-submitted MicroPython programs in a controlled environment and validates their behavior using simulated ESP32 abstractions.

The evaluator integrates with Yaksh's existing evaluation framework and relies on a custom MicroPython runner along with file-based abstractions for ESP32-specific modules.

8.4.2 Purpose of QemuStdIOEvaluator

The primary objectives of the QemuStdIOEvaluator are to:

- Execute student-submitted MicroPython code
- Provide simulated ESP32 hardware APIs (machine, time)
- Capture program output deterministically
- Compare execution output against expected results
- Generate detailed diagnostic information for debugging

The evaluator follows a **script-based execution model**, where the entire student program is executed, and grading is performed based on runtime output.

8.4.3 Class Structure

The evaluator is implemented as:

class QemuStdIOEvaluator(StdIOEvaluator)

By extending StdIOEvaluator, the class inherits Yaksh's existing mechanisms for:

- Standard input/output handling
- Output comparison
- Error reporting

This reuse minimizes changes to Yaksh's core architecture.

8.4.4 Initialization and Metadata Handling

- **Constructor (__init__)**

During initialization, the evaluator configures its internal state using metadata and test case information provided by Yaksh.

Key metadata fields include:

- user_answer – student-submitted source code
- file_paths – optional supporting files
- runner_path – path to the MicroPython execution script

- runner_args – optional runner arguments
- partial_grading – enables partial credit

Test case data includes:

- expected_input
- expected_output
- weight
- hidden

8.4.5 Compilation and Environment Preparation

Writing the Student Submission, The student's MicroPython code is written to a temporary file named:

submission.py

This file serves as the executable script for the MicroPython interpreter.

Injecting ESP32 Abstractions to simulate ESP32-specific behaviour, predefined abstraction modules are copied into the working directory:

- machine.py – simulates ESP32 hardware APIs
- time.py – overrides time-related functions

These files ensure that any import machine or import time statements in the student code resolve to the simulated implementations instead of host system modules.

8.4.6 Execution Phase

Runner Invocation, the student program is executed using a Python-based MicroPython runner:

python3 pyauto.py submission.py --output qemu_output.txt

The runner is responsible for:

- Launching the MicroPython interpreter
- Executing the student script
- Capturing standard output and errors
- Writing execution results to a file

8.4.7 Process Isolation and Control

Execution is launched using subprocess.Popen with:

- A separate process group
- Controlled environment variables
- Explicit stdout and stderr capture

This enables Yaksh to terminate runaway processes and enforce execution constraints safely.

8.4.8 Output Collection and Error Handling

After execution completes:

- Output is read from qemu_output.txt
- Runtime errors are detected by scanning for tracebacks and error strings
- Errors are normalized using Yaksh's output formatting utilities

If the process exits with a non-zero return code, execution is marked as failed and detailed diagnostics are generated.

8.4.9 Output Comparison and Grading

For successful executions, grading is performed by comparing:

- Actual program output
- Expected output defined in the test case

Output comparison is handled using Yaksh's compare_outputs utility. Partial grading is supported when enabled via metadata.

8.4.10 Limitations of the Script-Based MicroPython Evaluation Approach

While QemuStdIOEvaluator enabled early MicroPython support in Yaksh, several limitations were identified:

- Heavy reliance on printed output for grading
- Difficulty in evaluating event-driven logic
- Susceptibility to output manipulation
- Complex behavior of timers and callbacks

These limitations motivated the transition to a more robust evaluation strategy.

8.5 Summary

This chapter described the initial MicroPython evaluation framework based on `QemuStdIOEvaluator`. While functional, the approach relied primarily on output-based grading and was constrained in handling event-driven and asynchronous logic.

The next chapter introduces a **hook-based evaluation strategy**, which overcomes these limitations by shifting toward logic-level, function-based evaluation of MicroPython programs.

Chapter 9: Need for a Hook-Based Evaluation Framework

The initial evaluation approach for MicroPython programs relied on executing student code and analyzing the printed output generated by a simulated hardware environment. Instead of interacting with real hardware, a custom machine.py module was used to emulate ESP32 peripherals such as GPIO, PWM, ADC, and timers. These simulated components produced structured log messages (via print statements), which served as observable indicators of program behavior.

For given Input value(s): # no input

Line No.	Expected Output	User output	Status
1	[SIM][Pin] GPIO34 initialized	[SIM][Pin] GPIO3 initialized	✖
2	[SIM][ADC] GPIO34 initialized	[SIM][ADC] GPIO3 initialized	✖
3	[SIM][Pin] GPIO2 initialized	[SIM][Pin] GPIO2 initialized	✔
4	[SIM][ADC] GPIO34 read -> {value}	[SIM][ADC] GPIO3 read -> 20	✖
5	[SIM][Pin] GPIO2 -> HIGH	[SIM][Pin] GPIO2 -> HIGH	✔

Error: Incorrect Answer: Line number(s) 1, 2, 4 did not match. Abs GPIO2, GPIO3

While this method enabled basic validation of hardware-related logic, it remained inherently limited. The evaluator could only infer correctness by matching printed traces rather than directly inspecting internal state or function calls. As a result, correctness depended on consistent print behavior, and more complex interactions such as verifying proper sequencing, parameter correctness, or conditional execution were difficult to assess reliably.

This limitation motivated the transition toward a more structured hook-based evaluation model, where program behavior could be analyzed through controlled interception points rather than relying solely on textual output.

Chapter 10: Architecture of the Hook-Based Evaluation System

The hook-based evaluation system is designed to assess MicroPython programs without relying on physical hardware or real-time execution. Instead of executing code on an actual microcontroller, the system simulates the MicroPython runtime environment and observes how the student's program interacts with it.

This is achieved by dynamically injecting custom implementations of hardware-related modules and evaluating the internal state changes they produce.

10.1 Execution Model Overview

During evaluation, the student's program is executed using the standard Python interpreter under controlled conditions. Before execution begins, the evaluator prepares a custom runtime environment that replaces selected MicroPython modules with simulated equivalents.

The execution flow is as follows:

1. The evaluator loads the student's submission.
2. Custom implementations of **machine** and **time** are imported by the evaluator.
3. These modules are registered into Python's module system (**sys.modules**).
4. The student's code is executed normally.
5. All calls to import machine or import time inside the student code resolve to the injected modules.
6. Internal state changes inside these modules are later inspected by the hook logic to determine correctness.

This approach ensures that the student code runs unmodified while still allowing complete observation of hardware-related behavior.

The simulated implementations of machine and time used by the hook evaluation system are stored within the project's internal hook module directory.(~/online_test/yaksh/micropy_hook)

10.2 Runtime Module Injection Mechanism

Instead of modifying the student's source files or overwriting system libraries, the evaluator performs **runtime module injection**.

During initialization, the hook evaluator imports its own implementations of machine and time and explicitly registers them in Python's module cache:

sys.modules["machine"] = machine

sys.modules["time"] = time

Python resolves imports by first checking this module cache. As a result, when the student code executes statements such as:

import machine

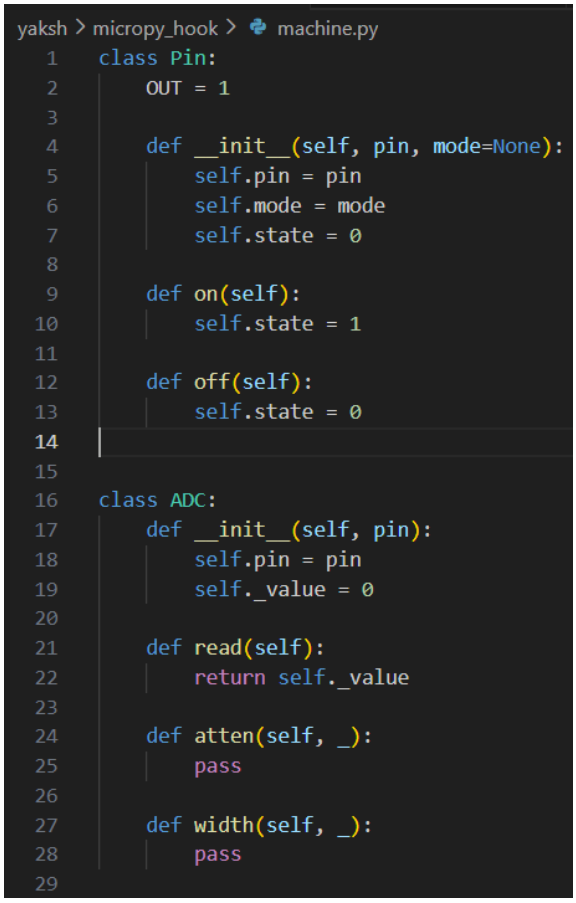
```
import time
```

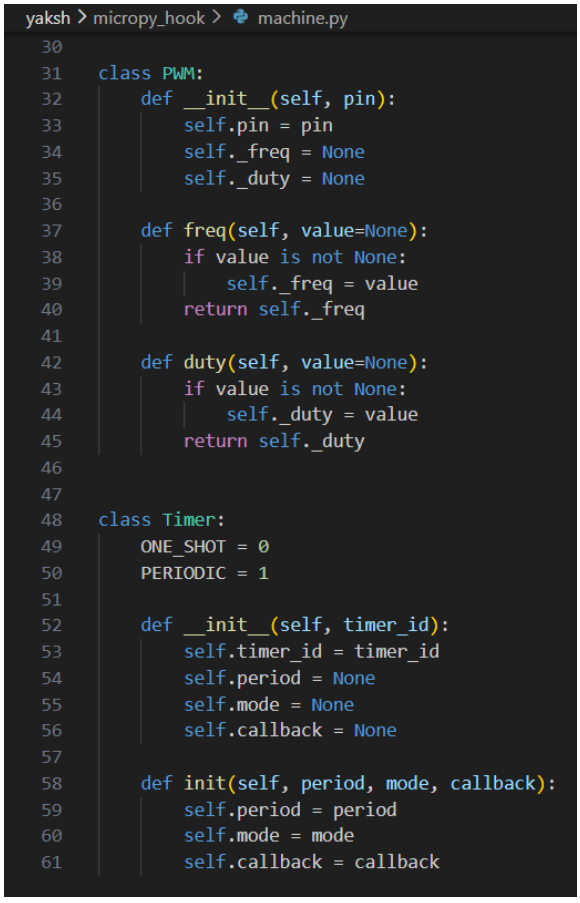
the interpreter returns the injected modules rather than the standard implementations. This redirection is completely transparent to the student program and requires no code modification.

10.3 Simulated Hardware Module (machine.py)

The custom machine.py module provides lightweight, state-based implementations of commonly used MicroPython classes, including:

- Pin
- ADC
- PWM
- Timer





Each class captures internal state rather than performing real hardware operations. For example:

- Pin records pin number and logical state (on / off)
- PWM stores frequency and duty cycle values
- ADC returns deterministic readings for predictable evaluation
- Timer stores configuration parameters without scheduling real callbacks

This design allows the evaluator to verify *what actions were requested* rather than depending on physical side effects.

10.4 Simulated Time Module (time.py)

The custom time module replaces real timing behavior with deterministic, non-blocking logic.

Functions such as sleep(), sleep_ms(), and sleep_us() simply record invocation data without introducing delays. Time-query functions like ticks_ms() and ticks_us() return predictable values based on host time.

This ensures that time-dependent student code executes instantly while still allowing the evaluator to confirm correct usage of timing APIs.

yaksh > micropy_hook > time.py

```python
import time as _host_time

# ----------------------------
# Internal tracking variables
# ----------------------------

_last_sleep = None
_last_sleep_ms = None
_last_sleep_us = None
_last_ticks_add_args = None
_last_ticks_diff_args = None


# ----------------------------
# Sleep functions
# ----------------------------

def sleep(seconds):
    global _last_sleep
    _last_sleep = seconds


def sleep_ms(ms):
    global _last_sleep_ms
    _last_sleep_ms = ms


def sleep_us(us):
    global _last_sleep_us
    _last_sleep_us = us
```

yaksh > micropy_hook > time.py

```python
# ----------------------------
# Time tracking
# ----------------------------

_start_time = _host_time.time()


def time():
    return int(_host_time.time())


def ticks_ms():
    return int((_host_time.time() - _start_time) * 1000)


def ticks_us():
    return int((_host_time.time() - _start_time) * 1_000_000)


def ticks_cpu():
    return ticks_us()


def ticks_add(ticks, delta):
    global _last_ticks_add_args
    _last_ticks_add_args = (ticks, delta)
    return ticks + delta


def ticks_diff(ticks1, ticks2):
    global _last_ticks_diff_args
    _last_ticks_diff_args = (ticks1, ticks2)
    return ticks1 - ticks2
```

Chapter 11: Hook Evaluator Execution Flow and Integration Logic

The hook-based evaluation mechanism builds on an existing evaluation framework in which the check_answer() function is already defined and executed by the system. Rather than modifying this structure, the MicroPython integration extends it by injecting a simulated hardware environment before the evaluation logic is invoked.

This design ensures backward compatibility with existing evaluators while enabling MicroPython-specific behavior to be tested using the same evaluation pipeline.

11.1 Role of the Hook Evaluator

The HookEvaluator class is responsible for coordinating the complete evaluation process. Its responsibilities include:

- Preparing the execution environment
- Injecting simulated MicroPython modules
- Executing the hook test code
- Capturing results and errors
- Returning standardized evaluation output

The actual grading logic remains inside the existing check_answer() function provided in the hook test case. The evaluator does **not** replace or modify this logic it only prepares the environment in which it runs.

11.2 Injection of MicroPython Runtime Components

Before executing the hook test, the evaluator explicitly injects simulated MicroPython modules by calling the internal method _inject_micropy_hook().

```
def _inject_micropy_hook(self):

    current_dir = os.path.dirname(os.path.abspath(__file__))
    base_dir = current_dir

    if base_dir not in sys.path:
        sys.path.insert(0, base_dir)

    import micropy_hook.machine as machine
    import micropy_hook.time as time

    # Register under the expected MicroPython name
    sys.modules["time"] = time
    sys.modules["machine"] = machine
```

This method performs the following actions:

1. Determines the directory containing the MicroPython simulation modules.
2. Ensures this directory is available in sys.path.

3. Imports the custom implementations of machine and time.
4. Registers these modules in sys.modules under their standard names.

sys.modules["machine"] = machine

sys.modules["time"] = time

As a result, any import machine or import time statement executed afterward, either inside the student's submission or within the hook test resolves to the simulated implementations rather than the real Python or MicroPython modules.

This mechanism allows the evaluator to transparently intercept all hardware-related calls without modifying student code or the hook test logic itself.

11.3 Execution Flow of check_answer()

Once the environment is prepared, the evaluator proceeds as follows:

1. The hook test code is compiled using Python's compile() function.
2. The compiled code is executed in an isolated namespace.
3. The check_answer() function is retrieved from that namespace.
4. check_answer() is invoked with the student submission as input.
5. The function evaluates correctness using the injected simulation modules.
6. A tuple (success, error_message, mark_fraction) is returned to the evaluator.

Chapter 12: Writing Hook Test Cases

With the execution environment and injection mechanism in place, the final component of the system is the hook test case itself. Hook test cases define the logic used to verify whether a student's submission behaves as expected under the simulated MicroPython environment.

Each hook test is written as a standard Python script and follows a fixed structural pattern that integrates seamlessly with the evaluator.

12.1 Structure of a Hook Test Case

Every hook test file must define a function named check_answer. This function serves as the single entry point for evaluation and is invoked by the hook evaluator after environment setup is complete.

The expected function signature is:

```
def check_answer(user_answer):
    ...
    return success, error_message, mark_fraction
```

Where:

- user_answer contains the student's submitted code or reference to it.
- success is a boolean indicating pass or fail.
- error_message is a descriptive string shown when evaluation fails.
- mark_fraction is a float representing the awarded score.

This standardized interface allows the evaluator to remain generic while supporting a wide variety of problem types.

12.2 Accessing Simulated Hardware Inside Tests

Because the MicroPython simulation modules (machine and time) are injected before execution, they can be used directly inside the hook test without any special imports or setup.

For example, a hook test can safely perform operations such as:

- Instantiating a Pin or PWM object
- Reading or validating duty cycle values
- Checking whether a pin was configured correctly
- Verifying that a timing function was invoked

All such operations interact with the simulated environment rather than real hardware.

This allows the test logic to focus purely on *behavioral correctness* rather than environment management.

12.3 Typical Validation Pattern

A common testing pattern inside check_answer() includes:

1. Executing the student's code.
2. Accessing internal state from simulated objects.
3. Comparing observed values against expected behavior.
4. Returning a structured result.

For example, a test might verify that:

- A PWM object was created on the correct pin.
- The frequency was set to a required value.
- The duty cycle was updated as expected.

Because the simulated modules retain internal state, such checks can be performed deterministically and without side effects.

12.4 Error Reporting and Partial Grading

The hook test can return detailed feedback when evaluation fails. Instead of terminating execution, it can provide:

- A descriptive error message explaining what was incorrect.
- A partial score if only some conditions were satisfied.

This allows for nuanced grading and clearer feedback to learners, especially in multi-step programming tasks.

Chapter 13: Execution and Validation Flow Inside a Hook Test

Once the runtime environment and simulated hardware are prepared, the actual evaluation of the student's solution begins. This process is implemented entirely inside the check_answer() function and is designed to execute the student's code safely while allowing fine-grained inspection of its behavior.

13.1 Executing Student Code in a Controlled Environment

The student's submission is provided to the evaluator as raw source code (a string). Instead of running this code as a standalone program, it is executed inside a controlled dictionary-based environment using Python's exec() function.

```
exec_env = {}

exec(user_code, exec_env)
```

This approach ensures:

- All variables, functions, and objects created by the student are stored inside exec_env.
- The global Python environment remains unaffected.
- The evaluator can directly inspect variables created by the student after execution.

This technique is central to the hook-based design, as it allows the evaluator to *observe program behavior without modifying student code*.

13.2 Accessing Objects Created by the Student

After execution, the evaluator inspects exec_env to verify that required objects were created correctly.

For example, if the assignment requires the student to create an ADC object:

```
if "adc" not in exec_env:
    return False, "ADC object not created", 0.0
```

Because the simulated machine.ADC class was injected earlier, any ADC object created by the student is automatically linked to the simulated environment. This allows the evaluator to examine its internal state safely.

13.3 Validating Object Configuration

Once the object is located, the evaluator verifies that it was initialized correctly. This includes checks such as:

- Whether the correct pin was used
- Whether the object type matches the expected class
- Whether required attributes are present

For example, validating the ADC pin:

```
if not isinstance(adc, machine.ADC):
    return False, "ADC must be created using machine.ADC", 0.0
```

and later:

```
if pin_num != 34:
    return False, "ADC must be connected to Pin 34", 0.0
```

These checks ensure that the student followed the intended hardware configuration, not just produced the correct numerical output.

13.4 Validating Functional Logic

Beyond structural checks, the hook test also validates functional correctness.

In the example shown, the evaluator computes the *expected* temperature using a reference formula and compares it against the value returned by the student's function:

```
expected = expected_temp(adc_val)
student_temp = read_temperature()
```

By injecting different ADC values and observing the resulting output, the evaluator verifies that the student's logic behaves correctly across multiple cases not just a single hardcoded input.

Chapter 14: Writing Hook Test Cases: Design Principles and Execution Model

Before examining concrete examples, it is important to understand a key design principle that applies to **all hook test cases** used in this evaluation framework.

14.1 Why Objects Must Be Defined at the Global Level

In the hook-based evaluation system, the student's code is executed inside a controlled environment using Python's exec() function. This execution happens *multiple times* during the evaluation process — for example, when validating different inputs or re-running checks.

Each execution creates a fresh local execution context. As a result:

- Any objects created **inside** a function are recreated on every run.
- Their internal state is lost between successive evaluations.
- This makes it difficult to reliably validate behavior across multiple test steps.

To avoid this issue, all hardware-related objects (such as Pin, ADC, PWM, and Timer) must be defined at the global scope in the student's code.

By defining these objects globally:

- Their state persists across multiple evaluation steps.
- The hook code can repeatedly access and inspect them.
- The test logic remains simple and deterministic.

This design choice significantly reduces complexity in the hook code and avoids the need to repeatedly reinitialize or patch objects during validation.

14.2 How Hook Tests Interact with Global Objects

Because the simulated machine module is injected before execution, any global objects created using it become accessible to the hook logic through the execution environment.

For example, if the student writes:

adc = machine.ADC(34)

That adc object becomes part of the execution environment and can later be retrieved by the hook test using:

adc = exec_env["adc"]

This allows the evaluator to:

- Inspect configuration values
- Modify internal state for testing
- Re-run logic under different simulated conditions

without re-executing or rewriting student code.

Chapter 15: Example: Temperature-Based LED Control

This section demonstrates how a complete hook test is written and how it validates student logic using a simulated MicroPython environment. The goal of this example is to verify whether a student correctly reads temperature data from an LM35 sensor and controls an LED based on a threshold.

15.1 Problem Summary

The student is required to:

- Read temperature data using an ADC connected to GPIO 34
- Convert the ADC value into temperature (°C)
- Define a constant threshold (TEMP_THRESHOLD = 20)
- Turn an LED ON when temperature ≥ 20°C
- Turn the LED OFF when temperature < 20°C

To make this test verifiable, all required objects (adc, led, and TEMP_THRESHOLD) must be defined **at the global level**.

```python
import textwrap
def check_answer(user_answer):
    EXPECTED_THRESHOLD = 20
    VREF = 3.3
    ADC_MAX = 4095

    # Execute student code
    exec_env = {}
    try:
        exec(textwrap.dedent(user_answer), exec_env)
    except Exception as e:
        return False, f"Runtime error: {e}", 0.0

    # --- VALIDATIONS ---
    if "TEMP_THRESHOLD" not in exec_env:
        return False, "TEMP_THRESHOLD not defined", 0.0
    if exec_env["TEMP_THRESHOLD"] != EXPECTED_THRESHOLD:
        return False, f"TEMP_THRESHOLD must be {EXPECTED_THRESHOLD}", 0.0
    if "adc" not in exec_env:
        return False, "ADC object not created", 0.0
    adc = exec_env["adc"]
    if adc.pin.pin != 34:
        return False, "ADC must be connected to Pin 34", 0.0
    if "led" not in exec_env:
        return False, "LED object not created", 0.0
    led = exec_env["led"]
    if led.pin != 2:
        return False, "LED must be connected to Pin 2", 0.0
    if "update_led" not in exec_env:
        return False, "update_led() function missing", 0.0

    # BELOW threshold
    adc._value = int(((EXPECTED_THRESHOLD - 1) * 0.01 / VREF) * ADC_MAX)
    exec_env["update_led"]()
    if led.state != 0:
        return False, "LED should be OFF below threshold", 0.0
```

```python

    # ABOVE threshold
    adc._value = int(((EXPECTED_THRESHOLD + 1) * 0.01 / VREF) * ADC_MAX)
    exec_env["update_led"]()
    if led.state != 1:
        return False, "LED should be ON above threshold", 0.0

    return True, "Correct implementation", 1.0
```

15.2 Why Global Objects Are Required

In the hook-based evaluation model, the student's code is executed **once**, and the evaluator then performs **multiple validation steps** by interacting with the objects created during that execution.

Unlike a traditional program that runs from start to finish only once, the hook system must repeatedly **inspect and manipulate program state** in order to verify correctness under different conditions. This design makes it essential that key objects such as adc, led, or pwm are created at the **global scope**.

If such objects are defined *inside a function*, they are recreated every time the function is called. This makes it impossible for the evaluator to reliably modify their state or observe changes across multiple validation steps.

By defining objects globally:

- Their internal state persists across multiple checks
- The hook can modify values (e.g., ADC readings) between validations
- The same object instance is reused without reinitialization
- Test logic remains deterministic and predictable

Consider the following validation logic used in a hook test:

```
# BELOW threshold
adc._value = int(((EXPECTED_THRESHOLD - 1) * 0.01 / VREF) * ADC_MAX)
exec_env["update_led"]()
if led.state != 0:
    return False, "LED should be OFF below threshold", 0.0

# ABOVE threshold
adc._value = int(((EXPECTED_THRESHOLD + 1) * 0.01 / VREF) * ADC_MAX)
exec_env["update_led"]()
if led.state != 1:
    return False, "LED should be ON above threshold", 0.0

return True, "Correct implementation", 1.0
```

Here, the evaluator **reuses the same ADC object** while changing its internal value between calls to update_led().

If the ADC object were recreated inside the function on each call, this approach would fail because:

- The injected value would be lost
- The hook would not be able to simulate different input conditions
- The evaluation would become non-deterministic

15.3 How the Hook Executes the Student Code

The student code is executed using:

exec(user_code, exec_env)

All variables created during execution including adc, led, and TEMP_THRESHOLD are stored inside exec_env.

The hook then validates:

1. That required variables exist
2. That hardware objects are correctly instantiated
3. That the logic behaves correctly under different inputs

15.4 Validating Hardware Configuration

The hook first checks structural correctness:

- Ensures TEMP_THRESHOLD exists and equals 20
- Confirms adc was created using Pin(34)
- Confirms led was created using Pin(2)

This ensures the student followed the hardware specification exactly.

15.5 Testing Functional Behavior

To validate behavior, the hook manually injects ADC values and calls the student's logic:

```
adc._value =  int(((EXPECTED_THRESHOLD + 1) * 0.01 / VREF) * ADC_MAX)

exec_env["update_led"]()
```

Two cases are tested:

1. **Below threshold:**
   The ADC value is set to simulate a temperature slightly below 20°C.
   The LED must remain OFF.
2. **Above threshold:**
   The ADC value is increased to simulate a temperature above 20°C.
   The LED must turn ON.

Because the ADC and LED are simulated objects, their internal state can be inspected directly without relying on real hardware or timing.

Chapter 16: Example: Temperature Measurement Using an NTC Thermistor

This example demonstrates how a hook test validates a temperature-measurement program that uses an NTC thermistor connected to an ESP32 ADC pin.

The goal of the exercise is to ensure that the student correctly implements:

- ADC reading
- Voltage conversion

- Resistance calculation
- Temperature computation using the Beta equation

All validation is performed without real hardware, using simulated ADC behavior.

16.1 Problem Overview

The student is required to:

- Create an ADC object connected to GPIO 34
- Read an analog value from the ADC
- Convert the ADC value to voltage
- Compute thermistor resistance using a voltage divider equation
- Calculate temperature using the Beta formula
- Return the temperature from a function named read_temperature()

The evaluator must verify both correct structure and correct numerical behavior.

```python
import sys
import textwrap
import math
import machine


def check_answer(user_answer):

    # ----------------------------
    # Execute student code
    # ----------------------------
    exec_env = {}
    try:
        exec(textwrap.dedent(user_answer), exec_env)
    except Exception as e:
        return False, f"Runtime error: {e}", 0.0

    # ----------------------------
    # Validate ADC object
    # ----------------------------
    if "adc" not in exec_env:
        return False, "ADC object not created", 0.0

    adc = exec_env["adc"]

    if not isinstance(adc, machine.ADC):
        return False, "ADC must be created using machine.ADC", 0.0

    # ----------------------------
    # Validate Pin
    # ----------------------------
    if not hasattr(adc, "pin"):
        return False, "ADC must be initialized with a Pin", 0.0

    pin = adc.pin

    # Accept all valid pin representations
    if hasattr(pin, "pin"):
        pin_num = pin.pin
    elif hasattr(pin, "id"):
        pin_num = pin.id()
    elif isinstance(pin, int):
        pin_num = pin
    else:
        return False, "Invalid Pin configuration", 0.0

    if pin_num != 34:
        return False, "ADC must be connected to Pin 34", 0.0

    # ----------------------------
    # Validate function existence
    # ----------------------------
    if "read_temperature" not in exec_env:
        return False, "Function read_temperature() not defined", 0.0

    read_temperature = exec_env["read_temperature"]

    # ----------------------------
    # Expected temperature logic
    # ----------------------------
    def expected_temp(adc_val):
        V_REF = 3.3
        ADC_MAX = 4095
        R_FIXED = 10000
        R0 = 10000
        BETA = 3950
        T0 = 298.15

        voltage = (adc_val / ADC_MAX) * V_REF
        resistance = R_FIXED * voltage / (V_REF - voltage)
        temp_k = 1 / ((1 / T0) + (1 / BETA) * math.log(resistance / R0))
        return temp_k - 273.15

    # ----------------------------
    # Functional validation
    # ----------------------------
    for adc_val in (1200, 2000, 3000):
        adc._value = adc_val

        try:
            student_temp = read_temperature()
        except Exception as e:
            return False, f"Error while executing read_temperature(): {e}", 0.0

        expected = expected_temp(adc_val)

        if abs(student_temp - expected) > 1.0:
            return False, f"Incorrect temperature calculation for ADC={adc_val}", 0.0

    return True, "Correct implementation", 1.0
```

16.2 Why Global Objects Are Required

The ADC object must be created outside the read_temperature() function.

This is essential because the hook test:

- Executes the student code once
- Then manually injects different ADC values
- Calls read_temperature() multiple times

If the ADC were created inside the function, its state would be recreated on every call, making validation unreliable. Defining it globally ensures that the hook can safely manipulate the ADC value between evaluations.

16.3 How the Hook Executes the Student Code

The student's submission is executed using:

exec(user_code, exec_env)

This places all defined variables, including adc and read_temperature, into a shared execution dictionary (exec_env).
The hook then accesses these objects directly for validation.

16.4 Structural Validation

Before testing functionality, the hook verifies that the required structure exists:

- An adc object is defined
- The ADC was created using machine.ADC
- The ADC is connected to GPIO 34
- A function named read_temperature() exists

These checks ensure that the student followed the problem specification before moving to numerical validation.

16.5 Functional Validation Logic

Once structure is confirmed, the hook performs behavioral testing.

1. A known ADC value is injected into the simulated ADC object.
2. read_temperature() is called.
3. The returned value is compared against the expected temperature computed using the Beta equation.

This process is repeated for multiple ADC values to ensure correctness across a range of inputs.

Because the ADC object is simulated, its internal value can be modified directly, allowing the evaluator to test different conditions without re-running the student code.

Chapter 17: Example: PWM-Based Motor Control

This example demonstrates how a hook test validates correct usage of Pulse Width Modulation (PWM) for motor control on an ESP32. The goal is to ensure that the student correctly configures a PWM output using the required pin, frequency, and duty cycle.

17.1 Problem Overview

The student is required to:

- Create a PWM object using Pin(5)
- Set the PWM frequency to **1000 Hz**
- Set the duty cycle to **512**
- Return the configured PWM object from a function named setup_motor()

The test verifies both *structural correctness* (correct object creation) and *behavioral correctness* (correct parameter values).

```python
import textwrap
import sys
import machine


def check_answer(user_answer):

    # ----------------------------
    # Execute student code
    # ----------------------------
    exec_env = {}
    try:
        exec(textwrap.dedent(user_answer), exec_env)
    except Exception as e:
        return False, f"Runtime error: {e}", 0.0

    # ----------------------------
    # VALIDATIONS
    # ----------------------------

    if "setup_motor" not in exec_env:
        return False, "Function setup_motor() not defined", 0.0

    if not callable(exec_env["setup_motor"]):
        return False, "setup_motor must be a function", 0.0

    try:
        motor = exec_env["setup_motor"]()
    except Exception as e:
        return False, f"Error while executing setup_motor(): {e}", 0.0

    # --- Type check ---
    if not isinstance(motor, machine.PWM):
        return False, "Returned object is not a PWM instance", 0.0

    # --- Pin validation ---
    if not hasattr(motor, "pin") or not isinstance(motor.pin, machine.Pin):
        return False, "PWM must be initialized using Pin()", 0.0

    if motor.pin.pin != 5:
        return False, "PWM must be initialized on Pin 5", 0.0

    # --- Frequency validation ---
    if motor.freq() != 1000:
        return False, "PWM frequency must be set to 1000 Hz", 0.0

    # --- Duty cycle validation ---
    if motor.duty() != 512:
        return False, "PWM duty cycle must be set to 512", 0.0

    return True, "Correct PWM implementation", 1.0
```

17.2 Execution Model

The student's code is executed using exec() inside a controlled environment. All variables created during execution are stored in a dictionary (exec_env), which allows the hook to inspect them afterward.

Once execution completes, the hook retrieves and validates the object returned by setup_motor().

17.3 Structural Validation

The hook first ensures that:

- A function named setup_motor() exists
- The function is callable
- The function returns an object of type machine.PWM

This ensures that the student followed the required program structure and used the correct API.

17.4 PWM Configuration Validation

After confirming the object type, the hook validates the configuration in a step-by-step manner:

1. **Pin Verification**
   The hook checks that the PWM object was created using Pin(5).
   This ensures correct hardware mapping.
2. **Frequency Validation**
   The PWM frequency is checked using motor.freq() and must be exactly 1000.
3. **Duty Cycle Validation**
   The duty cycle is verified using motor.duty() and must be set to 512.

Each of these checks directly corresponds to a requirement in the problem statement, ensuring strict compliance.

Chapter 18: Example: Using a Timer Callback

This example demonstrates how a timer-based callback can be validated using the hook evaluation framework. Unlike simple value checks, timer-based logic requires verifying that a callback function is correctly registered and invoked.

The goal of this task is to ensure that the student understands how to configure a Timer, attach a callback, and modify program state when the timer triggers.

18.1 Problem Overview

The student is required to:

- Import the Timer class from the machine module
- Define a callback function that updates a variable
- Create a timer using Timer(0)
- Configure the timer using the init() method
- Ensure the callback function is executed when the timer triggers

The callback must modify a variable named triggered, which is later inspected by the evaluator.

```python
import textwrap
import sys
import machine
import time


def check_answer(user_answer):

    # ----------------------------
    # Execute student code
    # ----------------------------
    exec_env = {}
    try:
        exec(textwrap.dedent(user_answer), exec_env)
    except Exception as e:
        return False, f"Runtime error: {e}", 0.0

    # ----------------------------
    # Validate function existence
    # ----------------------------
    if "start_timer" not in exec_env:
        return False, "Function start_timer() not defined", 0.0

    if not callable(exec_env["start_timer"]):
        return False, "start_timer must be a function", 0.0

    # ----------------------------
    # Execute function
    # ----------------------------
    try:
        timer = exec_env["start_timer"]()
    except Exception as e:
        return False, f"Error while executing start_timer(): {e}", 0.0

    # ----------------------------
    # Validate timer object
    # ----------------------------
    if not isinstance(timer, machine.Timer):
        return False, "Returned object is not a Timer instance", 0.0

    if timer.timer_id != 0:
        return False, "Timer ID should be 0", 0.0

    if timer.period != 1000:
        return False, "Timer period should be 1000 ms", 0.0

    if timer.mode != machine.Timer.ONE_SHOT:
        return False, "Timer mode should be ONE_SHOT", 0.0

    if not callable(timer.callback):
        return False, "Timer callback not set", 0.0

    # ----------------------------
    # Simulate timer trigger
    # ----------------------------
    exec_env["triggered"] = False
    timer.callback(timer)

    if exec_env["triggered"] is not True:
        return False, "Timer callback did not set 'triggered' to True", 0.0

    return True, "Correct Timer implementation", 1.0
```

18.2 Structure of the Student Code

The expected structure of the student's code includes:

- A global variable triggered, initially set to False
- A callback function that sets triggered = True
- A function named start_timer() that:
    - Creates a Timer(0) instance
    - Configures it using init()
    - Returns the configured timer object

This structure ensures the hook can both invoke the timer and verify that the callback was correctly executed.

18.3 How the Hook Executes the Code

The hook executes the student code inside an isolated execution environment using exec().

Once execution completes, the hook:

1. Confirms that start_timer() exists and is callable
2. Calls start_timer() to obtain the timer instance
3. Verifies the timer's configuration:
    - Correct timer ID
    - Correct mode (ONE_SHOT)
    - Correct period value
4. Manually triggers the callback by invoking it directly
5. Checks whether the global variable triggered has been updated

This controlled invocation allows the hook to simulate timer behavior without relying on real-time delays.

18.4 Conclusion

This section presented a structured approach to designing, validating, and evaluating MicroPython-based programs using a hook-driven testing framework. By abstracting hardware interactions through simulated modules and enforcing consistent execution patterns, the system enables reliable, repeatable, and hardware-independent assessment of embedded programming tasks.