Smart Thermostat

Balemarthy Vamsi Krishna

Using Test-Driven Development (TDD) for Smart Thermostat project

Test-Driven Development (TDD) is a software development approach where tests are written before the actual code. The basic idea behind TDD is to write a test for a function or feature before you write the code that makes the test pass. This helps ensure that the software is built with testing in mind from the start, promoting cleaner and more reliable code.

The TDD cycle typically follows these steps:

1. Write a test: Start by writing a test for the next bit of functionality you want to add.
2. Run the test: Run the test, which should fail since the functionality isn't implemented yet.
3. Write the code: Write the minimum amount of code required to make the test pass.
4. Run tests again: Run the test again, and if it passes, you've successfully added the new functionality.
5. Refactor: Look at the new code and any old code it interacts with, and make any necessary cleanups to improve efficiency and readability.
6. Repeat: Continue with the next piece of functionality.

The process follows a short iteration cycle known as Red-Green-Refactor:

1. Red: Write a test that defines a function or improvements of a function, which should fail initially because the function isn't implemented yet.
2. Green: Write just enough code to make the test pass. This usually means the function does exactly what is needed—no more, no less.
3. Refactor: Optimize the new code, making sure it adheres to the clean code standards without affecting the behavior (tests must still pass).

This approach ensures that all code changes are testable and have tests written for them, which can reduce bugs and improve design.

TDD helps in minimizing the possibility of introducing errors when adding new features or refactoring existing code, making it a popular practice in agile software development environments.

Applying TDD to Each Module of smart thermostat

1. JSON Library (`json_library.c`, `json_library.h`)

Test Case: JSON Initialization

- Objective: Test that `json_init` properly allocates and initializes a new JSON object.

- Expected Result: `json_init` returns a non-null pointer, `size` is correctly initialized, and the initial buffer contains only the opening curly brace `{`.

Test Case: Adding a Valid String to JSON

- Objective: Ensure `json_add_string` correctly formats and appends a string key-value pair to the JSON object.

- Expected Result: After adding `"key": "value"`, the buffer contains the correct JSON string.

Test Case: Handling Buffer Overflow in JSON Addition

- Objective: Verify that `json_add_string` correctly resizes the buffer when adding a large string that exceeds the initial buffer capacity.

- Expected Result: The buffer is resized without data loss, and the JSON string is correctly appended.

Test Case: Receiving Invalid JSON Strings

- Objective: Test the library's ability to reject malformed JSON strings (e.g., missing quotes, braces).

- Expected Result: The library should not crash or misbehave; ideally, it should return an error or handle the case gracefully.

Test Case: Adding Multiple Key-Value Pairs

- Objective: Test the ability of `json_add_string` to handle multiple key-value pairs sequentially.

- Expected Result: JSON object should contain all key-value pairs, properly separated by commas.

Test Case: Handling Special Characters in Values

- Objective: Ensure that special characters (e.g., newlines, tabs, quotes) in values are correctly escaped.

- Expected Result: JSON strings should include escaped characters where necessary, adhering to JSON standards.

Test Case: Adding Nested JSON Objects

- Objective: Verify that `json_add_string` can handle adding a nested JSON object correctly.

- Expected Result: The JSON string should correctly reflect nested structures with valid JSON syntax.

2. HTTP Packet Library (`http_packet.c`, `http_packet.h`)

Test Case: HTTP Response Initialization

- Objective: Test `http_init_response` to ensure it properly sets up the initial HTTP response structure.

- Expected Result: HTTP response contains a default status line and empty headers and body.

Test Case: Setting HTTP Status and Headers

- Objective: Ensure that `http_set_status` and `http_set_header` accurately update the HTTP response's status line and headers.

- Expected Result: The status line reflects the new status, and headers are appended correctly.

Test Case: Finalizing HTTP Response

- Objective: Test `http_finalize_response` to check if it correctly calculates and adds the `Content-Length` header based on the body length.

- Expected Result: `Content-Length` matches the length of the body, and the response is properly terminated with an empty line.

Test Case: Adding Multiple Headers

- Objective: Test `http_set_header` to ensure it can handle multiple headers being added to the HTTP response.

- Expected Result: All headers should be correctly formatted and appear in the response.

Test Case: Content-Type Handling

- Objective: Ensure that setting different `Content-Type` headers results in correct MIME types in the HTTP header.

- Expected Result: The HTTP response should reflect the exact MIME type set by the test case.

Test Case: Large HTTP Body

- Objective: Verify that `http_set_body` correctly handles a large body, testing buffer handling and Content-Length accuracy.

- Expected Result: The body should be correctly included in the response, and `Content-Length` should match the actual size of the body.

3. UART Driver (`uart_drv.c`, `uart_drv.h`)

Test Case: UART Initialization

- Objective: Test `USART2_Init` to ensure it configures the hardware registers correctly for standard operation.

- Expected Result: All specified registers are set to expected values for operation.

Test Case: Sending Data Over UART

- Objective: Ensure `USART2_SendData` correctly handles transmission of data over UART.

- Expected Result: Data is placed into the transmission register as expected without overflow, and transmission complete flag is set afterwards.

Test Case: Handling Receive Interrupts

- Objective: Test `USART2_IRQHandler` to verify it correctly reads data from the data register and stores it in a buffer upon a receive interrupt.

- Expected Result: Received data is correctly appended to the buffer, and buffer overflows are handled.

Test Case: Continuous Data Reception

- Objective: Test the USART2 driver's ability to handle continuous data reception without loss.

- Expected Result: All data received should be correctly stored in the buffer, handling wrap-around if necessary.

Test Case: Transmission Under Load

- Objective: Ensure that `USART2_SendData` can handle a high volume of data being sent, testing buffer management and ISR handling.

- Expected Result: Data should be transmitted correctly without loss or corruption, even under high load.

Test Case: Error Handling in Data Transmission and Reception

- Objective: Test how the UART driver handles common communication errors such as frame errors, overrun errors, and noise.

- Expected Result: Errors should be detected, and appropriate error handling measures should be triggered (such as buffer clearing, reinitialization, or error notifications).

Final Thoughts

By creating and running these tests before fully implementing the functionalities, developers ensure that each part of the system works as expected and meets the requirements without side effects. This is especially crucial in embedded systems where debugging can be more challenging than in higher-level software development.

Contact me at

www.balemarthyvamsi.com

https://linkedin.com/in/balemarthyvamsi