PHiLIP on the HiL: Automated Multi-platform OS Testing with External Reference Devices

KEVIN WEISS, Hamburg University of Applied Sciences, Germany

MICHEL ROTTLEUTHNER, Hamburg University of Applied Sciences, Germany

THOMAS C. SCHMIDT, Hamburg University of Applied Sciences, Germany

MATTHIAS WÄHLISCH, Freie Universität Berlin, Germany

Developing an operating system (OS) for low-end embedded devices requires continuous adaptation to new hardware architectures and components, while serviceability of features needs to be assured for each individual platform under tight resource constraints. It is challenging to design a versatile and accurate heterogeneous test environment that is agile enough to cover a continuous evolution of the code base and platforms. This mission is even more challenging when organized in an agile open-source community process with many contributors such as for the RIOT OS. Hardware in the Loop (HiL) testing and Continuous Integration (CI) are automatable approaches to verify functionality, prevent regressions, and improve the overall quality at development speed in large community projects.

In this paper, we present PHiLIP (Primitive Hardware in the Loop Integration Product), an open-source external reference device together with tools that validate the system software while it controls hardware and interprets physical signals. Instead of focusing on a specific test setting, PHiLIP takes the approach of a tool-assisted agile HiL test process, designed for continuous evolution and deployment cycles. We explain its design, describe how it supports HiL tests, evaluate performance metrics, and report on practical experiences of employing PHiLIP in an automated CI test infrastructure. Our initial deployment comprises 22 unique platforms, each of which executes 98 peripheral tests every night. PHiLIP allows for easy extension of low-cost, adaptive testing infrastructures but serves testing techniques and tools to a much wider range of applications.

CCS Concepts: • **Hardware** → **Hardware test**; • **Software and its engineering** → **Software testing and debugging**; **Operating systems**; • **Computer systems organization** → **Embedded systems**.

Additional Key Words and Phrases: IoT, hardware in the loop, operating system, constrained devices

1 INTRODUCTION

The rapidly expanding Internet of Things (IoT) faces the continuous arrival of new microcontroller units (MCUs), peripherals, and platforms. New and established components collectively comprise

---

(a) A local HiL test setup that connects a device under test (DUT) (1) to PHiLIP firmware in a bluepill board (2) which is mounted on a Raspberry Pi based test node (3).
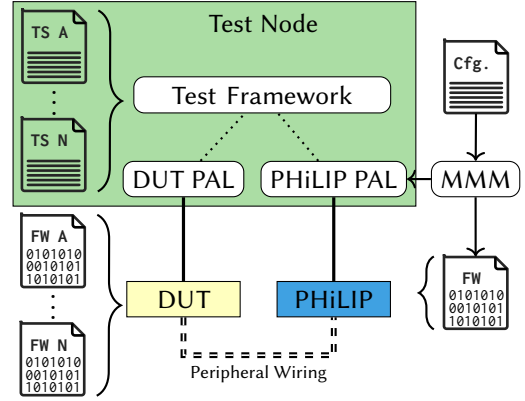
(b) Overview of the HiL test environment: The test node runs test suites (TS) and interfaces with the DUT and PHiLIP via Protocol Abstraction Layers (PALs). Each test suite corresponds to a test firmware (FW). PHiLIP firmware and PAL are configured via the Memory Map Manager (MMM).

Fig. 1. Overview of physical components of the test setup and their architecture integration.

a zoo of embedded hardware platforms that admit various capabilities and a distinctively diverse set of features. Application software is often requested to operate these devices with significant responsibility and high reliability. In this context, employing an embedded OS with hardware abstraction can significantly ease development by making common code reusable and hardware independent. High-quality requirements on such system software, however, can only be ensured after extensive validation in realistic testing procedures.

Testing an embedded OS can be challenging due to the constrained nature of the devices, the variety of hardware-specific behavior, and the requirements of real-world interactions. As embedded devices interact with external hardware and the physical world, testing must also verify this behavior. Rapidly evolving hardware and agile software development require tests to be run regularly on a vast number of individual devices. An automated, extensible way of testing a large variety of embedded devices is therefore required to develop and maintain a reliable embedded OS.

Many previous attempts in this area remained limited to individual use cases, without aiming for generality or focusing on a broad range of features and platforms. Respectively, multi-platform Hardware in the Loop (HiL) testing is currently not covered well. Moreover, previous solutions are often hard to acquire, set up, and maintain. A versatile testing tool filling this gap should instead be compatible with a wide set of platforms while being easy to obtain, use, and extend.

In this paper, we try to bridge this gap between early research and reality for the open source OS RIOT [11]. We propose a layered testing architecture that enables agile test development and employs an external reference device to provide automated HiL testing for a large variety of embedded devices. Following this approach, we intend to close the loop from research to design, engineering, and further to operations, which triggered a rich set of feedback. Lessons learned again substantiated research and design work during the years of developing and optimizing PHiLIP. A strong interaction with the large RIOT community was part of this process.

The concept considers three main components. One is a resource-constrained device-under-test (DUT) that is evaluated (see Figure 1(a)). The second is a reference device that takes measurements

and executes hardware interactions with the DUT. The third referred to as *test node*, coordinates, and controls the two previously introduced entities. To streamline the integration and maintenance of implemented tools and firmware, our concept also includes the memory map manager (MMM) that simplifies versioning and coordination of configuration data and documentation across device and tool boundaries. This is shown in Figure 1(b) and will be explained in Section 3. In particular, this paper makes the following key contributions:

(1) Following an in depths requirements analysis, we design a testing abstraction layer and a structured testing interface.
(2) We introduce PHiLIP, our Primitive HiL Integration Product, as a firmware with verified peripheral behavior together with a tool-set for agile test development.
(3) We integrate PHiLIP with multi-platform DUTs into a fully automated HiL testing environment and report on lessons learned based on our deployment.
(4) We evaluate our HiL testing proposal from the perspectives of testing impact, resource expenditures, and scalability.

About half of the current work on IoT testing concerns interoperability and testbeds [3]. Interoperability testing mainly targets networking [39], often more narrowly wireless protocol conformance [37], or protocol performance [27]. Device heterogeneity is still a major challenge [23], though. Only very few contributions exist that are validated against real open-source embedded software, whereas the majority uses simplified examples. The fast-paced business culture and lacking standardization in this area [61] leads to sidelining of serious challenges regarding product quality, security, and privacy [26]. Together with the prevalent resource constraints, devices are often shipped without complex code that prevents or corrects errors at run time—eventually leading to faulty behavior and reliability problems of applications [54]. All those observations strongly support the relevance of our multi-platform testing for hardware layer abstractions of a popular open-source OS.

In the remainder of this paper, we present PHiLIP together with its design concepts, functions, tools, and experiences from our long-term deployment in the wild. The presentation is organized as follows. Section 2 discusses background and challenges related to testing embedded devices and OSs. An overview of our proposed HiL testing architecture is given in Section 3. Section 4 dives into the details of PHiLIP, a key component of our solution. Section 5 explains how our setup is used for automated multi-platform testing of OS hardware abstraction modules. We evaluate our approach in Section 6 and report on its key performance metrics. Other work is related in Section 7. Some lessons learned and potential improvements are discussed in Section 8 together with conclusions and an outlook on future work.

2 TESTING EMBEDDED SYSTEMS: CHALLENGES AND REQUIREMENTS

Software testing commonly consists of functional [51] and non-functional tests [42]. Methodologies such as test-driven development [29] or model-based testing (MBT) [30] can be applied at almost any stage of the development process. Even though it seems suitable to approach testing of IoT solutions via conventional software testing levels [60], embedded systems at the lower end of IoT architectures put up a unique set of nontrivial challenges. In particular, their inherent technical properties and requirements, their heterogeneity and hardware proximity must be taken into account [3, 5, 12]. Even at its core, IoT systems comprise very specific hardware peripherals, among which are timers [**?**], energy management components [50], entropy sources [32], and crypto chips [31].

The low quality of IoT system tests indicates that existing testing approaches still face difficulties overcoming these domain-specific challenges [18]. Existing gaps are partially attributed to a mismatched focus between industry and academia [5]. Falling in line with others that identified

the need for joined work between academia and industry to enable new testing approaches in real environments [18].

2.1 Multi-platform Testing Needs Automation

Developers of IoT systems can easily access various hardware features and specific functions when relying on the hardware abstraction layer (HAL) and APIs of an OS for embedded devices. This flexibility for the application developer comes at a cost for the OS developers. Each device-specific implementation must be maintained and thoroughly tested. Whenever a feature is added or an API is changed, all implementations must be tested and validated again. Overall, testing often accounts for more than 50 percent of development costs [4], providing a very strong incentive to automate testing steps.

Several popular IoT OSs are open source and driven by a diverse and distributed community of users and developers who report problems, provide new features, and fix bugs. It is therefore imperative for these projects to automate testing and verification processes for community contributions. Many other popular embedded OSs such as RIOT [10], Contiki [20], Mbed [6], and Zephyr [66] include testing infrastructure that provides static and unit tests. Some of these OSs have limited board simulation support but are not included in automated testing. RIOT, Mbed, and Zephyr support HiL testing on a subset of boards but struggle with heterogeneous multi-platform testing with different architectures and boards.

Aligned to this context, the goal for our automated multi-platform testing involves two major parts. First, an architecture is needed that enables testing unified OS interfaces for the same deterministic behavior across all supported hardware. This includes correct program flows, handling of parameters, API return codes, internal state, and acquire/release operations for resources. Second, the verification of physical IO signals to comply with our specification and involved communication standards, *i.e.,* interactions to the physical world and other devices. Most importantly this involves the need for a reference device to instrument our DUTs in a generic way.

2.2 Agile and Reproducible HiL Testing

HiL setups are commonly realized by connecting the DUT to external hardware, which represents subsystems that are part of the final product or its environment. A motor control DUT could, for example, be connected to a real motor or the circuit that powers the motor. In our scenario of OS development, the general-purpose DUTs do not have such a single specific use case. Therefore, we simply call the external subsystem that pairs with our DUT *reference device*.

An external HiL reference device can be realized in many ways but should be qualified with proven methods to have consistent behavior. Logic analyzers (**LA**) can be expensive and cannot inject signals or error cases. USB-based bus controllers generally miss the timing requirements and features needed for testing. Emulators and Simulation tools (**ES**) are often limited to a certain platform and provide only limited features compared to the real hardware. For example, Renode or QEMU emulators do not cover a large enough range of platforms and peripherals. External slave devices (**SD**) such as sensors, which use a specific bus or peripheral, are quick to deploy for smoke tests but cannot exhaustively test the API or failure cases. An **FPGA** solution would allow for extendable fine-grained control but can be more

Table 1. Feature comparison of solutions for external test devices: Logic Analyzer (LA), Emulator/Simulator (ES), Slave Device (SD), or FPGA versus commodity MCU.

Metric	LA	ES	SD	FPGA	MCU
HW Control	(✓)	(✓)	(✓)	✓	✓
Reproducible Setup	✓	✓	✓	(✓)	✓
Agile Adaptability	✗	✗	✗	(✓)	✓
Multi-Platform	✓	✗	✓	✓	✓

expensive, take more development time, and has limited off-the-shelf solutions. An **MCU** solution sacrifices some control over the buses for ease of development and is available off-the-shelf. Table 1 compares the relevant features of each solution indicating not supported, partially supported, or well supported.

Trying to keep up with fast IoT development cycles, agile processes such as Continuous Integration (CI), or applying checks on every change, are increasingly applied to embedded systems [40]. While systems evolve, they are likely to incorporate new bugs and testing needs to follow the agile development. Bugs may arrive with new features or new usage patterns that were not considered by existing testing infrastructure before. In both cases, the test tool must allow for adding test capabilities quickly to cover those new features and also prevent future regressions. To keep pace with rapid feature changes, we aim for a testing tool that is easy to acquire, adapt, and extend for the test tool users. We see these objectives supported by off-the-shelf hardware with an open-source implementation. Being able to look at the software implementation of the testing tool gives insight on how exactly tests are performed and what kind of constraints or implications are bound to it. Developers possessing very detailed domain knowledge on devices or software implementations are provided with a clear path on how to transfer this knowledge into automated HiL tests. Considering this, the MCU solution fits best given the constraints and domain knowledge of our target audience.

3 SYSTEM OVERVIEW OF THE HIL ENVIRONMENT

Our proposed testing environment consists of an ensemble of three components and tooling (see Figure 1(b)). All components and the associated infrastructure are designed for testing an OS for embedded devices. Our implementations separate tester and testee via an abstraction layer, which makes most implementations in our system agnostic to the OS and devices used.

The test node conducts the tests orchestrated by a common framework. It executes the test firmware on the DUT, and the external reference device, PHiLIP. PHiLIP serves as a qualified reference to test peripheral API implementations across the supported DUTs. All three devices interact through the wiring of peripherals and GPIOs.

To solve the challenges outlined in Section 2, PHiLIP allows for automated tests on various target boards, provides low maintenance and deployment costs. PHiLIP runs a single firmware that uses auto-generated code from a tool called MMM. The MMM processes a simple configuration file and easily extends the API of PHiLIP allowing for the development of test capabilities to be agile. We refer to Section 4 for details.

3.1 Testing Abstraction Layer

Coordination between PHiLIP and the DUT happens without altering the firmware. The peripheral APIs of the DUT are exposed through an interactive shell in a structured way. This structure is shared with PHiLIP allowing a coordinator to apply the same instructions to both PHiLIP and the DUT. Python wrappers simplify and unify the interface to the test node by providing classes with structured output enabling queries for statistics and benchmarking.

Coordination is not the only benefit of a structured testing language. Following structured testing guidelines allows developers to write tests with a unified process for handling and executing tests. Implementation of test logic independent of the firmware reduces the flash cycles that are needed for every test. Exposing the API comes at a fixed overhead cost. As the number of tests grows, the size of the firmware does not. Our goal is to keep the constraints off the MCU but leave them to the more capable test node, which can easily verify all advanced test options.

3.2 The Structured Testing Interface

The structured interface is provided via the python wrappers. This allows test API grammar to be defined and exploited. Five conventions are applied to both PHiLIP and the DUT to simplify testing:

C1 The communication method is synchronous commands and responses.
C2 The response information is a dictionary adhering to a schema.
C3 Every response contains a result which will either be success, error, or timeout.
C4 Optional data in return is of simple, predefined types.
C5 Time-critical steps should be wrapped in a single synchronous command.

By following these basic conventions code reusability is increased. Additionally, the test structure is unified and lowers the barrier for developers to understand existing tests and expand the test base.

4 PHILIP: A MODIFIABLE REFERENCE DEVICE

PHiLIP is a reference device for automated peripheral testing. It consists of a nucleo-f103rb or bluepill board with open-source firmware. PHiLIP can use the MCU hardware to collect physical information from a DUT similar to a logic analyzer but can also inject specific peripheral behaviors. PHiLIP uses a serial connection wrapped with `philip_pal` to simplify interaction and provide an intelligent shell.

As illustrated by Figure 1(b), a core component of PHiLIP is the MMM which feeds memory map information to PHiLIP firmware, the `philip_pal` interface, and documentation. `philip_pal` allows integration of a CI system and a developer to interact with the `philip_pal` shell and read documentation from the memory map.

4.1 PHiLIP Objectives

The goal of PHiLIP is to have an extendable reproducible solution for testing real-world characteristics of the peripheral APIs of embedded devices (*i.e.,* UART, SPI, I2C, ADC, DAC, PWM, timers, GPIO). The peripherals should be able to:

(1) Read and write bytes and registers via I2C and SPI.
(2) Support different modes and addressing.
(3) Allow for various speeds (I2C 10 kHz to 400 kHz, SPI 100 kHz to 5 MHz, UART 9600 Bd to 115 200 Bd).
(4) Support different register sizes (8 bit, 16 bit) with both big and little endianness.
(5) Track peripheral interactions such as bytes sent and received.
(6) Inject error signals and artificial delays.
(7) Estimate bus speeds for I2C, SPI with 5% tolerance.
(8) Log GPIO events with timestamps with a precision of 200 ns.

The pinout on PHiLIP is static so that rewiring is not needed when testing different peripherals. The languages (C and Python) and tools used to develop PHiLIP are familiar to developers testing with it. PHiLIP serves as a specific example of the general concepts for agile test tool development.

Qualification. In an agile development environment, qualification must occur frequently and with little cost. PHiLIP enables this by taking an inexpensive piece of hardware, automating qualification with a single set of more costly or rented tools, and then distributing the inexpensive hardware to all test nodes. This process is valuable in many situations, including (*i*) requiring many copies of reference hardware where purchasing off-the-shelf qualified equipment is too costly; (*ii*) working with remote developers that require physical access to expensive qualified reference devices; (*iii*) having occasional access to costly tools.

Application Core		
app_shell_if	app_reg	map_if

byte read/write

eg. read_reg(offset=10, size=2, access=user)

Memory Map		
defaults	access	typedef

Reusable Code

peripheral structures - - - - - - - - -   PHiLIP Code

eg. i2c.mode.data_nack = 1;

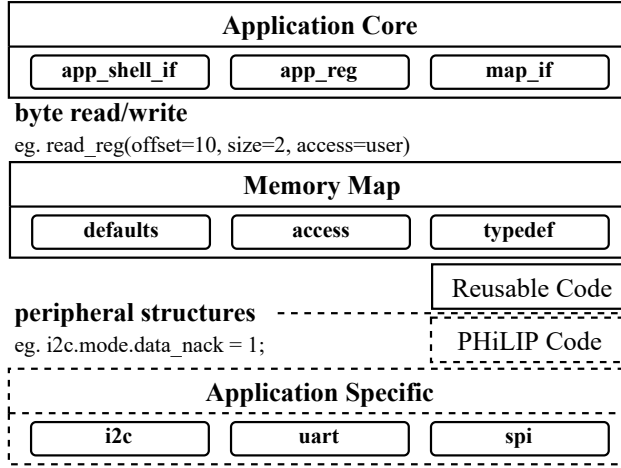| Application Specific		
i2c	uart	spi

Fig. 2. PHiLIP firmware module interaction with array access from the application core and structure access from the peripheral modules

Table 2. Basic PHiLIP firmware protocol commands

Command	Description	Example Return
rr <index> <size>	Read application registers	{"data": 42, "result": 0}
wr <index> [data]	Write application registers	{"result": 0}
ex	Apply changes to registers	{"result": 0}
-v	Print interface version	{"version": "1.2.3", "result": 0}

4.2 PHiLIP Firmware Implementation

PHiLIP firmware is designed to easily add peripheral functionality. It separates the peripherals and application-specific functions from communication, parameter access logic, and the memory map as shown in Figure 2. The application core code and memory map definition process of PHiLIP code is reusable in other projects and has versionable firmware components for structured communication. The application-specific code in PHiLIP implements the peripherals to test. Without optimizing for size, the PHiLIP application requires less than 36 kB and leaves at least 64 kB for future upgrades.

Application Core. null The application communication protocol provides a simple serial interface to read or write the parameters of the memory map as a byte array implemented in the firmware(see Table 2). The array is packed into a typedef structure allowing the C code to use descriptive names. To support multiple simultaneous changes in configuration, the parameter access functions require that the execution command is called after all changes to the memory map are completed. All parameter changes undergo access protection checks and safe handling (*e.g.,* disabling interrupts). Accessing the memory map data as an array is valuable for peripherals that can read or write registers such as SPI and I2C, and then verify with a different interface. PHiLIP contains 128 bytes of shared user data that can be accessed via both the peripherals and the app_shell_if. The size and offsets of the memory map can change and the version command is used to identify the correct mapping.

Time-critical Peripheral Event Handling. The PHiLIP firmware should allow all time-critical events to be stored for later access or prepared before the event occurs. PHiLIP uses peripheral
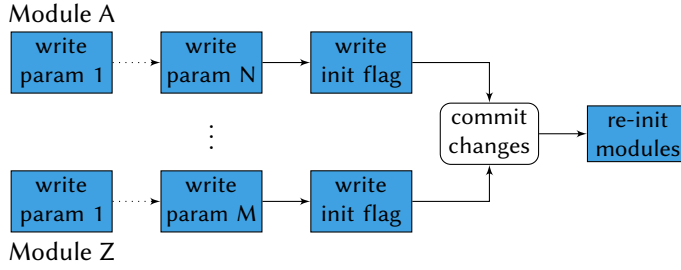
Fig. 3. Setting parameters for modules (A–Z) on PHiLIP.

hardware, interrupts, and polling to capture the information from events that occur. Using the MCU peripheral hardware allows a simpler implementation without requiring overclocking of PHiLIP with respect to the DUT. Specific peripheral behaviors can be triggered by preparing a state for an expected DUT action. If several time-critical events occur before data can be accessed, then the information can be stored as counts, sums, or as an array of events. For example, the GPIO module logs a timestamp per interrupt in a circular buffer that can be accessed after a series of rapid pin toggles.

4.3 PHiLIP Memory Map

To keep PHiLIP easily adaptable while maintaining a low memory footprint, we developed the MMM as a code generator for coordinating application information from a single configuration file. This reduces human errors when adding or changing runtime parameters, improves development speed, and the information can be fed into tests with various interfaces. The JSON configuration file follows a schema that can provide named packed structures to embedded devices, and allows for documentation of the register map. Structures and parameter properties such as type, array size, or testing flags are defined from the configuration file, as well as default values, access levels, and information for describing the parameters. The registers are serialized and can be accessed as a structure (by name) or byte array (by address).

Describing the memory map based on parameter names with the respective types and sizes combines the versatility of named access with the efficiency of serialized packed memory. The interface only needs to translate the name to an offset and size to get the information. The simplicity of implementing only read and write register commands to deal with each parameter reduces bugs on the embedded device.

The generated output of the MMM is C style data and consumed by the firmware application. By convention, parameters can be changed by writing registers, similar to MCUs or sensors. To initiate a change in properties, for example, alters the I2C address on PHiLIP, an initialization bit should be set before calling the command to execute changes. This allows for a peripheral to be configured only once, preventing possible initialization sequence errors. Figure 3 shows an example of changing many parameters and executing the changes.

4.4 The Abstraction `philip_pal`

`philip_pal` provides a Python wrapper that implements the structured testing interface outlined in Section 3.2. It takes the basic firmware protocol commands and maps bytes back to structure members using a CSV file generated from the MMM. `philip_pal` first checks the version then correlates that version to a specific mapping. As a result, the memory map can easily change or add parameters while maintaining backward compatibility for named access. `philip_pal` also provides

Table 3. Example showing how data traverses PHiLIP abstraction layers via name-mapped parameter access

Layer	Request	Response
`philip_pal`	`self.read_reg("i2c.r_count")`	`{"cmd": ["read_reg(i2c.r_count,0,1)"], "data": [1], "result": "Success"}`
Serial port	`rr 334 1`	`{"data": 1,"result": 0}`
PHiLIP FW	`printf("{\"data\":%u,\"result\":0}\n", read_regs(334,1));`	

the documentation of functions, validation of parameters, default arguments, and parsing of values, keeping this out of PHiLIP firmware. There are over 270 named fields that can be written or read in the map that corresponds to settings for parameters implemented in the PHiLIP firmware. For example, the `i2c.r_count` contains the number of bytes read from the I2C register. The logic is implemented in the firmware, counting the number of I2C data ready hardware interrupts that occur and storing it in the mapped C structure.

Along with a Python class, `philip_pal` provides a shell to assist developers in manual debugging with features such as autocompletion, self-documentation, and helper functions. Table 3 shows name-based parameter access via `philip_pal` being converted to addresses and offsets. `philip_pal` looks up the offset and sizes from the named parameter based on the versioned memory map indicated by PHiLIP, then writes the command via the serial port to PHiLIP. PHiLIP in turn, either prepares, applies, or reports the parameters. In this example, reporting the requested value `i2c.r_count` via the serial port, which then gets parsed by `philip_pal` according to the datatype indicated by the map. The `i2c.r_count` parameter in PHiLIP gets updated via I2C reads from the DUT and are only fetched from PHiLIP by the host computer when needed with a `read_reg("i2c.r_count")` command.

4.5 Adding New Testing Capabilities to PHiLIP

The agile process to adopt a new test capability takes five steps:

(1) Identify the parameter(s) needed based on a bug or issue.
(2) Add the parameter into the MMM configuration file and generate a new map.
(3) Implement functionality on the given module.
(4) Qualify the parameter on PHiLIP.
(5) Release the new firmware and Python package.

If PHiLIP cannot provide a way to either measure or induce the state where an issue occurred then a parameter is added to the memory map configuration file, *e.g.,* forcing a data NACK on the I2C bus. The parameter has a descriptive name and other valuable information such as access level or unique flags. A new memory map is generated that validates the JSON syntax and schema, recalculating the sizes and offsets of each parameter. C code and a CSV file are exported to the sources. Functionality is implemented in the C code for that given module, *e.g.,* if the `i2c.nack_data` parameter is enabled PHiLIP should set the `I2C_CR1_ACK` bit to 0. An automated qualification procedure based on standard tools then validates the expected behavior. Thereafter, the firmware can be released along with a Python package containing the new memory map data.

5 PHILIP PERFORMING MULTI-PLATFORM HIL TESTING

Automated and platform-independent tests are created using PHiLIP for the DUT (see Section 3). The design of the testing environment takes advantage of the structured testing interface and existing tools where available, adding custom implementations where needed. The tests can be run by developers or integrated with a CI system.

Table 4. Comparison of build sizes on nucleo-f103rb board

Test	ROM [Bytes]	RAM [Bytes]
periph_i2c	19,780	2,656
periph_gpio	13,828	2,520
periph_spi	20,268	2,816
periph_uart	20,672	4,536
periph_timer	16,624	2,548

5.1 Testing RIOT OS Peripherals

The firmware is implemented in C on RIOT OS, which enables the multi-platform testing based on the DUT HAL. Writing tests in this way can save flash cycles and limit code size as more tests are added. Table 4 shows build sizes of the peripheral-based tests. We group all tests that share similar properties in three groups, infrastructure testing, bus testing, and timer testing.

Infrastructure Testing. Errors can occur from infrastructure components or setup. Tests ensure that the infrastructure is operating properly within the testing system. There must be a connection to the DUT and PHiLIP, opening a connection and sending a sync message can resolve this. The wiring must be correct so toggling the GPIO of each wire can be used to verify the wiring. The flashing tool for the DUT must be functioning, reading a descriptor of the firmware can be a way to check the correct firmware is flashed on the DUT.

Bus Testing. A peripheral bus is stateful and involves exchanges between devices. Since hardware registers need to be set and cleared by external interactions to introduce persistent states or race conditions, simply completing code coverage is not exhaustive. PHiLIP can be configured to inject and check for errors related to the time and state of SPI, UART, and I2C buses. For example, it can alter clock stretching, flip bits, or record total interaction time. When the DUT executes an action to be tested, PHiLIP interacts with the DUT according to the configuration and collects metadata on the interaction. This information can be queried later from PHiLIP via the host. We consider the following five basic tests:

Initialization Tests: Initializing and acquiring any bus lock including powering on the hardware.
Usage Tests: Read or write operations using the default configuration.
Mode Tests: Varying the modes and settings of the bus and ensuring interactions are correct.
Negative Tests: Check improper configurations return appropriate error messages.
Recovery Tests: Check bus recovery after forcing an error state.

```
1  philip.write_and_execute("i2c.mode.nack_data", 1)
2  r = dut.i2c_read_reg(PHILIP_ADDR, 0)
3  assert r["result"] == ERROR_RESULT
4  assert r["data"] == -EIO
```

Listing 1. Example testing indication of I2C NACK condition

Listing 1 shows an example of PHiLIP injecting a challenging behavior for a test. First PHiLIP is prepared to NACK only data bytes in an I2C frame. Then the DUT executes an I2C read register command on PHiLIP. After the I2C is finished, the DUT returns the result to the test node. The expected result is an error with the -EIO error code indicating a NACK on the data has occurred.

```
1  r = dut.i2c_read_reg(PHILIP_ADDR, 0)
2  assert r["result"] == SUCCESS_RESULT
3  assert r["data"] == philip.read_reg("user_reg", 0)["data"]
```

```
4   assert philip.read_reg("i2c.r_count")["data"] == 1
5   assert philip.read_reg("i2c.w_count")["data"] == 1
```

Listing 2. Example for asserting metadata of I2C operation

Listing 2 collects metadata of an I2C read register command from PHiLIP. First, the DUT executes an I2C read register command on PHiLIP and stores the response. The result should be successful and the data of that register should match data from PHiLIP. Additional properties of the I2C read register command are verified such as the number of bytes read from and written to PHiLIP.

Initialization tests can catch bugs with powering and configuring the peripheral clock which may prevent startup. Usage tests verify the correctness of basic operation in corner case conditions like maximum transfer size. Mode test expose unimplemented or wrongly implemented configuration options. Negative tests probe if a module handles unexpected conditions appropriately. With recovery tests it is possible to find bugs that occur only after rare fault conditions.

Timer Testing. Timer tests are challenging, truly concurrent operations. Events are generated at very high rates and require precise timing. Isolated testing solely on the DUT is infeasible as a reliable internal reference time is often missing. Complex code for measuring and analyzing corrections on the DUT induces side effects, which cannot reliably be quantified nor compensated across platforms. Using pure command-response communication between test node and DUT does not allow for precise timing either, because the communication channel (*e.g.,* UART) and command parsing induce significant delays and jitter. Therefore, a clear separation between test overheads and time-critical hardware instrumentation is needed.

With PHiLIP, this is achieved by asynchronously logging GPIO signal events with timestamps. DUT timer operations are instrumented for signaling via designated pins. The pins are dynamically configured on PHiLIP before the test execution. Thereafter, timestamped GPIO-traces are acquired via `philip_pal`, and measurements such as frequency, drift, jitter, and accuracy are calculated and compared to tolerance values. Respective limits on timing accuracy of the current PHiLIP implementation are detailed in Section 6.2.

Figure 4(a) shows sample results of a cross-platform test on timer accuracy. The threshold given as 170 PPM (Parts Per Million) combines datasheet limits for crystal oscillator accuracy, stability, and aging values of the DUT and PHiLIP. Two of the tested boards clearly do not comply with this specification. Likely sources for such errors are incorrectly configured clock sources (*e.g.,* using an internal resistor-capacitor oscillator instead of a crystal), improper trimming configuration of the crystal oscillator, or faulty prescaler configurations.

A further example grounded on precise microsecond-scale measurements is the assessment of worst-case delay limits under specific edge cases, *e.g.,* n virtual timers triggering at the same time. Figure 4(b) exhibits the upper bound for the timer scaling behavior of virtual (software multiplexed) timers on nucleo-f091rc board. The plot shows a linear scaling behavior, indicating a maximum delay close to 300 μs when ten timers are scheduled for the same target time.

5.2 Developer Testing Locally

The automated test suites can be run by developers locally if the boards or wiring are not supported in the CI. The setup requires PHiLIP firmware to be flashed on a nucleo-f103rb or bluepill board. The testing repository along with python requirements must be installed. Only the wiring needed for the specific test must be connected from the DUT to PHiLIP. If wiring differs from the CI boards then it must be input to the test environment. A single make command allows the tests to be flashed and executed.
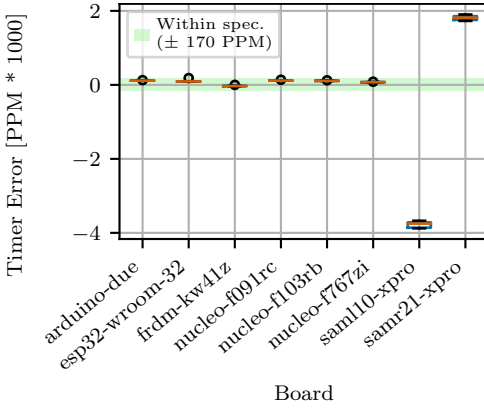
(a) Timer errors compared to oscillator specification threshold across multiple different boards
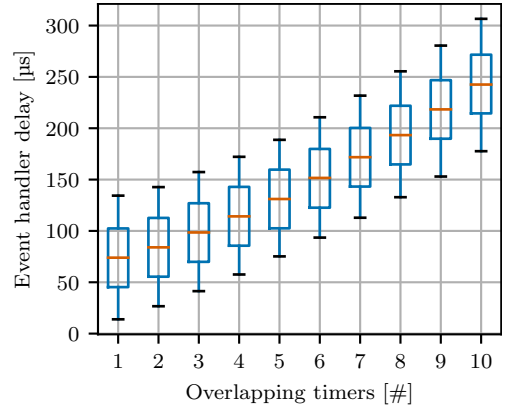
(b) Delays of timer event handler on nucleo-f091rc when multiple timer targets overlap

Fig. 4. Examples of two timer tests, accuracy, and delay

5.3 Flexible CI Integration

PHiLIP can be used from off-the-shelf components, however, a custom board was created to ease CI deployment. A CI test node consists of the custom board that provides connections between a Raspberry Pi and PHiLIP and a standard 20 pin ribbon connector to the DUT (see Figure 5). This allows for the simple wiring of DUTs in different form factors without developing specific breakout boards. A test is provided to ensure wires are correctly routed. The custom board also provides basic power measurement and control tools to help with low power testing, some protection circuitry, and signal conditioning.

Each test node is responsible for testing a single DUT in the CI setup. The 1:1 ratio of test node to DUT was chosen over a 1:n ratio for multiple reasons. Managing the test node environment is simplified. Downtimes of a single test node become less critical. DUTs do not need to share USB bandwidth. Computationally heavy tasks on the test node will not affect other running tests. Building is done on separate servers with docker and does not require any special hardware.

In our current setup, tests run on 22 different boards [1] of various form factors, vendors, and CPU architectures. We test nine CPU architectures (AVR, cortex-m0+, cortex-m0, cortex-m3, cortex-m4, cortex-m7, cortex-m23, esp32, esp8266) three peripheral buses (I2C, SPI, UART), timers, and GPIO.

At the software side, our CI environment is based on three common open-source tools. (*i*) Jenkins steers continuous integration across multiple test nodes. (*ii*) Robot Framework [2] implements the test logic with its keyword-based syntax that can easily be extended and adapted to a specific domain. (*iii*) Ansible is used to add and manage each test node. Text execution is orchestrated by Jenkins allowing tests to be triggered manually, every night, and on every change of the test code.

6 EVALUATION

PHiLIP has powered continuous HiL testing in RIOT for over a year with over 200 stable testing cycles. 1519 tests are executed per night for 22 unique boards, taking less than 45 minutes. In comparison, a developer may need the equivalent time to manually set up and execute a test on a single board.

---

[1] up-to-date board list: link omitted for double-blind review
[2] https://robotframework.org

Fig. 5. Up to 8 test nodes can be packed in 19" server rack

Table 5. List of bugs discovered by PHiLIP during an I2C rework. CWE (Common Weakness Enumeration) provided by https://www.mitre.org/

Family	Boards	Priority	Severity	CWE	Description
sam0	5	medium	moderate	474	Inconsistent Function Implementation
sam0	5	medium	moderate	394	Unexpected Status Code or Return Value
atmega	3	low	major	480	Use of Incorrect Operator
cc2538	5	medium	major	460	Improper Cleanup on Thrown Exception
stm32	18	low	minor	394	Unexpected Status Code or Return Value
stm32	18	high	major	835	Loop with Unreachable Exit Condition

In the following, we evaluate PHiLIP in detail from five perspectives. (*i*) A case study of using PHiLIP during a large I2C rework; (*ii*) the timing constraints of measuring and executing tests; (*iii*) system overhead introduced by tools, the MMM, and philip_pal; (*iv*) the memory consumption exposing an API vs. hardcoding test cases; (*v*) the guaranteed lifetime of a board due to flash endurance while testing nightly.

6.1 Impact: The I2C Rework Use Case

PHiLIP was used during a two-month rework of the I2C peripheral in RIOT. A small python script[3] was used to run automatic tests on developer machines. This required wiring of the I2C pins for each DUT to PHiLIP. The script initialized both PHiLIP and the DUT, then ran a number of checks with different parameters such as varying addresses, flags, sizes, *etc.* to check the new API being introduced. It discovered and prevented the bugs shown in Table 5. Bugs with high priority typically mean they exist in master or are about to be merged into master. Bugs with higher severity affect the current tests and drivers whereas minor severity bugs are edge cases that may occur in external applications.

The hidden CWE474 bug was discovered on the sam0 platform. It was a physical read of an extra byte, leading to additional time used on the I2C line as well as potentially increasing a register pointer on the secondary device. Since the byte was discarded in software, other tests with sensors

---

[3]link omitted for double-blind review

Table 6. Comparison of timing constraints for different instrumentation methods provided by PHiLIP.

Measurement Method	t_{min}	t_{jitter}
Timer Capture DMA	200 ns	28 ns
Timer Capture IRQ	1 us	200 ns
GPIO IRQ sampling	10 us	600 ns

were not able to detect the failure. The `i2c.r_count` parameter provided by PHiLIP about how many bytes were read from the I2C bus uncovered the bug for the whole sam0 platform. The test in Listing 2 discovered that the read byte count was 2 instead of 1. A CWE394 bug was also discovered in the process. In this case, the return value was always successful, even when failing, showing the importance of negative tests.

Testing on the `atmega` platform, a CWE480 bug caused I2C register writing to fail. This was due to missing an inversion of the bitfield when checking a status bit causing incorrect state transitions. The CWE460 discovered on the `cc2538` platform caused lockup after reading from a missing address. This was due to the internal hardware not clearing its error state before issuing another command. Both CWE394 and CEW835 were discovered on the `stm32` platform prevented multiple I2C register writes. This was due to stop signals being sent when busy causing the state to hang when attempting to issue another write command.

6.2 Timing Constraints

PHiLIP Qualified Timing Constraints. Temporal accuracy is a relevant constraint for PHiLIP as the timer tests described in Section 5 for instance, need high-resolution measurement methods. PHiLIP's capabilities are evaluated against accurate measurement equipment as part of the qualification procedure outlined in Section 4.1. This is done by setting the measurement equipment to toggle a pin at different rates and verifying the readings of PHiLIP. The limits of different time capture methods supported by PHiLIP are summarized in Table 6. The minimum time between two consecutive logging events is denoted by t_{min}. The maximum accepted jitter of the time measurements is listed as t_{jitter}. DMA instrumented timer capturing performs best but, due to MCU hardware limitations, can only capture either rising or falling edges. The Timer interrupt request (IRQ) variant can be triggered by rising and falling edges but relies on slower CPU instructions to read timer values. Both variants using designated timer hardware allow high precision but restrict the number of events by the associated buffer size to 128. The GPIO IRQ sampling uses interrupts of GPIO hardware to log timestamps directly to the memory map. With both edges and virtually infinite sampling duration, this gives the highest flexibility but limits precision.

Command Timing Constraints. PHiLIP solves the need for strict timing requirements between the test node and DUT by measuring time-critical parameters locally and reporting them later to the test node. The completion time varies depending on the command and communication method.

Figure 6 shows the execution time of various commands as generated by Robot Framework which was collected from the test artifacts produced in the nightly CI runs. The time for the python instruction includes sending the command from the test node to the DUT, its execution, returning results to the test node, and finally parsing it. The framework overhead is the time Robot Framework takes to log the steps and check the results. This depends on the speed of the test node, *i.e.,* a relatively slow Raspberry Pi 3.
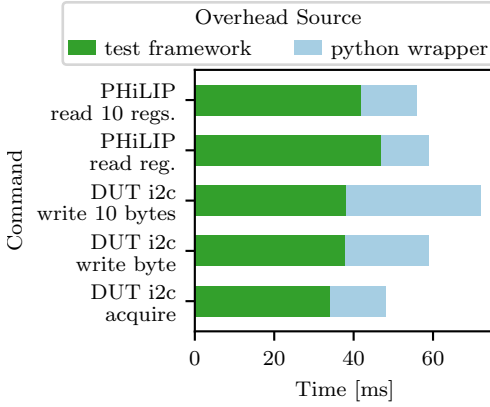
Fig. 6. Timing of framework overhead and command time for a nucleo-f103rb using the periph i2c test.
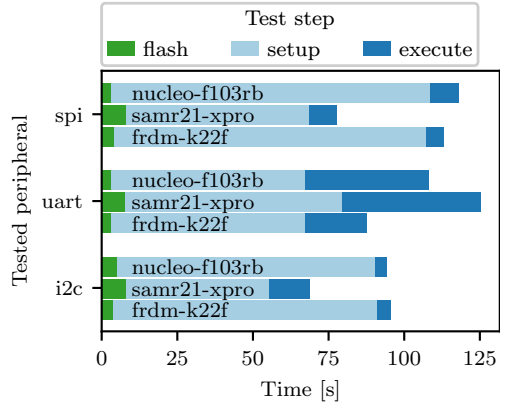
Fig. 7. Average test step time per board for a given tests.

These times help determine the limits using synchronous commands and when time-critical events should be offloaded to a grouped command. With this setup, anything that requires timing below milliseconds should be offloaded.

6.3 Test System Overhead

Duration of Tests. We present results for three test suites on three different boards in Figure 7 selected to show the largest variations of time. Each value is averaged over 30 nightly CI runs though time variances between runs are negligible. We focus on the time the microcontroller is in use excluding timing metrics related to the test node. The setup and execution time are captured from the Robot Framework output whereas the flash time is taken from the CI logs. The flash time depends on the binary size and the speed of the flasher. Some boards have slow flashers due to low baud rate UART communication. Flash times range between 3 and 10 s.

The test setup time is the longest across all cases. A setup phase occurs before each test and resets both the DUT and PHiLIP, then establishes a connection by reading the expected firmware version. The reset times of a DUT vary due to board-specific delays introduced by a failed initial connection attempt due to the spurious bits of the UART peripheral being reset. Depending on the bootloader, a silent period may be needed before connecting, for example, the Arduino bootloader requires 2.6 seconds after resetting.

During the test execution time, the DUT and PHiLIP are interacting via peripherals. This includes the time to send a command from the test node to the DUT, to execute the API call on the DUT, to return the result to the test node, and parse that result. The UART tests show a longer execution time due to the DUT sending a large amount of data at the baud rate limit. The frdm-k22f board finishes faster as it skips tests of unsupported modes.

Memory Map Overhead. The cost of using the MMM can be shown by evaluating both the memory footprint and the speed at which the data can be accessed. Table 7 shows the differences in the overhead of PHiLIP firmware between address-based access using the size and offset versus name-based access using a name and decoding the information in the firmware. PHiLIP's memory map contains 273 parameters taking a minimum of 1841 bytes, a total of 2048 bytes with padding.

We compare a variant of PHiLIP firmware that allows the memory map to be stored inside the firmware instead of using philip_pal to decode the map. Reading and writing parameters by name rather than address-based reads with size and offset increases flash size by 9352 bytes. The ability to
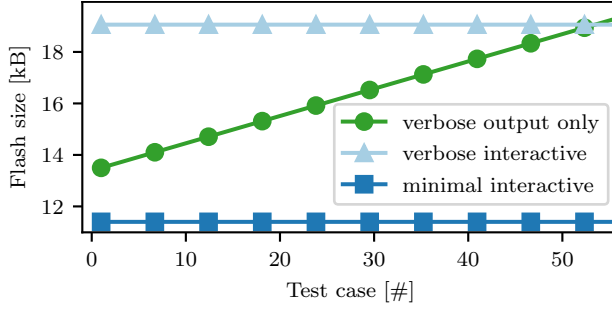
Fig. 8. Flash memory size for DUT test firmware vs. the number of test cases.

access the map properties through firmware adds 384 bytes but allows `philip_pal` to use the map without previous information. Accessing with names also adds to the response time due to increased parsing complexity. PHiLIP is instrumented to toggle a GPIO pin when a command is received resulting in 22.4 μs to parse a `rr 0 1` command mentioned in Table 2, that reads one byte from the user register. Reading with a name, for example, `r user_reg 0`, results in 38.55 μs parsing time.

6.4 Memory Usage

The memory footprint is evaluated by comparing build sizes for three variants of the DUT interface, differing in interaction functionality and verbosity. The two interactive variants communicate synchronously between the test node and the DUT, where the first one uses a text-based, human-readable shell and the second uses more concise binary encoding. We refer to both cases as *verbose interactive* and *minimal interactive* respectively. The third, self-contained variant still outputs verbosely but does not require input from the test node. Therefore, we refer to it by *verbose output only* in the following.

Figure 8 relates the memory usage of all three approaches based on an I2C test firmware for the nucleo-f103rb considering an empirical average memory size increment of around 106 bytes per test case. The reduced memory usage of the *minimal interactive* case always has an advantage over the more verbose alternatives. But even the *verbose interactive* version becomes more memory efficient than the self-contained firmware after surpassing the break-even point around 53 test cases. This shows that, despite its additional code to expose DUT functionality, offloading test cases to the test node saves memory for a large number of test cases.

6.5 Cost of Testing on Hardware

We analyze the cost associated with testing on real hardware in Table 8 in terms of capital expenditure (CAPEX) and operational expenditure (OPEX) by considering the main expenses of two usage scenarios for PHiLIP: the minimal developer desktop setup overhead as shown in Section 5.2 and the

Table 7. Comparison of memory and parse time for address-based parameter access vs. named-based parameter access.

Access	Flash (kB)	Parse Time (μs)
By Address	31.4	22.4
By Name	40.8	38.6

overhead from automated testing via dedicated CI infrastructure as shown in Section 5.3. Common to both scenarios are the costs related to the DUT which are displayed separately.

DUT. The average cost of our deployed DUTs is 40 €, with the most inexpensive board being the esp8266-esp-12x at 7 €, and the most expensive board being the frdm-k64f 136 €. The maintenance interval of the DUT is determined by its flash endurance. To obtain a realistic estimate on DUT lifetime, we survey manufacturer datasheets of 7 unique boards supported by the RIOT CI (arduino-due, arduino-mega2560, frdm-k22f, nucleo-f103rb, remote-revb, samr21-xpro, slstk3401a). The worst-case and most common flash cycle range is 10k. The OPEX of the DUT consists of the replacement cost that occurs after 1250 full test runs consisting of 8 flashes per run.

Testing on the Desk. Based on empirical values, students working with RIOT OS and PHiLIP need approximately 2 hours to initially set up the hardware and testing environment which is reduced to 30 minutes once they are familiar with the process. The operational expenditure for running the full test suite with the minimal setup is heavily dominated by only this labor time, effectively marginalizing other operational costs in this scenario. The capital expenditure overhead is the cost of a nucleo-f103rb kit, some wiring, and the time to flash PHiLIP firmware on it.

CI Testing. The operational expenditure per CI run consists of build server time, around 2 minutes per run[4], and labor costs for maintenance of replacing the DUT, which take around an hour per maintenance interval. Compared to the cost of the DUTs, the initial PHiLIP and test node investment fits in the cost range of the DUTs. The operational costs of the CI also are similar to the cost of replacing the DUTs at the maintenance interval.

Cost Reduction Potential. The flash cycle limitation could be overcome by executing tests from RAM at the cost of accurately representing target devices in production. Execution from RAM would additionally require per-target customization of linker scripts, limit the allowable code size, and alter timing due to missing flash wait states. The CAPEX of the CI could be reduced by using more DUTs per test node, however, it would only pay off with a large number of DUTs due to the additional complexity implying further maintenance and development costs.

7 RELATED WORK

Testing with HiL fills important gaps as discussed in Section 2.

On-target Testing. Strandberg [57] points at significant time barriers due to the duration of complex tests. Tight hardware coupling of the tested software further limits the availability of the environment as it requires direct hardware access. Orthogonal to our solution, Strandberg focuses on a management layer for optimized test selection and allocation of multiple networked devices. We target testing HALs with physical interactions, therefore, focusing on test execution with generic hardware instrumentation.

Mårtensson [40] considers test execution on real hardware as a hurdle for CI/CD integration because of limited access to custom target devices. Even though simulation-based testing can take

---

[4]Based on 4 CPU, 16 GB memory, On-demand from https://calculator.aws/#/createCalculator/EC2

Table 8. Cost breakdown of HiL usage for desktop setup overhead, CI overhead, and base DUT cost.

	Desktop	**CI**	**DUT**
OPEX	30 [$\frac{mins}{run}$]	0.05 [$\frac{€}{run}$]	0.01 to 0.12 [$\frac{€}{run}$]
CAPEX	10 [€]	80 [€]	7 to 136 [€]

over selected test tasks, *e.g.,* checking functional stability [25], it is not suited to test whether the software will correctly operate on hardware. Our solution overcomes these problems with on-target testing and shared access through CI, nightly, and on-demand testing workflow integration. Instead of verifying a specific application on a single device, we perform multi-platform testing on a wide range of heterogeneous DUTs to guarantee platform-independent firmware has deterministic behavior on all target devices.

Tools for Testing Embedded IoT Systems. Specialized solutions cover domains from the custom silicon layer [15] and printed circuit board schematics [34] to system modeling, simulation, and automatic code generation [41]. Common tools for testing usability, reliability, and compatibility of IoT devices, however, do not provide a solution for automated testing of tightly hardware-coupled software [43]. *IoT-TaaS* [33] deals with coordination of interoperability and conformance testing of higher layers (*i.e.,* network protocols). Testbeds are a common approach to test embedded software on real hardware under realistic conditions and in larger setups [2, 24, 28, 36]. While they are mostly used for experimentation and performance evaluations of protocols and applications, some also explicitly focus on the integration of heterogeneous multi-node setups [35, 62]. Aiming for improved reliability, Woehrle *et al.* [65] propose a distributed testing framework that combines simulation and testbed support tailored to the development process of wireless sensor networks. Tools such as *Greentea* [7], *Icetea* [8], *LAVA* [38], and ICAT [16] focus on deployment, execution, or operative management abstractions, leaving measurement of physical hardware interactions out of scope, or target mobiles instead of constrained IoT devices.

Izinto [45] is a pattern-based test automation framework for integration testing. This solution does not require technical knowledge about the tested system but is limited to user perception use cases, whereas our work targets low-level misbehavior.

The Importance of Testing Peripheral Abstractions in Embedded Software. System standards such as POSIX are not applicable for deriving test items for hardware interfaces [58]. Therefore, Sung *et al.* [58] contribute a test model, defining a list of test features for interfaces of OS and hardware layers. Seo *et al.* [55] used this model to show that the likelihood of finding errors is significantly higher in interface functions that cross heterogeneous abstraction layers. Their analysis further indicates that bugs in this type of code are harder to discover with classical unit testing approaches. Justitia [56] was developed using the same model to automate the identification of interfaces to be tested together with generating and executing test cases. The employed fault detection method is tailored very closely to a specific target platform and only applies to time-invariant errors (*e.g.,* in memory management and allocation). It poses significant limits when timing critical operations with true hardware concurrency and connections to the physical world are considered. Although the DUT instrumentation principle is orthogonal, our focus on peripheral abstractions explicitly targets the important areas where execution flows cross layer boundaries.

Feng *et al.* [22] automate implementations of peripheral models that approximate the behavior of real peripherals for use in emulation-based fuzzy testing. In contrast to our work, their approach aims for generic testing of the firmware on top of peripherals and is therefore neither able to model arbitrary hardware, DMA peripherals in particular, nor correct physical layer interactions.

Testing Physical Interactions With HiL. HiL solutions are commonly used in the automotive domain [19, 23, 30, 63]. Vejlupek *et al.* [63] propose a hybrid HiL approach, *i.e.,* stimuli signal injection allows them to use a parameterized model instead of a complex physical test setup. Keränen *et al.* [30] investigate benefits of MBT in HiL setups. The employed *online* MBT approach generates test input on the fly and injects random test steps. Randomness injection was also shown to be useful for testing interrupt-driven software by covering execution paths that are highly unlikely to occur [47]. Combining MBT and code generation was previously also demonstrated to support verification

of model-based embedded systems [59]. Downsides inherent to MBT still apply, though: a correct model of the DUT and additional development effort are needed, and vulnerability to human error exists.

Virzonis *et al.* [64] demonstrate how HiL, simulation, and CI can transform the linear development process of embedded control systems into iterative cycles. Their work focuses on a process and local setup for the development phase whereas our work targets a distributed CI setup for the testing phase.

Muresan and Pitica [44] indicate that Software in the Loop is appropriate when a control algorithm is the test item. In such cases, simulation benefits such as full parameter control and virtualization of timing critical aspects outweigh the downsides of not considering any hardware-related properties.

Common between the discussed HiL implementations is their usage in a very limited and well-defined domain for single specific purpose DUTs. In contrast to that, we aim for a system that tests a wider range of general-purpose devices.

Emulation and Simulation. Embedded systems are sensitive to their physical environment making them cumbersome to test and debug. Simulation and emulation solutions avoid these problems to make testing of embedded software easier for isolated aspects. They promise benefits like reproducible experiments, major scalability improvements through parallel execution, faster operation by controlling virtual time, and many more.

QEMU [46] is a well-established machine emulator that provides many features for virtualization, user-mode emulation, up to full-system emulation. Even though it can emulate constrained ARM MCUs as well as full-sized x86 and PowerPC systems it is more targeted towards the latter class of systems.

Renode [48] is an open-source software development framework for running, debugging, and testing unmodified embedded software that is more focused on small embedded devices. It employs simulation of the CPU in addition to its peripherals, externally connected sensors, and the communication medium between nodes to make multi-node system testing more reliable, scalable, and effective.

The benefit of such emulators is evident for cross-platform development and for providing an accurately controllable execution environment with a high degree of flexibility. However, it is important to highlight that there are major obstacles that make it impossible to use these existing solutions for the same purpose targeted by our approach. These obstacles mostly stem from two dimensions: the number of compatible devices and the basic applicability to test the behavior of low-level driver code with respect to the physical world.

RIOT as our test subject supports both aforementioned emulators [49], but none of them can provide the required level of compatibility. To evaluate how far away current state-of-the-art emulation is from serving our intended purpose a look at the current support of devices and peripherals provides guidance. According to the QEMU documentation, only two different embedded microcontrollers of the Cortex-M CPU family are supported[5]. For Renode, the situation appears much better at first sight, with 9 targets being supported of our currently deployed 22 MCUs. Though, as of now, feature availability puts up further limits with only two targets supporting all peripherals that our setup is testing.

Apart from existing solutions not being available for a relevant number of target devices there is also a fundamental mismatch of objectives compared to our solution. Per definition, the objective of an emulator is to mimic or mock (*i.e.,* emulate) the behavior or a specific system. While this covers the behavior of the (emulated) hardware towards the firmware, it does not cover the internal state of the real hardware and its interactions to the physical world. In fact, existing emulators deliberately deviate internal system behavior to improve emulation performance [21]. Therefore, they lack the

---

[5]https://wiki.qemu.org/Documentation/Platforms/ARM

required level of simulation granularity to test for correct peripheral interactions with the physical world. Albeit a hypothetical full-fledged *simulator* would be capable of testing these aspects, to the best of our knowledge, currently no such simulator exists that covers both important aspects of device compatibility and basic applicability. An accurate simulator will always require a high-quality model. But the primary source to get detailed descriptions of hardware peripherals are manufacturer datasheets and reference manuals which differ considerably in terms of provided details and quality. Additionally, they are subject to human interpretation and undiscovered hardware errata, arguably making it extremely challenging to derive exact models of hardware peripherals. Implementing new device models would require non-negligible time whereas adding a new CPU with PHiLIP only requires basic knowledge on how to provide peripheral configurations and wire the device.

We conclude that compared to emulation and simulation our approach has a conceptual advantage in terms of device compatibility, applicability to find errors in low-level driver code, and scalability when adding new devices. With a scaleable way to qualify hardware behavior, our solution further contributes an important building block for future development and evaluation of generalized cross-platform MCU peripheral simulators.

Test Generation. Research on test generation provided several methods to attain high coverage and unveil hard-to-find bugs with automated software testing. Combinatorial, search-based, and MBT allows optimizing test selection in order to slim down huge parameter spaces into manageable numbers of cases that still provide good coverage. Symbolic execution is particularly interesting because of its ability to implicitly test huge sets of values by considering execution paths symbolically via constraints instead of concrete variable values. A vast amount of tools is available from academia and industry that showed practical relevance and notable impact on automated software testing [14]. However, there are still multiple open problems related to *path explosion*, *path divergence* and *complex constraints* that complicate employing symbolic execution for testing real production software at scale [4]. For the constrained devices class we are interested in, symbolic execution was previously shown to be useful for testing firmware implementations of networked nodes [52]. Even though that approach can detect even inter-node bugs in a network of embedded devices, it only considers network and application layer behavior, leaving MCU peripheral code out of scope.

In our specific domain — testing peripheral drivers including their interactions with the physical world — complex constraints, environment interactions, and device-specific characteristics can be expected to strongly interfere with symbolic execution, as is already visible with virtual devices [17]. The approach developed by Cong *et al.* [17] employs symbolic execution for virtual QEMU devices via the KLEE[13] engine. It is evaluated with five different network devices and methods are proposed to reduce the manual implementation effort for the required device-specific models. As indicated by the authors, however, the manual effort to enable symbolic execution for devices can not be completely avoided. This leaves major blockers regarding applicability to our domain: a widely applicable set of accurate software models for MCU peripheral devices does not exist, and platform-specific hardware interfaces towards MCU-internal peripherals come in almost arbitrary variants making it very hard to automate model generation.

8 DISCUSSION, CONCLUSION, AND OUTLOOK

In this paper, we presented and evaluated a concept and implementation for HiL testing low-end IoT devices. Part of the solution is two crucial components: PHiLIP, an open-source external reference device, and a tool-assisted process that simplifies verification of peripheral behavior on different embedded hardware platforms. In the following, we take a retrospective look at design decisions, report on lessons learned, and address shortcomings with potential solutions worth considering in future designs.

8.1 Concept Validity

The evaluation results show that PHiLIP serves as a versatile tool to instrument currently 22 heterogeneous DUT devices with significant extensions planned in the near future. Approximately 67% of the peripheral implementation variants (I2C, SPI, UART) supported by RIOT OS are covered. The timing analysis confirms the test throughput is high enough to increase the current $\approx 1,500$ nightly tests by an order of magnitude. Future work could reduce the overhead of the test firmware on the DUT further by leveraging a more efficient serialization of commands and responses.

Employing the Memory Map Manager (MMM) proved beneficial to simplify maintenance and enable resource-efficient operation in terms of processing time and memory. The MMM saves time compared to manually altering and reviewing the memory map when extending and maintaining PHiLIP. Instead of using a custom JSON-based configuration format, the memory map could in the future be encoded with the widely used SVD format [9]. This would allow using tools developed for SVD files alongside the MMM.

Exposing low-level peripheral APIs via DUT commands enables offloading test logic to the test node, keeping the size of test firmware static. Offloading allows operation delays to stay below the order of 100 ms. For time-critical operations, we showcased how asynchronous signal triggers of the DUT can be captured by PHiLIP with a measurement precision on the microsecond scale. Our practical case study during the I2C rework confirmed that PHiLIP enabled better coverage of test cases and boards with less effort while ensuring easy to reproduce test setups and execution. Results of PHiLIP are continuously used by developers to identify and fix bugs.

The structured testing interface helped to evolve a multitude of customized test applications with unspecified architecture and coding styles into a uniform shape to build test suites that follow a consistent test design. The developed wrapper tools provided machine-readable entry points that described available low-level APIs in a well-defined format. This information can be used for higher-level test generation and orchestration tools for further automation of end-to-end testing processes.

8.2 Limitations

MCU as a Reference. We chose to implement PHiLIP with an MCU (opposed to, *e.g.,* an FPGA) to simplify its design and make it accessible to embedded OS developers, though, an MCU has limitations. The MCU must reply immediately when data is requested via peripheral buses, which cannot be guaranteed when the bus clock speed is high relative to the MCU clock speed. Thus, advanced capabilities such as dynamic data response are provided only for lower transfer speeds, limiting high-speed transfers to predefined data. Using an FPGA or a very fast MCU are potential solutions, however, they interfere with our original goal of being low-cost and accessible to typical embedded developers.

Static Wiring. Most MCUs support several options to multiplex peripheral signals onto different pins or set pins to alternate functions. In our current setup, peripherals are only tested using a fixed pin configuration based on the default pins provided by RIOT OS. Tests are skipped if the wiring is not supported, for example, a peripheral is not exposed on the DUT Even though this can validate peripheral behavior, it does not cover all possible deployment configurations. While this can be addressed with a custom FPGA implementation or the addition of signal muxes, we argue that misconfigured pins are easier to find than software bugs.

Time-critical Test Commands. Due to the command response-based interaction method, only synchronous commands that contain the communication overhead can be issued. Wrapping time-critical interactions in firmware is a solution but loses the benefit of the agile process when many variations of wrapped commands are needed.

Capturing and Reporting Bugs. Outside of the I2C rework case, the bugs that were produced in RIOT are not included as the focus was not on studying the bugs that were caught but providing

reproducible tools for developers. As of now the capabilities of PHiLIP are still beyond the features used in the current test suites. We prioritized maturity and reliability for developers over very sophisticated features to avoid false positives and support community adoption.

Coverage Feedback. Code coverage metrics serve as valuable feedback to measure test completeness and for controlling test case generation. Static analysis can be used for preliminary offline coverage assessment but the runtime dynamics of peripheral interactions render such approaches incomplete because external events (like an altered signal) affect the paths executed in low-level peripheral driver code. Covering this requires on-target execution and end-to-end testing of firmware with the involved hardware peripherals, as proposed in this paper. In our setup, MCU-embedded trace macrocells or external debuggers can be employed for collecting runtime information on the test node to deduct code coverage. A generic implementation of this is considered future work as platform-specific debugging interfaces need dedicated tooling and considerable integration work without providing intrinsic benefits on its own. However, combining this with approaches for automated test generation is expected to be worthwhile the effort and mandates for separate examination.

8.3 Stability

Board Quality Issues. A low-cost board such as the bluepill may introduce quality issues, especially when purchasing from different vendors. The quality of the oscillator used for the clock source is also fairly low, affecting the minimum jitter and overall accuracy for timing tests. This still allows us to find misconfigured clock trees. Some calibration can be done to improve long-term drift tests. The infrastructure tests and qualification procedures are also able to reduce quality issues.

Reliable DUT Interface. Some vendor implementations of the USB interface used for communication between the test node and the DUT are not reliable, causing occasional test failures. To improve the reliability we add retries which increased the setup time but handled spurious failures.

Flashing Problems. Development tools for flashing DUTs are sometimes only available as closed-source software. This either restricts the test node to a specific CPU architecture and OS or requires manual integration effort. Even if flashers are readily available, many have reliability issues and cause occasional device lockups. To cope with locked-up DUTs, PHiLIP was equipped with automated flasher recovery mechanisms.

8.4 Outlook

In future work, we will use PHiLIP and the process described in this paper to improve test coverage on an increasing amount of platforms. To use PHiLIP with other OSs, its tooling can be supplemented with additional implementations of protocol abstraction layers that wrap DUT interactions. Information derived from regular HiL testing results and benchmarks of evolving software versions can be incorporated in various ways. It provides valuable feedback for well-informed technical decisions and strategic planning of development efforts and gives users a comprehensive and realistic overview of the quality and performance of supported features across available target devices. The system can further be used to qualify peripheral implementations of simulators and emulators by verifying them against the ground truth of a real device in the HiL setup. With the instrumented hardware and automatic test execution in place, another promising direction is to equip our system with methods for coverage feedback and automatic test generation.

Education and Research (BMBF) within the project *RAPstore* and the Free and Hanseatic City of Hamburg within *ahoi.digital*.

A NOTE ON REPRODUCIBILITY

We explicitly support reproducible research [1, 53]. The source code and documentation of our designs and implementations (including tools and scripts to set up the testing) are available on Github, see https://philip.riot-apps.net/.

REFERENCES

[1] ACM. Result and Artifact Review and Badging. http://acm.org/publications/policies/artifact-review-badging, Jan., 2017.

[2] ADJIH, C., BACCELLI, E., FLEURY, E., HARTER, G., MITTON, N., NOEL, T., PISSARD-GIBOLLET, R., SAINT-MARCEL, F., SCHREINER, G., VANDAELE, J., AND WATTEYNE, T. FIT IoT-LAB: A large scale open experimental IoT testbed. In *2015 IEEE 2nd World Forum on Internet of Things (WF-IoT)* (Dec 2015), pp. 459–464.

[3] AHMED, B. S., BURES, M., FRAJTAK, K., AND CERNY, T. Aspects of Quality in Internet of Things (IoT) Solutions: A Systematic Mapping Study. *IEEE Access 7* (Jan. 2019), 13758–13780.

[4] ANAND, S., BURKE, E. K., CHEN, T. Y., CLARK, J., COHEN, M. B., GRIESKAMP, W., HARMAN, M., HARROLD, M. J., MCMINN, P., BERTOLINO, A., JENNY LI, J., AND ZHU, H. An orchestrated survey of methodologies for automated software test case generation. *Journal of Systems and Software 86*, 8 (August 2013), 1978–2001.

[5] AO PEDRO DIAS, J., COUTO, F., PAIVA, A. C., AND FERREIRA, H. S. A Brief Overview of Existing Tools for Testing the Internet-of-Things. In *2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)* (April 2018), pp. 104–109.

[6] ARM LTD. Mbed OS. https://www.mbed.com, last accessed 07-17-2020, 2020.

[7] ARM MBED. Greentea testing applications. https://os.mbed.com/docs/mbed-os/v5.15/tools/greentea-testing-applications.html, last accessed 01-01-2021, 2021.

[8] ARM MBED. Icetea testing applications. https://os.mbed.com/docs/mbed-os/v5.13/tools/icetea-testing-applications.html, last accessed 01-01-2021, 2021.

[9] ARM SOFTWARE. CMSIS System View Description. https://arm-software.github.io/CMSIS_5/SVD/html/svd_Format_pg.html, last accessed 01-01-2021, 2020.

[10] BACCELLI, E., GÜNDOGAN, C., HAHM, O., KIETZMANN, P., LENDERS, M., PETERSEN, H., SCHLEISER, K., SCHMIDT, T. C., AND WÄHLISCH, M. RIOT: an Open Source Operating System for Low-end Embedded Devices in the IoT. *IEEE Internet of Things Journal 5*, 6 (December 2018), 4428–4440.

[11] BACCELLI, E., HAHM, O., GÜNES, M., WÄHLISCH, M., AND SCHMIDT, T. C. RIOT OS: Towards an OS for the Internet of Things. In *Proc. of the 32nd IEEE INFOCOM. Poster* (Piscataway, NJ, USA, 2013), IEEE Press, pp. 79–80.

[12] BURES, M., CERNY, T., AND AHMED, B. S. Internet of Things: Current Challenges in the Quality Assurance and Testing Methods. In *Proc. of International Conference on Information Science and Applications* (Singapore, 2019), K. J. Kim and N. Baek, Eds., vol. 514 of *LNEE*, Springer Singapore, pp. 625–634.

[13] CADAR, C., DUNBAR, D., AND ENGLER, D. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation* (USA, December 2008), OSDI'08, USENIX Association, p. 209–224.

[14] CADAR, C., GODEFROID, P., KHURSHID, S., PĂSĂREANU, C. S., SEN, K., TILLMANN, N., AND VISSER, W. Symbolic Execution for Software Testing in Practice: Preliminary Assessment. In *Proceedings of the 33rd International Conference on Software Engineering* (New York, NY, USA, May 2011), ICSE '11, ACM, p. 1066–1071.

[15] CADENCE. All Products A-Z. https://www.cadence.com/en_US/home/tools/tools-a-z.html, last accessed 01-01-2021, 2021.

[16] CHEN, W.-K., LIU, C.-H., LIANG, W. W.-Y., AND TSAI, M.-Y. ICAT: An IoT Device Compatibility Testing Tool. In *Proc. of 25th Asia-Pacific Software Engineering Conference (APSEC)* (NJ, USA, Dec. 2018), IEEE, pp. 668–672.

[17] CONG, K., XIE, F., AND LEI, L. Symbolic Execution of Virtual Devices. In *Proceedings of the 2013 13th International Conference on Quality Software* (USA, July 2013), QSIC '13, IEEE Computer Society, p. 1–10.

[18] CORTÉS, M., SARAIVA, R., SOUZA, M., MELLO, P., AND SOARES, P. Adoption of Software Testing in Internet of Things: A Systematic Literature Mapping. In *Proceedings of the IV Brazilian Symposium on Systematic and Automated Software Testing* (New York, NY, USA, September 2019), SAST 2019, Association for Computing Machinery, p. 3–11.

[19] DSPACE. dSPACE ECU Testing. https://www.dspace.com/de/gmb/home/products/systems/ecutest.cfm, last accessed 01-01-2021, 2021.

[20] DUNKELS, A., GRÖNVALL, B., AND VOIGT, T. Contiki - A Lightweight and Flexible Operating System for Tiny

Networked Sensors. In *Proc. of IEEE Local Computer Networks (LCN)* (Los Alamitos, CA, USA, 2004), IEEE Computer Society, pp. 455–462.

[21] ERIC BLAKE. Understanding QEMU devices. https://www.qemu.org/2018/02/09/understanding-qemu-devices/, last accessed 28-05-2021, 2018.

[22] FENG, B., MERA, A., AND LU, L. P2IM: Scalable and Hardware-independent Firmware Testing via Automatic Peripheral Interface Modeling. In *29th USENIX Security Symposium (USENIX Security 20)* (Aug. 2020), USENIX Association, pp. 1237–1254.

[23] GAROUSI, V., FELDERER, M., ÇAĞRI MURAT KARAPIÇAK, AND YILMAZ, U. Testing embedded software: A survey of the literature. *Information and Software Technology 104* (12 2018), 14–45.

[24] GEISSDOERFER, K., CHWALISZ, M., AND ZIMMERLING, M. Shepherd: A Portable Testbed for the Batteryless IoT. In *Proceedings of the 17th Conference on Embedded Networked Sensor Systems* (New York, NY, USA, 2019), SenSys '19, ACM, pp. 83–95.

[25] GEORGIEV, I., AND GEORGIEV, I. Simulation-Based Self-Testing in IoT-Enabled Manufacturing. In *Proc. of International Conference on Information Technologies (InfoTech)* (Piscataway, NJ, USA, Sept. 2019), IEEE.

[26] GOMEZ, A. K., AND BAJAJ, S. Challenges of Testing Complex Internet of Things (IoT) Devices and Systems. In *Proc. of 11th Intern. Conf. on Knowledge and Systems Engineering (KSE)* (Piscataway, NJ, USA, Oct. 2019), IEEE.

[27] GÜNDOGAN, C., KIETZMANN, P., LENDERS, M. S., PETERSEN, H., FREY, M., SCHMIDT, T. C., SHZU-JURASCHEK, F., AND WÄHLISCH, M. The Impact of Networking Protocols on Massive M2M Communication in the Industrial IoT. *IEEE Transactions on Network and Service Management (TNSM)* (2021). Accepted for publication.

[28] HANDZISKI, V., KÖPKE, A., WILLIG, A., AND WOLISZ, A. TWIST: a scalable and reconfigurable testbed for wireless indoor experiments with sensor networks. In *REALMAN '06: Proceedings of the 2nd international workshop on Multi-hop ad hoc networks: from theory to reality* (May 2006), ACM.

[29] KARLESKY, M., WILLIAMS, G., BEREZA, W., AND FLETCHER, M. Mocking the Embedded World: Test-Driven Development, Continuous Integration, and Design Patterns. In *Embedded Systems Conference* (Apr. 2007), ESC 413, pp. 1518–1532.

[30] KERÄNEN, J. S., AND RÄTY, T. Model-Based Testing of Embedded Systems in Hardware in the Loop Environment. *IET Software 6*, 4 (Aug. 2012), 364–376.

[31] KIETZMANN, P., BOECKMANN, L., LANZIERI, L., SCHMIDT, T. C., AND WÄHLISCH, M. A Performance Study of Crypto-Hardware in the Low-end IoT. In *International Conference on Embedded Wireless Systems and Networks (EWSN)* (New York, USA, February 2021), ACM.

[32] KIETZMANN, P., SCHMIDT, T. C., AND WÄHLISCH, M. A Guideline on Pseudorandom Number Generation (PRNG) in the IoT. *ACM Comput. Surv. 54*, 6 (2021). in print.

[33] KIM, H., AHMAD, A., HWANG, J., BAQA, H., GALL, F. L., ORTEGA, M. A. R., AND SONG, J. IoT-TaaS: Towards a Prospective IoT Testing Framework. *IEEE Access 6* (Apr. 2018), 15480–15493.

[34] LABCENTER ELECTRONICS NORTH AMERICA. Proteus Design Suite. https://www.labcenter.com, last accessed 01-01-2021, 2021.

[35] LIM, R., FERRARI, F., ZIMMERLING, M., WALSER, C., SOMMER, P., AND BEUTEL, J. FlockLab: A Testbed for Distributed, Synchronized Tracing and Profiling of Wireless Embedded Systems. In *Proceedings of the 12th International Conference on Information Processing in Sensor Networks* (New York, NY, USA, April 2013), IPSN '13, ACM, pp. 153–166.

[36] LIM, R., MAAG, B., DISSLER, B., BEUTEL, J., AND THIELE, L. A testbed for fine-grained tracing of time sensitive behavior in wireless sensor networks. In *2015 IEEE 40th Local Computer Networks Conference Workshops (LCN Workshops)* (October 2015), IEEE.

[37] LIN, W., ZENG, H., GAO, H., MIAO, H., AND WANG, X. Test Sequence Reduction of Wireless Protocol Conformance Testing to Internet of Things. *Security and Communication Networks 2018* (2018), 1–13.

[38] LINARO LIMITED. LAVA. https://docs.lavasoftware.org/lava/index.html, last accessed 01-01-2021, 2019.

[39] MALIK, B. H., KHALID, M., MARYAM, M., ALI, M. N., YOUSAF, S., MEHMOOD, M., SALEEM, H., AND STRAY, V. IoT Testing-as-a-Service: A New Dimension of Automation. *International Journal of Advanced Computer Science and Applications 10*, 5 (2019), 364–371.

[40] MÅRTENSSON, T. *Continuous Integration and Delivery Applied to Large-Scale Software-Intensive Embedded Systems*. PhD thesis, University of Groningen, 2019.

[41] MATHWORKS. Simulink. https://mathworks.com/products/simulink.html, last accessed 01-01-2021, 2021.

[42] METSA, J., KATARA, M., AND MIKKONEN, T. Testing Non-Functional Requirements with Aspects: An Industrial Case Study. In *Seventh International Conference on Quality Software* (Oct. 2007), QSIC'07, pp. 5–14.

[43] MURAD, G., BADARNEH, A., QUSCF, A., AND ALMASALHA, F. Software Testing Techniques in IoT. In *Proc. of 8th Intern. Conf. on Computer Science and Information Technology (CSIT)* (NJ, USA, July 2018), IEEE, pp. 17–21.

[44] MURESAN, M., AND PITICA, D. Software in the Loop Environment Reliability for Testing Embedded Code. In *IEEE*

18th Intern. Symposium for Design and Technology in Electronic Packaging (SIITME) (October 2012), pp. 325–328.

[45] PONTES, P. M., LIMA, B., AND FARIA, J. P. Izinto: A pattern-based iot testing framework. In *Companion Proceedings for the ISSTA/ECOOP 2018 Workshops* (New York, NY, USA, 2018), ACM, pp. 125–131.

[46] QEMU. QEMU - the FAST! processor emulator. https://www.qemu.org, last accessed 28-05-2021, 2021.

[47] REGEHR, J. Random Testing of Interrupt-Driven Software. In *Proc. 5th ACM International Conference on Embedded Software* (New York, NY, USA, September 2005), EMSOFT '05, ACM, pp. 290–298.

[48] RENODE. Renode Homepage. https://renode.io, last accessed 28-05-2021, 2021.

[49] RIOT. RIOT Emulators. https://api.riot-os.org/emulators.html, last accessed 28-05-2021, 2021.

[50] ROTTLEUTHNER, M., SCHMIDT, T. C., AND WÄHLISCH, M. Sense Your Power: The ECO Approach to Energy Awareness for IoT Devices. *ACM Transactions on Embedded Computing Systems (TECS) 20*, 3 (March 2021), 24:1–24:25.

[51] RUNESON, P. A Survey of Unit Testing Practices. *IEEE Software 23*, 4 (July 2006), 22–29.

[52] SASNAUSKAS, R., LANDSIEDEL, O., ALIZAI, M. H., WEISE, C., KOWALEWSKI, S., AND WEHRLE, K. KleeNet: Discovering Insidious Interaction Bugs in Wireless Sensor Networks before Deployment. In *Proceedings of the 9th ACM/IEEE International Conference on Information Processing in Sensor Networks* (New York, NY, USA, April 2010), IPSN '10, ACM, p. 186–196.

[53] SCHEITLE, Q., WÄHLISCH, M., GASSER, O., SCHMIDT, T. C., AND CARLE, G. Towards an Ecosystem for Reproducible Research in Computer Networking. In *Proc. of ACM SIGCOMM Reproducibility Workshop* (New York, NY, USA, August 2017), ACM, pp. 5–8.

[54] SENGUPTA, A., LEESATAPORNWONGSA, T., ARDEKANI, M. S., AND STUARDO, C. A. Transactuations: Where Transactions Meet the Physical World. In *2019 USENIX Annual Technical Conference* (Renton, WA, USA, July 2019), USENIX Association, pp. 91–106.

[55] SEO, J., KI, Y., CHOI, B., AND LA, K. Which Spot Should I Test for Effective Embedded Software Testing? In *Second International Conference on Secure System Integration and Reliability Improvement* (08 2008), pp. 135–142.

[56] SEO, J., SUNG, A., CHOI, B., AND KANG, S. Automating Embedded Software Testing on an Emulated Target Board. In *Proc. Second Intern. WS on Automation of Software Test* (USA, June 2007), AST '07, IEEE Computer Society, p. 9.

[57] STRANDBERG, P. E. Automated System Level Software Testing of Networked Embedded Systems. Licentiate Theses 275, Mälardalen University Press, Embedded Systems, 2018.

[58] SUNG, A., CHOI, B., AND SHIN, S. An interface test model for hardware-dependent software and embedded OS API of the embedded system. *Computer Standards and Interfaces 29*, 4 (May 2007), 430–443.

[59] TAN, L., KIM, J., SOKOLSKY, O., AND LEE, I. Model-based Testing and Monitoring for Hybrid Embedded Systems. In *Proceedings of the 2004 IEEE International Conference on Information Reuse and Integration, 2004. IRI 2004.* (May 2004), IEEE, pp. 487–492.

[60] TAN, T.-B., AND CHENG, W.-K. Software testing levels in internet of things (iot) architecture. In *Proc. of International Computer Symposium (ICS 2018). New Trends in Computer Technologies and Applications* (Singapore, 2019), C.-Y. Chang, C.-C. Lin, and H.-H. Lin, Eds., vol. 1013 of *CCIS*, Springer Nature Singapore, pp. 385–390.

[61] TAVERAS, P. A Systematic Exploration on Challenges and Limitations in Middleware Programming for IoT Technology. *International Journal of Hyperconnectivity and the Internet of Things 2*, 2 (Dec. 2018).

[62] TRÜB, R., FORNO, R. D., SIGRIST, L., MÜHLEBACH, L., BIRI, A., BEUTEL, J., AND THIELE, L. FlockLab 2: Multi-Modal Testing and Validation for Wireless IoT. In *3rd Workshop on Benchmarking Cyber-Physical Systems and Internet of Things (CPS-IoTBench 2020)* (September 2020), ETH Zurich, Computer Engineering and Networks Laboratory (TIK).

[63] VEJLUPEK, J., GREPL, R., KREJČÍ, P., LESÁK, F., AND MATOUŠ, K. Hardware-In-the-Loop Simulation for Automotive Parking Assistant Control Units. *Proc. 16th International Conference on Mechatronics, Mechatronika 2014* (December 2014), 325–330.

[64] VIRZONIS, D., JUKNA, T., AND RAMUNAS, D. Design of the Embedded Software Using Flexible Hardware-In-the-Loop Simulation Scheme. In *Proc. 12th IEEE Mediter. Electrotechnical Conf.* (June 2004), pp. 351–354, Vol.1.

[65] WOEHRLE, M., PLESSL, C., BEUTEL, J., AND THIELE, L. Increasing the Reliability of Wireless Sensor Networks with a Distributed Testing Framework. In *4th WS on Embedded Networked Sensors* (2007), EmNets'07, ACM, pp. 93–97.

[66] ZEPHYR PROJECT. Zephyr. https://www.zephyrproject.org, last accessed 07-17-2020, 2020.