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 __ грудня 2025р.

завідувач кафедри _____ Максим ХРУСЛОВ
(підпис)

«___»_____ 2025 р.

Кваліфікаційна робота

здобувача другого (магістерського) рівня вищої освіти

«МЕТОД КЕРУВАННЯ LEDC ТАЙМЕРАМИ МІКРОКОНТРОЛЕРА ESP32»

Спеціальність ***174 – Автоматизація, комп'ютерно-інтегровані технології та робототехніка***
Освітня програма ***Комп'ютеризовані системи управління та автоматика***

Виконавець _____ **Даніель ГОРЕНКО**
(підпис)

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

Харків – 2025

АНОТАЦІЯ

Пояснювальна записка до магістерської атестаційної роботи складається зі вступу, трьох розділів, висновків, списку використаних джерел і додатків. Загальний обсяг роботи складає 84 сторінок, із яких 58 сторінок основної частини з 29 рисунками, 1 таблицею, 15 найменуваннями списку використаних джерел та 6 додатками.

Метою кваліфікаційної роботи є підвищення ефективності керування високошвидкісними та низькошвидкісними таймерами та каналами підсистеми LEDC мікроконтролера ESP32 за допомогою прямого доступу до регістрів.

Об'єкт дослідження – процес генерації широтно-імпульсних (ШІМ) сигналів апаратною підсистемою мікроконтролерів.

Предмет дослідження – методи та засоби прямого регістрового програмування для конфігурування таймерів і каналів підсистеми LEDC мікроконтролера ESP32.

Проблема, яка вирішується в кваліфікаційній роботі, полягає в усуненні програмних затримок та обмежень, що вносяться стандартними високорівневими інтерфейсами (HAL/API), шляхом створення оптимізованого алгоритму прямого керування апаратними ресурсами для забезпечення детермінованості системи.

Область застосування — розробка вбудованих систем жорсткого реального часу. Розроблений програмний продукт може широко використовуватися в робототехніці (керування двигунами), силовій електроніці та системах цифрової обробки сигналів.

Ключові слова: Мікроконтролер, ESP32, LEDC, ШІМ, регістри, латентність, драйвер, реальний час, API, швидкодія.

ABSTRACT

The explanatory note to the master's thesis consists of an introduction, three chapters, conclusions, a list of references, and appendices. The total volume of the work is 84 pages, including 58 pages of the main body containing 29 figures, 1 table, 15 entries in the list of references, and 6 appendices.

The purpose of the qualification work is to improve the efficiency of controlling high-speed and low-speed timers and channels of the ESP32 microcontroller's LEDC subsystem using direct register access.

The object of research is the process of generating pulse-width modulation (PWM) signals by the hardware subsystem of microcontrollers.

The subject of research concerns the methods and tools of direct register programming for configuring timers and channels of the LEDC subsystem of the ESP32 microcontroller.

The problem addressed in the qualification work is the elimination of software delays and limitations introduced by standard high-level interfaces (HAL/API) by creating an optimized algorithm for direct hardware resource control to ensure system determinism.

The field of application is the development of hard real-time embedded systems. The developed software product can be widely used in robotics (motor control), power electronics, and digital signal processing systems.

Keywords: Microcontroller, ESP32, LEDC, PWM, registers, latency, driver, real-time, API, performance.

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ І УМОВНИХ ПОЗНАЧЕНЬ

API – Application Programming Interface — Прикладний програмний інтерфейс.

IDF – IoT Development Framework – Фреймворк розробки пристроїв Інтернету речей.

САК - системах автоматичного керування

DSP – Digital Signal Processing – Цифрова обробка сигналів.

ШІМ – Широтно-імпульсна модуляція.

GPIO – General-Purpose Input/Output – Порти введення-виведення загального призначення.

IoT – Internet of Things – Інтернет речей

ВСТУП

Розвиток сучасних вбудованих систем та технологій Інтернету речей (IoT) висуває все вищі вимоги до точності, швидкодії та надійності керуючих пристроїв. Мікроконтролери стали невід'ємною частиною систем автоматизації, робототехніки та силової електроніки, забезпечуючи інтелектуальне керування складними технологічними процесами. Серед них особливе місце посідає платформа ESP32, яка поєднує високу обчислювальну потужність із широкими комунікаційними можливостями.

Одним із ключових завдань у системах керування є точна генерація сигналів, зокрема широтно-імпульсної модуляції (ШІМ), яка використовується для керування двигунами, освітленням та перетворювачами енергії. Однак стандартні високорівневі засоби розробки часто створюють надлишкові рівні абстракції, що вносить неконтрольовані затримки та обмежує ефективність використання апаратних ресурсів у задачах жорсткого реального часу.

Вирішення цієї проблеми полягає у переході до методів прямого керування апаратним забезпеченням. Реалізація низькорівневого доступу до регістрів підсистеми LEDC дозволяє досягти максимальної детермінованості та мінімізувати час реакції системи. Враховуючи це, дослідження методів прямого регістрового програмування є актуальним напрямком, що відкриває нові можливості для створення високошвидкісних та прецизійних систем автоматичного керування на базі доступних мікроконтролерів.

РОЗДІЛ 1
АНАЛІЗ АРХІТЕКТУР ТА ПРОГРАМНИХ ЗАСОБІВ КЕРУВАННЯ У СУЧАСНИХ ВБУДОВАНИХ СИСТЕМАХ

1.1. Роль вбудованих систем у сучасній автоматизації та Інтернеті Речей

У сучасних технологічних системах усе більшого значення набувають вбудовані обчислювальні модулі. На відміну від комп'ютерів загального призначення, таких як персональні комп'ютери або сервери, вбудована система (embedded system) — це спеціалізована комп'ютерна система, розроблена для виконання конкретного набору завдань і інтегрована в пристрій, яким вона керує [1]. Вони є апаратно-програмною «начинкою», що забезпечує інтелектуальне керування процесами, збір даних і комунікацію між пристроями.

Сфера застосування вбудованих систем охоплює практично всі аспекти сучасного життя. Вони є основою робототехнічних комплексів, автоматизованих виробничих ліній, систем керування автомобільним транспортом, медичного обладнання, сенсорних мереж і побутових пристроїв. Ключовими факторами, що зумовили таке широке розповсюдження, є високий рівень інтеграції компонентів, компактність, надійність та енергоефективність, що робить їх ключовим елементом сучасної електроніки.

Розвиток мережевих технологій, зокрема бездротових, призвів до появи концепції Інтернету Речей (Internet of Things, IoT). IoT по суті є глобальною мережею, що об'єднує мільярди вбудованих систем («речей»), оснащених давачами, виконавчими механізмами та програмним забезпеченням для підключення та обміну даними через Інтернет [2]. Це дозволяє створювати складні, розподілені системи автоматизації, здатні до самодіагностики, віддаленого моніторингу та адаптивного керування в реальному часі.

Таким чином, вбудовані системи перетворилися з простих контролерів на складні обчислювальні вузли, що лежать в основі четвертої промислової революції (Індустрія 4.0) та концепції «розумних» міст і домівок.

Особливої актуальності в сучасній електроніці набувають системи жорсткого реального часу (Hard Real-Time). У таких сферах, як керування безколекторними двигунами (FOC), цифрова обробка сигналів або силові перетворювачі напруги, критичним параметром є не лише правильність обчислень, а й час реакції системи на подію. Затримка оновлення керуючого сигналу навіть на десятки мікросекунд може призвести до втрати стійкості контуру регулювання, виникнення вібрацій або зниження енергоефективності пристрою. Тому розробка методів, що мінімізують програмні затримки при роботі з периферією, є пріоритетним завданням.

Ефективність, точність та швидкість реакції цих систем безпосередньо залежить від того, наскільки повно та раціонально програмне забезпечення використовує апаратні можливості мікроконтролера, що є їх ядром.

1.2. Мікроконтролери як обчислювальне ядро вбудованих систем

Обчислювальним ядром, або «мозком», абсолютної більшості вбудованих систем, є мікроконтролер (МК). Важливо відрізняти мікроконтролер від мікропроцесора (МП), який є центральним процесорним пристроєм (CPU) у комп'ютерах загального призначення. Мікропроцесор потребує зовнішніх компонентів — оперативної пам'яті, постійної пам'яті та модулів вводу-виводу — для функціонування.

Натомість, мікроконтролер є високоінтегрованою «системою-на-кристалі» (System-on-a-Chip, SoC) (Рисунок 1).

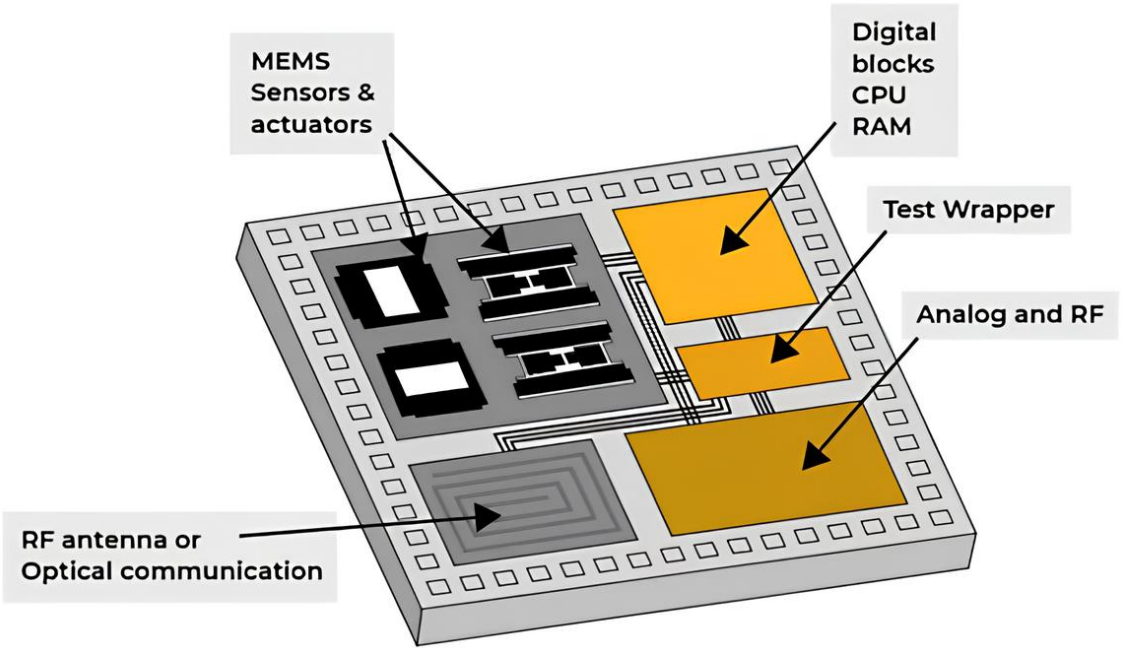

Рисунок 1.1 – Схема SoC

Він об'єднує на одній мікросхемі:

1. Процесорне ядро (наприклад, AVR, ARM, Xtensa), що виконує обчислення.
2. Блоки пам'яті, зокрема Flash-пам'ять (ПЗП) для зберігання програми та EEPROM для даних, а також оперативну пам'ять (ОЗП) для тимчасових розрахунків.
3. Набір периферійних модулів — апаратних блоків, що виконують спеціалізовані задачі.

Саме цей інтегрований набір периферійних пристроїв є ключовою перевагою мікроконтролера, що дозволяє йому безпосередньо взаємодіяти з фізичним світом. До типових периферійних модулів належать:

- Інтерфейси зв'язку (UART, SPI, I^2C) для обміну даними з іншими мікросхемами, давачами чи комп'ютером.

- Аналого-цифрові перетворювачі (АЦП) для вимірювання аналогових сигналів з давачів.
- Порти вводу-виводу загального призначення (GPIO).
- Апаратні таймери та лічильники, які відповідають за точне вимірювання часових інтервалів та генерацію сигналів керування [3].

Таким чином, мікроконтролер є автономним пристроєм, здатним зчитувати дані з давачів (через АЦП), обробляти їх (за допомогою CPU) та керувати виконавчими механізмами (наприклад, за допомогою сигналів, згенерованих таймерами). Наступним логічним кроком є розгляд одного з найпоширеніших сигналів керування, який генерується саме за допомогою апаратних таймерів.

1.3. Огляд та порівняльний аналіз мікроконтролерних платформ

Вибір апаратної платформи є фундаментальним етапом при проєктуванні будь-якої вбудованої системи, оскільки він визначає обчислювальні можливості, набір доступної периферії, енергоспоживання та кінцеву вартість продукту. Ринок мікроконтролерів пропонує тисячі варіантів, однак у сфері прототипування та розробки IoT-пристроїв чітко виділяються два основних напрямки.

Протягом тривалого часу домінуючою платформою для освітніх цілей, прототипування та аматорських розробок були плати Arduino, зокрема Arduino Uno та Arduino Mega. Їхньою основою слугують 8-бітні мікроконтролери серії ATmega компанії Atmel (зараз Microchip), такі як ATmega328P та ATmega2560 [4, 5].

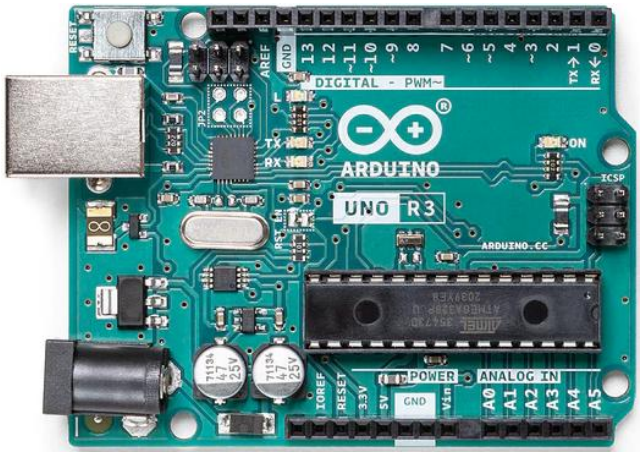

Рисунок 1.2 – Arduino UNO

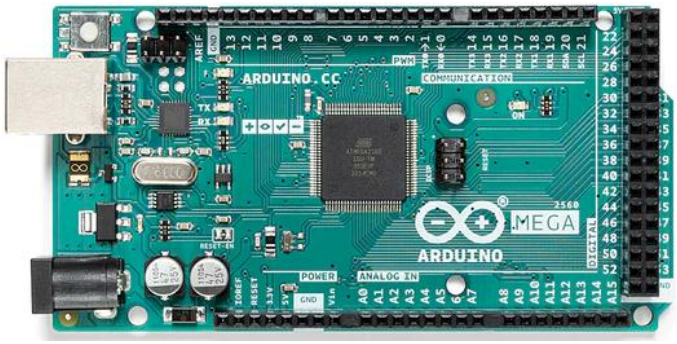

Рисунок 1.3 – Arduino MEGA

Популярність цих платформ зумовлена їхньою простотою, наявністю зручного середовища програмування (Arduino IDE) та величезною кількістю готових бібліотек і навчальних матеріалів [6]. Вони значно знизили поріг входження у світ вбудованих систем. Проте, з точки зору сучасних вимог, вони мають суттєві обмеження: низька тактова частота (зазвичай 16 МГц), малий обсяг пам'яті, відсутність вбудованих мережевих інтерфейсів та порівняно висока вартість офіційних плат.

У класичних мікроконтролерах родини ATmega реалізація широтно-імпульсної модуляції (ШІМ) ґрунтується на жорсткій прив’язці кожного апаратного таймера до строго визначених вихідних каналів і, відповідно, до

конкретних контактів (виводів) мікросхеми. Наприклад, в ATmega328P таймер Timer0 обслуговує лише виходи OC0A та OC0B (цифрові піни 6 та 5 в Arduino Uno), а Timer1 — виходи OC1A та OC1B (піни 9 та 10) [7].

Такий підхід спрощує апаратну логіку мікроконтролера, але водночас істотно обмежує гнучкість розробника при проєктуванні плати. Якщо потрібні виводи вже зайняті іншими функціями, розробник змушений або шукати компроміси, або використовувати програмну емуляцію ШІМ, яка є нестабільною і сильно завантажує процесор. Таким чином, головним недоліком класичної архітектури є апаратна фіксація зв'язку «таймер → канал → вивід».

На противагу класичному підходу, мікроконтролери нового покоління, такі як ESP32, реалізують значно більш гнучкий принцип апаратного розв’язання між таймером, каналом і GPIO-виходом.

Рисунок 1.4 – ESP32 моделі ESP-WROOM-32

Серед периферійних підсистем ESP32 особливе місце посідає LEDC (LED PWM Controller) — модуль генерації широтно-імпульсних сигналів, здатний формувати до 16 незалежних каналів PWM із підтримкою високошвидкісного (до 40 МГц) та низькошвидкісного режимів. ШІМ-модуляція є базовим інструментом для керування світлодіодами, електродвигунами, аудіо-модулями та іншими

навантаженнями, де стабільність і точність параметрів сигналу визначають якість роботи всієї системи [8].

Головна архітектурна перевага LEDC полягає у гнучкому мультиплексуванні. Тут реалізовано розділення логічного рівня управління (канал) і генератора періодичності (таймер) :

1. Таймер (наприклад, HSTIMER0) відповідає лише за генерацію базової «опорної» частоти та налаштування розрядності сигналу.
2. Канал (наприклад, HSCH0) відповідає за формування коефіцієнта заповнення (duty cycle).
3. GPIO Matrix (Матриця виводів) дозволяє вивести сигнал з будь-якого каналу на будь-який вільний цифровий вивід мікроконтролера.

Це дає змогу використовувати один таймер для синхронної роботи кількох каналів (які матимуть однакову частоту, але різний коефіцієнт заповнення) і виводити ці сигнали на зручні для розробника піни. Така архітектура забезпечує надзвичайну гнучкість при реалізації складних систем керування (наприклад, багатоканальних драйверів двигунів чи RGB-матриць) без зміни апаратної частини плати.

Варто зазначити важливу архітектурну відмінність між розглянутими платформами. На відміну від 8-бітних архітектур (AVR), де програма часто виконується безпосередньо на «голому залізі», ESP32 є складною двоядерною системою, що працює під управлінням операційної системи реального часу FreeRTOS.

З одного боку, це надає потужні можливості багатозадачності. З іншого боку, це накладає обмеження на прямий доступ до ресурсів. Стандартні драйвери змушені використовувати механізми синхронізації (спінлоки, м’ютекси, семафори) для запобігання конфліктам між ядрами. Це створює додаткові накладні витрати процесорного часу (overhead), які можуть бути надлишковими

для задач генерації простих сигналів, де пріоритетом є максимальна швидкодія [9].

1.4. Проблема програмної абстракції та аналіз існуючих методів керування LEDC

Сучасна екосистема розробки для 32-бітних мікроконтролерів, зокрема ESP32, базується на концепції шарів апаратної абстракції (HAL — Hardware Abstraction Layer). Цей підхід покликаний приховати від розробника складність архітектури, надаючи уніфіковані програмні інтерфейси (API) для роботи з периферією.

Хоча такий підхід значно прискорює процес розробки та забезпечує переносимість коду, він неминуче вносить компроміс між зручністю використання та ефективністю керування . Для задач керування ШІМ-сигналами (підсистема LEDC) основними інструментами є середовище Arduino та фреймворк ESP-IDF. Розглянемо їхні можливості та обмеження детальніше.

1.4.1. Високорівневий підхід: Arduino API та його функціональні межі

Платформа Arduino для ESP32 (зокрема версії ядра 3.x) пропонує найбільш абстрагований інтерфейс ledcAttach та ledcWrite. Цей підхід орієнтований на максимальну простоту використання, проте такий дизайн накладає жорсткі рамки на доступ до апаратних ресурсів.

Функціональні можливості:

Стандартний API надає розробнику контроль лише над трьома базовими параметрами сигналу:

1. Частота (f_{PWM}): Користувач задає бажане значення у Герцах. API автоматично розраховує дільник, намагаючись підібрати найближче можливе значення.

2. Розрядність (RES): Вказується у бітах. Це визначає дискретність керування скважністю.
3. Коефіцієнт заповнення (DUTY): Змінюється у процесі роботи функцією ledcWrite.
4. Прив’язка до піна: API дозволяє обрати будь-який GPIO, автоматично налаштовуючи матрицю комутації.

Апаратні обмеження:

Аналіз вихідного коду бібліотек Arduino та технічної документації ESP32 дозволяє виділити ряд критичних апаратних можливостей LEDC, які недоступні через стандартний API:

1. Керування фазою (HPOINT):

У регістрі каналу LEDC_HSCHn_HPOINT_REG можна задати момент початку імпульсу, що дозволяє створювати фазовий зсув між різними каналами.

- Обмеження API: Стандартні функції завжди записують у цей регістр 0.
- Наслідок: Неможливо реалізувати синхронне керування зі зсувом фаз (наприклад, для багатофазних DC-DC перетворювачів або мостових схем), де критично важливо рознести моменти вмикання силових ключів у часі.

2. Вибір таймера та синхронізація:

ESP32 має 4 незалежних таймери. Arduino API використовує алгоритм автоматичного розподілу ресурсів: функція сама шукає вільний таймер і призначає його каналу.

- Обмеження API: Розробник не може примусово вказати, який саме таймер використовувати.

- Наслідок: Ускладнюється синхронізація групи каналів. Якщо API помилково призначить пов’язані канали на різні таймери, їх синхронна робота стане неможливою.

3. Вибір джерела тактування:

Модуль може тактуватися від APB_CLK (80 МГц) або REF_TICK (1 МГц).

- Обмеження API: Джерело обирається автоматично залежно від бажаної частоти.
- Наслідок: Розробник не може примусово обрати стабільніший REF_TICK для низьких частот або гарантувати роботу від APB_CLK для мінімізації джиттеру.

4. Апаратний дізерінг (Dithering):

ESP32 підтримує режим дізерінгу для підвищення ефективної роздільної здатності ШІМ.

- Обмеження API: Цей функціонал повністю ігнорується стандартними функціями.

Отже, метод Arduino API перетворює гнучку підсистему LEDC на простий генератор прямокутних імпульсів, відсікаючи можливості, необхідні для професійних систем керування.

1.4.2. Драйверний підхід: ESP-IDF та його особливості

Офіційний фреймворк розробки від Espressif Systems (ESP-IDF) надає більш гнучкий, ніж у Arduino, інструментарій для керування периферією. Драйвер ledc працює на рівні HAL (Hardware Abstraction Layer) і забезпечує доступ до більшості функцій модуля через заповнення конфігураційних структур ledc_timer_config_t та ledc_channel_config_t [10].

Функціональні можливості:

На відміну від спрощеного підходу Arduino, драйвер ESP-IDF надає розробнику контроль над архітектурними особливостями підсистеми:

1. Вибір режиму швидкості: Розробник явно вказує режим LEDC_HIGH_SPEED_MODE або LEDC_LOW_SPEED_MODE, що визначає набір доступних таймерів та джерел тактування.
2. Ручний розподіл ресурсів: Можливість конкретного вибору номера таймера (LEDC_TIMER_0...3) та каналу, що дозволяє реалізовувати складні схеми синхронізації, прив'язуючи групу каналів до одного джерела тактування.
3. Керування перериваннями: Драйвер дозволяє налаштовувати генерацію переривань при закінченні циклу фейдингу (LEDC_INTR_FADE_END), що є корисним для створення світлових ефектів.
4. Апаратний фейдинг: Підтримка функцій автоматичної плавної зміни шпаруватості без участі процесора (hardware fading).

Системні обмеження та недоліки:

Незважаючи на розширений функціонал, використання драйвера ESP-IDF вносить значні накладні витрати, зумовлені архітектурою операційної системи FreeRTOS, під управлінням якої працює ESP32:

1. Накладні витрати на синхронізацію (Thread Safety):

Оскільки ESP32 є двоядерною системою, драйвер зобов'язаний захищати доступ до спільних апаратних регістрів від стану гонитви (race conditions). Для цього використовуються примітиви синхронізації — спінлоки (spinlocks).

- Наслідок*:* При кожному виклику функції зміни скважності (ledc_set_duty), драйвер захоплює спінлок, забороняючи перемикання контексту та переривання на даному ядрі. Це гарантує стабільність, але вносить неминучу затримку на вхід/вихід з критичної секції.

2. Складність динамічного переналаштування:

Зміна частоти або розрядності вже запущеного таймера в ESP-IDF реалізована через повний цикл перезапуску. Функція ledc_timer_config виконує зупинку таймера, скидання його лічильника, перерахунок дільників та повторну ініціалізацію.

- Наслідок*:* Як показали експериментальні дослідження (див. Розділ 3), цей процес займає значний час (понад 18 мкс), що робить неможливим використання стандартного драйвера в задачах адаптивного керування частотою (наприклад, у резонансних перетворювачах).

3. Приховування низькорівневих механізмів:

Драйвер все ще абстрагує процес розрахунку значень регістрів. Наприклад, дробова частина дільника частоти розраховується автоматично, і розробник не має прямого інструменту для її ручної корекції, якщо потрібна нестандартна прецизійна частота. Також обмежений доступ до регістру фази (HPOINT) для статичних налаштувань (без фейдингу), що ускладнює реалізацію фазового зсуву сигналів.

Отже, драйвер ESP-IDF є надійним інструментом для загальних задач, але його архітектура, орієнтована на безпеку в багатозадачному середовищі, створює бар'єр для досягнення мінімальної латентності, необхідної в системах жорсткого реального часу.

Висновки до розділу 1

У першому розділі здійснено системний аналіз сучасних підходів до проектування вбудованих систем керування на базі мікроконтролерів. Встановлено, що перехід від класичних 8-бітних архітектур (AVR) до високопродуктивних 32-бітних платформ, таких як ESP32, відкриває нові можливості завдяки гнучкій апаратній архітектурі, зокрема наявності

комутаційної матриці GPIO та потужної підсистеми генерації сигналів LEDC. Водночас виявлено, що робота під управлінням операційної системи реального часу (FreeRTOS) накладає специфічні вимоги до механізмів доступу до апаратних ресурсів.

Проведений порівняльний аналіз існуючих програмних засобів показав, що стандартні високорівневі інтерфейси (API), які пропонуються в середовищах Arduino та ESP-IDF, реалізують концепцію апаратної абстракції (HAL). Хоча цей підхід спрощує розробку, він створює суттєві обмеження для задач жорсткого реального часу. Виявлено, що універсальність функцій Arduino API призводить до приховування критично важливих параметрів налаштування (таких як вибір таймера чи фазовий зсув) та надлишкових обчислювальних витрат.

Детальне вивчення драйверного підходу ESP-IDF дозволило встановити, що навіть офіційні засоби розробки мають значні недоліки в контексті швидкодії. Забезпечення потокобезпеки у багатозадачному середовищі вимагає використання механізмів синхронізації (спінлоків), що вносить неминучі затримки при виконанні команд оновлення параметрів генерації. Крім того, складні алгоритми динамічного розрахунку дільників частоти та перевірки валідності даних перетворюють прості апаратні операції на тривалі програмні процедури.

Таким чином, сформульовано науково-технічну проблему: наявні стандартні засоби не дозволяють реалізувати потенціал швидкодії та детермінованості апаратної частини ESP32, необхідний для високодинамічних систем керування (векторне керування двигунами, цифрова генерація). Вирішенням цієї проблеми є розробка методу прямого регістрового доступу, який дозволить усунути програмні бар'єри HAL, забезпечити повний контроль над конфігурацією таймерів та мінімізувати латентність системи, що і є метою даної кваліфікаційної роботи.

РОЗДІЛ 2
АПАРАТНА АРХІТЕКТУРА ТА РЕГІСТРИ ПІДСИСТЕМИ LEDC ESP32

2.1. Контекст налаштування: активація модуля та маршрутизація сигналу

Для коректної роботи підсистеми LEDC (LED PWM Controller) недостатньо налаштувати лише її власні регістри. Як і для більшості периферійних модулів у мікроконтролері ESP32, необхідно виконати два критично важливих підготовчих кроки:

1. Активувати апаратний модуль LEDC, подавши на нього тактовий сигнал.
2. Маршрутизувати вихідний сигнал з модуля на фізичний вивід (пін) мікроконтролера.

Ці операції виконуються через спеціалізовані системні регістри, що не належать до самого модуля LEDC, а саме через DPort та GPIO Matrix [11].

2.1.1. Управління тактуванням через DPort

DPort (Digital Peripheral Registers) — це набір спеціальних системних регістрів, які, серед іншого, відповідають за керування тактуванням (клокінгом) та скиданням (reset) периферійних модулів мікроконтролера ESP32 . Для економії енергії більшість периферійних блоків, включно з LEDC, за замовчуванням вимкнені.

Для активації модуля LEDC необхідно взаємодіяти з двома регістрами DPort:

DPORT_PERIP_CLK_EN_REG – регістр увімкнення тактування периферії. Цей регістр містить набір бітових прапорів, кожен з яких вмикає або вимикає подачу тактового сигналу на відповідний периферійний модуль . Для

увімкнення модуля LEDC необхідно встановити (записати '1') біт **DPORT_LEDC_CLK_EN**.

DPORT_PERIP_RST_EN_REG – регістр увімкнення скидання периферії. Цей регістр керує станом скидання (reset) модулів. Встановлення біта утримує модуль у стані скидання. Для нормальної роботи модуль необхідно вивести зі стану скидання. Це досягається скиданням (записом '0') біта **DPORT_LEDC_RST**.

```
// 1. Увімкнути тактування для модуля LEDC
DPORT_REG_WRITE(DPORT_PERIP_CLK_EN_REG, DPORT_REG_READ(DPORT_PERIP_CLK_EN_REG) | DPORT_LEDC_CLK_EN);

// 2. Вивести модуль LEDC зі стану скидання
DPORT_REG_WRITE(DPORT_PERIP_RST_EN_REG,  DPORT_REG_READ(DPORT_PERIP_RST_EN_REG) & ~DPORT_LEDC_RST);
```

Рисунок 2.1 - Активація модуля LEDC через регістри DPort

Лише після виконання цих операцій модуль LEDC стає готовим до приймання конфігураційних команд.

2.1.2. Маршрутизація сигналу через GPIO Matrix

Наступним кроком є фізичне підключення вихідного сигналу ШІМ до піна мікроконтролера. На відміну від архітектури AVR , ESP32 використовує **GPIO Matrix** (Матрицю GPIO). Цей механізм забезпечує надзвичайну гнучкість, дозволяючи програмно «підключити» будь-який внутрішній периферійний сигнал до практично будь-якого фізичного виводу (піна) GPIO

Для виведення згенерованого ШІМ-сигналу (наприклад, ledc_hs_sig_out0) на конкретний пін (наприклад, GPIO2) необхідно налаштувати конфігураційний регістр, що відповідає цьому піну.

Ключовим регістром для цієї операції є **GPIO_FUNCn_OUT_SEL_CFG_REG**, де 'n' — номер піна. Структура цього регістру містить декілька полів, але для маршрутизації нас цікавлять два:

1. **GPIO_FUNCn_OUT_SEL** (Біти 8:0): Поле вибору джерела вихідного сигналу. Сюди необхідно записати індексний номер сигналу периферії. Кожен периферійний сигнал (наприклад, ledc_hs_sig_out0, ledc_hs_sig_out1 і т.д.) має унікальний індекс. Для ledc_hs_sig_out0 цей індекс дорівнює 71, для якого в заголовкових файлах soc/gpio_sig_map.h визначено константу **LEDC_HS_SIG_OUT0_IDX** .
2. **GPIO_FUNCn_OEN_SEL** (Біт 10): Поле вибору керування виходом (Output Enable).
    - 1: Керування виходом здійснюється через загальний регістр GPIO_ENABLE_REG.
    - 0: Керування виходом передається периферійному модулю. Для коректної роботи ШІМ необхідно, щоб сам модуль LEDC керував станом піна, тому цей біт має бути встановлений в '0'.

Практична реалізація підключення сигналу LEDC_HS_SIG_OUT0_IDX до піна GPIO2 виглядає так:

```
// 3. Підключити вихідний сигнал LEDC_HS_SIG_OUT0 на GPIO2
REG_WRITE(GPIO_FUNC2_OUT_SEL_CFG_REG, LEDC_HS_SIG_OUT0_IDX);
```

Рисунок 2.2 - Маршрутизація сигналу LEDC через GPIO Matrix

Додатково, хоча керування виходом (OEN_SEL) передано периферії, рекомендується явно вимкнути цей пін у загальному регістрі GPIO_ENABLE_REG, щоб уникнути можливих конфліктів. Це можна зробити через регістр "Write 1 To Clear" (W1TC) :

```
// 4. Вимкнути керування виходом для GPIO2 у загальному регістрі
REG_WRITE(GPIO_ENABLE_W1TC_REG, (1 << 2));
```

Рисунок 2.3 - Налаштування GPIO_ENABLE_REG

Виконавши описані кроки, досягається повна підготовка системи до роботи: модуль LEDC активовано (DPort) та його вихідний канал маршрутизовано на фізичний пін (GPIO Matrix). Наступним етапом є налаштування безпосередньо внутрішніх регістрів підсистеми LEDC для генерації ШІМ-сигналу із заданими параметрами.

2.2. Загальна архітектура підсистеми LEDC

LEDC – це апаратний ШІМ-контролер із 16 незалежними каналами, які можуть генерувати ШІМ-сигнали різної частоти та скважності. Ключовими особливостями LEDC є:

- Підтримка високошвидкісних та низькошвидкісних режимів
- Висока точність (дробовий поділ частоти)
- Фейдинг (автоматична зміна скважності)

LEDC має два логічні класи каналів:

High-Speed (HS) — 8 каналів, призначених для високошвидкісної генерації PWM (користуються таймерами HSTIMER0..3);

Low-Speed (LS) — 8 каналів, орієнтованих на низькочастотні або енергозберігаючі режими (таймери LSTIMER0..3).

У технічній документації наводиться така архітектура модуля LED_PWM:

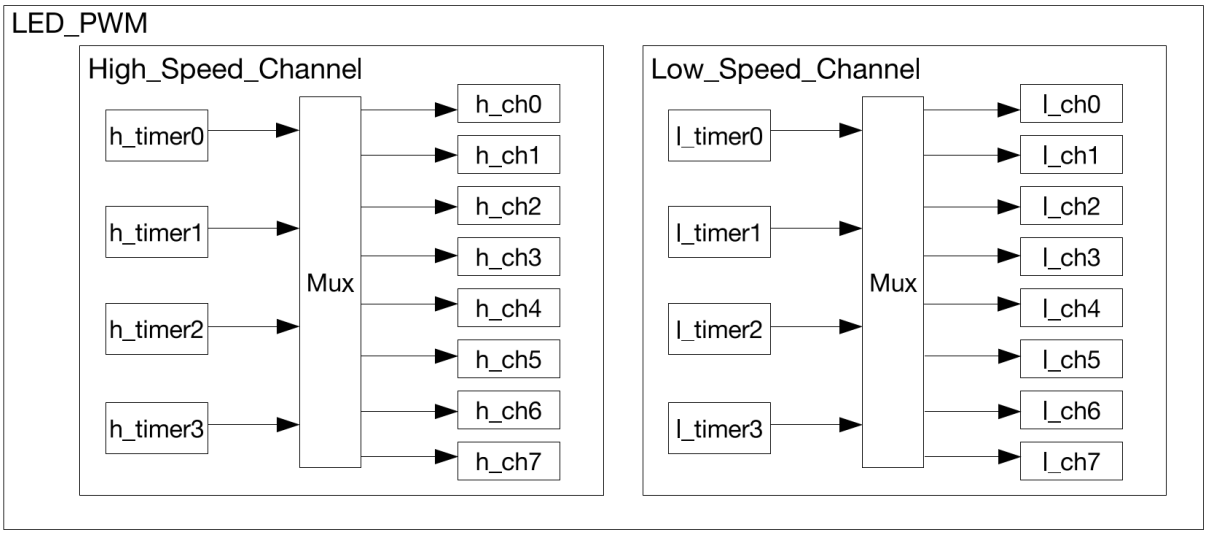

Рисунок 2.4 - LED_PWM архітектура

З якої видно, кожен вихідний канал (сигнал), наприклад h_chn, може працювати від будь-якого таймера HSTIMERx. Усі ці вихідні сигнали описані у таблиці з периферійними сигналами матриці GPIO та мають номери з 71 по 86.

2.3. Регістри LEDC

Підсистема LEDC мікроконтролера ESP32 реалізує керування каналами широтно-імпульсної модуляції через набір спеціалізованих регістрів. Саме робота на рівні цих регістрів забезпечує максимальну гнучкість налаштування параметрів сигналу та мінімізацію затримок.

2.3.1 Регістри високошвидкісних таймерів LEDC

Розглянемо структуру високошвидкісного таймера та каналу.

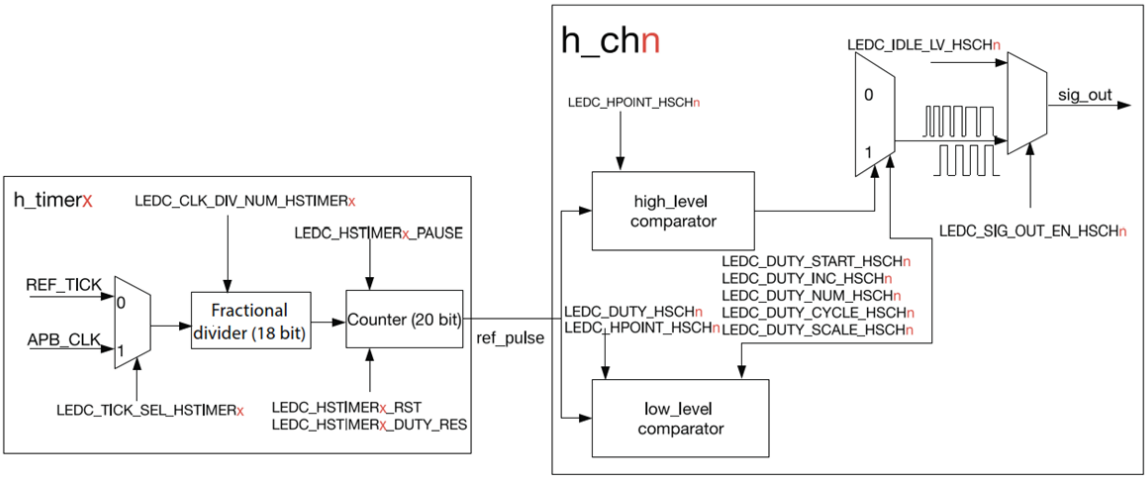

Рисунок 2.5 - LED_PWM Діаграма високошвидкісного каналу

З цієї структури видно, що потрібно налаштувати таймер h_timerx, канал h_chn та зв'язок між ними.

Регістр **LEDC_HSTIMERx_CONF_REG** відповідає за налаштування параметрів високошвидкісного таймера та має структуру:

(reserved)	LEDC_TICK_SEL_HSTIMERx	EDC_HSTIMERx_RST	LEDC_HSTIMERx_PAUSE	LEDC_CLK_DIV_NUM_HSTIMERx		LEDC_HSTIMERx_DUTY_RES
				10 біт цілої частини	8 біт дробової частини	
31 26	25	24	23	22 21 20 19 18 17 16 15 14 13	12 11 10 9 8 7 6 5	4 3 2 1 0
0 0	0	1	0	0 0 0 0 0 0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	0 0 0 0 0

Рисунок 2.6 - Структура регістру LEDC_HSTIMERx_CONF_REG

LEDC_TICK_SEL_HSTIMERx (біт 25) Цей біт використовується для вибору джерела імпульсів **APB_CLK** або **REF_TICK** для високошвидкісного таймера x.

1: **APB_CLK** (80 МГц) це основна шина периферійних пристроїв ESP32.

0: **REF_TICK** (1 МГц) це стабільне, але повільніше опорне джерело тактування.

LEDC_HSTIMERx_RST (біт 24) Цей біт використовується для скидання високошвидкісного таймера x. Значення лічильника дорівнюватиме «нулю» після скидання.

LEDC_HSTIMERx_PAUSE (біт 23) Цей біт використовується для призупинення лічильника у високошвидкісному таймері x.

LEDC_CLK_DIV_NUM_HSTIMERx (біт 22 – 5) Це поле використовується для налаштування коефіцієнта поділу для дільника у високошвидкісному таймері x.

Старші 10 біт (22–13): Визначають цілу частину дільника.

Молодші 8 біт (12–5): Визначають дробову частину дільника (з точністю до 1/256).

LEDC_HSTIMERx_DUTY_RES (біт 4 – 0) Це поле використовується для визначення розрядності лічильника (від 1 до 20 біт), тобто кількість кроків у періоді PWM у високошвидкісному таймері x.

Частота PWM обчислюється за формулою:

$$f_{PWM} = \frac{f_{CLK}}{DIV \cdot 2^{RES}} \quad (2.1)$$

де f_{CLK} – джерело такту (**APB_CLK** або **REF_TICK**),

DIV – дільник,

RES – розрядність лічильника.

Після налаштування таймера **HSTIMER0** треба налаштувати вихідний канал. Параметри окремого каналу високошвидкісного таймера x визначає регістр **LEDC_HSCHn_CONF0_REG**.

(reserved)	LEDC_IDLE_LV_HSCHn	LEDC_SIG_OUT_EN_HSCHn	LEDC_TIMER_SEL_HSCHn
31 4	3	2	1 0
0	0	0	0 0

Рисунок 2.7 - Структура регістру LEDC HSCHn_CONF0_REG

Регістр має такі поля:

LEDC_IDLE_LV_HSCHn (біт 3) – визначає, який сигнал буде на виході, коли канал неактивний.

0 — вихід в LOW (0 В).

1 — вихід в HIGH (3.3 В).

LEDC_SIG_OUT_EN_HSCHn (біт 2) — дозвіл виходу сигналу на GPIO.

0 — відключає вихід PWM (незалежно від роботи таймера).

1 — включає вихід PWM (якщо таймер запущений).

LEDC_TIMER_SEL_HSCHn (біти 0-1) — визначає, який таймер керує каналом PWM. ESP32 має 4 високошвидкісні таймери (**HSTIMER 0 – HSTIMER3**).

Наступним регістром, який потрібно розглянути, буде регістр **LEDC_HSCHn_CONF1_REG.** Регістр використовується для конфігурації моментів перемикання сигналу ШІМ у каналі n.

Він визначає, коли саме (у яких тактах таймера) відбувається зміна стану вихідного сигналу — з високого на низький і навпаки (фейдинг).

LEDC_DUTY_START_HSCHn	LEDC_DUTY_INC_HSCHn	LEDC_DUTY_NUM_HSCHn	LEDC_DUTY_CYCLE_HSCHn	LEDC_DUTY_SCALE_HSCHn
31	30	29 20	19 10	9 0
0	1	0 0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 0

Рисунок 2.8 - Структура регістру LEDC_HSCHn_CONF1_REG

Регістр має такі поля:

LEDC_DUTY_START_HSCHn (біт 31) Визначає старт зміни скважності. Коли встановлено 1, зміна скважності починається. Аппаратно скидається в 0 після старту. Поки цей біт не встановлений, зміни **LEDC_DUTY_NUM_HSCHn**, **LEDC_DUTY_CYCLE_HSCHn** та **LEDC_DUTY_SCALE_HSCHn** не набудуть чинності.

LEDC_DUTY_INC_HSCHn (біт 30) – Визначає напрям зміни скважності.

1 – збільшувати скважність (duty).

0 – зменшувати скважність.

LEDC_DUTY_NUM_HSCHn (біти 29-20) – Визначає кількість циклів збільшення або зменшення скважності. Якщо 0, зміна буде нескінченною.

LEDC_DUTY_CYCLE_HSCHn (біти 19-10) – Визначає кількість циклів таймера, протягом яких відбувається зміна duty cycle. Чим більше число, тим повільніше відбуватиметься зміна.

LEDC_DUTY_SCALE_HSCHn (біти 9-0) – Визначає крок зміни скважності. Чим більше число, тим різкішою буде зміна.

У випадку, якщо фейдинг не використовується, цей регістр часто залишається у нульовому стані — LEDC виводить сталий рівень ШІМ, заданий у **LEDC_HSCHn_DUTY_REG**

Для опису зв'язку між каналом та таймером треба спочатку розібрати, як працює ШІМ. Сигнал генерується наступним чином:

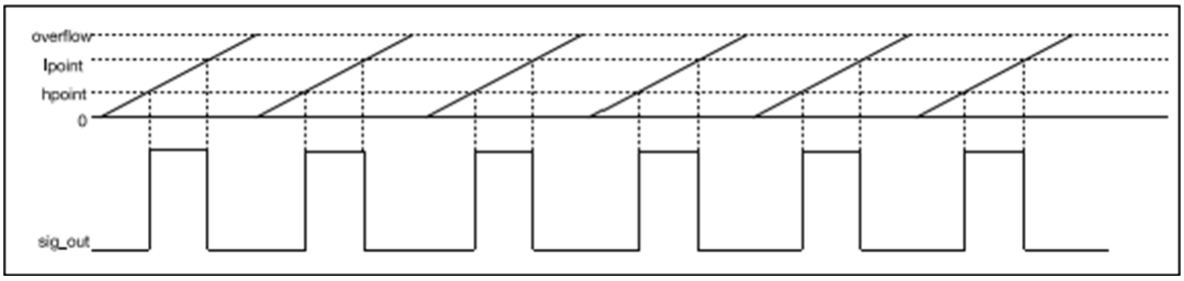

Рисунок 2.9 - Діаграма вихідного ШІМ сигналу

Кожен канал ШІМ отримує 20-бітове значення від лічильника, прив’язаного до вибраного високошвидкісного таймера. Це значення порівнюється з двома регістрами для формування сигналу:

LEDC_HPOINT_HSCHn — коли лічильник досягає цього значення, вихідний сигнал ШІМ стає **HIGH**.

LEDC_HPOINT_HSCHn + LEDC_DUTY_HSCHn[24:4] — коли лічильник досягає цієї суми, сигнал ШІМ повертається в **LOW**.

Таким чином, HPOINT визначає момент початку імпульсу, а DUTY задає тривалість HIGH рівня, формуючи потрібний коефіцієнт заповнення.

Для задання значення HPOINT використовується регістр **LEDC_HSCHn_HPOINT_REG**.

(reserved)	LEDC_HPOINT_HSCHn
31 20	19 0
0 0 0 0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

Рисунок 2.10 - Структура регістру LEDC_HSCHn_HPOINT_REG

Поле **LEDC_HPOINT_HSCHn (біт 19 - 0)** визначає умову, коли вихідний сигнал переключиться на 1 (HIGH).

Це відбувається тоді, коли значення лічильника таймера **LEDC_HSTIMERx_CNT** у регістрі **LEDC_HSTIMERx_VALUE_REG** збігається зі значенням **LEDC_HPOINT_HSCHn** у регістрі

LEDC_HSCHn_HPOINT_REG. У цей момент на відповідному виході PWM встановлюється логічна одиниця (HIGH).

Для завдання значення DUTY використовується регістр **LEDC_HSCHn_DUTY_REG**

(reserved)	LEDC_DUTY_HSCHn
31 25	24 0
0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

Рисунок 2.11 - Структура регістру LEDC_HSCHn_DUTY_REG

LEDC_DUTY_HSCHn (біт 24 - 0) визначає, наскільки довго сигнал залишається високим (HIGH) протягом одного періоду ШІМ.

Коли лічильник таймера **HSTIMERx**, який прив'язаний до каналу n, досягає значення **LEDC_LPOINT_HSCHn**, вихідний сигнал PWM встановлюється в **LOW** (0).

Час перебування сигналу у високому стані вимірюється в тактах (ticks) таймера. Значення **LEDC_LPOINT_HSCHn** обчислюється за формулою:

$$LEDC\_LPOINT\_HSCHn = (LEDC\_HPOINT\_HSCHn1 + LEDC\_DUTY\_HSCHn2) \quad (2.2)$$

або

$$LEDC\_LPOINT\_HSCHn = (LEDC\_HPOINT\_HSCHn1 + LEDC\_DUTY\_HSCHn2 + 1) \quad (2.3)$$

де $LEDC\_HPOINT\_HSCHn1 - LEDC\_HPOINT\_HSCH[19:0]$

$LEDC\_DUTY\_HSCHn2 - LEDC\_DUTY\_HSCH[24:4]$; в залежності від налаштувань.

Головне, молодші 4 біти поля DUTY не використовуються!

2.3.2 Регістри високошвидкісних таймерів LEDC

Низькошвидкісні таймери (l_timerx) на низькошвидкісного каналу відрізняються від високошвидкісних таймерів (h_timerx) двома аспектами:

1. У той час як джерелом тактування високошвидкісного таймера може бути **REF_TICK** або **APB_CLK**, низькошвидкісні таймери тактуються або від **REF_TICK**, або від **SLOW_CLOCK**. Джерелом **SLOW_CLOCK** може бути **APB_CLK** (80 МГц) або **RC_FAST_CLK** (8 МГц.), і його можна обрати за допомогою регістра **LEDC_APB_CLK_SEL**.
2. Високошвидкісні лічильник та дільник працюють без збоїв (**glitch-free**), що означає, що якщо програмне забезпечення змінює максимальне значення лічильника або дільника, оновлення набуде чинності після наступного переривання при переповненні. На противагу цьому, низькошвидкісні лічильник та дільник оновлять ці значення лише тоді, коли встановлено (біт) **LEDC_LSTIMERx_PARA_UP**.

Регістр **LEDC_LSTIMERx_CONF_REG** відповідає за налаштування параметрів низькошвидкісного таймера та має структуру:

(reserved)	LEDC_LSTIMERx_PARA_UP	LEDC_TICK_SEL_LSTIMERx	EDC_LSTIMERx_RST	LEDC_LSTIMERx_PAUSE	LEDC_CLK_DIV_NUM_LSTIMERx		LEDC_LSTIMERx_DUTY_RES
					10 біт цілої частини	8 біт дробової частини	
31 27	26	25	24	23	22 21 20 19 18 17 16 15 14 13	12 11 10 9 8 7 6 5	4 3 2 1 0
0 0	0	0	1	0	0 0 0 0 0 0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	0 0 0 0 0

Рисунок 2.12 - Структура регістру LEDC_LSTIMERx_CONF_REG.

LEDC_LSTIMERx_DUTY_RES - Цей біт визначає розрядність лічильника (від 1 до 20 біт) в низькошвидкісному таймері x.

LEDC_CLK_DIV_NUM_LSTIMERx - Цей біт використовується для налаштування коефіцієнта ділення для дільника в низькошвидкісному таймері x.

Старші 10 біт (22–13): Визначають цілу частину дільника.

Молодші 8 біт (12–5): Визначають дробову частину дільника (з точністю до 1/256).

LEDC_LSTIMERx_PAUSE - Цей біт використовується для призупинення лічильника в низькошвидкісному таймері x.

1: зупиняє лічильник у поточному стані (freeze),

0: таймер працює.

LEDC_LSTIMERx_RST - Цей біт використовується для примусового скидання низькошвидкісного таймера x у 0. Автоматично повертається у 0 після виконання.

LEDC_TICK_SEL_LSTIMERx - Цей біт використовується для вибору джерела тактового сигналу: **RTC_SLOW_CLK** або **REF_TICK** для низькошвидкісного таймера x.

1: **RTC_SLOW_CLK;**

0: **REF_TICK (1 МГц.).**

LEDC_LSTIMERx_PARA_UP – Головна відмінність від високошвидкісних таймерів – сигнал оновлення. Після зміни параметрів (**LEDC_CLK_DIV_NUM_LSTIMERx** та **LEDC_LSTIMERx_DUTY_RES**) обов'язково встановлюється в 1, щоб нові значення були застосовані. Скидається апаратно.

Після налаштування таймера **LSTIMER0** треба налаштувати вихідний канал. Параметри окремого каналу низькошвидкісного таймера x визначає регістр **LEDC_LSCHn_CONF0_REG**.

(reserved)	LEDC_PARA_UP_LSCHn	LEDC_IDLE_LV_LSCHn	LEDC_SIG_OUT_EN_LSCHn	LEDC_TIMER_SEL_LSCHn
31 5	4	3	2	1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	0	0	0	0 0

Рисунок 2.13 - Структура регістру LEDC_LSCHn_CONF0_REG

LEDC_TIMER_SEL_LSCHn – Визначає таймер для цього каналу. Існує чотири низькошвидкісні таймери, два біти використовуються для вибору одного з них для низькошвидкісного каналу n.

LEDC_SIG_OUT_EN_LSCHn – Дає дозвіл на вивід PWM-сигналу на фізичний GPIO.

0: вихід вимкнено, пін не змінює стан;

1: вихід активний, сигнал PWM передається на відповідний пін.

LEDC_IDLE_LV_LSCHn - Визначає рівень сигналу на виході, коли канал неактивний.

0: логічний 0 (LOW),

1: логічна 1 (HIGH).

LEDC_PARA_UP_LSCHn – Головна відмінність від високошвидкісних таймерів – сигнал оновлення. Після зміни параметрів регістрів **LEDC_LSCHn_HPOINT** та **LEDC_LSCHn_DUTY** потрібно встановити цей біт у 1, щоб передати нові значення з тіньових регістрів у робочі. Автоматично скидається апаратно.

Наступним регістром, який потрібно розглянути, буде регістр **LEDC_HSCHn_CONF1_REG.** Регістр використовується для конфігурації моментів перемикання сигналу ШІМ у каналі n.

Він визначає, коли саме (у яких тактах таймера) відбувається зміна стану вихідного сигналу — з високого на низький і навпаки (фейдинг).

LEDC_DUTY_START_LSCHn	LEDC_DUTY_INC_LSCHn	LEDC_DUTY_NUM_LSCHn	LEDC_DUTY_CYCLE_LSCHn	LEDC_DUTY_SCALE_LSCHn
31	30	29 20	19 10	9 0
0	1	0 0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 0

Рисунок 2.14 - Структура регістру LEDC_HSCHn_CONF1_REG

Регістр має такі поля:

LEDC_DUTY_START_LSCHn (біт 31) Визначає старт зміни скважності. Коли встановлено 1, зміна скважності починається. Аппаратно скидається в 0 після старту. Поки цей біт не встановлений, зміни **LEDC_DUTY_NUM_LSCHn**, **LEDC_DUTY_CYCLE_LSCHn** та **LEDC_DUTY_SCALE_LSCHn** не набудуть чинності.

LEDC_DUTY_INC_LSCHn (біт 30) – Визначає напрям зміни скважності.

1 – збільшувати скважність (duty).

0 – зменшувати скважність.

LEDC_DUTY_NUM_LSCHn (біти 29-20) – Визначає кількість циклів збільшення або зменшення скважності. Якщо 0, зміна буде нескінченною.

LEDC_DUTY_CYCLE_LSCHn (біти 19-10) – Визначає кількість циклів таймера, протягом яких відбувається зміна duty cycle. Чим більше число, тим повільніше відбуватиметься зміна.

LEDC_DUTY_SCALE_LSCHn (біти 9-0) – Визначає крок зміни скважності. Чим більше число, тим різкішою буде зміна.

У випадку, якщо фейдинг не використовується, цей регістр часто залишається у нульовому стані — LEDC виводить сталий рівень ШІМ, заданий у **LEDC_LSCHn_DUTY_REG**

Для задання значення HPOINT використовується регістр **LEDC_LSCHn_HPOINT_REG**.

(reserved)	LEDC_HPOINT_LSCHn
31 20	19 0
0 0 0 0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

Рисунок 2.15 - Структура регістру LEDC_LSCHn_HPOINT_REG

Поле **LEDC_LPOINT_HSCHn (біт 19 - 0)** визначає умову, коли вихідний сигнал переключиться на 1 (HIGH).

Це відбувається тоді, коли значення лічильника таймера **LEDC_LSTIMERx_CNT** у регістрі **LEDC_LSTIMERx_VALUE_REG** збігається зі значенням **LEDC_HPOINT_LSCHn** у регістрі **LEDC_LSCHn_HPOINT_REG.** У цей момент на відповідному виході PWM встановлюється логічна одиниця (HIGH).

Для завдання значення DUTY використовується регістр **LEDC_LSCHn_DUTY_REG**

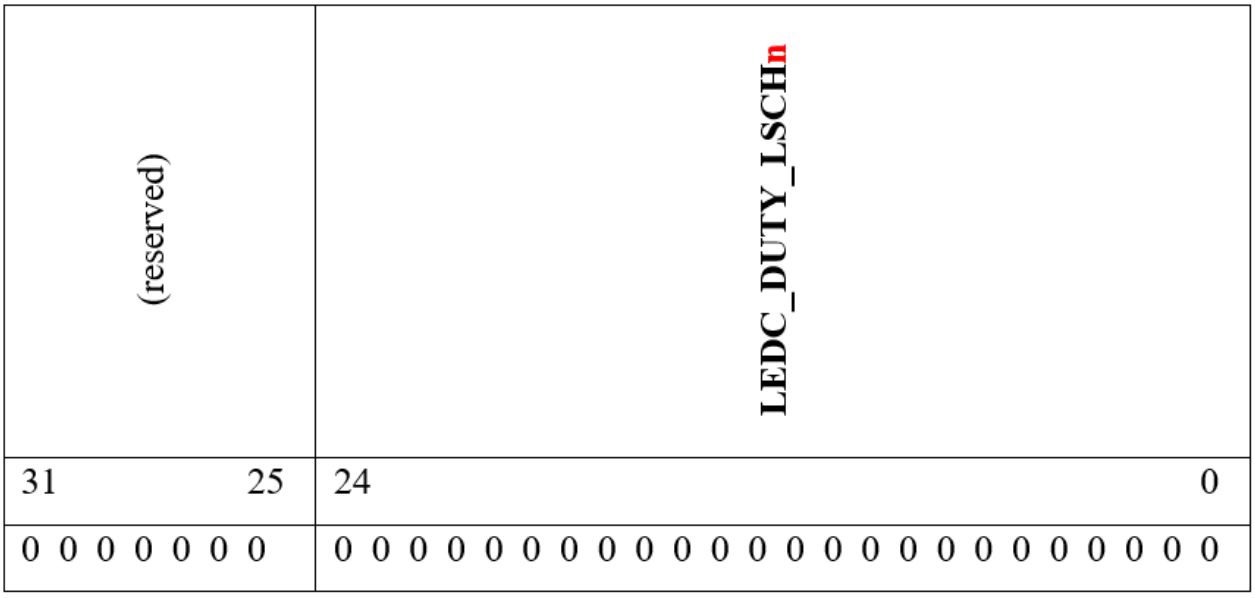

Рисунок 2.16 - Структура регістру LEDC_LSCHn_DUTY_REG

LEDC_DUTY_LSCHn (біт 24 - 0) визначає, наскільки довго сигнал залишається високим (HIGH) протягом одного періоду ШІМ.

Коли лічильник таймера **LSTIMERx**, який прив'язаний до каналу n, досягає значення **LEDC_LPOINT_LSCHn**, вихідний сигнал PWM встановлюється в **LOW** (0).

Час перебування сигналу у високому стані вимірюється в тактах (ticks) таймера. Значення **LEDC_LPOINT_HSCHn** обчислюється за формулою:

$$LEDC\_LPOINT\_LSCHn = (LEDC\_HPOINT\_LSCHn1 + LEDC\_DUTY\_LSCHn2) \quad (2.2)$$

або

$$LEDC\_LPOINT\_HSCHn = (LEDC\_HPOINT\_LSCHn1 + LEDC\_DUTY\_LSCHn2 + 1) \quad (2.3)$$

де $LEDC\_HPOINT\_LSCHn1 - LEDC\_HPOINT\_LSCH[19:0]$

$LEDC\_DUTY\_LSCHn2 - LEDC\_DUTY\_LSCH[24:4]$; в залежності від налаштувань.

Головне, молодші 4 біти поля DUTY не використовуються!

Опишемо ще один важливий регістр - **LEDC_CONF_REG.** Даний регістр використовується у випадках, коли потрібно працювати з таймером RTC (Real-Time Clock) – модуль, що споживає мало енергії та може працювати незалежно від основного процесора (CPU).

(reserved)	LEDC_APB_CLK_SEL
31 … 1	0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	0

Рисунок 2.17 - Структура регістру LEDC_CONF_REG

LEDC_APB_CLK_SEL – Цей біт використовується для встановлення частоти **RTC_SLOW_CLK**.

0: **RC_FAST_CLK** (8 МГц.)

1: **APB_CLK** (80 МГц.)

RC_FAST_CLK є частиною домену RTC (Real-Time Clock) і, за замовчуванням, вимкнений для основного «цифрового ядра» (Digital Core) та

його периферійних пристроїв, таких як LEDC. Для використання треба його увімкнути для цифрових периферійних пристроїв записавши відповідний біт у поле **RTC_CNTL_DIG_CLK8M_EN** регістру конфігурації RTC-таймерів – **RTC_CNTL_CLK_CONF_REG.**

Висновок до розділу 2

У другому розділі проведено детальний аналіз апаратної архітектури та регістрового простору підсистеми LEDC (LED PWM Controller) мікроконтролера ESP32. Встановлено, що для коректної роботи модуля необхідна попередня конфігурація системного контексту, яка включає активацію тактування через регістри DPort та налаштування маршрутизації сигналів через матрицю GPIO . Цей етап є критично важливим, оскільки за замовчуванням периферійні блоки вимкнені для економії енергії, а зв’язок між внутрішніми сигналами та фізичними виводами не встановлено .

Аналіз внутрішньої структури підсистеми показав, що LEDC розділена на дві незалежні групи — високошвидкісну та низькошвидкісну, кожна з яких містить по 4 таймери та 8 каналів . Дослідження регістрів конфігурації виявило, що ключовою перевагою архітектури є можливість довільної прив’язки будь-якого каналу до будь-якого таймера в межах своєї групи, що забезпечує значну гнучкість при проектуванні систем керування.

Детальний розгляд регістрів керування дозволив формалізувати механізми точного налаштування параметрів сигналу. Зокрема, визначено роль дільників частоти з дробовою частиною для забезпечення точності часових характеристик та важливість бітових зсувів при запису значень у регістри шпаруватості . Також встановлено критичні відмінності між режимами роботи: якщо високошвидкісні таймери оновлюють параметри автоматично, то низькошвидкісні вимагають примусового встановлення біта оновлення в регістрі конфігурації, а також мають специфічні джерела тактування .

Отримані результати формують теоретичне підґрунтя для розробки програмного методу прямого регістрового керування. Розуміння бітової структури регістрів та логіки взаємодії апаратних блоків (DPort, GPIO Matrix, LEDC) дозволяє перейти до практичної реалізації драйвера, який забезпечить максимальну швидкодію та детермінованість генерації сигналів, що є метою наступного етапу роботи.

РОЗДІЛ 3
РОЗРОБКА МЕТОДУ ТА ЕКСПЕРИМЕНТАЛЬНЕ ДОСЛІДЖЕННЯ

3.1. Постановка задачі експерименту

Як було встановлено у першому розділі роботи, використання стандартних високорівневих програмних інтерфейсів (API), таких як Arduino або ESP-IDF, створює шар абстракції (Hardware Abstraction Layer). Цей шар, спрощуючи розробку, водночас приховує апаратні можливості мікроконтролера та вносить програмні затримки , що обмежує застосування таких засобів у системах жорсткого реального часу.

Метою даного розділу є практична верифікація та комплексна оцінка ефективності методу прямого регістрового керування підсистемою LEDC мікроконтролера ESP32, теоретичні основи якого були розглянуті у другому розділі.

Для досягнення поставленої мети необхідно вирішити такі **науково-технічні задачі**:

1. Розробити алгоритм керування таймерами та каналами LEDC шляхом прямого запису конфігураційних бітових масок у регістри спеціального призначення (LEDC_HSTIMERx_CONF_REG, LEDC_HSCHn_DUTY_REG тощо), виключаючи використання стандартних бібліотечних функцій налаштування.
2. Сформувати апаратно-програмний стенд на базі налагоджувальної плати ESP32-DevKitC V4 та логічного аналізатора для об’єктивної фіксації параметрів вихідних сигналів.
3. Визначити набір тестових конфігурацій, які охоплюють різні режими роботи підсистеми LEDC, варіюючи розрядність лічильника (RES), коефіцієнт поділу частоти (DIV) та коефіцієнт заповнення (DUTY).

4. Виконати теоретичний розрахунок еталонних часових характеристик (частоти f_{PWM}, періоду T_{PWM} та тривалості імпульсу t_H) для кожної конфігурації на основі математичних залежностей, виведених у другому розділі
5. Провести серію експериментальних вимірювань точності генерації сигналів та здійснити порівняльний аналіз отриманих практичних даних з теоретичними розрахунками для оцінки абсолютної та відносної похибки методу.
6. Дослідити часові характеристики (латентність) розробленого методу, вимірявши час повної ініціалізації периферії та час оновлення параметрів сигналу, і порівняти ці показники з аналогічними для методів Arduino API та ESP-IDF Driver.

Вирішення цих задач дозволить експериментально підтвердити гіпотезу про те, що прямий регістровий доступ забезпечує не лише необхідну точність формування сигналів, але й суттєву перевагу у швидкодії, що є критичним для високодинамічних систем керування.

3.2. Методика та налаштування експериментального стенду

Для проведення експериментального дослідження та практичної перевірки розробленого методу регістрового керування був сформований апаратно-програмний стенд. Методика дослідження передбачає генерацію ШІМ-сигналів мікроконтролером з наперед заданими параметрами, їх захоплення за допомогою зовнішнього вимірювального пристрою та подальший аналіз часових характеристик для оцінки точності.

3.2.1. Апаратне забезпечення

Апаратна частина експериментального стенду складається з двох ключових компонентів: пристрою, що генерує сигнал, та пристрою, що його вимірює.

В якості обчислювального ядра та джерела ШІМ-сигналів у дослідженні використовується налагоджувальна плата ESP32-DevKitC V4, що побудована на базі модуля ESP32-WROOM-32D.

Рисунок 3.1 - ESP32 моделі Wroom-32D

Ця платформа була обрана як об'єкт дослідження завдяки поєднанню високих технічних характеристик та економічної доцільності. Її основою є двоядерний 32-бітний процесор Tensilica Xtensa LX6, що працює на тактовій частоті до 240 МГц. Обчислювальні ядра мають розвинуту систему енергозбереження, що дозволяє динамічно змінювати продуктивність відповідно до потреб застосунку.

Для фіксації та вимірювання часових параметрів вихідних ШІМ-сигналів використовується 8-канальний логічний аналізатор з максимальною частотою дискретизації 24 МГц.

Рисунок 3.2 - Логічний аналізатор

Цей пристрій підключається до вихідного GPIO-піна мікроконтролера і дозволяє з високою точністю захоплювати цифрові сигнали. Використання логічного аналізатора є необхідною умовою для об'єктивної оцінки реальних часових характеристик згенерованих сигналів, оскільки він забезпечує незалежне зовнішнє вимірювання періоду, частоти та тривалості імпульсу для їх подальшого порівняння з теоретичними розрахунками.

3.2.2. Програмне забезпечення

Програмний комплекс, що забезпечує проведення експерименту, складається з трьох основних компонентів: середовища розробки, програми для захоплення сигналів та програми для їх аналізу.

Середовище розробки та прошивка МК

Для написання, компіляції та завантаження програмного коду в мікроконтролер ESP32 було обрано інтегроване середовище розробки (IDE)

Arduino IDE. Для коректної роботи з платою ESP32-DevKitC V4, у середовище було додатково встановлено набір плат «esp32» від Espressif Systems через «Boards Manager». Оскільки мікросхема USB-to-UART на платі (CP2102) вимагає специфічних драйверів, для забезпечення стабільного зв'язку з ПК було також інстальовано драйвери CP210x VCP (Virtual COM Port) [12].

Ключовою особливістю програмної реалізації є відмова від використання стандартних високорівневих функцій Arduino (таких як ledcAttach або ledcWrite). Натомість, для реалізації методу прямого доступу, програмний код безпосередньо взаємодіяв з апаратурою шляхом підключення офіційних заголовкових файлів ESP-IDF (Espressif IoT Development Framework). Зокрема, були використані бібліотеки soc/ledc_reg.h (для доступу до регістрів LEDC), soc/dport_reg.h (для керування тактуванням), soc/io_mux_reg.h та soc/gpio_sig_map.h (для налаштування GPIO Matrix) . Такий підхід дозволяє здійснювати прямий запис конфігураційних бітів у апаратні регістри, що є основою даного дослідження.

Програма захоплення сигналів

Для роботи з логічним аналізатором, візуалізації захоплених цифрових сигналів та їх експорту використовувалася програма «Logic 2.4» від компанії Saleae [13]. Для кожної тестової конфігурації вихідний сигнал з піна захоплювався протягом 0.5 секунд. Цей проміжок часу є достатнім для накопичення великої кількості періодів сигналу, що дозволяє отримати статистично достовірні середні значення. Після захоплення, необроблені дані (часові мітки та стани) експортувалися у вигляді файлу з кома-роздільниками (.csv) для подальшої математичної обробки.

Програма аналізу даних

Для фінальної обробки .csv-файлів, отриманих з програми Logic 2.4, була розроблена спеціалізована програма мовою Python [14]. Ця програма автоматизує

процес аналізу та розраховує всі необхідні параметри сигналу на основі масиву семплів.

Функціонал програми включає:

1. Завантаження .csv файлу.
2. Оптимізований пошук усіх точок перемикання сигналу (фронтів).
3. Обчислення масиву тривалостей кожного періоду (на основі наростаючих фронтів).
4. Обчислення масиву тривалостей кожного високого рівня (high time).
5. Розрахунок середніх значень для Періоду, Частоти та Часу високого рівня.
6. Форматований вивід результатів для їх подальшого внесення у порівняльні таблиці.

Використання скрипту на Python дозволило уникнути ручних вимірювань окремих імпульсів у графічному інтерфейсі, забезпечивши високу точність та повторюваність аналізу даних. Лістинг програми наведено у Додатку Г.

3.3. Програмна реалізація методу прямого регістрового керування

Для реалізації запропонованого методу було розроблено спеціалізоване програмне забезпечення. Ключовою відмінністю від стандартних підходів є відмова від використання високорівневих функцій API та пряма взаємодія з адресною простором периферії. Алгоритм налаштування має певні відмінності для високошвидкісного (High Speed) та низькошвидкісного (Low Speed) режимів, що зумовлено архітектурними особливостями модуля LEDC.

3.3.1. Алгоритм налаштування високошвидкісних таймерів (HSTIMER)

Високошвидкісний режим призначений для генерації сигналів високої частоти та використовує тактування від системної шини APB_CLK (80 МГц) або джерела REF_TICK . Структура програмного алгоритму для цього режиму реалізована у функції setup() і складається з п'яти кроків.

Крок 1. Активація модуля (DPort):

Насамперед виконується подача тактового сигналу на периферійний модуль LEDC. Це досягається шляхом читання-модифікації-запису регістру DPORT_PERIP_CLK_EN_REG для встановлення біта DPORT_LEDC_CLK_EN. Одночасно модуль виводиться зі стану скидання (Reset) шляхом запису в DPORT_PERIP_RST_EN_REG.

```
// Enable LEDC clocking
DPORT_REG_WRITE(DPORT_PERIP_CLK_EN_REG, DPORT_REG_READ(DPORT_PERIP_CLK_EN_REG) | DPORT_LEDC_CLK_EN);
DPORT_REG_WRITE(DPORT_PERIP_RST_EN_REG, DPORT_REG_READ(DPORT_PERIP_RST_EN_REG) & ~DPORT_LEDC_RST);
```

Рисунок 3.3 - Активація модуля LEDC (DPort)

Крок 2. Маршрутизація сигналу (GPIO Matrix):

Програмно «підключається» вихідний сигнал внутрішнього каналу HSCH0 до фізичного піна GPIO2. Це виконується шляхом запису індексу сигналу (константи LEDC_HS_SIG_OUT0_IDX, що дорівнює 71) у регістр GPIO_FUNC2_OUT_SEL_CFG_REG. Додатково, керування виходом (Output Enable) для GPIO2 у загальному регістрі GPIO_ENABLE_REG вимикається, щоб уникнути конфліктів.

```
//Connect the LEDC_HS_SIG_OUT0 output signal to GPIO2
REG_WRITE(GPIO_FUNC2_OUT_SEL_CFG_REG, LEDC_HS_SIG_OUT0_IDX);//LEDC_HS_SIG_OUT0_IDX = 71 = 0x47
REG_WRITE(GPIO_ENABLE_W1TC_REG, (1 << 2));
```

Рисунок 3.4 – Налаштування GPIO Matrix

Крок 3. Конфігурація таймера HSTIMER0:

Усі базові параметри таймера (джерело тактування, дільник та розрядність) «упаковуються» в одне 32-бітне слово і записуються у регістр LEDC_HSTIMER0_CONF_REG. Наприклад, для конфігурації з джерелом APB_CLK (80 МГц), дільником DIV = 4 та розрядністю RES = 6 біт, у регістр записується комбінація (1<<25)|(4<<13)|(6).

```
//(1<<25) on APB_CLK (80 MHz); (4<<13) integer divider = 4; (6) set PWM to 6 bits
REG_WRITE (LEDC_HSTIMER0_CONF_REG, ((1<<25)|(4<<13)|(6)));
```

Рисунок 3.5 – Конфігурація таймера HSTIMER0

Крок 4. Конфігурація каналу HSCH0:

Налаштовуються параметри безпосередньо вихідного сигналу:

- Регістр **LEDC_HSCH0_CONF0_REG**: Встановлюється біт LEDC_SIG_OUT_EN_HSCHn («Enable»), а біти LEDC_TIMER_SEL_HSCHn залишаються нульовими, що автоматично прив'язує канал HSCH0 до таймера HSTIMER0.
- Регістр **LEDC_HSCH0_HPOINT_REG**: Записується 0 для встановлення нульового фазового зсуву (сигнал починається одразу при скиданні лічильника).
- Регістр **LEDC_HSCH0_DUTY_REG**: Записується значення коефіцієнта заповнення DUTY. Критично важливо, що, згідно з

технічною документацією, значення DUTY необхідно зсунути на 4 біти вліво (DUTY << 4), оскільки молодші 4 біти цього регістру не використовуються при розрахунку точки перемикання.

```
//(1<<2) = 4 sets the ENable enable bit; bits 0 and 1 equal results -> use htimer0
REG_WRITE (LEDC_HSCH0_CONF0_REG, (1<<2));

REG_WRITE (LEDC_HSCH0_HPOINT_REG, 0); // the initial phase is zero
REG_WRITE (LEDC_HSCH0_DUTY_REG, (10<<4)); //PWM parameter 10 of 64
```

Рисунок 3.6 – Конфігурація каналу HSCH0

Крок 5. Запуск таймера та оновлення каналу.

Це фінальний етап, що активує генерацію.

- Спочатку скидається біт LEDC_HSTIMERx_PAUSE у регістрі LEDC_HSTIMER0_CONF_REG, що фізично запускає лічильник таймера.
- Потім встановлюється біт LEDC_DUTY_START_HSCHn у регістрі LEDC_HSCH0_CONF1_REG. Ця дія включає оновлення DUTY, навіть якщо ми не використовуємо фейдинг

```
//It is very important to start the timer by writing 0 to the pause bit
REG_WRITE(LEDC_HSTIMER0_CONF_REG,REG_READ(LEDC_HSTIMER0_CONF_REG)&~(1<<23));
REG_WRITE (LEDC_HSCH0_CONF1_REG, (1 << 31)); //Start updating the duty cycle even though we don't use automatic fade
```

Рисунок 3.7 – Запуск таймера та оновлення каналу

3.3.2. Алгоритм налаштування низькошвидкісних таймерів (LSTIMER)

Низькошвидкісний режим відрізняється використанням інших джерел тактування (REF_TICK або RTC_SLOW_CLK) та специфічним механізмом оновлення параметрів .

Алгоритм налаштування має такі відмінності:

1. **Джерело тактування:** У регістрі LEDC_LSTIMERx_CONF_REG біт LEDC_TICK_SEL_LSTIMERx вибирає між REF_TICK (0) та RTC_SLOW_CLK (1) .
2. **Механізм оновлення (Update):** На відміну від високошвидкісних таймерів, де оновлення дільника та розрядності відбувається автоматично при переповненні, у низькошвидкісному режимі для застосування нових параметрів таймера необхідно вручну встановити біт LEDC_LSTIMERx_PARA_UP.

Програмна реалізація для таймера LSTIMER0 та каналу LSCH0 виглядає наступним чином:

```
// Set the parameters: Source (Bit 25), Divider, Bit Size
// IMPORTANT: Set bit 26 (LEDC_LSTIMERx_PARA_UP) to apply the changes!
REG_WRITE(LEDC_LSTIMER0_CONF_REG, ((1<<26) | (1<<25) | (DIV<<13) | (RES)));

// 2. LSCH0 channel configuration (similar to HS, but different register addresses)
// The registers have an offset corresponding to the low-speed group
REG_WRITE(LEDC_LSCH0_CONF0_REG, (1<<2)); // Enable, Timer0
REG_WRITE(LEDC_LSCH0_HPOINT_REG, 0);
REG_WRITE(LEDC_LSCH0_DUTY_REG, (DUTY<<4));

// 3. Start and update
// Unpause the timer
REG_WRITE(LEDC_LSTIMER0_CONF_REG, REG_READ(LEDC_LSTIMER0_CONF_REG) & ~(1<<23));
// Update channel parameters (Duty)
REG_WRITE(LEDC_LSCH0_CONF1_REG, (1 << 31));
```

Рисунок 3.8 – Особливості налаштування низькошвидкісного таймера

Таким чином, розроблений програмний метод є універсальним і дозволяє керувати обома групами таймерів, враховуючи при цьому архітектурну вимогу щодо примусового оновлення параметрів для низькошвидкісної підсистеми.

3.4. Дослідження точності генерації сигналів (Accuracy Test)

Першим етапом експериментального дослідження стала перевірка здатності розробленого методу формувати сигнали, параметри яких відповідають теоретичним розрахункам. Це необхідно для підтвердження того, що ручне керування регістрами забезпечує коректну роботу таймерів у всьому робочому діапазоні.

3.4.1. Параметри, що вимірюються, та розрахункові формули

Під час проведення експериментального дослідження, аналізу та оцінці похибки підлягали три ключові часові характеристики ШІМ-сигналу. Для кожної тестової конфігурації спочатку обчислювались теоретичні (еталонні) значення цих параметрів.

1. **Частота сигналу (f_{PWM}).** Теоретична частота ШІМ-сигналу визначається параметрами, що були записані у регістр LEDC_HSTIMERx_CONF_REG, а саме: джерелом тактування, дільником та розрядністю лічильника. Вона розраховується за формулою (2.1):

$$f_{PWM} = \frac{f_{CLK}}{DIV \cdot 2^{RES}} \tag{2.1}$$

- f_{CRK} – частота обраного джерела тактування (у даному дослідженні APB_CLK, що дорівнює 80 МГц);
- DIV – 18-бітне значення дільника (ціла та дробова частина);
- RES – значення розрядності лічильника.

2. **Період сигналу (T_{PWM}).** Період сигналу є величиною, оберненою до його частоти. Він розраховується за формулою (3.1):

$$T_{PWM} = \frac{1}{f_{PWM}} \tag{3.1}$$

3. **Тривалість високого рівня (t_H).** Тривалість, протягом якої сигнал знаходиться у високому логічному стані, безпосередньо залежить від значення, записаного у регістр LEDC_HSCHn_DUTY_REG, а також від розрядності лічильника та періоду сигналу. Розрахунок ведеться за формулою (3.2):

$$t_H = \frac{DUTY}{2^{RES}} * T_{PWM} \quad (3.2)$$

де:

- DUTY – 25-бітне значення коефіцієнта заповнення;
- RES – розрядність лічильника;
- T_{PWM} – період сигналу.

Практичні значення цих параметрів вимірювались безпосередньо на виході пінів мікроконтролера за допомогою логічного аналізатора та програмного забезпечення Logic 2. Подальша обробка даних та розрахунок середніх значень проводились за допомогою програми на Python. Вихідний код програми представлений у Додатку Д.

3.4.2. Процес тестування

Для перевірки точності було сформовано масив з 18 тестових конфігурацій, що охоплюють різні режими роботи підсистеми LEDC у високошвидкісному режимі. Змінюваними параметрами виступали:

- Розрядність (RES): 4, 5 та 6 біт.
- Дільник (DIV): 1, 2 та 4.
- Коефіцієнт заповнення (DUTY): Значення, що відповідають різним рівням заповнення (фіксований крок, ~33%, 50%).

Процес тестування для кожної конфігурації виглядав наступним чином:

1. У програмному коді налаштовувався один таймер (HSTIMER0) та два вихідних канали (HSCH0, HSCH1) для генерації сигналів з різними значеннями DUTY, що дозволило оптимізувати час збору даних.
2. Після завантаження програми у мікроконтролер, до вихідних пінів GPIO2 та GPIO4 підключався логічний аналізатор.
3. За допомогою ПЗ Logic 2 проводилося захоплення сигналу протягом 0,5 секунд.
4. Отриманий масив даних експортувався у файл формату .csv.
5. Спеціалізована програма на Python обробляла файл, вираховуючи середні практичні значення усіх необхідних показників (частоти, періоду та тривалість високого сигналу)
6. Порівнювала їх з теоретичними.

3.4.3. Аналіз результатів

Повні результати вимірювань для всіх 18 тестових конфігурацій, що включають розраховані теоретичні значення, отримані практичні дані та обчислені похибки, наведені у таблиці (див. Додаток Е). Аналіз отриманих результатів дозволяє зробити наступні висновки:

1. **Стабільність частотних характеристик:** У ході всіх експериментів зафіксовано стабільне систематичне відхилення практичної частоти генерації від теоретично розрахованої на рівні -0,03%. Аналогічне, але дзеркальне відхилення (+0,03%) спостерігається і для періоду сигналу. Оскільки величина похибки залишається незмінною незалежно від обраного дільника (DIV) чи розрядності (RES), можна стверджувати, що вона зумовлена фізичною похибкою (допуском) кварцового резонатора конкретного екземпляра налагоджувальної плати ESP32-DevKitC, а не недоліками програмного алгоритму розрахунку дільників.

2. **Точність формування імпульсу (t_H):** Відносна похибка тривалості високого рівня (параметр, що відповідає за коефіцієнт заповнення) для всього масиву досліджуваних конфігурацій знаходиться в межах від -0,01% до -0,57%. Такий низький рівень похибки свідчить про високу точність методу при налаштуванні скважності сигналу у всьому робочому діапазоні частот.

Висновок: Результати експерименту підтверджують, що метод прямого регістрового доступу забезпечує стабільне та детерміноване формування ШІМ-сигналів. Отримані часові характеристики повністю відповідають теоретичним моделям з урахуванням апаратної похибки тактового генератора.

3.5. Дослідження швидкодії (Latency Test)

Другий етап експериментальних досліджень був спрямований на оцінку динамічних характеристик розробленого методу. У системах автоматичного керування (САК) та цифрової обробки сигналів (DSP) критично важливим параметром є не лише точність, а й час, який витрачає мікроконтролер на зміну параметрів генерації (латентність).

3.5.1. Класифікація тестів швидкодії

Для комплексної оцінки ефективності методів керування було проведено три типи тестів, що моделюють різні сценарії використання ШІМ у реальних пристроях:

Тест №1: **Швидкість первинної ініціалізації (Setup Latency).** Цей тест вимірює час повного налаштування каналу з «нуля». Він включає повний цикл налаштування: активацію тактування периферії, маршрутизацію сигналу через матрицю GPIO, конфігурацію таймера (частота, розрядність) та запуск генерації. Цей параметр важливий для систем, що потребують швидкого запуску після увімкнення живлення.

Тест №2: Швидкість повного переналаштування (Full Reconfiguration Latency). Вимірюється час, необхідний для зміни всіх параметрів генерації «на льоту» (частоти, розрядності та скважності). Сценарій передбачає перемикання між двома режимами з різною частотою, розрядністю та коефіцієнтом заповнення (наприклад, перехід з 5 кГц/8 біт на 10 кГц/7 біт). Цей тест є найбільш навантажувальним, оскільки вимагає перерахунку дільників таймера.

Тест №3: Швидкість оновлення коефіцієнта заповнення (Duty Update Latency). Вимірюється час виконання команди зміни скважності сигналу (DUTY) без зміни частотних характеристик. Це найбільш часто виконувана операція в замкнутих контурах регулювання (наприклад, PID-регуляторах), де значення виходу оновлюється сотні або тисячі разів на секунду.

3.5.2. Методика проведення вимірювань

Для забезпечення високої точності вимірювання часових інтервалів (порядку наносекунд) та об’єктивного порівняння ефективності було застосовано методику «тригерного піна» (Trigger Pin) з апаратною фіксацією подій.

У всіх тестах порівнювалися три методи керування, що представляють різні рівні абстракції:

1. **High-Level (Arduino API):** Використання функцій ledcAttach (для v3.0+) та ledcWrite
2. **Mid-Level (ESP-IDF Driver):** Використання функцій драйвера ledc_timer_config, ledc_channel_config, ledc_set_duty та ledc_update_duty
3. **Low-Level (Регістровий доступ):** Прямий запис у регістри DPORT, GPIO та LEDC без використання проміжних функцій

Для мінімізації впливу вимірювального коду на результати, керування тригерним піном (GPIO4) здійснювалося виключно через прямий запис у регістри GPIO_OUT_W1TS_REG (встановлення одиниці) та

GPIO_OUT_W1TC_REG (скидання в нуль). Це забезпечує виконання маркера «Старт/Стоп» за мінімальну кількість тактів процесора (на відміну від повільної функції digitalWrite) .

Вимірювання проводилися для трьох сценаріїв:

1. Методика вимірювання швидкості ініціалізації (Test 1: Setup Latency)

Цей тест імітує «холодний старт» системи. Вимірювався проміжок часу, необхідний мікроконтролеру для повного налаштування периферії з нуля до моменту появи сигналу.

Алгоритм тесту

1. У функції setup() виконується попередня підготовка тригерного піна (встановлення в LOW).
2. Безпосередньо перед початком блоку налаштування формується фронт на тригерному піні шляхом запису в регістр GPIO_OUT_W1TS_REG.
3. Виконується код ініціалізації обраним методом.
4. Логічний аналізатор фіксує часовий інтервал Δt між наростаючим фронтом на тригерному піні (початок виконання) та першим наростаючим фронтом на виході ШІМ (початок генерації).

2. Методика вимірювання швидкості повного переналаштування (Test 2: Full Reconfiguration)

Цей тест є найбільш навантажувальним і моделює ситуацію, коли необхідно «на льоту» змінити не тільки шпаруватість, але й частоту та розрядність сигналу (наприклад, адаптивна зміна частоти комутації).

Алгоритм тесту

1. У циклі loop() реалізовано перемикання між двома принципово різними режимами роботи:

- **Режим A:** Частота 5 кГц, Розрядність 8 біт, Duty 50%.
- **Режим B:** Частота 10 кГц, Розрядність 7 біт, Duty 25%.

2. Зміна бітності та частоти вимагає перерахунку дільників таймера та переконфігурації апаратної частини.
3. Аналогічно до попереднього тесту, вимірюється ширина імпульсу на тригерному піні, яка охоплює час виконання всіх команд, необхідних для переходу з Режиму A в Режим B (оновлення конфігурації таймера + оновлення конфігурації каналу).

Такий підхід дозволив отримати точні дані про латентність кожного методу в статичному (ініціалізація) та динамічному (оновлення) режимах роботи.

3. Методика вимірювання швидкості оновлення скважності (Test 3: Duty Update)

Цей тест імітує роботу в замкнутому контурі керування, де частота та розрядність залишаються незмінними, а змінюється лише коефіцієнт заповнення.

Алгоритм тесту

1. Повна ініціалізація ШІМ виконується заздалегідь (до початку вимірювань).
2. У циклі loop() відбувається періодична зміна значення duty (перемикання між значеннями 64 та 192 для 8-бітного режиму).
3. Перед викликом функції оновлення (ledcWrite, ledc_set_duty або запис у DUTY_REG) тригерний пін встановлюється у HIGH.
4. Одразу після повернення з функції оновлення тригерний пін скидається у LOW.

5. Логічний аналізатор вимірює ширину сформованого імпульсу на тригерному піні, що відповідає чистому часу виконання команди процесором.

3.5.3. Аналіз результатів

Результати вимірювань часових інтервалів для трьох досліджуваних сценаріїв зведені у табл. 3.1.

Таблиця 3.1

Результати дослідження латентності

Тип тесту	Метод керування	Середній час (Δt)	Прискорення (відносно Arduino)
1. Ініціалізація (Setup)	Arduino API	438,00 мкс	1.0x
	ESP-IDF Driver	380,00 мкс	1.15x
	Регістровий доступ	9,29 мкс	47.1x
2. Повне оновлення (Full Reconfig)	Arduino API	5,56 мкс	1.0x
	ESP-IDF Driver	18,29 мкс	0.3x (сповільнення)
	Регістровий доступ	0,208 мкс	26.7x
3. Оновлення Duty (Duty Only)	Arduino API	4,46 мкс	1.0x
	ESP-IDF Driver	4,06 мкс	1.1x
	Регістровий доступ	0,167 мкс	26.7x

Детальний аналіз отриманих результатів:

1. **Аналіз швидкості ініціалізації (Test 1):** У тесті «холодного старту» розроблений метод продемонстрував найбільш суттєву перевагу, скоротивши час запуску каналу з 438 мкс (Arduino API) до 9,29 мкс. – прискорення у 47 разів.

- Імовірна причина*:* Стандартні бібліотеки витрачають левову частку часу на динамічний розподіл пам'яті під структури драйвера, пошук вільного таймера/каналу та математичний розрахунок дільників частоти з використанням операцій з плаваючою комою. Регістровий метод оперує попередньо обчисленими константами, що усуває ці накладні витрати.

2. **Аналіз швидкості повного переналаштування (Test 2):** Цей тест виявив цікаву особливість роботи драйвера ESP-IDF. При необхідності змінити частоту та розрядність таймера «на льоту», драйвер ESP-IDF витрачає 18,29 мкс , що виявилося навіть повільнішим за високорівневий Arduino API (5,56 мкс).

   - Імовірна причина*:* Офіційний драйвер реалізує складний механізм безпечної зупинки таймера, скидання його лічильника та повторної ініціалізації для запобігання апаратним конфліктам у багатопотоковому середовищі.

   - Ефективність регістрового методу*:* Прямий запис у конфігураційний регістр таймера виконується всього за 208 наносекунд. Це свідчить про те, що апаратура ESP32 здатна змінювати частоту генерації практично миттєво, без необхідності виконання складних програмних процедур зупинки/запуску.

3. **Аналіз швидкості оновлення шпаруватості (Test 3):** Операція зміни коефіцієнта заповнення (Duty Cycle) є найбільш критичною для систем автоматичного регулювання.

   - Стандартні методи (Arduino та ESP-IDF) демонструють близькі результати: 4,46 мкс та 4,06 мкс відповідно. Основний час тут витрачається на пошук адреси регістра каналу за його номером та забезпечення атомарності доступу.

- Регістровий метод виконує цю операцію за 167 наносекунд, що у 26 разів швидше. Така швидкодія дозволяє використовувати мікроконтролер у надшвидких контурах зворотного зв'язку з частотою дискретизації, що перевищує 1 МГц, залишаючи при цьому обчислювальні ресурси процесора для реалізації алгоритмів керування.

Висновок до розділу 3

У третьому розділі проведено комплексне експериментальне дослідження розробленого методу прямого регістрового керування підсистемою LEDC мікроконтролера ESP32. Для цього було створено апаратно-програмний стенд та розроблено спеціалізоване програмне забезпечення для автоматизованого аналізу сигналів.

Основні результати дослідження полягають у наступному:

1. Підтверджено точність генерації. На основі аналізу 18 тестових конфігурацій встановлено, що метод забезпечує формування сигналів з високою точністю. Фіксоване відхилення частоти на рівні -0,03% є систематичним та зумовлене апаратною похибкою кварцового резонатора, а не програмним алгоритмом. Похибка формування коефіцієнта заповнення для більшості режимів не перевищує 0,6%.
2. Доведено перевагу у швидкодії (латентності). Порівняльний аналіз показав, що прямий регістровий доступ дозволяє скоротити час ініціалізації периферії у 47 разів (до 9,29 мкс) порівняно зі стандартним Arduino API.
3. Визначено ефективність у динамічних режимах. Час виконання команди оновлення шпаруватості (Duty Cycle) дорівнює 167 нс, що у 26 разів швидше за стандартні методи. Це дозволяє використовувати мікроконтролер у високошвидкісних контурах керування з частотою дискретизації понад 1 МГц.

4. Виявлено обмеження стандартних драйверів. Експериментально доведено, що драйвер ESP-IDF має значні накладні витрати (до 18,29 мкс) при повному переналаштуванні таймера, що пов’язано з механізмами захисту ресурсів операційної системи FreeRTOS. Розроблений метод позбавлений цих затримок (0,208 мкс).

Таким чином, експериментально підтверджено, що запропонований метод є ефективним інструментом для розробки вбудованих систем жорсткого реального часу, забезпечуючи поєднання високої точності, детермінованості та мінімальної латентності.

ВИСНОВКИ

У кваліфікаційній роботі вирішено актуальне науково-прикладне завдання підвищення ефективності керування підсистемою широтно-імпульсної модуляції (LEDC) мікроконтролера ESP32. Шляхом системного аналізу встановлено, що використання стандартних високорівневих програмних інтерфейсів (Arduino API, ESP-IDF) створює надлишковий шар абстракції, який вносить неконтрольовані затримки та обмежує швидкодію системи, що є критичним недоліком для задач жорсткого реального часу.

У ході теоретичного дослідження детально проаналізовано апаратну архітектуру мікроконтролера. Визначено, що гнучкість підсистеми LEDC базується на розділенні таймерів та каналів через комутаційну матрицю GPIO. Встановлено, що для коректної реалізації драйвера низького рівня необхідна комплексна конфігурація суміжних модулів: блоку керування тактуванням (DPort) для активації периферії та матриці входів-виходів для маршрутизації сигналів, що часто приховано у стандартних бібліотеках.

На основі проведеного аналізу розроблено програмний метод прямого регістрового керування. Запропонований підхід передбачає відмову від використання проміжних функцій-обгорток та базується на прямому маніпулюванні бітовими полями регістрів конфігурації. Це дозволило реалізувати алгоритм, який оперує попередньо обчисленими константами та виключає накладні витрати на динамічні розрахунки, перевірки валідності даних та механізми синхронізації операційної системи.

Експериментальна верифікація підтвердила високу точність розробленого методу. Аналіз 18 тестових конфігурацій у широкому діапазоні частот показав, що систематична похибка частоти генерації не перевищує 0,03% і зумовлена виключно апаратними допусками кварцового резонатора. Похибка формування шпаруватості сигналу для більшості режимів знаходиться в межах 0,6%, що

свідчить про повну відповідність фізичних параметрів сигналу теоретичним моделям.

Визначальним результатом роботи є доведення радикальної переваги методу у швидкодії (латентності). Порівняльний аналіз показав, що прямий регістровий доступ дозволяє скоротити час ініціалізації периферії у 47 разів (з 438 мкс до 9,29 мкс). Час виконання команди динамічного оновлення параметрів скорочено у 26 разів — до 167 наносекунд. Така швидкодія дозволяє використовувати ESP32 у контурах керування з частотою дискретизації понад 1 МГц.

Практична цінність отриманих результатів полягає у створенні інструментарію, що дозволяє застосовувати доступні мікроконтролери ESP32 у високодинамічних системах, таких як векторне керування електродвигунами (FOC), цифрові джерела живлення та прецизійна робототехніка, де мінімізація затримок є необхідною умовою стабільності системи.

За результатами кваліфікаційної роботи подано статтю до фахового журналу "Вісник Харківського національного університету імені В.Н. Каразіна, серія «Математичне моделювання. Інформаційні технології. Автоматизовані системи управління»"

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Valvano J. W., Yerraballi R. Embedded Systems — Shape the World: A Cyber-Physical Systems Approach. Austin : The University of Texas at Austin, 2014. URL: https://users.ece.utexas.edu/~valvano/Volume1/E-Book/ (дата звернення: 15.02.2025).
2. Barr M. Programming Embedded Systems in C and C++. Sebastopol : O'Reilly, 1999. 174 p. URL: https://archive.org/details/programmingembed0000barr (дата звернення: 28.02.2025).
3. Pulse-width Modulation (PWM) Timers in Microcontrollers / All About Circuits. 2021. URL: https://www.allaboutcircuits.com/technical-articles/introduction-to-microcontroller-timers-pwm-timers/ (дата звернення: 10.03.2025).
4. ATmega328P : 8-bit AVR Microcontroller with 32K Bytes In-System Programmable Flash : Datasheet / Microchip Technology Inc. 2015. URL: https://ww1.microchip.com/downloads/en/DeviceDoc/Atmel-7810-Automotive-Microcontrollers-ATmega328P_Datasheet.pdf (дата звернення: 25.03.2025).
5. ATmega640/1280/1281/2560/2561 : 8-bit AVR Microcontroller : Datasheet / Microchip Technology Inc. 2014. URL: https://ww1.microchip.com/downloads/en/devicedoc/atmel-2549-8-bit-avr-microcontroller-atmega640-1280-1281-2560-2561_datasheet.pdf (дата звернення: 25.03.2025).
6. Getting Started with Arduino : official documentation / Arduino. URL: https://docs.arduino.cc/learn/starting-guide/getting-started-arduino/ (дата звернення: 12.04.2025).
7. Lecture 7: ATmega328 Timers and Interrupts : Course CSE P567: Embedded Systems / University of Washington. Seattle, 2010. 32 p. URL:

https://courses.cs.washington.edu/courses/csep567/10wi/lectures/Lecture7.pdf (дата звернення: 05.05.2025).

8. ESP32 Series Datasheet : Version 4.5 / Espressif Systems. 2024. URL: https://www.espressif.com/sites/default/files/documentation/esp32_datasheet_en.pdf (дата звернення: 20.06.2025).
9. Barry R. Mastering the FreeRTOS Real Time Kernel. Real Time Engineers Ltd, 2016. 400 p.
10. LED Control (LEDC) : Programming Guide for ESP32 ; ESP-IDF v5.5.1 / Espressif Systems. 2024. URL: https://docs.espressif.com/projects/esp-idf/en/stable/esp32/api-reference/peripherals/ledc.html (дата звернення: 15.08.2025).
11. ESP32 Technical Reference Manual : Version 5.5 / Espressif Systems. 2020. 661 p. URL: https://www.espressif.com/sites/default/files/documentation/esp32_technical_reference_manual_en.pdf (дата звернення: 01.09.2025).
12. CP210x USB to UART Bridge VCP Drivers / Silicon Labs. URL: https://www.silabs.com/developers/usb-to-uart-bridge-vcp-drivers (дата звернення: 10.09.2025).
13. Logic 2 Software Download / Saleae. URL: https://www.saleae.com/downloads/ (дата звернення: 05.10.2025).
14. Python 3.12 Documentation : official documentation / Python Software Foundation. URL: https://docs.python.org/3/ (дата звернення: 25.10.2025).

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
Рівень вищої освіти (освітньо-кваліфікаційний рівень) ***Магістр***
Галузь знань: 17 – Електроніка, автоматизація та електронні комунікації
Спеціальність: 174 – Автоматизація, комп'ютерно-інтегровані технології та робототехніка
Освітня програма «Комп'ютеризовані системи управління та автоматика»

ЗАТВЕРДЖУЮ
Завідувач кафедри комп'ютерних
систем та робототехніки
к. ф.-м. н., доц. ХРУСЛОВ М. М.
«02» жовтня 2024року

З А В Д А Н Н Я
НА КВАЛІФІКАЦІЙНУ РОБОТУ

ГОРЕНКО Данієль Васильович
(прізвище, ім'я, по батькові студента)

1. Тема роботи **«Метод керування LEDC таймерами мікроконтролера ESP32»**

керівник роботи **КОТВИЦЬКИЙ Альберт Тадеушевич, кандидат фізико-математичних наук, доцент.**
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

затверджені наказом по університету ***від 30 вересня 2025 року № 4101-5/3554***

2. Строк подання студентом роботи ***30 листопада 2025 року***

3. Перелік питань, які потрібно розробити
1) Обгрунтування актуальності регістрового доступу для ESP32 на противагу обмеженням API.
2) Аналіз літератури: порівняння платформ (AVR, ESP32) та вивчення технічної документації ESP32.
3) Вивчення апаратної архітектури та регістрів підсистеми LEDC.
4) Розробка програмного методу прямого регістрового керування LEDC.
5) Розробка методики та налаштування експериментального стенду.
6) Проведення експериментів, аналіз практичних даних та оцінка похибок вимірювань.
7) Формування результатів та висновків.

4. План роботи

№ з/п	Назви етапів роботи	Термін виконання етапів роботи
1.	Затвердження теми роботи та керівника	30.09.2024 – 21.02.2025
2.	Аналіз актуальності теми, огляд літератури: порівняння платформ (AVR, ESP32) та вивчення технічної документації .	21.02.2025 - 30.04.2025
3.	Вивчення апаратної архітектури та регістрів підсистеми LEDC.	01.05.2025 - 15.06.2025
4.	Розробка програмного методу прямого регістрового керування LEDC.	16.06.2025 - 15.07.2025
5.	Розробка методики та налаштування експериментального стенду.	16.07.2025 - 31.07.2025
6.	Підготовка і оформлення основних розділів кваліфікаційної роботи.	01.08.2025 - 30.08.2025
7.	Оформлення звіту про науково-дослідну практику. Проведення серії експериментів, аналіз даних, оцінка похибок та формування висновків. Написання статті.	01.09.2025- 30.10.2025-
8.	Підготовка і оформлення звітних матеріалів кваліфікаційної роботи. Оформлення списку літератури	10.10.2025- 30.10.2025-
9.	Оформлення пояснювальної записки кваліфікаційної роботи відповідно вимогам до звітів про НДР.	10.10.2025- 30.11.2025
10.	Підготовка і оформлення звітних матеріалів та додатків кваліфікаційної роботи.	01.11.2025- 30.11.2025
11.	Оформлення звіту про переддипломну практику	24.11.2025 - 30.11.2025
12.	Представлення кваліфікаційної роботи керівнику та рецензенту	24.11.2025 - 30.11.2025

5. Дата видачі завдання ***02 жовтня 2024 року.***

Студент **Д. В. Горенко** _____
ініціали, прізвище підпис

Керівник роботи **А. Т. Котвицький** _____
ініціали, прізвище підпис

Додаток Б

Затверджую

_____

«_____»_____2020 р.

Технічне завдання

на розробку програмного виробу «Метод керування LEDC таймерами мікроконтролера ESP32»

1.	Введення	1.1. Назва: Метод керування LEDC таймерами мікроконтролера ESP32. 1.2. Галузь застосування: вбудовані системи (Embedded Systems), робототехніка, силова електроніка, IoT.
2.	Підстава для розробки	2.1. Навчальний план за спеціальністю 151 – Автоматизація та комп'ютерно-інтегровані технології 2.2. Завдання на дипломну роботу магістра № № 4101-5/3554 від «30» вересня 2025 (представити як Додаток А до пояснювальної записки до дипломної роботи).
3.	Призначення розробки	3.1. Мета розробки: створення програмного методу для прямого керування апаратними регістрами підсистеми LEDC мікроконтролера ESP32. 3.2. Призначення розробки: надати можливість розробникам вбудованих систем отримувати високоточні, стабільні та детерміновані ШІМ-сигнали, оминаючи обмеження стандартних API, для задач, критичних до часу (керування двигунами, генерація частот). 3.3. Вихідні дані розробки: технічна документація ESP32 (Technical Reference Manual), специфікації регістрів LEDC.
4.	Технічні вимоги до	4.1. Вимоги до функціональних характеристик: немає 4.2. Вимоги до надійності: забезпечувати стабільну генерацію сигналу.

	програмного виробу	4.3.Вимоги до умов експлуатації: мікроконтролер ESP32 (серія ESP32-WROOM або аналоги). 4.4. Вимоги до складу і параметрів технічних засобів: ПК з середовищем розробки (VS Code / ESP-IDF / Arduino IDE), плата ESP32, логічний аналізатор для налагодження. 4.5. Вимоги до інформаційної та програмної сумісності: мова програмування C/C++ 4.6. Вимоги до маркування та упаковки: немає 4.7. Вимоги до транспортування і зберігання: на звичайних носіях інформації 4.8. Спеціальні вимоги: немає.	
5.	Вимоги до програмної документації	Програмною документацією до виробу «Метод керування LEDC таймерами мікроконтролера ESP32» вважати: 1) Справжнє Технічне завдання на розробку виробу (представити у вигляді Додатку Б до пояснювальної записки до кваліфікаційної роботи). 2) Опис алгоритму налаштування регістрів (у розділі 2 пояснювальної записки). 3) Інструкцію користувача / лістинг коду (у додатках до кваліфікаційної роботи).	
6.	Вимоги до техніко-економічних показників	Програмною документацією до виробу «Метод керування LEDC таймерами мікроконтролера ESP32» вважати: 1) Справжнє Технічне завдання на розробку виробу (представити у вигляді Додатку Б до пояснювальної записки до кваліфікаційної роботи). 2) Опис алгоритму налаштування регістрів (у розділі 2 пояснювальної записки). 3) Інструкцію користувача / лістинг коду (у додатках до кваліфікаційної роботи).	
7.	Стадії і етапи розробки	Дата	Назва етапу
		21.02.2025 – 15.04.2025	Аналіз предметної області: вивчення архітектури ESP32, обмежень API та регістрової

			карти LEDC. Обґрунтування актуальності.
		16.04.2025 – 31.05.2025	Детальне вивчення регістрів таймерів та каналів LEDC. Розробка математичної моделі розрахунку параметрів ШІМ.
		01.06.2025 – 15.07.2025	Розробка програмного коду (алгоритму) для прямого конфігурування регістрів LEDC.
		16.07.2025 – 30.08.2025	Розробка методики тестування та налаштування експериментального стенду.
		01.09.2025 – 30.10.2025	Тестування розробленого методу на реальному обладнанні (серія експериментів). Аналіз результатів та оцінка похибок.
		01.11.2025 – 20.11.2025	Оформлення результатів. Написання пояснювальної записки.
		24.11.2025 – 30.11.2025	Представлення кваліфікаційної роботи керівнику та рецензенту.
8.	Порядок контролю і приймання програмного	1. Перевірку ходу розробки виконувати згідно з календарним планом (раз на місяць/етап). 2. Захист розробленого методу провести на засіданні Атестаційної комісії.	

	продукту (моделі)	3. Пояснювальну записку подати на паперових носіях та в електронному вигляді.

Виконавець
студент групи КУ- 61
Горенко Д. В.
_____

Замовник
к.ф.-м.н., доцент
Котвицький А. Т.
_____

Додаток В

Програма і методика випробувань програмного виробу

«**Метод керування LEDC таймерами мікроконтролера ESP32**»

1. Об'єкт випробувань

1. Назва програмного виробу : «Метод керування LEDC таймерами мікроконтролера ESP32»
2. Галузь застосування : Вбудовані системи (Embedded Systems), робототехніка, силова електроніка.

2. Мета випробувань

Перевірка відповідності функціональності програмної реалізації методу прямого регістрового керування заявленим вимогам у технічному завданні (Додаток Б до пояснювальної записки до кваліфікаційної роботи), зокрема перевірка точності та стабільності генерації ШІМ-сигналів.

3. Загальні положення

1. Підстави для проведення випробувань: Підставою для проведення випробувань є наказ про призначення атестаційної комісії.
2. Місце і тривалість випробувань: Приймальні випробування проводяться на базі лабораторії кафедри в період роботи атестаційної комісії.
3. Обсяг випробувань: Приймальні випробування програмного виробу проводяться в обсязі, відповідному цій програмі і методиці випробувань.
4. Організації, які беруть участь у випробуваннях: Приймальні випробування проводяться атестаційної комісією за участю Замовника (керівника), Виконавця (студента) та інших осіб, присутніх на засіданні.

4. Вимоги до програми або програмного виробу

Модель повинна задовольняти наступним вимогам:

1. працювати на мікроконтролері ESP32 (серія ESP32-WROOM);
2. забезпечувати можливість налаштування частоти, розрядності та коефіцієнта заповнення ШІМ через регістри;
3. забезпечувати точність генерації частоти з похибкою не більше ±0.03%;

4. забезпечувати точність формування тривалості імпульсу з похибкою не більше ±0.6%;
5. бути сумісним з середовищем розробки ESP-IDF;
6. вимоги до маркування та упаковки (не висуваються);
7. вимоги до транспортування і зберігання (на звичайних носіях інформації
8. Спеціальні вимоги (не пред’являються).

5. Вимоги до програмної документації

1. Справжнє технічне завдання на розробку (представити як Додаток Б до пояснювальної записки).
2. Програму і методику випробувань (представити як Додаток В до пояснювальної записки).
3. Опис алгоритму та інструкцію з налаштування регістрів (представити в Розділі 2 пояснювальної записки).
4. Лістинг програмного коду (представити як Додаток Г до пояснювальної записки).

6. Засоби і порядок випробувань

6.1 Засоби випробувань

Для проведення випробувань необхідні:

1. Персональний комп'ютер з встановленим середовищем ESP-IDF.
2. Апаратна плата ESP32-DevKitC V4.
3. Логічний аналізатор з програмним забезпеченням Saleae Logic.
4. З'єднувальні провідники.

6.2 Порядок проведення випробувань

Як правило, випробування проводяться в два етапи:

- 1-й етап (ознайомчий): Перевірка комплектності документації та наявності технічних засобів.
- 2-й етап (випробування): Перевірка функціональних характеристик шляхом виконання тестових сценаріїв.

Перелік перевірок, що проводяться на 2 етапі випробувань, включає в себе:

1. Завантаження прошивки з реалізованим методом у мікроконтролер ESP32.
2. Підключення логічного аналізатора до відповідних GPIO.
3. Запуск вимірювання та аналіз отриманих осцилограм.

Для проведення випробувань пропонується **Тест 1**, **Тест 2** та **Тест 3**.

Тест 1. Перевірка генерації сигналу базової частоти

Мета: Перевірити правильність розрахунку та встановлення частоти через дільник (DIV) та розрядність (RES).

Вхідні дані: Налаштування регістрів для RES = 6 біт, DIV = 4, DUTY = 10

$$f_{PWM} = \frac{f_{CLK}}{DIV \cdot 2^{RES}} = \frac{80000000}{4 \cdot 2^6} = 312500 \text{ Гц}.$$

Очікуваний результат: На виході GPIO наявний прямокутний сигнал з частотою 312.5 кГц ±0.3%.

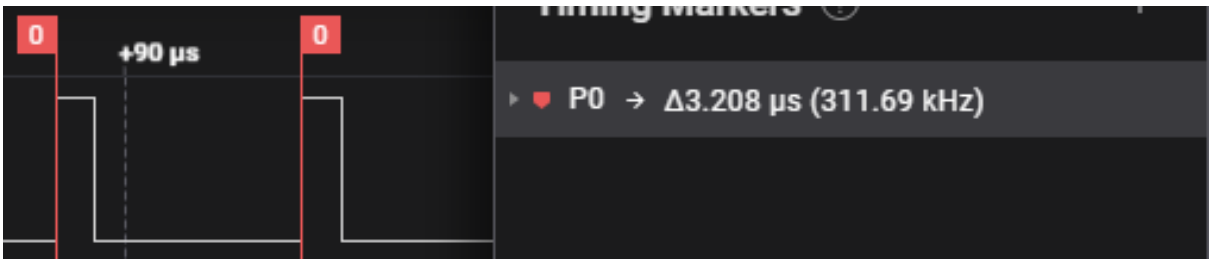

Рис. В.1 – Тест 1.

Результат: Логічний аналізатор зафіксував сигнал частотою 311.69 МГц. Похибка 0.26%. Тест пройдено.

Тест 2. Перевірка зміни коефіцієнта заповнення (Duty Cycle)

Мета: Перевірити точність формування тривалості імпульсу.

Вхідні дані: Ті самі налаштування (RES = 6 біт, DIV = 4), DUTY = 32 (50% з 64).

Період

$$T_{PWM} = \frac{1}{f_{PWM}} = \frac{1}{312500} = 0{,}0000032 \text{ с.} = 3.2 \text{ мкс.}$$

Очікувана тривалість високого рівня

$$t_H = \frac{DUTY}{2^{RES}} * T_{PWM} = \frac{32}{2^6} \cdot 0{,}0000032 = 0{,}0000016 \text{ с.} = 1.6 \text{ мкс.}$$

Очікуваний результат: Тривалість імпульсу на осцилограмі відповідає розрахунковій з похибкою не більше ±1.2%.

Рис. В.2 – Тест 2.

Результат: Виміряна тривалість t_H = 1.583 мкс. Похибка 1.06%. Тест пройдено.

Тест 3. Перевірка стабільності при зміні параметрів

Мета: Перевірити коректність роботи при зміні конфігурації (збільшення розрядності).

Вхідні дані: Переналаштування регістрів на RES = 7 біт, DIV = 2.

$$f_{PWM} = \frac{f_{CLK}}{DIV \cdot 2^{RES}} = \frac{80000000}{2 \cdot 2^7} = 312500 \text{ Гц.}$$

Очікуваний результат: Частота сигналу залишається стабільною (312.5 кГц), сигнал генерується без зривів.

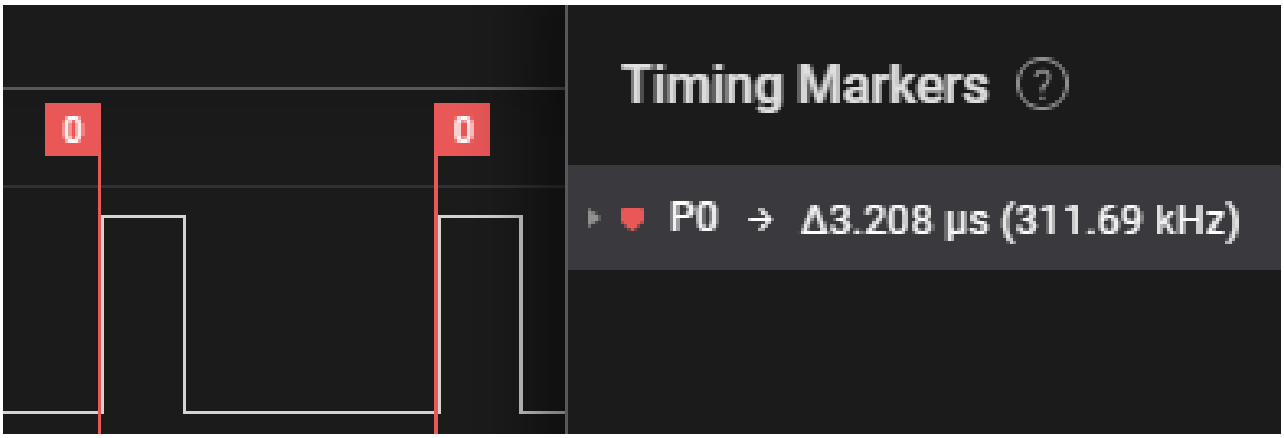

Рис. В.3 – Тест 3.

Результат: Отримано стабільний сигнал частоти 311.69 МГц. Похибка 0.26%. Тест пройдено.

Висновки: Тест 1, Тест 2 та Тест 3 успішно пройшли випробування. Програмний виріб відповідає вимогам технічного завдання.

Виконавець: студент групи КУ-61, Горенко Д. В. _____

Додаток Г

Вихідний код програми тестування конфігурацій

```
#include "soc/ledc_reg.h"
#include "soc/io_mux_reg.h"
#include "soc/gpio_sig_map.h"  // For LEDC_HS_SIG_OUT0_IDX
#include "soc/gpio_reg.h"
#include "soc/soc.h"
#include "soc/dport_reg.h" // Clock signal control
#include "soc/dport_access.h"  //For DPORT_REG_WRITE() and DPORT_REG_READ()

void setup() {

    // Enable LEDC clocking
    DPORT_REG_WRITE(DPORT_PERIP_CLK_EN_REG, DPORT_REG_READ(DPORT_PERIP_CLK_EN_REG) | DPORT_LEDC_CLK_EN);
    DPORT_REG_WRITE(DPORT_PERIP_RST_EN_REG, DPORT_REG_READ(DPORT_PERIP_RST_EN_REG) & ~DPORT_LEDC_RST);

    //Connect the LEDC_HS_SIG_OUT0 output signal to GPIO2
    REG_WRITE(GPIO_FUNC2_OUT_SEL_CFG_REG, LEDC_HS_SIG_OUT0_IDX);//LEDC_HS_SIG_OUT0_IDX = 71 = 0x47
    REG_WRITE(GPIO_ENABLE_W1TC_REG, (1 << 2));

    //(1<<25) on APB_CLK (80 MHz); (4<<13) integer divider = 4; (6) set PWM to 6 bits
    REG_WRITE (LEDC_HSTIMER0_CONF_REG, ((1<<25)|(4<<13)|(6)));

    //(1<<2) = 4 sets the ENable enable bit; bits 0 and 1 equal results -> use htimer0
    REG_WRITE (LEDC_HSCH0_CONF0_REG, (1<<2));

    REG_WRITE (LEDC_HSCH0_HPOINT_REG, 0); // the initial phase is zero
    REG_WRITE (LEDC_HSCH0_DUTY_REG, (10<<4)); //PWM parameter 10 of 64

    //It is very important to start the timer by writing 0 to the pause bit
    REG_WRITE(LEDC_HSTIMER0_CONF_REG,REG_READ(LEDC_HSTIMER0_CONF_REG)&~(1<<23));
    REG_WRITE (LEDC_HSCH0_CONF1_REG, (1 << 31)); //Start updating the duty cycle even though we don't use automatic fade

}
void loop() {}

```

Додаток Д

Вихідний код Python програми для розрахунків

```
import pandas as pd
import numpy as np
import argparse
import os
import concurrent.futures
from typing import Tuple, Dict, Any, Union

# --- КОНСТАНТА ОБМЕЖЕННЯ СЕМПЛІВ ---
SAMPLE_LIMIT = None
# SAMPLE_LIMIT = 100000

# --- КОНСТАНТИ ОПТИМІЗАЦІЇ РОЗРАХУНКІВ ---
CALC_PERIOD = True
CALC_FREQ = True
CALC_HIGH_TIME = True


def analyze_signal(df: pd.DataFrame, channel: str) -> Tuple[float, float,
float]:

    p_s, f_hz, h_s = 0.0, 0.0, 0.0

    ch = df[channel]
    times = df['Time [s]'].to_numpy(dtype=float)

    # --- Пошук фронтів ---
    diff = ch.diff().to_numpy()
    all_edges_idx = np.where(diff != 0)[0]

    if len(all_edges_idx) < 2:
        raise ValueError("Недостатньо фронтів сигналу для аналізу")

    edge_types = diff[all_edges_idx]
    all_edge_times = times[all_edges_idx]

    edge_times_rising = all_edge_times[edge_types == 1.0]
    edge_times_falling = all_edge_times[edge_types == -1.0]
    # --- Кінець пошуку фронтів ---

    # --- Розрахунок Періоду та Частоти ---
    if CALC_PERIOD or CALC_FREQ:
        if len(edge_times_rising) < 2:
            raise ValueError("Недостатньо наростаючих фронтів (min 2)")

        periods_s = np.diff(edge_times_rising)
        mean_period_s = np.mean(periods_s)

        if CALC_PERIOD:
            p_s = mean_period_s

        if CALC_FREQ:
            f_hz = 1 / mean_period_s if mean_period_s > 0 else 0

    # --- Розрахунок Часу HIGH ---
    if CALC_HIGH_TIME:
        high_times_s = []
        if len(edge_times_falling) > 0 and len(edge_times_rising) > 0:
            # Знаходимо найближчий спадаючий фронт після кожного наростаючого
```

```
            next_falling_indices = np.searchsorted(edge_times_falling,
edge_times_rising, side='right')
            for i, r_time in enumerate(edge_times_rising):
                if next_falling_indices[i] < len(edge_times_falling):
                    f_time = edge_times_falling[next_falling_indices[i]]
                    high_times_s.append(f_time - r_time)

        if high_times_s:
            h_s = np.mean(np.array(high_times_s))

    return p_s, f_hz, h_s


def print_channel_results(channel_name: str, result: Union[Tuple, Exception]):

    header_title = f"--- Аналіз каналу: '{channel_name}' ---"
    table_width = 0
    print("\n")

    time_formatters = {
        "(с)": "{:.9f}".format, "(мс)": "{:.4f}".format,
        "(мкс)": "{:.4f}".format, "(нс)": "{:.4f}".format
    }
    freq_formatters = {
        "(Гц)": "{:.4f}".format, "(кГц)": "{:.4f}".format, "(МГц)":
"{:.4f}".format
    }
    col_spacing = 15

    try:
        if isinstance(result, Exception):
            raise result

        # Розпаковка результату
        p_s, f_hz, h_s = result

        # --- Формування таблиць на основі констант ---
        time_data = []
        if CALC_PERIOD:
            time_data.append(
                {"Параметр": "Середній Період", "(с)": p_s, "(мс)": p_s * 1e3,
"(мкс)": p_s * 1e6, "(нс)": p_s * 1e9})


        if CALC_HIGH_TIME:
            time_data.append(
                {"Параметр": "Середній Час HIGH", "(с)": h_s, "(мс)": h_s * 1e3,
"(мкс)": h_s * 1e6, "(нс)": h_s * 1e9})

        freq_data = []
        if CALC_FREQ:
            freq_data.append({"Параметр": "Середня Частота", "(Гц)": f_hz,
"(кГц)": f_hz / 1e3, "(МГц)": f_hz / 1e6})
        # --- Кінець формування ---

        time_table_str = ""
        if time_data:
            df_time = pd.DataFrame(time_data)
            time_table_str = df_time.to_string(index=False,
formatters=time_formatters, justify='right',
                                               col_space=col_spacing)

        freq_table_str = ""
```

```
        if freq_data:
            df_freq = pd.DataFrame(freq_data)
            freq_table_str = df_freq.to_string(index=False,
formatters=freq_formatters, justify='right',
                                               col_space=col_spacing)

        # Визначаємо ширину рамки
        w_time = max(len(line) for line in time_table_str.split('\n')) if
time_table_str else 0
        w_freq = max(len(line) for line in freq_table_str.split('\n')) if
freq_table_str else 0
        table_width = max(w_time, w_freq, len(header_title))

        # Вивід
        print(header_title.center(table_width, '-'))

        if time_table_str:
            print("  Часові параметри:")
            print(time_table_str)

        if time_table_str and freq_table_str:
            print("-" * table_width)  # Розділювач

        if freq_table_str:
            print("\n  Частотні параметри:")
            print(freq_table_str)

        print("-" * table_width + "\n")

    except Exception as e:
        error_str = f"  Помилка аналізу: {e}"
        table_width = max(len(header_title), len(error_str))
        print(header_title.center(table_width, '-'))
        print(error_str)
        print("-" * table_width + "\n")


def main():
    parser = argparse.ArgumentParser(
        description="Обчислення параметрів ШІМ-сигналу (Період, Частота, High
Time) з CSV-файлу.")
    parser.add_argument('file', nargs='?', default='digital.csv', help="CSV-файл
з даними (default: digital.csv).")
    args = parser.parse_args()

    pd.set_option('display.width', 1000)

    try:
        if not os.path.exists(args.file):
            raise FileNotFoundError(f"Файл '{args.file}' не знайдено.")

        df_full = pd.read_csv(args.file)

        if 'Time [s]' not in df_full.columns:
            raise ValueError("Обов'язкова колонка 'Time [s]' відсутня в файлі.")

        total_samples = len(df_full)
        print(f"Файл: '{args.file}'")
        print(f"Загальна кількість семплів у файлі: {total_samples:,}")

        if SAMPLE_LIMIT is not None and total_samples > SAMPLE_LIMIT:
            print(f"(!) Обмеження аналізу першими {SAMPLE_LIMIT:,} семплами.")
            df = df_full.iloc[:SAMPLE_LIMIT].copy()
```

```
            print(f"Використовується для аналізу: {len(df):,} семплів")
        else:
            df = df_full
            print(f"Використовується для аналізу: {total_samples:,} семплів")

        data_channels = [col for col in df.columns if col != 'Time [s]']
        if not data_channels:
            print("\nУ файлі не знайдено колонок з даними (окрім 'Time [s]').")
            return

        print(f"\nПочаток паралельного аналізу (використовується {os.cpu_count()
or 1} процесів)...")

        with concurrent.futures.ProcessPoolExecutor(max_workers=None) as
executor:

            future_to_channel = {
                executor.submit(analyze_signal, df, channel): channel
                for channel in data_channels
            }

            results: Dict[str, Any] = {}
            for future in concurrent.futures.as_completed(future_to_channel):
                channel_name = future_to_channel[future]
                try:
                    data = future.result()
                    results[channel_name] = data
                except Exception as e:
                    results[channel_name] = e

        print("Аналіз завершено. Форматування виводу...")

        for channel_name in data_channels:
            result = results[channel_name]
            print_channel_results(channel_name, result)

    except (FileNotFoundError, ValueError, Exception) as e:
        print(f"\nКритична помилка: {e}")


if __name__ == "__main__":
    main()
```

Додаток Е

Результати вимірювань 18 тестових конфігурацій

№	RES (біт)	DIV	DUTY	$f_{теор}$, кГц	$f_{практ}$, кГц	Похибка f, %	$T_{теор}$, мкс	$T_{практ}$, мкс	Похибка T, %	t_H теор, мкс	t_H практ, мкс	Похибка t_H, %
1	4	1	10	5000	4998,37	-0,03%	0,2	0,20007	0,03%	0,125	0,12445	-0,44%
2	4	1	8	5000	4998,37	-0,03%	0,2	0,20007	0,03%	0,1	0,09943	-0,57%
3	5	1	10	2500	2499,19	-0,03%	0,4	0,40013	0,03%	0,125	0,12446	-0,43%
4	5	1	16	2500	2499,19	-0,03%	0,4	0,40013	0,03%	0,2	0,19951	-0,25%
5	6	1	10	1250	1249,59	-0,03%	0,8	0,80026	0,03%	0,125	0,12446	-0,43%
6	6	1	32	1250	1249,59	-0,03%	0,8	0,80026	0,03%	0,4	0,39958	-0,11%
7	4	2	10	2500	2499,18	-0,03%	0,4	0,40013	0,03%	0,25	0,24953	-0,19%
8	4	2	8	2500	2499,19	-0,03%	0,4	0,40013	0,03%	0,2	0,19951	-0,24%
9	5	2	10	1250	1249,59	-0,03%	0,8	0,80026	0,03%	0,25	0,24954	-0,18%
10	5	2	16	1250	1249,59	-0,03%	0,8	0,80026	0,03%	0,4	0,39961	-0,10%
11	6	2	10	625	624,8	-0,03%	1,6	1,60052	0,03%	0,25	0,24952	-0,19%
12	6	2	32	625	624,8	-0,03%	1,6	1,60052	0,03%	0,8	0,79972	-0,03%
13	4	4	10	1250	1249,59	-0,03%	0,8	0,80026	0,03%	0,5	0,49961	-0,08%
14	4	4	8	1250	1249,59	-0,03%	0,8	0,80026	0,03%	0,4	0,39958	-0,11%
15	5	4	10	625	624,8	-0,03%	1,6	1,60052	0,03%	0,5	0,49951	-0,10%
16	5	4	16	625	624,8	-0,03%	1,6	1,60052	0,03%	0,8	0,79963	-0,05%
17	6	4	10	312,5	312,4	-0,03%	3,2	3,20104	0,03%	0,5	0,49951	-0,10%
18	6	4	32	312,5	312,4	-0,03%	3,2	3,20104	0,03%	1,6	1,5999	-0,01%