Technical White Paper

Unit Testing Embedded Systems with Zephyr/Ztest

Written by **Parker Lloyd**, Firmware Developer at MistyWest

Table Of Contents

1. Abstract 1
2. Introduction 2
3. Project Structure 6
4. Faking the Kernel 9
5. Test Fixtures 13
6. Execution 16
7. Generating and Reviewing Coverage Reports 17
8. Conclusion 20
9. Appendix 20

1. Abstract

Firmware development for embedded systems presents numerous challenges, particularly in ensuring the reliability and robustness in real-world conditions. Late-stage bug detection is expensive, time-consuming to resolve, and can lead to product failures in the field. This white paper demonstrates how Zephyr's Ztest framework can effectively address these challenges by enabling efficient unit testing. It also covers common difficulties developers face, such as breaking down complex, monolithic codebases, handling intricate kernel dependencies, and ensuring isolated, reliable tests through structured fixtures.

By employing the techniques outlined here, developers can identify and fix defects early in the process, minimizing the risk of costly issues later on.

2. Introduction

Unit testing is the process of verifying individual software components in isolation, ensuring that each "unit" of code functions as expected. Unlike test-driven development (TDD), where tests are written before the code to guide its design, unit testing is often applied after the code is written to validate its behavior.

Unit testing offers several benefits, including early bug detection, improved code quality, and more efficient development.

2.1. Notable Incidents of Firmware Malfunctions

In 1996, the Ariane 5 rocket exploded 37 seconds after launch due to an unhandled integer overflow in its software. A well-designed unit test covering extreme input values could have identified this flaw during development, potentially preventing the $370 million disaster.

Another notable incident is the Therac-25 radiation therapy machine in the 1980s, which caused patient deaths by overdose of radiation due to race conditions in the control software. Proper unit testing of concurrency issues would have exposed these defects, ensuring crucial safety mechanisms worked as intended.

Similarly, in 1999, NASA’s Mars Climate Orbiter was lost due to a software error caused by a mismatch between imperial and metric units. Unit tests that enforce consistent data formats across the system could have caught this discrepancy early, saving the $125 million mission.

2.2. Comparison of Unit Testing Frameworks

This white paper provides a comparison of unit testing frameworks and outlines scenarios where unit testing may not be appropriate. It also breaks down the structure of a typical embedded systems project, illustrating methods for isolating and testing individual components effectively. In addition, strategies for handling kernel dependencies are discussed, with examples of how to wrap and mock kernel functions to create isolated test environments. The use of test fixtures to manage shared resources is also covered, ensuring reliable and repeatable test results.

Finally, we explore the process of executing tests, reviewing coverage reports, and managing testing efforts to ensure comprehensive coverage and maintainability of firmware over time.

Embedded systems—especially those using operating systems like Zephyr—present unique challenges for unit testing. These include managing hardware dependencies, mocking kernel functions, and ensuring test isolation, which make achieving comprehensive test coverage difficult without the right framework.

Zephyr's Ztest framework is designed to address these challenges. While other unit testing frameworks, such as GoogleTest (Gtest) and Unity, are widely used, Ztest is specifically tailored for embedded systems, offering features that make testing in resource-constrained environments more practical.

Feature	Ztest	Gtest (GoogleTest)	Unity
EMBEDDED FOCUS	Yes – designed for Zephyr, integrates well with RTOS features	No – primarily designed for host systems	Yes – designed for embedded systems, minimal footprint
KERNEL INTEGRATION	Tight integration with Zephyr kernel, allows mocking of kernel features	No kernel integration – external tools needed for embedded use	No built-in RTOS support, but lightweight for constrained systems
TEST ISOLATION	Offers built-in mechanisms for test isolation, fixtures and kernel object testing	Provides strong isolation but not tailored for embedded or RTOS-specific needs	Basic test isolation suited for embedded projects
RESOURCE OVERHEAD	Low – designed for constrained environments	High – requires more system resources, better suited for host-based systems	Very low – optimized for minimal resource usage
MOCKING SUPPORT	Supports kernel and hardware mocking via wrappers and stubs	Extensive support but not optimized for kernel-specific features	Basic mocking suitable for small embedded applications
EASE OF USE	Moderate – learning curve specific to Zephyr and kernel features	High – widely used with extensive documentation	Simple, minimal setup, but fewer advanced features

Ztest is primarily used for embedded systems running on the Zephyr RTOS, making it ideal for projects requiring deep integration with real-time operating system features. It excels in IoT devices, wearables, and sensor-driven systems where kernel-level testing and RTOS-specific features are critical.

Gtest (GoogleTest) is best suited for host-based systems with more system resources. It is commonly used in applications like desktop software or server environments where real-time performance isn't as crucial. Its extensive feature set makes it ideal for complex, non-embedded systems.

Unity is a testing framework designed for small, resource-constrained embedded systems, such as microcontrollers in consumer electronics or medical devices. Its lightweight design and low overhead make it the best choice for simple embedded applications where minimal resource usage is key.

2.3. Pros and Cons of Ztest

Pros:

- **Designed for Zephyr**: Seamlessly integrates with Zephyr RTOS, making it ideal for embedded projects that rely on Zephyr.
- **Kernel Integration**: Directly supports kernel objects (e.g., mutexes, queues), allowing you to test real-time OS features without excessive mocking.
- **Low Resource Overhead**: Optimized for resource-constrained environments, making it a great fit for embedded systems with limited memory and processing power.
- **Built-in Fixtures and Test Isolation**: Provides mechanisms to isolate tests and manage fixtures, which is crucial for reliable, reproducible unit tests in embedded contexts.
- **Native Support for Hardware Simulation**: Supports mocking of hardware interfaces and kernel functions, reducing the need for external tools in testing hardware-dependent code.

Cons:

- Learning Curve: Specific to Zephyr's architecture and kernel, meaning there's a steep learning curve if you're unfamiliar with Zephyr internals.
- Limited Outside of Zephyr: The framework is tightly coupled with Zephyr RTOS, making it less useful or applicable in projects that don't use Zephyr.
- Less Extensive Documentation and Community Support: Compared to larger frameworks like GoogleTest, Ztest has a smaller community and fewer learning resources.
- More Manual Mocking: While Ztest does support mocking, it may require more manual effort to mock complex hardware or kernel interactions compared to other frameworks that have larger libraries of mock objects readily available.

2.4. When Unit Testing May Not Be the Best Option

There are times when unit testing may not be worth the effort, depending on the complexity, rate of change, or criticality of the project:

2.4.1. Overly Simple Code

Unit tests might provide limited value for trivial code that doesn't interact with hardware or complex dependencies. However, as code complexity increases, the value of unit testing grows.

- *Very Simple*: A basic blinking LED program toggling a GPIO pin. This code has no dependencies or complex logic, making unit testing unnecessary.
- *Moderately Complex*: A Bluetooth-connected IMU sensor that requires reading and processing data from the sensor. While more complex, its core logic may or may not be simple enough to skip extensive unit testing.
- *Very Complex*: A full communication stack that manages a variety of sensors and actuators in a home automation system. Here, the interaction of many components makes unit testing essential to ensure reliability.

2.4.2. Rapidly Changing Code

Codebases that evolve quickly can make it difficult to maintain meaningful tests. The faster the changes, the more effort is required to keep the tests up to date.

- *Slowly Changing*: A static configuration driver that is rarely updated after the initial development. Unit tests can provide long-term stability and regression checks.
- *Faster Changing*: A networking protocol stack being enhanced with new features every few sprints. Unit tests can help with regression but may require frequent updates.
- *Very Fast Changing*: A rapidly prototyped application where the business logic and structure change weekly. In this case, the maintenance burden of unit tests outweighs their immediate benefit.

2.4.3. Low-Criticality Projects

For projects where the potential impact of bugs is low, investing in unit testing may not be worthwhile. But for high-criticality projects, unit testing is crucial to avoid serious consequences.

- *Low-Critical*: A simple remote control for a consumer electronics device. A minor issue, like a misfired button press, is unlikely to cause significant disruption.
- *Medium-Critical*: A wearable fitness tracker. Incorrect readings may be annoying for the user but aren't life-threatening.
- *High-Critical*: A medical device that monitors heart rate and triggers emergency responses. Here, unit testing is critical to ensure reliable performance in life-or-death situations.

In such cases, depending on the complexity, speed of development, and criticality, focusing on integration or system-level testing may provide more value than unit testing alone.

2.5. Summary

Unit testing may not always be the best investment of time and effort, depending on the complexity, pace of change, and criticality of a project. For very simple code, such as a blinking LED, unit tests offer limited value, while more complex systems, like those with multiple sensors or a communication stack, benefit greatly from unit testing to ensure reliability.

In fast-changing codebases, the maintenance burden of keeping tests up to date can outweigh their usefulness, whereas slowly evolving code, like configuration drivers, can gain stability from unit tests. Similarly, low-criticality projects, like consumer remote controls, may not justify the time spent on unit testing, but in high-criticality applications, such as medical devices, it is essential to prevent serious failures. In the cases that do not warrant unit testing, integration or system-level testing may be a more practical approach.

3. Project Structure

Unit testing large firmware modules can be challenging, especially when functions within a module depend heavily on each other. Testing a module as-is often requires making numerous assertions and navigating unpredictable control paths when unmocked functions are called.

This results in overly complex test cases and increased risk of missing edge cases.
A solution is to break the module into smaller, isolated submodules. Individual mocking of dependent functions is facilitated by this approach, allowing specific functionality to be tested without the need to navigate through complex chains of function calls.

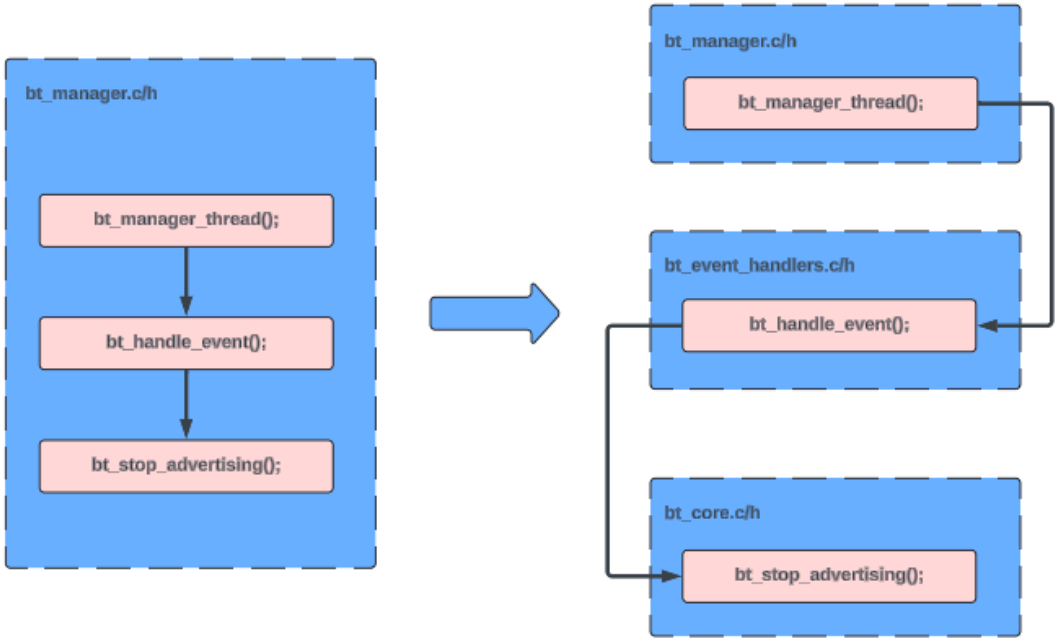

Figure 1. Separating a large module into smaller submodules.

However, a limitation of Ztest is that it is primarily designed for testing libraries rather than the entire codebase. It focuses on testing public functions, meaning that only functions exposed through interfaces can be directly tested. As a result, you may need to redefine some internal or private functions as public to ensure testability, or accept that certain internal logic will not be directly testable.

3.1. Example: Bluetooth Manager Module

Let's consider a Bluetooth Manager module responsible for several tasks such as:

- Connection Management
- Data Transfer
- Low-Level Bluetooth Handling

If you try to test the entire module as a single unit, interdependencies between functions can cause issues. You will need to make numerous assertions that may not even be relevant to the specific test case. Additionally, when an unmocked function is called during the test, the control path may become unpredictable, making the test difficult to manage and unreliable.

The ideal solution is to first break large modules into smaller submodules, making testing more manageable and ensuring that each dependency can be mocked individually. By doing this, you can focus only on the control path of the function being tested, without being affected by the cascading effects of unmocked function calls. If you don't break apart these modules and attempt to create mocks, you are likely to encounter multiple definition errors as your test module would need to include both the real function (contained in the module under test) and its fake counterpart. By splitting the module and isolating each dependency, you avoid this conflict and can more easily mock each function for individual testing.

However, there is a tradeoff to this approach: it may require creating more submodules than you initially expected, with much finer granularity. As you continue breaking down the module to mock interdependent functions, the number of submodules can grow significantly. This can add complexity to your project structure, but the benefit is that you avoid having to repeatedly check irrelevant assertions in your test cases.

3.1.1. Breaking Down the Module

Here are some tips and strategies for breaking a large module into testable submodules:

3.1.1.1. Group by Responsibility

Start by grouping related functionalities into submodules based on their responsibilities. For example:

- **Connection Management** handles tasks like connecting, pairing, and disconnecting.
- **Data Transfer** focuses on sending and receiving data over an active Bluetooth connection.
- **Low-Level Handling** manages interactions with hardware, such as initializing the Bluetooth controller.

3.1.1.2. Move Functions That Call Other Functions to Separate Modules

To simplify testing, consider moving functions that call other functions into separate submodules. This enables you to mock those dependencies independently. Without this separation, you would need to repeatedly mock and make assertions about internal function calls within the function under test, even when they are irrelevant to the current test case.

For instance, if there is a function bt_connect() that calls a hardware-level initialization function, it's better to move this initialization function to the Low-Level Bluetooth Handling module and mock it in the unit tests for the Connection Management module. This ensures that you're testing only the connection logic in isolation, without worrying about hardware interactions.

3.1.1.3. Use Interfaces for Cross-Module Communication

Define clear interfaces, such as header files, to allow submodules to communicate while keeping them loosely coupled. This reduces the complexity of managing dependencies during

testing and allows you to mock only the necessary parts of each submodule. For example, the Connection Management module can expose functions like those related to connecting or disconnecting, while the Data Transfer module can expose functions related to sending data.

Keep in mind that this interface will be the only way to test its associated module, as Ztest only allows for directly testing public functions. Some parts of the code will remain encapsulated and private, as they do not need to be directly called. However, this also means that these private functions cannot be directly tested, so it's important to ensure that the exposed interface covers all necessary testable behaviors.

3.1.1.4. Iterative Refactoring

Breaking a large module into submodules doesn't have to be done all at once. Start by identifying the most obvious groupings of related functions and refactor them into submodules. Over time, as you work through your codebase and tests, you'll discover additional areas where further breakdown is necessary.

3.1.1.5 Test the Submodules Individually

Once the module is broken down into submodules, test each submodule independently. Create mocks for the functions in other submodules that are dependencies, and focus each test on the behavior of the submodule under test. This ensures that your tests remain modular, concise and focused on individual functionalities.

By following these strategies, you reduce the complexity of your tests and ensure that your project remains maintainable over time. Testing becomes more efficient, as you are no longer bogged down by irrelevant function calls or excessive assertions. However, remember that with Ztest's focus on public functions, some internal logic might require redefinition or might not be directly testable.

3.2. Summary

Breaking large firmware modules into smaller, testable submodules simplifies embedded systems testing. This approach allows for focused, independent testing of specific functions, reducing complexity and improving maintainability. Although it may add more submodules, it enhances testability by isolating dependencies and avoiding conflicts between real and mocked functions.

4. Faking the Kernel

When working with Zephyr's Ztest framework, your tests may need to interact with kernel-level functions, such as for message queues, mutexes, or timers. Instead of mocking the entire kernel, a more efficient approach is to create wrappers for the kernel functions used in your code. This way, you can replace the real kernel functions with fake implementations, allowing you to simulate different behaviors during testing.

Let's walk through an example of how to create a fake for the k_msgq_put kernel function by wrapping it in a custom function.

4.1. Create the Kernel Wrapper Header

The first step is to wrap the kernel function you want to fake in your own function. This function will act as a wrapper around the real k_msgq_put function, allowing us to later replace it with a mock in the test environment.

Header File (msgq_wrapper.h)

```C/C++
// Header file for the message queue wrapper
int msgq_wrapper_put(struct k_msgq *queue, void *data, k_timeout_t timeout);
int msgq_wrapper_get(struct k_msgq *queue, void *data, k_timeout_t timeout);
```

This defines the wrapper function msgq_wrapper_put, which will call the actual k_msgq_put function. We will use this wrapper in the application code instead of calling the kernel function directly.

4.2. Implement the Wrapper Function

Next, implement the wrapper in the source file. This function will simply call k_msgq_put internally.

Source File (msgq_wrapper.c)

```C/C++
// Source file for the message queue wrapper
int msgq_wrapper_put(struct k_msgq *queue, void *data, k_timeout_t timeout) {
    // Call the real kernel function
    return k_msgq_put(queue, data, timeout);
}

int msgq_wrapper_get(struct k_msgq *queue, void *data, k_timeout_t timeout) {
    // Call the real kernel function
    return k_msgq_get(queue, data, timeout);
}
```

The msgq_wrapper_put function calls the real kernel function k_msgq_put. By using this wrapper in the actual application code, we gain the ability to replace msgq_wrapper_put with a fake in our unit tests. This enables us to continue using real kernel functions in our test module to help with testing, while also mocking them in our application code where needed.

4.3. Update Application Code

After implementing the wrapper, the next step is to update your application code to use the wrapper function (`msgq_wrapper_put`) instead of directly calling the kernel function (`k_msgq_put`). By doing this, you decouple your application from the real kernel function, allowing you to easily mock or fake the behavior of the wrapper during unit testing. This flexibility ensures that you can simulate various conditions in your tests (such as message queue success or failure) without interacting with the actual kernel.

Replacing the kernel function with the wrapper in your application code is essential because it centralizes how kernel functions are accessed, making the code easier to test, maintain, and extend in the future.

4.4 Create a Fake for the Kernel Wrapper

In the unit test, you will fake the behavior of msgq_wrapper_put to control how it behaves during testing. This allows you to simulate different return values or conditions.

Here's how to create a fake implementation of msgq_wrapper_put for testing.

Fake Header File (fake_msgq_wrapper.h)

```C/C++
#pragma once
#include <zephyr/fff.h>
#include <zephyr/kernel.h>

#ifdef __cplusplus
extern "C" {
#endif

DECLARE_FAKE_VALUE_FUNC(int, msgq_wrapper_get, struct k_msgq *, void *,
k_timeout_t);
DECLARE_FAKE_VALUE_FUNC(int, msgq_wrapper_put, struct k_msgq *, const void *,
k_timeout_t);

#ifdef __cplusplus
}
#endif
```

Fake Source File (fake_msgq_wrapper.c)

```C/C++

#include "fake_msgq_wrapper.h"

DEFINE_FAKE_VALUE_FUNC(int, msgq_wrapper_get, struct k_msgq *, void *,
k_timeout_t);
```

```
DEFINE_FAKE_VALUE_FUNC(int, msgq_wrapper_put, struct k_msgq *, const void *,
k_timeout_t);
```

Explanation:

- The `DECLARE_FAKE_VALUE_FUNC` macro is used to declare fake versions of the message queue functions (`msgq_wrapper_get` and `msgq_wrapper_put`).
- The `DEFINE_FAKE_VALUE_FUNC` macro defines how these fake functions behave during tests.

By using this setup, you can control return values within your unit tests, allowing you to simulate both success and failure scenarios. This ensures that your tests remain isolated and that the message queue behavior is predictable, regardless of real kernel function calls.

4.5. Use the Fake in Your Test Case

In your test case, replace the real msgq_wrapper_put with the fake function using a function pointer or a mocking framework. Here's an example of how to set up a test that uses the fake function.

```
C/C++
int msgq_wrapper_get_custom_fake(struct k_msgq *queue, void *data, k_timeout_t
timeout) {
    // Add desired custom functionality
    return 0; // Return success (no error)
}

ZTEST_F(bt_manager_suite, test_bt_manager_msgq_fake) {
    // Set custom fake
    msgq_wrapper_get_fake.custom_fake = msgq_wrapper_get_custom_fake;
    // Call function that calls msgq_wrapper_get()
    bt_man_calls_msgq_get();
    // Assert call count
    zassert_true(msgq_wrapper_get_fake.call_count == 1, "msgq_wrapper_get call
count does not match expected.");
}
```

Explanation:

- `msgq_wrapper_get_custom_fake` function:
  This is a custom fake implementation of the `msgq_wrapper_get` function. It adds any desired custom functionality for the test and returns `0` to simulate success (no error).
- `ZTEST_F(bt_manager_suite, test_bt_manager_msgq_fake)`:
  This is the actual test case where:

- The `custom_fake` for `msgq_wrapper_get` is set to `msgq_wrapper_get_custom_fake`, replacing the real function with the fake one.
- The test calls `bt_man_calls_msgq_get()` (a function under test that internally calls `msgq_wrapper_get`).
- Finally, the test asserts that `msgq_wrapper_get` was called exactly once by checking its `call_count`. If the call count does not match the expectation, an error message is provided.

4.6. Summary

By creating a wrapper around the kernel function and faking that wrapper in your tests, you can simulate various kernel behaviors without calling the actual kernel functions. This approach helps maintain the integrity of your test environment while giving you full control over how kernel functions behave in specific test scenarios.

You can follow these same steps to fake other kernel functions such as mutexes or timers, allowing you to comprehensively test your application's interaction with Zephyr's kernel.

5. Test Fixtures

Ztest allows you to define test fixtures, which provide a clean and reusable setup for each test case. Fixtures are particularly useful when multiple tests share common resources, such as mock data or test configurations. However, one challenge with Ztest is that it runs all tests within a suite simultaneously by default. This can cause interference if tests modify shared resources, such as fixture members, concurrently.

To ensure that tests are isolated and do not interfere with each other, you can add a mutex to your fixture. This mutex will lock and unlock critical sections of the fixture, forcing tests to run consecutively rather than simultaneously. Let’s break this process down.

5.1. Define the Fixture Structure

First, define the fixture structure that includes the shared resources (like expected_return_value) and the mutex (fixture_mutex) that will control test execution.

```C/C++
struct test_fixture {
    int expected_return_value;  // Shared value among tests
    struct k_mutex fixture_mutex;  // Mutex to control test execution
};

static struct test_fixture *g_fixture = NULL; // Global fixture instance
```

5.2. Setup the Fixture

Next, create a setup function that initializes the shared resources and the mutex. This function is called before each test to prepare the fixture environment.

```
C/C++
static void *test_suite_setup( void )
{
    struct test_fixture *fixture = malloc( sizeof( struct test_fixture ) );
    zassume_not_null( fixture, NULL );
    k_mutex_init( &fixture->mutex );
    g_fixture = fixture;

    return fixture;
}
```

Explanation:

- `struct test_fixture *fixture = malloc(sizeof(struct test_fixture));`
  This line dynamically allocates memory for the test fixture. The `malloc` function is used to allocate enough memory to hold an instance of the `struct test_fixture`. This ensures that each test case has its own isolated fixture to work with.

- `zassume_not_null(fixture, NULL);`
  This is a Zephyr-specific macro (`zassume_not_null`) that ensures the fixture was successfully allocated. If `fixture` is `NULL`, the test will stop and fail immediately because the memory allocation failed. This prevents the test from proceeding with a `NULL` pointer, which would lead to runtime errors.

- `k_mutex_init(&fixture->mutex);`
  This initializes the mutex in the fixture. The `k_mutex_init` function is called to prepare the mutex for use in the test cases, ensuring that the shared resources within the fixture are properly synchronized.

- `g_fixture = fixture;`
  This line assigns the newly created `fixture` to a global pointer (`g_fixture`) so that the fixture can be easily accessed throughout the test suite if needed. This can be useful when multiple test cases need to refer to the same fixture.

- `return fixture;`
  Finally, the fixture is returned, allowing it to be passed into each test case. By returning the fixture, you ensure that the test case constructor (or test case itself) receives a properly initialized fixture to work with.

5.3. Setup Test Case Constructor

In each test case, you should lock the mutex before modifying any shared resources and unlock it afterward. This ensures that only one test at a time can access the shared resources, preventing any potential race conditions or test interference.

Here's how you can set up the constructor for the test case:

```C/C++
static void test_case_constructor( void *f )
{
    struct test_fixture *fixture = (struct test_fixture *)f;
    k_mutex_lock( &fixture->mutex, K_FOREVER );
}
```

Explanation:

- The `test_case_constructor` function is called before each test case. It locks the mutex (`fixture->mutex`), ensuring that only one test case can access the shared resources (such as test fixture members) at a time.

- By using `k_mutex_lock(&fixture->mutex, K_FOREVER)`, the function will wait indefinitely until the mutex is available, ensuring proper synchronization between test cases.

This setup guarantees that test cases are isolated, avoiding issues related to concurrent access to shared resources.

5.4. Releasing the Mutex in Test Destructor

If needed, you can also ensure the mutex is handled correctly in a test destructor (cleanup) to ensure proper resource management after the test completes.

```C/C++
static void test_case_destructor( void *f )
{
    struct test_fixture *fixture = (struct test_fixture *)f;
    k_mutex_unlock( &fixture->mutex );
}
```

5.5. Summary

By including a mutex in your fixture structure and locking/unlocking it within each test case, you ensure that tests do not run simultaneously, preventing conflicts and interference

between tests that modify shared fixture members. This practice helps maintain isolation between tests, reducing the risk of unpredictable test behavior.

6. Execution

6.1. Running Tests

Ztest integrates with the Zephyr build system, allowing you to execute tests using the west twister command.

```
Unset
west twister -vvv --clobber-output -p native_posix_64 -T tests/test_module
--enable-asan
```

You can simplify test execution by creating a script for your test modules, as follows:

```
Unset
#!/bin/bash
west twister -vvv --clobber-output -p native_posix_64 -T tests/test_module
--enable-asan
```

Explanation of each argument:

- `west twister`: Runs the Zephyr testing tool.
- `-vvv`: Maximum verbosity for detailed output.
- `--clobber-output`: Clears previous test results before running.
- `-p native_posix_64`: Runs tests on the 64-bit native POSIX platform (your computer, no hardware needed).
- `-T tests/test_module`: Specifies the test directory to run.
- `--enable-asan`: Enables AddressSanitizer to catch memory issues like buffer overflows.

6.2. Checking for Failures

1. Build Failures: If the build process fails (due to compilation errors or missing dependencies), the logs will display specific errors. Look for keywords like "error" or "failed" in the output. The twister tool will stop running and return a non-zero exit code if the build fails.

2. Test Failures: After a successful build, the tests will execute and their results will be displayed. For each test case, you will see whether it passed, failed, or was skipped. A summary at the end of the log will indicate the number of tests run and how many

succeeded or failed.

3. AddressSanitizer Issues: If you use `--enable-asan`, any memory-related issues will be flagged with clear error messages and stack traces, showing where the memory violation occurred (e.g., buffer overflow or invalid memory access).

6.3. Summary

Ztest integrates with the Zephyr build system, allowing for automated test execution and monitoring through the `west twister` command. Be mindful of build and test failures, as well as memory issues flagged by AddressSanitizer, which provide clear logs and error messages for easier debugging.

7. Generating and Reviewing Coverage Reports

To ensure complete test coverage, you can generate coverage reports using a script like the one below:

```
Unset
#!/bin/bash
west twister --clobber-output --coverage --coverage-tool gcovr
--coverage-basedir /workspaces/project/application/src -p native_sim_64 -T .
```

Explanation of each argument:

- `west twister`:
  `twister` is the Zephyr test harness used to run tests, including unit tests and other types of tests across multiple configurations, platforms, or environments. It works with the Zephyr build system and allows you to manage test execution, results, and coverage.

- `--clobber-output`:
  This option forces Twister to clean (delete) any existing output directory before running the tests. It ensures a fresh start for test execution, preventing old test data or artifacts from affecting the new run.

- `--coverage`:
  This flag enables code coverage analysis during the test run. Coverage analysis tracks how much of your code is exercised by the tests, providing insights into untested areas (e.g., functions or lines not covered by the test suite).

- `--coverage-tool gcovr`:
  Specifies that `gcovr` will be used as the tool for generating code coverage reports.

`gcovr` is a popular tool for creating human-readable coverage reports, such as HTML or XML, from `gcov`-based data (produced by GCC compilers). It simplifies the generation of coverage data.

- `--coverage-basedir /workspaces/project/application/src`:
  Defines the base directory for the coverage report. This tells `gcovr` where the source code being tested is located (in this case, `/workspaces/project/application/src`). This ensures that the coverage report reflects which parts of the source files have been tested.

- `-p native_sim_64`:
  This option specifies the platform on which the tests will be run. In this case, `native_sim_64` refers to a simulated 64-bit native platform (also known as `native_posix`). This allows the tests to run on a host machine (e.g., your local PC) rather than on embedded hardware, making it useful for running unit tests or coverage checks.

- `-T .`:
  This argument tells Twister where to find the test cases. The `-T` flag specifies the directory containing the test suite or unit tests. In this case, the `.` indicates the current directory is used as the test root, where Twister will search for tests.

This script will output an HTML report in the twister-out/coverage/ directory. The generated report contains details on function, line and branch coverage, providing a high-level view of which parts of your code are exercised by the tests.

7.1. Steps to Review the Coverage Report

1. Locate the Coverage Report: After running the script, navigate to the twister-out/coverage/ directory and open the index.html file in your browser.

2. Understanding the Report:
   a. Overall Coverage: The main page will show the percentage of coverage for your project. Focus on the function and branch coverage percentages, as these indicate how much of your code logic is tested.
   b. Per-File Details: Clicking on individual files will give you a more granular view of which functions and lines have been tested or missed.

3. Identifying Gaps: Look for:
   a. Untested Functions: Functions or methods not covered by any tests should be reviewed. These could represent areas of the code that need further unit testing.
   b. Untested Branches: Even if a function is covered, certain branches within the function might not be. Examine branches with conditions (e.g., if statements)

that haven't been fully tested.

4. Investigating and Correcting Errors:
    a. Check for Critical Code Paths: Focus on critical sections of the code, such as error-handling paths, and ensure they are adequately tested.
    b. Refactor Tests: If necessary, refactor your tests to cover additional branches or edge cases. This might involve adding new test cases or modifying existing ones to include untested conditions.

7.2. Managing Large Reports

For larger projects, coverage reports can contain vast amounts of data, making it difficult to sift through manually. Here are some tips to manage the complexity:

1. Filter by Coverage Type:
    a. Use the report's filtering options (if available) to focus on low coverage files or specific coverage types (e.g., branch, line, or function).

2. Use Tools for Analysis:
    a. Many coverage tools provide built-in metrics to highlight areas with low coverage. Look for features that allow you to sort files by their coverage percentage or by functions with missing coverage.

3. Prioritize Critical Code:
    a. Start by prioritizing coverage for files and modules that are critical to your application's functionality. Once those are sufficiently tested, move on to less critical areas.

4. Automation:
    a. Consider automating the generation and review of coverage reports within your CI/CD pipeline. This allows the team to track coverage changes over time and ensures that newly introduced code meets the coverage criteria before it is merged.

By systematically reviewing the coverage report, focusing on untested areas and utilizing tools to manage the data, you can ensure comprehensive test coverage for your application. Address any missing coverage by adding or updating unit tests to improve reliability and maintainability.

7.3. Summary

Generating and reviewing coverage reports helps identify untested areas in your code, such as missed functions or branches, ensuring thorough test coverage. Focus on critical code paths, especially error-handling sections, and prioritize improving coverage for these areas to enhance reliability. By automating coverage analysis and systematically addressing gaps, you can maintain high code quality and ensure comprehensive testing.

8. Conclusion

Unit testing with Zephyr's Ztest framework provides a powerful approach to improving firmware reliability in embedded systems. Through careful project structuring, including breaking large modules into smaller, testable submodules and utilizing interfaces for communication, testing becomes more manageable and focused.

While Ztest is primarily designed for testing libraries and public functions, the strategies outlined in this whitepaper help mitigate the framework's limitations, such as redefining internal functions for better testability. Additionally, implementing kernel fakes and using test fixtures to control execution order further ensures reliable, isolated tests.

Despite its learning curve and specificity to Zephyr, Ztest remains a valuable tool for those working within this RTOS. By incorporating unit tests into your development process, you can reduce bugs early, ensure code coverage and deliver robust, maintainable firmware.

9. Appendix

9.1. References and Additional Reading

- Ztest Documentation: https://docs.zephyrproject.org/latest/develop/test/ztest.html
- Google-Test Documentation: https://google.github.io/googletest/
- Unity Documentation: https://docs.unity3d.com/Packages/com.unity.test-framework@1.1/manual/manual.html
- Cost of Late Bug Detection: https://www.functionize.com/blog/the-cost-of-finding-bugs-later-in-the-sdlc
- Ariane Flight V88: https://en.wikipedia.org/wiki/Ariane_flight_V88
- Therac-25 Race Condition: https://en.wikipedia.org/wiki/Therac-25
- Mars Climate Orbiter: https://en.wikipedia.org/wiki/Mars_Climate_Orbiter

---