ATTEST: Automated and Thorough Testing of Embedded Software in Teaching

Meinhard Kissich
meinhard.kissich@tugraz.at
Graz University of Technology
Graz, Austria

Klaus Weinbauer
klaus.weinbauer@student.tugraz.at
Graz University of Technology
Graz, Austria

Marcel Baunach
baunach@tugraz.at
Graz University of Technology
Graz, Austria

ABSTRACT

Dependability requirements are getting increasingly stringent in embedded systems, demanding highly skilled developers. One crucial point in building up expertise is getting precise feedback in programming courses at university to recognize flaws and learn from mistakes. Depending on the assignment and learning outcome, the assessment may include testing for the implementation's completeness, correctness, performance, and robustness. A timely and in-depth review for a large number of course participants relies on test automation. However, embedded software often includes hardware-dependent code that can only be executed on the target device. Thus, we provide an open-source and remote hardware-in-the-loop testing solution with pre-defined test cases for embedded software particularly designed for teaching in university courses. This paper defines and elaborates on the requirements, gives an insight into design decisions, and evaluates the test system on metrics of our *Real-Time Operating Systems* course.

CCS CONCEPTS

• **Social and professional topics** → **Student assessment**; • **Computer systems organization** → **Embedded software**; *Real-time operating systems.*

KEYWORDS

student assessment, embedded software, embedded systems, testing, real-time operating systems

ACM Reference Format:
Meinhard Kissich, Klaus Weinbauer, and Marcel Baunach. 2023. ATTEST: Automated and Thorough Testing of Embedded Software in Teaching. In *ECSEE 2023: European Conference on Software Engineering Education (ECSEE 2023), June 19–21, 2023, Seeon/Bavaria, Germany.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3593663.3593678

1 INTRODUCTION AND MOTIVATION

Embedded software challenges. Computing is getting increasingly ubiquitous in modern society, and embedded devices found their way into almost all areas of daily life. This trend is expected to continue [1, 12], demanding a considerable amount of well-engineered embedded software to cope with increasing dependability requirements. In such cases, embedded Operating Systems (OSs) can decompose complexity and increase reusability by introducing abstraction layers between the application and the underlying hardware. Developers can significantly benefit from having an OS, and most do not implement IoT devices bare metal [5]. However, the OS must be well-designed, robust to untrusted user code, and thoroughly tested to ensure safe operation.

Real-time operating systems course. We encourage students to build expertise in the abovementioned fields. In our course, students design and implement an embedded Real-Time Operating System (RTOS) – a spin-off of our research OS *SmartOS* [10]. They explore the impacts and trade-offs of design decisions and challenge themselves against others by defined benchmarks. We use an MSP430 microcontroller as a simple-to-understand target platform while providing sufficient features for real-world applications.

Building expertise. At the beginning of the semester, students receive a development kit they can use to work independently at home. While supported by a comprehensive script, the RTOS lecture, and face-to-face Q&A sessions, we want to drive self-engagement as programming and software design is a skill that can only be obtained by (a) actively working on involving tasks, (b) getting precise and constructive feedback on the design, implementation, and potential flaws, and (c) being motivated to pro-actively driving clean and maintainable code, and paying attention to robust design.

Student assessment and feedback. One prerequisite to providing meaningful and individual feedback to a large number of participants is automated submission testing. Besides being used for grading, students can submit their assignments and receive a report on completeness and code quality. Although many test system requirements are consistent with widely used test solutions, a need for teaching-specific features and adaptions emerged. Thus, we propose a testing solution for embedded software that precisely targets the needs of a university course from the very core. In a nutshell, we summarize the **contributions of this paper** as follows:

- We define and elaborate on the requirements when designing a testing solution for university courses that demand implementing code (Section 2).
- We give an insight into the design and implementation of the proposed open-source test system[1] for embedded software in teaching (Section 3).
- We evaluate design decisions for a real-world embedded software course (Section 4).

Section 5 discusses related work and Section 6 concludes with an outlook on: additions, experience evaluations and student feedback.

[1] https://iti.tugraz.at/attest

ECSEE 2023, June 19–21, 2023, Seeon/Bavaria, Germany

ACM ISBN 978-1-4503-9956-2/23/06.
https://doi.org/10.1145/3593663.3593678

2 REQUIREMENTS AND FEATURES

Table 1 summarizes the high-level requirements for the test system: first and foremost, the test system shall be easily deployable on Linux-based systems but also **(RQ1)** be able to run on most common systems without elaborated setup procedures. Furthermore, **(RQ2)** different test types to check the submitted code shall be supported, including (a) tests that compare sent messages from the target device against a reference (*message tests*), (b) tests that probe the target device's pins by a measurement device (*probe tests*), and (c) tests on the built software without executing the binary on the target device (*static tests*). For message tests, we focus on inspecting the occurrence and temporal order of check marks that the device sends to the host by `print` statements within the code. Probe tests monitor voltage patterns at the target device's pins and shall particularly be used to check the timing of pulses. Finally, static tests do not rely on the target device, as no code is executed. They can, e.g., be used to track the size of the built software.

When interacting with many participants in a course, **(RQ3)** a secure and reliable interface is needed that is well-known to a broad audience. As Git fulfills the abovementioned criteria and is the preferred version control system by many software developers, Git shall be supported to submit code to the test system. However, the test system shall not be limited to Git, as **(RQ4)** modularity and extensibility are primary design objectives. Each building block, such as the interface to data or communication channels, shall be easily adaptable. The test system is thus to be understood as a framework that can be adapted to different university courses and use cases with little effort.

Testing hardware-dependent software inherently relies on the target device when no simulation models are available. Thus, **(RQ5)** the test system throughput shall be increased by executing test cases in parallel on multiple target devices. In addition, **(RQ6)** each device is part of a heterogeneous *test unit* that can consist either solely of a target device or a target device attached to a measurement device. Depending on the capabilities, a test unit can execute all or only a subset of the test types. A test unit without a measurement device, e.g., cannot run a probe test. Although all test units may be equipped with a measurement device, a heterogeneous approach can substantially contribute to lowering the costs of the setup.

Different reports shall be generated **(RQ7)** with the level of detail adjusted to the recipient. Supervisors shall see all test results, including possible build errors and deviations from the reference. Students, however, shall only see the test case names and the final result. Thus, students get early feedback on their submissions by the number of passed/failed test cases and a slight indicator of the reason. Still, students need to understand their code thoroughly to locate the flaws. In addition, the report includes an anonymized ranking with fellow students as part of the report to drive constructive competition and innovative ideas. Finally, **(RQ8)** the test system implementation needs to be sufficiently tested to run stably over the semester without manual intervention and little maintenance.

In addition to the requirements above that we treat as a *must*, we define four features that have proved useful. The test system primarily targets microcontrollers that store their firmware in flash memory. Due to physics, flash memory suffers from a limited number of write cycles [11]. Thus, **(F1)** a monitoring system should be added for predictive maintenance to prevent failures during the course due to flash wear-out. Testing performance is increased by adding test units but is still limited by a practicable number of devices. Thus, **(F2)** supervisors should have means to influence the scheduling order, e.g., by assigning priorities relative to the number of already performed test runs by a student. Therefore, the test system should be runtime-reconfigurable **(F3)**. As a final feature, **(F4)** test units should be plug-and-play by an automated detection process that renders manual configuration at startup unnecessary.

Table 1: High-level requirements (RQ) for the test system and additionally implemented features (F).

Name	Description	Approach
(RQ1)	host system independent	Docker, Python
(RQ2)	diverse test types	{message, probe, size} tests
(RQ3)	secure and well-known interface	Git
(RQ4)	modular and extensible framework	modular, abstraction layers
(RQ5)	scalable by parallelization	worker pattern
(RQ6)	heterogeneous test units	state awareness, scheduling
(RQ7)	customized reports	report generation
(RQ8)	testing of the test system	testing suite
(F1)	predictive test units maintenance	flash cycle counter
(F2)	usage supervision	scheduling logic
(F3)	runtime configurability	config variables, config files
(F4)	easy test unit installation	automatic test unit detection

3 DESIGN & IMPLEMENTATION

Design overview. Two types of users primarily interact with the test system, as shown in Figure 1: students participating in the course and supervisors. After students have tested their implementation locally by self-crafted tests, they can submit the code to the test system. Data exchange currently happens via a Git server, according to **(RQ3)**. A commit to the `main` branch or merging a `dev` branch triggers a *test run*, i.e., one evaluation of the student's code.

The test system software is executed in a Linux-based Docker container on the test system's *host* computer. It stores the state, e.g., the already tested commit hashes, in an internal database, exchanges data with students, builds the embedded software, delegates the test units, and generates reports. In case newly submitted code is found, tests are started to check the completeness (*Is all functionality for the assignment implemented?*), robustness (*Does the submitted code break in certain situations?*), and other benchmark metrics.

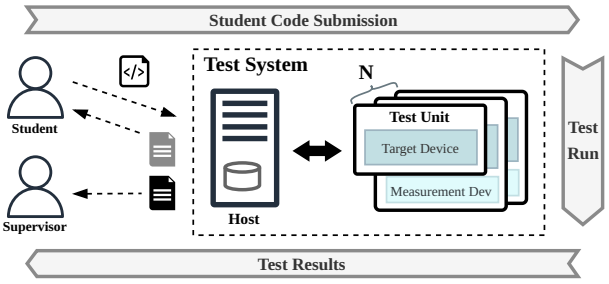

Figure 1: Overview of the test system setup and interactions with involved parties. Students submit their code to be tested with the available N test units in the test system. After completing the test run, reports are generated and published.

After executing all test cases, i.e., finishing a test run, reports are generated for the individual recipients according to **(RQ7)** and published. When using Git, students receive the report by a commit to a `testresults` branch in their working repositories.

A test run usually evaluates a set of individual test cases. From the host's perspective, each test case is one *task* intended to run on a test unit. Static tests are still tasks that may use a test unit, although it is not utilized to obtain the results. Processing a test case includes building, deploying, and executing the submitted code – and evaluating the result. However, a task can generally be any job that needs to be processed. It is registered to the system's *task queue* and waits for scheduling. The scheduler manages the queue and selects the tasks according to a prioritization metric.

Host system independence. As deployability and host system independence are essential objectives, the test system is implemented in Python and deployed via Docker to satisfy **(RQ1)** and contribute to **(RQ4)**. The system core is implemented as a standalone Python package. Its modular design enables the creation of a sophisticated test framework with end-to-end testability required by **(RQ8)**. Also, Docker is an elegant way to reduce the repetitive effort of setting up the environment and installing dependencies on the host system, e.g., Software Development Kits (SDKs) required by the test units can be included in the Dockerfile.

Test types. A rich set of scenarios shall be covered during a test run to detect flaws in the implementation by different test types according to **(RQ2)**. The test system builds the submitted code, including specifically designed test case code. In the RTOS course, test cases are user space tasks that invoke the student's OS kernel implementation. For a **message test**, the target device sends messages to the host. In the simplest case, executed `print` statements transmit an ASCII-encoded message via UART. The host receives the data and compares it against a reference text file, yielding a binary pass or fail result. However, more than message tests are required to check the OS implementation thoroughly. Embedded systems usually interact closely with the physical world and depend on accurate and correct timing. Thus, the system uses **probe tests** to check the timing of output signals. In our case, PicoScopes 2205A MSO [9] oscilloscopes sample the target device's I/O pins and send the data to the host. It computes the time between falling and rising signal edges generated by the test case code.

As the memory footprint is a critical metric in small embedded systems, **static tests** are invoked to inspect the size of the binary that is programmed into the flash memory. The probe tests to check timing and static tests to get the binary size do not directly produce a pass or fail status but rather a metric that can be further evaluated.

Interface. The test system does not expose a direct interface to students or supervisors. Instead, it builds on existing version control and file-sharing services. Thus, data can be exchanged via GitLab, as in the presented implementation, but also via SVN or plain folder sharing with minimal modifications in the corresponding module (cf. Figure 2, *Filesystem Abstraction*). The data exchange interface needs a reliable authentication and authorization framework, which is challenging and time-consuming to maintain. If not correctly implemented and frequently updated, it will lead to security issues. The utilization of existing services elegantly circumvents this problem and fulfills **(RQ3)**. Also, it provides a well-known interface, as software developers are usually familiar with Git.

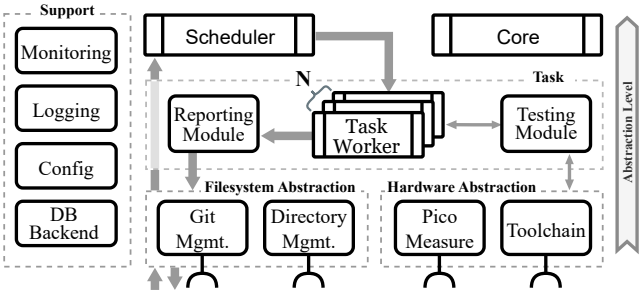

Figure 2: Block diagram of the host software's internal system components organized as a logical stack: from low abstraction at the bottom to high abstraction at the top. The *Scheduler*, *Core*, and *Task Worker* are independent threads. The grey arrows depict the logic flow of one test run through the test system.

System structure. Figure 2 depicts the block diagram of the host software. Interfaces to external dependencies are kept low-level and in a small scope for increased modularity and reusability. The *Core* component is the main thread that starts the parallelized test system application by spawning the *Scheduler* and *Task Worker* threads. Once the scheduler and worker threads are created, the main thread is primarily idle and only intervenes for some system-level events, like a shutdown request.

Among others, the scheduler manages the task queue, i.e., the waiting tasks such as test cases, and regularly checks for new code submissions. Each task is finally assigned to a worker, i.e., the execution thread of a test unit that utilizes functionality provided by *Testing* and *Reporting* modules. Although tasks are selected by their priority, the scheme is designed to have no interleaving test runs of different students. When using heterogeneous test units, the scheduler also needs to consider task requirements in addition to priorities, e.g., the availability of a measurement device in the test unit for probe tests. Also, the *Core* component manages the global system state, and the *Support* modules accompany the system stack. Its modules do not strictly belong to one of the abstraction layers and are used throughout the system – but with a clear and thin interface to reduce coupling.

In addition, the arrows in Figure 2 illustrate the logical flow of one test run. Submitted code enters the flow through the *Filesystem Abstraction*, e.g., through *Git Management*. The *Scheduler* takes the code and creates tasks for the individual test cases that are eventually processed by a set of *Task Worker* threads. Once a task is in progress, a test unit executes the test (if not a static test), and data is returned to the task. Finally, a report is generated by the *Reporting Module* and sent to the recipients.

Modularity and scalability. (RQ4) demands well-defined abstraction layers to limit the scope of dependencies to external tools and systems. Also, abstraction simplifies the software architecture by hiding data structures, process flow details, communication protocol, and exception handling for calls to third-party components.

The *Filesystem Abstraction* is responsible for data exchange between students, supervisors, and the test system. This includes loading student code, publishing test reports, and maintaining the file and directory structure for the build and deployment process.

The *Hardware Abstraction* block encapsulates the utilization of the specific target and measurement devices used in the test units. Thus, it prevents being locked into one device vendor, as the interface is clearly defined and can be easily adapted. The used PicoScopes oscilloscopes in the presented setup for probe tests are, e.g., operated through a vendor-specific SDK. The concept of abstraction allows the easy replacement of measurement devices in the future.

Parallelization. The majority of time for a task is spent waiting for the target device, primarily waiting for the software flashing process to finish. As hardware-dependent code relies on the target device, performance is increased by adding additional test units according to **(RQ5)**. We neglect test case and test pattern optimization, as these are use-case-specific and out of the scope of the test system. The parallelization potential is estimated to be high, as the host is not utilized during the waiting time. Additionally, heterogeneous test units, required by **(RQ6)**, largely contribute to increasing the performance. PicoScopes are by an order of magnitude more expensive than the used MSP430 target devices and test units only consisting of MSP430 devices allow scaling with minimized cost overhead. The scheduler is aware of the test units' capabilities and selects a sufficiently equipped one for each test case. Thus, the ratio of test units with a measurement device must be aligned to the execution time ratio of probe tests to all tests.

Reporting. The test system produces data on various topics that shall only be visible to the correct recipient. Students who submit the code and supervisors shall receive the test outcome. In contrast, information on the test system state is only meant for supervisors. In this sense, **(RQ7)** led to three types of reports. For GitLab as a data exchange platform, each student has a working repository also used to publish the student's test reports in a `testresult` branch. This report also includes an anonymized leaderboard of all students participating in the given semester. In a team assignment, a group repository is used instead of an individual repository, with all involved students being members. Supervisors get the other two types of reports into their repository: one for the overall system status and one for each submitted code with comprehensive information on the test run. The students' report and the test run report for supervisors are created when a test run is completed. System status reports are typically generated upon completed test runs but may also be generated on other occasions, e.g., system startup. After completing the last task of a test run, the executing task worker invokes the *Reporting Module* (cf. Figure 2) for report generation. It is responsible for the content and style of the reports. After the reporting module layouts the data in Markdown format, the report is handed over to the filesystem abstraction layer for publication.

Testing. Testing the test system implementation is crucial for reliable operation. Two test approaches are derived in addition to unit tests to satisfy **(RQ8)**. Both focus on high-level end-to-end tests to ensure system correctness. Testing calibration of the test units' measurement device is currently not implemented as device manufacturer guarantees on calibration proved sufficient for the described use case.

The first approach embeds Python code of the test system and allows the execution of tests from a student perspective. Its significant advantage is performing complex end-to-end tests (i.e., including the whole process chain from receiving the code to returning a report) on the test system via a simple Application Programming Interface (API). The second approach utilizes existing data. An independent script flow creates tuples of code and known test results from the predecessor test system that shall be replaced by the more modular and parallelizable solution proposed in this paper. Then, the code is submitted to the test system, and the report is compared against the known reference. While the first approach is used to check the correct functionality for a particular set of code and report tuples, the second method stress tests the system by running hundreds of test runs. This checks for rare issues in long-term operation and evaluates the performance under high utilization.

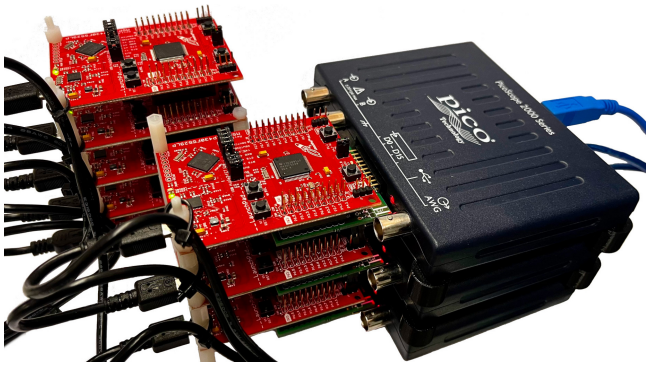

Figure 3: Test system setup for the RTOS course consisting of eight test units. Three of them are equipped with a measurement device (front stack).

4 EVALUATION

Design trade-offs are often not clear from the very beginning, especially for university courses with evolving content and a largely varying number of participants. Thus, this section evaluates some design decisions regarding scalability and the chosen Git interface.

Setup. The evaluation is performed on the setup for the RTOS course shown in Figure 3. It consists of eight test units, of which three are equipped with a measurement device. One test run currently consists of 57 test cases; only a small subset of eight test cases utilize the measurement device. Thus, the parallelization speedup is determined by the number of MSP430 boards instead of the PicoScopes in the subsequent evaluation. However, three test units include a measurement device for redundancy.

Parallelization capabilities. As most of the execution time of a test case is spent waiting for the target device, there is a high potential to increase the throughput by parallelization significantly. The waiting time hardly consumes any processing time from the host, who can, in the interim, prepare the following test case to be executed on another test unit. Amdahl's law [2] describes the theoretical speedup when increasing the computational resources. It can be used to quantify the performance gain when adding test units to parallelize test case execution. Table 2 lists the measured execution time for a test run. It shows the number of used test units and the speedup compared to a single test unit. By assuming that the parallelizable parts have a speedup according to the number of test units, Amdahl's law can be written as

$$p = \frac{N \cdot (s-1)}{s \cdot (N-1)}, \tag{1}$$

Table 2: Measured execution times for a test run with N test units, the resulting speedup s, and the parallelization portion according to Amdahl's law p.

# Test units N	Exec. time / s	Speedup s	Parall. portion p
1	788	-	-
2	411	1.92	0.958
4	222	3.55	0.958
8	131	6.02	0.953

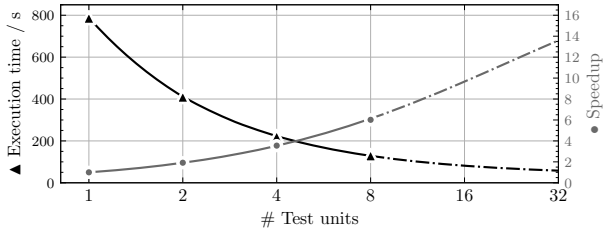

Figure 4: Measured reduction of execution time and speedup by adding test units.

where s is the speedup and N is the number of test units. All three computed parallelization portions show a solid consistency with a small decrease from 95.8% to 95.3% depending on the number of test units according to (1) and depicted in Table 2. This confirms the good parallelization capabilities. The measured execution times and achieved speedups are plotted in Figure 4. Although this is the theoretical speedup, it provides a fairly accurate prediction for what can be expected by further increasing resources.

Consistency to predecessor test solution. As mentioned in Section 3, the tool was stress-tested by hundreds of test runs with known testing results. The stress test was performed in four waves with over 450 test runs in total (>25k test cases) and achieved consistency above 99%. The deviation from complete accuracy is most likely due to physical conditions.

Git for reports. Besides being used to check for correctness and stable operations, the test also provides actual data on the required disk space for reports. Unknown error message lengths, frequency and type of errors in the submitted code, and how Git stores files left room for either highly optimistic or pessimistic estimations. Figure 5 depicts the required disk space for the supervisors' reports on an ext4 file system with a 4 KiB block size as reported by the Linux du command. It compares a Git repository as a report container to a plain folder structure. As can be seen, both approaches are suitable for a large number of test runs, but Git outperforms.

5 RELATED WORK

While testing embedded systems has been subject to active research for decades [3], most solutions are industry oriented. Nonetheless, educational test frameworks such as *Canary Framework* [4], *Xest* [8], or the workflow provided by Mattos [7] exist. Although they use a remote test server, not all required features are addressed, like testing on dedicated hardware with measurement devices or sufficient abstraction to students. *EmbedInsight* [6], a web service-based framework, has good parallelization capabilities but no direct version control integration, and a dedicated web interface opposes

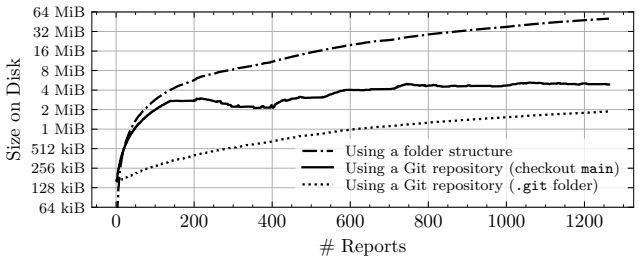

Figure 5: Size of the reports on the file system using a Git repository or a folder structure.

our needs. With the proposed test system, we aim to provide more flexibility, as elucidated in Section 2, with a focus on parallelization capabilities and heterogeneous reporting to meet our requirements.

6 CONCLUSION AND FUTURE WORK

Test automation for university courses differs from test solutions used in industry. A test system targeting the requirements elaborated in Section 2 adds significant value to embedded programming courses by providing meaningful feedback to students during assignments. Learning from mistakes based on the evaluation results is crucial to building expertise and coping with tight dependability demands. We give an insight into the design of the proposed test system and elaborate on the implementation details of the setup used in our RTOS course. An evaluation based on actual course data demonstrates good parallelization capabilities and advocates a Git repository storing reports. Additional features, such as the integration of plagiarism detection and gamification elements, as well as an evaluation that incorporates student feedback, will be part of future work after the first semester of use.

REFERENCES

[1] Thomas Alsop. 2022. Global embedded computing market revenue from 2018 to 2027. [Online] https://www.statista.com/statistics/1058799/worldwide-embedded-computing-market-revenue/ (March 2023).
[2] Gene M Amdahl. 1967. Validity of the single processor approach to achieving large scale computing capabilities. In *Proc. of Spring Joint Computer Conf.*
[3] Vahid Garousi, Michael Felderer, Çağrı Murat Karapıçak, and Uğur Yılmaz. 2018. Testing embedded software: A survey of the literature. *Information and Software Technology.*
[4] Sarah Heckman and Jason King. 2018. Developing Software Engineering Skills using Real Tools for Automated Grading. In *Computer Science Education Symp.*
[5] IEEE (Internet of Things), European Commission (Agile IoT), Eclipse IoT Working Group. 2016. Distribution of operating systems used for Internet-of-Things (IoT) devices, as of 2016. [Online] https://www.statista.com/statistics/659581/worldwide-internet-of-things-survey-operating-systems/ (March 2023).
[6] Hao Li, Bo-Jhang Ho, Bharathan Balaji, Yue Xin, Paul Martin, and Mani Srivastava. 2017. EmbedInsight: Automated Grading of Embedded Systems Assignments.
[7] André Mattos. 2023. Tests for embedded systems. https://github.com/andrempmattos/tests-embedded-systems (March 2023).
[8] Matthew H Netkow and Dennis Brylow. 2010. Xest: An Automated Framework for Regression Testing of Embedded Software.
[9] Pico Technology 2016. *PicoScope 2200A Series.* Pico Technology.
[10] Tobias Scheipel, Leandro Batista Ribeiro, Tim Sagaster, and Marcel Baunach. 2022. SmartOS: An OS Architecture for Sustainable Embedded Systems. In *Tagungsband des FG-BS Frühjahrstreffens 2022.*
[11] Texas Instruments Incorporated. 2006. *MSP430 Flash Memory Characteristics.* Application Report SLAA334B. Revised August 2018.
[12] Lionel Sujay Vailshery. 2022. Number of Internet of Things (IoT) connected devices worldwide from 2019 to 2030, by use case. [Online] https://www.statista.com/statistics/1194701/iot-connected-devices-use-case/ (March 2023).