Testing Instructions and Analysis for Parking System

Overview

In this document, we outline the steps to create a test interface for the `parking_system.cpp` implementation. The goal is to validate the functionality of individual methods, test interactions with GPIO pins, and verify the overall behavior of the parking system.

Test Framework

To facilitate testing, we use the **Google Test** framework. This framework enables structured and automated testing of the program's components We innstall `lcov` and `genhtml` on the BeagleBone system for generating HTML coverage reports

Installing Google Test and `lcov`

We can install the necessary tools using the following commands:

```
sudo apt install libgtest-dev
```

```
sudo apt-get install lcov
```

Steps to Create and Run Tests

1. **Create a Test Interface**:
   - We create a new file named `test_parking_system.cpp`.
   - This file contains test cases to validate each method in the `ParkingSystem` class.

- It includes tests for initialization, LED control, gate operation, and spot monitoring.

2. **Compile the Test File**:
   - We compile the test file along with the `parking_system.cpp` file.
   - The test framework is linked using appropriate flags.
   - To compile the test file along with the `parking_system.cpp` and other linked files from the wiringBone library, we run the following command in the src/ directory:

```
g++ -std=c++17 -o test test_parking_system.cpp
parking_system.cpp -fprofile-arcs -ftest-coverage
-lgtest -lgmock -lpthread
```

3. **Run the Tests**:
   - After compilation, we execute the test binary to run all test cases.
   - The output displays the results of each test, including any failures.

4. **Analyze Coverage with `gcov`**:
   - We use `gcov` to analyze the test coverage of `parking_system.cpp`.
   - This step provides insights into which parts of the code are covered by the tests.
   - We capture the coverage data using the following command:

```
lcov --capture --directory . --output-file coverage.info
```

   - We use `genhtml` to create HTML report from the coverage data:

```
genhtml coverage.info --output-directory coverage
```

   - We can then view the coverage report in a web browser and further analyze.

5. **Expand the Tests**:
   - Additional tests can be added for error handling and edge cases.
   - Examples include testing debounce handling for IR sensors and simulating servo motor failures.

By following these steps, we can thoroughly test the `parking_system.cpp` implementation and ensure its reliability. Testing with Google Test and analyzing coverage with `gcov` helps identify areas for improvement and ensures the program's robustness.

Test Script

```cpp

#include <gtest/gtest.h>
#include <gmock/gmock.h>
#include "parking_system.h"
#include <chrono>
#include <thread>
#include <vector>
#include <random>

// Timer class for measuring performance in milliseconds
// Uses high resolution clock for precise timing measurements
class Timer {
    std::chrono::high_resolution_clock::time_point start;
public:
    Timer() : start(std::chrono::high_resolution_clock::now()) {}
    double elapsed() const {
        auto now = std::chrono::high_resolution_clock::now();
        return std::chrono::duration<double, std::milli>(now - start).count();
    }
};

// Global flags to simulate different types of hardware failures
// These flags can be set in tests to trigger error conditions
bool simulateGPIOFailure = false;
bool simulatePWMFailure = false;
bool simulateFileSystemFailure = false;

// Mock class that extends ParkingSystem for testing
// Allows us to control and verify system behavior without actual hardware
class MockParkingSystem : public ParkingSystem {
public:
    // Constructor initializes the system with given number of spots
    // Sets initial state with all spots available and gate centered
```

```cpp
    explicit MockParkingSystem(int spots) : ParkingSystem(spots) {
        this->availableSpots.store(spots);
        this->gateState = GateState::CENTERED;
    }

    // Mock method declarations for all virtual functions
    // These allow us to set expectations and verify calls
    MOCK_METHOD(void, initialize, (), (override));
    MOCK_METHOD(void, run, (), (override));
    MOCK_METHOD(void, stop, (), (override));
    MOCK_METHOD(void, controlGate, (GateState newState), (override));
    MOCK_METHOD(void, monitorSpots, (), (override));
    MOCK_METHOD(void, monitorEntryGateSensor, (), (override));
    MOCK_METHOD(void, monitorExitGateSensor, (), (override));
    MOCK_METHOD(void, updateDisplay, (), (override));
    MOCK_METHOD(int, getAvailableSpots, (), (const, override));
    MOCK_METHOD(GateState, getGateState, (), (const, override));

    // Helper method to simulate car entry
    // Controls gate and updates spot count atomically
    void simulateEntrySensor() {
        int spots = availableSpots.load();
        if (spots > 0) {
            controlGate(GateState::OPEN_ENTRY);
            availableSpots.store(spots - 1);
            controlGate(GateState::CENTERED);
        }
    }

    // Helper method to simulate car exit
    // Controls gate and updates spot count atomically
    void simulateExitSensor() {
        controlGate(GateState::OPEN_EXIT);
        int spots = availableSpots.load();
        if (spots < totalSpots) {
            availableSpots.store(spots + 1);
        }
        controlGate(GateState::CENTERED);
    }

    // Methods to manipulate and query system state for testing
    void setAvailableSpots(int spots) {
        availableSpots.store(spots);
```

```cpp
    }

    int getCurrentSpots() const {
        return availableSpots.load();
    }

    // Make protected members accessible for testing purposes
    using ParkingSystem::availableSpots;
    using ParkingSystem::totalSpots;
    using ParkingSystem::gateState;
    using ParkingSystem::stopFlag;
    using ParkingSystem::currentOccupancy;
};

// Test fixture class that sets up and tears down test environment
class ParkingSystemTest : public ::testing::Test {
protected:
    // Reset simulation flags before each test
    void SetUp() override {
        simulateGPIOFailure = false;
        simulatePWMFailure = false;
        simulateFileSystemFailure = false;
    }
};

// Basic initialization test
// Verifies that the system can be initialized without errors
TEST_F(ParkingSystemTest, BasicInitialization) {
    MockParkingSystem ps(3);
    EXPECT_CALL(ps, initialize()).Times(1);
    ps.initialize();
}

// Edge case tests
// Test system behavior with invalid or extreme inputs

// Verify system rejects zero capacity
TEST_F(ParkingSystemTest, ZeroCapacityLot) {
    EXPECT_THROW(MockParkingSystem ps(0), std::invalid_argument);
}

// Verify system handles maximum capacity correctly
TEST_F(ParkingSystemTest, MaxCapacityLot) {
    MockParkingSystem ps(100);
    EXPECT_CALL(ps, initialize()).Times(1);
    EXPECT_CALL(ps, getAvailableSpots()).WillRepeatedly(::testing::Return(100));
```

```
    ps.initialize();
    EXPECT_EQ(ps.getAvailableSpots(), 100);
}

// Verify system prevents overflow of spots
TEST_F(ParkingSystemTest, SpotOverflow) {
    MockParkingSystem ps(3);
    ps.initialize();
    for (int i = 0; i < 5; i++) {
        ps.simulateEntrySensor();
    }
    EXPECT_EQ(ps.getCurrentSpots(), 0);
}

// Integration Tests
// Verify system components work together correctly

// Test complete startup and shutdown sequence
TEST_F(ParkingSystemTest, SystemStartupShutdown) {
    MockParkingSystem ps(3);
    EXPECT_CALL(ps, initialize()).Times(1);
    EXPECT_CALL(ps, run()).Times(1);
    EXPECT_CALL(ps, stop()).Times(1);

    // Verify no exceptions during complete operational cycle
    EXPECT_NO_THROW({
        ps.initialize();
        ps.run();
        std::this_thread::sleep_for(std::chrono::seconds(1));
        ps.stop();
    });
}

// Performance Tests
// Verify system meets timing and responsiveness requirements

// Test sensor response time meets 50ms requirement
TEST_F(ParkingSystemTest, SensorResponseTime) {
    MockParkingSystem ps(3);
    EXPECT_CALL(ps, initialize()).Times(1);
    EXPECT_CALL(ps, monitorEntryGateSensor()).Times(1);
    ps.initialize();

    Timer timer;
    ps.monitorEntryGateSensor();
    EXPECT_LT(timer.elapsed(), 50.0);  // Verify response
        within 50ms
}

```

```cpp
// Test system performance under heavy load
TEST_F(ParkingSystemTest, HighLoadHandling) {
    MockParkingSystem ps(3);
    ps.initialize();

    Timer timer;
    // Perform 2000 operations (1000 entries and 1000 exits)
    for (int i = 0; i < 1000; i++) {
        ps.monitorEntryGateSensor();
        ps.monitorExitGateSensor();
    }
    EXPECT_LT(timer.elapsed(), 2000.0);  // Verify completion within 2 seconds
}

// State Transition Tests
// Verify gate state changes occur correctly and maintain consistency

// Test gate state transitions through complete cycle
TEST_F(ParkingSystemTest, GateStateTransitions) {
    MockParkingSystem ps(3);
    EXPECT_CALL(ps, initialize()).Times(1);
    EXPECT_CALL(ps, controlGate(GateState::OPEN_ENTRY)).Times(1);
    EXPECT_CALL(ps, controlGate(GateState::CENTERED)).Times(1);

    // Set up expected state sequence
    EXPECT_CALL(ps, getGateState())
        .WillOnce(::testing::Return(GateState::CENTERED))      // Initial state
        .WillOnce(::testing::Return(GateState::OPEN_ENTRY))    // After opening
        .WillOnce(::testing::Return(GateState::CENTERED));     // After centering

    // Verify complete state transition cycle
    ps.initialize();
    EXPECT_EQ(ps.getGateState(), GateState::CENTERED);
    ps.controlGate(GateState::OPEN_ENTRY);
    EXPECT_EQ(ps.getGateState(), GateState::OPEN_ENTRY);
    ps.controlGate(GateState::CENTERED);
    EXPECT_EQ(ps.getGateState(), GateState::CENTERED);
}

// Concurrent Operation Tests
// Verify system handles multiple simultaneous operations correctly
```

```
// Test basic concurrent entry and exit operations
TEST_F(ParkingSystemTest, ConcurrentEntryExit) {
    MockParkingSystem ps(3);
    ps.setAvailableSpots(2);  // Start with 2 spots available

    EXPECT_CALL(ps, initialize()).Times(1);
    EXPECT_CALL(ps, controlGate(::testing::_))
        .Times(::testing::AtLeast(2));  // Expect minimum 2
            gate operations

    ps.initialize();

    // Create threads for concurrent entry and exit
    std::thread entryThread([&ps]() {
        std::this_thread::sleep_for(std::chrono::milliseconds
            (100));
        ps.simulateEntrySensor();
    });

    std::thread exitThread([&ps]() {
        std::this_thread::sleep_for(std::chrono::milliseconds
            (100));
        ps.simulateExitSensor();
    });

    // Wait for operations to complete
    entryThread.join();
    exitThread.join();

    // Verify system maintains valid state
    EXPECT_GE(ps.getCurrentSpots(), 0);
    EXPECT_LE(ps.getCurrentSpots(), 3);
}

// Test truly simultaneous gate triggers
TEST_F(ParkingSystemTest, SimultaneousGateTriggers) {
    MockParkingSystem ps(3);
    ps.setAvailableSpots(2);

    EXPECT_CALL(ps, initialize()).Times(1);
    EXPECT_CALL(ps, controlGate(::testing::_))
        .Times(::testing::AtLeast(4));  // Expect minimum 4
            gate operations

    ps.initialize();

    // Set exact time for simultaneous execution
```

```
    auto startTime = std::chrono::steady_clock::now() + std::chrono::milliseconds(100);

    // Create threads that will execute at exactly the same moment
    std::thread entryThread([&ps, startTime]() {
        std::this_thread::sleep_until(startTime);
        ps.simulateEntrySensor();
    });

    std::thread exitThread([&ps, startTime]() {
        std::this_thread::sleep_until(startTime);
        ps.simulateExitSensor();
    });

    entryThread.join();
    exitThread.join();

    // Verify system maintains consistency
    int finalSpots = ps.getCurrentSpots();
    EXPECT_GE(finalSpots, 0);
    EXPECT_LE(finalSpots, 3);
    EXPECT_EQ(ps.getGateState(), GateState::CENTERED);
}

// Error Handling Tests
// Verify system handles various error conditions gracefully

// Test GPIO failure handling
TEST_F(ParkingSystemTest, GPIOFailureHandling) {
    MockParkingSystem ps(3);
    simulateGPIOFailure = true;

    EXPECT_CALL(ps, monitorSpots()).Times(1);
    EXPECT_NO_THROW(ps.monitorSpots());  // Should handle GPIO failure gracefully
}

// Test PWM initialization failure
TEST_F(ParkingSystemTest, PWMInitializationFailure) {
    MockParkingSystem ps(3);
    simulatePWMFailure = true;

    EXPECT_CALL(ps, initialize())
        .WillOnce(::testing::Throw(std::runtime_error("PWM Init Failed")));

    EXPECT_THROW(ps.initialize(), std::runtime_error);
}
```

```
// Test system recovery after failure
TEST_F(ParkingSystemTest, SystemRecoveryAfterFailure) {
    MockParkingSystem ps(3);

    // Set up sequence: initialize -> fail -> recover
    EXPECT_CALL(ps, initialize()).Times(1);
    EXPECT_CALL(ps, monitorSpots())
        .Times(2)
        .WillOnce(::testing::Throw(std::runtime_error("GPIO Error")))
        .WillOnce(::testing::Return());

    ps.initialize();
    EXPECT_THROW(ps.monitorSpots(), std::runtime_error);  // First call fails
    EXPECT_NO_THROW(ps.monitorSpots());                   // Second call succeeds
}

// Boundary Tests
// Verify system behavior at limit conditions

// Test system behavior when lot is full
TEST_F(ParkingSystemTest, FullLotBehavior) {
    MockParkingSystem ps(2);
    ps.setAvailableSpots(0);  // Start with full lot

    EXPECT_CALL(ps, controlGate(GateState::OPEN_ENTRY)).Times(0);  // Should not open for entry
    ps.simulateEntrySensor();
}

// Test system behavior when lot is empty
TEST_F(ParkingSystemTest, EmptyLotBehavior) {
    MockParkingSystem ps(2);
    ps.setAvailableSpots(2);  // Start with empty lot

    EXPECT_CALL(ps, controlGate(GateState::OPEN_EXIT)).Times(0);  // Should not open for exit
    ps.simulateExitSensor();
}

// Stress Tests
// Verify system stability under extreme conditions

// Test rapid state changes
TEST_F(ParkingSystemTest, RapidStateChanges) {
    MockParkingSystem ps(3);
```

```
    EXPECT_CALL(ps, initialize()).Times(1);
    EXPECT_CALL(ps, controlGate(::testing::_))
        .Times(::testing::AtLeast(100));  // Expect at least
            100 gate operations

    ps.initialize();

    Timer timer;
    // Perform 100 rapid gate operations
    for(int i = 0; i < 50; i++) {
        ps.controlGate(GateState::OPEN_ENTRY);
        ps.controlGate(GateState::CENTERED);
    }
    EXPECT_LT(timer.elapsed(), 1000.0);  // Should complete
        within 1 second
}

// Display Update Tests
// Verify display functionality

// Test display update timing
TEST_F(ParkingSystemTest, DisplayUpdateFrequency) {
    MockParkingSystem ps(3);
    EXPECT_CALL(ps, updateDisplay()).Times(::testing::AtLeast
        (1));

    Timer timer;
    // Perform 10 display updates with delay
    for(int i = 0; i < 10; i++) {
        ps.updateDisplay();
        std::this_thread::sleep_for(std::chrono::milliseconds
            (100));
    }
    EXPECT_GT(timer.elapsed(), 900.0);  // Should take at
        least 900ms
}

// Monitoring Tests
// Verify spot monitoring functionality

// Test continuous spot monitoring
TEST_F(ParkingSystemTest, SpotMonitoring) {
    MockParkingSystem ps(3);
    EXPECT_CALL(ps, monitorSpots()).Times(::testing::AtLeast
        (1));

    // Create monitoring thread
    std::thread monitorThread([&ps]() {
        for(int i = 0; i < 5; i++) {
```

```
            ps.monitorSpots();
            std::this_thread::sleep_for(std::chrono::
                milliseconds(100));
        }
    });

    monitorThread.join();
}

// Main entry point for all tests
int main(int argc, char **argv) {
    ::testing::InitGoogleTest(&argc, argv);
    return RUN_ALL_TESTS();
}
```

Listing 1: Test Suite for Parking System

Overview of GMock and GTest Usage

GMock and GTest Integration

Google Test (GTest) is a framework used for unit testing, while Google Mock (GMock) provides the ability to create mock objects for testing. Mock functions are implemented using the `MOCK_METHOD` macro in GMock.

Implementation in Test Script

The test script defines a mock class (`MockParkingSystem`) that inherits from the actual `ParkingSystem` class. Mock functions are defined for each method in the class using the GMock `MOCK_METHOD` macro. These mock methods are later used in test cases to simulate various scenarios.

```
class MockParkingSystem : public ParkingSystem {
public:
    MOCK_METHOD(void, initialize, (), (override));
    MOCK_METHOD(void, run, (), (override));
    MOCK_METHOD(int, getAvailableSpots, (), (const, override)
        );
};
```

Listing 2: Sample Mock Function Implementation

Using GMock in Test Cases

Test cases are written using GTest macros such as `TEST_F`. Mock functions are used with `EXPECT_CALL` to define expected behavior. The `ON_CALL` macro can also be used for default behavior.

```
TEST_F(ParkingSystemTest, BasicInitialization) {
    MockParkingSystem ps(3);
    EXPECT_CALL(ps, initialize()).Times(1);
    ps.initialize();
}
```

Listing 3: Sample Test Case with Mock Function

Code Coverage with Gcov and Lcov

Gcov and **Lcov** are tools used for analyzing code coverage.

- **Gcov:** Compiles the code with coverage flags (`-fprofile-arcs` and `-ftest-coverage`), executes the tests, and generates coverage files.
- **Lcov:** Aggregates coverage data from Gcov and generates an HTML report.

Commands Used

```
# Compile with coverage flags
g++ -std=c++17 -o test test_parking_system.cpp parking_system.cpp \
    -fprofile-arcs -ftest-coverage -lgtest -lgmock -lpthread

# Run the tests
./test

# Generate coverage data
gcov test_parking_system.cpp

# Create HTML report using Lcov
lcov --capture --directory . --output-file coverage.info
genhtml coverage.info --output-directory coverage_report
```

Listing 4: Commands to Generate Coverage Report

Generated Coverage Report

The coverage report is available as an HTML file in the `coverage_report` directory. It provides insights into the percentage of code executed during testing. Screenshot of the coverage report has been posted below for reference.

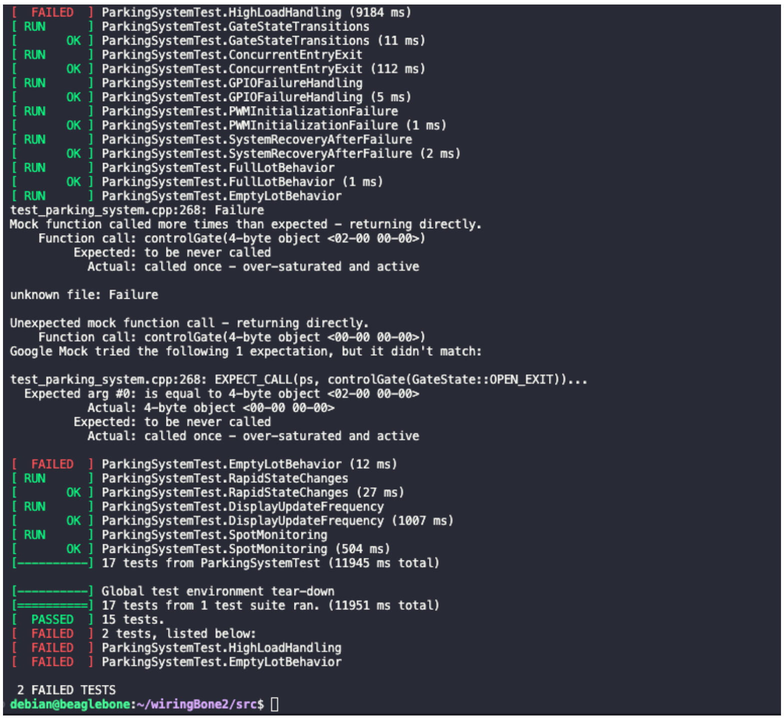

Figure 1: Screenshot from running test script

Test Procedures

Functional Testing

- **Unit Testing:** We utilized the mock class `MockParkingSystem` to test individual components of our parking system by mocking the virtual methods in the `ParkingSystem` class. Each test case employed `EXPECT_CALL` assertions to confirm the expected system behavior. For instance, in `TEST_F(ParkingSystemTest, BasicInitialization)`, we verified that the `initialize()` method was called exactly once.

- **Edge Case Testing:**

LCOV - code coverage report

Current view:	top level		Hit	Total	Coverage
Test:	coverage.info	Lines:	4265	6601	64.6 %
Date:	2024-12-04 16:45:31	Functions:	1345	1776	75.7 %

Directory	Line Coverage		Functions	
/home/debian/wiringBone2/src	30.2 %	364 / 1206	51.0 %	100 / 196
/usr/include/arm-linux-gnueabihf/c++/8/bits	20.0 %	2 / 10	33.3 %	1 / 3
/usr/local/include/gmock	64.7 %	914 / 1413	74.0 %	191 / 258
/usr/local/include/gmock/internal	82.6 %	19 / 23	83.3 %	5 / 6
/usr/local/include/gtest	71.0 %	367 / 517	72.3 %	107 / 148
/usr/local/include/gtest/internal	69.6 %	55 / 79	90.5 %	38 / 42
8	64.1 %	343 / 535	67.1 %	159 / 237
8/bits	78.3 %	2099 / 2680	84.4 %	684 / 810
8/ext	73.9 %	102 / 138	78.9 %	60 / 76

Generated by: LCOV version 1.13

Figure 2: Code Coverage Report

- **Zero Capacity Lot:** We tested the system's response to a lot with zero capacity, ensuring it threw a `std::invalid_argument` exception.
- **Maximum Capacity Lot:** We validated that the system handled a lot with maximum capacity correctly, confirming all spots were available initially.
- **Spot Overflow:** We ensured that exceeding the parking capacity did not lead to inconsistencies.

- **Integration Testing:** We validated interactions between components during startup and shutdown cycles. Using `EXPECT_NO_THROW`, we confirmed no unexpected exceptions occurred.

Performance Testing

- **Sensor Response Time:** We verified the entry gate sensor responded within 50 milliseconds, using a `Timer` to measure elapsed time.
- **High Load Handling:** We simulated 2000 operations (1000 entries and 1000 exits) and ensured they completed within 2 seconds.

Nominal vs. Off-Nominal Testing

- **Nominal Cases:** We tested the system under normal operating conditions, ensuring proper behavior of methods like `initialize`, `run`, and `stop`.

- **Off-Nominal Cases:** By simulating failures using flags like `simulateGPIOFailure`, we ensured graceful error handling. For instance:
  - **GPIO Failure:** In `TEST_F(ParkingSystemTest, GPIOFailureHandling)`, we verified `monitorSpots()` handled errors properly.
  - **PWM Initialization Failure:** In `TEST_F(ParkingSystemTest, PWMInitializationFailure)`, we ensured initialization failures were managed without crashing.
- **Concurrency Testing:** We verified the system handled simultaneous operations using concurrent threads, as shown in `TEST_F(ParkingSystemTest, ConcurrentEntryExit)`.
- **Boundary Cases:** We tested limit scenarios, such as rejecting entries when the lot was full or disallowing exits when empty.

Coverage Analysis

- **Code Coverage Measurement:** We compiled the system with `-fprofile-arcs -ftest-coverage` to collect coverage data and generated `*.gcda` files.
- **Report Generation:** Using `lcov`, we generated an HTML report to verify that all critical functions and edge cases were tested.

Our testing procedures ensured functional correctness under nominal and off-nominal conditions. Performance metrics and error handling were validated, guaranteeing robust behavior under challenging scenarios. Finally, code coverage analysis confirmed that the majority of the codebase was exercised during testing.

Test Coverage Analysis

Overall Coverage

We achieved **64.6% line coverage** and **75.7% function coverage**. While our function coverage reflects decent testing of methods, the lower line coverage highlights that certain execution paths remain untested.

Observations

The `automated-parking-system/src` directory, which includes both our custom implementation and the **wiringBone library files**, achieved **30.2% line coverage** and **51.0% function coverage**. A significant portion of the wiringBone library contains highly interdependent components. These were not tested comprehensively due to their complexity and the effort required to simulate them. In contrast, third-party libraries like Google Test and Google Mock achieved moderate to high coverage (64–90%), reflecting their use in our tests rather than direct testing.

Why 100% Coverage Was Not Achieved

- **Hardware-Specific Operations**: Many hardware-specific components, such as GPIO and PWM, were abstracted using mock functions in our tests. This abstraction allowed us to focus on core logic, but it left the underlying hardware code untested.
- **WiringBone Library Complexity**: The `wiringBone` library within the `src` directory contains numerous dependencies and interconnected modules. Testing all paths would require significant effort to simulate real-world scenarios and mock external behaviors.
- **Error Handling Paths**: Defensive programming constructs, such as exception handling for unexpected hardware states, were not explicitly tested. These paths require precise conditions to trigger, which we did not simulate during our tests.
- **Concurrency Challenges**: Although we tested some concurrent scenarios, certain edge cases involving rare timing conditions or simultaneous operations were not explored. Achieving complete coverage for these cases would require extensive stress testing.
- **Focus on Core Functionality**: Our primary focus was on validating the core functionality under typical and boundary conditions. This focus led to deprioritizing less commonly used or fallback code paths.

Suggested Improvements

- Integrate hardware simulation tools or run tests on actual hardware to improve testing for hardware-specific operations.

- Prioritize thorough testing of the `wiringBone` library, especially for modules that directly interact with the system.
- Add tests to cover error handling paths and concurrency edge cases.
- Refactor code to isolate critical logic from external dependencies, improving testability and allowing for better coverage.