Test Plan

AltosRust Project, Version 0.1

AltosRust Team
15 March 2017

Table Of Contents

Introduction

Purpose

The purpose of this document is to outline the approach the AltosRust team is taking to creating and performing tests for the AltOS Rust operating system, and to describe the details of tests being created. This document is intended for software developers on the AltOS Rust project. The document describes the methods used to perform tests, the features of the system being tested, and the requirements and instructions necessary to run tests.

References

AltOS Home Page – http://altusmetrum.org/AltOS/
Toolchain Guide - https://github.com/AltOS-Rust/altos-rust/tree/master/rust-toolchain

Definitions

AltOS: The current operating system for Altus Metrum components implemented in the C programming language.
AltOS Rust: The working name of the software system to be developed. This also refers to the project as a whole. This is also referred to as the *target system.*
Cortex-M0: the ARM processor used in the STM32F0xx family of microcontrollers.
Rust: A systems programming language that supports compile-time memory safety, speed and concurrency.
STM32F0xx: A microcontroller family produced by STMicroelectronics.
Task: A self contained unit of execution isolated from any other task except through the access of shared resources.

Testing Approach

The system is designed to decouple as many of the modules as possible so that each module can be tested in isolation. Helper modules have been created in order to ease the process of writing tests in both the kernel layer and the portability layer. Unit test exist for as many of the module level functions as possible, and provide automated system tests in order to verify the behavior of more tightly coupled modules. For the hardware register representations in the portability layer, test are written for each function on the register, these tests only verify software behavior, such as turning on certain bits of a register. Other hardware specific behavior cannot be tested by our automated testing suite. In order to verify hardware behavior we write test tasks which can be run in order to produce some expected output.

Types of Tests

The target system will be tested using a combination of manual testing, performed by developers, and automatic testing, using various forms of tests written in the Rust programming language. Automatic tests cover a range of features of the system and attempt to test at different levels (individual components in isolation vs many components operating together). Automatic tests are intended to quickly verify that functions and modules within the system operate in a way that matches the desired functionality.

Unit Testing

These tests should be entirely automatic and should cover individual functions within the system. Unit tests should check the behavior of individual functions in isolation, ensuring that they perform their intended operations correctly. These tests do not rely on separate functions or parts of the system. Each unit test verifies one function, however a function may have multiple unit tests associated with it.

System Testing

These are automatic tests which cover larger parts of the system as they are integrated with each other. Unlike unit tests, these are not meant to test individual functions in isolation, but the interaction of multiple functions or components within the system. These should test the functionality of the entire system, to ensure that it functions appropriately when components of the system are integrated together.

Acceptance Testing

Test plans are written to provide steps for manually testing functionality of the system. This is used for manual testing, and allows for verification by multiple sources. The input, actions taken, and expected output are provided to make the tests repeatable.

Test Description

This section should describe the testing of features of the operating system from a higher level. We do not attempt to document every test in this section, but to describe the testing strategy for individual components of the system, as well as details on how those tests will be conducted and what is required to conduct them.

Sync Module

The sync module provides primitives for synchronization of tasks. These are mostly system level tests as they interact with the scheduler and the currently running task. They ensure that the behavior of tasks with regards to these synchronization primitives is correct.

Task Module

The tests in this module are mainly unit tests focusing on the task handle to verify that the behaviors of its functions are correct.

Scheduler

The scheduler uses system tests to verify that tasks are scheduled with the proper behavior.

Syscall Module

These tests verify that system calls have the desired behavior on the state of the kernel. The state of the kernel is tested through system tests, as each call modifies state of the system.

Queue Module

The queue module provides basic queuing mechanisms for use in the kernel. The queuing mechanisms are unit tested as they require no outside dependencies. Each collection provided has a suite of unit tests to verify their required behavior.

Hardware Registers

The portability layer defines a helper module used to automate testing of the board's hardware registers through software representations. In this way verification can be performed on hardware registers that could not be done so otherwise.

Environmental Requirements

This section describes details on the hardware and software used to conduct tests, and tools being used in the testing process.

Software

The nightly build of the Rust toolchain, along with Cargo.

Hardware

Nucleo-32 board for running hardware tests

Running Tests

To run unit tests, navigate to the altos-rust folder within the altos-rust project and type 'make test'.