# Tock Operating System Hardware Continuous-Integration Testing

Arvin Lin (a7lin@ucsd.edu)
Jack Sheridan (j1sherid@ucsd.edu)
Jefferson Chien (jkchien@ucsd.edu)

## Abstract

Tock is an embedded operating system that supports more than 20 microcontroller models and has a growing community of developers. Currently, those developers need to test each supported board individually for every biannual software release. Our hardware CI system aims to streamline the testing process and allow for real time automated testing by creating a top level system to manage many individual test harnesses. Our design is scalable, configurable, and modular. Anyone can now instantiate a new test harness, configure what should be tested on the chosen board, and connect it to our larger test network to trigger true hardware tests on every change of the codebase.

## Background

"Tock is an embedded operating system designed for running multiple concurrent, mutually distrustful applications on Cortex-M and RISC-V based embedded platforms. Tock's design centers around protection, both from potentially malicious applications and from device drivers". [1] It is an open source software used for lightweight applications such as sensor networks and wearables. It currently supports more than 20 microcontroller units (MCU), and the growth to support different MCU is highly anticipated.

## Introduction

The development cycle of TockOS lasts around 6 months, and developers usually only start to test the compatibility and the correctness of the pre-released changes right before the official release. However, this development cycle introduced a significant number of issues to developers. The untested and untracked changes made throughout the year increased the difficulty to diagnose problems, and it forced developers to retrace all changes to the TockOS codebase during the development cycle to locate the problematic code. As a result, developers can spend a tremendous amount of time identifying the origin of the bug on one single MCU, and since Tock supports more than 20 MCU's, developers sometimes even had to track down different bugs on different hardware models.

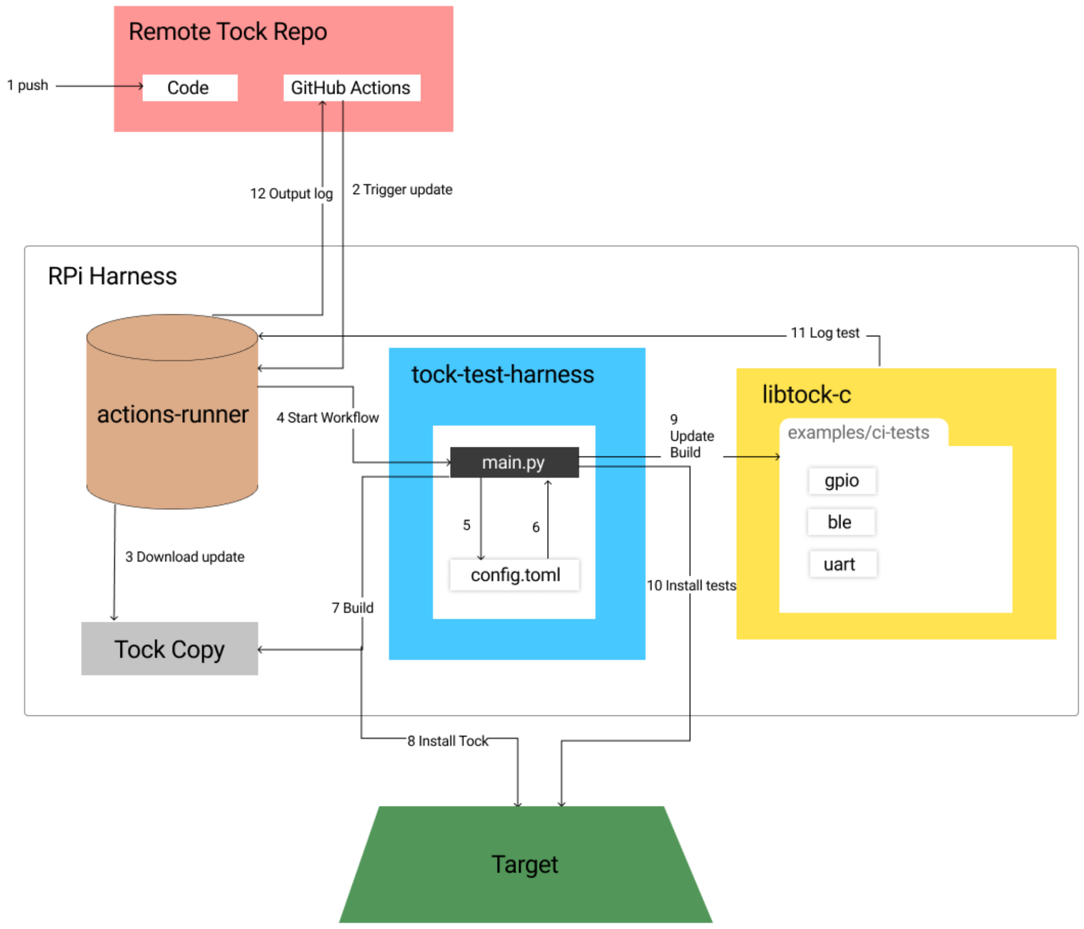
Testing each change and feature throughout the year potentially solves the aforementioned issue of the current development cycle. However, in order to do so, every developer would need to have all supported MCUs and manually flash Tock onto each of them every time they run tests. Moreover, each developer has a different environment, and having different settings scattered throughout the community can introduce more variables in the stability of the tests. Eventually, the anticipated time each developer spends to test the correctness of their changes greatly dictates and overwhelms the time spent on development of features. The tradeoff between testing and development could potentially impose more issues and slow development drastically.

Continuous-Integration (CI) testing solves the current problem without introducing new issues and tradeoffs. It provides immediate feedback and test results as soon as developers make changes to the Tock OS, and developers don't need to obtain equipment and spend time setting up the test environment themselves. An ideal CI testing system for Tock takes away the work from individual developers and creates a centralized, controlled environment to test new changes, and it should provide test results of all supported hardware to the developers, assisting them to identify and to resolve the bugs right away. To achieve this, we have created a harness system running on a Raspberry Pi and connected it to the remote Tock's codebase and the supported MCU - in our case, we are using a Nordic nrf52840dk as a proof of concept. We implemented a scalable design in the harness system, which will benefit future developers to set up new Raspberry Pi harnesses, and we also utilized the Tock's C library to create CI tests, which will be executable in all Tock supported MCU's. The creation of this end-to-end hardware continuous-integration system for the Tock embedded operating system greatly accelerates the Tock development cycle through real-time test automation, universal hardware tests, and a scalable design to multiply Raspberry Pi harnesses in the future.

## Technical Material

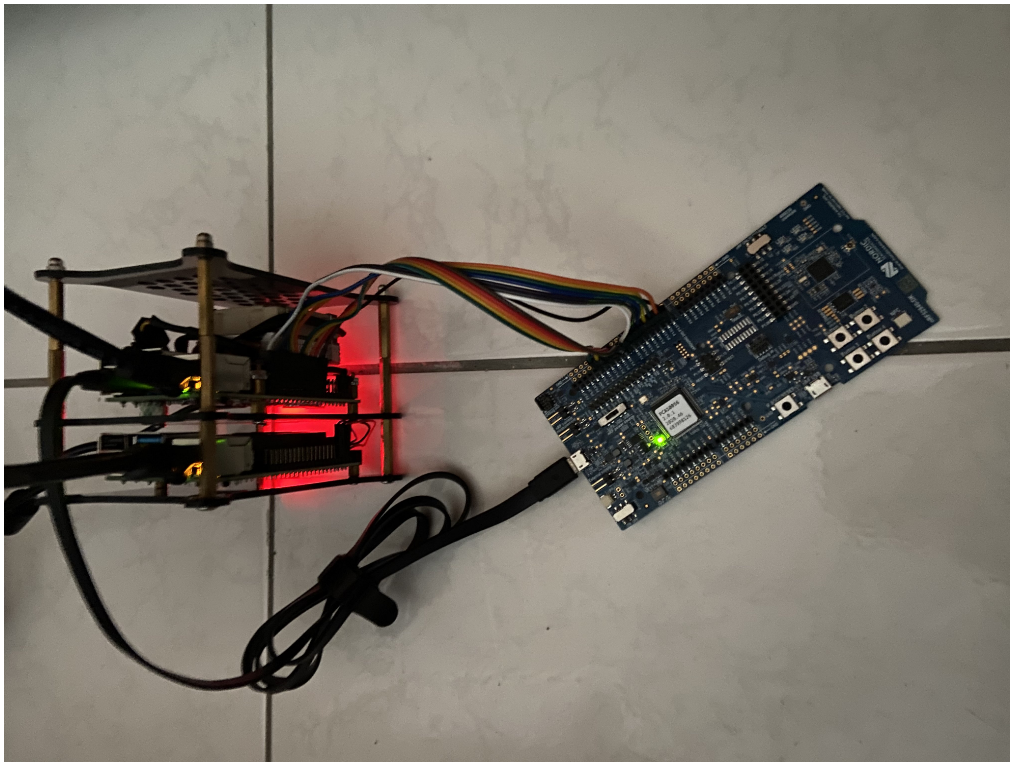
### Design

This section will be a more in-depth description of the design of our system. The best way to describe each moving part is by running through the entire process of one "commit" that starts the Github workflow. Here is a top level workflow diagram of all of the steps that are completed during one testing cycle:



We will enumerate each step and describe its function in detail below and describe how the entire design provides automation, scalability and configuration in later sections.

1. Code is pushed or accepted via a pull request.
    - This is the initial “trigger” to the entire system, and provides the true definition of “continuous integration”
2. Github servers send a message to each Raspberry Pi.
    - This is done through Github’s own system called a “Github Action Runner.” It is configured locally on the Raspberry Pi and acts as a web server that maintains a connection to Github at all times, listening for a change to the single repository it is tied to. This “tying” is done through a token that is provided during initial setup of the test harness and is described in our Getting Started Guide.
3. The Raspberry Pi starts by getting the most recent copy of Tock OS.
    - This is also handled through the Github Action Runner and there is a predefined folder on the Pi where the most recent commit is stored.
4. The Raspberry Pi starts its tests.
    - This is done via two parts: a YAML configuration file (which is stored on the main Tock repository in a folder called .github), and a python script that is on the Pi which is called by a command stored in the YAML file.
5. The Pi reads its local config file for information on what board is connected to it.
    - There are actually two TOML configuration files that are read in by Python during this step. The first one is as described, which stores local information about what target this Raspberry Pi is connected to, as well as a harness ID uniquely identifying this test harness. The second one is stored on the main Tock repository, which stores information about what each test-harness should be testing (defined by ID), as well as custom scripts
6. This information is parsed.
    - This is done via a simple Python TOML parser.
7. The Pi builds the version of Tock respective to its own embedded target board.
    - Using the information from the local configuration file, the script knows the path to the target board in the “boards” directory and therefore can compile using its respective Makefile.
8. The Pi flashes the newly built binary onto the embedded target board.
    - For our implementation, this is done using a “J-Link” application running on the Pi that can flash the operating system. For future scalability more flashing technologies should be used and tested.
9. The Pi retrieves the newest version of the tests and builds them.
    - The two-part tests are stored in the libtock-c repository in examples/ci-tests/. The two parts they consist of are: a python script that runs on the Raspberry Pi and actually does the unit testing, and a C file that is compiled and loaded onto the target board that acts as the “test app” that the test harness is testing.
10. The Pi installs and runs the test apps serially.
    - This is done via Tock’s own “tockloader” application that can quickly and easily install, run, and manage applications.
11. The Action Runner listens to standard out where the Pi logs the output.
    - Once the Github Action runner calls the main python script to run the entire system, it begins listening to the standard output stream for the Raspberry Pi. All output that the python application prints to standard out will be captured and uploaded in real time to Github.
12. Finally, the action runner sends the output and return code to Github where it is displayed in the "Actions" tab.
    - The live uploading mentioned in step 12 is how Github is able to display the tests happening live on the Github Actions Tab, and developers can see in real time on each of the boards whether their change was breaking.



An image of the Raspberry Pi connected to the target Nordic nrf52840dk via a Jlink USB and GPIO jumpers.

## Repository Overview

Our design consists of one main repo which uses two existing Tock repos.

- Tock Test Harness (Our main repo) (Owned by us for now until we officially launch the CI) which consists of:
    - lib: a folder containing all python code that runs on the Pi during testing.
    - runner_init.py: a script used in the initial setup of the Pi
    - config.toml: a configuration file created by runner_init that saves information about the target board for that Pi as well as a harness identifier.
- Tock: (The main Tock OS repo) (Forked for now until we officially launch the CI) from which we use:
    - /.github/workflows/ci-test.yaml: a configuration file that specifies to Github what to run and where (all of the tests for each Raspberry Pi)
    - boards: a directory we use to compile Tock for the target board as well as save board-specific test information in a configuration file called test.config.toml.
- libtock-c: (The main repo for Tock OS apps) (Forked for now until we officially launch the CI) from which we use:
    - /examples/ci-test, where there is a directory for each test.
    - These are general and can be tested by any board with the necessary features tested in the test. The name of a test directory can be uniquely specified in the "test.config.toml" file. Inside the directory is a test.py file and main.c file for the Raspberry Pi test and Tock test app respectively, as well as a Makefile to compile the test app.

## Automation

In order to accomplish automation for the HW CI Testing, we have incorporated native Github action runner, tock-test-harness repository, and libtock-c repository into a single complete workflow that automatically performs tests on targeted hardware boards. The results are automatically sent to Github through the Github action runner and display the results with a clear and concise manner. In addition to the implementation and configuration of the Github action runner, local testing scripts, and test code snippets, we have unified configuration toml files to truly achieve automation functionalities.

Firstly, to set up the automated testing workflow, a developer would only need to complete two manually steps:

1. Configure local configuration on the Raspberry Pi through the configuration script which would generated a config.toml file for local testing environment specifications
2. Setup Github action runner following a specific naming convention within the developer's developing community. Then simply run Github action runner with sysd, so that it could be run in the background.
3. (Optional) If the developer wish to carry out CI tests on hardware that is not included in the original TockOS repository or the developer wants to add more tests to an existing board, then the developer should add or edit the test.config.toml for each board

Then, the configuration files are separated into three parts:

1. Github action runner workflow configuration
2. Local testing environment specification config.toml file
3. Board specific toml files

The Github action runner workflow configuration would automate the entire workflow, and the config.toml file would achieve automation for the local test running for different boards. Finally, the board specific test.config.toml file would achieve specific tests for each board. Combining all three configuration files, our workflow is fully automated, highly flexible and configuration, and greatly scalable for more functionalities and features.

Lastly, documentation is an extremely important part of automating the TockOS CI workflow. We have insisted on clear documentation for specific hardware and software setup to ensure the automated workflow can be carried out smoothly. This includes specific pin setup for different features such as uart, gpio, ble, or spi. In addition to the hardware setup, software documentation and highly unified formats are important for a highly automated workflow. We have discussed and agreed on specific log formats, installation procedures, and building processes to ensure all previous scripts, configurations, and setup would work the same on every single hardware board that TockOS covers. Through the combination of all these implementations and design details, we have achieved a highly automated workflow which requires minimal human work, and allows for extensions, maintenance, and scaling. At the same time, its formats and constructions are simple and intuitive, which we believe all developers can understand quickly and easily.

## Configuration & Scalability

Our design allows for convenient but powerful configuration of each test harness. As seen in step 5 above, there are multiple configuration files that the user can edit to configure the system. We adopted a "source of truth" approach, in which the main test configuration file is stored in the main Tock repository, and editing that file is the only change that needs to be made because on each successive test cycle, the test harnesses will all query this file for the newest information the test manager has provided. This allows for little to know manual updating, because simple file edits are all that is needed to change what is tested on which board.

The more difficult configuration to edit or change is the local configuration on the Raspberry Pi. This had to be stored locally because it describes information that uniquely describes each Pi, and therefore could not be stored on the repository. We still wanted to provide convenience for this side of configuration also, so we opted to construct an initialization script that acts as a “wizard” for the creation of that local configuration file. It is called “runner_init.py” and it prompts the user for information and then immediately stores it in a local configuration file called “config.toml.” The reason we chose not to design a more complicated configuration that could be more easily editable is because unlike the test configuration in the repository, this local configuration will most likely never change in the lifetime of the test harness. There is always exactly one Raspberry Pi per board, and as long as that board is supported by Tock and therefore needs to be tested, the local test harness configuration can remain static.

Why is configuration even necessary? Important parts of testing are modularity and standardization. When a new test is created, we wanted the developer’s job to be as easy as possible, and we decided that would be simply adding an element to an array. There is no need for compilation and storage of the executable binary on the Raspberry Pi, nor any manual test distribution system, but instead a folder with the code in one repository (libtock-c), and a string element in a list of test in a configuration file in another repository (test.config.toml in the main Tock repo). This ease of use allows for developers to focus less time on manual arduous tasks, and more time on the development of new features and tests to test those features. Of course this works in reverse, if a test needs to be changed, updated or removed, the list of tests can simply be updated. All of this provides that first part of testing: modularity. Secondly, the notion of standardization can also be provided via this configuration. It is comforting for a developer to know that simply put: if two configuration files for two different boards are identical, they will both be tested on the exact same tests. This allows for that widespread parallel testing that was mentioned earlier in automation. Every test can be tested on every board, and the development cycle will not be halted on board-specific bugs that went undetected.

Even more configuration can be done in our “custom scripts” section of the configuration file. This is where users can specify some initialization script for the Pi that they have custom made and put on the Pi in order to initialize anything for a future test. Similarly, this can be done in post, which perhaps could be a cleanup script for some test that had been run. All of this customization and configuration allows for a developer centered testing experience, and values the community of researchers and professor’s number one resource: time.

The other benefit that comes from this customizable configuration is scalability. Right now, the over 20 boards supported by Tock can all be tested in parallel using our system. This is because of our dual configuration system with board identification mentioned above. However, to describe the benefit of scalability in more detail, each board model can also have multiple test harnesses of the same model testing different tests. This feature was added in case boards of the same model might have slight variations which can be individually tested. Also, because there is a one-to-one mapping between Raspberry Pi’s and target testing boards, the Raspberry Pi acts as a standalone “server” and can be placed anywhere in the world. Where usually a public research facility or university would have to host a lab with a server architecture to do any sort of hardware testing, these units can be placed anywhere with an internet connection. This also allows for the unique opportunity for proprietary hardware testing. If a company or organization wanted to look into using Tock OS, but has proprietary hardware they do not want to release to Tock developers, they can simply construct our test harness and connect it to the system, and our design allows for that abstraction layer where the hardware never has to be seen or touched by any Tock developer, but can still be tested on every codebase change.

## Milestones

1. Installation of Tock OS on nRF52840 dev kit
2. Compilation and installation of sample Tock OS testing App on nRF52840 dev kit
3. Ubuntu server setup on Raspberry Pi
4. Test Github runner on Raspberry Pi and nRF52840 dev kit

5. Setup GPIO testing software
6. Setup mechanism to run testing Apps on the embedded device
    a. Generate configuration files for test specification
    b. Create and design workflow to integrate config files into custom tests
7. ~~Record behaviors to generate logs which is then send back to Raspberry Pi~~
8. ~~Compile all logs, including compilation results and messages on GPIO, into a single compressive report on Raspberry Pi.~~
9. ~~Automatically upload CI results onto the Github repository under the CI event in a reasonable format.~~
10. Construct new tests or adapt more existing Tock OS test
    a. Custom UART test
    b. Custom BLE test
11. (Optional) Add colors to log and add different level to log: info, warning, error
12. (Optional) Setup embedded system boot controller
13. Design documentation and Installation guide

As instructed by Professor Pat Pannuto, our MVP simply consisted of a “closed loop” continuous integration implementation. Our milestones in the beginning reflected that, and we were quickly able to successfully construct a minimum viable product. After construction of the more basic static MVP, we decided to continue on to construct a more general design for more boards.

Some changes had to be made to our milestones once we had researched Github's continuous integration as well as designed our system less for “hard-coding” and more for dynamic scalability.

Milestone 7: Since the system monitor could not be done inside the target board, we decided that the Raspberry Pi should monitor the status of the target board externally. The target board should not send any logs to the Raspberry Pi, instead, the Raspberry Pi now monitors the target board and generates the logs itself.

Milestone 8 and 9: It turns out that the Github Action runner does not need to be passed a “report” or file at all, but instead passes all data written to *standard out* to Github. Due to efficiency and to keep specificity of each test harness, we decided not to specify each and every test in the top level yaml file that configures all test harnesses, but instead in a test configuration TOML file in each board folder. Therefore, these two milestones were deleted.

Milestone 10: This milestone is not necessarily changed, but we decided to add specification so that it is more explicit. We have decided to create a new directory called ci-test in the example folder of the libtock-c repository. This directory will have many unique directories which can be specified in each test harness’ TOML configuration file. Each of these folders will hold ‘test.py’ and ‘main.c’ files for the embedded app as well as it’s respective python test to run on the Pi. We also decided to complete the stretch goal of writing our own tests including a UART test and a Bluetooth Low Energy test.

We ended up completing all milestones except 11 and 12. The reason for not completing 12 was simply because we did not have the hardware or the time to create a true power cycling implementation. We wanted to be able to provide the functionality for a “simulated power failure” test by completely shutting off power to the target board. Milestone 11 was also deemed unnecessary as a priority because it is purely aesthetic and is not required for the functionality of the system.

## Conclusion

Hardware Continuous-Integration (HW CI) Testing for Tock aims to eliminate the arduous task of testing and retracing months of untracked works, and we have created a harness system running on Raspberry Pi to provide immediate feedback and test results when developers make changes to the Tock codebase. The harness system utilizes the existing framework in Tock heavily, including the Makefile system and the Tock C library, and this ensures that when Tock expands or changes in the future, the harness system can continue to support hardware continuous integration Testing. To setup a new test harness, developers simply install our harness system and the Tock environment onto their Raspberry Pi, and once the harness is set up, they can create their own test in the existing Tock C library and create some changes in Tock’s codebase to see if the new changes pass their own tests.

Although this project has focused more on the proof of concept, and we have only tested the system with one MCU, Nordic nrf52840dk, the system was designed with scalability in mind, and the overall architecture of HW CI Testing is a good starting point to expand to all Tock's supported MCUs. Soon enough HW CI Testing can be expanded throughout all supported MCUs, and developers won't need to experience the tedious debugging process they currently face.

## References

1. Amit Levy, Bradford Campbell, Branden Ghena, Daniel B. Giffin, Pat Pannuto, Prabal Dutta, and Philip Levis. 2017. Multiprogramming a 64kB Computer Safely and Efficiently. In Proceedings of the 26th Symposium on Operating Systems Principles (SOSP '17). Association for Computing Machinery, New York, NY, USA, 234–251. DOI: https://doi.org/10.1145/3132747.3132786