Міністерство освіти і науки України
Харківський національний університет імені В. Н. Каразіна
Навчально-науковий інститут комп'ютерних наук та штучного інтелекту Кафедра комп'ютерних систем та робототехніки

До захисту допущено
Кафедрою комп'ютерних систем та робототехніки
протокол № __ від __ червня 2026 р.
завідувач кафедри _____ Максим ХРУСЛОВ
(підпис)
«___»_____ 2026 р.

Кваліфікаційна робота

здобувача першого (бакалаврського) рівня вищої освіти

«КОМП'ЮТЕРНА СИСТЕМА УПРАВЛІННЯ ПІНАМИ МІКРОКОНТРОЛЕРА ESP32 ЗА ДОПОМОГОЮ РЕГІСТРІВ»

Спеціальність *151 – «**Автоматизація та комп'ютерно-інтегровані технології**»*
Освітня програма **Автоматизація та комп'ютерно-інтегровані технології**

Виконавець _____ **Юрій СОЛЯНИК**
(підпис)

Науковий керівник _____ **Альберт КОТВИЦЬКИЙ**
(підпис)

Харків – 2026

АНОТАЦІЯ

Пояснювальна записка до кваліфікаційної роботи бакалавра складається зі вступу, чотирьох розділів, висновків, списку використаних джерел і чотирьох додатків. Загальний обсяг роботи складає 76 сторінок, із яких 34 сторінки основної частини з 5 рисунками, 1 таблицею, 16 найменуванням у списку використаних джерел.

Метою кваліфікаційної роботи є мінімізація апаратних і програмних затримок при керуванні пінами мікроконтролера ESP32 шляхом розробки комп'ютерної системи прямого доступу до регістрів.

Об'єкт дослідження – процес програмного керування підсистемою вводу-виводу сучасних мікроконтролерів.

Предмет дослідження – апаратні регістри мультиплексора (IO_MUX) та GPIO Matrix мікроконтролера ESP32.

Проблема, яка вирішується в роботі, полягає в усуненні затримок від стандартних бібліотек та забезпеченні високошвидкісного детермінованого керування портами.

Розроблений Статичний API (CRTP) дозволяє зменшити час перемикання піна до 50,01 нс (у 11,63 раза швидше за digitalWrite()).

Динамічний API забезпечує прискорення у 6,49 раза (89,61 нс).

Максимальна частота генерації досягла 10 МГц.

Область застосування – автоматизація, вбудовані системи реального часу, Інтернет речей (IoT).

Ключові слова: автоматизація та комп'ютерні інтегровані технології, ESP32, GPIO, регістри, апаратні затримки, мікроконтролер, оптимізація, FreeRTOS, Zero-Cost Abstractions.

ABSTRACT

The explanatory note to the bachelor's qualification work consists of an introduction, four chapters, conclusions, a list of used sources and four appendices. The total volume of the work is 78 pages, of which 35 pages of the main part with 5 figures, 1 table, 16 names in the list of used sources and four appendices.

The purpose of the qualification work is to minimize the hardware and software delays in controlling the pins of the ESP32 microcontroller by developing a computer system for direct register access.

The object of the study is the process of software control of the input-output subsystem of modern microcontrollers.

The subject of the study is the hardware registers of the multiplexer (IO_MUX) and GPIO Matrix of the ESP32 microcontroller.

The problem solved in the work is to eliminate delays from standard libraries and ensure high-speed deterministic port control.

The developed Static API (CRTP) allows you to reduce the pin switching time to 50.01 ns (11.63 times faster than digitalWrite()).

The Dynamic API provides an acceleration of 6.49 times (89.61 ns).

The maximum generation frequency reached 10 MHz.

Scope – automation, embedded real-time systems, Internet of Things (IoT).

Keywords: automation and computer integrated technologies, ESP32, GPIO, registers, hardware delays, microcontroller, optimization, FreeRTOS, Zero-Cost Abstractions.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ ТА СКОРОЧЕНЬ……..6

ВСТУП.....7

РОЗДІЛ 1. АНАЛІЗ МЕТОДІВ КЕРУВАННЯ GPIO ТА АРХІТЕКТУРИ ESP32…….8

1.1. Аналіз існуючих методів керування портами мікроконтролерів.....8

1.2. Архітектурні особливості підсистеми GPIO: GPIO Matrix та IO_MUX.....9

1.3. Memory-Mapped I/O та проблема атомарності операцій.....10

1.4. Вплив операційної системи реального часу (FreeRTOS) на затримки та апаратний джитер.....11

1.5. Розглянуті альтернативи та обґрунтування обраного підходу.....12

1.6. Аналіз вимог до системи керування GPIO……..13

Висновки за розділом 1.....14

РОЗДІЛ 2. РОЗРОБКА АРХІТЕКТУРИ СИСТЕМИ КЕРУВАННЯ ПІНАМИ.....16

2.1. Обґрунтування гібридної архітектури керування GPIO.....17

2.2. Парадигма Zero-Cost Abstractions як основа проектування……18

2.3. Проектування статичного API на базі патерну CRTP.....19

2.4. Проектування динамічного API через кешування адрес пам’яті.....20

2.5. Строга типізація (Type Safety) та патерн Фасад.....22

2.6. Взаємодія підсистем та цілісність архітектури.....22

Висновки за розділом 2.....23

РОЗДІЛ 3. ПРОГРАМНА РЕАЛІЗАЦІЯ СИСТЕМИ НА C++.....24

3.1. Загальна організація програмного модуля.....25

3.2. Реалізація статичного API.....24

3.3. Реалізація динамічного API.....25

3.4. Реалізація обгорток типобезпеки та патерну Фасад.....27
3.5. Реалізація вимірювального стенду.....28
Висновки за розділом 3.....29
РОЗДІЛ 4. ЕКСПЕРИМЕНТАЛЬНЕ ДОСЛІДЖЕННЯ ШВИДКОДІЇ.....30
4.1. Методика проведення експерименту.....30
4.2. Тест 1. Базова лінія: стандартний digitalWrite().....33
4.3. Тест 2. Статичний API (CRTP) 33
4.4. Тест 3. Динамічний API.....34
4.5. Випробування механізму типобезпеки.....35
4.6. Зведений порівняльний аналіз та оцінка достовірності.....36
4.7. Область практичного застосування.....39
Висновки за розділом 4.....40
ВИСНОВКИ.....41
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ.....43
ДОДАТКИ.....45

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ ТА СКОРОЧЕНЬ

ALU — Arithmetic Logic Unit (Арифметико-логічний пристрій)

API — Application Programming Interface (Інтерфейс прикладного програмування)

APB — Advanced Peripheral Bus (Шина периферійних пристроїв)

CPU — Central Processing Unit (Центральний процесор)

CRTP — Curiously Recurring Template Pattern (Шаблон рекурсивного успадкування)

GPIO — General-Purpose Input/Output (Інтерфейс вводу-виводу загального призначення)

HAL — Hardware Abstraction Layer (Рівень апаратних абстракцій)

I2C — Inter-Integrated Circuit (Міжмікросхемний інтерфейс)

IDE — Integrated Development Environment (Інтегроване середовище розробки)

IoT — Internet of Things (Інтернет речей)

ISR — Interrupt Service Routine (Підпрограма обробки переривань)

MMIO — Memory-Mapped Input/Output (Ввід-вивід, відображений на пам'ять)

PWM — Pulse Width Modulation (Широтно-імпульсна модуляція)

RTOS — Real-Time Operating System (Операційна система реального часу)

SPI — Serial Peripheral Interface (Послідовний периферійний інтерфейс)

WDT — Watchdog Timer (Сторожовий таймер)

ООП — Об'єктно-орієнтоване програмування

LEDC — LED Control (контролер світлодіодного ШІМ)

MCPWM — Motor Control PWM (контролер ШІМ для керування двигунами)

NOP — No Operation (інструкція «немає операції»)

vtable — Virtual Method Table (таблиця віртуальних методів)

ВСТУП

Актуальність роботи. В епоху стрімкого розвитку Інтернету речей (IoT) та вбудованих систем мікроконтролери серії ESP32 стали стандартом де-факто завдяки високій продуктивності. Проте під час розробки систем реального часу розробники стикаються з проблемою апаратних та програмних затримок при роботі зі стандартними бібліотеками (Arduino HAL або ESP-IDF). Використання прямого доступу до регістрів є актуальною задачею для усунення цих затримок.

Метою дослідження є мінімізація апаратних і програмних затримок при керуванні пінами мікроконтролера ESP32 шляхом розробки комп'ютерної системи прямого доступу до регістрів.

Об’єкт дослідження – процес програмного керування підсистемою вводу-виводу сучасних мікроконтролерів.

Методи дослідження: аналіз технічної документації, емпіричне вимірювання затримок за допомогою програмного бенчмаркінгу та осцилографа, тестування програмного забезпечення.

Предмет дослідження – апаратні регістри мультиплексора (IO_MUX) та GPIO Matrix мікроконтролера ESP32.

Завдання дослідження

1. Провести аналіз існуючих методів керування портами мікроконтролерів (стандартні бібліотеки проти прямого доступу) та визначити їхні недоліки для задач автоматизації.
2. Дослідити архітектурні особливості мікроконтролера ESP32 в контексті роботи підсистеми введення-виведення (GPIO Matrix, IO_MUX).
3. Розробити програмну систему для прямого керування пінами через регістри з метою мінімізації затримок.
4. Провести експериментальне дослідження розробленої системи та оцінити її ефективність.

РОЗДІЛ 1
АНАЛІЗ МЕТОДІВ КЕРУВАННЯ GPIO ТА АРХІТЕКТУРИ ESP32

Сучасні мікроконтролери родини ESP32 посіли провідні позиції в розробці вбудованих систем завдяки поєднанню високої обчислювальної потужності (до 240 МГц).

Однак під час розробки критичних до часу (time-critical) додатків інженери часто стикаються з проблемою апаратних та програмних затримок при роботі з підсистемою введення-виведення (GPIO). Для подолання цих обмежень виникає об'єктивна необхідність переходу від високорівневих абстракцій до прямого керування периферією на рівні регістрів.

1.1. Аналіз існуючих методів керування портами мікроконтролерів

У класичній парадигмі програмування мікроконтролерів ESP32 розробники покладаються на рівень апаратних абстракцій (Hardware Abstraction Layer - HAL).

У середовищі ESP-IDF базовою функцією для зміни стану піна є gpio_set_level()[2], тоді як у фреймворку Arduino використовується digitalWrite().

Ці високорівневі API забезпечують безпеку, зручність використання та портативність коду між різними архітектурами, проте вносять значні накладні витрати на рівні процесорного часу.

При виклику таких функцій процесор змушений виконувати багатоетапний шлях трансляції: перевірку валідності вхідних параметрів, трансляцію логічного номера піна у фізичний індекс, зчитування поточної конфігурації порту, і лише після цього — фізичний запис у пам'ять периферії.

Як показують теоретичні оцінки продуктивності в ідеальних синтетичних умовах, на мікроконтролері ESP32 із тактовою частотою 240 МГц базова функція digitalWrite() витрачає від 58 до 80 процесорних тактів на одне перемикання, що теоретично обмежує частоту на рівні 1,5 – 2 МГц [5].

Проте в реальних умовах жорсткого реального часу, під управлінням FreeRTOS та з обов'язковим використанням критичних секцій для захисту від джитера, ці показники значно погіршуються через накладні системні витрати.

Функція gpio_set_level() є ще більш ресурсоємною: її виконання займає понад 88 тактів, а латентність становить близько 370 наносекунд (максимальна частота ~1,35 МГц).

Отже, використання стандартних методів для програмної емуляції високошвидкісних протоколів (bit-banging), наприклад, паралельних інтерфейсів дисплеїв чи протоколів I2C/SPI з нестандартними параметрами, є неефективним.

1.2. Архітектурні особливості підсистеми GPIO: GPIO Matrix та IO_MUX

Для розуміння природи апаратних затримок необхідно детально розглянути шлях проходження сигналу від процесорного ядра до фізичного піна (контакту) мікроконтролера.

ESP32 використовує гнучку систему маршрутизації, що складається з мультиплексора вводу-виводу (IO_MUX) та матриці комутації (GPIO Matrix).

IO_MUX відповідає за фінальну конфігурацію фізичного контакту [3], включаючи його підтягуючі резистори (pull-up/pull-down) та струм навантаження [4].

GPIO Matrix розташована між внутрішніми периферійними модулями та IO_MUX, дозволяючи перенаправляти будь-який внутрішній сигнал на майже будь-який вільний фізичний пін.

Попри надзвичайну гнучкість, проходження сигналу через матрицю GPIO має свою ціну: воно вносить затримку синхронізації, що дорівнює одному такту шини APB [1] (Advanced Peripheral Bus), яка працює на частоті 80 МГц.

Ця затримка у 12,5 наносекунд може накопичуватися, що призводить до порушення таймінгів (setup and hold times) у високошвидкісних шинах передачі даних.

Щоб уникнути цього обмеження, найважливіші периферійні пристрої можна підключати до їхніх «рідних» пінів безпосередньо через «швидкий шлях» (Fast Path) модуля IO_MUX, повністю обходячи матрицю GPIO.

У такому режимі сигнал не проходить через комутаційну матрицю, а отже, не накопичує такт синхронізації шини APB. Це принципово важливо для високошвидкісних інтерфейсів (SPI, паралельні шини дисплеїв), де навіть одиничний такт затримки на кожній лінії здатний порушити узгодженість фронтів (setup and hold times) між сигналами даних і тактування.

Саме тому розроблена в роботі система орієнтована на прямий запис у регістри портів, що дозволяє керувати пінами з мінімальною кількістю проміжних апаратних ланок.

1.3. Memory-Mapped I/O та проблема атомарності операцій

Керування станом пінів у мікроконтролерах ESP32 здійснюється через механізм вводу-виводу, відображеного на пам'ять (Memory-Mapped I/O — MMIO).

Традиційне перемикання стану піна здійснюється зміною біта у базовому регістрі GPIO_OUT_REG.

Проте такий підхід у багатоядерному середовищі є вкрай небезпечним.

Звичайна бітова операція на апаратному рівні виконується як послідовність Read-Modify-Write (Прочитати-Змінити-Записати).

Якщо в цей час інше ядро процесора (або переривання) спробує змінити стан сусіднього піна, це призведе до стану перегонів (race condition) і втрати однієї з команд.

Для розв'язання цієї проблеми інженери Espressif передбачили атомарні регістри керування [1]: GPIO_OUT_W1TS_REG (Write 1 To Set) для встановлення високого рівня та GPIO_OUT_W1TC_REG (Write 1 To Clear) для скидання в нуль.

Окремої уваги заслуговує організація адресного простору регістрів виводу.

Через те, що мікроконтролер ESP32 має понад 32 канали GPIO, а ширина апаратного регістра становить лише 32 біти, фізичні піни розподілено між двома

незалежними банками регістрів. Перший банк обслуговує піни 0–31 (регістри GPIO_OUT_W1TS_REG / GPIO_OUT_W1TC_REG), тоді як другий банк відповідає за піни 32–39 (регістри GPIO_OUT1_W1TS_REG / GPIO_OUT1_W1TC_REG).

Така апаратна особливість означає, що для коректного формування бітової маски необхідно не лише обчислити зсув (1 << pin), а й визначити, до якого саме банку належить пін, із відповідним коригуванням номера для другого банку (pin − 32).

У стандартних бібліотеках цей вибір виконується під час кожного виклику функції, що додає умовне розгалуження. Як буде показано в Розділі 2, розроблена система усуває ці витрати, переносячи вибір банку на етап компіляції за допомогою конструкції if constexpr.

Запис одиниці у відповідний біт цих регістрів виконується за одну апаратну транзакцію, гарантуючи потокобезпеку (thread safety) без необхідності використання ресурсоємних програмних м'ютексів.

1.4. Вплив операційної системи реального часу (FreeRTOS) на затримки та апаратний джитер

Значним джерелом програмних затримок та "джитера" (нестабільності часових інтервалів сигналу) є вбудована ОС реального часу FreeRTOS [6 , 7].

За замовчуванням в середовищі Arduino для ESP32 планувальник задач (scheduler) генерує системний тік (tick) щомілісекунди.

Під час кожного тіка ядро оцінює потребу в перемиканні контексту між задачами, що вимагає збереження поточного стану процесора у пам'ять.

Якщо в цей момент генерується високочастотний сигнал, контекстне перемикання внесе непередбачувану мікросекундну затримку.

Іншим критичним аспектом є механізм обробки переривань (Interrupt Handling).

Коли виникає апаратне переривання, ядро процесора повинно зберегти поточний контекст задачі, перейти до вектора переривання та виконати підпрограму обробки (ISR).

Час реакції системи від моменту фізичної зміни сигналу на піні до виконання першої інструкції в ISR (interrupt latency) може коливатися від 1 до 5 мікросекунд через накладні витрати операційної системи та диспетчера переривань.

Щоб пом'якшити цю проблему, розробникам доводиться обгортати критичні секції коду макросами заборони переривань, проте тривале блокування призводить до спрацювання сторожового таймера (WDT) та перезавантаження мікроконтролера.

1.5. Розглянуті альтернативи та обґрунтування обраного підходу

Для вирішення проблеми апаратних та програмних затримок розробники можуть використовувати кілька підходів, кожен з яких має свої обмеження:

Механізм Dedicated GPIO (Fast I/O): окремі модифікації мікроконтролерів родини ESP32 містять виділену пряму шину [1] між ядром процесора та групою пінів, що повністю оминає повільну матрицю GPIO. Згідно з тестами, такий метод дозволяє генерувати стабільний сигнал типу "меандр" з частотою понад 30–40 МГц. Проте цей функціонал апаратно відсутній на цільовій платформі, обраній для цієї роботи, тому застосувати його неможливо.

Спеціалізовані апаратні ШІМ-контролери (LEDC та MCPWM): Альтернативою програмному керуванню портами є використання вбудованих апаратних ШІМ-контролерів. Оскільки перемикання відбуваються на апаратному рівні (без генерації програмних переривань), джитер повністю відсутній. Проте цей метод має суттєвий недолік: він жорстко прив'язує генерацію сигналу до конкретних апаратних виводів мікроконтролера (таймерних каналів), що критично обмежує гнучкість розробки.

Обґрунтування обраного підходу: Оскільки апаратні таймери обмежують гнучкість, а Dedicated GPIO залежить від архітектури чипа, найбільш універсальним та ефективним рішенням є програмне керування через прямий запис у атомарні регістри пам'яті (W1TS/W1TC). Щоб уникнути помилок ручного керування адресами та накладних витрат у процесі виконання (runtime), у даній роботі пропонується застосувати парадигму сучасного C++ — Абстракції з нульовою вартістю (Zero-Cost Abstractions) [8, 9].

Такий підхід дозволяє перенести обчислення на етап компіляції, створюючи зручний об'єктно-орієнтований інтерфейс (API), який за швидкодією не поступається прямому асемблерному коду.

1.6. Аналіз вимог до системи керування GPIO

На підставі проаналізованих недоліків стандартних бібліотек та архітектурних особливостей мікроконтролера ESP32 сформульовано перелік вимог, яким повинна відповідати розроблювана система прямого керування підсистемою вводу-виводу. Вимоги поділено на функціональні, що визначають склад можливостей системи, та нефункціональні, що визначають якісні характеристики її роботи.

До функціональних вимог віднесено такі:

По-перше, система повинна забезпечувати ініціалізацію будь-якого доступного піна мікроконтролера в режимі входу або виходу через пряме звернення до апаратних регістрів, без використання повільних високорівневих функцій.

По-друге, система має підтримувати повний набір операцій керування станом виходу: встановлення високого рівня, скидання в низький рівень та перемикання стану, а також зчитування поточного логічного рівня входу.

По-третє, система повинна коректно працювати з усіма каналами GPIO, враховуючи апаратний поділ пінів на два банки регістрів (0–31 та 32–39), і автоматично визначати правильний банк для кожного піна.

До нефункціональних вимог віднесено такі:

Швидкодія. Час одного перемикання стану піна має бути суттєво меншим за показник стандартної функції digitalWrite(), а досяжна частота генерації сигналу — наближатися до межі 10 МГц, необхідної для програмної емуляції високошвидкісних протоколів.

Потокобезпека. Операції зміни стану піна повинні бути захищені від виникнення стану перегонів (race condition) у багатоядерному середовищі за рахунок використання атомарних апаратних транзакцій.

Безпека типів. Архітектура має унеможливлювати помилкові операції (наприклад, спробу запису у пін, налаштований як вхід) ще на етапі компіляції, запобігаючи апаратним конфліктам на фізичному пристрої.

Детермінованість. Система повинна забезпечувати мінімальний джитер вихідного сигналу, що вимагає ізоляції критичних ділянок коду від впливу планувальника операційної системи FreeRTOS.

Гнучкість та переносимість. Інтерфейс системи має залишатися зручним і об'єктно-орієнтованим, а її архітектура — придатною для адаптації до інших мікроконтролерних платформ із мінімальними змінами коду.

Сформульовані вимоги є основою для проектування архітектури системи, що розглядається в наступному розділі.

Висновки за розділом 1

У першому розділі проаналізовано існуючі високорівневі методи керування портами мікроконтролерів (digitalWrite, gpio_set_level) та виявлено їхні недоліки, пов'язані з великими програмними затримками.

Досліджено архітектуру ESP32 (GPIO Matrix, IO_MUX) та проаналізовано джерела апаратного джитера, викликані роботою планувальника ОС FreeRTOS та обробкою переривань.

Обґрунтовано відмову від стандартних бібліотек та апаратних таймерів на користь прямого атомарного доступу до регістрів пам'яті із застосуванням парадигми Zero-Cost Abstractions у C++.

РОЗДІЛ 2
РОЗРОБКА АРХІТЕКТУРИ СИСТЕМИ КЕРУВАННЯ ПІНАМИ

Розробка ефективного інструменту прямого керування підсистемою вводу-виводу мікроконтролера ESP32 вимагає вирішення класичного компромісу між максимальною швидкодією та гнучкістю коду. Для розв'язання цього завдання в роботі спроектовано комп'ютерну систему керування пінами, яка є не окремою функцією чи бібліотекою, а сукупністю взаємопов'язаних програмних підсистем, об'єднаних спільною архітектурою та єдиною точкою доступу. До складу системи входять п'ять функціонально завершених підсистем: підсистема фасаду (диспетчер пінів), підсистема типобезпеки, дві підсистеми керування (статичний та динамічний API), підсистема прямого доступу до апаратури через регістри, а також вимірювально-діагностична підсистема. Структурну схему системи та взаємодію її підсистем наведено на Рисунку 2.1.

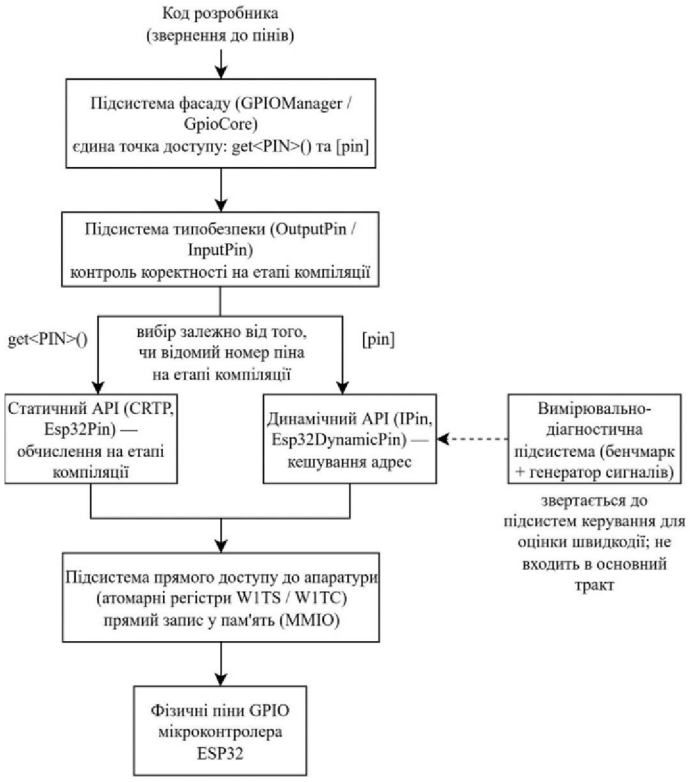


Рисунок 2.1 – Структурна схема комп'ютерної системи керування пінами ESP32.

Як видно з Рисунку 2.1, система побудована за багаторівневим (шаруватим) принципом. Запит розробника надходить через підсистему фасаду, проходить рівень типобезпеки, далі — через одну з двох підсистем керування (залежно від того, чи відомий номер піна на етапі компіляції), і завершується атомарним записом у регістри підсистеми доступу до апаратури, яка безпосередньо змінює стан фізичних пінів. Окремо функціонує вимірювально-діагностична підсистема, що взаємодіє з підсистемами керування для експериментальної оцінки швидкодії всієї системи. Така організація забезпечує одночасно високу швидкодію, безпеку типів та зручність використання, а також робить систему масштабованою для перенесення на інші мікроконтролерні архітектури.

2.1. Обґрунтування гібридної архітектури керування GPIO

Для забезпечення максимальної масштабованості архітектуру поділено на два незалежні простори: статичний та динамічний API.

Статичний API призначений для критичних за часом задач (генерація високочастотних сигналів, bit-banging), де номер піна відомий на етапі написання коду.

Динамічний API використовується для ситуацій, коли номер піна передається як змінна під час виконання програми (наприклад, при ініціалізації зовнішніх драйверів).

2.2. Парадигма Zero-Cost Abstractions як основа проектування

Традиційний підхід спирається на HAL-бібліотеки (наприклад, digitalWrite), які приймають номер піна як змінну, динамічно розраховують бітові маски та використовують умовні розгалуження, що споживає процесорні такти.

Для вирішення цієї проблеми у статичному API застосовано парадигму сучасного C++ — Абстракції з нульовою вартістю (Zero-Cost Abstractions).

Проблему runtime-обчислень масок вирішено за допомогою метапрограмування (Templates) [11].

Передача номера піна як параметра шаблону (template <uint8_t PIN>) дозволяє виконати всі обчислення на етапі компіляції (constexpr / consteval), що заощаджує такти ALU.

Крім того, проблему умовних розгалужень при виборі банку регістрів (для пінів 0-31 або 32-39) усунуто за допомогою конструкції if constexpr, [10, 15] яка дозволяє компілятору повністю видалити "неактивну" гілку коду з фінального бінарного файлу.

Принципова різниця між звичайним розгалуженням if та конструкцією if constexpr полягає в моменті прийняття рішення. Звичайний оператор if обчислює умову під час виконання програми (runtime): на кожному виклику процесор перевіряє значення, виконує порівняння й здійснює перехід (branch), витрачаючи на це процесорні такти.

Натомість if constexpr обчислюється на етапі компіляції (compile-time): оскільки номер піна передається як параметр шаблону (template <uint8_t PIN>) і є константою, відомою компілятору, рішення про вибір гілки приймається ще до генерації машинного коду.

Розглянемо це на прикладі вибору банку регістрів. Метод встановлення високого рівня містить умову if constexpr (PIN < 32), яка визначає, у який саме банк регістрів (gpio_out_w1ts для пінів 0–31 чи gpio_out1_w1ts для пінів 32–39) слід виконати запис.

Якщо об'єкт створено для піна 4, компілятор на етапі збірки обчислює умову як істинну й залишає у фінальному коді лише першу гілку, фізично видаляючи другу.

Якщо ж об'єкт створено для піна 33, у код потрапляє виключно друга гілка. У результаті згенерований машинний код не містить жодної інструкції порівняння чи переходу — лише єдину команду прямого запису в потрібний регістр.

Завдяки поєднанню шаблонів, обчислень constexpr та директиви __attribute__((always_inline)), яка примусово вбудовує тіло методу в місце виклику, абстрактний об'єктно-орієнтований виклик виду pin.setHigh() перетворюється компілятором на одну асемблерну інструкцію запису в пам'ять.

Саме це і є практичною реалізацією принципу «абстракції з нульовою вартістю»: розробник отримує зручний і безпечний інтерфейс, не сплачуючи за нього жодними додатковими витратами процесорного часу порівняно з ручним написанням низькорівневого коду.

2.3. Проектування статичного API на базі патерну CRTP

Традиційне об'єктно-орієнтоване програмування часто використовує динамічний поліморфізм, проте звернення до віртуальних таблиць (vtable) створює суттєві накладні витрати у вбудованих системах [8].

Тому для оптимізованого керування регістрами GPIO спроектовано статичне зв'язування на базі ідіоми CRTP (Curiously Recurring Template Pattern) [11].

Створено базовий шаблонний клас PinBase, який містить загальний інтерфейс керування, та апаратно-залежний клас Esp32Pin, що його успадковує.

Такий підхід зберігає інкапсуляцію ООП, але компілятор агресивно вбудовує (inline) ці методи, перетворюючи абстрактний виклик на єдину асемблерну інструкцію прямого запису в атомарні регістри GPIO_OUT_W1TS_REG та GPIO_OUT_W1TC_REG.

2.4. Проектування динамічного API через кешування адрес пам’яті

Для реалізації динамічного керування спроектовано абстрактний інтерфейс IPin з віртуальними методами.

Щоб перевершити продуктивність стандартних функцій попри наявність vtable-викликів, застосовано архітектурну оптимізацію: обчислення маски біта та

визначення абсолютних адрес регістрів керування виконується лише один раз у конструкторі об'єкта.

Під час звернення до піна в головному циклі програми відбувається лише миттєве розіменування закешованого вказівника.

Важливо зазначити, що в конструкторі кешується не окремий регістр, а повний комплект апаратних ресурсів, необхідних для всіх можливих операцій з піном. Поряд із вказівниками на регістри встановлення та скидання рівня (_w1ts_reg, _w1tc_reg) одноразово обчислюються й зберігаються вказівник на регістр читання вхідного стану (_in_reg), вказівник на регістр поточного стану виходу (_out_reg), а також пара вказівників на регістри ввімкнення та вимкнення режиму виходу (_enable_w1ts_reg, _enable_w1tc_reg).

Усі вони визначаються в межах того самого розгалуження за номером банку (піни 0–31 або 32–39), що й бітова маска, тобто без жодних додаткових витрат.

Такий підхід забезпечує, що жоден метод об'єкта під час виконання не потребує повторних обчислень адрес.

Метод читання стану read() одразу застосовує побітове «І» закешованої маски із вмістом регістра _in_reg, реалізуючи миттєве опитування входу без звернення до повільних бібліотечних функцій.

Метод ініціалізації initOutput() використовує закешований вказівник _enable_w1ts_reg для прямого ввімкнення режиму виходу, а операція перемикання toggleNonAtomic() спирається на _out_reg для визначення поточного стану піна перед його зміною.

Отже, одноразові обчислення в конструкторі повністю окуповуються: вони переносять усю адресну арифметику на етап ініціалізації, завдяки чому кожна подальша операція введення-виведення зводиться до простого розіменування готового вказівника, що й дозволяє динамічному API наближатися за швидкодією до статичного.

Слід чітко розмежувати застосування блокувань у системі. Для атомарного перемикання стану піна (метод toggle()) використовується спінлок portENTER_CRITICAL(&gpio_spinlock), оскільки ця операція є послідовністю «прочитати-змінити-записати» і потребує захисту від стану перегонів між ядрами [16]. Натомість прості операції встановлення рівня (setHigh() / setLow()) виконуються одним атомарним записом у регістри W1TS/W1TC і блокування не потребують.

Окремим свідомим архітектурним обмеженням є відсутність блокування під час першого динамічного виділення пам'яті для об'єкта піна (оператор [pin] фасаду). Це рішення ґрунтується на припущенні, що первинна ініціалізація всіх пінів відбувається в однопотоковому режимі у функції setup(), ще до запуску планувальника FreeRTOS на обох ядрах. Відмова від блокування саме на цьому етапі дозволила уникнути накладних витрат без ризику для потокобезпеки при штатному використанні системи.

2.5. Строга типізація (Type Safety) та патерн Фасад

Для усунення ризику короткого замикання (коли пін апаратно налаштований як вхід, але програма намагається подати на нього сигнал) архітектуру доповнено концепцією строгої типізації.

Розроблено спеціалізовані класи-обгортки OutputPin та InputPin. Якщо розробник спробує змінити стан вхідного піна, компілятор миттєво згенерує помилку, не допускаючи виконання небезпечного коду на пристрої.

Обидва API (статичний та динамічний) об'єднані архітектурним патерном Фасад [12, 13] у єдиний клас GPIOManager (глобальна точка доступу GpioCore).

Це дозволяє зручно звертатися до статичних пінів через метод get<PIN>(), а до динамічних — через оператор масиву [].

2.6. Взаємодія підсистем та цілісність архітектури

Ключовою ознакою, що відрізняє розроблений програмний продукт від окремої бібліотеки-обгортки, є наявність чітко розмежованих підсистем із визначеними інтерфейсами взаємодії між ними. Кожна підсистема відповідає за власний рівень абстракції та не дублює функції інших, що відповідає принципу єдиної відповідальності.

Підсистема фасаду (GPIOManager) виконує роль диспетчера: вона приховує від розробника внутрішню складність вибору між статичним і динамічним керуванням, надаючи єдиний уніфікований інтерфейс. Звернення через шаблонний метод get<PIN>() маршрутизується до статичної підсистеми, тоді як звернення через оператор індексування [pin] — до динамічної, з автоматичним створенням і кешуванням відповідного об'єкта піна.

Підсистема типобезпеки розташована між фасадом і підсистемами керування та виконує функцію контролю коректності операцій. Вона гарантує, що піни, налаштовані як входи, фізично не можуть отримати команду запису, а піни-виходи — команду, призначену лише для входів. Таким чином, ця підсистема забезпечує цілісність роботи всієї системи на рівні синтаксису мови ще до завантаження коду в мікроконтролер.

Підсистеми керування (статичний та динамічний API) є взаємозамінними з погляду фасаду, але реалізують різні стратегії: статична орієнтована на максимальну швидкодію за рахунок обчислень на етапі компіляції, динамічна — на гнучкість за рахунок одноразового кешування адрес. Обидві звертаються до спільної підсистеми доступу до апаратури, що усуває дублювання низькорівневого коду роботи з регістрами.

Вимірювально-діагностична підсистема є самостійним компонентом, який не входить до основного тракту керування, але взаємодіє з підсистемами керування для об'єктивної оцінки їхньої швидкодії. Вона забезпечує ізоляцію вимірювань від планувальника FreeRTOS та фіксацію часових характеристик, що робить систему

придатною не лише для застосування, а й для подальшого верифікаційного тестування.

Така чітка декомпозиція на підсистеми з визначеними зв'язками підтверджує, що розроблений продукт є цілісною комп'ютерною системою керування, а не ізольованим програмним модулем.

Висновки за розділом 2

У другому розділі спроектовано гібридну архітектуру керування GPIO, яка безпосередньо вирішує проблеми, виявлені в Розділі 1.

Статичний API на базі ідіоми CRTP та парадигми Zero-Cost Abstractions усуває накладні витрати часу виконання.

Динамічний API забезпечує гнучкість завдяки кешуванню адрес, а впровадження строгої типізації (Type Safety) та патерну Фасад підвищує загальну надійність вбудованого програмного забезпечення на рівні компіляції.

РОЗДІЛ 3
ПРОГРАМНА РЕАЛІЗАЦІЯ СИСТЕМИ НА C++

Цей розділ описує безпосередню програмну реалізацію гібридної архітектури, спроектованої в Розділі 2, а також структуру вимірювальних інструментів (скетчів), створених для проведення експериментальних досліджень. Весь сирцевий код бібліотеки винесено у Додаток Г.

3.1. Загальна організація програмного модуля

Програмну систему реалізовано як єдиний заголовний файл FastGPIO.h. Для забезпечення чистоти простору імен та уникнення конфліктів зі стандартними бібліотеками Arduino, код розділено на логічні простори: hal::static_api та hal::dynamic_api.

Ключовим елементом реалізації концепції Memory-Mapped I/O є визначення глобальних константних вказівників на апаратні регістри ESP32. Щоб компілятор не оптимізував звернення до апаратури, застосовано кваліфікатор volatile.

```
// Bank 0 registers (GPIOs 0-31)
inline volatile uint32_t* const gpio_out_w1ts = (volatile uint32_t*)GPIO_OUT_W1TS_REG;
inline volatile uint32_t* const gpio_out_w1tc = (volatile uint32_t*)GPIO_OUT_W1TC_REG;
```

Лістинг 3.1 – Визначення вказівників на атомарні регістри пам'яті

Використання inline змінних (стандарт C++17) , як показано в Лістингу 3.1, гарантує, що ці вказівники існують в єдиному екземплярі на всю програму.

3.2. Реалізація статичного API

Статичний інтерфейс базується на ідіомі CRTP. Базовий шаблонний клас PinBase визначає загальний інтерфейс і делегує виконання методів до похідного класу через static_cast на етапі компіляції.

Похідний клас Esp32Pin містить безпосередню апаратну реалізацію. Для усунення runtime-розгалужень (вибору банку регістрів для пінів 0-31 або 32-39) успішно застосовано конструкцію if constexpr.

```
template <uint8_t PIN>
class Esp32Pin : public PinBase<Esp32Pin<PIN>, PIN> {
public:
    static_assert(PIN < 40, "ESP32 pin number must be in range 0-39");

    __attribute__((always_inline)) void setHighImpl() const {
        if constexpr (PIN < 32) {
            *gpio_out_w1ts = (1UL << PIN);
        } else {
            *gpio_out1_w1ts = (1UL << (PIN - 32));
        }
    }
};
```

Лістинг 3.2 – Фрагмент реалізації методу встановлення високого рівня (CRTP)

Як видно з Лістингу 3.2, макрос __attribute__((always_inline)) примушує компілятор повністю відмовитися від створення функції в пам'яті, вбудовуючи асемблерні інструкції прямо в місце виклику.

Конструкція static_assert додатково захищає систему від невалідних номерів пінів ще до запуску коду.

3.3. Реалізація динамічного API

Для динамічного керування створено абстрактний клас IPin з чисто віртуальними методами.

Головна оптимізація швидкодії реалізована у конструкторі класу Esp32DynamicPin.

```
Esp32DynamicPin(uint8_t pin) : _pin(pin) {
    if (pin < 32) {
        _mask = (1UL << pin);
```

```
            _w1ts_reg = gpio_out_w1ts;
            _w1tc_reg = gpio_out_w1tc;
        } else {
            _mask = (1UL << (pin - 32));
            _w1ts_reg = gpio_out1_w1ts;
            _w1tc_reg = gpio_out1_w1tc;
        }
    }

    void setHigh() const override {
        *_w1ts_reg = _mask;
    }
```

Лістинг 3.3 – Кешування адрес регістрів у конструкторі динамічного піна

Така реалізація (Лістинг 3.3) переносить усі повільні математичні операції зі зсуву бітів та розгалуження виключно на етап створення об'єкта.

У методі setHigh() виконується лише швидке розіменування закешованого вказівника _w1ts_reg. Окрім генерації вихідних сигналів, динамічний API забезпечує високошвидкісне зчитування стану пінів.

```
    bool read() const override {
        return (*_in_reg & _mask) != 0;
    }
```

Лістинг 3.4 – Реалізація зчитування стану піна через GPIO_IN_REG

Як показано в Лістингу 3.4, зчитування вхідного сигналу виконується прямим опитуванням регістра GPIO_IN_REG (polling) за допомогою побітового «І» з попередньо обчисленою маскою.

Таке рішення дозволяє зчитувати стан повністю оминаючи механізм апаратних переривань. Це безпосередньо вирішує проблему латентності ISR (затримки від 1 до 5 мікросекунд), яка була проаналізована в підрозділі 1.4, та забезпечує миттєву і детерміновану реакцію системи на зміну вхідного сигналу.

3.4. Реалізація обгорток типобезпеки та патерну Фасад

Для забезпечення Type Safety розроблено шаблонні класи OutputPin та InputPin. Патерн Фасад реалізовано через клас GPIOManager, єдиний глобальний екземпляр якого (GpioCore) об'єднує статичний та динамічний підходи.

```
template <uint8_t PIN>
static constexpr static_api::Esp32Pin<PIN> get() {
    return static_api::Esp32Pin<PIN>();
}

dynamic_api::IPin* operator[](uint8_t pin) const {
    if (pin >= 40) return nullptr;
    if (!_dynamic_pins[pin]) {
        _dynamic_pins[pin] = new dynamic_api::Esp32DynamicPin(pin);
    }
    return _dynamic_pins[pin];
}
```

Лістинг 3.5 – Реалізація єдиної точки доступу (GPIOManager)

Метод get<PIN>() повертає легковаговий статичний об'єкт: оскільки клас Esp32Pin не містить полів даних (номер піна закодовано в параметрі шаблону), а його методи примусово вбудовуються (inline), створення такого об'єкта не спричиняє реальних витрат пам'яті чи процесорного часу під час виконання. Натомість перевантажений оператор [] використовує динамічне виділення пам'яті через new (Лістинг 3.5).

Для того, щоб обгортки типобезпеки могли безшовно працювати як зі статичними об'єктами (які передаються за значенням), так і з динамічними (які повертаються як вказівники IPin*), було застосовано спеціалізацію шаблонів.

```
template <>
class OutputPin<dynamic_api::IPin*> {
private:
    dynamic_api::IPin* _pin;
public:
    __attribute__((always_inline)) OutputPin(dynamic_api::IPin* pin) : _pin(pin) {}

    __attribute__((always_inline)) void init() {
```

```
        if (_pin) _pin->initOutput();
    }

    __attribute__((always_inline)) void setHigh() const {
        if (_pin) _pin->setHigh();
    }

    __attribute__((always_inline)) void setLow() const {
        if (_pin) _pin->setLow();
    }

    __attribute__((always_inline)) void toggle() const {
        if (_pin) _pin->toggle();
    }

    __attribute__((always_inline)) void toggleNonAtomic() const {
        if (_pin) _pin->toggleNonAtomic();
    }
};
```

Лістинг 3.6 – Спеціалізація шаблону OutputPin для динамічних вказівників

Лістинг 3.6 демонструє, що одна й та сама обгортка OutputPin завдяки шаблонній спеціалізації однаково успішно обслуговує обидва API.

При цьому для динамічних об'єктів автоматично зберігається критично важлива перевірка на нульовий вказівник (if (_pin)), що запобігає збоям під час виконання коду і яскраво підтверджує гнучкість реалізованого патерну Фасад.

3.5. Реалізація вимірювального стенду

Для експериментального підтвердження швидкодії було розроблено два спеціалізовані скетчі: test_sketch.ino (для програмного бенчмаркінгу) та scope_test.ino (генератор для осцилографа).

Щоб усунути джиттер від операційної системи FreeRTOS, у вимірювальному скетчі застосовано апаратну ізоляцію. Задача генерації імпульсів жорстко

прив'язана до другого ядра мікроконтролера за допомогою функції xTaskCreatePinnedToCore(..., 1).

Блоки перемикання пінів додатково захищені критичними секціями portENTER_CRITICAL(&test_mux), які тимчасово блокують обробку апаратних переривань.

Час виконання циклів фіксується за допомогою вбудованої функції micros().

Для скетчу-генератора scope_test.ino було застосовано потужний механізм метапрограмування — спеціалізацію шаблону GpioPinSelector.

```
template <uint8_t PIN, int MODE> struct GpioPinSelector;

// Спеціалізація для FastGPIO Static (MODE = 1)
template <uint8_t PIN>
struct GpioPinSelector<PIN, 1> {
    hal::OutputPin<hal::static_api::Esp32Pin<PIN>> pin;
    __attribute__((always_inline)) void setHigh() const { pin.setHigh(); }
};
```

Лістинг 3.7 Шаблонний селектор методів генерації для осцилографа

Такий підхід (Лістинг 3.7) дозволив через просту зміну макросу конфігурації (наприклад, #define GPIO_MODE_A 1) призначати різним пінам різні API на етапі компіляції, що забезпечило гнучкість налаштування стенду без необхідності дублювання коду генератора.

Висновки за розділом 3

У третьому розділі продемонстровано програмну реалізацію розробленої гібридної архітектури. Наведено конкретні фрагменти C++ коду, які підтверджують використання ідіоми CRTP, метапрограмування (if constexpr) та кешування адрес пам'яті. Описано програмну структуру вимірювального стенду, який базується на ізоляції процесів від планувальника FreeRTOS, що забезпечує необхідні умови для переходу до експериментального тестування систем.

РОЗДІЛ 4
ЕКСПЕРИМЕНТАЛЬНЕ ДОСЛІДЖЕННЯ ШВИДКОДІЇ

Цей розділ присвячено експериментальному доведенню ефективності розробленої гібридної архітектури керування GPIO. Вимірювання спрямовані на кількісне підтвердження заявленого зменшення апаратних і програмних затримок порівняно зі стандартними високорівневими бібліотеками.

4.1. Методика проведення експерименту

Вимірювання проводилися на відлагоджувальній платі з мікроконтролером ESP32, що працює на базовій тактовій частоті ядра 240 МГц (тривалість одного такту становить ~4,16 нс).

Для проведення експерименту було налаштовано три окремі цифрові піни мікроконтролера на вивід: пін 32 ініціалізувався стандартною функцією pinMode(), пін 4 — через розроблений Статичний API, а пін 33 — через Динамічний API.

Вибір конкретних номерів пінів для тестування не є випадковим, а зумовлений необхідністю забезпечити коректність і репрезентативність порівняння.

Насамперед, для кожного з трьох методів виділено окремий фізичний пін, що унеможливлює взаємний вплив тестів один на одного та виключає спотворення результатів через залишковий стан спільного виводу.

Окрім того, розподіл пінів між тестами враховує апаратний поділ каналів GPIO на два банки регістрів.

Пін 4, задіяний для Статичного API, належить до першого банку (піни 0–31) і керується через регістри GPIO_OUT_W1TS_REG / GPIO_OUT_W1TC_REG.

Натомість пін 32 (стандартний метод) та пін 33 (Динамічний API) належать до другого банку (піни 32–39), що обслуговується регістрами GPIO_OUT1_W1TS_REG / GPIO_OUT1_W1TC_REG.

Таким чином, експеримент охоплює обидва банки регістрів, що додатково підтверджує універсальність розробленої системи та коректність реалізованого в ній механізму автоматичного вибору банку залежно від номера піна.

Слід також зауважити, що абсолютний номер піна не впливає на швидкодію операції перемикання, оскільки після обчислення маски та вибору банку (на етапі компіляції для статичного API або в конструкторі для динамічного) усі методи зводяться до однакової за вартістю операції прямого запису в регістр.

Це дозволяє коректно порівнювати методи між собою незалежно від того, до якого банку належить конкретний тестовий пін.

Процедура тестування базувалася на генерації сигналів шляхом виконання циклу із 100 000 ітерацій для кожного методу (що дорівнює 200 000 перемикань станів з низького на високий і навпаки).

Тривалість виконання фіксувалася апаратним таймером мікроконтролера за допомогою функції micros().

Час одного перемикання розраховувався шляхом ділення загального часу на загальну кількість змін логічного рівня, що дозволило нівелювати мікропохибки та отримати високоточний усереднений результат.

Застосована методика усереднення забезпечує високу достовірність отриманих значень. Окреме перемикання піна триває одиниці-десятки наносекунд, тоді як роздільна здатність системного таймера, доступного через функцію micros(), становить лише одну мікросекунду. Безпосереднє вимірювання тривалості одного перемикання таким таймером було б неможливим, оскільки похибка дискретизації (±1 мкс) у десятки разів перевищувала б сам вимірюваний інтервал.

Для усунення цієї проблеми застосовано вимірювання сумарного часу великого пакета операцій: фіксується тривалість виконання 100 000 ітерацій циклу, що відповідає 200 000 окремих перемикань логічного рівня. За такого підходу постійна похибка таймера (одиниці мікросекунд) розподіляється на сотні тисяч

операцій і стає статистично незначущою — її внесок у середнє значення одного перемикання зменшується пропорційно кількості ітерацій і складає частки відсотка.

Другою умовою чистоти експерименту є усунення впливу операційної системи на результат вимірювання. Оскільки тестова задача виконується під керуванням FreeRTOS на багатоядерному процесорі, у будь-який момент її виконання могло б перерватися системним тіком планувальника, обробкою апаратного переривання або перемиканням контексту на інше завдання.

Будь-яка з цих подій додала б до виміряного часу сторонні мікросекундні затримки, спотворивши результат. Щоб цього уникнути, кожен вимірюваний цикл обгорнуто критичною секцією portENTER_CRITICAL(&test_mux) / portEXIT_CRITICAL(&test_mux), яка на час виконання тимчасово блокує обробку переривань на відповідному ядрі.

Таким чином, протягом усього вимірювання процесор виконує виключно цільовий код перемикання пінів, не відволікаючись на службові операції, що гарантує відтворюваність і достовірність зафіксованих часових характеристик.

Важливо розмежувати ролі вимірювальних інструментів. Двоканальний осцилограф використовувався виключно для якісного візуального підтвердження форми сигналу (відсутність завалів фронтів при використанні прямого доступу) [14].

Через технічні обмеження наявного осцилографа (смуга пропускання) візуалізувати граничну частоту 10 МГц безпосередньо на екрані приладу неможливо.

Тому всі наведені нижче осцилограми є ілюстративними (знятими на знижених частотах), а кількісні висновки щодо мікросекундних затримок спираються виключно на високоточний програмний бенчмарк.

4.2. Тест 1. Базова лінія: стандартний digitalWrite()

Перший етап випробувань передбачав вимірювання латентності стандартного HAL середовища Arduino за допомогою функцій digitalWrite(pin, HIGH) та digitalWrite(pin, LOW).

Результати вимірювання зафіксували наступне: загальний час виконання 100 000 ітерацій склав 116 295 мікросекунд.

Середня тривалість одного перемикання склала 581,47 наносекунди.

У перерахунку на частоту 240 МГц мікроконтролер витрачає близько 140 машинних тактів, а розрахункова максимальна частота генерації меандру склала ~0,86 МГц.

Така розбіжність із теоретичними показниками (58-80 тактів) пояснюється тим, що реальний експеримент фіксує повне виконання функції digitalWrite() у ядрі Arduino (перевірка валідності піна, трансляція логічного номера у фізичний індекс) та накладні витрати на організацію циклу перемикання.

4.3. Тест 2. Статичний API (CRTP)

Другий етап був спрямований на дослідження швидкодії розробленого класу Esp32Pin, який реалізує парадигму абстракцій з нульовою вартістю (Zero-Cost Abstractions).

Результати виявили кардинальне поліпшення характеристик: загальний час виконання тесту склав лише 10 002 мікросекунди.

Середній час одного перемикання знизився до 50,01 наносекунди (лише 12 тактів процесора).

Розрахункова частота генерації сигналу досягла 10 МГц.

Оскільки обчислення бітової маски порту та вибір банку регістрів відбулося ще на етапі компіляції, ці операції взагалі не споживали процесорного часу, а компілятор напряму згенерував асемблерну інструкцію запису в атомарні регістри.

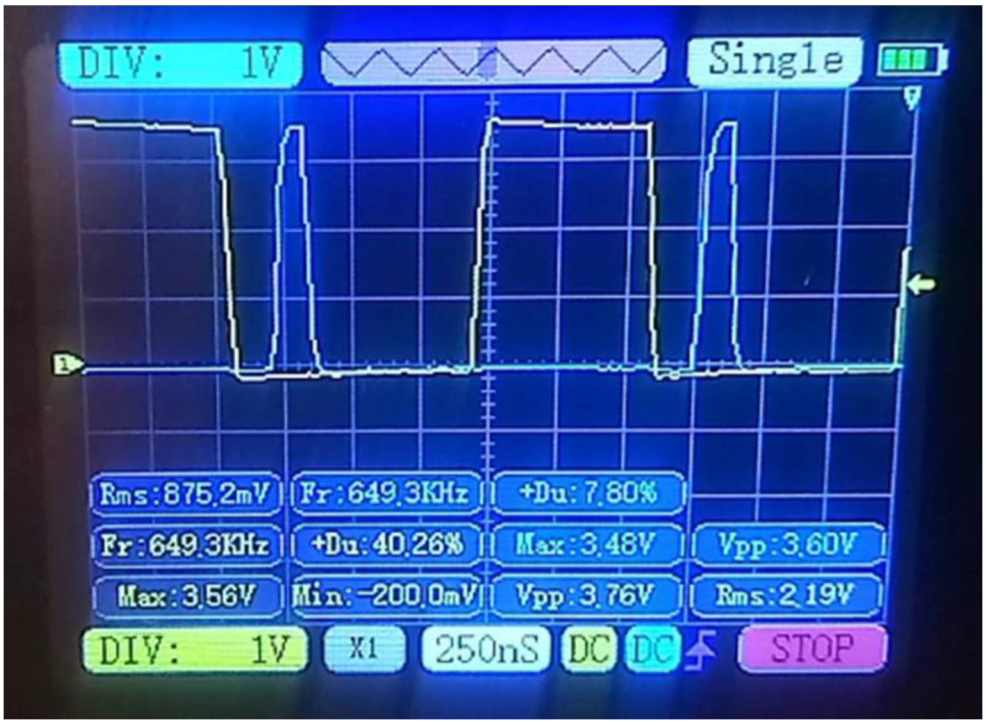


Рисунок 4.1 – Порівняння форм сигналів: жовтий канал — стандартна функція digitalWrite(), блакитний канал — Статичний API CRTP.

На Рисунку 4.1 візуально ілюструється різниця: блакитний канал формує значно вужчий імпульс, що відповідає швидшому перемиканню.

4.4. Тест 3. Динамічний API

Третій етап аналізував продуктивність Динамічного API, який використовує механізм віртуальних функцій для випадків, коли номер піна стає відомим лише під час виконання програми.

Загальний час виконання склав 17 923 мікросекунди.

Середня тривалість одного перемикання — 89,61 наносекунди (близько 21 такту процесора).

Розрахункова частота сигналу склала ~5,58 МГц.



Рисунок 4.2 – Порівняння форм сигналів: жовтий канал — стандартна функція digitalWrite(), блакитний канал — Динамічний API.

Хоча використання vtable додає накладні витрати порівняно зі статичним підходом, завдяки архітектурній оптимізації (одноразовому обчисленню та кешуванню фізичних вказівників у конструкторі) Динамічний API показав відмінний результат, що візуально підтверджується звуженням імпульсу на Рисунку 4.2.

4.5. Випробування механізму типобезпеки

Окрім вимірювання швидкодії, була проведена практична перевірка надійності архітектури шляхом тестування механізму строгої типізації (Type Safety).

Методика перевірки полягала у створенні об'єкта InputPin та спробі викликати з нього метод встановлення високого рівня setHigh().

Оскільки клас InputPin інкапсулює виключно метод зчитування, компілятор миттєво зупинив збірку проекту з помилкою (Compile-time error).

Це випробування доводить, що розроблена програмна система надійно захищає обладнання від коротких замикань на рівні синтаксису C++, виключаючи можливість виконання небезпечного коду на пристрої.

4.6. Зведений порівняльний аналіз та оцінка достовірності

Якщо Рисунки 4.1 та 4.2 порівнювали кожен із розроблених інтерфейсів зі стандартною функцією digitalWrite() окремо, то на Рисунку 4.3 наведено пряме порівняння двох розроблених підходів між собою — Статичного та Динамічного API. Жовтий канал відповідає Статичному API (CRTP), блакитний — Динамічному. Видно, що обидва канали формують значно вужчі імпульси порівняно зі стандартним методом, проте імпульс Статичного API є дещо коротшим за імпульс Динамічного, що візуально підтверджує вищу швидкодію статичного підходу через відсутність накладних витрат на звернення до таблиці віртуальних функцій (vtable). Ця якісна відмінність повністю узгоджується з кількісними показниками, зведеними далі в Таблиці 4.1.

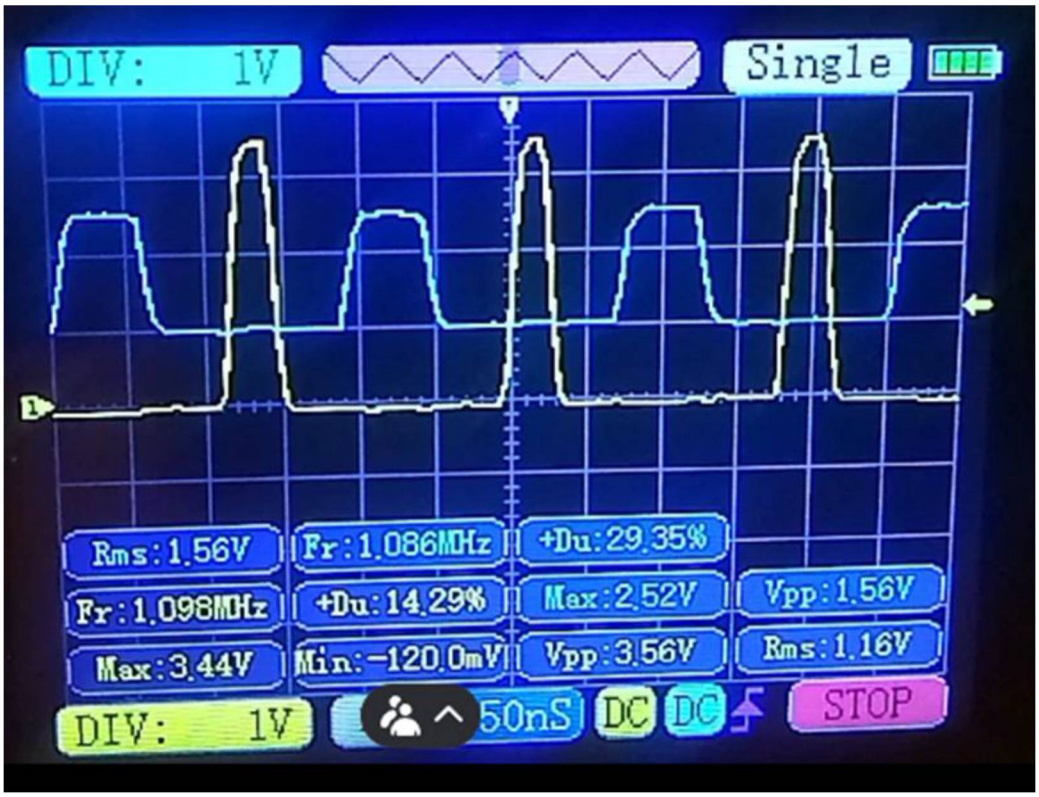


Рисунок 4.3 – Порівняння форм сигналів: жовтий канал — Статичний API CRTP, блакитний канал — Динамічний API.

Для наочного співставлення отриманих емпіричних даних результати всіх трьох тестів було зведено у загальну Таблицю 4.1.

Таблиця 4.1

Порівняльний аналіз швидкодії методів керування GPIO на мікроконтролері ESP32

Метод керування	Середня тривалість одного перемикання, нс	Витрати процесорного часу, тактів	Розрахункова частота генерації меандру, МГц
Стандартний (digitalWrite)	581,47	~140	0,86

Динамічний API (Віртуальний поліморфізм)	89,61	~ 21	5,58
Статичний API (CRTP шаблони)	50,01	12	10,00

Емпірична оцінка продуктивності повністю підтверджує ефективність розробленої гібридної архітектури.

Розроблений Статичний API перевершив за швидкістю стандартну функцію digitalWrite() у 11,63 раза, а динамічний API з кешуванням масок — у 6,49 раза.

```
Configuring pins: Standard=32, Static=4, Dynamic=33
Starting measurements (100,000 write cycles per test)...

================ BENCHMARK RESULTS ================
1. Standard digitalWrite()  : 116295 us (581.475000 ns per toggle)
2. FastGPIO Static (CRTP)   : 10002 us (50.010000 ns per toggle)
3. FastGPIO Dynamic (Virtual): 17923 us (89.615000 ns per toggle)
---------------------------------------------------
Static API Speedup : 11.63 times faster than standard
Dynamic API Speedup: 6.49 times faster than standard
===================================================

Generating fast square wave on Static Pin  (PIN_4) in bursts of 100k toggles.
Interrupts on Core 1 are disabled during each burst to prevent jitter.
```

Рисунок 4.4 – Результати виконання програмного бенчмарку.

Для документального підтвердження достовірності вимірювань на Рисунку 4.4 наведено скриншот, отриманий під час виконання тестового скетча.

Значення повністю відповідають даним, зведеним у Таблицю 4.1.

Таким чином, головну мету Технічного завдання щодо оптимізації програмного керування підсистемою GPIO та досягнення максимальної швидкодії (цільові 10 МГц) успішно виконано.

4.7. Область практичного застосування

Досягнута швидкодія прямого керування пінами (час перемикання 50,01 нс та стабільна частота генерації до 10 МГц) відкриває низку практичних застосувань, які є недосяжними для стандартних високорівневих бібліотек. Найбільш показовими сценаріями використання є:

- По-перше, програмна емуляція високошвидкісних послідовних протоколів (bit-banging). Зокрема, керування адресними світлодіодами типу WS2812 («розумними» RGB-стрічками) вимагає формування імпульсів із жорстким часовим бюджетом порядку сотень наносекунд на біт; стандартна функція digitalWrite() із її затримкою 581 нс фізично не здатна забезпечити такі таймінги, тоді як розроблений Статичний API має достатній запас швидкодії.
- По-друге, реалізація паралельних інтерфейсів дисплеїв, де потрібно одночасно й синхронно перемикати групу ліній даних з мінімальним перекосом (skew) між ними. Прямий атомарний запис у регістр гарантує детермінованість, недоступну при послідовних викликах HAL-функцій.
- По-третє, реалізація нестандартних варіантів шин I2C та SPI з нетиповими частотами або кількістю ліній, коли апаратні периферійні контролери мікроконтролера не підтримують необхідну конфігурацію.
- По-четверте, генерація точних синхронізаційних та тактових сигналів у системах автоматизації реального часу, де нестабільність фази (джитер) є критичною. Розроблений механізм ізоляції від планувальника FreeRTOS у поєднанні з прямим доступом до регістрів дозволяє формувати сигнали з мінімальним джитером, що було підтверджено експериментально під час генерації безперервного меандру частотою 10 МГц.

Таким чином, розроблена система є універсальним інструментом для широкого класу задач вбудованих систем та промислової автоматизації, що потребують детермінованого високошвидкісного керування підсистемою вводу-виводу.

Висновки за розділом 4

У четвертому розділі проведено експериментальне тестування розробленої гібридної архітектури. Доведено, що застосована методика програмного бенчмаркінгу з ізоляцією від FreeRTOS дозволяє отримувати достовірні результати. Експериментально підтверджено, що Статичний API зменшує час перемикання піна до 50,01 нс (у 11,63 раза швидше за стандартні бібліотеки), а Динамічний API пришвидшує роботу у 6,49 раза.

Додатково підтверджено ефективність механізму строгої типізації для забезпечення безпеки вбудованого коду.

ВИСНОВКИ

У кваліфікаційній роботі вирішено актуальне науково-прикладне завдання щодо розробки та експериментального дослідження оптимізованої програмної системи керування підсистемою вводу-виводу мікроконтролера ESP32. За результатами виконання роботи зроблено такі висновки:

1. На основі аналізу технічної документації та архітектурних особливостей мікроконтролера ESP32 встановлено, що використання стандартних високорівневих бібліотек (HAL Arduino та ESP-IDF) є неефективним для критичних за часом задач. Проходження сигналу через матрицю маршрутизації (GPIO Matrix) та втручання планувальника операційної системи FreeRTOS вносять непередбачувані затримки. Обґрунтовано необхідність прямого керування периферією через атомарні регістри пам'яті (GPIO_OUT_W1TS_REG, GPIO_OUT_W1TC_REG).

2. Спроектовано гібридну програмну архітектуру (Hardware Abstraction Layer) на базі мови C++, яка поєднує поліморфізм часу компіляції та динамічний поліморфізм. Для екстремально швидких операцій розроблено Статичний API з використанням патерну CRTP (Curiously Recurring Template Pattern), що забезпечує «абстракції з нульовою вартістю». Для задач, що вимагають гнучкості, розроблено Динамічний API з механізмом одноразового кешування адрес регістрів у конструкторі.

3. Впроваджено парадигму строгої типізації (Type Safety) за допомогою розробки спеціалізованих класів-обгорток InputPin та OutputPin. Доведено, що такий підхід повністю усуває ризик виникнення апаратних конфліктів (коротких замикань) ще на етапі компіляції програми, підвищуючи загальну надійність вбудованої системи.

4. Експериментально доведено, що розроблений Статичний API на базі патерну CRTP перевершує стандартну функцію digitalWrite() за швидкодією у 11,63

раза — час одного перемикання знизився з 581,47 нс до 50,01 нс (12 тактів процесора), а максимальна частота генерації сигналу досягла 10 МГц. Динамічний API з кешуванням вказівників показав прискорення у 6,49 раза (89,61 нс, ~5,58 МГц), що робить його оптимальним рішенням для універсальних драйверів периферії. Таким чином, вимоги Технічного завдання щодо мінімізації апаратних і програмних затримок виконано в повному обсязі.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. ESP32 Technical Reference Manual [Електронний ресурс] / Espressif Systems. – 2024. – URL: https://www.espressif.com/sites/default/files/documentation/esp32_technical_reference_manual_en.pdf (дата звернення: 25.05.2026).
2. ESP-IDF Programming Guide [Електронний ресурс] / Espressif Systems. – 2024. – URL: https://docs.espressif.com/projects/esp-idf/en/latest/esp32 (дата звернення: 25.05.2026).
3. ESP32 Series Datasheet [Електронний ресурс] / Espressif Systems. – 2024. – URL:https://www.espressif.com/sites/default/files/documentation/esp32_datasheet_en.pdf (дата звернення: 25.05.2026).
4. ESP32 Hardware Design Guidelines. Version [Електронний ресурс] / Espressif Systems. – 2023. – URL: https://docs.espressif.com/projects/esp-hardware-design-guidelines/en/latest/esp32/index.html (дата звернення: 25.05.2026).
5. Arduino Core for ESP32. Official Documentation [Електронний ресурс] / Espressif Systems. – 2024. – URL: https://docs.espressif.com/projects/arduino-esp32/ (дата звернення: 25.05.2026).
6. FreeRTOS Kernel Documentation [Електронний ресурс] / Amazon Web Services. – 2024. – URL: https://www.freertos.org/Documentation/02-Kernel/07-Books-and-manual/01-RTOS_book (дата звернення: 27.05.2026).
7. Barry R. Mastering the FreeRTOS Real Time Kernel. Bristol : Real Time Engineers Ltd., 2016. 396 p.
8. Kormanyos C. Real-Time C++: Efficient Object-Oriented and Template Microcontroller Programming. 4th ed. Berlin : Springer, 2021. 437 p.
9. Meyers S. Effective Modern C++. 42 Specific Ways to Improve Your Use of C++11 and C++14. Sebastopol : O'Reilly Media, 2014. 334 p.

10. Stroustrup B. The C++ Programming Language. 4th ed. Boston : Addison-Wesley, 2013. 1368 p.
11. Vandevoorde D., Josuttis N. M., Gregor D. C++ Templates: The Complete Guide. 2nd ed. Boston : Addison-Wesley, 2017. 832 p.
12. Douglass B. P. Design Patterns for Embedded Systems in C. Oxford : Newnes, 2010. 464 p.
13. White E. Making Embedded Systems: Design Patterns for Great Software. Sebastopol : O'Reilly Media, 2011. 330 p.
14. Johnson H. W., Graham M. High-Speed Digital Design: A Handbook of Black Magic. Englewood Cliffs : Prentice Hall, 1993. 447 p.
15. ISO/IEC 14882:2020. Programming languages — C++. Geneva : International Organization for Standardization, 2020. 1853 p.
16. Williams A. C++ Concurrency in Action. 2nd ed. Shelter Island : Manning Publications, 2019. 592 p.

ДОДАТКИ

Додаток А

Міністерство освіти і науки України
Харківський національний університет імені В. Н. Каразіна
Навчально-науковий інститут комп'ютерних наук та штучного інтелекту
Кафедра комп'ютерних систем та робототехніки

ЗАТВЕРДЖУЮ
Завідувач кафедри комп'ютерних
систем та робототехніки
Максим ХРУСЛОВ
_____
(підпис)
«13» жовтня 2025року

З А В Д А Н Н Я
НА КВАЛІФІКАЦІЙНУ РОБОТУ

Здобувача першого (бакалаврського) рівня вищої освіти
(першого (бакалаврського) / другого (магістерського))

СОЛЯНИКА ЮРІЯ ВЯЧЕСЛАВОВИЧА
(прізвище, ім'я, по батькові студента)

Спеціальність: 151 – Автоматизація та комп'ютерно-інтегровані технології
Освітня програма «Автоматизація та комп'ютерно інтегровані технології»
1. Тема роботи **КОМП'ЮТЕРНА СИСТЕМА УПРАВЛІННЯ ПІНАМИ МІКРОКОНТРОЛЕРА ESP32 ЗА ДОПОМОГОЮ РЕГІСТРІВ.**
керівник роботи **Котвицький Альберт Тадеушевич, доцент,**
(прізвище, ім'я, по батькові, наукова ступінь, вчене звання)
затверджені наказом по університету від ***№ 4101-5/1245 від 28.05.2026 року.***
2. Строк подання студентом роботи ***30 травня 2026року***
3. Вихідні дані до роботи:
   1) Офіційна технічна документація мікроконтролера ESP32 (Datasheet, Technical Reference Manual).
   2) Відлагоджувальна плата на базі ESP32 та базові компоненти для макетування.
   3) Програмний інструментарій для системного програмування (інструментів Arduino IDE).
   4) Апаратні таймери мікроконтролера (функція micros()) для точного програмного бенчмаркінгу та вимірювальне обладнання (осцилограф) для візуальної фіксації сигналів.
   5) Науково-технічна література щодо прямого доступу до апаратних регістрів та обходу HAL.
4. Перелік питань, які потрібно розробити
   1) Аналіз існуючих методів керування портами мікроконтролерів (стандартні бібліотеки проти прямого доступу) та визначення їхніх недоліків для задач автоматизації.
   2) Дослідження архітектурних особливостей мікроконтролера ESP32 в контексті роботи підсистеми введення-виведення (GPIO Matrix, IO_MUX).
   3) Розробка програмної моделі для прямого керування пінами через регістри з метою мінімізації затримок.
   4) Експериментальне дослідження розробленої моделі та оцінка її ефективності.

4. План роботи

№ з/п	Назви етапів роботи	Термін виконання етапів роботи
1	Затвердження теми роботи та узгодження плану	01.10.2025-13.10.2025
2	Аналіз існуючих методів керування портами мікроконтролерів та визначення їхніх недоліків	14.10.2025-30.11.2025
3	Аналіз апаратної частини ESP32 для роботи з портами	01.12.2025-31.01.2026
4	Розробка програмної моделі та її експериментальне дослідження.	01.02.2026-31.03.2026
5	Підготовка і оформлення звітних матеріалів та додатків кваліфікаційної роботи. Оформлення списку літератури	01.04.2026-30.04.2026
6	Оформлення пояснювальної записки кваліфікаційної роботи відповідно до вимог	01.05.2026-30.05.2026
7	Оформлення звіту про переддипломну практику	01.05.2026-30.05.2026
8	Представлення кваліфікаційної роботи керівнику та рецензенту	30.05.2026

5. Дата видачі завдання ***13 жовтня 2025 року.***

Здобувач вищої освіти _____ Юрій СОЛЯНИК
(підпис)

Керівник роботи _____ Альберт КОТВИЦЬКИЙ
(підпис)

Додаток Б

Затверджую

_____

«_____»_____2026 р.

Технічне завдання
на розробку програмного виробу
«Комп’ютерна система управління пінами мікроконтролера ESP32
за допомогою регістрів»

1.	Введення	1.1. Назва програмного виробу – Комп’ютерна система управління пінами мікроконтролера ESP32 за допомогою регістрів» 1.2. Галузь застосування – Інформаційні технології: автоматизація, вбудовані системи, програмування мікроконтролерів.
2.	Підстава для розробки	2.1. Навчальний план за спеціальністю 151 Автоматизація та комп'ютерно-інтегровані технології. 2.2. Завдання на кваліфікаційну роботу бакалавра № 4101-5/1245 від 28.05.2026р. (представити як Додаток А до пояснювальної записки до кваліфікаційної роботи).
3.	Призначення розробки	3.1. Мета розробки: розробка програмної системи для прямого керування пінами через регістри з метою мінімізації затримок. 3.2. Призначення розробки: полягає у створенні ефективного інструменту для оптимізації швидкодії роботи підсистеми введення-виведення (GPIO Matrix, IO_MUX) ESP32 для виконання критичних за часом задач автоматизації. 3.3. Початкові дані для розробки: офіційна документація та архітектурні специфікації мікроконтролера ESP32, технічні вимоги до підсистем введення-виведення.

4.	Технічні вимоги до програмного виробу	4.1. Вимоги до функціональних характеристик: 1) Ініціалізація портів вводу/виводу через пряме звернення до регістрів. 2) Зчитування та запис стану пінів з мінімальними апаратними затримками. 3) Обхід стандартних бібліотек високого рівня для підвищення продуктивності. 4.2. Вимоги до надійності: Забезпечення стабільної роботи систем при високій частоті опитування та перемикання пінів. 4.3. Вимоги до умов експлуатації: немає 4.4. Вимоги до складу і параметрів технічних засобів: Мікроконтролер ESP32, персональний комп'ютер для розробки, осцилограф або логічний аналізатор для замірів. 4.5. Вимоги до інформаційної та програмної сумісності: Сумісність із середовищем розробки ESP-IDF або Arduino IDE (на рівні мов C/C++). 4.6. Вимоги до маркування та упаковки: відсутні. 4.7. Вимоги до транспортування і зберігання: відсутні. 4.8. Спеціальні вимоги: відсутні.
5.	Вимоги до програмної документації	Програмною документацією до виробу вважати: 1) Справжнє Технічне завдання на розробку (Додаток Б). 2) Програму і методику випробувань (Додаток В). 3) Опис розробленої системи (у відповідному розділі пояснювальної записки).
6.	Вимоги до техніко-економічних показників	В даному розділі можуть бути представлені: 1) Справжнє Технічне завдання на розробку програмного виробу. 2) Методика оцінки ефективності розробленої системи. 3) Опис програмної реалізації виробу.

<table>
<tr>
<td>7.</td>
<td>Стадії і етапи розробки</td>
<td>
<table>
<tr><th>Дата</th><th>Назва етапу</th></tr>
<tr><td>01.10.2025 -30.11.2025</td><td>Аналіз існуючих методів керування портами мікроконтролерів та визначення їхніх недоліків</td></tr>
<tr><td>01.12.2025 - 31.01.2026</td><td>Аналіз апаратної частини ESP32 (GPIO Matrix, IO_MUX)</td></tr>
<tr><td>01.02.2026 - 31.03.2026</td><td>Розробка комп'ютерної системи та її експериментальне дослідження.</td></tr>
<tr><td>01.04.2026 - 30.05.2026</td><td>Підготовка, тестування та оформлення звітних матеріалів</td></tr>
</table>
</td>
</tr>
<tr>
<td>8.</td>
<td>Порядок контролю і приймання програмного продукту</td>
<td>1) Перевірку ходу розробки виконувати раз в 3 тижні.
2) Випробування провести відповідно до програми та методики випробувань із застосуванням апаратних засобів вимірювання (осцилограф/аналізатор).
3) Захист роботи провести на засіданні Екзаменаційної комісії.
4) Пояснювальну записку подати в електронному та друкованому вигляді.</td>
</tr>
</table>

Виконавець
студент групи КУ-41
СОЛЯНИК Ю.В.

Замовник:
керівник роботи,.доц
КОТВИЦЬКИЙ А.Т.

Додаток В

Програма і методика випробувань програмного виробу

«Комп’ютерна система управління пінами мікроконтролера ESP32 за допомогою регістрів»

1. Об'єкт випробувань

1. Назва програмного виробу: «Комп’ютерна система управління пінами мікроконтролера ESP32 за допомогою регістрів».
2. Галузь застосування: Інформаційні технології: автоматизація, вбудовані системи, програмування мікроконтролерів.
3. Перераховані відомості запозичуються з відповідних розділів Технічного завдання.

2. Мета випробувань

Перевірка відповідності функціональності та швидкодії розробленої програмної C++ системи керування пінами через регістри заявленим вимогам у технічному завданні (Додаток Б) , а також експериментальне підтвердження мінімізації апаратних та програмних затримок порівняно зі стандартними високорівневими бібліотеками (digitalWrite).

3. Загальні положення

Підстави для проведення випробувань: наказ про призначення екзаменаційної комісії.

Місце і тривалість випробувань: Приймальні випробування проводяться на базі лабораторії/комп'ютерного класу кафедри комп'ютерних систем та робототехніки в період роботи комісії.

Обсяг випробувань: Приймальні випробування проводяться в обсязі, відповідному цій програмі і методиці.

Організації, які беруть участь у випробуваннях: Приймальні випробування проводяться екзаменаційною комісією за участю Керівника (доц. Котвицького А. Т.), Виконавця (студента Соляника Ю. В.) та інших осіб, присутніх на засіданні.

4. Вимоги до програми або програмного виробу

Система повинна задовольняти наступним вимогам:

1. Стабільна робота на базі мікроконтролера ESP32 під керуванням операційної системи FreeRTOS.

2. Повна сумісність із середовищем розробки Arduino IDE / ESP-IDF на рівні мов C/C++.
3. Забезпечення стабільної роботи при тривалих і високочастотних циклах перемикання станів пінів (toggling).
4. Експериментально підтверджена вища швидкодія (мінімізація затримок) розробленого Статичного та Динамічного API порівняно зі стандартними HAL-функціями.
5. Забезпечення захисту від коротких замикань на рівні синтаксису мови за рахунок строгої типізації (Type Safety) та розподілу на класи InputPin і OutputPin.

5. Вимоги до програмної документації

1. Програмною документацією до виробу вважати: Справжнє
2. Технічне завдання на розробку програми (Додаток Б).
3. Програму і методику випробувань розробленого виробу (Додаток В).
4. Розділи пояснювальної записки з архітектурним описом розробленої системи та результатами експериментальних досліджень (Розділ 2).
5. Повний лістинг розробленого C++ коду системи (Додаток Г).

6. Засоби і порядок випробувань

6.1 Засоби випробувань

Для проведення випробувань необхідні:

1. Персональний комп'ютер інженера-розробника із встановленим середовищем розробки;
2. Відлагоджувальна плата на базі мікроконтролера ESP32 (тактова частота ядра 240 МГц);
3. Вимірювальне обладнання (двоканальний осцилограф) для візуального контролю та використання апаратних таймерів ESP32 для програмної фіксації наносекундних часових затримок.

6.2 Порядок проведення випробувань

Випробування проводяться в два етапи:

- Ознайомчий (1-й етап): Перевірка комплектності програмної документації згідно з ТЗ, а також наявності необхідного апаратного та лабораторного забезпечення.
- Випробування програмного виробу (2-й етап): Безпосередній запуск бенчмарку та апаратне вимірювання затримок.

Методика проведення перевірок на 2-му етапі випробувань:

1. Програма компілюється та завантажується в мікроконтролер ESP32. Конфігурація пінів виконується через глобальний екземпляр GpioCore.
2. Запуск тестового скетчу виконує генерацію сигналів шляхом виконання 100 000 ітерацій (200 000 перемикань станів) для трьох досліджуваних методів. Для забезпечення чистоти експерименту планувальник FreeRTOS ізолюється макросами заборони переривань portENTER_CRITICAL.
3. Тест 1 (Базовий): Вимірювання латентності стандартного HAL Arduino за допомогою функцій digitalWrite(pin, HIGH) та digitalWrite(pin, LOW).
4. Тест 2 (Оптимізований, Статичний API): Вимірювання часу зміни стану піна за допомогою розробленого шаблонного класу Esp32Pin на базі патерну CRTP через прямий запис в атомарні регістри GPIO_OUT_W1TS_REG та GPIO_OUT_W1TC_REG.
5. Тест 3 (Динамічний API): Аналіз продуктивності динамічного керування з чисто віртуальними функціями інтерфейсу IPin та механізмом одноразового кешування адрес регістрів у конструкторі класу Esp32DynamicPin.
6. Порівняння та фіксація отриманих часових показників за допомогою програмного бенчмарку (використання функції micros()).

Виконавець: студент групи КУ-41, Ю. В. Соляник 

Додаток Г

Лістинг Г.1 — FastGPIO

```
#ifndef FAST_GPIO_H
#define FAST_GPIO_H

#include <Arduino.h>
#include <soc/gpio_reg.h>
#include <freertos/FreeRTOS.h>
#include <freertos/task.h>
#include <utility>

namespace hal {

    // Global/static spinlock for atomic toggle operation across cores
    inline portMUX_TYPE gpio_spinlock = portMUX_INITIALIZER_UNLOCKED;

    // Register pointers for faster memory-mapped I/O (MMIO)
    // Bank 0 registers (GPIOs 0-31)
    inline volatile uint32_t* const gpio_out = (volatile uint32_t*)GPIO_OUT_REG;
    inline volatile uint32_t* const gpio_out_w1ts = (volatile uint32_t*)GPIO_OUT_W1TS_REG;
    inline volatile uint32_t* const gpio_out_w1tc = (volatile uint32_t*)GPIO_OUT_W1TC_REG;
    inline volatile uint32_t* const gpio_in = (volatile uint32_t*)GPIO_IN_REG;
    inline volatile uint32_t* const gpio_enable_w1ts = (volatile uint32_t*)GPIO_ENABLE_W1TS_REG;
    inline volatile uint32_t* const gpio_enable_w1tc = (volatile uint32_t*)GPIO_ENABLE_W1TC_REG;

    // Bank 1 registers (GPIOs 32-39)
    inline volatile uint32_t* const gpio_out1 = (volatile uint32_t*)GPIO_OUT1_REG;
    inline volatile uint32_t* const gpio_out1_w1ts = (volatile uint32_t*)GPIO_OUT1_W1TS_REG;
    inline volatile uint32_t* const gpio_out1_w1tc = (volatile uint32_t*)GPIO_OUT1_W1TC_REG;
    inline volatile uint32_t* const gpio_in1 = (volatile uint32_t*)GPIO_IN1_REG;
    inline volatile uint32_t* const gpio_enable1_w1ts = (volatile uint32_t*)GPIO_ENABLE1_W1TS_REG;
    inline volatile uint32_t* const gpio_enable1_w1tc = (volatile uint32_t*)GPIO_ENABLE1_W1TC_REG;

    namespace static_api {

        // Base class utilizing the Curiously Recurring Template Pattern (CRTP)
        template <typename Derived, uint8_t PIN>
        class PinBase {
        public:
            __attribute__((always_inline)) void setHigh() const {
                static_cast<const Derived*>(this)->setHighImpl();
            }

            __attribute__((always_inline)) void setLow() const {
                static_cast<const Derived*>(this)->setLowImpl();
            }

            __attribute__((always_inline)) void toggle() const {
```

```
        static_cast<const Derived*>(this)->toggleImpl();
    }

    __attribute__((always_inline)) void toggleNonAtomic() const {
        static_cast<const Derived*>(this)->toggleNonAtomicImpl();
    }

    __attribute__((always_inline)) bool read() const {
        return static_cast<const Derived*>(this)->readImpl();
    }
};

// Hardware-dependent pin class (Static API)
template <uint8_t PIN>
class Esp32Pin : public PinBase<Esp32Pin<PIN>, PIN> {
public:
    static_assert(PIN < 40, "ESP32 pin number must be in range 0-39");

    __attribute__((always_inline)) void initOutput() {
        // Initialize pin multiplexer (IO_MUX) to function as a GPIO via Arduino Core
        pinMode(PIN, OUTPUT);
        // Fast direct register write for enabling output
        if constexpr (PIN < 32) {
            *gpio_enable_w1ts = (1UL << PIN);
        } else {
            *gpio_enable1_w1ts = (1UL << (PIN - 32));
        }
    }

    __attribute__((always_inline)) void initInput() {
        // Initialize pin multiplexer (IO_MUX) to function as a GPIO input via Arduino Core
        pinMode(PIN, INPUT);
        // Fast direct register write for disabling output
        if constexpr (PIN < 32) {
            *gpio_enable_w1tc = (1UL << PIN);
        } else {
            *gpio_enable1_w1tc = (1UL << (PIN - 32));
        }
    }

    __attribute__((always_inline)) void setHighImpl() const {
        if constexpr (PIN < 32) {
            *gpio_out_w1ts = (1UL << PIN);
        } else {
            *gpio_out1_w1ts = (1UL << (PIN - 32));
        }
    }

    __attribute__((always_inline)) void setLowImpl() const {
```

```
            if constexpr (PIN < 32) {
                *gpio_out_w1tc = (1UL << PIN);
            } else {
                *gpio_out1_w1tc = (1UL << (PIN - 32));
            }
        }

        __attribute__((always_inline)) void toggleImpl() const {
            portENTER_CRITICAL(&gpio_spinlock);
            toggleNonAtomicImpl();
            portEXIT_CRITICAL(&gpio_spinlock);
        }

        __attribute__((always_inline)) void toggleNonAtomicImpl() const {
            if constexpr (PIN < 32) {
                if ((*gpio_out & (1UL << PIN))) {
                    *gpio_out_w1tc = (1UL << PIN);
                } else {
                    *gpio_out_w1ts = (1UL << PIN);
                }
            } else {
                if ((*gpio_out1 & (1UL << (PIN - 32)))) {
                    *gpio_out1_w1tc = (1UL << (PIN - 32));
                } else {
                    *gpio_out1_w1ts = (1UL << (PIN - 32));
                }
            }
        }

        __attribute__((always_inline)) bool readImpl() const {
            if constexpr (PIN < 32) {
                return (*gpio_in & (1UL << PIN)) != 0;
            } else {
                return (*gpio_in1 & (1UL << (PIN - 32))) != 0;
            }
        }
    };

} // namespace static_api

namespace dynamic_api {

    // Abstract interface for Runtime Polymorphism
    class IPin {
    public:
        virtual ~IPin() = default;
        virtual void initOutput() = 0;
        virtual void initInput() = 0;
        virtual void setHigh() const = 0;
```

```
    virtual void setLow() const = 0;
    virtual void toggle() const = 0;
    virtual void toggleNonAtomic() const = 0;
    virtual bool read() const = 0;
};

// Implementation of IPin with cached registers and masks (Dynamic API)
class Esp32DynamicPin : public IPin {
private:
    uint8_t _pin;
    uint32_t _mask;
    volatile uint32_t* _w1ts_reg;
    volatile uint32_t* _w1tc_reg;
    volatile uint32_t* _out_reg;
    volatile uint32_t* _in_reg;
    volatile uint32_t* _enable_w1ts_reg;
    volatile uint32_t* _enable_w1tc_reg;

public:
    Esp32DynamicPin(uint8_t pin) : _pin(pin) {
        if (pin < 32) {
            _mask = (1UL << pin);
            _w1ts_reg = gpio_out_w1ts;
            _w1tc_reg = gpio_out_w1tc;
            _out_reg = gpio_out;
            _in_reg = gpio_in;
            _enable_w1ts_reg = gpio_enable_w1ts;
            _enable_w1tc_reg = gpio_enable_w1tc;
        } else {
            _mask = (1UL << (pin - 32));
            _w1ts_reg = gpio_out1_w1ts;
            _w1tc_reg = gpio_out1_w1tc;
            _out_reg = gpio_out1;
            _in_reg = gpio_in1;
            _enable_w1ts_reg = gpio_enable1_w1ts;
            _enable_w1tc_reg = gpio_enable1_w1tc;
        }
    }

    void initOutput() override {
        pinMode(_pin, OUTPUT);
        *_enable_w1ts_reg = _mask;
    }

    void initInput() override {
        pinMode(_pin, INPUT);
        *_enable_w1tc_reg = _mask;
    }
```

```
        void setHigh() const override {
            *_w1ts_reg = _mask;
        }

        void setLow() const override {
            *_w1tc_reg = _mask;
        }

        void toggle() const override {
            portENTER_CRITICAL(&gpio_spinlock);
            toggleNonAtomic();
            portEXIT_CRITICAL(&gpio_spinlock);
        }

        void toggleNonAtomic() const override {
            if (*_out_reg & _mask) {
                *_w1tc_reg = _mask;
            } else {
                *_w1ts_reg = _mask;
            }
        }

        bool read() const override {
            return (*_in_reg & _mask) != 0;
        }
    };

} // namespace dynamic_api

// Type Safety wrappers
template <typename PinType>
class OutputPin {
private:
    PinType _pin;
public:
    template <typename... Args>
    __attribute__((always_inline)) OutputPin(Args&&... args) : _pin(std::forward<Args>(args)...) {}

    __attribute__((always_inline)) void init() {
        _pin.initOutput();
    }

    __attribute__((always_inline)) void setHigh() const {
        _pin.setHigh();
    }

    __attribute__((always_inline)) void setLow() const {
        _pin.setLow();
    }
```

```
    __attribute__((always_inline)) void toggle() const {
        _pin.toggle();
    }

    __attribute__((always_inline)) void toggleNonAtomic() const {
        _pin.toggleNonAtomic();
    }
};

// Specialization of OutputPin for IPin* to support Dynamic API
template <>
class OutputPin<dynamic_api::IPin*> {
private:
    dynamic_api::IPin* _pin;
public:
    __attribute__((always_inline)) OutputPin(dynamic_api::IPin* pin) : _pin(pin) {}

    __attribute__((always_inline)) void init() {
        if (_pin) _pin->initOutput();
    }

    __attribute__((always_inline)) void setHigh() const {
        if (_pin) _pin->setHigh();
    }

    __attribute__((always_inline)) void setLow() const {
        if (_pin) _pin->setLow();
    }

    __attribute__((always_inline)) void toggle() const {
        if (_pin) _pin->toggle();
    }

    __attribute__((always_inline)) void toggleNonAtomic() const {
        if (_pin) _pin->toggleNonAtomic();
    }
};

template <typename PinType>
class InputPin {
private:
    PinType _pin;
public:
    template <typename... Args>
    __attribute__((always_inline)) InputPin(Args&&... args) : _pin(std::forward<Args>(args)...) {}

    __attribute__((always_inline)) void init() {
        _pin.initInput();
```

```
    }

    __attribute__((always_inline)) bool read() const {
        return _pin.read();
    }
};

// Specialization of InputPin for IPin* to support Dynamic API
template <>
class InputPin<dynamic_api::IPin*> {
private:
    dynamic_api::IPin* _pin;
public:
    __attribute__((always_inline)) InputPin(dynamic_api::IPin* pin) : _pin(pin) {}

    __attribute__((always_inline)) void init() {
        if (_pin) _pin->initInput();
    }

    __attribute__((always_inline)) bool read() const {
        return _pin ? _pin->read() : false;
    }
};

// GPIOManager (Facade / Single Point of Entry)
class GPIOManager {
private:
    mutable dynamic_api::Esp32DynamicPin* _dynamic_pins[40];

public:
    GPIOManager() {
        for (int i = 0; i < 40; ++i) {
            _dynamic_pins[i] = nullptr;
        }
    }

    ~GPIOManager() {
        for (int i = 0; i < 40; ++i) {
            delete _dynamic_pins[i];
        }
    }

    GPIOManager(const GPIOManager&) = delete;
    GPIOManager& operator=(const GPIOManager&) = delete;

    template <uint8_t PIN>
    static constexpr static_api::Esp32Pin<PIN> get() {
        static_assert(PIN < 40, "ESP32 pin number must be less than 40");
        return static_api::Esp32Pin<PIN>();
```

```
        }

        dynamic_api::IPin* operator[](uint8_t pin) const {
            if (pin >= 40) return nullptr;
            if (!_dynamic_pins[pin]) {
                _dynamic_pins[pin] = new dynamic_api::Esp32DynamicPin(pin);
            }
            return _dynamic_pins[pin];
        }
    };

    // Global GPIO manager instance
    inline GPIOManager GpioCore;

} // namespace hal

#endif // FAST_GPIO_H
```

Лістинг Г.2 — test_sketch

```
#include "../FastGPIO.h"

TaskHandle_t BenchmarkTaskHandle = NULL;

void benchmarkTask(void* parameter);

void setup() {
  Serial.begin(115200);
  while (!Serial) {
    delay(10);
  }

  Serial.println("\n==========================================");
  Serial.println("ESP32 FastGPIO High-Performance Benchmark");
  Serial.println("==========================================");

  xTaskCreatePinnedToCore(
    benchmarkTask,
    "BenchmarkTask",
    8192,
    NULL,
    configMAX_PRIORITIES - 1,
    &BenchmarkTaskHandle,
    1
  );
}

void loop() {
```

```
  vTaskDelay(pdMS_TO_TICKS(1000));
}

void benchmarkTask(void* parameter) {

  vTaskDelay(pdMS_TO_TICKS(1000));

  const uint8_t PIN_STANDARD = 32;
  const uint8_t PIN_STATIC   = 4;
  const uint8_t PIN_DYNAMIC  = 33;

  Serial.printf("Configuring  pins:  Standard=%d,  Static=%d,  Dynamic=%d\n",  PIN_STANDARD,  PIN_STATIC,
PIN_DYNAMIC);

  hal::OutputPin<hal::static_api::Esp32Pin<PIN_STATIC>> fastStaticPin;
  hal::OutputPin<hal::dynamic_api::IPin*> fastDynamicPin(hal::GpioCore[PIN_DYNAMIC]);

  pinMode(PIN_STANDARD, OUTPUT);
  fastStaticPin.init();
  fastDynamicPin.init();


  const uint32_t iterations = 100000;
  uint32_t start_time = 0;
  uint32_t duration_standard = 0;
  uint32_t duration_static = 0;
  uint32_t duration_dynamic = 0;

  portMUX_TYPE test_mux = portMUX_INITIALIZER_UNLOCKED;

  Serial.println("Starting measurements (100,000 write cycles per test)...");

  vTaskDelay(pdMS_TO_TICKS(100));

  portENTER_CRITICAL(&test_mux);
  start_time = micros();
  for (uint32_t i = 0; i < iterations; ++i) {
    digitalWrite(PIN_STANDARD, HIGH);
    digitalWrite(PIN_STANDARD, LOW);
  }
  duration_standard = micros() - start_time;
  portEXIT_CRITICAL(&test_mux);

  vTaskDelay(pdMS_TO_TICKS(100));

  portENTER_CRITICAL(&test_mux);
  start_time = micros();
  for (uint32_t i = 0; i < iterations; ++i) {
    fastStaticPin.setHigh();
```

```
    fastStaticPin.setLow();
  }
  duration_static = micros() - start_time;
  portEXIT_CRITICAL(&test_mux);

  vTaskDelay(pdMS_TO_TICKS(100));

  portENTER_CRITICAL(&test_mux);
  start_time = micros();
  for (uint32_t i = 0; i < iterations; ++i) {
    fastDynamicPin.setHigh();
    fastDynamicPin.setLow();
  }
  duration_dynamic = micros() - start_time;
  portEXIT_CRITICAL(&test_mux);

  Serial.println("\n================= BENCHMARK RESULTS =================");
  Serial.printf("1. Standard digitalWrite()  : %u us (%f ns per toggle)\n",
                duration_standard, (double)duration_standard * 1000.0 / (iterations * 2));
  Serial.printf("2. FastGPIO Static (CRTP)  : %u us (%f ns per toggle)\n",
                duration_static, (double)duration_static * 1000.0 / (iterations * 2));
  Serial.printf("3. FastGPIO Dynamic (Virtual): %u us (%f ns per toggle)\n",
                duration_dynamic, (double)duration_dynamic * 1000.0 / (iterations * 2));
  Serial.println("-----------------------------------------------------");

  if (duration_static > 0) {
    Serial.printf("Static API Speedup : %.2f times faster than standard\n",
                  (double)duration_standard / duration_static);
  } else {
    Serial.println("Static API Speedup : N/A (duration is 0)");
  }

  if (duration_dynamic > 0) {
    Serial.printf("Dynamic API Speedup: %.2f times faster than standard\n",
                  (double)duration_standard / duration_dynamic);
  } else {
    Serial.println("Dynamic API Speedup: N/A (duration is 0)");
  }
  Serial.println("=====================================================\n");
  Serial.flush();

  Serial.printf("Generating fast square wave on Static Pin (PIN_%d) in bursts of 100k toggles.\n",
PIN_STATIC);
  Serial.println("Interrupts on Core 1 are disabled during each burst to prevent jitter.");
  Serial.flush();
  vTaskDelay(pdMS_TO_TICKS(100));

  while (true) {
    portENTER_CRITICAL(&test_mux);
```

```
    for (uint32_t i = 0; i < 100000; ++i) {
      fastStaticPin.toggleNonAtomic();
    }
    portEXIT_CRITICAL(&test_mux);

    vTaskDelay(pdMS_TO_TICKS(10));
  }
}
```

Лістинг Г.3 — scope_test

```
#include "../FastGPIO.h"

// Define GPIO pins for the oscilloscope test
// TEST_PIN_A: GPIO 25 is where we output the pulse signals for Mode A
// TEST_PIN_B: GPIO 27 is where we output the pulse signals for Mode B
// SYNC_PIN: GPIO 26 is the trigger synchronization signal for the oscilloscope
const uint8_t TEST_PIN_A = 25;
const uint8_t TEST_PIN_B = 27;
const uint8_t SYNC_PIN = 26;

// Вибір режимів роботи з GPIO для кожного з пінів:
// 0 - Standard digitalWrite()
// 1 - FastGPIO Static (CRTP)
// 2 - FastGPIO Dynamic (Virtual)
#define GPIO_MODE_A 0 // Режим для TEST_PIN_A (GPIO 25)
#define GPIO_MODE_B 2 // Режим для TEST_PIN_B (GPIO 27)

template <uint8_t PIN, int MODE>
struct GpioPinSelector;

// Спеціалізація для Standard digitalWrite()
template <uint8_t PIN>
struct GpioPinSelector<PIN, 0> {
  __attribute__((always_inline)) void init() {
    pinMode(PIN, OUTPUT);
  }
  __attribute__((always_inline)) void setHigh() const {
    digitalWrite(PIN, HIGH);
  }
  __attribute__((always_inline)) void setLow() const {
    digitalWrite(PIN, LOW);
  }
};

// Спеціалізація для FastGPIO Static (CRTP)
template <uint8_t PIN>
struct GpioPinSelector<PIN, 1> {
  hal::OutputPin<hal::static_api::Esp32Pin<PIN>> pin;
```

```
  __attribute__((always_inline)) void init() {
    pin.init();
  }
  __attribute__((always_inline)) void setHigh() const {
    pin.setHigh();
  }
  __attribute__((always_inline)) void setLow() const {
    pin.setLow();
  }
};

// Спеціалізація для FastGPIO Dynamic (Virtual)
template <uint8_t PIN>
struct GpioPinSelector<PIN, 2> {
  hal::dynamic_api::IPin* pinPtr = nullptr;
  __attribute__((always_inline)) void init() {
    pinPtr = hal::GpioCore[PIN];
    pinPtr->initOutput();
  }
  __attribute__((always_inline)) void setHigh() const {
    pinPtr->setHigh();
  }
  __attribute__((always_inline)) void setLow() const {
    pinPtr->setLow();
  }
};

GpioPinSelector<TEST_PIN_A, GPIO_MODE_A> testPinA;
GpioPinSelector<TEST_PIN_B, GPIO_MODE_B> testPinB;
GpioPinSelector<SYNC_PIN, 1> syncPin; // Синхронізація завжди швидка (Static)

// Spinlock for critical sections
portMUX_TYPE test_mux = portMUX_INITIALIZER_UNLOCKED;

// Command and settings variables shared between Serial thread and high priority task (Core 1)
volatile int pendingCommand = 0;
volatile bool continuousMode = false;
volatile int continuousType = 0;
volatile int pulseCount = 1;

// Serial parsing variables
unsigned long lastCharTime = 0;
String inputBuffer = "";

// Task handle
TaskHandle_t PulseTaskHandle = NULL;

// Declare external ROM delay function to ensure compilation on all ESP32 BSP versions
extern "C" void ets_delay_us(uint32_t us);
```

```
        // Assembly NOPs macros for nanosecond-precision delays at 240 MHz (1 cycle = 4.17 ns)
        #define NOP __asm__ __volatile__ ("nop")
        #define NOP5 NOP; NOP; NOP; NOP; NOP
        #define NOP10 NOP5; NOP5
        #define NOP20 NOP10; NOP10
        #define NOP50 NOP20; NOP20; NOP10
        #define NOP100 NOP50; NOP50
        #define NOP200 NOP100; NOP100
        #define NOP500 NOP200; NOP200; NOP100

        // Compile-time precision delay helper macros
        #define DELAY_10NS()  do { NOP; NOP; } while(0)                 // ~8.3 ns delay + write instruction time
(~12ns total)
        #define DELAY_50NS()  do { NOP10; NOP; NOP; } while(0)           // ~12 NOPs (~50 ns)
        #define DELAY_100NS() do { NOP20; NOP; NOP; NOP; NOP; } while(0)  // ~24 NOPs (~100 ns)
        #define DELAY_500NS() do { NOP100; NOP20; } while(0)             // ~120 NOPs (~500 ns)

        // Helper inline function to toggle Sync pin for pre-trigger sync signal
        __attribute__((always_inline)) inline void startSync() {
          syncPin.setHigh();
          ets_delay_us(1); // 1 us pre-trigger time to stabilize oscilloscope sweep
        }

        __attribute__((always_inline)) inline void endSync() {
          syncPin.setLow();
        }

        // Helper function to calculate safe chunk sizes (approx 15-20 ms of generation per chunk)
        // to prevent starving the CPU/watchdog when generating large numbers of pulses
        int getChunkSize(int mode) {
          switch (mode) {
            case 1: return 100000;
            case 2: return 100000;
            case 3: return 100000;
            case 4: return 20000;
            case 5: return 10000;
            case 6: return 1000;
            default: return 1000;
          }
        }

        // Forward declarations
        void pulseTask(void* parameter);
        void processCommand(String cmd);
        void printMenu();

        void setup() {
          Serial.begin(115200);
```

```
while (!Serial) {
  delay(10);
}

Serial.println("\n=============================================");
Serial.println("ESP32 FastGPIO Oscilloscope Pulse Generator");
Serial.println("=============================================");
Serial.printf("TEST PIN A (Channel 1): GPIO %d\n", TEST_PIN_A);
Serial.printf("TEST PIN B (Channel 2): GPIO %d\n", TEST_PIN_B);
Serial.printf("SYNC PIN (Ext Trig): GPIO %d\n", SYNC_PIN);

#if GPIO_MODE_A == 0
  Serial.println("  Pin A Mode: Standard digitalWrite()");
#elif GPIO_MODE_A == 1
  Serial.println("  Pin A Mode: FastGPIO Static (CRTP)");
#elif GPIO_MODE_A == 2
  Serial.println("  Pin A Mode: FastGPIO Dynamic (Virtual)");
#endif

#if GPIO_MODE_B == 0
  Serial.println("  Pin B Mode: Standard digitalWrite()");
#elif GPIO_MODE_B == 1
  Serial.println("  Pin B Mode: FastGPIO Static (CRTP)");
#elif GPIO_MODE_B == 2
  Serial.println("  Pin B Mode: FastGPIO Dynamic (Virtual)");
#endif
Serial.println("---------------------------------------------");

// Initialize output pins using selection wrapper
testPinA.init();
testPinB.init();
syncPin.init();

// Make sure pins start LOW
testPinA.setLow();
testPinB.setLow();
syncPin.setLow();

// Create high-priority task pinned to Core 1 for jitter-free pulse generation
xTaskCreatePinnedToCore(
  pulseTask,                    // Task function
  "PulseTask",                  // Task name
  8192,                         // Stack size
  NULL,                         // Parameters
  configMAX_PRIORITIES - 1,     // Priority (highest)
  &PulseTaskHandle,             // Task handle
  1                             // Pinned to Core 1
);
```

```
  printMenu();
}

void loop() {
  while (Serial.available() > 0) {
    char key = Serial.read();
    inputBuffer += key;
    lastCharTime = millis();
  }

  // If we have data and either we got a newline, or 50ms of silence passed since the last character
  if (inputBuffer.length() > 0 &&
      (inputBuffer.indexOf('\n') != -1 || inputBuffer.indexOf('\r') != -1 || (millis() - lastCharTime >
50))) {

    processCommand(inputBuffer);
    inputBuffer = ""; // Reset buffer
  }

  delay(10); // Small delay to let loop sleep
}

void processCommand(String cmd) {
  cmd.trim();
  if (cmd.length() == 0) return;

  // Check if command contains a colon for the [Mode]:[Count] format
  int colonIndex = cmd.indexOf(':');
  if (colonIndex != -1) {
    String modeStr = cmd.substring(0, colonIndex);
    String countStr = cmd.substring(colonIndex + 1);

    modeStr.trim();
    countStr.trim();

    int mode = modeStr.toInt();
    int count = countStr.toInt();

    if (mode >= 1 && mode <= 6 && count > 0) {
      if (count > 10000000) {
        Serial.println("Помилка: Занадто велика кількість імпульсів (макс. 10 000 000).");
        return;
      }
      continuousMode = false;
      pulseCount = count;
      pendingCommand = mode;

      const char* modeName = "";
      if (mode == 1) modeName = "~10 нс";
```

```
    else if (mode == 2) modeName = "~50 нс";
    else if (mode == 3) modeName = "~100 нс";
    else if (mode == 4) modeName = "~500 нс";
    else if (mode == 5) modeName = "1 мкс";
    else if (mode == 6) modeName = "10 мкс";

    Serial.printf("CMD: Генеруємо пачку з %d імпульсів (Режим %d - %s)\n", count, mode, modeName);
  } else {
    Serial.printf("Помилка парсингу: Режим має бути 1-6, кількість > 0. Отримано: %s\n", cmd.c_str());
  }
} else {
  // Single character command
  if (cmd.length() == 1) {
    char key = cmd[0];
    if (key >= '1' && key <= '6') {
      continuousMode = false;
      pulseCount = 1; // Default to 1 pulse
      pendingCommand = key - '0';
      Serial.printf("CMD: Генеруємо 1 імпульс (Режим %d)\n", pendingCommand);
    }
    else if (key == '7') {
      continuousMode = false;
      pulseCount = 1;
      pendingCommand = 7; // Double pulse
      Serial.println("CMD: Sending Double Pulse sequence...");
    }
    else if (key == 'a' || key == 'A') {
      continuousMode = false;
      pulseCount = 1;
      pendingCommand = 10; // Sweep mode
      Serial.println("CMD: Starting pulse sweep (10ns -> 10us)...");
    }
    else if (key == '8') {
      continuousMode = true;
      continuousType = 8;
      Serial.println("CMD: Generating continuous 100 kHz Square Wave (in bursts)...");
    }
    else if (key == '9') {
      continuousMode = true;
      continuousType = 9;
      Serial.println("CMD: Generating continuous 1 MHz Square Wave (in bursts)...");
    }
    else if (key == '0') {
      continuousMode = true;
      continuousType = 11; // 10 MHz
      Serial.println("CMD: Generating continuous 10 MHz Square Wave (in bursts)...");
    }
    else if (key == 's' || key == 'S') {
      continuousMode = false;
```

```
        continuousType = 0;
        testPinA.setLow();
        testPinB.setLow();
        syncPin.setLow();
        Serial.println("CMD: Stopped continuous wave.");
      }
      else if (key == 'h' || key == 'H' || key == '?') {
        printMenu();
      }
      else {
        Serial.printf("Unknown command: '%c'\n", key);
      }
    } else {
      Serial.printf("Невідома команда: \"%s\"\n", cmd.c_str());
    }
  }
}

void pulseTask(void* parameter) {
  // Wait for Serial configuration to settle
  vTaskDelay(pdMS_TO_TICKS(500));

  while (true) {
    if (pendingCommand > 0) {
      int cmd = pendingCommand;
      pendingCommand = 0; // Reset command flag

      switch (cmd) {
        case 1: { // 10 ns Pulse Burst (Chunked)
          int remaining = pulseCount;
          int chunkSize = getChunkSize(1);
          startSync();
          while (remaining > 0) {
            int chunk = (remaining > chunkSize) ? chunkSize : remaining;
            portENTER_CRITICAL(&test_mux);
            for (int i = 0; i < chunk; ++i) {
              testPinA.setHigh();
              testPinA.setLow(); // Minimum hardware pulse width (~12-16 ns)

              DELAY_100NS(); // Gap between Pin A and Pin B

              testPinB.setHigh();
              testPinB.setLow(); // Minimum hardware pulse width

              if (i < chunk - 1 || remaining > chunk) {
                DELAY_100NS(); // Gap between pulse pairs
              }
            }
            portEXIT_CRITICAL(&test_mux);
```

```
    remaining -= chunk;
    if (remaining > 0) {
      vTaskDelay(1); // Yield to prevent watchdog reset
    }
  }
  endSync();
  break;
}

case 2: { // 50 ns Pulse Burst (Chunked)
  int remaining = pulseCount;
  int chunkSize = getChunkSize(2);
  startSync();
  while (remaining > 0) {
    int chunk = (remaining > chunkSize) ? chunkSize : remaining;
    portENTER_CRITICAL(&test_mux);
    for (int i = 0; i < chunk; ++i) {
      testPinA.setHigh();
      DELAY_50NS();
      testPinA.setLow();

      DELAY_100NS(); // Gap

      testPinB.setHigh();
      DELAY_50NS();
      testPinB.setLow();

      if (i < chunk - 1 || remaining > chunk) {
        DELAY_100NS(); // Gap
      }
    }
    portEXIT_CRITICAL(&test_mux);
    remaining -= chunk;
    if (remaining > 0) {
      vTaskDelay(1); // Yield to prevent watchdog reset
    }
  }
  endSync();
  break;
}

case 3: { // 100 ns Pulse Burst (Chunked)
  int remaining = pulseCount;
  int chunkSize = getChunkSize(3);
  startSync();
  while (remaining > 0) {
    int chunk = (remaining > chunkSize) ? chunkSize : remaining;
    portENTER_CRITICAL(&test_mux);
    for (int i = 0; i < chunk; ++i) {
```

```
      testPinA.setHigh();
      DELAY_100NS();
      testPinA.setLow();

      DELAY_100NS(); // Gap

      testPinB.setHigh();
      DELAY_100NS();
      testPinB.setLow();

      if (i < chunk - 1 || remaining > chunk) {
        DELAY_100NS(); // Gap
      }
    }
    portEXIT_CRITICAL(&test_mux);
    remaining -= chunk;
    if (remaining > 0) {
      vTaskDelay(1); // Yield to prevent watchdog reset
    }
  }
  endSync();
  break;
}

case 4: { // 500 ns Pulse Burst (Chunked)
  int remaining = pulseCount;
  int chunkSize = getChunkSize(4);
  startSync();
  while (remaining > 0) {
    int chunk = (remaining > chunkSize) ? chunkSize : remaining;
    portENTER_CRITICAL(&test_mux);
    for (int i = 0; i < chunk; ++i) {
      testPinA.setHigh();
      DELAY_500NS();
      testPinA.setLow();

      DELAY_500NS(); // Gap

      testPinB.setHigh();
      DELAY_500NS();
      testPinB.setLow();

      if (i < chunk - 1 || remaining > chunk) {
        DELAY_500NS(); // Gap
      }
    }
    portEXIT_CRITICAL(&test_mux);
    remaining -= chunk;
    if (remaining > 0) {
```

```
      vTaskDelay(1); // Yield to prevent watchdog reset
    }
  }
  endSync();
  break;
}

case 5: { // 1 us Pulse Burst (Chunked)
  int remaining = pulseCount;
  int chunkSize = getChunkSize(5);
  startSync();
  while (remaining > 0) {
    int chunk = (remaining > chunkSize) ? chunkSize : remaining;
    portENTER_CRITICAL(&test_mux);
    for (int i = 0; i < chunk; ++i) {
      testPinA.setHigh();
      ets_delay_us(1);
      testPinA.setLow();

      ets_delay_us(1); // Gap

      testPinB.setHigh();
      ets_delay_us(1);
      testPinB.setLow();

      if (i < chunk - 1 || remaining > chunk) {
        ets_delay_us(1); // Gap
      }
    }
    portEXIT_CRITICAL(&test_mux);
    remaining -= chunk;
    if (remaining > 0) {
      vTaskDelay(1); // Yield to prevent watchdog reset
    }
  }
  endSync();
  break;
}

case 6: { // 10 us Pulse Burst (Chunked)
  int remaining = pulseCount;
  int chunkSize = getChunkSize(6);
  startSync();
  while (remaining > 0) {
    int chunk = (remaining > chunkSize) ? chunkSize : remaining;
    portENTER_CRITICAL(&test_mux);
    for (int i = 0; i < chunk; ++i) {
      testPinA.setHigh();
      ets_delay_us(10);
```

```
      testPinA.setLow();

      ets_delay_us(10); // Gap

      testPinB.setHigh();
      ets_delay_us(10);
      testPinB.setLow();

      if (i < chunk - 1 || remaining > chunk) {
        ets_delay_us(10); // Gap
      }
    }
    portEXIT_CRITICAL(&test_mux);
    remaining -= chunk;
    if (remaining > 0) {
      vTaskDelay(1); // Yield to prevent watchdog reset
    }
  }
  endSync();
  break;
}

case 7: // Double Pulse Test
  portENTER_CRITICAL(&test_mux);
  startSync();
  // Pin A Double Pulse
  testPinA.setHigh();
  ets_delay_us(1);
  testPinA.setLow();
  ets_delay_us(2);
  testPinA.setHigh();
  ets_delay_us(1);
  testPinA.setLow();

  ets_delay_us(5); // Gap between Pin A and Pin B

  // Pin B Double Pulse
  testPinB.setHigh();
  ets_delay_us(1);
  testPinB.setLow();
  ets_delay_us(2);
  testPinB.setHigh();
  ets_delay_us(1);
  testPinB.setLow();

  endSync();
  portEXIT_CRITICAL(&test_mux);
  break;
```

```
case 10: // Sweep Mode
  // 10ns
  portENTER_CRITICAL(&test_mux);
  startSync();
  testPinA.setHigh(); testPinA.setLow();
  DELAY_100NS();
  testPinB.setHigh(); testPinB.setLow();
  endSync();
  portEXIT_CRITICAL(&test_mux);
  vTaskDelay(pdMS_TO_TICKS(10));

  // 50ns
  portENTER_CRITICAL(&test_mux);
  startSync();
  testPinA.setHigh(); DELAY_50NS(); testPinA.setLow();
  DELAY_100NS();
  testPinB.setHigh(); DELAY_50NS(); testPinB.setLow();
  endSync();
  portEXIT_CRITICAL(&test_mux);
  vTaskDelay(pdMS_TO_TICKS(10));

  // 100ns
  portENTER_CRITICAL(&test_mux);
  startSync();
  testPinA.setHigh(); DELAY_100NS(); testPinA.setLow();
  DELAY_100NS();
  testPinB.setHigh(); DELAY_100NS(); testPinB.setLow();
  endSync();
  portEXIT_CRITICAL(&test_mux);
  vTaskDelay(pdMS_TO_TICKS(10));

  // 500ns
  portENTER_CRITICAL(&test_mux);
  startSync();
  testPinA.setHigh(); DELAY_500NS(); testPinA.setLow();
  DELAY_500NS();
  testPinB.setHigh(); DELAY_500NS(); testPinB.setLow();
  endSync();
  portEXIT_CRITICAL(&test_mux);
  vTaskDelay(pdMS_TO_TICKS(10));

  // 1us
  portENTER_CRITICAL(&test_mux);
  startSync();
  testPinA.setHigh(); ets_delay_us(1); testPinA.setLow();
  ets_delay_us(1);
  testPinB.setHigh(); ets_delay_us(1); testPinB.setLow();
  endSync();
  portEXIT_CRITICAL(&test_mux);
```

```
      vTaskDelay(pdMS_TO_TICKS(10));

      // 10us
      portENTER_CRITICAL(&test_mux);
      startSync();
      testPinA.setHigh(); ets_delay_us(10); testPinA.setLow();
      ets_delay_us(10);
      testPinB.setHigh(); ets_delay_us(10); testPinB.setLow();
      endSync();
      portEXIT_CRITICAL(&test_mux);
      vTaskDelay(pdMS_TO_TICKS(10));
      break;
  }
}

if (continuousMode) {
  if (continuousType == 8) { // 100 kHz Square Wave in bursts
    for (int i = 0; i < 50; ++i) {
      testPinA.setHigh();
      testPinB.setHigh();
      ets_delay_us(5);
      testPinA.setLow();
      testPinB.setLow();
      ets_delay_us(5);
    }
    vTaskDelay(1); // Yield to feed WDT and allow background tasks
  }
  else if (continuousType == 9) { // 1 MHz Square Wave in bursts
    for (int i = 0; i < 500; ++i) {
      testPinA.setHigh();
      testPinB.setHigh();
      DELAY_500NS();
      testPinA.setLow();
      testPinB.setLow();
      DELAY_500NS();
    }
    vTaskDelay(1); // Yield to feed WDT
  }
  else if (continuousType == 11) { // 10 MHz Square Wave in bursts
    // Generate in bursts of 5000 cycles (500 us total)
    for (int i = 0; i < 5000; ++i) {
      testPinA.setHigh();
      testPinB.setHigh();
      __asm__ __volatile__ (
          "nop\n\t"
          "nop\n\t"
          "nop\n\t"
          "nop\n\t"
          "nop\n\t"
```

```
            "nop\n\t"
            "nop\n\t"
            "nop\n\t"
          );
          testPinA.setLow();
          testPinB.setLow();
          __asm__ __volatile__ (
            "nop\n\t"
            "nop\n\t"
            "nop\n\t"
            "nop\n\t"
            "nop\n\t"
          );
        }
        vTaskDelay(1); // Yield to feed WDT
      }
    }
    else {
      // Idle wait
      vTaskDelay(pdMS_TO_TICKS(5));
    }
  }
}

void printMenu() {
  Serial.println("\n----------------- КОМАНДНЕ МЕНЮ -----------------");
  Serial.println("Формат вибору імпульсів: [Режим]:[Кількість]");
  Serial.println("Приклад: '1:10' - 10 імпульсів по 10 нс, '1:100000' - 100 000 імпульсів по 10 нс.");
  Serial.println("Якщо ввести просто цифру режиму (наприклад '1'), згенерується 1 імпульс.");
  Serial.println("-------------------------------------------------");
  Serial.println("Режими імпульсів (макс. 10 000 000):");
  Serial.println("  1 - Імпульси ~10 нс");
  Serial.println("  2 - Імпульси ~50 нс");
  Serial.println("  3 - Імпульси ~100 нс");
  Serial.println("  4 - Імпульси ~500 нс");
  Serial.println("  5 - Імпульси 1 мкс");
  Serial.println("  6 - Імпульси 10 мкс");
  Serial.println("  7 - Подвійний імпульс (1 мкс / 2 мкс / 1 мкс)");
  Serial.println("  a - Автоматичний перебір (Sweep) всіх типів імпульсів");
  Serial.println("Безперервні режими:");
  Serial.println("  8 - Безперервний меандр 100 кГц");
  Serial.println("  9 - Безперервний меандр 1 МГц");
  Serial.println("  0 - Безперервний меандр 10 МГц");
  Serial.println("  s - Зупинити безперервний сигнал");
  Serial.println("  h - Показати це меню знову");
  Serial.println("-------------------------------------------------");
}
```