INTL JOURNAL OF ELECTRONICS AND TELECOMMUNICATIONS, 2026, VOL. 72, NO. 3, PP. 1–10
Manuscript received February 28, 2026; revised July 2026. doi: 10.24425/ijet.2026.157953

NXDART: Distributed automated build and runtime test environment for Apache NuttX RTOS

Tomasz Cedro, Gregory Nutt, Lup Yuen Lee, Grzegorz Południewski, Szymon Wilk, and Wojciech Glinkowski

***Abstract*—This paper presents the concept and initial implementation of a distributed automated build and runtime test environment for NuttX, called NXDART. NuttX is Apache 2.0 licensed Free and Open-Source (FOSS) Real-Time Operating System (RTOS), created by Gregory Nutt and released to the public in 2007, with strong focus on portability, small footprint, scalability, and compatibility with POSIX, ANSI, and other standards commonly used by Unix and other RTOSes (e.g. VxWorks).**

NuttX community consists of volunteering hobbyists, academia, and small/medium/large enterprise professionals from around the world. NuttX supports over 15 different 8..64-bit CPU architectures (RISC-V, ARM, ARM64, AVR, CEVA, HC, MIPS, Misoc, OpenRISC, Renesas, SPARC, TriCore, x86, AMD64, Xtensa, Z16, Z80, FPGA) on over 340 unique hardware boards with over 1600 example target firmware configurations. Project development intensity is measured in thousands of files changed every month with hundreds thousands of code lines modifications.

In order to assure self-compatibility and long term maintenance for supported platform all new source code contributions are always processed manually by independent reviewers and automated Continuous Integration (CI) software tools. Unfortunately, not all breaking changes can be detected that way. Code may build for various architectures and even run correctly on emulators but it may still not work as expected on real world hardware. Therefore, comprehensive runtime tests are required.

Hardware testing of hundreds of different devices in one place turned out impossible for our free and community driven project. To address this problem we have invented distributed hardware-in-the-loop (HIL) test environment that is easy to set up and maintain as it reuses hardware components already owned by hundreds of users around the world. This approach assures maximum coverage of possible boards and build environments, independence and uninterrupted redundant availability of build and runtime test automation, with feedback logs reporting directly to the project upstream, at virtually zero cost.

Tomasz CEDRO is with Apache NuttX RTOS, Polish Telemedicine and e-Health Society, IQCREDO, and CeDeROM, Warsaw, Poland (e-mail: tomek@cedro.info, ORCID: https://orcid.org/0000-0003-0069-1908).

Gregory NUTT is with Apache NuttX RTOS, Costa Rica (e-mail: spudaneco@gmail.com).

Lup Yuen LEE is with Apache NuttX RTOS, Singapore (e-mail: luppy@appkaki.com).

Grzegorz POLUDNIEWSKI is with Polish Telemedicine and e-Health Society, and IQCREDO, Warsaw, Poland (e-mail: grzegorz@poludniewski.pl, ORCID: https://orcid.org/0009-0006-2479-9592).

Szymon WILK is with Polish Telemedicine and e-Health Society, and Poznan University of Technology, Poznań, Poland (e-mail: szymon.wilk@cs.put.poznan.pl, ORCID: https://orcid.org/0000-0002-7807-454X).

Wojciech GLINKOWSKI is with Polish Telemedicine and e-Health Society, and Warsaw Medical University, Warsaw, Poland (e-mail: wojciech.glinkowski@wum.edu.pl, ORCID: https://orcid.org/0000-0003-2602-1128).

***Keywords*—embedded; NuttX; RTOS; OS; Open-Source; FOSS; Apache; testing; automation; distributed; build; runtime; hardware; CPU; MCU; SoC; microprocessor; microcontroller; HIL, hardware-in-the-loop**

I. Introduction

A. Background

NUTTX [1] is a Free and Open-Source (FOSS) Real Time Operating System (RTOS) with full access to source code that can be adapted and extended by anyone for free. Invented by Gregory Nutt around 1997 [2], [3], with a first working prototype around 2005, it was first released to the public in February 2007 under a 3-clause BSD license. After gaining traction and widespread popularity NuttX went under the Apache Software Foundation umbrella in 2019, switching the license to Apache 2.0. Source code was initially available on BitBucket over web frontend and CVS (Concurrent Versioning System). Around 2010 CVS was replaced with SVN (Apache Subversion). In 2013 SVN was replaced by Git thus git history started that year. Finally in 2019 main project repository was moved from BitBucket to GitHub (and Apache git mirror servers). Changes in version control systems caused partial loss of initial project history ($\sim$ 35000 commits).

Today NuttX can be cross-compiled on various operating systems (OSs) such as FreeBSD, OpenBSD, NetBSD, Linux, macOS, Windows, and most likely others, however, it cannot build itself yet. NuttX runs on over 15 different 8..64-bit microprocessor architectures on over 340 different hardware boards [4], mainly memory constrained ultra-low-power embedded microcontrollers, powering all sorts of consumer and industrial products, as well as academic and hobby projects all around the world. Project development is based on voluntary work of the community.

NuttX Project Management Committee follows responsibilities assigned by the Apache Software Foundation, and "The Inviolable Principles of NuttX" [5] set by Gregory Nutt. These define ultimate goals in terms of branding, open licensing model, standards compliance, modular architecture, documentation, coding style, democratized community model, but also point out underlying "potential enemies" of the project such as reducing the effort at the expense of quality or portability.

Source code is hosted on GitHub and split into three functional repositories: RTOS, Applications, and Website. Documentation [6] is part of the RTOS repository and its online

version is built automatically and published as part of the Website project. Four releases are planned each year and there are two separate release packages (RTOS and Apps).

Project activity metrics can be monitored in real time by GitHub Insights. Figure 1 presents contribution activity diagrams of the project over time and reveals a monthly peak of over 1500 commits for the RTOS (1673 in 2015Q1, 1460 in 2020Q4, 1291 in 2024Q4) and a monthly peak of over 300 commits for Apps (316 in 2014Q3, 286 in 2023Q1, 249 in 2024Q4). There are 51378 clones of the RTOS repository and 48433 of the Apps one. Moreover, there are overall 662 contributors to RTOS and 302 to Apps.

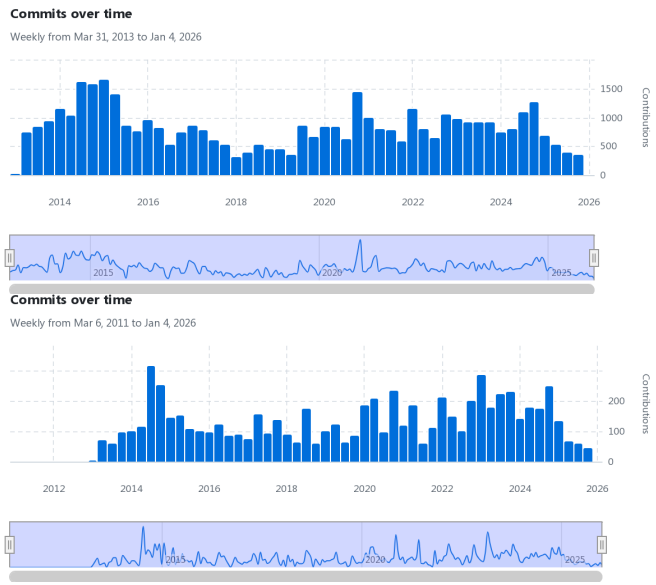


Fig. 1: NuttX RTOS (top) and Apps (bottom) commits over time.

To better understand how many changes are introduced monthly, we provide a few examples from the past. In the RTOS repository 44 contributors provided 264 commits that changed 2791 files with 666473 additions and 15814 deletions (peak numbers are too large to compute and are not provided by GitHub Insights), while in the Apps repository 8 contributors provided 20 commits that changed 44 files with 1720 additions and 77 deletions (peak numbers are 237169/206091 in 2018Q3). All these changes are handled daily by a small team of only few volunteering maintainers located all around the world.

B. Challenges

The ultimate goal of assuring self-compatibility that enables proper long-term maintenance and interoperability, depends on the contribution quality and their validation before code is merged becoming an integral part of the project source code. Increasing pace of changes in the surrounding world comes not only from exponential advancements in microelectronics technologies aligned with the Moore's Law, but also various geo-political, economical, and market factors [7]. Although NuttX by design constitutes a self-contained and independent ecosystem by minimizing the amount of external dependencies, it depends on external tools, libraries, compilers, build host setups, etc., that tend to change at increasing frequency.

Introducing new features sometimes requires a change in the existing code that may break compatibility. Improvements and fixes in one area sometimes result in breaking other areas. So called "breaking changes" and "software bugs" are usually introduced unintentionally, and unfortunately may be omitted at the code review stage, but may also lead to fixes resulting in new bugs that result in an avalanche of problems. Therefore, it is extremely important to properly test each change before it becomes an integral part of the project.

NuttX source code repository is equipped with Continuous Integration (CI) software automation mechanisms built-in to the source code repository, called GitHub Actions. CI validates software build process for the system, applications, website, and documentation components of the project on Linux, macOS, and Windows operating systems as the build hosts, including cross-compilation to only limited subset of most popular target CPU architectures (ARM, ARM64, RISC-V, SIM, x86, x86_64, Xtensa). This allows not only for daily source code build verification after changes have been introduced, but more importantly, before changes are introduced. This process is performed automatically for every single incoming Pull Request (PRs). Additional checks are performed too, like code formatting and structure verification, merge-ability, etc. CI scripts are part of the source code repository and can be also launched locally in virtual machines running various operating systems. Some companies working with NuttX are known to have their own in-house test environments, unfortunately, these are not publicly available, and their details are not revealed.

However, useful [8]–[12] CI/virtualization/emulation systems are not free of financial costs and various limitations. Not all necessary build hosts and tools are provided by GitHub Actions (e.g. BSD Unix). Virtualization of some operating systems is not possible without actual expensive additional hardware (e.g. macOS licensing restrictions). Test builds for hundreds of incoming PRs on so many build hosts for so many targets/architectures/boards results in hundreds of hours of build time consuming expensive and limited computing resources which may lead to quickly draining available commercial service limits and risk of losing the CI completely. We encountered this situation several times in the past [13].

There are many other limitations of the CI/virtualization/emulation technologies in general, therefore test coverage is always limited, and that impacts quality verification possibilities. One of the most obvious but crucial limitation here is that software components can only test software. Apache NuttX RTOS is meant to run on real world embedded hardware devices and that requires a new dedicated set of tools.

C. Project Objectives

This paper presents the novel concept and initial implementation of a community based distributed automated build and runtime test environment for NuttX RTOS. Distributed nature of the project allows anyone to join with useful already owned hardware resources which results in zero additional financial

cost. Automation allows silent background work with limited interaction with the user. FOSS approach enables anyone to contribute and adapt to even most specific requirements and features unavailable on closed-source commercial platforms. Users of specific chip architectures can attach their boards to local test nodes that will perform tests and report back the test results. Enterprises developing commercial products based on NuttX can perform in-house testing too, without sharing the details when prohibited, and report issues with limited scope when problems are encountered. As more users join testing, expanded coverage is provided even for very unique configurations. Also cost and availability burden is distributed among volunteers from around the world, and the infrastructure has no single dependency or point of failure.

II. Methods

A. Licensing

All project components used and created within the project are based on FOSS Software, preferably with Apache 2.0 compatible licenses to align with current Apache NuttX RTOS licensing model. Where possible only Open-Source Hardware is used. Components with "viral taint" licenses such as GPL are avoided (but possible) because presented solutions may be used in commercial environments/products and that could exclude a large part of the community. NuttX itself contains thousands ready for use software components with various licensing models but with clear separation of license groups that needs to be intentionally enabled by hand. There is no preference nor enforcement of any specific operating system, except it should be a POSIX compatible runtime environment. In the future support for additional operating system specific features may be developed and provided as an optional choice.

B. Architecture

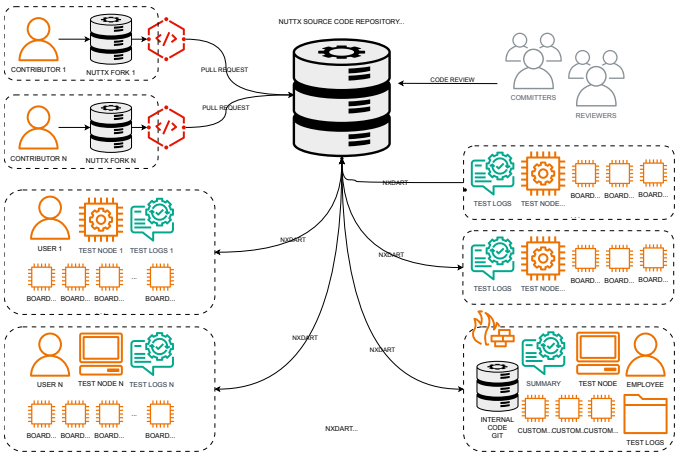


Fig. 2: NXDART Architecture.

Figure 2 presents the Distributed Automated Build and Runtime Test Environment for Apache NuttX RTOS (NXDART) architecture overview diagram. Standard workflow consists of contributors working on a project source code fork, changes are then wrapped into git commits with description on why/how/what has changed (i.e. fix or new functionality), then bundle of changes is submitted to the upstream project source code repository as PR. NuttX maintainers then perform collectively and in iterations a manual code review (backed by the CI automation builds) that results in blocking, change request, or approval.

Because code review is done by a small team of volunteers, available hardware to perform build and runtime tests is limited, therefore review scope is the best effort approach. In order to overcome this limitation the workload can be distributed among users worldwide to help in firmware build and target runtime verification in order to identify and eliminate negative impact on other project before changes are accepted.

The ultimate goal is to prevent breaking changes from landing into the project source code unnoticed in order to improve project quality, self-compatibility, and long term maintenance.

C. Automation Pipeline

As a starting point FreeBSD, Linux, and macOS were selected as the build host OS, that cross-compiles NuttX to most familiar and inexpensive boards (STM32 Nucleo-L432KC, Oz64 SG2000, ESP32C6 DevKit, ESP32S-DevKit) representing most popular CPU architectures (ARM Cortex-M, RISC-V, Xtensa). Figure 3 presents automation steps resembling usual manual process.

D. Test Design and Portability

Automation, at this point, is implemented with simple shell scripts that mimic manual command execution that assure portability and zero additional dependencies. In future popular open test frameworks such as PyTest, Twister, Renode, or others, may be considered. However, initial analysis revealed possible long term maintenance and self-incompatibility risks such as programming languages syntax changes, dependencies management issues, insufficient handling of exceptions/errors/-timeouts, lack of updates and fixes, no support for all required platforms, lack of required customization, etc. These problems are often called "maintenance nightmare" that means software becoming a liability rather than asset. Commercial automated testing solutions are not even considered, as they may seemingly offer short term win, but at the cost of freedom of choice, independence, self-incompatibility, and enforced product life cycles.

Automation scripts are used to control execution of Toolchains and/or Software Development Kits (SDK) that are, in short, required to cross-compile source code to a format required by target CPU, for use with other binary utilities (such as debuggers), link all binaries into a single image aligned with internal target microcontroller (MCU) or system on a chip (SoC) memory map, etc. Some targets require additional dedicated tools, for instance, firmware image format converters specific for DFU/BootROM loaders. Sometimes a specific set of versions may be required to accomplish successful build or for backward compiler compatibility verification. Most of the tools are available as OS packages, but some test cases may require compilation of additional tools what usually takes far more time that the firmware build itself. Commercial and

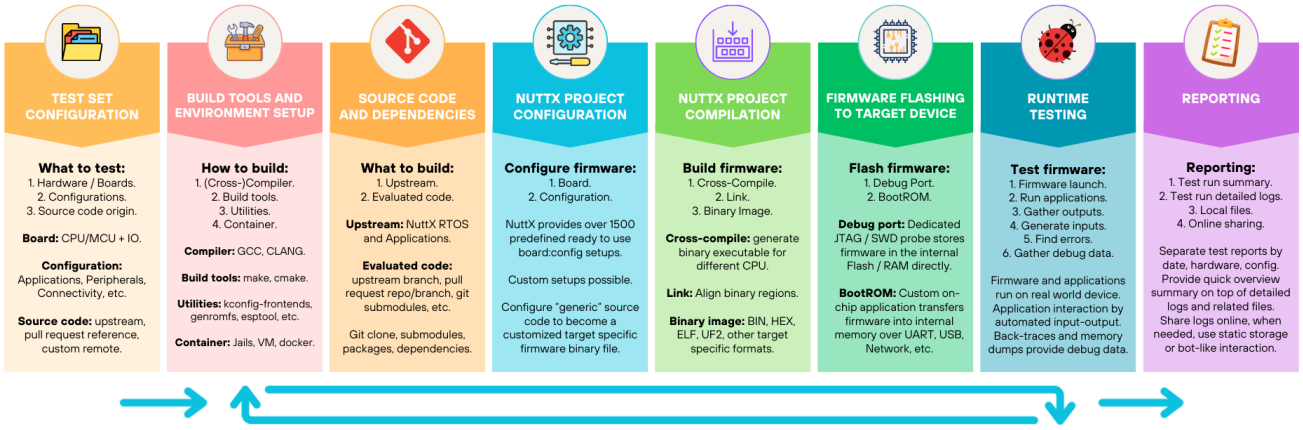


Fig. 3: Test automation pipeline.

dedicated closed-source tools can be used too when required by the custom project/product.

Software portability, compatibility, and dependencies need extra attention, as the usually most problematic part. Although shell scripting is the oldest and simplest possible self-contained programming method with zero dependencies (scripts just execute other programs and analyze execution results), there are known compatibility issues caused by differences in features implementation by different shells. For instance `/bin/sh` on Unix/POSIX is the historical "Command Interpreter (Shell)", but on Linux and macOS `/bin/sh` was replaced "silently" with `/bin/bash` or `/usr/bin/bash` (GNU Bourne-Again Shell) still holding the original name `/bin/sh` what is misleading and causes compatibility/portability problems. Recent Linux Debian and Ubuntu releases may silently replace `/bin/sh` with `/usr/bin/dash` that is neither fully compatible with `/bin/sh` nor `/bin/bash`. Scripts must work on various systems where "silent assumptions" may come from "bleeding-edge innovative design" or misconfiguration. Different OSs offer alternative sets of tools for the same task using exactly the same program name. For instance, standard BSD and GNU tools are both named `make`, `sed`, `awk`, that are neither fully syntax nor feature-set compatible.

E. Test Execution

As illustrated in Figure 2, *TestNode* is a local computer system (i.e. desktop, laptop, or embedded system) where *Boards* under evaluation are attached in order to run the tests. Every test consists of test steps that may include other test steps, parametrized with global (available for all tests) and local (test specific) configuration. To avoid long and complex scripting, code of each test step is as simple and generic as possible, self-sufficient, modular, independent of each other, and stored in a separate file. This allows building chain of parametrized events made with reusable blocks. Shell interpreters can assess command execution status, and can be configured to continue or fail in case of error. That way expected failure may be recorded without interrupting the longer test sequence, but also unexpected failure can be handled and propagated back to the caller. Tests and test steps may contain loops in which parameters are provided from the lists, for instance, a list of configurations to build with a list of runtime tests to perform per configuration, that can use different compiler versions, different sets of dependencies versions. All this allows cross-compatibility, self-compatibility, and backward/forward-compatibility validation.

F. Test Results

Each test step defines its boolean Assert/Expect conditions that result in *PASS* condition (result code) for *True* evaluation and *FAIL* condition for *False* evaluation. This way tests that are designed to fail (e.g. invalid parameter handling validation) will return *PASS* evaluation on expected failure.

Aside from the *PASS* and *FAIL* codes, adding more test result codes in future may be useful and provide compatibility with existing test suites and associated tools. For instance PyTest framework uses following "expected outcomes" *f/-failed, E/error, s/skipped, x/xfailed, X/xpassed, p/passed, P/-passed_with_output*, while test groups may be (de)selected with codes: *a/all_except_pP, A/all, N/none*. Twister framework uses the following result codes: *FILTER, NOTRUN, BLOCK, SKIP, ERROR, FAIL, PASS*, and uses *TestInstance, TestSuite, TestCase, Harness* names for test organization. According to our needs and experiences the following additional result codes may be useful and added in future: *ERROR*, *TIMEOUT*, *UNAVAILABLE*, and *SKIP*.

III. Results

A. Setup

NuttX RTOS is free, Open-Source, and platform-agnostic. That means it can be built (cross-compiled) on any supported OS and run (executed) on any supported CPU architecture (target platform). Running on over 15 different CPU architectures requires dedicated cross-compilation toolchains that allow building software dedicated for selected instruction sets on different instruction sets machines. These toolchains, whether free or commercial, must be available to the test environment.

NXDART's goal is to provide a platform agnostic solution that enables anyone to run on the hardware already owned with no additional costs or licenses. Table I presents initial identification of supported hardware (architectures, MCUs, boards) versus currently owned ones (addressed by owner initials). For clarification, a single NuttX architecture family may group shared code parts of various similar CPU models unless separation is necessary. For instance 32-bit *nuttx/arch/arm* directory contains ARMv6-M, ARMv7-M, ARMv7E-M architectures also known as ARM Cortex M0, M0+, M3, M4, M7, but code for 64-bit ARM cores is located in *nuttx/arch/arm64*, while *nuttx/arch/risc-v* for now contains both 32-bit and 64-bit CPU architecture variants.

TestNode performs a cross-compiled build, flashes firmware images, controls runtime tests execution on the attached target boards, gathers logs, and reports test results to the selected location. TestNode can be any computer system that is able to run required NuttX build tools and is equipped with a network interface and a USB host port. Free and Open-Source OSs such as FreeBSD and Linux can run on virtually any computer or embedded system, especially on obsolete hardware where commercial operating systems such as macOS or Windows are no longer supported. This brings new life to old but still functional devices.

B. Power-Time-Cost

If power consumption and overall low cost are more important than compilation time, then there are several possible scenarios that we have analyzed. The "absolute minimum" TestNode, priced below 25EUR (including 64GB uSD card and power supply) based on Raspberry Pi Zero 2W (4-core ARM-Cortex-A53 CPU 1GHz with 512MB RAM) requires 412 seconds to build *raspberrypi-pico:nsh* configuration at 5W peak and 1.5W idle power consumption (5VDC). MacStudio 2023 (12-core AppleSilicon ARM64 M2Max CPU 3.5GHz with 32GB RAM) computer priced around 3000EUR scored 22s/53W/9W (230VAC), resulting in almost 20x speedup at 10x higher power consumption, but also 120x higher initial price. Ten years old desktop PC (8-core AMD FX8320 3.5GHz CPU with 32GB RAM) scored 45s/280W/175W (230VAC). An old, pre-owned notebook (4-core Intel Pentium N5030 with 8GB RAM) scored 176s/8.4W/6.5W (230VAC/19.5VDC). RaspberryPi 4B scored 153s/5W/2W, RaspberryPi 5 (using uSD card not M2 SSD drive) scored 44s/7W/2.3W, and Intel Core Ultra 9 285K scored build results of 12s/355W/100W. Detailed measurements are presented in Table II.

Modern computer systems clearly provide best computational performance at reduced power consumption – latest SBC build time is comparable to 10 years old PC at 40..75x energy improvement. Therefore one powerful TestNode can build firmware images that are then executed by smaller energy efficient TestNodes. Build time highly depends on the number of available CPU cores, CPU family / generation / clock, RAM memory frequency, and data storage read / write speeds where files are stored (i.e. SATA or NVME, RAID, RamDisk, etc). RamDisk (virtual disk created in RAM) not only provides fastest possible storage and thus best build times but also does not wear-out SSD drives (*Total Bytes Written* vs. *Mean Time Between Failures*). Maintenance cost can be potentially reduced to zero with natural reusable energy sources (i.e. custom open-source hardware wind turbines or small solar panels) maybe even energy harvesting techniques but only for low peak power consumption devices.

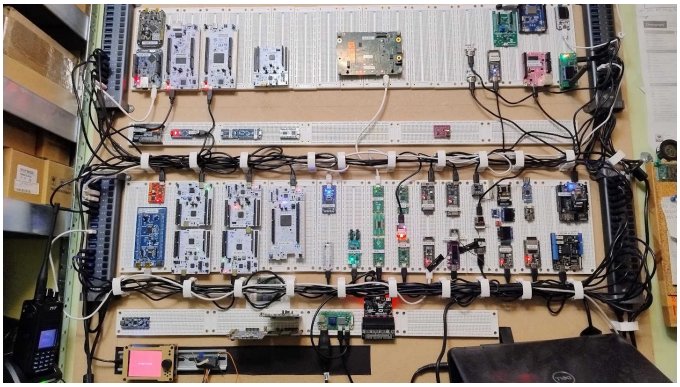


Fig. 4: Example NXDART installation.

C. Real World Hardware Validation

Runtime testing is performed on a physical hardware device called TargetBoard (Figure 4). Most boards in use have built-in *Debug Probe*, that allows firmware flashing and debugging using JTAG or SWD port [14], and access to UART serial console for NuttX Shell, all over the single USB port connection including power supply. This Debug Probe is usually implemented with a separate MCU component, it may contain Open-Source firmware such as DAPLink, BlackMagic, or proprietary J-Link or ST-LINK. Some of the modern microcontrollers such as Espressif's ESP32-C or S family have built-in on-chip USB-Serial-JTAG block that serves both flashing and debugging with no additional external components which is really smart and useful solution.

Many boards can be operated over a single USB connection too, but in a different way, for instance using *UART BootROM* flashing functionality, or *USB DFU* that initially only allows firmware update and then NuttX firmware exposes reverse shell over the same USB port. Some boards, however, may need external Debug Probe and/or USB-to-UART converter devices that may consume one or two additional USB ports. OpenOCD and pyOCD are most popular Open-Source utilities used for firmware flash and debug supporting various debug probes and target CPU architectures. Most NuttX boards handle a `make flash` command that automates flashing. Firmware can be equipped with assertions support and verbose backtraces to track application exceptions and/or kernel panic cases. Although most tests do not really require dedicated full debug setup even on critical system failures, this is possible and usually available by default. More details are included in the Discussion section.

D. Firmware Configuration

Every runtime test performed on a target board needs a predefined firmware build configuration in order to produce

TABLE I: Possible hardware Architectures, MCUs, boards, versus currently owned boards.

Possible Hardware			Available Hardware (user-owned boards)								
Arch	MCU	Board	AC	TC	MS	LY	TM	SL	MG	TH	SL
13	97	310	55	45	39	8	6	5	5	1	1

TABLE II: Comparison of build hosts hardware parameters vs build time vs power consumption.

	rPI Zero 2W	rPI 4B	rPI 5	Laptop PC	Desktop PC	Lenovo Think Centre M900	HP DL380 G7	Mac Studio 2023	Mac Mini 2023	Lenovo Think Station P510	Desktop PC
CPU Name	ARM Cortex-A53	ARM Cortex-A72	ARM Cortex-A76	Intel Pentium N5030	AMD FX-8320	Intel i5 6400T	Intel Xeon X5660	Apple M2Max	Apple M2Pro	Intel Xeon E5-2650 v4	Intel Core Ultra 9 285K
CPU Arch	ARM64	ARM64	ARM64	x86_64	x86_64	x86_64	x86_64	ARM64	ARM64	x86_64	x86_64
CPU cores	4	4	4	4	8	4	24	12	12	24	24
CPU frequency	1GHz	1.8GHz	2.4GHz	1.1GHz	2.8GHz	2.8GHz	2.8GHz	3.5GHz	3.5GHz	2.9GHz	3.2/5.7GHz
RAM	0.5GB	8GB	8GB	8GB	32GB	8GB	128GB	32GB	32GB	64GB	96GB
Disk type	uSD card	uSD card	uSD card	NVM SSD	SATA HDD	SATA HDD	SATA HDD	M2M SSD	M2M SSD	SATA HDD	NVM SSD
OS	FreeBSD 14.2	Linux rPiOS	Linux rPiOS	FreeBSD 14.2	FreeBSD 14.2	Linux Ubuntu 24.04.2	FreeBSD 14.2	macOS 15.3.1	macOS 15.3.1	Linux Ubuntu 24.04.2	FreeBSD 14.2
Compiler	GCC 10.3.1	GCC 14.2	GCC 14.2	GCC 10.3.1	GCC 10.3.1	GCC 14.2.1	GCC 10.3.1	GCC 14.2	GCC 13.2.1	GCC 14.2.1	GCC 14.2.1
Price [EUR]	25	85	90	150	250	356	500	3000	2828	377	2000
Power [W]: Idle	1.5	2	2.3	6.5	175	6.5	180	9	8	200	100
Power [W]: Build	5	5	7	8.4	280	8.4	346	53	50	380	355
Build [s]: Single Core	1187	294	97	571	198	102	358	90	41	102	101
Build [s]: Multi Core	412	153	44	176	45	36	23	22	15	13	12
HardDrive R/W[MB/s]	23 / 7	35/21	140/32	912 / 550	2040 / 1171	942 / 308	1579 / 665	7744 / 3817	4873 / 1560	1331 / 138	5088/2902
RamDisk R/W [MB/s]	6 / 9	633/505	3400/1600	97 / 319	326 / 975	340 / 128	2434 / 880	7661 / 3555	5850 / 1767	4608 / 90	9034/3913

valuable and comparable test results. NuttX is a Unix-like RTOS dedicated for small microcontrollers. Just as BSD or Linux it contains various drivers and applications, in NuttX however these are build-time selectable and then compiled into a single small (kilobytes in size) firmware image.

NuttX runs on microcontrollers as small as 8KB of Flash and 8KB RAM that can fit only a minimal set of features, also various boards that contain the same MCU but surrounded with completely different peripherals. There are over 1600 various predefined firmware configurations named according to the *board:configuration* nomenclature that provide ready-for-use setup and a reference point for further development. Each board has an absolutely minimalistic configuration called *board:nsh*, containing only NuttX Shell Interpreter, and the *board:ostest* configuration that enables built-in core system and kernel level runtime verification application but requires higher memory footprint. Both configurations are fully controllable over UART and can be easily logged to a local TestNode results file.

E. Result Analysis

Example build and runtime logs are available online (GitHub Gist) for build host running FreeBSD (ESP32, ESP32C6, STM32F432, SAMV71, [15]) and Linux (StarPro64, PinePhone, AvaotaA1, SG2000, BL808 [16]).

Plain-text build and runtime test results needs processing and export to a location where combined analysis, comparison, and presentation takes place from all remote test nodes. There are several possible scenarios, research in this area is still ongoing.

NuttX Dashboard [17] (Figure 5) is our own solution for information gathering and presentation in one place. It was created as an independent and alternative solution to overcome limitations of GitHub – to support missing build hosts (e.g. BSD), to cover more boards and configurations scenarios, and most importantly provide sufficient computational resources that far exceeds assigned GitHub Action usage limits. NuttX project is large, but has CI quotas equal to any other Apache project, thus we are very often at risk of loosing CI completely due to excessive overloads [18] [19].

NuttX TestBot [20] is our solution for GitHub interactions that provides a summary of test results directly into the the

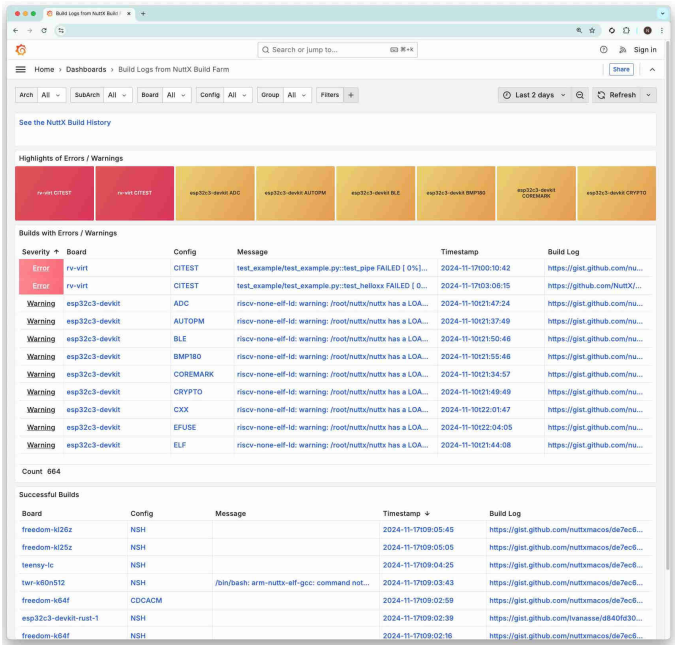


Fig. 5: NuttX Dashboard web portal.

PR conversation. Initially it was an Large Language Model (LLM) based program [21] using GitHub REST API [22] to analyze incoming PRs by performing quality overview and conformance to the Apache NuttX RTOS Contributing Guide [23]. Later on, commands such as `@nuttxpr test milkv_duos:nsh` were implemented triggering a remote build and runtime test on a real world hardware [20]. This makes whole process fully automated and therefore far more efficient.

This way not only a cyclical project upstream build and runtime validation on various real world hardware boards is possible, but also verification of incoming contributions before they are merged into the project upstream which is crucial for detecting / blocking / handling breaking changes as soon as possible.

F. NuttX Hardware Survey

In August 2025 an initial onlinie NuttX Hardware Survey was performed among the core developers team ($n = 20$) in order to identify the most popular targets, build tools, and use cases currently used. Results are presented in Figure 6. Most popular Target CPU architectures are ARM, RISC-V, Xtensa, Simulator, and AVR. Architectures like MIPS, Renesas, x86/AMD64, FPGA, and Z80 are sporadic. Most popular Build Host CPU architectures include x86_64/AMD64 and ARM(64), while x86 is used rarely, and RISC-V use still seems infrequent. Linux is most popular the choice as the Build Host OS, while macOS, BSD, and Windows are less popular. NuttX is almost equally distributed among hobby/academic/non-profit and professional/commercial community. People reported developing NuttX in Europe, North and South Americas, and Asia. This may indicate commercial vendors did not take part in the survey as NuttX powers large number of product from the Asian market. We will perform next survey in 2026 during the NuttX International Workshop in order to gather better and more detailed insight.

IV. Discussion

A. Testing Improves Quality

Creating a local hardware/firmware testing environment indeed is a complex task. NXDART aims at providing easy to use distributed solution, based on Free and Open-Source components, lowest possible hardware cost design, and automation of time-consuming tedious tasks. We believe that this will lower the cost of end-product development, maintenance, and servicing, for the project owners, as well as increase overall product quality and consumer satisfaction [24]–[29].

B. Independence

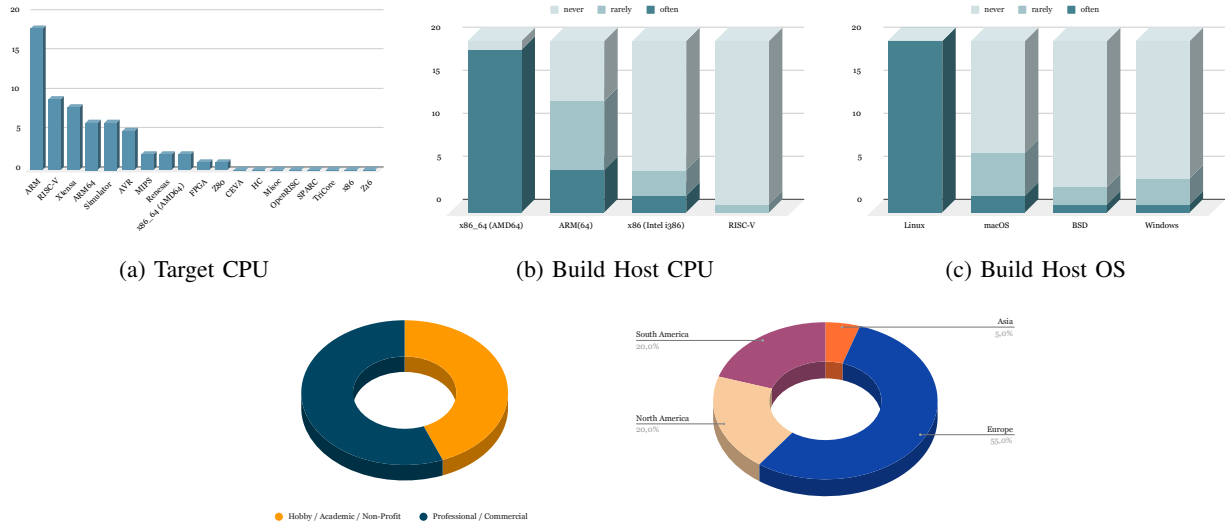
Having ready-to-go solution saves time and cost of custom firmware development, especially important for the SME (Small and Medium-sized Enterprises) sector. It makes results standardized and comparable. Provides tool set for autonomous unsupervised implementation of well-defined scenarios, as well as ad-hoc manual discovery/identification/verification/fix for unique implementation specific problems, particularly the intrinsic sparse software bugs. Anyone taking care only for their own custom hardware/product, at the same time verifies overall upstream quality, functionality, and compatibility, as the commonly shared code base.

As large companies are contributing plenty of code to NuttX, it is now possible to ensure that all code is properly tested on real-world hardware, without bias towards any specific company or vendor. Build and runtime verification can be performed by independent unpaid volunteers or professionals not affiliated with the companies contributing the code. This provides additional cross-examination of the changes, and assures long term NuttX independence, trust, adaptability and sustainability.

C. Breaking Changes

It is not entirely the contributor's fault if the code breaks at runtime because no single individual or company has access to all NuttX supported boards. Nowadays even very popular hardware vendors tend to provide new CPU/MCU/SoC devices in shorter time spans at cheaper prices, but often with low quality or completely missing documentation and no drivers. Porting code in that case is usually a time consuming experimental best-effort approach. All code contributions are welcome and appreciated, but the upstream project must be sure that code can be adopted by anyone in the community and there are no breaking changes introduced, especially those impacting overall self-compatibility and long term maintenance.

The biggest and common problem we were facing included low quality contributions with breaking code not respecting project self-compatibility, with insufficient descriptions, no documentation, no proof of testing before change was submitted. Although NXDART creates a "fire-and-forget" like solution it will not do the actual work of contributors if they are not willing to cooperate or prefer "bleeding edge"



(a) Target CPU (b) Build Host CPU (c) Build Host OS

(d) Use Cases (e) Location

Fig. 6: NuttX Hardware Survey 2025 results.

development style. It is a set of tools that are co-created by the community to protect against such extreme cases. Unlike other popular projects with big funding, NuttX is all individual, self-funded, free-time, best-effort, and purely voluntary work. We are aware that not all scenarios or devices will be covered right from the start, but the first step is made.

The future success metrics may include number of active testing users/locations, commits validated on a real world hardware, number of positive/negative reports, increased number of PRs processed monthly, decreased number of breaking changes missed and merged to upstream, decreased number of awaiting PRs, decreased number of reverts, etc. For now our greatest success is that contributions quality has already increased, changes tend to have better descriptions (or have description at all), committers are aware that code supplied needs to be tested on their side before sharing with the public (as opposed to sending untested code before), and breaking changes needs special care with blocking approach by default. NXDART is also very useful for release testing and validation, that is, when most problems are discovered and reported.

D. Self-Compatibility and Long-Term Maintenance

We decided that using classic tools for NXDART's implementation with a long history (30+ years) has an indisputable advantage – they change infrequently – proving their solid grounds of long-term stability, self-compatibility, portability, and maintenance. Unfortunately, most "modern" software tools (such as libraries, frameworks, even programming languages), although very popular, can have extremely short life cycles (measured in months or even weeks), depend on complex and fragile tree of external dependencies, require long hours of prior compilation time, that may introduce hidden maintenance costs and risks that we want to avoid.

Every senior developer or project manager experienced at least once a situation called a "maintenance nightmare" where more time and energy goes to keeping things working instead of focusing on new features development. Therefore, a minimalistic approach should provide best portability and long-term maintenance experience. Distributed build and runtime automation will also help in tool-related problem identification, because there is a high probability that different users will use different toolsets and report different problems we would not know otherwise.

E. Testing Scope

Aside from basic firmware configurations such as *board:nsh* or *boad:ostest*, there are also other in-firmware runtime testing applications already available, such as benchmarks, memory stress-testing, device drivers and communication interfaces tests, monkey tests, and many more. These may be combined with runtime tests scenarios in future and provide additional metrics such as efficiency, capabilities, power effectiveness, etc.

A careful review and cleanup of existing board configurations is planned in order to make sure all selected functionalities are coherent, for instance that *board:nsh* configuration does not contain *ostest* and other built-in features in order to provide smallest possible memory footprint reference. A matrix of available memory and hardware features for each board needs to be created, in order to have a clear view on specific hardware features and limitations that can be used per configuration.

F. Security

Security needs special consideration if the system is configured for unsupervised automatic validation of not only project upstream code but also all incoming PRs, as they may contain malicious code and scripts crafted for local in-house test nodes and network exploitation. This is especially important for nodes that run inside the enterprise environment. Master repository history may be also altered for similar actions,

therefore upstream history verification may be an optional extra feature in demand for development in future.

Well known best practices and standards such as minimal possible permissions and privileges, network separation, dedicated functional user accounts, compartmentalization, etc., should increase solution security. Virtualization/jails/sandboxing may help here also by creating a disposable single-use environment for firmware binaries and runtime test logs generation. The build process may be divided into separate stages, with only the initial fetch stage having access to the network before any code/script content is accessed and executed (just as FreeBSD Ports do).

This will, however, require a build structure update for many NuttX boards and configurations, because external dependencies fetch (i.e. SDK or libraries) is sometimes part of the build process. Security incidents have happened before in similar projects where hardware and build machines were exposed to the public without additional protection. Zero trust to user input and worst-case scenario design should be followed.

G. Hardware Limitations

During our research some hardware limitations were discovered, both on TestNode and TargetBoard side. Single USB port utilization per test board seems ideal but is not always possible. Some boards may require additional external DebugProbe for flashing and UART interaction. Some DebugProbes, although rich in features, may not work correctly with USB 2.0 ports (i.e. STLINK-V3). Some boards cannot be easily programmed over USB at all and require dedicated solutions in order to work. More details below.

H. USB Related Issues

The first and most common problem is the board identification from the TestNode operating system perspective. USB devices enumeration is rarely linear and repeatable. It is safe to assume filesystem inodes that represent real world USB devices will get random names and cannot be referenced between reboots/reconnects reliably. 50 different boards will show up as 50 serial ports and debug interfaces always in random order, unless OS specific tricks are used, but this impacts portability. To make things worse different boards (different target MCU) may expose exactly the same *VID:PID* and Serial identifiers (because of vendor mistake or purposeful cost cutting action). For us it is really hard to overcome.

UsbHubCtl utility can be a solution here. Its main purpose is power control of specified USB HUB ports, and thus attached devices. It also can list devices based on the hub hierarchy structure that is constant in time. Although this program is still in early development, may not support all hubs, and may not be yet fully ported to all possible operating systems, it seems the best possible candidate.

Some boards may be equipped with a chip (i.e. *CP2102*) that allows modification of the USB device serial string and thus board identifier customization. Not all chips allow that, while the others have OTP (One Time Programmable) memory that needs extra attention when programming as the effect of first write is irreversible. Some chips like *CH3*** / *CH4*** have still experimental drivers and incorrect support for control lines used for target board control (i.e. reset / boot signals) or even baudrate selection.

In the worst case scenario, in order to avoid manual interaction, some target boards (especially the “cheap clones”) may need a replacement with the same MCU but different USB interface chip that enables customization, use of additional external DebugProbe, or sequential power on/off to specific USB HUB ports may be required.

When more boards are available for testing, a single USB port can be extended with a good quality and medium price 16-port USB Hub, but these hubs cannot be chained together indefinitely, that limits possible USB ports count. Also long USB cable expanders usually disturbs the operations, therefore, TestNode needs to be located close to the Board. USB cables that only provide *charge* but no *data* are another very popular problem that needs additional verification. A setup with four 16-port hubs with their own dedicated power supply each was tested positively with two remarks: 1. three hubs had to be connected to the first hub as the longer chain was not possible (hubs did not enumerate), 2. hubs introduce long enumeration times by the host OS. This is not a problem for a desktop PC with standard count of 5..10 USB ports available directly from the mainboard, but can be a limitation for a laptop with only two available USB ports or SBC such as RaspberryPi Zero with only one single USB host port available. This limits the amount of possible boards depending on the underlying hardware capabilities. On the other hand small inexpensive TestNodes can be multiplied in order to cover more boards separately and provide faster parallel results. Problems pave the way for new solutions.

I. Small Boards

Most of the *small* modern development boards aside from the main MCU provide on-board debug probe exposing a single USB interface for firmware flashing, on-chip trace/debug, UART console transport, and Hardware Reset ciruit, providing full low-level access and control required for firmware development. This debug probe may even be implemented as on-chip debug block directly inside the MCU (i.e. ESP32* USB-Serial-JTAG) which seems the most convenient solution. This may not be always present, especially for older or more complex boards that by design required external programming interfaces. Some boards still require manual interaction via buttons or special dedicated in-firmware routines to select BootROM mode in order to flash firmware and perform target reset (i.e. RaspberryPi-Pico family requires in-firmware routines for reset-to-bootloader switch). All this makes full test automation harder, if possible at all, especially in case of hard kernel/driver faults. It is important to know what boards are development friendly and which ones are not (by design).

J. Big Boards

Big boards, such as single board computers (SBC) or smartphones, are known to provide only *U-Boot* or similar bootloaders, that require SD card / TFTP / X-Serial firmware image, or dedicated USB flasher application. On one hand,

this standardizes installation of various different ready to use (Real Time) OSs, but for us makes things a bit harder because the process is completely different from the rest of the boards and requires additional setup steps. Some boards will require setting up a local TFTP server and network connection to the board. Testing boards that only accept firmware images from the SD card still can be automated but requires additional *SDWire* like hardware. It may also be possible to completely replace existing bootloader with *NXBoot*, *MCUboot*, or similar that supports simpler flashing mechanisms, but the board will not be "generic" anymore and will need bootloader restoration in order to work with other operating systems as before.

V. Conclusions

Creating distributed automated build and runtime test environment is a response to problems encountered during continuous development and maintenance of the Apache NuttX RTOS.

NXDART offers almost zero-cost local test environment setup by utilizing older but still functional computer and embedded hardware. Using power efficient Single Board Computers for TestNodes implementation reduces operational costs to minimum and may even open ways for energy harvesting implementations.

NXDART fills in the crucial gap that software-only testing cannot cover by providing hardware-in-the-loop (HIL) solution that may be applied as well in various other projects.

Although its early stage of development community feedback reveals high demand for this type of tool as it may help improve project quality, compatibility, interoperability, and long-term maintenance. Its Open-Source nature makes this solution highly portable and customizable. The distributed architecture enables validation directly by end-users on real world in-house hardware, often an end-product, that provides independence and resilience against single point of failure.

References

[1] G. Nutt. Apache NuttX RTOS. [Online]. Available: https://nuttx.apache.org

[2] G. Nutt, "NuttX RTOS Beginnings," Technolution B.V. Burgemeester Jamessingel 1 2803 WV Gouda The Netherlands, Jul 2019. [Online]. Available: https://events.nuttx.apache.org/index.php/past-events/nuttx2019-international-workshop-output

[3] G. Nutt, "NuttX RTOS History," Oct. 2019. [Online]. Available: https://cwiki.apache.org/confluence/pages/viewpage.action?pageId=139629394

[4] Apache NuttX RTOS Supported Platforms. [Online]. Available: https://nuttx.apache.org/docs/latest/platforms/index.html

[5] G. Nutt, "The Inviolable Principles of NuttX." [Online]. Available: https://github.com/apache/nuttx/blob/master/INVIOLABLES.md

[6] Apache NuttX Documentation. [Online]. Available: https://nuttx.apache.org/docs/latest/

[7] R. D. H. Thomas N. Duening and M. A. Lechter, *Technology Entrepreneurship*. Elsevier. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/C20090191602

[8] A. Costin, A. Zarras, and A. Francillon, "Automated Dynamic Firmware Analysis at Scale: A Case Study on Embedded Web Interfaces," in *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*. Xi'an China: ACM, May 2016, pp. 437–448. [Online]. Available: https://dl.acm.org/doi/10.1145/2897845.2897900

[9] R. T. Tiburski, C. R. Moratelli, S. F. Johann, E. De Matos, and F. Hessel, "A lightweight virtualization model to enable edge computing in deeply embedded systems," *Softw Pract Exp*, vol. 51, no. 9, pp. 1964–1981, Sep. 2021. [Online]. Available: https://onlinelibrary.wiley.com/doi/10.1002/spe.2968

[10] A. Alvarez, R. Orea, S. Cabrero, X. G. Pañeda, R. García, and D. Melendi, "Limitations of network emulation with single-machine and distributed ns-3," in *Proceedings of the 3rd International ICST Conference on Simulation Tools and Techniques*. Malaga, Spain: ICST, 2010. [Online]. Available: http://eudl.eu/doi/10.4108/ICST.SIMUTOOLS2010.8630

[11] P. Garcia, T. Gomes, F. Salgado, J. Monteiro, and A. Tavares, "Towards hardware embedded virtualization technology: architectural enhancements to an ARM SoC," *SIGBED Rev.*, vol. 11, no. 2, pp. 45–47, Sep. 2014. [Online]. Available: https://dl.acm.org/doi/10.1145/2668138.2668145

[12] R. Verdegem and J. Van Der Hoeven, "Emulation: To Be or Not To Be," *archiving*, vol. 3, no. 1, pp. 56–60, Jan. 2006. [Online]. Available: https://library.imaging.org/archiving/articles/3/1/art00015

[13] L. Y. Lee, "Optimising the Continuous Integration for Apache NuttX RTOS (GitHub Actions)." [Online]. Available: https://lupyuen.org/articles/ci3

[14] T. Cedro, M. Kuzia, and A. Grzanka, "Libswd serial wire debug open framework for low-level embedded systems access," in *2012 Federated Conference on Computer Science and Information Systems (FedCSIS)*, 2012, pp. 615–620.

[15] T. Cedro. Build and Runtime logs. [Online]. Available: https://gist.github.com/cederom

[16] L. Y. Lee. Build and Runtime logs. [Online]. Available: https://gist.github.com/lupyuen

[17] L. Y. Lee. NuttX Dashboard. [Online]. Available: https://nuttx-dashboard.org

[18] L. Y. Lee, "Continuous Integration Dashboard for Apache NuttX RTOS." [Online]. Available: https://lupyuen.codeberg.page/articles/ci4.html

[19] L. Y. Lee. High utilisation of github runners. [Online]. Available: https://github.com/apache/nuttx/issues/17914

[20] L. Y. Lee, "Test Bot for Pull Requests ... Tested on Real Hardware (Apache NuttX RTOS / Oz64 SG2000 RISC-V SBC)." [Online]. Available: https://lupyuen.codeberg.page/articles/testbot.html

[21] L. Y. Lee, "LLM Bot that reviews Pull Requests for Apache NuttX RTOS." [Online]. Available: https://lupyuen.codeberg.page/articles/llm.html

[22] Github REST API endpoints for Pull Requests. [Online]. Available: https://docs.github.com/en/rest/pulls

[23] "Apache NuttX RTOS Contributing Guide." [Online]. Available: https://github.com/apache/nuttx/blob/master/CONTRIBUTING.md

[24] P. D. T. O'Connor, *Test engineering: a concise guide to cost-effective design, development, and manufacture*, ser. Wiley series in quality and reliability engineering. New York: Wiley, 2001.

[25] D. E. Harter and S. A. Slaughter, "Quality Improvement and Infrastructure Activity Costs in Software Development: A Longitudinal Analysis," *Management Science*, vol. 49, no. 6, pp. 784–800, Jun. 2003. [Online]. Available: https://pubsonline.informs.org/doi/10.1287/mnsc.49.6.784.16023

[26] Z. Zhou, J. Ren, X. Liu, and P. M. Pardalos, "Collaborating product development and maintenance service: quality incentive with inspection and warranty strategies," *International Journal of Production Research*, vol. 62, no. 20, pp. 7486–7503, Oct. 2024. [Online]. Available: https://www.tandfonline.com/doi/full/10.1080/00207543.2022.2110019

[27] C. Jones and O. Bonsignour, *The economics of software quality*. Upper Saddle River, NJ: Addison-Wesley, 2012.

[28] A. Mitra, *Fundamentals of quality control and improvement*, fourth edition ed. Hoboken, N.J: Wiley, 2016.

[29] E. Alégroth, R. Feldt, and P. Kolström, "Maintenance of automated test suites in industry: An empirical study on Visual GUI Testing," *Information and Software Technology*, vol. 73, pp. 66–80, May 2016. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S0950584916300118