УДК 004.45
DOI https://doi.org/10.32782/2663-5941/2026.1.2/11

Дьячук Т.С.
https://orcid.org/0000-0002-2478-0588
Національний університет «Запорізька політехніка»

Шкрябець В.І.
https://orcid.org/0009-0002-0041-6164
ТОВ «ЛОГІТЕК ЮКРЕЙН»

Голуб Т.В.
https://orcid.org/0000-0001-6024-008X
Національний університет «Запорізька політехніка»

ОЦІНЮВАННЯ ПРОДУКТИВНОСТІ ТА АПАРАТНОЇ ВЗАЄМОДІЇ GPIO У СЕРЕДОВИЩАХ KOTLIN/NATIVE ТА C/C++

Статтю присвячено експериментальному порівняльному дослідженню продуктивності та часових характеристик доступу до інтерфейсу GPIO у середовищах Kotlin/Native та C/C++. GPIO є базовим механізмом апаратної взаємодії у вбудованих і кіберфізичних системах, а його часові параметри істотно впливають на коректність функціонування систем, чутливих до затримок і нестабільності виконання. Традиційно реалізація доступу до GPIO здійснюється мовами низькорівневого програмування, зокрема C/C++, що забезпечують високу швидкодію, проте ускладнюють супровід і масштабування програмних рішень. У цьому контексті актуальним є дослідження можливостей використання високорівневих мов програмування з нативною компіляцією, зокрема Kotlin/Native, для апаратно-орієнтованих задач. У роботі розроблено експериментальний стенд на базі одноплатного комп'ютера Orange Pi Zero 3 під керуванням ОС Linux загального призначення та незалежного вимірювального модуля на мікроконтролері RP2040. Проведено серію контрольованих експериментів з генерацією періодичного цифрового сигналу на GPIO-виводі та високоточного вимірювання часових інтервалів між фронтами сигналу. Для аналізу результатів використано статистичні показники середніх значень, RMS-jitter, а також квантильний аналіз розподілу періоду та міжподієвих відхилень. Отримані результати показують, що реалізація на Kotlin/Native забезпечує часові характеристики, близькі до нативних реалізацій мовою C/C++. Середні значення міжподієвих відхилень періоду для Kotlin/Native перевищують відповідні показники C/C++ лише на 8–10 %, що є несуттєвим для більшості прикладних задач вбудованих систем. Водночас квантильний аналіз засвідчив більш компактний розподіл часових характеристик для Kotlin/Native та меншу кількість екстремальних відхилень. Виявлені поодинокі аномалії мають подібну природу для обох середовищ і зумовлені особливостями планування та керування ресурсами ОС Linux. Результати дослідження підтверджують доцільність застосування Kotlin/Native як альтернативи традиційним реалізаціям мовою C/C++ у задачах апаратної взаємодії з GPIO, де важливим є поєднання достатньої продуктивності, часової стабільності та переваг сучасної мови програмування.

Ключові слова: *GPIO, Kotlin/Native, C/C++, RMS-jitter, продуктивність, libgpiod, Linux, вбудовані системи, апаратна взаємодія, кіберфізичні системи.*

Постановка проблеми. Інтерфейс General-Purpose Input/Output (GPIO) є базовим механізмом апаратної взаємодії у вбудованих та кіберфізичних системах. Він забезпечує програмну конфігурацію цифрових ліній вводу/виводу, виконання низькорівневих операцій читання й запису, а також керування електричними режимами роботи пінів. Завдяки універсальності GPIO широко використовується на одноплатних комп'ютерах і мікроконтролерах для інтеграції сенсорів, керування виконавчими пристроями та синхронізації апаратних модулів у реальних

© Дьячук Т.С., Шкрябець В.І., Голуб Т.В., 2026
Стаття поширюється на умовах ліцензії відкритого доступу CC BY 4.0

системах, мобільних та автономних пристроях моніторингу енергетичної інфраструктури, системах обліку ресурсів, бездротових телеметричних комплексах та засобах контролю параметрів довкілля [1]. Використання відкритих апаратних платформ (Arduino, Raspberry Pi, ESP32, STM32) дає змогу оперативно створювати функціональні прототипи з оптимальними витратами, що є важливим фактором за умов обмежених бюджетів та необхідності швидкої адаптації інженерних рішень [2]. Традиційно доступ до GPIO реалізується засобами низькорівневого програмування мовою C/C++, що забезпечує високу продуктивність і мінімальні накладні витрати, однак вимагає від розробника глибокого розуміння апаратної специфіки, ручного керування ресурсами та підвищує складність супроводу програмного коду. Це ускладнює розробку масштабованих та підтримуваних систем, особливо в умовах зростаючої складності сучасних вбудованих рішень. У цьому контексті зростає інтерес до використання високорівневих мов програмування, здатних поєднувати зручність розробки з нативною продуктивністю. Kotlin, зокрема у варіанті Kotlin/Native, надає можливість створення безпосередньо виконуваного коду без віртуальної машини та підтримує прямий виклик нативних бібліотек через механізм C Interop. Це відкриває перспективи використання Kotlin для розробки апаратно-орієнтованих компонентів і предметно-орієнтованих мов (DSL) для конфігурації та керування GPIO. Разом з тим, використання додаткових рівнів абстракції та runtime-механізмів може впливати на часові характеристики доступу до апаратних ресурсів, що є критичним для задач з жорсткими або квазіжорсткими часовими обмеженнями. Відсутність достатньої кількості експериментальних досліджень, у яких Kotlin/Native та C/C++ порівнюються в однакових апаратних і програмних умовах, ускладнює обґрунтований вибір технології для систем, чутливих до затримок та jitter.

Таким чином, постає науково-практична проблема кількісного оцінювання продуктивності та часової стабільності операцій доступу до GPIO при використанні Kotlin/Native у порівнянні з традиційною реалізацією мовою C/C++, що й зумовлює необхідність проведення даного дослідження.

Аналіз останніх досліджень і публікацій. Проблематика оцінювання продуктивності та часових характеристик інтерфейсу GPIO широко представлена у сучасних дослідженнях з вбудованих і кіберфізичних систем. Наявні публікації демонструють два взаємопов’язані напрями: експериментальне вимірювання латентності та jitter доступу до GPIO на реальних апаратних платформах, а також розвиток інструментів і програмних підходів, що підвищують рівень абстракції апаратної взаємодії [3, 4]. Роботи, присвячені вимірюванню часових характеристик GPIO та переривань, показують істотний вплив способу доступу до апаратури та особливостей програмного середовища на максимальну частоту перемикань і варіативність затримок, особливо в системах під керуванням операційних систем загального призначення [4, 5]. Це обумовлює необхідність використання контрольованих експериментальних методик і зовнішніх вимірювальних засобів для отримання відтворюваних результатів. Паралельно розвиваються практичні інструменти та бібліотеки для побудови бенчмарків і автоматизованих платформ вимірювання часових та енергетичних характеристик вбудованих систем, зокрема на базі GPIO [6]. Такі підходи дозволяють аналізувати поведінку систем під навантаженням і порівнювати вплив архітектури програмного середовища та runtime на часову стабільність виконання. Разом з тим більшість існуючих робіт зосереджені на порівнянні нативних реалізацій мовою C/C++ із середовищами, що використовують віртуальні машини або інтерпретацію. Розвиток тестових наборів для реактивних і реального часу систем, зокрема EMSBench, пропонує універсальні шаблони для оцінювання часових характеристик, однак не охоплює специфіку Kotlin/Native у контексті низькорівневої апаратної взаємодії [7].

Таким чином, аналіз літератури свідчить про наявність методик і інструментів для вимірювання часових характеристик GPIO, але водночас виявляє відсутність систематичних експериментальних досліджень, у яких реалізації доступу до GPIO на Kotlin/Native та C/C++ порівнюються в однакових апаратних і програмних умовах, що визначає наукову новизну та доцільність даного дослідження.

Постановка завдання. Мета статті полягає в експериментальному порівняльному оцінюванні продуктивності та часових характеристик доступу до інтерфейсу GPIO у середовищах Kotlin/Native та C/C++. Дослідження спрямоване на кількісне порівняння затримок, jitter та швидкодії виконання низькорівневих операцій керування GPIO в однакових апаратних і програмних умовах, а також на визначення доцільності застосування Kotlin/Native як альтернативи традиційним реалізаціям мовою C/C++ у задачах апаратної взаємодії. Отримані результати мають слугувати

підґрунтям для обґрунтованого вибору технології розробки апаратно-орієнтованого програмного забезпечення залежно від вимог до продуктивності та часової стабільності системи.

Виклад основного матеріалу. На рис. 1 подано структурну схему експериментального стенду, розробленого для оцінювання продуктивності операцій доступу до GPIO у середовищах Kotlin/Native та C/C++. Стенд складається з трьох взаємопов'язаних компонентів: робочої станції, одноплатного комп'ютера Orange Pi Zero 3 та вимірювального модуля на базі мікроконтролера RP2040. Робоча станція використовується як інструментальне середовище для розробки та компіляції тестових програм, передавання виконуваних файлів на цільову платформу та подальшого збору й аналізу експериментальних даних, виконуючи координуючу роль у проведенні експерименту. Як обчислювальну платформу застосовано одноплатний комп'ютер Orange Pi Zero 3 на базі SoC Allwinner H618 (ARM Cortex-A53, 64-бітна архітектура), що функціонує під керуванням Armbian Linux v6.12 (Ubuntu 24.04) з ядром 6.12.47-current-sunxi64. Експериментальні вимірювання проводилися в середовищі стандартного (non-RT) ядра Linux, що відповідає типовим умовам використання одноплатних комп'ютерів загального призначення, за мінімального фонового навантаження для зменшення впливу сторонніх чинників. У межах дослідження Orange Pi Zero 3 використовується для запуску тестових програм, реалізованих мовами Kotlin/Native та C/C++, які формують контрольований періодичний цифровий сигнал на обраному GPIO-виводі. Згенерований сигнал подається на вимірювальний модуль, де фіксуються часові параметри перемикання та інтервали між фронтами сигналу, що дозволяє оцінити jitter і ефективну швидкодію операцій доступу до GPIO. Мікроконтролер RP2040 виконує функцію незалежного високоточного вимірювального засобу, ізольованого від впливу операційної системи. За допомогою апаратних засобів PIO та Pico SDK здійснюється фіксація фронтів сигналу й вимірювання часових інтервалів, а результати передаються на робочу станцію через інтерфейс USB/UART. Такий підхід забезпечує відтворюваність вимірювань і об'єктивність порівняння часових характеристик реалізацій на Kotlin/Native та C/C++.

Масив експериментальних даних записується на робочій станції у вигляді CSV-файлу. Кожне значення у CSV-файлах відповідає двом тактовим циклам RP2040 PIO. З урахуванням робочої частоти RP2040 PIO $f_{PIO} = 125$МГц, тривалість одного такту становить: $T_{PIO} = \frac{1}{f_{PIO}} = 8$нс, а часовий інтервал, що відповідає одному виміряному значенню: $T_{sample} = 2 \cdot T_{PIO} = 16$нс.

Для кожної реалізації (C/C++ та Kotlin/Native) було проаналізовано понад 260 тис. вимірювань

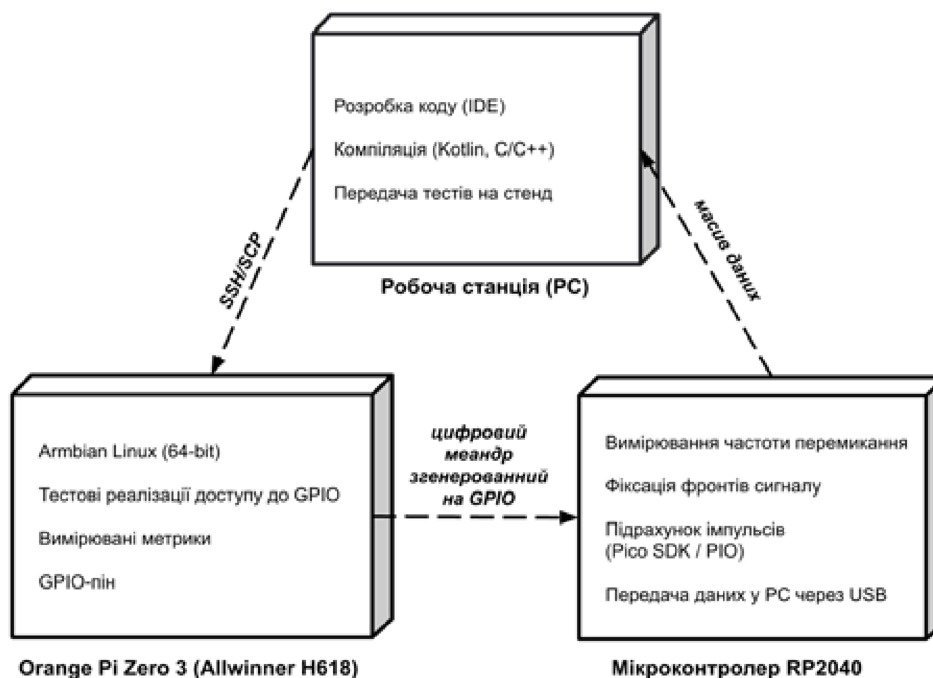


Рис. 1. Структурна схема експериментального стенду

тривалостей логічних рівнів сигналу. На основі цих даних обчислювалися такі характеристики:

– період сигналу для кожного вимірювання: $T_i = \left(t_i^{(1)} + t_i^{(0)}\right) \cdot T_{sample}$, де $t_i^{(1)}$, $t_i^{(0)}$ – тривалості логічної «1» та «0» відповідно;

– середнє значення періоду сигналу: $\overline{T} = \frac{1}{N}\sum_{i=1}^{N} T_i$, де N – кількість вимірювань;

– середню частоту перемикання GPIO визначено як обернену величину до середнього періоду сигналу, обчисленого за результатами вимірювань тривалостей логічних рівнів: $\overline{f} = \frac{1}{\overline{T}}$;

– RMS-jitter, визначений як середньоквадратичне відхилення періоду сигналу від його середнього значення, використано як метрику часової стабільності: $J_{RMS} = \sqrt{\frac{1}{N}\sum_{i=1}^{N}\left(T_i - \overline{T}\right)^2}$;

– максимальні відхилення (аномальні значення): $J_{max} = max\left|T_i - \overline{T}\right|$.

Для переходу до фізичних одиниць часу всі значення з CSV ($value_{csv}$) були помножені на $T_{sample} = 16$нс: $T_i = value_{csv} \cdot T_{sample}$.

Аналіз експериментальних даних (табл. 1) свідчить, що реалізація на C/C++ забезпечує менший середній період сигналу та, відповідно, вищу середню частоту перемикання GPIO, що узгоджується з мінімальними накладними витратами нативного коду та прямою взаємодією з бібліотекою libgpiod. Реалізація на Kotlin/Native характеризується більшим середнім періодом і нижчою частотою перемикання, що може бути зумовлено додатковими рівнями абстракції та накладними витратами, пов’язаними з використанням механізмів FFI. Значення RMS-jitter для обох реалізацій мають однаковий порядок величини (≈200 нс), однак для C/C++ воно є дещо вищим. Це пояснюється наявністю поодиноких вимірювань із суттєвим відхиленням тривалості періоду, що підтверджується значно більшим максимальним відхиленням (≈30800 нс). Для реалізації на Kotlin/Native розподіл тривалостей періоду є більш компактним і містить меншу кількість екстремальних відхилень, унаслідок чого RMS-jitter є нижчим, попри більший середній період сигналу. Таким чином, реалізація на C/C++ забезпечує вищу пікову продуктивність операцій доступу до GPIO, тоді як Kotlin/Native демонструє більш стабільну часову поведінку за рахунок меншої кількості аномальних відхилень.

Для детальнішої оцінки часових характеристик сигналу GPIO, поряд із середніми значеннями та RMS-метриками, було застосовано квантильний аналіз розподілу періоду та cycle-to-cycle jitter. Такий підхід дозволяє охарактеризувати як типовий режим роботи, так і крайні часові відхилення, а також є менш чутливим до поодиноких викидів, ніж RMS-оцінки. Результати квантильного аналізу (табл. 2) показують, що для реалізації на C/C++ нижні та медіанні квантили періоду (P5–P50) зосереджені у вузькому діапазоні значень, що відповідає стабільній генерації сигналу у типовому режимі та забезпечує вищу номінальну частоту перемикання GPIO. Водночас у верхніх квантилях (P95) спостерігається різке зростання як тривалості періоду, так і модуля jitter, що вказує на наявність поодиноких значних часових відхилень. Саме ці рідкісні аномалії формують підвищені значення RMS-jitter, незважаючи на високу концентрацію більшості вимірювань поблизу середнього значення. Для реалізації на Kotlin/Native розподіл періоду зміщений у бік більших значень, що відображає додаткові накладні витрати runtime та механізмів інтероперабельності. Разом

Таблиця 1

Узагальнені статистичні характеристики сигналів GPIO для реалізацій на C/C++ та Kotlin/Native

Реалізація	Середній період, нс	Середня частота, МГц	RMS-jitter, нс	Макс. відхилення, нс
C/C++	2199	0.455	210	30760
Kotlin/Native	2393	0.419	202	26390

Таблиця 2

Квантильний аналіз періоду та jitter GPIO

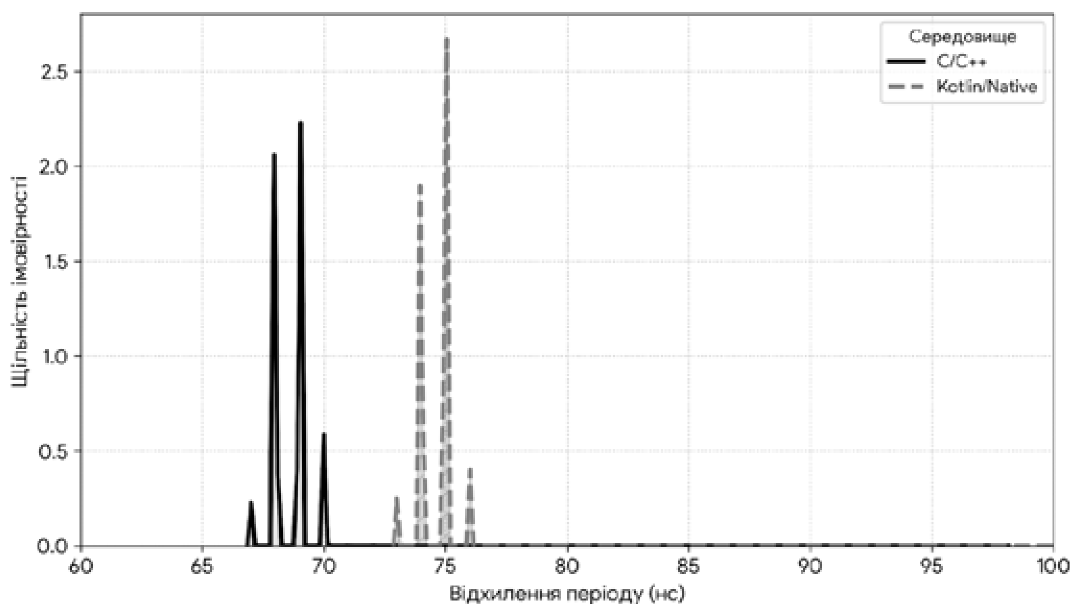
Квантиль	C/C++ період, нс	Kotlin/Native період, нс	C/C++ jitter, нс	Kotlin/Native jitter, нс
0.05	2176	2368	7.42	7.22
0.25	2192	2384	7.42	7.22
0.50	2192	2384	7.42	8.78
0.75	2208	2400	8.58	8.78
0.95	2208	2400	23.42	24.78

з тим квантильний аналіз демонструє більш компактний розподіл і відсутність різко виражених екстремальних викидів, унаслідок чого зростання модуля jitter у верхніх квантилях є більш плавним і передбачуваним. Таким чином, квантильний аналіз доповнює результати середніх і RMS-оцінок та дозволяє виявити принципову відмінність часової поведінки досліджуваних підходів: реалізація на C/C++ забезпечує максимальну швидкодію GPIO, однак характеризується наявністю рідкісних, але значних часових аномалій, тоді як Kotlin/Native демонструє більш рівномірну та детерміновану часову поведінку за рахунок зменшення частоти перемикання.

Для візуалізації типової часової поведінки сигналу на рис. 2 представлено основний діапазон міжподієвих відхилень періоду. Графік розподілу щільності показує, що основний масив значень для обох середовищ зосереджений у вузькому діапазоні, що свідчить про стабільне виконання тестових програм. Для реалізації на C/C++ медіанне відхилення інтервалу становить 69 нс, тоді як для Kotlin/Native – 75 нс, що відповідає різниці близько 8,7% і підтверджує ефективність механізмів інтероперабельності Kotlin/Native при роботі з бібліотекою libgpiod. У таблиці до рис. 2 також наведено максимальні значення міжподієвих відхилень, які досягають 1714–1979 нс. Оскільки такі аномалії спостерігаються в обох середовищах з порівнюваною амплітудою, вони інтерпретуються як прояви недетермінованості виконання в операційній системі загального призначення, а не як особливості конкретної мови програмування.

Узагальнюючи результати дослідження, можна відзначити, що реалізація на C/C++ забезпечує максимальну швидкодію доступу до GPIO за рахунок мінімальних накладних витрат, однак може характеризуватися поодинокими значними часовими відхиленнями, які впливають на RMS-jitter. Kotlin/Native, у свою чергу, демонструє більш рівномірну та передбачувану часову поведінку, хоча й поступається C/C++ за середньою частотою перемикання. Отримані результати свідчать, що вибір мови програмування для задач GPIO повинен враховувати не лише максимальну швидкодію, але й вимоги до стабільності та передбачуваності часових характеристик виконання.

Висновки. У ході дослідження виконано експериментальне порівняння продуктивності доступу до інтерфейсу GPIO у середовищах Kotlin/Native та C/C++. Отримані результати свідчать, що Kotlin/Native забезпечує часові характеристики, наближені до нативних реалізацій мовою C/C++.



Показник (нс)	C/C++	Kotlin/Native
Медіана (типове)	69.0	75.0
Середнє значення	68.73	74.77
Максимум (викид)	1979.0	1714.0

Рис. 2. Розподіл щільності імовірності часових відхилень періоду та порівняльні статистичні характеристики сигналу GPIO для реалізацій на C/C++ та Kotlin/Native

Середні значення міжподієвих відхилень періоду (що відображають ефективну часову затримку доступу до GPIO) для Kotlin/Native становлять близько 74–75 нс, що лише на 8–10 % перевищує відповідні показники для C/C++ (68–69 нс) і є несуттєвим для більшості прикладних задач вбудованих систем. Обидві реалізації демонструють високу повторюваність і стабільність часових характеристик. Виявлені поодинокі аномальні відхилення мають подібну природу в обох середовищах, що вказує на їх зумовленість особливостями планування та керування ресурсами в ОС Linux загального призначення [3], а не специфікою мов програмування чи механізмів компіляції. Це підтверджує, що використання Kotlin/Native не призводить до додаткової часової нестабільності під час роботи з низькорівневими апаратними інтерфейсами. Застосування механізму C Interop у Kotlin/Native забезпечує ефективну взаємодію з бібліотекою libgpiod із мінімальними накладними витратами, що робить дане середовище придатним для розробки апаратно-орієнтованого програмного забезпечення, яке поєднує вимоги до продуктивності з перевагами сучасної мови програмування, зокрема безпекою типів і високим рівнем абстракції. Перспективи подальших досліджень пов'язані з аналізом впливу механізмів керування пам'яттю Kotlin/Native на часові характеристики за умов підвищеного навантаження, а також із проведенням аналогічних експериментів у середовищі ядер Linux з підтримкою реального часу для оцінювання граничних можливостей застосування Kotlin/Native у системах із жорсткими часовими вимогами.

Список літератури:

1. Harms L., Richter C., Landsiedel O. Grace: Low-Cost Time-Synchronized GPIO Tracing for IoT Testbeds. *18th Annual International Conference on Distributed Computing in Sensor Systems (DCOSS).* 2022. P. 9–16. DOI: https://doi.org/10.1109/DCOSS54816.2022.00013
2. Mathe S.E., Kondaveeti H.K., Vappangi S., Vanambathina S.D., Kumaravelu N.K. A comprehensive review on applications of Raspberry Pi. *Computer Science Review.* 2024. Vol. 52. DOI: https://doi.org/10.1016/j.cosrev.2024.100636
3. Adam G.K. Real-Time Performance and Response Latency Measurements of Linux Kernels on Single-Board Computers. *Computers*. 2021, 10(5), 64. DOI: https://doi.org/10.3390/computers10050064
4. Lombardi D., Barbareschi M., Barone S., Casola V. A comprehensive evaluation of interrupt measurement techniques for predictability in safety-critical systems. *Proceedings of the 19th International Conference on Availability, Reliability and Security (ARES 2024). ACM.* 2024. Article No.: 172, P. 1–10. DOI: https://doi.org/10.1145/3664476.3670451.
5. Alonso S., Lázaro J., Jiménez J., Bidarte U., Muguira L. Evaluating Latency in Multiprocessing Embedded Systems for the Smart Grid. *Energies.* 2021. 14(11), 3322. DOI: https://doi.org/10.3390/en14113322.
6. Wu H., Chen C., Weng K. Two Designs of Automatic Embedded System Energy Consumption Measuring Platforms Using GPIO. *Applied Sciences*. 2020. 10(14), 4866. DOI: https://doi.org/10.3390/app10144866.
7. Kluge F., Rochange C., Ungerer T. EMSBench: Benchmark and Testbed for Reactive Real-Time Systems. *Leibniz Transactions on Embedded Systems (LITES)*. 2017. 4(2), 02:1–02:23. DOI: https://doi.org/10.4230/LITES-v004-i002-a002.

Diachuk T.S., Shkriabets V.I., Holub T.V. PERFORMANCE EVALUATION AND HARDWARE INTERACTION OF GPIO IN KOTLIN/NATIVE AND C/C++ ENVIRONMENTS

The article presents an experimental comparative study of the performance and temporal characteristics of GPIO interface access in Kotlin/Native and C/C++ environments. The GPIO interface serves as a fundamental mechanism for hardware interaction in embedded and cyber-physical systems, where its timing parameters significantly impact the operational correctness of systems sensitive to latency and execution jitter. Traditionally, GPIO access is implemented using low-level programming languages, specifically C/C++, which ensure high performance but pose challenges for the maintenance and scaling of software solutions. In this context, it is relevant to investigate the feasibility of using high-level programming languages with native compilation, such as Kotlin/Native, for hardware-oriented tasks. In this work, an experimental testbed was developed based on the Orange Pi Zero 3 single-board computer running a general-purpose Linux OS and an independent measurement module based on the RP2040 microcontroller. A series of controlled experiments were conducted, involving the generation of a periodic digital signal on a GPIO pin and high-precision measurement of time intervals between signal edges. For the results analysis, statistical indicators such as mean values, RMS-jitter, and quantile analysis of period distribution and inter-event deviations were utilized. The obtained results demonstrate that the Kotlin/Native implementation provides temporal characteristics close to those of native C/C++ implementations. The mean values of inter-event period deviations for Kotlin/Native exceed the

corresponding C/C++ metrics by only 8–10%, which is insignificant for most embedded system applications. Furthermore, quantile analysis revealed a more compact distribution of temporal characteristics for Kotlin/Native and a lower frequency of extreme deviations. The identified isolated anomalies have a similar nature in both environments and are attributed to the peculiarities of the Linux OS scheduler and resource management. The research results confirm the feasibility of employing Kotlin/Native as a viable alternative to traditional C/C++ implementations in GPIO hardware interaction tasks where a balance of sufficient performance, temporal stability, and the advantages of a modern programming language is required.

Keywords: *GPIO, Kotlin/Native, C/C++, RMS-jitter, performance, libgpiod, Linux, embedded systems, hardware interaction, cyber-physical systems.*

Дата першого надходження статті до видання: 15.01.2026
Дата прийняття статті до друку після рецензування: 18.02.2026
Дата публікації (оприлюднення) статті: 08.04.2026