Unit Testing

Purpose

Unit tests in this repository are designed to validate **component behavior**, not entrypoint wiring.

- Tests target logic in `components/common/*` and `components/<board>/*`.
- `src/<board>/main.c` remains focused on initialization and task orchestration.
- Test ownership mirrors component ownership: each component should have corresponding tests under `test/<owner>/`.

Test stack

The project uses PlatformIO's unit testing framework with Unity.

- Each test binary follows the Unity lifecycle:
  - `UNITY_BEGIN()`
  - `RUN_TEST(...)`
  - `UNITY_END()`
- Global Unity output configuration is provided via:
  - `test/unity_config.h`
  - `test/unity_config.c`
- The current output implementation initializes UART and writes test output character-by-character over a COM interface.

Test layout

Tests are organized by ownership and module:

```
test/
├─ common/
│  ├─ test_bucketed_pqueue/
│  ├─ test_kv_pool/
```

```
├─ sensor_board/
│   ├─ test_gps_sensor/
│   ├─ test_imu_sensor/
│   ├─ test_ph_sensor/
│   ├─ test_sensor_basics/
├─ driving_board/
│   ├─ test_calculator/
│   ├─ test_motor/
├─ debugging_board/
│   ├─ test_input_handler/
├─ unity_config.c
└─ unity_config.h
```

Naming conventions:

- Directory: `test/<owner>/test_<module>/`
- File: `test_<behavior>.c` or `test_<module>.c`
- Function: `test_<expected_behavior>()`

Running tests

Run all tests for a specific environment:

```
pio test -e sensor_board
```

Run all tests across all environments:

```
pio test
```

Run a specific test directory:

```
pio test -e sensor_board -f test_imu_sensor
```

Environment test selection (`platformio.ini`)

Test execution is controlled per environment using the `test_filter` setting.

Current configuration:

- `env:sensor_board` → `test_filter = sensor_board/*`
- `env:driving_board` → `test_filter = driving_board/*`
- `env:debugging_board` → `test_filter = common/*`

When adding new tests, ensure the corresponding environment includes the test path in its `test_filter`. Otherwise, the tests will not be executed.

Writing a new unit test

1) Place it by ownership

Tests must follow the same ownership structure as the components.

Example:

- Component: `components/common/kv_pool`
- Test location: `test/common/test_kv_pool/`

2) Use Unity structure

```
#include "unity.h"

void setUp(void) {}
void tearDown(void) {}

void test_example_behavior(void) {
    TEST_ASSERT_TRUE(1);
}

int main(void) {
```

```
    UNITY_BEGIN();
    RUN_TEST(test_example_behavior);
    return UNITY_END();
}
```

---

3) Assert behavior, not implementation

- Use public APIs instead of accessing internal state directly.
- Cover both success and failure cases.
- Use precise assertions:
  - `TEST_ASSERT_EQUAL`
  - `TEST_ASSERT_FLOAT_WITHIN`
  - etc.

4) Keep tests deterministic

- Avoid reliance on shared or previous test state.
- Reset all required state in `setUp`.
- Use time-based operations only when necessary, and keep them bounded.

What to test

- Public functions in component modules
- Input validation and error handling
- Boundary conditions and invalid arguments
- State transitions and invariants

What not to test as unit tests

The following are outside the scope of unit testing:

- Full board startup flows in `main.c`
- End-to-end hardware integration
- Multi-component system orchestration

These belong to integration or system-level testing.

Troubleshooting

- No tests executed: verify `test_filter` for the selected environment (`-e`)
- Test not detected: ensure correct directory naming (`test_<module>`) under `test/`
- No Unity output: check UART/COM configuration in `test/unity_config.c` and board connection settings

---

Revision #1
Created 2026-04-16 07:47:58 UTC by Nikolaos Diamantopoulos
Updated 2026-04-16 07:49:24 UTC by Nikolaos Diamantopoulos