Bachelorarbeit

Muhammad Aiman Ismail

Automated Testing of the RIOT-OS Timer Subsystem

Fakultät Technik und Informatik
Department Informatik

Faculty of Computer Science and Engineering
Department Computer Science

Muhammad Aiman Ismail

Automated Testing of the RIOT-OS Timer Subsystem

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung
im Studiengang *Bachelor of Science Technische Informatik*
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Thomas C. Schmidt
Zweitgutachter: Prof. Dr. Zhen Ru Dai

Eingereicht am: 26. December 2020

Muhammad Aiman Ismail

Thema der Arbeit

Automated Testing of the RIOT-OS Timer Subsystem

Stichworte

Automated Testing, RIOT-OS, Timer, IoT

Kurzzusammenfassung

Ein Timer-Subsystem ist ein Grundbaustein eines modernen Betriebssystems. Netzwerkprotokolle erfordern präzise Zeitplanung, um Pakete in das richtige Zeitfenster zu senden. Sensoren senden periodisch Daten an einem zentralen Server. Das Testen ist kritisch, um diese Anwendungsfälle zu erfüllen und ein fehlerfreier und korrekter Systemablauf zu halten. Das manuelle Testverfahren kostet aber viel Zeit. Ein automatisches Verfahren für das Testen kann diese Kosten verringern und den Entwicklern für andere wichtigere Aspekte Zeit lassen. In dieser Arbeit kategorisieren wir die Probleme und entwickeln daraus eine Sammlung von Test-Suites, die automatisch ausgeführt werden können, um die Probleme des Timersubsystems zu lösen. Diese Test-Suites werden auch für die Performanzmessung von den verfügbaren Timersubsystemen in RIOT-OS benutzt. Die Ergebnisse können für die Evaluierung von Design-Entscheidungen benutzt werden und um die Performanz der Timer weiter aufzubauen.

Muhammad Aiman Ismail

Title of Thesis

Automated Testing of the RIOT-OS Timer Subsystem

Keywords

Automated Testing, RIOT-OS, Timer, IoT

Abstract

A timer subsystem is one of the building blocks of modern operating systems. Network protocols require precise timing to allocate packets in the correct slots. Sensors on the field periodically send data to a central server by the virtue of a reliable timer system. To achive that, testing is crucial to make sure that the timer behaves correctly and avoid bugs. However, manual testing takes a lot of time. Therefore, automated runs of those tests are necessary to free developers to focus on other things. The work in this thesis looks at the existing issues in the currently available timer subsystem in RIOT-OS and design tests that can be run automatically to detect these issues in the future.These test suites are then used to compare the performance of available timer subsystems in RIOT-OS. This helps evaluate the design decisions that are taken in each of the timer, which can be used as guidance to further improve the performance in the future.

Contents

List of Figures

Acronyms

API application programming interface.

CI Continuous Integration.

DUT Device Under Test.

GPIO General Purpose Input/Output.

HIL Hardware-in-the-Loop.

OS operating system.

PHiLIP Primitive Hardware in the Loop Integration Product.

1 Introduction

The timer subsystem is one of the crucial parts of every operating system. Systems depend on a reliable timer to provide an accurate value to their users. Sensors rely on low-level communication protocol, which in turn, requires robust timers to make sure that it can interface with other devices correctly. In a real-time system, it is important to make sure that a network packet arrives at the destination before the deadline. Should the packet arrive later, it can at worst cause unrepairable damage. Therefore, a component as crucial in the timer subsystem should be thoroughly and continuously tested to make sure its quality does not regress for every change introduced into the codebase.

Manual testing however requires a tremendous amount of time should it be used to achieve the above goals. The solution, therefore, is to have the test run be automated thus reducing the cost only to the initial development effort and maintenance of the automation infrastructure. Test automation also comes with other features such as regression detection for every new change introduced into the codebase.

This work aims to use automated testing methodologies to provide thorough testing for the RIOT-OS high-level timer subsystem. Other than the core part of the RIOT-OS kernel, almost every other component in RIOT-OS relies on the high-level timer subsystem. This means a bug in this subsystem can causes errors in lots of parts. In this work, we are focusing on the automated testing of the high-level timer.

This work is structured as follows: in Chapter 2, we provide some background knowledge of the components that are used. We described the reasoning why testing is needed, the stages of testing, the importance of automation in the testing field and introduces the concept of hardware-in-the-loop testing. Then we look at RIOT-OS and how the timer subsystem is structured.

In Chapter 3, we look at what others have done in the field of testing with a focus on timer testing, test automation. Here we also look at the history of HIL testing, how it is used, and its goal.

Next, in Chapter 4, we start our work by looking at existing issues that have been reported on the timer subsystem in RIOT-OS. This is done by collecting all issues related to timers from the RIOT-OS issue tracker and categorizing it to provide a general idea what is the common problem that is encountered in the existing implementation. Based on this information, we concluded several categories of bugs that can occur. These categories are then used as a guideline for us to design the test suites for the test automation.

This approach is explained in more detail in Chapter 5. We described how the end-to-end testing process looks like. This includes the infrastructure that is needed for the automation, how we interface with the Device Under Test (DUT) to run the tests, and how we extract the data for the result analysis. Furthermore, we list the test suites that we have come up with. These test suites are written based on the conclusion that we have made from our analysis done in Chapter 4.

In the next chapter, we evaluate our testing setup and how well it has worked. We look at the process of writing a testfrom start to finish. Then, we look at the empirical test result and look at the comparison of the performance for each test across multiple implementations. We provide a comparison between two timer subsystem implementation. Other than that, we look at the results when using the test setup in a pull request. This would be useful for usage in the application development pipeline as part of the continuous integration step.

Lastly, in Chapter 7 we conclude our work by summarizing the works that we have done.

2 Background Knowledge

In this chapter, we will look at the base components for the automated testing of the RIOT-OS timer subsystem.

2.1 Testing

Why test software?

Software composes a large part of our daily lives. Almost every object in our surroundings has software built into the system. Therefore, a bug in the implementation could affect our experience when going about our daily lives. Glenford et al. define testing as the process of executing the program with the ultimate goal of finding errors [1]. It is a given that every software can have bugs either due to design problems, environment errors, or a simple typing mistake by the programmer. Therefore, it is considered a mandatory practice to test the software before it is used in production.

Software testing also gives us confidence that our software will work as intended and most importantly, does not do what it is not supposed to do. In other words, through rigorous testing, we want to make sure that our software is **correct** and **robust**. Correctness can be defined as the software that satisfies the requirements outlined by the specification. Meanwhile, Kropp et al. define robustness as the degree to which it functions correctly in the presence of exceptional inputs or stressful environmental conditions [2].

Though with that said, we must remember that we can never be sure that *all* bugs are found. We can find 100 bugs, but there might still be 100 more but at least by finding the first 100 bugs, we eliminate the possibility that these bugs will affect us in the future. In the same line of thinking, if we found no bugs after a testing round, that doesn't mean that our program is free of bugs. It just means that our testing does not uncover any. Our test might not be effective enough, we might be testing the wrong components.

The testing process will still need to be done as part of the overall software development process.

To summarise all that, through testing, we want to gain the confidence that when executed, our program will function as intended and, in the presence of invalid inputs, it does not goes haywire and sabotages the whole system.

How to test software?

The software testing procedure can generally be broken into three steps: (i) test case design (ii) test execution (iii) result analysis. This section explains what each section means and what it entails.

Test design

In the first step, we must find out what and how we want to test our DUT. There are numerous techniques available for use but generally, it can be divided into two categories which are black-box and white-box techniques.

Black-box techniques employ the external description of the software. This can be the application programming interface (API) specification and requirements. Examples of black-box testing techniques are boundary value analysis, equivalence partitioning, and classification tree. In all three examples given, the techniques manipulate the input to be passed onto the DUT according to some specification. This usually catches common holes in the test case when writing the tests, especially for less-experienced developers.

White-box techniques, on the other hand, leverages pieces of information from the software such as the internal structure and design. The most popular technique of this category is code coverage. There are multiple variants of code coverage techniques, one is test coverage where the total lines of code executed at least once by the tests are compared to the total lines of code. However, this is proven [3] to not be enough. Modern white-box testing techniques usually use the internal code structure such as the syntax tree to parse the decision testing coverage. There are also other techniques such as mutation testing where the source code is manipulated using specified rules to intentionally introduce errors into the code.

Test execution

Now that we have our tests designed, we need to execute them. Test execution can either be done manually or automated. Manual test execution requires that the developer executes the test cases one-by-one by hand. This is usually a very involved process and requires a tremendous effort from the developer.

That said, this approach is not all bad. The initial overhead of manual testing is less compared to automated testing. Automated testing normally requires bigger investment upfront, in terms of man-hour, to develop the test automation setup. Therefore, manual testing is still reasonable for use during the initial development process to get things running fast. Later when the initial development is finished, it is recommended to automate the test execution. We will discuss more on automated test execution in the next section.

Results analysis

To gain value from all the testing that we've done, we must then analyze the results. Here a **test oracle** is used to determine if our test fails or passes based on the output of the executed tests.

In its simplest form, a test oracle consists of a tester manually verifying that when given an input, the software produces an expected output. Barr et. al [4] outlines the importance of test oracle automation to overcome the human bottleneck in the testing process. They also categorized existing test oracle approaches into specified, derived, implicit, and human categories.

Automated test oracles usually make use of documentations, formal specifications, and design documents to decide whether a test passes or fails. On the other hand, human test oracles are based on the experience, checklist, or gut-instinct of the stakeholders.

Automated Testing

Automated software testing can replace the role of human testers in designing and executing the test cases [5, 6], freeing up the resources for the software developers to improve on other aspects of the software.

As stated before, the testing process requires laborious work when executed manually. For a small set of tests, manually executing these steps is still doable and in the initial development phase, it might even be desirable to get fast feedback. After the initial development phase is over, however, that effort could be better used for other development aspects where the direct involvement of the developer will return more profit.

Each of the test stages can be automated. Traditional test case design techniques usually take a lot of time. The entry-level is also high as experience is needed to produce high-quality test cases. Modern techniques such as model-based and mutation testing now automatically generate test cases based on the information from the source code. Other techniques such as fuzzing and evolutionary testing go a step further by using the feedback loop mechanism where the next input is manipulated based on the last output.

Nowadays, there are plenty of tools and frameworks that can be used to automate test execution. Unit testing frameworks like PyTest and JUnit encourages the developer to write repeatable tests. Once a test is written, the execution can be automated. Even though in practice it can do much more than just executing tests, Continuous Integration (CI) tools such as Jenkins also suitable for automated test executions.

One of the areas where automated test execution is useful is regression testing. Regression testing is a practice in the software development process where the tests are rerun after new changes are introduced into the codebase to make sure that all the existing test cases still passes. It depends on the tester whether all of the tests or only a subset of them are run.

Automated testing is used as part of CI practices. CI is the process of automating the build and testing of code each time a new change is introduced into the codebase. With this, the developers can get early feedback on their changes and as a result, can deliver software faster and with lesser bugs [7]. Normally, the CI process comes included with a preconfigured environment through the use of containers. This also significantly boosts the reproducibility of the tests. No more situations where a test case passes on one machine but fails on another due to differences in their environment.

2.2 Hardware-in-the-Loop

HIL is a branch in the wide testing field that has been around for a long time [8]. It requires the involvement of real hardware in the simulation [9].

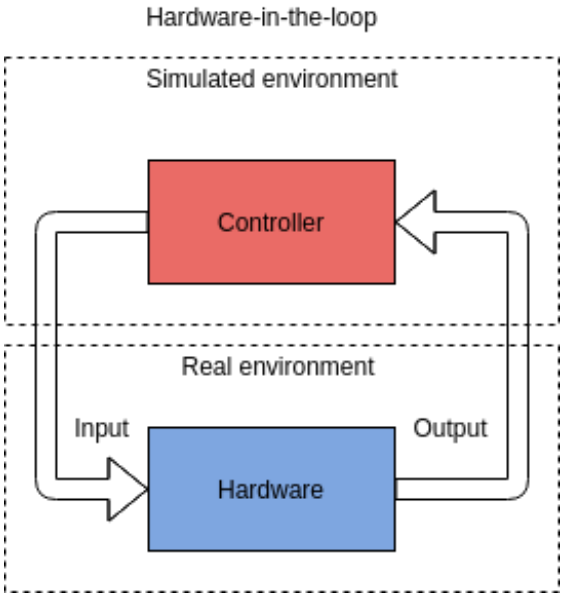


Figure 2.1: HIL testing structure

In the simplest form of testing, we would run our test on the real hardware and observe the effects directly when possible but this methodology is not suitable, especially when the DUT operates in an environment that is not easily observable or dangerous [10]. Sometimes, full hardware testing also limits the tester to only the local environment such as the weather or other uncontrollable factors, which makes it harder to trigger edge cases and reproduce the test cases.

Therefore, an approach was developed where *a part* of the system is modeled in the software. It is important to note here that only *a part* is modeled in software because modeling the whole physical world in software is not feasible as that is either outright impossible or will involve a significant amount of labor. Regardless of the work required, this approach still provides low guarantees that a fully passing software simulation will not fail when tested on the real system.

Here is where HIL simulation comes. In HIL simulations, the software is now tested alongside real hardware as shown by Figure 2.1. The software part can be the interface for the model while the execution of the test is now on the hardware [9]. This eliminates the extra layer of uncertainties generated when modeling the physical world into software [11]. It will be useful to include HIL simulation as soon as the implementation is started. This way, the implementers can start using it already for unit testing, and later it can be

expanded to include integration testing before the system verification is done with the environment.

2.3 Timer

A timer subsystem is crucial to any embedded system. It is the building block for many other operating system (OS) components. A timer can be used to record the timestamp at any given moment. It can also be used for scheduling a task to a later time, firing only once or periodically.

The networking system in RIOT-OS [12] relies heavily on a robust timer subsystem. A reliable timer will make sure that a timer will expire after a set period and trigger the predefined callbacks. With an unreliable timer, the callback might not be called in a timely manner, causing the packet to be sent after the deadline has passed. This might not cause any apparent issue at first but after the issue accumulates for a while it can lead to bugs that are hard to debug. Another use case where the timer system is crucial is in energy harvesting operations such as the work done by Rotthleuthner et. al [13]. Continuously sampling a will incur performance penalty on the DUT. Therefore, a timer is used to periodically trigger the measurements. However, if the timer incurs too large of an overhead, it might influences the measured sample, thus harming the validity of the experiment.

Figure 2.2 shows an example architecture of a timer subsystem. A timer subsystem usually consists of multiple abstraction layers that sit on top of each other. On the lowest layer, a clock is provided by hardware oscillators or any frequency generating hardware on the DUT. Users interact with the clock hardware registers through a communication protocol. On top of that, we have a low-level timer API that wraps the hardware register access in modular low-level functions. Above it, we have the high-level timer API that uses the low-level timer API to provide a more user friendly and easier to use timer API.

Each of these layers consumes the CPU processing cycles, thus adding overhead each time it is used. Therefore, a good timer subsystem should minimize the latencies due to all of the above factors. On top of that, there is also a variable duration added to the overhead due to the layers being scheduled by the OS scheduler. This added overhead will affect the timer, causing its sleep duration to deviate slightly for each run from the

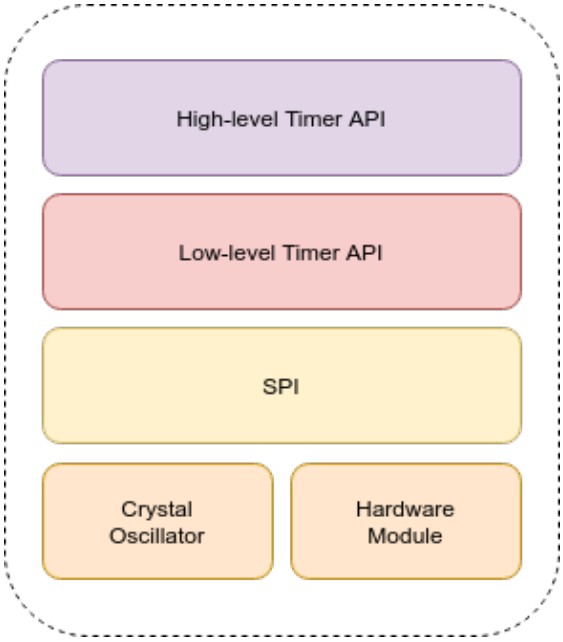


Figure 2.2: Example of a timer subsystem architecture

target duration· A good timer subsystem should be stable and its result should not have a large variation.

Other than that, each clock will inevitably drift from the supposed value. This phenomenon is already taken into account by the manufacturers of the crystal oscillators and each of the oscillators comes with a specified accuracy. However, developers must also be careful not to implement the timer in a way that will significantly affect the specified accuracy.

To summarise this section, a good timer should have (a) minimal latency, (b) small variation in its result, and (c) does not significantly reduce the specified accuracy of the underlying hardware module.

2.4 RIOT-OS

RIOT-OS [14, 15] is an open-source operating system for microcontrollers. It has a modular design where each of the subsystems is configured as modules and can be selectively included during compile time through the Makefile-based build system. This guarantees minimum memory usage as only the required components are included and

the application developer can choose which of the modules to be included. The application can be programmed using either C99 or C++11, without any special syntax. Furthermore, RIOT-OS supports multitudes of microcontrollers and architectures. This flexibility makes it among the popular choices for researchers doing experiments on their field of interest [16, 17, 18].

RIOT-OS currently has two timer subsystems in its codebase, that is the xtimer and the ztimer. Like any other software project, after a long period, there are always many new insights that are gained in regards to designing a better implementation of the software. Certain design decisions that are taken before might prove to be not the best choice. Armed with the new knowledge, members of the RIOT-OS community have come up with a new timer subsystem. ztimer is the successor of the xtimer subsystem, just like xtimer is the successor of the previous timer implementation that existed in RIOT-OS. The goal is to address the deficiencies of xtimer and provide a better high-level timer API for the users.

3 Related Works

In this chapter, we will look at the works done by others in the field of testing in general and more specifically automated testing and HIL.

3.1 Timer Testing

We look at existing works done on benchmarking and latency measurement of timer subsystems. We are also looking for interesting metrics to be considered for our quantitative measurements.

Abeni et al. [19] point out that latencies in a timer subsystem are caused by multiple factors. The first factor is the timer resolution. Timer subsystem based on periodic timer ticks operates at a defined ticks interval. Any event that is supposed to trigger in between those ticks will be rounded up to the next tick, thus producing the latency. The next factor happens when an event in the queue is ready but it might still have to wait before it is scheduled to run. Abeni et al. call this the scheduling jitter. The last factor is non-preemptable code sections. The scheduler will have to wait for the section to finish before it can schedule a timer event to run.

These three factors can be measured in isolation by taking measures to eliminate other factors when measuring the latency caused by one factor. An example of this is to measure latency caused by timer resolution, we can set the highest priority to the measurement thread and run our test on an idle system so that the scheduling jitter and non-preemptable code section will not affect our results. This will help in pointing us to the components that are causing the issues more accurately.

That said, to analyze the performance of the timer, we think that measuring the latency as a whole and not in the granularity suggested by Abeni et al. is enough as this is the same latency observed by external components.

Fröhlich et al. [20] investigates the performance of periodic timers against a one-shot timer and discovered that, given that the periodic timer is configured accordingly, the performance of both of the timers is comparable. As conventionally periodic timers are seen as lesser due to the added overhead, this gives us a fresh perspective to judge the implementation with. They also provide a test parameter — using the same sleep period as the timer frequency to get the ideal period where the timer resolution latency will be minimized. This is in line with what is suggested by Abani et al.

Costello et al. [21] measured the time to set and reset the timers as part of the evaluation of their redesign of the BSD callout and timer facilities. Current RIOT tests already have these tests implemented as a simple performance indicator of the implementation. Note, however, these measurement does not represent the timer latency, instead, they give us a quantitative value as to how efficient the timer handling is implemented. This information is useful when evaluating a design decision.

Lastly, Jupyung et al. [22] employed manual code analysis to identify non-preemptible sections of the code and then measured the lock hold time in these sections. Knowing the duration our program spent in the critical part of the code is useful to get an idea of the effect of non-preemptible sections our our program. In our timer subsystem, the `xtimer_now()` function contains a code section that is non-preemptable. Therefore, it is a suitable candidate for this analysis.

Other than the measurement mentioned, all of the works above employ the same three steps for measuring the latency — First, the current time is read and saved as the start time. Then, the process will go into sleep for a defined period with the timer facility under test. After the period ended and the process has woken up, the time is read again and the difference between the current time and the start time is calculated. These steps are repeated multiple times to get a better mean of the actual value and minimizes random errors that might occur during the experiments. For each of the tests, there might be a variation in the configuration e.g. the sleep period is set to a very long time to observe the drift.

3.2 Tools & Frameworks

Other than the benchmarking metrics, there are also components around the benchmarking itself to increase efficiency of the testing itself. Yumei Wu et al. [23] introduces the

Software Reliability Testing Application Framework (SRTAF) for testing the reliability of embedded systems and underlines the idea that embedded software testing should be automatic, real-time, closed-loop, and non-invasive. This framework contains the following steps: (i) test preparation (ii) test run (iii) test result analysis. Due to the nature of our test subject, we currently are not using an automated mechanism for the test preparation where test cases are generated but for the other steps, we have implemented a pipeline combining automation library and CI tools. Further information for this subject is described in Chapter 5.

3.3 Hardware-in-the-loop

Earlier HIL simulations were mostly used in high-funding laboratories such as the US army [8, 24, 25, 11]. They are used to test the active missile system, where real-world testing is limited due to the missile being high up in the air during use. Therefore, it is a good candidate for HIL simulations.

At the time, however, most HIL laboratories require expensive facilities. This might be due to the nature of the DUT e.g. missile system where specials have to be developed to generate the stimulus to test against the system. Later, Brennan et al. utilize this testing methodology albeit slightly different — using a scaled model of the hardware [26, 10] — to save on the cost of hardware. In places where suitable, common hardware is used for the simulation.

This approach is investigated by McNeal et al. where off-the-shelves hardware like computer soundcards are suggested as an alternative to specialized tools for testing a power protection system [27]. This effectively lowers the cost for implementing a HIL simulation system although the capabilities of the hardware component are limited as it is not the exact replica of what is used in the real system. The idea of lowering the costs of entry for a HIL system is continued by Lu et al. by using open-source software [28] as the base of its simulation system, thus avoiding licensing cost required when using proprietary tools.

In the embedded software field, Mozumdar et al. introduced HILAC [29], a framework for HIL simulation and automatic code generation for Wireless Sensor Network (WSN). They use model-based simulation built on top of the MathWorks toolchain connected to the real sensor node for the HIL component. The framework is not limited only to testing

the logic of the application, it also goes one step further, producing machine-generated application code that can be deployed on TinyOS based platforms.

4 Issues Analysis

To get insights on which component of the current timer system is problematic, we have analyzed the issues on the RIOT Github repository issue tracker. All issues in the repository labeled with `Area: Timers` are scraped and dumped into a spreadsheet and then analyzed.

4.1 Overview

Figure 4.1 shows the summary of our result from this analysis. The label in the chart is defined in Table 4.1.

We have in total of 78 issues labeled `Area: Timers` in the repository. At a glance, we can see that programming error is the major part of the issues with 35.1%. Then it is followed by misconfiguration which accounts for 15 issues total.

4.2 Programming Errors

Programming errors can be due to various factors which can be broken into multiple error categories:

- wrong comparison operator
- wrong return type
- off-by-one errors
- missing implementation
- implementation not according to spec

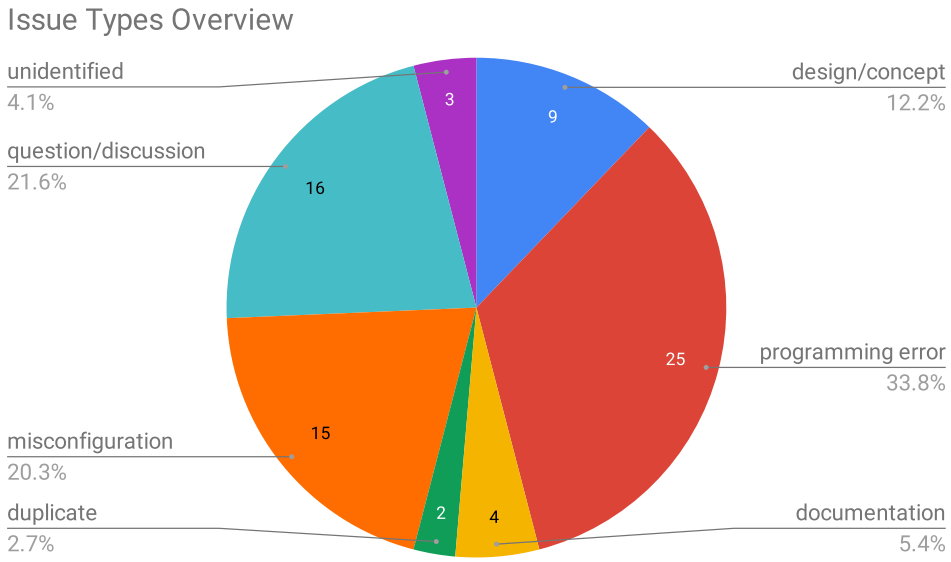


Figure 4.1: Breakdown of issues by dimensions

Table 4.1: Descriptions of the issues dimensions

Dimension	Description
design/concept	Errors that are caused by the design of the timer
documentation	Add/fix documentation
duplicate	The issue is a duplicate of another issue. See the related column for the original issue.
misconfiguration	Errors caused by wrong or nonexistent configuration of timer/board
programming error	Errors that are caused by the implementation failure
question/discussion	The issue is a question/discussion and not an error
unidentified	Error unknown/cannot be identified

In this work, we will not investigate programming errors. These errors are ideally discovered through unit testing each of the components as it usually does not depends on the execution environment. This means the execution of the tests in the native environment is enough. If that fails, automated testing methods such as coverage or mutation testing can be used to uncover holes in the existing tests and the result can be used as guidance to further improve the tests.

4.3 Design/Concept

Absolute vs. Offset-based

The most common issue aside from programming errors stems from the design of the timer itself. In the earlier version of xtimer, it uses an absolute timer target to know when to trigger the callback. This is problematic because xtimer only has one information to decide the expiry of a timer: target time, which is calculated once when a new timer is set. If due to some circumstances, the timer passes its target time without triggering, it will have to wait for another whole timer cycle before it will trigger. Due to this, we see issues where the timer seems to just hangs and stops the whole system for inexplicable reasons.

The new offset-based design uses two information, the start time and period. The absolute time is then calculated each time when comparing to determine when to expire. Even if the absolute target the timer is supposed to trigger has passed, it will trigger soon after because we know the timer period had passed its target sleep period.

These types of issues cannot be fixed with a small change. HyungSin did a rework of the xtimer in PR #**9530** to use the offset-based design. This PR fixes an array of issues previously reported such as #**8388**, #**5338**, #**7114**, #**5103**. However, we will also need to have performance tests to give quantitative metrics to judge the new design or implementation.

List Type

Next is the design choice for the timer implementation. xtimer uses a linked list to maintains its timer list. As is normal with a linked list, we will need to traverse through

the list, each time we want to access an element from the list. That means an increased number of timers will result in reduced performance of the timer. Finding out how the pattern for the performance reduction is useful to determine for our use case.

Interrupt Safety

In RIOT releases before 2020.01, the function `xtimer_now()` is not interrupt safe. During the execution of the function, an interrupt could cause the timer to be rescheduled. As a result, the returned time is no longer valid as there is now a delay between the time when the function is called or the time is read. This is seen in issues #**5338**, #**7114**, #**8388**, and #**6442**.

Multiple Timer Triggering at the Same Time

Another possible issue is when we have a lot of timers that are supposed to trigger at the same time. Due to hardware constraints, this scenario is will never occur. In reality, the timers will trigger sequentially, causing the later timer to be delayed. A performance test to quantize this delay would be useful for the assessment.

Hardware Misconfiguration

The same hardware with identical components can operate very differently depending on how it is configured. We identified common configuration errors and the critical configuration parameters that contribute to most problems.

Unsupported Default Configuration

Access to the timer peripherals on the boards causes delay, the amount of which depends on the specific hardware being used. If the user sets the timer to a duration smaller than the fixed overhead delay, this causes the timer to sleep more than the specified duration as now, the total amount slept is the sum of overhead delay and the duration sleep specified.

xtimer compensates for this by setting `XTIMER_BACKOFF` to the overhead delay of the specific board. In cases where the user-specified a sleep duration smaller than the threshold, instead of using the hardware sleep, xtimer avoids the overhead by actively waiting instead. Therefore, `XTIMER_BACKOFF` macro must be configured correctly for each board.

One common issue we see is that the board uses the default `XTIMER_BACKOFF` without testing if the value is suitable for the board. This can be seen in issue #**11523**, #**7347**, and #**9052**. With an unoptimized configuration value, the system might lose some performance. However, under heavy load, the system might stop functioning altogether, producing issues such as system hangs. Therefore, workloads depending on a small sleep period (<100 us) are recommended to be tested extensively before usage in production systems.

Other than `XTIMER_BACKOFF`, issues can also occur if `XTIMER_SHIFT` and `XTIMER_HZ` are not configured correctly as can be seen in issue #**6419**. Here we also observed the same symptom, where the system will stop responding after a while.

4.4 Other Issues

Drift

Drift is an inherent element in any clock and it is normal as long as the value stays in the acceptable range. Running the system for a long time, or sleeping for an extended period may induce a significant drift.

In #**9049**, this causes the sleep time to vary a lot when sleeping for a longer period. In #**5103**, missed timer deadlines caused the system to sleep for multiple seconds longer than it is supposed to. In #**10523**, the system time measured using `localtime()` differs to about 15 seconds in one hour. In #**6052**, when running on the `native` platform, the test application `xtimer_drift` hangs after running for about 30 to 90 seconds. The same is reported in #**6442**.

4.5 Conclusion

The most common issue experienced by the user is that the timer stops responding after a while and needs a reset to start working again. In other cases, the time returned by `xtimer_now()` is delayed significantly, therefore causing other functions depending on that function to return the wrong results.

Based on the explanation, the following is the identified most failure-prone component from this analysis in no particular order:

- timer misconfiguration: `XTIMER_BACKOFF`, `XTIMER_HZ`, `XTIMER_SHIFT`
- very small sleep period
- long sleep period (drift)
- interrupt safety for timer operations
- missed timer deadlines

5 HIL Setup & Design

In this chapter, we will explain the overall design for our HIL setup component-by-component. In Section 5.1 we will explain the infrastructure that runs the automated tests. Next, in Section 5.2 we described the steps required to write a new test and the tools needed. Lastly, Section 5.3 describes the test suites that we have defined for our HIL setup.

5.1 Infrastructure

A good infrastructure setup is critical to any automated project. Making sure that the infrastructure is maintainable over time is even more critical. That is why we opt for a combination of off-the-shelves software that is popular among the open-source community such as Jenkins along with custom software designed by members of the RIOT-OS community, Primitive Hardware in the Loop Integration Product (PHiLIP). This combination gives us the longevity of popular open-source software for the general infrastructure while also satisfying our specific need for embedded hardware testing.

Jenkins

Jenkins [30] is an open-source automation server used by a large community of users. Due to its popularity, Jenkins has extensive documentation and there is a large collection of plugins written to support a multitude of use cases.

For every test run, a user can specify which RIOT version, which tests to run, and which boards to test on. After the user has finished configuring the test, Jenkins will send all the required source code to a build server. The build server will compile the source code and outputs a test firmware binary. The test binary is then sent over the network to the specific Raspberry Pi connected to the DUT. The Raspberry Pi then flashes the DUT

with the test firmware and starts the tests. Details on this are described in Section 5.1. To decrease testing time, each of the tests is run in parallel across multiple DUTs. When the test is finished, Jenkins collects all the results back and archive it for further analysis. A summary of the pipeline is shown in Figure 5.1.

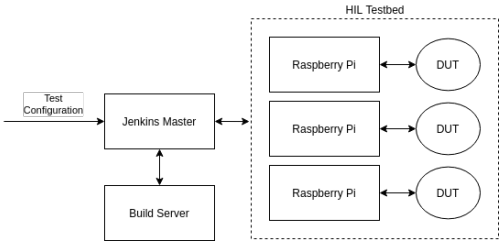


Figure 5.1: Test Pipeline Flow

Measurement Setup

For all of our measurements, we used external reference hardware running our novel firmware for testing peripherals called PHiLIP [31]. PHiLIP is a qualified firmware to test the peripherals of other devices connected to it. It can be used to inject specific peripheral behavior into the DUT and records the reaction. The firmware can be used with a raw serial interface, but it also provides a Python interface that abstracts the details when using PHiLIP for writing tests. This makes it suitable for both local development use and also automation using CI software.

Each of the DUTs is connected to a Raspberry Pi equipped with a bluepill board flashed with the PHiLIP firmware. PHiLIP provides multiple pins to control the DUT but for our usage, we are only using the input capture pin (`DUT_IC`) to record the timing measurement. Starting a timer in this setup is done by setting the General Purpose Input/Output (GPIO) pin to HIGH and stopping the timer is done by setting it to LOW. The `DUT_IC` pin specifies the accuracy of 14 ns.

5.2 Test Writing Process

Writing a test for use with the HIL framework requires multiple steps as summarised in Figure 5.2. The goal is to have a test firmware that is configurable through parameters, giving us more flexibility to implement specific test cases using a higher-level language.

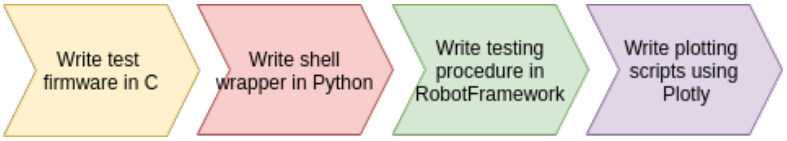


Figure 5.2: Steps to writing a test

First, we need to write the test firmware in C. Each test command is exposed through the shell of the OS. The test firmware specifies actions that can be executed but leave the specific parameters configurable through the shell. The result of the test is then printed on the shell. It is formatted with our custom schema so that it is easier to parse later with RobotFramework. The use of the shell as the testing interface gives us the ability to run the test manually, which is useful during test development, and also automated through the CI. It also enables us to reduce the writes on the flash memory of the DUT, thus increasing its lifetime.

Next, we need a wrapper written in Python for the shell commands. The wrapper gives RobotFramework access to the shell for our test description.

After that, we need to write the testing procedure in RobotFramework [32]. RobotFramework is a generic open-source automation framework. It is used to test a variety of software by many industry-leading companies. It has its own, human-readable syntax and can be extended using Python or Java. We use RobotFramework to specify the test cases and test suites described in Section 5.3.

In the RobotFramework description, we describe which test to run and the parameters for the specific tests such as the number of timers to use and the target sleep duration. Here we also parse the test result from the shell and write it to a file for analysis later.

Having all the result is meaningless if we cannot extract useful information for it. Therefore, as the last step, we will create diagrams that will help others understand the result better. For this, we used a Python library called Plotly. One of the advantages of this library is that it can generate interactive HTML diagrams. The users can zoom in to specific parts of the diagrams and also show or hide parts of the results. This can help users to focus on the specific area of the diagram.

5.3 Test Suites

In this section, we will look at the test suites that we defined for testing using the HIL setup defined above.

Our current test suite is built upon existing testing work done by the community. It is extended for usage in our automated HIL setup. We also added new tests to further extend the test suites. The following describes each of the test suites in detail.

The following is a summary of all the test suites defined:

- Overhead
  - GPIO toggle
  - Timer now
  - Timer set
  - Timer remove
- Sleep Accuracy
- Jitter
- Drift

5.3.1 Overhead

This test suite is designed to provide metrics on the performance of the timer subsystems. It consists of a combination of small tests designed for a specific aspect of the timer.

First, we measure the overhead of setting and clearing the GPIO. In our setup, setting and clearing the GPIO is used to start and stop the external reference timer. Therefore, this mini benchmark will give us information on how much overhead our testing setup adds when doing the measurement. We set and clear the GPIO pin repeatedly in a loop.

Next, we measure the overhead of the TIMER NOW function. In this benchmark, we measured the time taken to call the timer facility providing the get current time function in the timer subsystem. It is usually used when calculating the difference between the

start and finish time of an operation to get the elapsed time. By taking into consideration how long the delay in calls of the TIMER NOW function, users can consider whether this satisfies their requirements. In xtimer, this function is called `xtimer_now()` and in ztimer `ztimer_now()`. The measurement is started just before calling the TIMER NOW function and stopped directly after the TIMER NOW function returns.

After that, we measure the overhead of setting and removing timers. Both of the two implementations of the timer subsystem available in the RIOT-OS use a linked list internally to manage its timers. This benchmark gives us an overview of how the timer subsystem scales when tasked with managing a lot of timers. We measure the time taken to set or remove the nth timer in the list where $n = 1, \ldots, N$ and $N = 25$.

For measuring the time taken to set a timer, first, we set $n - 1$ timers, setting the offset longer than the previous timers, so that the last set timer is always the last in the internal timer list. Then, we start the external reference timer and set the Nth timer. We stop the external reference timer after the function setting the Nth timer returns.

For measuring the time taken to remove a timer, it is similar. First, we set N timers. Then, we start the external reference timer and removed the Nth timer. We stop the external reference timer after the function removing the Nth timer returns.

5.3.2 Sleep Accuracy

This test suite is designed to show the difference in sleep time when the DUT is tasked with sleeping for a short duration. Ideally, when a timer is set to sleep for a given duration, it will sleep for exactly the given duration. However, in the real world, this is not the case. Setting a timer to sleep consists of doing parameter validation checks, reading the current time from the register, and adding the timer to the list. Each of those steps has its overhead. In this benchmark, we want to see how long does the timer actually sleeps when set to sleep for a certain duration of time.

We defined the sleep duration set by the user to the timer as the target sleep duration. The duration that the timer actually sleeps, measured by the external reference timer is defined as the real sleep duration.

Large sleep duration is normally not affected by timer overhead as it covers only a small portion of the whole sleep duration. In a really small sleep duration, however, we might

found that the overhead is much larger than the target sleep duration. For that reason, we specified durations from 1 µs to 100 µs as the target sleep duration.

Both our timer subsystems have a blocking and a non-blocking variant of sleep function, represented here as TIMER SLEEP and TIMER SET, respectively. TIMER SLEEP will block the process until the sleep duration ends while the TIMER SET continues the process and calls a user-defined callback when the set target duration is over. For TIMER SLEEP, we started the external reference timer just before calling the TIMER SLEEP function and stops the external reference timer after the sleep function returns. For TIMER SET, we started the external reference timer just before calling the TIMER SET function and stops the external reference timer when the callback fires.

5.3.3 Sleep Jitter

In the real world, every operation on hardware can vary for each run. Abeni et al. [19] stated that the sources for these variations can come from process scheduling, timer resolution, and non-preemptable sections in the systems. All three components can affect the timer in use, giving us varying results each time our measurement is done. Therefore quantifying it will give the users an idea of what to expect when designing their application on top of RIOT-OS. In this work, we define jitter as the variations in the wakeup time across multiple sequential measurements when tested with different numbers of simultaneous timers. The following paragraphs in this section will define this more formally.

For our measurements, we set up a timer to trigger at a fixed interval of 10 ms. The start time is recorded and relative to it, the timer will calculate the target wakeup time for the next iteration. After sleeping for the given interval, the timer will wake up and calculate the target wakeup time for the next iteration. This is defined in Equation 5.1:

$$T_{target_{n+1}} = T_{start} + (n \cdot T_{interval}) \tag{5.1}$$

where $n = 1, \ldots, N$, and N is the total number of iterations chosen in Section 5.4, $T_{target_{n+1}}$ is the target wakeup time for the next iteration, T_{start} is the recorded start time, and $T_{interval}$ is the specified sleep interval.

In reality, the actual wakeup time of the timer will deviate from the target wakeup time. To compensate for this, the current time is subtracted from the next target wakeup time to get the offset until the next target wakeup time. This is calculated using Equation 5.2:

$$T_{offset} = T_{target_{n+1}} - T_{now} \tag{5.2}$$

where T_{offset} is the offset until the next target wakeup time, $n = 1, \ldots, N$, and N is the total number of iterations chosen in Section 5.4, $T_{target_{n+1}}$ is the next target wakeup time, and T_{now} is the current time.

To see how the timer performs when more timer is used, we run the measurement using up to 10 timers, maintaining N number of iterations for each timer count. For each iteration, all the timers are set to trigger at the same time and then we record the wakeup times of the last timer in the list. This is done to intentionally induce collisions between the timers and measure the latency of the last timer. The callbacks will actually be executed sequentially as RIOT-OS does not support executing two callbacks simultaneously.

For each number of timer used, we calculate the difference between the actual wakeup time and the target wakeup time of each iteration as shown in Equation 5.3:

$$T_{jitter_m} = T_{wakeup_{(m,n)}} - T_{target_{(m,n)}} \tag{5.3}$$

where $m = 1, \ldots, M$, and M is the number of timers used, $n = 1, \ldots, N$, and N is the total number of iterations chosen in Section 5.4 and T_{jitter_m} is the differences between the wakeup and target time when using m timers. This means for each T_{jitter_m} we have n results.

For each timer count, m, in T_{jitter_m}, we will look at at the maximum and minimum result across results from N iterations. These metrics allows us to quantitatively assess the amount of jitter of the timer.

5.3.4 Clock Skew

Crystal oscillators are used to drive the clock on an embedded board. Manufacturers specify the operating frequency of the oscillators, however, depending on external factors,

the DUT clock can be running faster or slower frequency than specified. This test suite aims to show how this affects the behavior of the DUT.

This test suite is similar to the sleep accuracy test suite, albeit with a longer target sleep duration. We set the board to sleep for a specified target duration and then measure the actual sleep duration using the external reference clock. The target sleep duration is set to 1, 10, 20, 30, 40, 50, and 59 seconds. Ideally, longer sleep duration is used but due to the limitations of our measurement setup, we can only test up to 59 seconds. Then, we calculate the difference between the actual and the target sleep duration and plot it in a graph. For the evaluation, we compare the difference of the actual sleep duration from the target sleep duration when using xtimer and ztimer.

Aside from the result of our test, we also included an estimate of the accuracy threshold. This estimate is based on several values specified by the crystal manufacturers such as frequency tolerance, frequency stability, and the aging rate. Essentially, the sum of these values specifies the tolerance budget of the clock, which is the frequency range of the clock.

For the threshold line, we prefer to err on under specifiying the tolerance. We compared the values from a few boards and chose the one that is the highest. We use 50 ppm for the frequency stability and tolerance and 5 ppm/year for the aging rate. Additionally, we add 50 ppm more on top of the tolerance of the DUT to account for the tolerance of the external reference timer clock. In total, the tolerance budget used for the threshold in Figure 6.7 is 150 ppm.

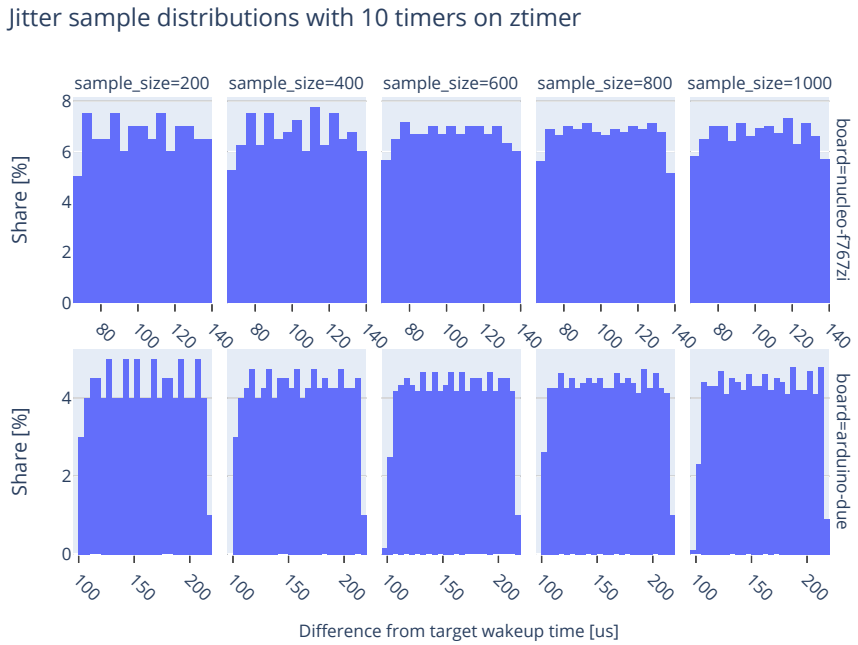
5.4 Choosing the sample size

We want to have confidence in our result but in the real world there is no determinism in the result and it might change in each run. However, we can make sure that our result covers the whole range of the result. Here the sample size is really important. Choosing a too small sample size would skew our result when there are extreme values among the samples. Choosing a too large sample size however will multiply the cost of running the test, requiring more time to finish. We need to strike a balance between the ‘accuracy’ and cost.

To get an idea of how the whole test result will be, we ran our test suite with varying sample sizes. This is run only on the nucleo-f767zi and the arduino-due board. The

former represents a high-performance board and the latter for the lesser performance. In this investigation, we opt not to use all boards because we are only trying to get a feel on how the result will be on varying sample sizes. This also helps lessens the complexity of analyzing the result from this stage.

Figure 5.3: Jitter sample sizes



For jitter stats, each run gives us 100 samples. We repeat the test 2, 4, 6, 8, and 10 times to give us 200, 400, 600, 800, and 1000 samples. Based on the results in Figure 5.3, we see that below 400 samples we can see a large difference in the result distribution. At 600 and more the result seems to more stable with less difference in the distribution. Therefore, we consider 600 as a suitable sample size for this test.

For the overhead test suite, we ran the test with 50, 250, 500, 750, and 1000 samples. The TIMER NOW overhead test results in Figure 5.5a does not show a big difference between test run with low and higher sample size. The majority of the samples sit at around 40%-50%. The midrange however shows that it is affected by the sample size. Having more sample size gives us a more constant result as shown by the result from runs with a sample size of 750 and 1000. Therefore, for TIMER NOW, we chose 1000

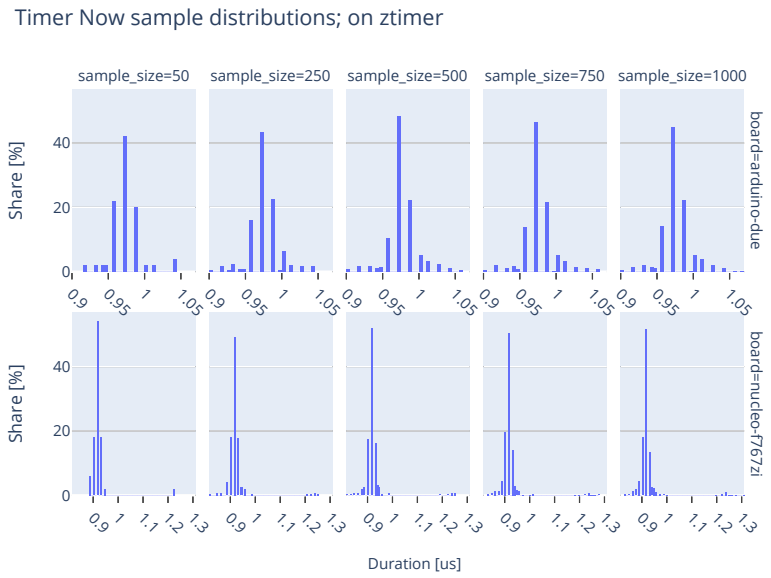
as the sample size, as gives us constant result and the test run does not take too long to finish.

In Figure 5.5b and Figure 5.6, we see that results from runs with sample size 50 shows a significant difference. However, starting from sample sizes 250, although there are still small differences, the share distribution of the samples are similar. Both of these tests take longer to finish for each cycle therefore we consider 250 samples is enough for both the TIMER SET and TIMER REMOVE tests.

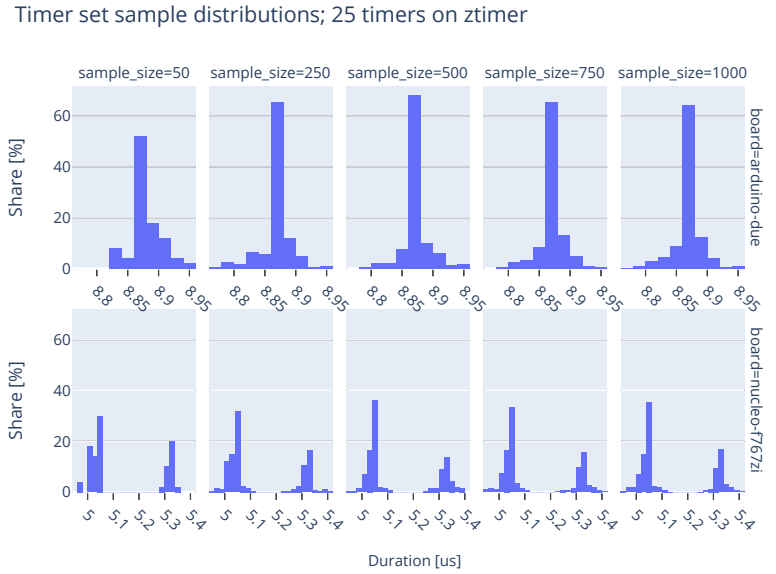
The accuracy test suite takes longer to finish for each run so we ran the test with 50, 100, 150, 200, and 250 only. The results shown in Figure 5.8a and Figure 5.8a vary a lot with only 50 and 100 samples but start stabilizing at 150 samples. This test also takes longer to finish a complete run. Therefore, we find 150 samples are enough for this test.

For the drift test, as this test takes the longest, we only run each test once. Ideally, this should be higher as that will further decrease the chance of any random errors affecting the overall result but due to the cost we decided against that and settle for only 1 sample for each test.

Figure 5.4: Overhead test suite sample sizes test result (i)



(a) TIMER NOW



(b) TIMER SET

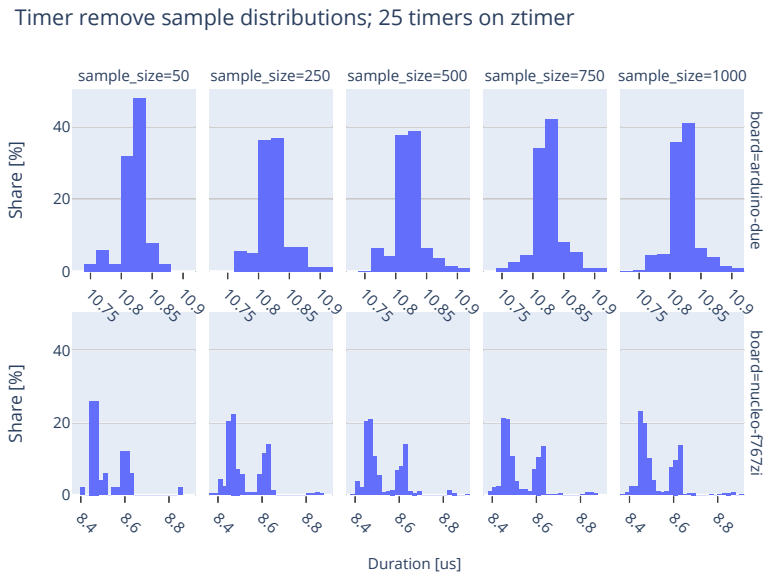
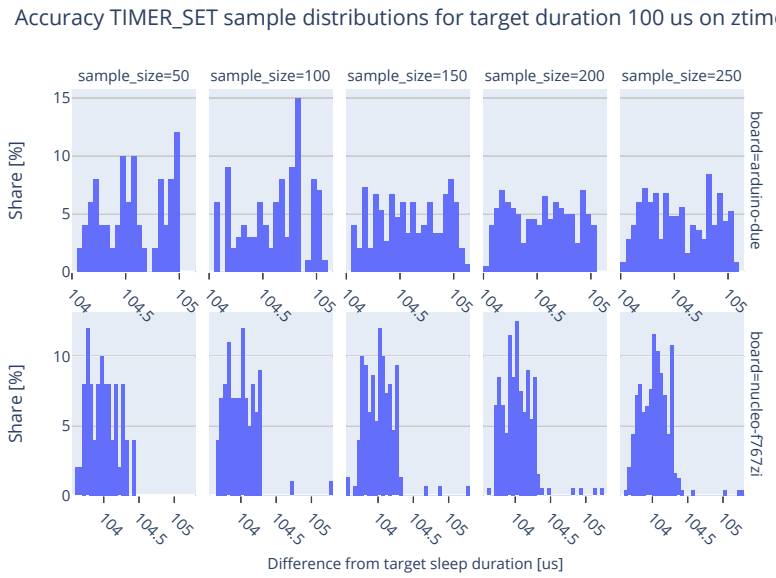
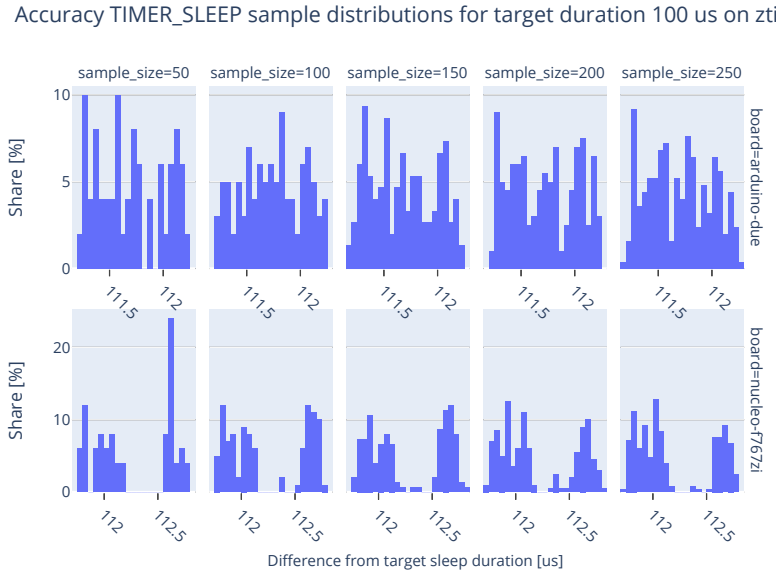


Figure 5.6: Overhead test suite sample sizes test result (ii)

Figure 5.7: Sleep accuracy test suite sample sizes test result



(a) TIMER SET



(b) TIMER SLEEP

6 Evaluations

In this chapter, we will evaluate the testing setup described in Chapter 5. In Section 6.1 we assessed how the infrastructure setup works for us. Then, in Section 6.2, we evaluate the test writing process. Finally, in Section 6.3, we look at empirical data obtained from running the tests against the current timer subsystems and also see issues that we found in the RIOT-OS timer subsystems through our testing.

6.1 HIL Setup

There is some limitation on PHiLIP itself that influences the way the tests is written. Firstly, PHiLIP uses a circular ring buffer with a size of 128 bytes to store the measurement from the DUT. This is due to the memory constraint in the microcontroller used for running the PHiLIP firmware. This means that the results from our measurements must fit in this 128 bytes buffer. Writing more than 128 bytes onto the buffer will cause us to lose the data from earlier measurements, as a ring buffer will overwrite the start of the ring when it is full.

This forces us to limit our result to only 128 samples per run before we need to reset the PHiLIP device itself. For each operation duration measurement, we need 2 bytes of buffer for the start and stop timestamp. Therefore, each of our test run repeats the test for only 50 times maximum. In the case where we only need one timestamp, this can be extended to 100 samples per test run. If a test needs more samples for better accuracy, the RobotFramework test procedure is used to repeatedly run the test and group the test result together.

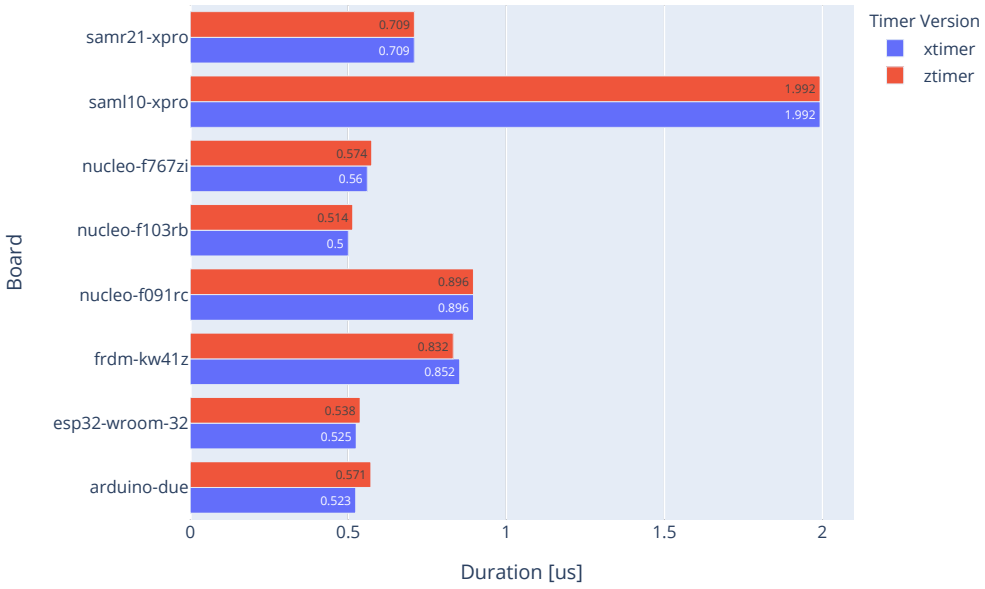
Other than that, PHiLIP has another issue that limits the duration of our testing. For a more accurate result, we use the Input Capture (IC) pin on PHiLIP for our measurement. This pin has a lower latency and gives us a more accurate timing measurement from the DUT. However, the pin limits the measured time to only around 1 minute. If we exceed

this duration, the measurement value will overflow and reports the wrong value back as a result.

There are several ways to get around this limitation. The first is to not measure more than 1 minute at a time. This means the duration from the start timestamp to the end timestamp must not be more than a minute. With this approach, we will not be able to do any long-running tests with our HIL setup. Second, we can manually compensate for the overflow in our calculation, if we estimated that the measurement will require more than a minute. The next workaround is to use another pin for the measurement that is less accurate but allows for a longer testing period.

After some testing and looking at the result, we are satisfied with the measurement values that we got by limiting our tests to 1 minute. Therefore, we proceeded with the first option which does not require much effort on our side. This is an acceptable tradeoff that we have decided based on our testing requirements.

Figure 6.1: Overhead GPIO



As mentioned in Section 5.1, we use a GPIO pin to start and stop the external reference timer. Based on Figure 6.1, the overhead of toggling GPIO on most of the board except for the saml10-xpro is lower than 1 µs. The saml10-xpro is the worst board in this aspect,

requiring around 2 µs. The results from our test suites are usually in the range of tens of microseconds. Assuming a measurement of 10 µs, the GPIO toggling overhead amounts to around 10% of the measured value. With a higher duration, the percentage of the GPIO will also be further reduced. Therefore, we consider the duration measurement using the external reference timer is enough for our use case.

6.2 Writing Test

Our HIL setup introduces an additional step in the test development process by requiring RobotFramework for test automation. The syntax for RobotFramework is based on common programming languages construct. It provides variables, lists, and loops which provides familiarity and a less steep learning curve.

Although writing a test now requires more steps than before, if the test firmware is written correctly in a generic way, it will promote test reusability. Changing the test parameter no longer requires changing the source code or recompiling the firmware because it can be changed directly from the RobotFramework testing procedure. With that said, we admit that learning a new language might add complexity to the adoption of this setup.

6.3 Timer Insights

In this section, we applied our testing suite to the RIOT-OS timer subsystem and collected insights on the behavior of the timer. Other than that, we also applied our test suite to a pull request from the RIOT-OS repository which introduces changes to the timer subsystem in the codebase and compares how that change affects the metrics compared to the version in the master branch. To make sure that our results are reproducible across test runs, we are using the 2020.10 release version of RIOT-OS with a patch to fix an issue we found on the arduino-due board as described below in issue 15530.

Issue 15530: interrupt register not cleared on arduino-due

During the development of our test suite, we found that the jitter test will always fail on the arduino-due board when using ztimer. We found that after removing a timer that has already triggered once, the interrupt flag in the hardware register is not cleared

correctly, causing the next timer set will trigger almost immediately. The same issue does not occur when using xtimer due to the difference in the implementation. This issue is recorded as `#15330` on the RIOT-OS Github repository issue tracker.

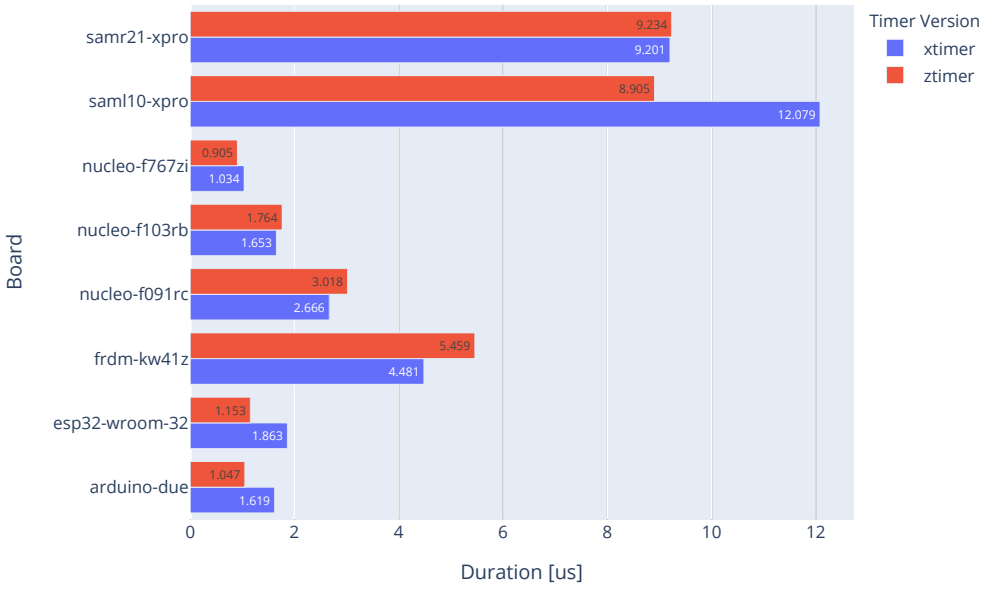
Members of the RIOT-OS community identified the problem briefly after the issue is reported and a fix is later provided and merged into the master branch. This fix however does not make it into the 2020.10 release version that we are using for our testing. Therefore, for our testing, we patched the 2020.10 release version with the fix provided.

This issue shows the merit of having an automated HIL testing setup to cover more boards that are usually left out during testing due to the overhead of manually running the test on each board.

Performance comparison of xtimer vs. ztimer

In this benchmark, we pitted our 2 timer subsystems, **xtimer**, and **ztimer**, against each other. Our goal is to find out how the different design decisions in each of the timer subsystems will affect the behavior and performance of the board.

Figure 6.2: Overhead TIMER NOW



From the result of the overhead test suite, first, we will look at the TIMER NOW test. From Figure 6.2, at first glance, it is clear that the saml10-xpro and samr21-xpro board is taking the longest to finish a call to the TIMER NOW function. Looking at the xtimer and ztimer comparison of other boards, there is no recognizable pattern that can be found. On some boards, ztimer is faster while on other boards, xtimer is faster.

Next in Figure 6.3, we have the result of the setting and removing the timer test from our overhead test suite. In the top row, we see that removing a timer from a list is slower on some boards with ztimer. In the worst case, the saml10-xpro board took around 20 µs longer to remove a timer. Other boards like the frdm-kw41z and arduino-due do not show an apparent difference with both timer subsystems.

In the bottom row, we see the result of setting a timer. This time ztimer shows a more promising result, reducing the time taken to set a timer by almost half of the original duration, especially at a higher timer count. The saml10-xpro board again shows a significant change, although this time the result is better in ztimer compared to xtimer.

We will look at the results from our jitter test suite in Figure 6.4. Apart from the frdm-kw41z board, all other boards show an expected increase in the minimum difference between actual and target wakeup time for both xtimer and ztimer. However, in most cases, the wakeup time for ztimer is in the lower range compared to xtimer. We can also see that both the samr21-xpro and the saml10-xpro have significantly worse results compared to others. In the worst case, the former wakes up 2500 µs later and the latter 3000 µs earlier from the target wakeup time.

Moving on to the accuracy tests, we can see a clear difference in how the design decision in the timer implementation affects the accuracy of the timer shown by the shape of the curve. Figure 6.5 shows the result of the TIMER SET test and 6.6 shows the result of the TIMER SLEEP test.

xtimer has a drastic jump after the first 20 to 40 µs before stabilizing while ztimer starts with less accuracy and drops to higher accuracy later on. The behavior in xtimer can be attributed to the `XTIMER_BACKOFF` config option. In xtimer, when a user sets a timer with a duration less than the value of `XTIMER_BACKOFF`, xtimer will actively block the thread instead of setting a new timer. This config option is meant to compensate for the overhead when accessing the timer register on each board. However, as we discussed in Chapter 4, some boards are misconfigured resulting in bugs and issues in production.

Figure 6.3: Overhead setting and removing timers

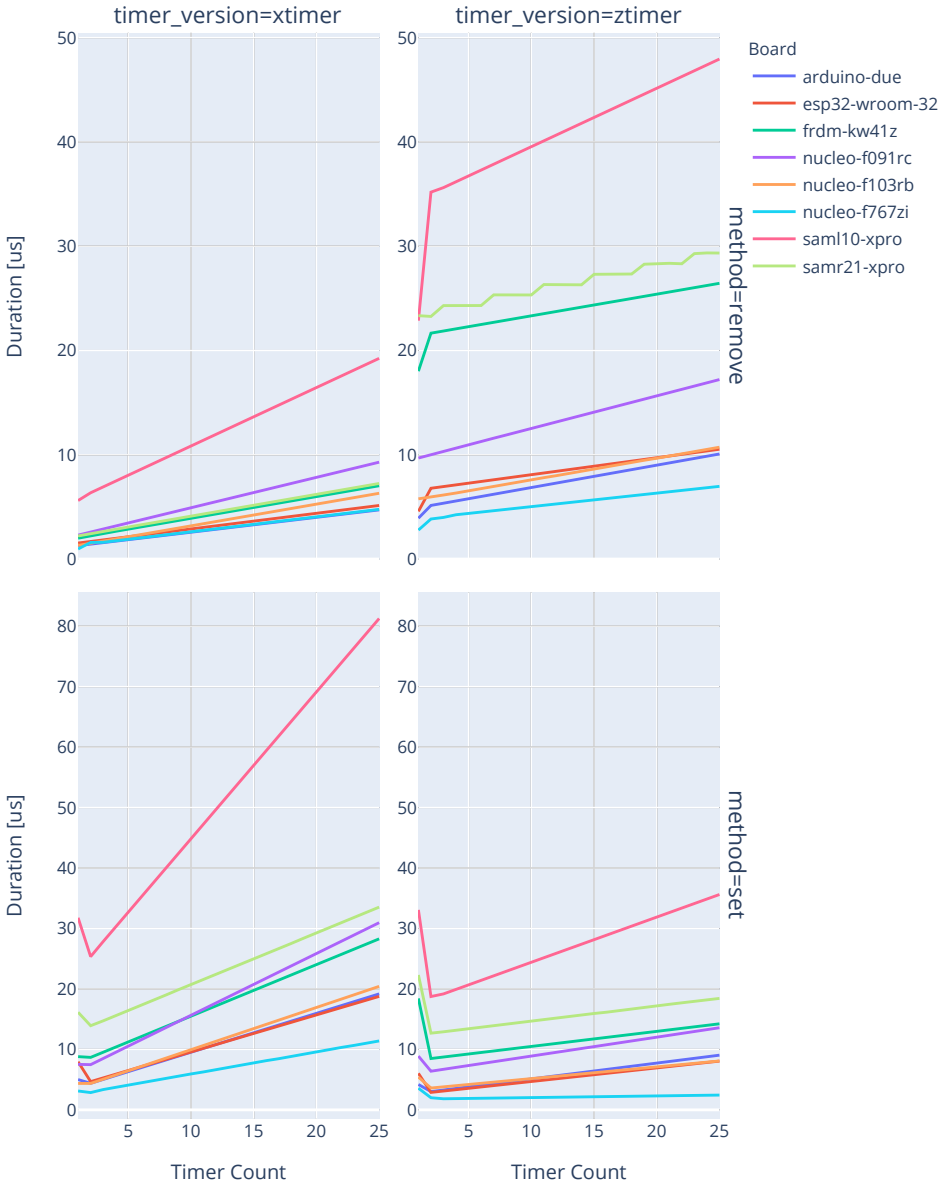


Figure 6.4: Jitter comparison

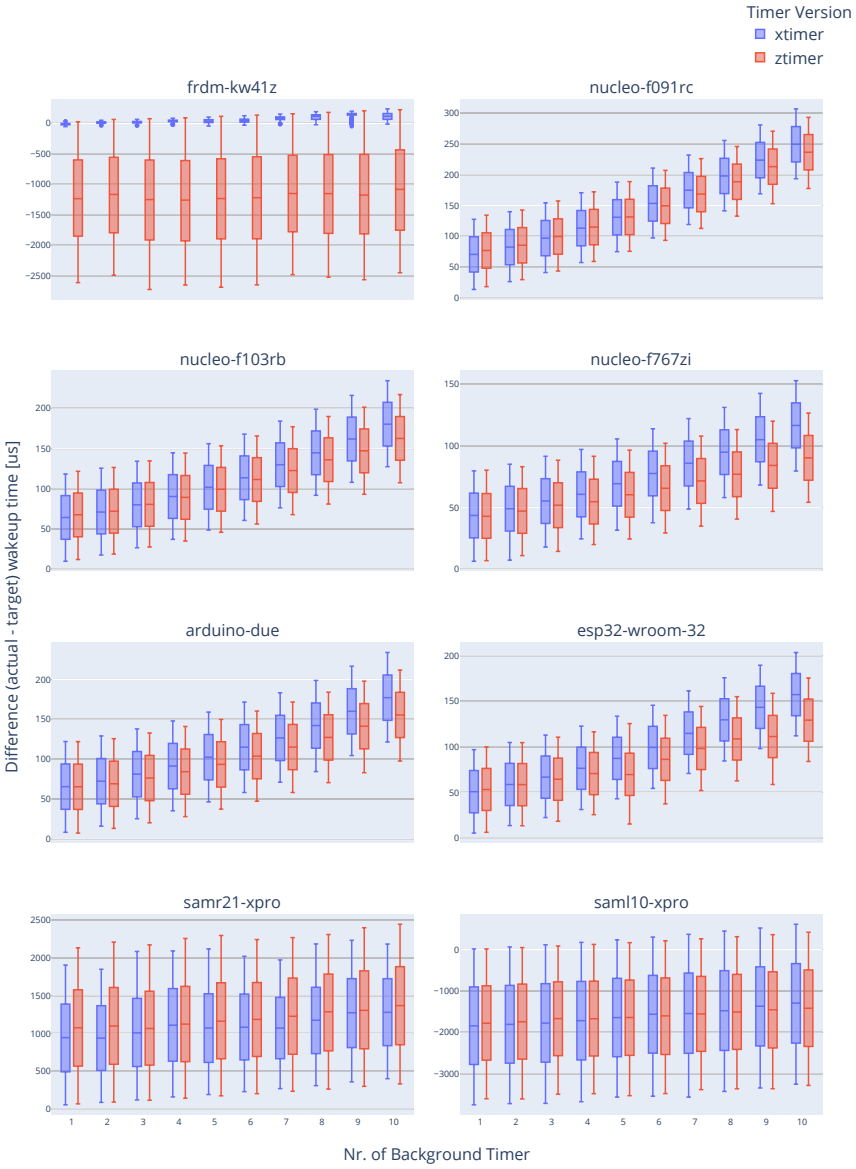
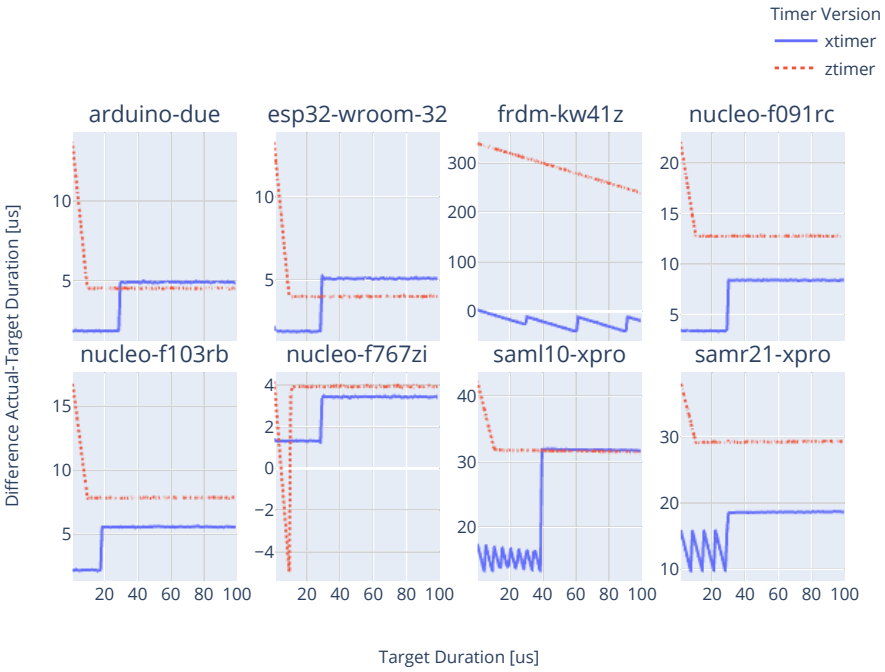


Figure 6.5: TIMER SET Accuracy

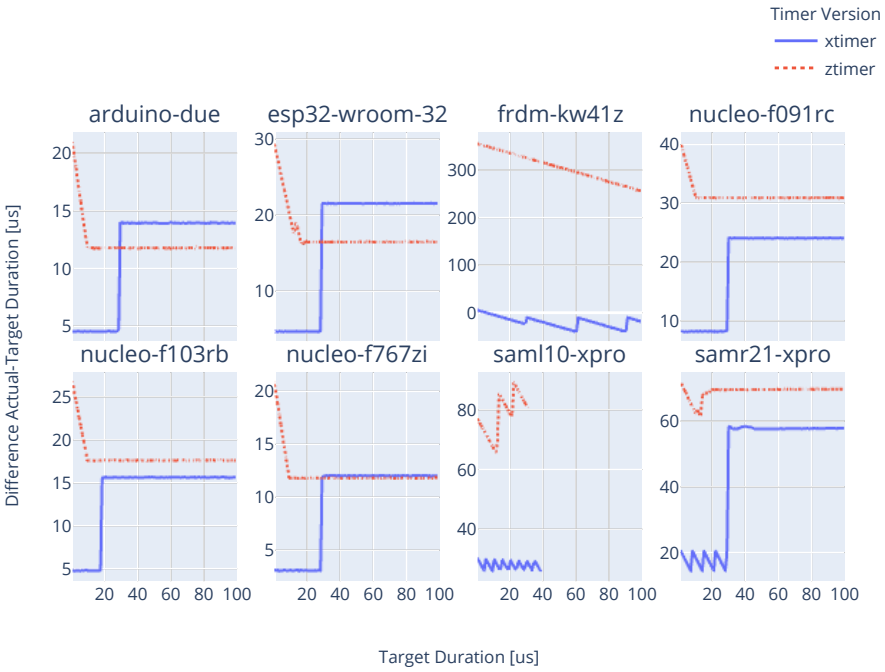


Other than that, with xtimer, we see that with smaller sleep duration the actual sleep time might vary drastically, shown by the zigzag line in the diagram, making the timer unstable. Despite the praise for ztimer, we can see in Figure 6.6, ztimer fluctuates in the smaller sleep duration range on the saml1-xpro and samr21-xpro boards. This observation can be further investigated and might point us to a possible misconfiguration of the timer subsystem on the said boards, which can be fixed to improve ztimer performance.

Figure 6.8a shows the result of the drift test suite. In the figure, for both xtimer and ztimer, the actual sleep duration of samr21-xpro, saml10-xpro, and frdm-kw41z have a big difference from the target sleep duration. However, between the two timers, there is no significant difference between their results except for the frdm-kw41z board, where the clock skew gets significantly larger with ztimer.

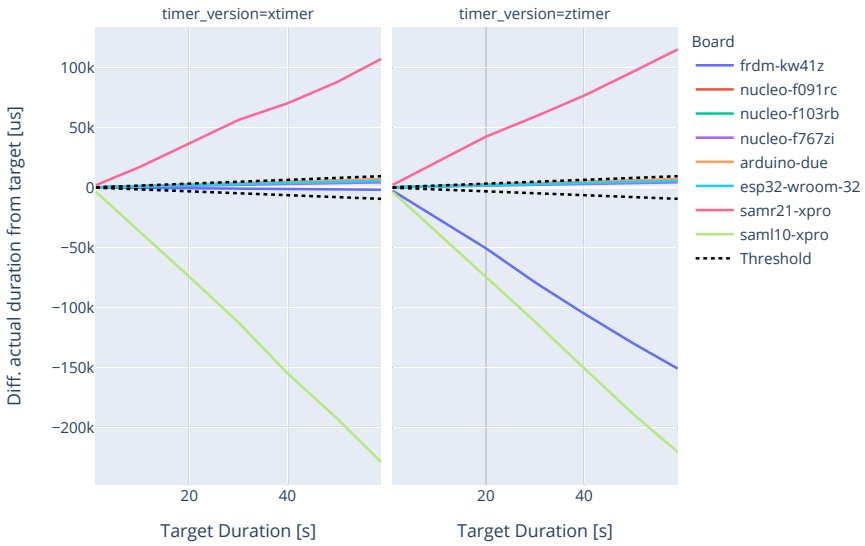
Although Figure 6.8a shows the interesting behavior of one of the boards, the significant difference from other boards hid the characteristics of the results from other boards. Therefore, in Figure 6.8b, we show the same result but without the three dominant board from before. In this figure, the threshold around each line represents the range of possible values based on the calculated tolerance budget. There is no significant difference

Figure 6.6: TIMER SLEEP Accuracy

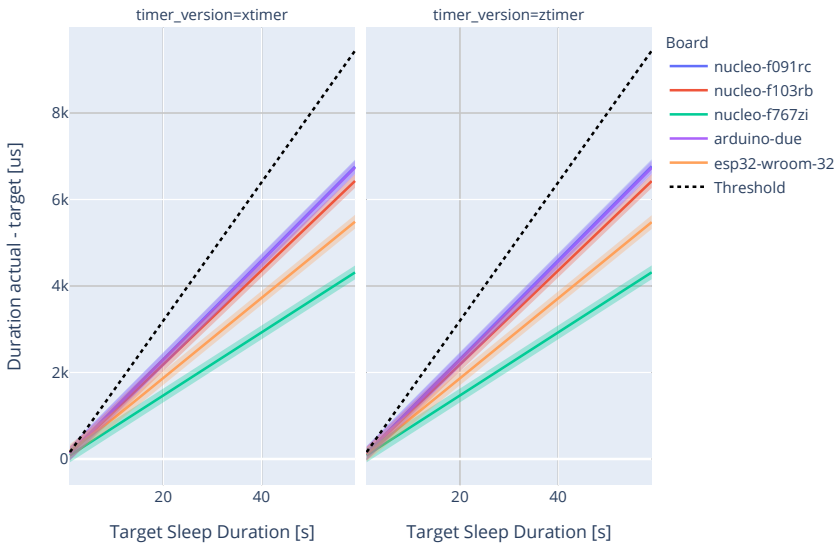


between xtimer and ztimer here and all the results are also lower than our threshold of acceptable sleep duration.

Figure 6.7: Clock skew comparison across boards



(a) All boards



(b) Without samr21-xpro, saml10-xpro, frdm-kw41z

PR13103: xtimer simplification

Other than using the test suite to uncover the behavior of the timer subsystems, we also have used it in pull requests to provide a performance evaluation of the introduced changes and verifies that the changes do not significantly hamper the performance of the timer subsystems.

PR13103 refactors the xtimer code to simplify its implementation by combining functions where possible and making the TIMER REMOVE function more straightforward. The manual review process by the maintainers shows no apparent problems with the changes but until now there have been no numeric values that can show how these changes affect the performance of xtimer. Therefore, the PR is a perfect candidate to be used with our test suite.

For our measurements, we used RIOT-OS release 2020.10 with the patch from issue 15530 applied onto it as the base. The PR is originally based on an earlier commit but to make sure that our result is comparable to other measurements that we did in this work, we rebased the PR onto the same RIOT-OS release with some minor modifications to fix the rebase errors. The following paragraphs in this subsection discuss the result of our measurements.

For the overhead test suite, we do not see any significant difference between the performance of the changes from PR13103 and the base 2020.10 release version. The result is shown in Figure 6.9 and Figure 6.10.

From Figure 6.11, most of the boards show slight improvements with the changes, having a smaller max wakeup time from the target. Otherwise, there is not much improvement to the performance of the xtimer subsystem.

For accuracy, the TIMER SET test does not show much difference in performance with the performance applied as shown in Figure 6.5. The result from 2020.10 and PR13103 timer versions on the frdm-kw41z is similar in that it almost completely overlapped in the figure. However, a more significant can be observed in the TIMER SLEEP result in Figure 6.6. Almost all boards show a reduction in the difference in actual sleep duration from the target duration. At the minimum, this amounts to 2 ms but on saml1-xpro and samr21-pro this could be as large as 10 ms.

Figure 6.9: Overhead TIMER NOW

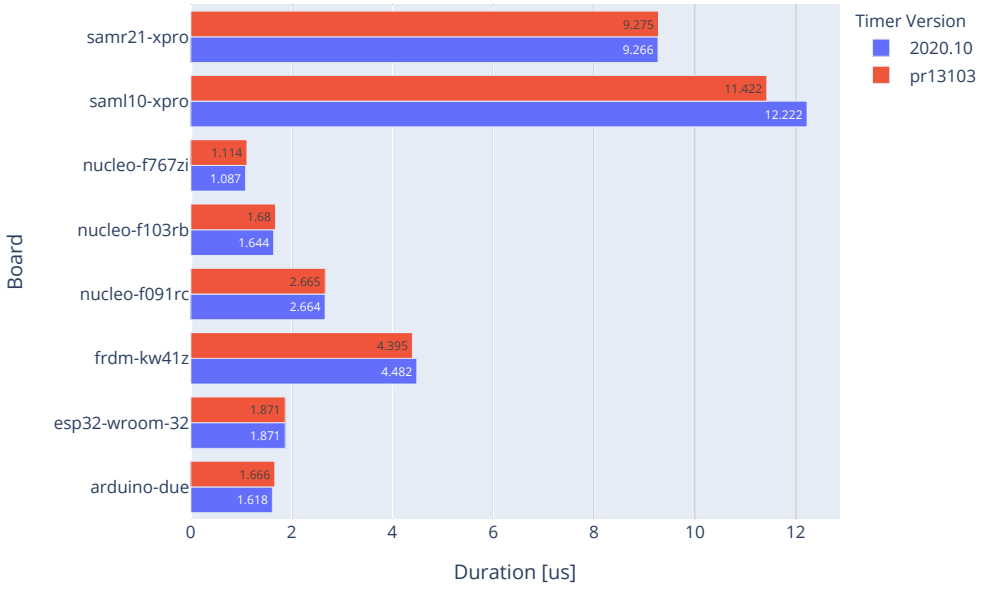


Figure 6.10: Overhead setting and removing timers

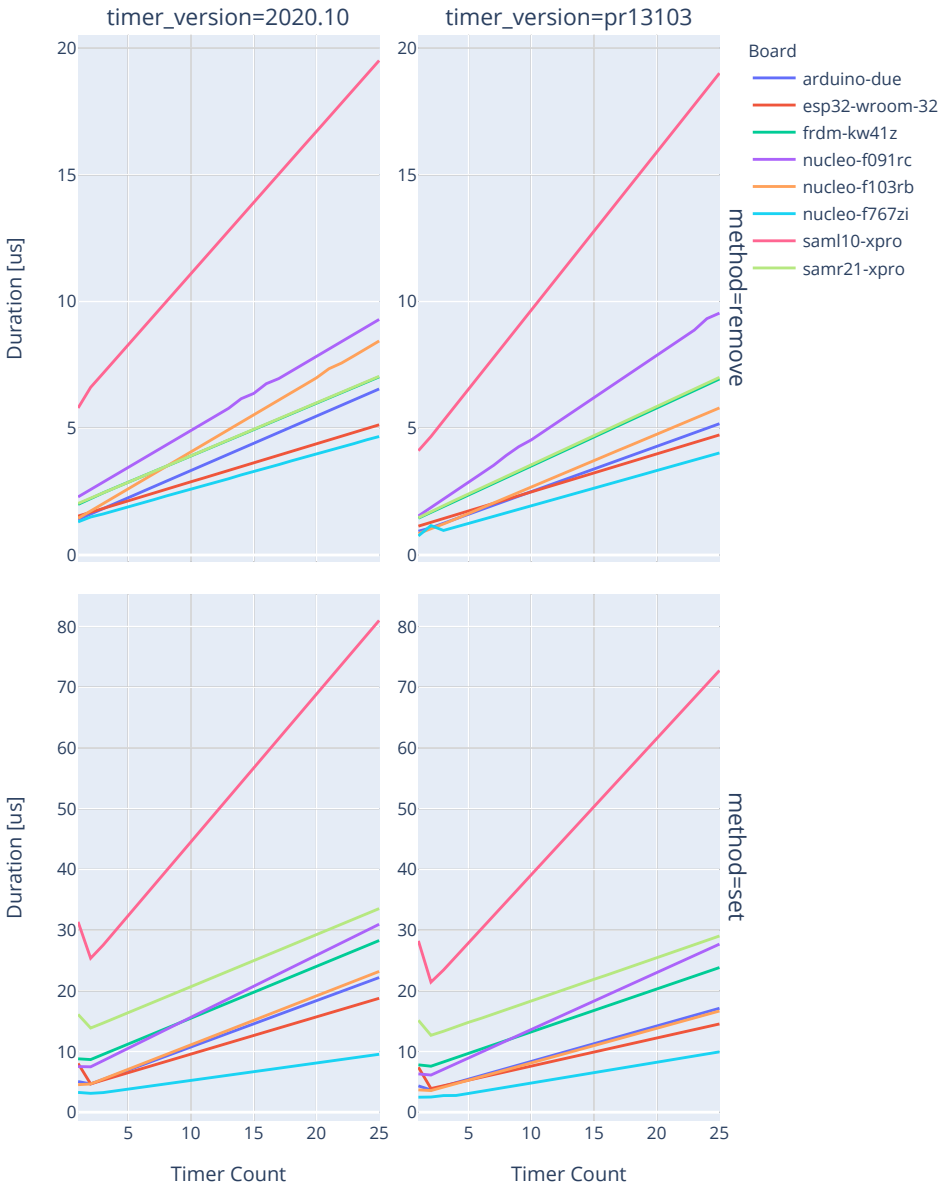


Figure 6.11: Jitter comparison between release 2020.10 and PR13103

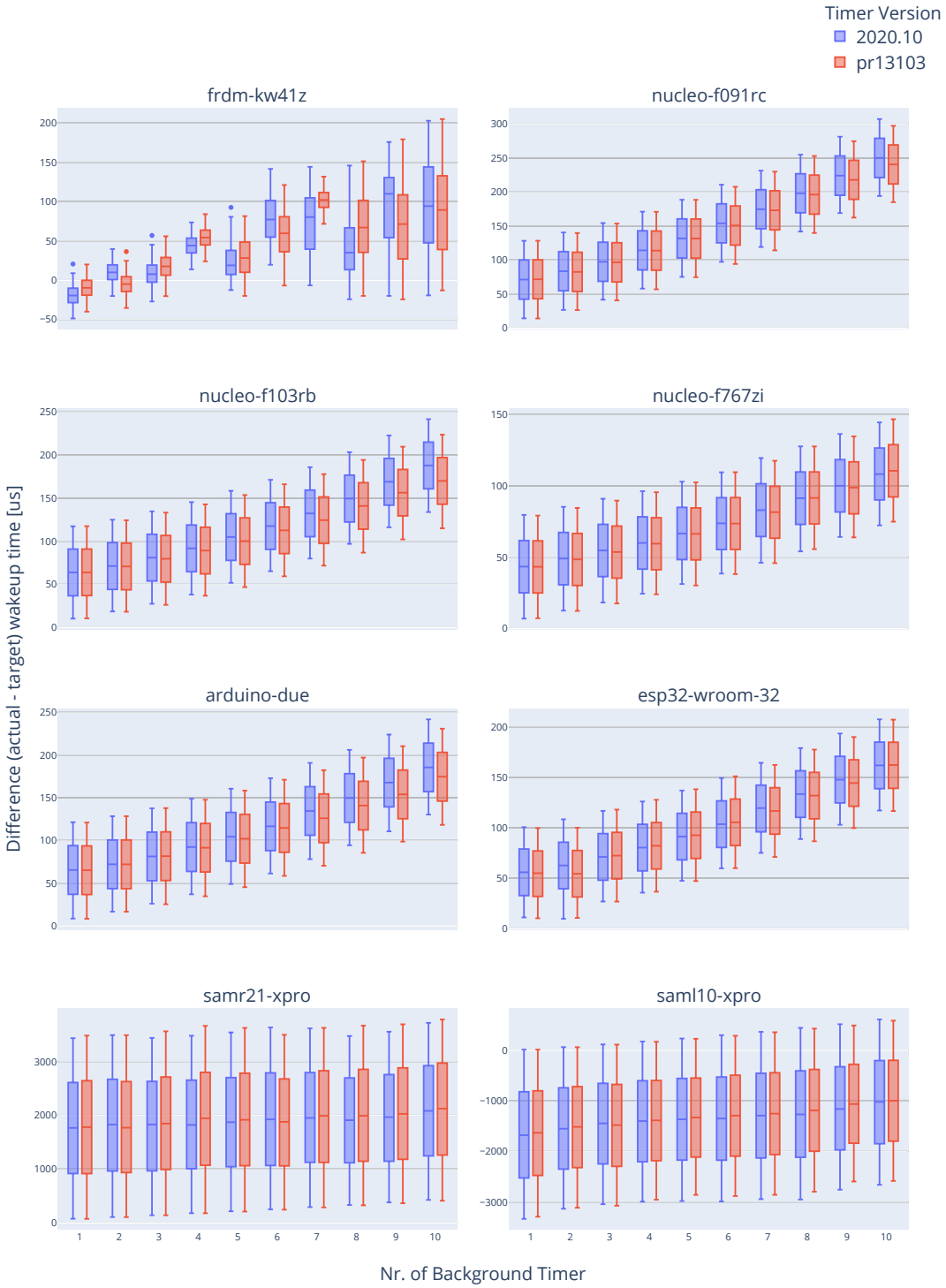


Figure 6.12: TIMER SET Accuracy

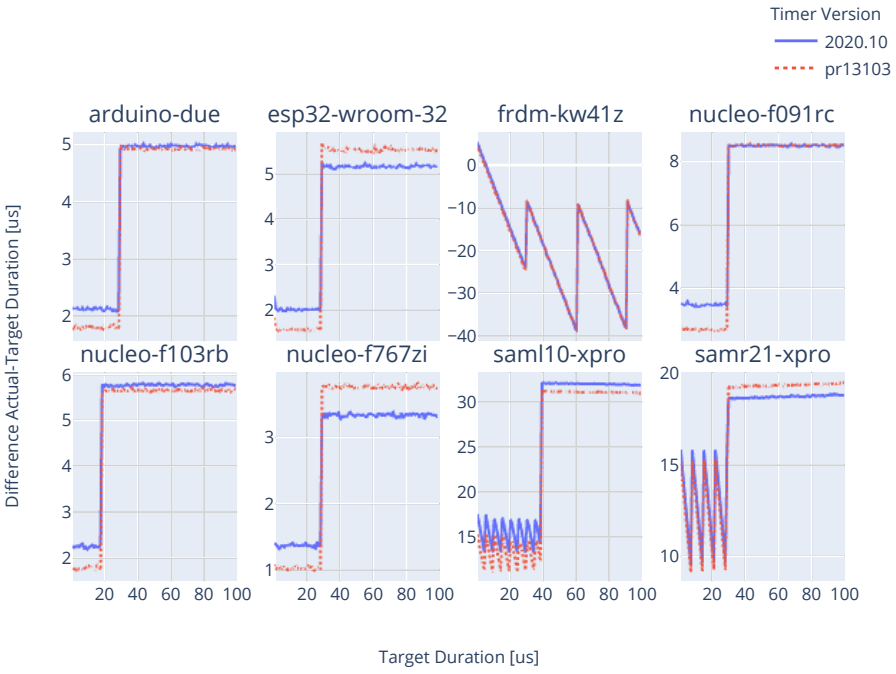
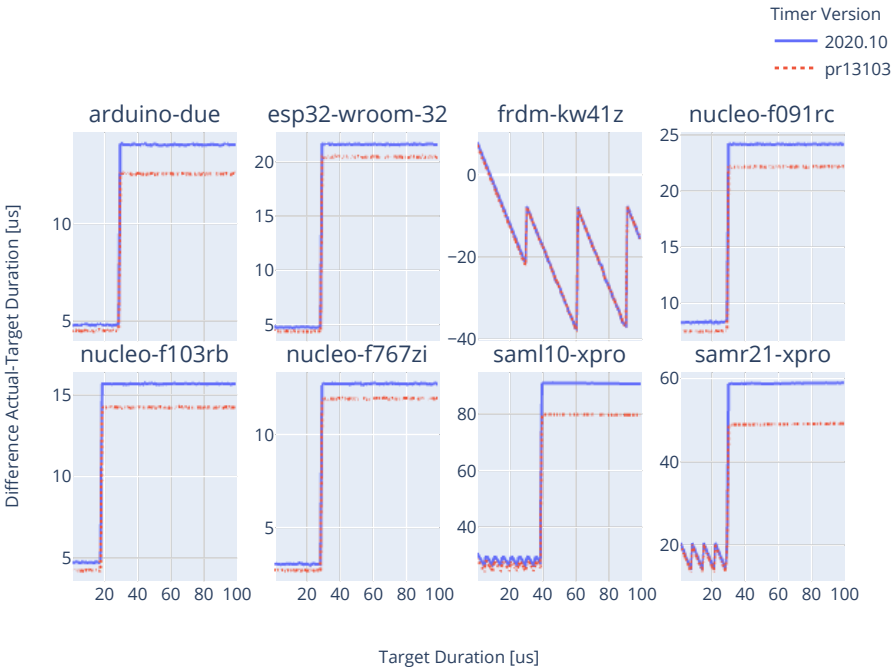


Figure 6.13: TIMER SLEEP Accuracy



7 Conclusion

In this body of work, we suggested a new setup for doing automated Hardware-in-the-Loop testing. We showed how off-the-shelves software can be used to build the infrastructure and increase the reusability of our tests. We also designed example test suites that can be used with the aforementioned testing infrastructure. Finally, we demonstrated how we use the testing infrastructure to generate a standardized result across a multitude of boards without manually running the test one-by-one.

The results from our testing also uncover new and interesting aspects of the two timer subsystems in RIOT-OS that are not previously known by the maintainers. This will be a useful guideline for choosing which parts of the timer subsystem can be further improved.

By comparing the results from each board, we found suspicious behavior that may point to misconfigurations of the boards. Through the integration of real hardware in our HIL testing setup, we uncovered a bug in one of the tested boards, proving the merit of HIL testing in assuring the quality of the OS.

Our testing also provided concrete metrics that can be used to evaluate the changes introduced in a pull request. This helps the maintainers in avoiding pitfalls when making changes to the timer subsystem, thus reducing regression in the performance.

Bibliography

[1] M. Glenford J., *The Art of Software Testing.*, ser. Business Data Processing: A Wiley Series. Wiley, 2004, vol. 2nd ed.

[2] N. P. Kropp, P. J. Koopman, and D. P. Siewiorek, "Automated Robustness Testing of Off-the-Shelf Software Components," in *Digest of Papers. Twenty-Eighth Annual International Symposium on Fault-Tolerant Computing (Cat. No.98CB36224).* IEEE, 1998, pp. 230–239.

[3] L. Inozemtseva and R. Holmes, "Coverage is Not Strongly Correlated with Test Suite Effectiveness," in *Proceedings of the 36th International Conference on Software Engineering (ICSE '14).* New York, NY, USA: Association for Computing Machinery, 2014, p. 435–445.

[4] E. T. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo, "The Oracle Problem in Software Testing: A Survey," *IEEE Transactions on Software Engineering*, vol. 41, no. 5, pp. 507–525, May 2015.

[5] L. du Bousquet and N. Zuanon, "An Overview of Lutess a Specification-based Tool for Testing Synchronous Software," in *14th IEEE International Conference on Automated Software Engineering.* IEEE, 1999, pp. 208–215.

[6] K. Karhu, T. Repo, O. Taipale, and K. Smolander, "Empirical Observations on Software Testing Automation," in *2009 International Conference on Software Testing Verification and Validation.* IEEE, 2009, pp. 201–209.

[7] J. Humble and D. Farley, *Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation.* Pearson Education, Inc, 2012.

[8] M. Bailey and J. Doerr, "Contributions of Hardware-in-the-Loop Simulations to Navy Test and Evaluation," in *Technologies for synthetic environments: Hardware-in-the-loop testing*, vol. 2741. International Society for Optics and Photonics, 1996, pp. 33–43.

[9] M. Bacic, "On Hardware-in-the-Loop Simulation," in *44th IEEE Conference on Decision and Control (CDC'44)*. IEEE, 2005, pp. 3194–3198.

[10] S. Brennan and A. Alleyne, "Using a Scale Testbed: Controller Design and Evaluation," *IEEE Control Systems Magazine*, vol. 21, no. 3, pp. 15–26, 2001.

[11] H. Eguchi and T. Yamashita, "Benefits of HWIL Simulation to Develop Guidance and Control Systems for Missiles," in *Technologies for Synthetic Environments: Hardware-in-the-Loop Testing V*, vol. 4027. International Society for Optics and Photonics, 2000, pp. 66–73.

[12] M. Lenders, P. Kietzmann, O. Hahm, H. Petersen, C. Gündoğan, E. Baccelli, K. Schleiser, T. C. Schmidt, and M. Wählisch, "Connecting the World of Embedded Mobiles: The RIOT Approach to Ubiquitous Networking for the Internet of Things," 2018.

[13] M. Rottleuthner, T. C. Schmidt, and M. Wählisch, "Sense Your Power: The ECO Approach to Energy Awareness for IoT Devices," *ACM Transactions on Embedded Computing Systems (TECS)*, 2021.

[14] E. Baccelli, O. Hahm, M. Günes, M. Wählisch, and T. C. Schmidt, "RIOT OS: Towards an OS for the Internet of Things," in *Proc. of the 32nd IEEE INFOCOM. Poster*. Piscataway, NJ, USA: IEEE Press, 2013, pp. 79–80.

[15] E. Baccelli, C. Gündogan, O. Hahm, P. Kietzmann, M. Lenders, H. Petersen, K. Schleiser, T. C. Schmidt, and M. Wählisch, "RIOT: an Open Source Operating System for Low-end Embedded Devices in the IoT," *IEEE Internet of Things Journal*, vol. 5, no. 6, pp. 4428–4440, December 2018.

[16] W. Shang, A. Afanasyev, and L. Zhang, "The design and implementation of the NDN protocol stack for RIOT-OS," in *2016 IEEE Globecom Workshops (GC Wkshps)*. IEEE, 2016, pp. 1–6.

[17] K. Roussel, Y.-Q. Song, and O. Zendra, "RIOT OS Paves the Way for Implementation of High-performance MAC Protocols," in *Proceedings of the 4th International Conference on Sensor Networks*, 2015, pp. 5–14.

[18] S. Challouf, L. Kriaa, and L. A. Saidane, "Power consumption comparison of synchronized IoT devices running FreeRTOS and RIOT," in *2019 8th International Conference on Performance Evaluation and Modeling in Wired and Wireless Networks (PEMWN)*. IEEE, 2019, pp. 1–5.

[19] L. Abeni, A. Goel, C. Krasic, J. Snow, and J. Walpole, "A Measurement-Based Analysis of the Real-Time Performance of Linux," in *8th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'02)*. IEEE, 2002, pp. 133–142.

[20] A. A. Fröhlich, G. Gracioli, and J. F. Santos, "Periodic Timers Revisited: The Real-Time Embedded System Perspective," *Computers & Electrical Engineering*, vol. 37, no. 3, pp. 365–375, 2011.

[21] A. M. Costello and G. Varghese, "Redesigning the BSD Callout and Timer Facilities," no. WUCS-95-23, 1995.

[22] J. Lee and K.-H. Park, "Delayed Locking Technique for Improving Real-Time Performance of Embedded Linux by Prediction of Timer Interrupt," in *11th IEEE Real Time and Embedded Technology and Applications Symposium (RTAS'05)*. IEEE, 2005, pp. 487–496.

[23] Y. Wu, S. Wang, and Z. Yu, "Embedded Software Reliability Testing and its Practice," in *ICCDA 2010*, vol. 2. IEEE, 2010, pp. V2–24.

[24] J. S. Cole Jr and A. C. Jolly, "Hardware-in-the-Loop Simulation at the US Army Missile Command," in *Technologies for Synthetic Environments: Hardware-in-the-Loop Testing*, vol. 2741. International Society for Optics and Photonics, 1996, pp. 14–19.

[25] M. E. Sisle and E. D. McCarthy, "Hardware-in-the-Loop Simulation for an Active Missile," *Simulation*, vol. 39, no. 5, pp. 159–167, 1982.

[26] S. Brennan, A. Alleyne, and M. DePoorter, "The Illinois Roadway Simulator-a Hardware-in-the-Loop Testbed for Vehicle Dynamics and Control," in *Proceedings of the 1998 American Control Conference. ACC (IEEE Cat. No. 98CH36207)*, vol. 1. IEEE, 1998, pp. 493–497.

[27] R. McNeal and M. Belkhayat, "Standard Tools for Hardware-in-the-Loop (HIL) Modeling and Simulation," in *2007 IEEE Electric Ship Technologies Symposium*. IEEE, 2007, pp. 130–137.

[28] B. Lu, X. Wu, H. Figueroa, and A. Monti, "A Low-cost Real-time Hardware-in-the-Loop Testing Approach of Power Electronics Controls," *IEEE Transactions on Industrial Electronics*, vol. 54, no. 2, pp. 919–931, 2007.

[29] M. M. R. Mozumdar, L. Lavagno, L. Vanzago, and A. L. Sangiovanni-Vincentelli, "HILAC: A Framework for Hardware-in-the-Loop Simulation and multi-platform Automatic Code Generation of WSN Applications," in *International Symposium on Industrial Embedded System (SIES)*. IEEE, 2010, pp. 88–97.

[30] Jenkins CI. [Online]. Available: https://www.jenkins.io/index.html

[31] PHiLIP. [Online]. Available: https://github.com/riot-appstore/PHiLIP

[32] RobotFramework. [Online]. Available: https://robotframework.org/

Glossary

RIOT-OS RIOT operation system.

TIMER NOW Timer function to get the current time from DUT.

TIMER REMOVE Timer function to remove a timer from list.

TIMER SET Timer function to set a timer in the future.

TIMER SLEEP Timer function to go to sleep for a specified duration.

Erklärung zur selbstständigen Bearbeitung einer Abschlussarbeit

Gemäß der Allgemeinen Prüfungs- und Studienordnung ist zusammen mit der Abschlussarbeit eine schriftliche Erklärung abzugeben, in der der Studierende bestätigt, dass die Abschlussarbeit „— bei einer Gruppenarbeit die entsprechend gekennzeichneten Teile der Arbeit [(§ 18 Abs. 1 APSO-TI-BM bzw. § 21 Abs. 1 APSO-INGI)] — ohne fremde Hilfe selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel benutzt wurden. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich zu machen."

Quelle: § 16 Abs. 5 APSO-TI-BM bzw. § 15 Abs. 6 APSO-INGI

Erklärung zur selbstständigen Bearbeitung der Arbeit

Hiermit versichere ich,

Name: _____

Vorname: _____

dass ich die vorliegende Bachelorarbeit — bzw. bei einer Gruppenarbeit die entsprechend gekennzeichneten Teile der Arbeit — mit dem Thema:

Automated Testing of the RIOT-OS Timer Subsystem

ohne fremde Hilfe selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel benutzt habe. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich gemacht.

_____	_____	_____
Ort	Datum	Unterschrift im Original