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

До захисту допущено
Кафедрою комп'ютерних систем та робототехніки
протокол №__ від ____ червня 2026 р.

завідувач кафедри _____ Максим ХРУСЛОВ
(підпис)

«___»_____ 2026 р.

Кваліфікаційна робота

здобувача першого (бакалаврського) рівня вищої освіти

«МЕТОД ОДНОЧАСНОГО КЕРУВАННЯ КРОКОВИМИ ДВИГУНАМИ З ВИКОРИСТАННЯМ АПАРАТНИХ ТАЙМЕРІВ МІКРОКОНТРОЛЕРА»

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

Освітня програма ***Автоматизація та комп'ютерно-інтегровані технології***

Виконавець _____ **Денис ЧУХЛІБ**
(підпис)

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

Харків – 2026

АНОТАЦІЯ

Пояснювальна записка до кваліфікаційної роботи бакалавра складається зі вступу, чотирьох розділів, висновків, списку використаних джерел і чотирьох додатків. Загальний обсяг роботи складає 86 сторінок, із яких 46 сторінок основної частини з 6 рисунками, 4 таблицями та 25 найменуваннями у списку використаних джерел.

Метою кваліфікаційної роботи є мінімізація часового джитера під час генерації керуючих імпульсів у системах багатоосьового одночасного керування кроковими двигунами шляхом розробки програмної моделі на базі апаратних таймерів мікроконтролера.

Об'єкт дослідження — процеси багатоосьового синхронного керування кроковими двигунами у системах автоматизації та мобільних роботизованих платформах із нелінійною кінематикою.

Предмет дослідження — методи та програмно-апаратні засоби генерації синхронних керуючих імпульсів на основі апаратних таймерів мікроконтролера з мінімізацією часового джитера.

Проблема полягає в тому, що програмна генерація імпульсів на перевантаженому мікроконтролері спричиняє часовий джитер, який порушує синхронність осей. Для його усунення обчислення кінематики ізолюються від низькорівневої генерації сигналів за допомогою апаратних таймерів.

Область застосування — системи автоматизації, верстати з ЧПК, мобільні роботизовані платформи з кінематикою Аккермана.

Ключові слова: автоматизація та комп'ютерно-інтегровані технології, крековий двигун, часовий джитер, апаратний таймер, кінематика Аккермана, інтерполяція Брезенхема, S-подібний профіль, CRTP, ATmega2560.

ABSTRACT

The explanatory note to the bachelor's qualification work consists of an introduction, four chapters, conclusions, a list of references and four appendices. The total volume of the work is 86 pages, of which 46 pages are the main part with 6 figures, 4 tables, and 25 items in the list of references.

The purpose of the qualification work is to minimize timing jitter during the generation of control pulses in systems for multi-axis control of stepper motors by developing a software model based on the hardware timers of a microcontroller.

The object of research is the process of multi-axis synchronous control of stepper motors in automation systems and mobile robotic platforms with nonlinear kinematics.

The subject of research is the methods and hardware-software means of generating synchronous control pulses based on the hardware timers of a microcontroller with minimization of timing jitter.

The problem is that software-based pulse generation on an overloaded microcontroller causes timing jitter, which disrupts axis synchronization. To eliminate it, the kinematic computations are isolated from the low-level signal generation by means of hardware timers.

Area of application — automation systems, CNC machines, mobile robotic platforms with Ackermann kinematics.

Keywords: automation and computer-integrated technologies, stepper motor, timing jitter, hardware timer, Ackermann kinematics, Bresenham interpolation, S-curve profile, CRTP, ATmega2560.

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ І УМОВНИХ ПОЗНАЧЕНЬ 6
ВСТУП 8
РОЗДІЛ 1. АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ БАГАТООСЬОВОГО СИНХРОННОГО КЕРУВАННЯ КРОКОВИМИ ДВИГУНАМИ 10
1.1 Аналіз базових методів генерації керуючих імпульсів крокових двигунів та дослідження проблеми часового джитера 10
1.2 Огляд базових алгоритмів керування рухом у сучасних відкритих мікропрограмних системах (Marlin та Klipper) 12
1.3 Порівняльний аналіз апаратних платформ для задач багатоосьового синхронного керування (AVR ATmega2560 та ARM Cortex-M STM32) 14
Висновки за розділом 1 17
РОЗДІЛ 2. ПРОЄКТУВАННЯ СИСТЕМИ СИНХРОННОГО КЕРУВАННЯ КРОКОВИМИ ДВИГУНАМИ 19
2.1 Розробка концептуальної моделі та структурної схеми оптимальної системи керування 19
2.2 Математичне моделювання кінематики Аккермана та алгоритмів генерації траєкторій (S-подібний профіль швидкості) 21
2.3 Архітектура програмного забезпечення та файлова структура модулів (HAL, Kinematics, Motion, Common) 23
Висновки за розділом 2 25
РОЗДІЛ 3. АПАРАТНО-ПРОГРАМНА РЕАЛІЗАЦІЯ СИСТЕМИ КЕРУВАННЯ 27
3.1 Апаратна реалізація генерації кроків з використанням 16-бітних таймерів мікроконтролера 27
3.2 Реалізація синхронного руху осей на основі апаратних таймерів 29
3.3 Розробка програмних обгорток (API/Adapter) для інтеграції незалежного модуля у базовий код існуючих систем 31

Висновки за розділом 3 33
РОЗДІЛ 4. ТЕСТУВАННЯ ТА ВЕРИФІКАЦІЯ РОЗРОБЛЕНОЇ МОДЕЛІ 35
4.1 Програма, методика та засоби проведення випробувань розробленої моделі 35
4.2 Модульне тестування (Unit Tests) алгоритмів обчислення кінематики на ПК 36
4.3 Апаратна верифікація та осцилографічний аналіз синхронності осей (оцінка відсутності фазового зсуву) 39
4.4 Оцінка переваг запропонованої концептуальної моделі та надання рекомендацій щодо її практичного впровадження 42
Висновки за розділом 4 45
ВИСНОВКИ 47
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ 49
ДОДАТКИ 52

ПЕРЕЛІК СКОРОЧЕНЬ І УМОВНИХ ПОЗНАЧЕНЬ

4WD — повний привід (Four-Wheel Drive)

4WS — повне рульове керування (Four-Wheel Steering)

ОС — операційна система

ПЗ — програмне забезпечення

ПК — персональний комп'ютер

ЧПК — числове програмне керування

API (Application Programming Interface) — інтерфейс прикладного програмування

ARM (Advanced RISC Machine) — архітектура мікропроцесорів зі скороченим набором команд

AVR — 8-бітна мікроконтролерна архітектура компанії Microchip (Atmel)

CRTP (Curiously Recurring Template Pattern) — ідіома шаблонів, що рекурсивно успадковуються

CTC (Clear Timer on Compare Match) — режим таймера зі скиданням за збігом значень

DI (Dependency Injection) — впровадження залежностей

DMA (Direct Memory Access) — прямий доступ до пам'яті

FPU (Floating Point Unit) — апаратний модуль арифметики з плаваючою комою

GPIO (General-Purpose Input/Output) — порти введення-виведення загального призначення

GTCCR (General Timer/Counter Control Register) — регістр загального керування таймерами-лічильниками

HAL (Hardware Abstraction Layer) — шар апаратних абстракцій

HEX — формат файлу прошивки (Intel HEX)

IEEE 754 — стандарт представлення чисел із плаваючою комою

ISR (Interrupt Service Routine) — обробник переривання

MCU (Microcontroller Unit) — мікроконтролер

NVIC (Nested Vectored Interrupt Controller) — контролер вкладених векторизованих переривань

OCR (Output Compare Register) — регістр порівняння таймера

PWM (Pulse-Width Modulation) — широтно-імпульсна модуляція

SRP (Single Responsibility Principle) — принцип єдиної відповідальності

STM32 — сімейство 32-бітних мікроконтролерів на ядрі ARM Cortex-M

VCD (Value Change Dump) — формат запису зміни значень цифрових сигналів

ВСТУП

Актуальність роботи. Сучасні системи автоматизації, верстати з числовим програмним керуванням, роботизовані платформи та адитивні виробничі комплекси покладаються на крокові двигуни як на основні виконавчі механізми точного просторового позиціювання. Керування кроковими двигунами у багатоосьових системах вимагає суворої синхронізації між фізичними осями у режимі реального часу, причому ця задача суттєво ускладнюється для мобільних платформ із нелінійною кінематикою Аккермана, яка потребує розрахунку індивідуальних швидкостей і кутів повороту для кожного колеса.

Оскільки крокові двигуни переважно працюють у системах відкритого контуру, без зворотного зв'язку за фактичним положенням ротора, висока часова стабільність керуючих імпульсів стає критичною умовою їхньої коректної роботи. Нестабільність у часі між сусідніми імпульсами — часовий джитер — призводить до втрати крутного моменту, генерації акустичного шуму, виникнення механічних резонансів та пропуску кроків, що є фатальним для точності позиціювання. Цей дефект неминуче виникає за умови програмної генерації імпульсів на мікроконтролері, перевантаженому обчисленням нелінійної кінематики. Отже, задача мінімізації часового джитера шляхом ізоляції математичних обчислень від низькорівневої генерації сигналів є вкрай актуальною як для систем автоматизації, так і для мобільної робототехніки.

Метою роботи є мінімізація часового джитера під час генерації керуючих імпульсів у системах багатоосьового одночасного керування кроковими двигунами шляхом розробки програмної моделі на базі апаратних таймерів мікроконтролера.

Об'єкт дослідження — процеси багатоосьового синхронного керування кроковими двигунами у системах автоматизації та мобільних роботизованих платформах із нелінійною кінематикою.

Предмет дослідження — методи та програмно-апаратні засоби генерації синхронних керуючих імпульсів на основі апаратних таймерів мікроконтролера з мінімізацією часового джитера.

Методи дослідження: методи системного аналізу та порівняльного аналізу апаратних і мікропрограмних архітектур; математичне моделювання кінематики Аккермана; методи цифрової інтерполяції (алгоритм Брезенхема); методи об'єктно-орієнтованого та шаблонного програмування (ідіома CRTP, статичний поліморфізм); методи комп'ютерної симуляції та осцилографічної верифікації цифрових сигналів.

Завдання дослідження:

1. Виконати аналіз існуючих базових методів генерації керуючих імпульсів крокових двигунів та дослідити проблему виникнення часового джитера при програмній генерації імпульсів на перевантаженому мікроконтролері.

2. Провести порівняльний аналіз апаратних платформ для задач багатоосьового синхронного керування з оцінкою можливостей 8-бітних мікроконтролерів (AVR) та 32-бітних архітектур (ARM Cortex-M).

3. Здійснити огляд базових алгоритмів керування рухом (генерація траєкторій, лінійна інтерполяція) та проаналізувати їх практичну реалізацію у сучасних відкритих мікропрограмних системах (на прикладі Marlin та Klipper).

4. Розробити концептуальну модель та структурну схему оптимальної системи керування на основі проведеного аналізу обраних апаратно-програмних архітектур.

5. Виконати програмну та апаратну реалізацію методу з використанням таймерів мікроконтролера ATmega2560.

6. Розробити програму і методику випробувань та провести верифікацію моделі у середовищі симуляції.

7. Оцінити переваги запропонованої концептуальної моделі та надати рекомендації щодо її практичного впровадження в сучасні системи автоматизації.

РОЗДІЛ 1.
АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ БАГАТООСЬОВОГО СИНХРОННОГО КЕРУВАННЯ КРОКОВИМИ ДВИГУНАМИ

1.1 Аналіз базових методів генерації керуючих імпульсів крокових двигунів та дослідження проблеми часового джитера

Крокові двигуни належать до основних виконавчих механізмів дискретної дії, здатних відпрацьовувати задане переміщення без давача зворотного зв'язку за положенням ротора [1, 2]. Саме ця властивість зумовлює їх поширення у верстатах з ЧПК, адитивних установках та роботизованих платформах. Водночас за переходу від одноосьового до багатоосьового керування на перший план виходить не точність окремого приводу, а узгодженість руху осей між собою: кожен імпульс однієї осі має бути жорстко співвіднесений у часі з імпульсами решти, інакше задана траєкторія спотворюється. Це зміщує інженерний акцент із керування окремим двигуном на забезпечення синхронності всієї багатоосьової системи.

Базовий принцип керування рухом полягає у формуванні дискретних логічних сигналів (імпульсів), кожен з яких ініціює поворот ротора двигуна на фіксований кут (крок). Практична реалізація генерації таких імпульсів базується на двох основних підходах:

Програмна генерація (Software Delay). Формування періоду між імпульсами відбувається за рахунок використання блокуючих циклів або програмних перевірок часу у головному потоці виконання коду. Цей метод є обчислювально витратним і вразливим до будь-яких системних переривань чи паралельних задач, що призводить до непрогнозованих затримок видачі сигналів.

Апаратна генерація (Hardware Timers). Процес керування сигналами делегується апаратним таймерам-лічильникам мікроконтролера. Обчислювальне ядро лише конфігурує необхідний період спрацьовування таймера, після чого генерація керуючих імпульсів відбувається апаратно у фоновому режимі.

Оскільки крокові двигуни переважно працюють у системах відкритого контуру (open-loop) — без зворотного зв'язку за фактичним положенням ротора — висока часова стабільність керуючих імпульсів є критичною умовою їхньої коректної роботи. Нестабільність у часі між сусідніми імпульсами (відхилення фактичного періоду від розрахункового) визначається як часовий джитер (jitter) [3].

Будь-яке порушення синхронізації або наявність джитера керуючих імпульсів спричиняє деградацію фізичних характеристик системи, зокрема:

втрату номінального крутного моменту на валу двигуна;

генерацію акустичного шуму та вібрацій;

виникнення механічних резонансів;

пропуск кроків, що є фатальним для точності просторового позиціювання всієї системи [3].

У багатоосьових конфігураціях проблема джитера експоненційно масштабується. Синхронність у таких системах визначається не лише одночасним початком руху, а й підтримкою жорстко заданих математичних пропорцій між швидкостями, прискореннями та позиціями кожної осі протягом усього робочого циклу.

Для мобільних роботизованих платформ із незалежним керуванням осями ситуація додатково ускладнюється нелінійною кінематикою. Зокрема, для забезпечення руху чотириколісної платформи без бічного ковзання (проковзування) коліс під час виконання маневрів застосовується кінематична модель Аккермана. Вона вимагає розрахунку унікальних кутів повороту та швидкостей для кожного колеса окремо, спираючись на миттєвий центр обертання платформи у просторі [4].

Обчислення нелінійних залежностей Аккермана у режимі реального часу створює інтенсивне навантаження на мікроконтролер. Використання методів програмної генерації імпульсів у таких умовах є технічно недоцільним через високу ймовірність виникнення часового джитера. Відповідно, єдиним надійним

методом усунення проблеми нестабільності є повна ізоляція математичних обчислень від низькорівневого керування виводами шляхом використання апаратних таймерів [5].

1.2 Огляд базових алгоритмів керування рухом у сучасних відкритих мікропрограмних системах (Marlin та Klipper)

Трансляція складних математичних моделей кінематики у фізичні сигнали керування вимагає спеціалізованого мікропрограмного забезпечення (firmware). Сучасні системи автоматизації та 3D-друку переважно базуються на двох відкритих рішеннях: Marlin та Klipper [6, 7]. Ці системи представляють дві фундаментально різні архітектурні філософії управління рухом і вирішення проблеми часового джитера.

Монолітна архітектура (Marlin). Marlin є класичним представником монолітної архітектури вбудованих систем. У цій парадигмі єдиний мікроконтролер виконує весь спектр завдань: зчитування команд (G-коду), планування траєкторій із передбаченням (Look-ahead), обчислення кінематики та безпосередню генерацію крокових імпульсів через переривання таймера (ISR) [6].

Для багатоосьової синхронізації Marlin покладається на цілочисельний алгоритм Брезенхема [6]. Його суть полягає у визначенні «домінантної» осі, що має виконати найбільшу кількість кроків, і розрахунку кроків підпорядкованих осей через акумулятор похибки. Це дозволяє синхронізувати рух без використання важких операцій із плаваючою комою.

Незважаючи на ефективність алгоритму Брезенхема, монолітний підхід має суворі обмеження продуктивності на 8-бітних системах (наприклад, AVR ATmega2560). При спробі досягти частоти кроків понад 10-40 кГц виникає ефект процесорного голодування. Для уникнення повної зупинки системи Marlin застосовує техніку мультиплікації кроків (step doubling/quad stepping) – відправку кількох імпульсів за один виклик переривання. Це критично порушує рівномірність потоку імпульсів, генеруючи пачки (bursts) сигналів замість

плавного меандру, що експоненційно збільшує часовий джитер, викликає акустичний шум та механічні резонанси. Крім того, обчислювальні обмеження унеможливлюють повноцінне застосування S-подібних (S-Curve) профілів розгону, змушуючи використовувати менш ефективні трапецієподібні профілі з нескінченним математичним ривком (jerk) у точках зміни прискорення [8].

Розподілена архітектура (Klipper). Мікропрограма Klipper вирішує проблему обчислювального "пляшкового горлечка" шляхом фізичного розділення високорівневої математики та низькорівневої генерації сигналів (Host-MCU архітектура) [7].

Усі ресурсомісткі завдання, включаючи обчислення кінематики з плаваючою комою, генерацію S-подібних профілів швидкості та алгоритми гасіння резонансів (Input Shaping), виконуються на потужному головному процесорі (host), яким зазвичай виступає комп'ютер на базі ОС Linux (наприклад, Raspberry Pi).

Мікроконтролер (MCU) у цій системі звільнений від алгоритму Брезенхема та логіки прискорення, виконуючи роль виключно таймера-виконавця подій. Для передачі мільйонів кроків без затримок через стандартні інтерфейси зв'язку, Klipper застосовує власний протокол стиснення кроків на основі квадратичного рівняння: $step[i]=step+interval\cdot i+add\cdot\frac{i\cdot(i-1)}{2}$ [9] де interval задає базовий час між імпульсами, а add відображає прискорення.

Головний процесор передає компактний пакет параметрів, а мікроконтролер лише завантажує їх у регістри апаратного таймера. Це дозволяє генерувати імпульси з частотами 175-500 кГц [9] без застосування мультиплікації кроків, зводячи часовий джитер до апаратного мінімуму самого мікроконтролера.

Для наочного зіставлення проаналізованих підходів сформовано зведену таблицю 1.1 [10].

Таблиця 1.1

Порівняльна характеристика архітектур мікропрограмних систем керування рухом

Характеристика	Монолітна архітектура (Marlin)	Розподілена архітектура (Klipper)
Локалізація обчислень	Безпосередньо на мікроконтролері керування двигунами.	Розподілено: математика на Host-процесорі, імпульси на MCU.
Синхронізація осей	Програмна інтерполяція (алгоритм Брезенхема) у циклі MCU.	Попередньо обчислений точний часовий розклад перемикання пінів.
Максимальна частота	Низька (~10-40 кГц на 8-біт). Вимагає застосування step doubling.	Надвисока (до 500 кГц). Обмежена лише тактовою частотою таймера.
Складні профілі руху	Спрощені (переважно трапеція), викликають навантаження на CPU.	Повноцінні S-криві, Input Shaping (апаратний FPU на Host-процесорі).
Синхронізація плат	Практично неможлива з високою часовою точністю.	Нативна підтримка синхронізації кількох MCU через єдиний розклад.

Проміжний висновок: Аналіз сучасних базових алгоритмів демонструє, що усунення часового джитера неможливе в рамках чисто програмної генерації імпульсів на перевантаженому мікроконтролері. Оптимальне керування вимагає або перенесення математичного ядра на зовнішні обчислювальні потужності (як у Klipper), або радикальної оптимізації програмної моделі шляхом прямої взаємодії з апаратними таймерами мікроконтролера з мінімізацією накладних витрат на переривання, що і формує базис для подальшого проєктування.

1.3 Порівняльний аналіз апаратних платформ для задач багатоосьового синхронного керування (AVR ATmega2560 та ARM Cortex-M STM32)

Реалізація багатоосьових кінематичних моделей (зокрема кінематики Аккермана) вимагає не лише складної математичної бази, а й достатніх апаратних ресурсів для забезпечення паралельної генерації імпульсів у режимі

жорсткого реального часу. Вибір мікроконтролерної архітектури безпосередньо визначає межі продуктивності системи, рівень часового джитера та загальну стабільність управління рухом. Традиційно для цих задач розглядаються дві фундаментально різні архітектури: 8-бітна AVR та 32-бітна ARM Cortex-M.

Особливості 8-бітної архітектури (на прикладі AVR ATmega2560) Мікроконтролер ATmega2560 є базовим стандартом для відкритих систем автоматизації та містить чотири 16-бітні таймери-лічильники, що дозволяє апаратно генерувати сигнали керування у режимі CTC (Clear Timer on Compare). За тактової частоти 16 МГц один тік таймера з дільником 8 становить 0,5 мкс, що є достатнім для генерування базових частот кроків [11].

Проте 8-бітна архітектура має серйозні обмеження при роботі з високим рівнем мікрокрокування (наприклад, 1/256 кроку). Обчислення 32-бітних значень вимагає значної кількості тактів (наприклад, 4 цикли читання та 4 цикли додавання для однієї операції). Крім того, архітектура AVR характеризується високою та недетермінованою сумарною латентністю переривань [12]. Хоча апаратне входження у переривання займає лише 4 такти, програмне збереження контексту (регістрів загального призначення) у стек під час виклику обробника переривань (ISR) додає десятки тактів, через що загальна затримка стає недетермінованою.

При досягненні частоти кроків близько 40 кГц обробка переривань здатна поглинати понад 60% процесорного часу. Це викликає ефект «голодування процесора» (CPU starvation). Для уникнення повної зупинки системи застосовується техніка мультиплікації кроків (step doubling) [8], яка провокує нерівномірність потоку імпульсів і збільшує часовий джитер до рівня 2-10 мкс, що негативно позначається на плавності ходу механіки [12].

Особливості 32-бітної архітектури (на прикладі ARM Cortex-M). Перехід до мікроконтролерів сімейства ARM Cortex-M (наприклад, STM32 з ядрами M4/M7) концептуально вирішує проблему обчислювального навантаження. Завдяки наявності апаратного модуля арифметики з плаваючою комою (FPU)

базові операції з числами одинарної точності (множення, додавання, множення з накопиченням) виконуються апаратно приблизно за один такт, що на порядок прискорює обчислення тригонометричних функцій кінематики порівняно з програмною емуляцією на 8-бітній архітектурі AVR [13].

Визначальною перевагою для систем керування рухом є інтегрований контролер переривань NVIC. Він забезпечує жорстко детерміновану латентність переривань, яка становить рівно 12 тактів апаратного перемикання. Технології ланцюжкового виконання (tail-chaining) та апаратного збереження контексту дозволяють мінімізувати затримки під час послідовного спрацьовування таймерів різних осей. Завдяки цьому часовий джитер на рівні переривань знижується до 0,1-1 мкс [13, 14].

Більше того, застосування контролера прямого доступу до пам'яті (DMA) на платах STM32 [13] дозволяє повністю виключити центральний процесор із процесу генерації сигналів. Перемикання станів пінів (GPIO) відбувається апаратно з прив'язкою до таймера, зводячи джитер до нуля. За умови генерації частоти 40 кГц, завантаження процесора Cortex-M4 (168 МГц) становить лише 5-10%, вивільняючи ресурси для розрахунку S-подібних профілів швидкості та фільтрації резонансів.

Для наочного обґрунтування апаратних можливостей сформовано порівняльну таблицю 1.2 [11, 12, 13].

Таблиця 1.2

Порівняльна характеристика мікроконтролерних архітектур для генерації імпульсів

Характеристика	8-біт AVR (ATmega2560)	32-біт ARM Cortex-M (STM32)
Базова робоча частота	16 МГц	72 – 240+ МГц
Апаратна підтримка FPU	Відсутня (програмна емуляція)	Наявна (стандарт IEEE 754)
Латентність переривань	Недетермінована (4 такти апаратно + збереження контексту, разом ~30–40)	Жорстко детермінована (12 тактів)
Очікуваний часовий джитер	2-10 мкс (високий)	0,1-1 мкс (мінімальний)
Навантаження на CPU (40 кГц)	> 60% (ризик CPU starvation)	5-10% (вільні ресурси для математики)
Генерація через DMA	Відсутня/Обмежена	Повна апаратна підтримка

Проміжний висновок: Проведений порівняльний аналіз доводить, що 8-бітна архітектура AVR є вузьким місцем для розв'язання задач багатоосьової синхронізації високої точності. Проблеми недетермінованих переривань та обчислювальна слабкість унеможливлюють мінімізацію часового джитера на програмному рівні. Перенесення системи на 32-бітну архітектуру ARM Cortex-M з використанням NVIC та FPU є технічно обґрунтованим кроком для забезпечення мікросекундної точності багатоосьового позиціювання та стабільності генерації керуючих сигналів.

Висновки за розділом 1

У першому розділі проведено комплексний аналіз проблеми багатоосьового синхронного керування кроковими двигунами, що є критично важливою для забезпечення прецизійного руху автономних мобільних платформ із нелінійною кінематикою Аккермана. Встановлено, що головною перешкодою для досягнення високої точності, збереження крутного моменту та уникнення

механічних резонансів є часовий джитер (нестабільність у часі) керуючих імпульсів. Цей дефект неминуче виникає за умови використання програмних методів генерації затримок та перевантаження мікроконтролера обчислювальними задачами.

Огляд базових алгоритмів сучасних мікропрограмних рішень продемонстрував фундаментальні обмеження монолітних архітектур (на прикладі системи Marlin). Об’єднання високорівневої математики та низькорівневої генерації кроків на одному обчислювальному ядрі призводить до процесорного голодування, що змушує використовувати техніку мультиплікації кроків, яка експоненційно збільшує джитер. Водночас аналіз розподілених систем (на прикладі Klipper) довів технологічну необхідність розділення процесів обчислення кінематики (зокрема S-подібних профілів розгону та гальмування) та безпосередньої генерації кроків для досягнення апаратного мінімуму нестабільності.

Порівняльний аналіз апаратних платформ підтвердив неможливість ефективного розв'язання задачі синхронізації на базі 8-бітних мікроконтролерів сімейства AVR (наприклад, ATmega2560) через відсутність апаратного модуля FPU (програмна емуляція операцій із плаваючою комою) та високу недетерміновану латентність переривань. Оптимальним апаратним фундаментом визначено 32-бітну архітектуру ARM Cortex-M (STM32), яка завдяки модулю FPU та контролеру переривань NVIC здатна гарантувати жорсткий часовий детермінізм.

Отримані результати теоретичного аналізу повністю обґрунтовують відмову від традиційної програмної генерації імпульсів і формують вимогу до розробки нової архітектури системи керування. Ця система має базуватися на суворій апаратній ізоляції таймерів та ефективній математичній моделі інтерполяції для кінематики Аккермана, що формує базис для етапу проєктування у наступному розділі.

РОЗДІЛ 2.
ПРОЄКТУВАННЯ СИСТЕМИ СИНХРОННОГО КЕРУВАННЯ КРОКОВИМИ ДВИГУНАМИ

2.1 Розробка концептуальної моделі та структурної схеми оптимальної системи керування

На основі проведеного у першому розділі аналізу встановлено, що для мінімізації часового джитера необхідно чітко розмежувати високорівневі обчислення траєкторій та низькорівневу генерацію крокових імпульсів. Запропонована концептуальна модель системи керування базується на принципі повної ізоляції математичного ядра (яке обчислює кінематику) від апаратного рівня керування виводами (пінами) мікроконтролера.

Для прецизійного синхронного керування рухом у базовій реалізації на AVR застосовуються 16-бітні таймери-лічильники (наприклад, Timer3 та Timer4 для мікроконтролерів сімейства AVR). Використання 16-бітних таймерів забезпечує роздільну здатність регістрів порівняння (OCR) в діапазоні від 0 до 65535 [11]. Цей параметр є критично важливим для генерації точних низьких частот, оскільки дозволяє уникнути значного фракційного дрейфу (накопичення часової похибки) між лівим та правим колесами під час виконання маневрів.

З метою забезпечення максимальної швидкодії мікроконтролера та усунення накладних витрат (overhead) на виклик віртуальних функцій у поліморфних ієрархіях, архітектуру програмного забезпечення спроєктовано за принципом Header-only. В основу абстрагування апаратного забезпечення покладено ідіому CRTP (Curiously Recurring Template Pattern), що дозволяє створювати універсальні шаблони для класів таймерів та виводів без використання таблиць віртуальних методів (vtable), переносячи поліморфізм на етап компіляції [15, 16].

Логічно концептуальна модель розділена на чотири взаємопов'язані шари (директорії), що відображено на структурній схемі:

Модуль Common (Спільна конфігурація). Відповідає за зберігання глобальних типів даних (напрямок руху, ідентифікатори осей) та статичних конфігураційних параметрів. До них належать фізичні розміри мобільної платформи (наприклад, колісна база WHEEL_BASE_M=0,25 м, ширина колії TRACK_WIDTH_M=0,18 м), а також параметри тактування мікроконтролера.

Модуль Kinematics (Кінематика). Ізольоване математичне ядро, що відповідає за розрахунок нелінійних залежностей. Модуль реалізує принцип Аккермана, визначаючи індивідуальні швидкості та точні кути повороту для кожного колеса окремо, що необхідно для виконання повороту без бічного ковзання платформи.

Модуль Motion (Керувач рухом). Здійснює просторове планування траєкторій та лінійну інтерполяцію. Модуль також інтегрує логіку обчислення профілів швидкості, зокрема складних S-подібних кривих, що математично розділяють рух на 7 фаз (позитивний ривок, константне прискорення, негативний ривок, крейсерська швидкість, негативний ривок, константне гальмування, позитивний ривок) для забезпечення плавності механіки.

Модуль HAL (Hardware Abstraction Layer). Шар апаратних абстракцій, що містить CRTP-шаблони базових класів (HardwareTimer, StepperPin). Він формує єдиний контракт для роботи з регістрами конкретного мікроконтролера, повністю приховуючи низькорівневу специфіку від математичних модулів.

Взаємодія між описаними компонентами будується за моделлю "виробник-споживач" (producer-consumer). Головний цикл програми (виробник) асинхронно отримує координати цільової точки, після чого модулі Kinematics та Motion розраховують кількість необхідних кроків і точні часові інтервали. Ці дані перетворюються у значення регістрів та завантажуються в буфери модуля HAL.

Безпосередня генерація сигналів (споживач) делегується апаратним таймерам. Вся реальна робота зі зміни станів виводів відбувається у фоновому режимі через переривання (ISR) або контролер прямого доступу до пам'яті (DMA), що забезпечує генерацію частот без накопичення часового дрейфу

періоду навіть за умови повної зайнятості або блокування головного обчислювального потоку.

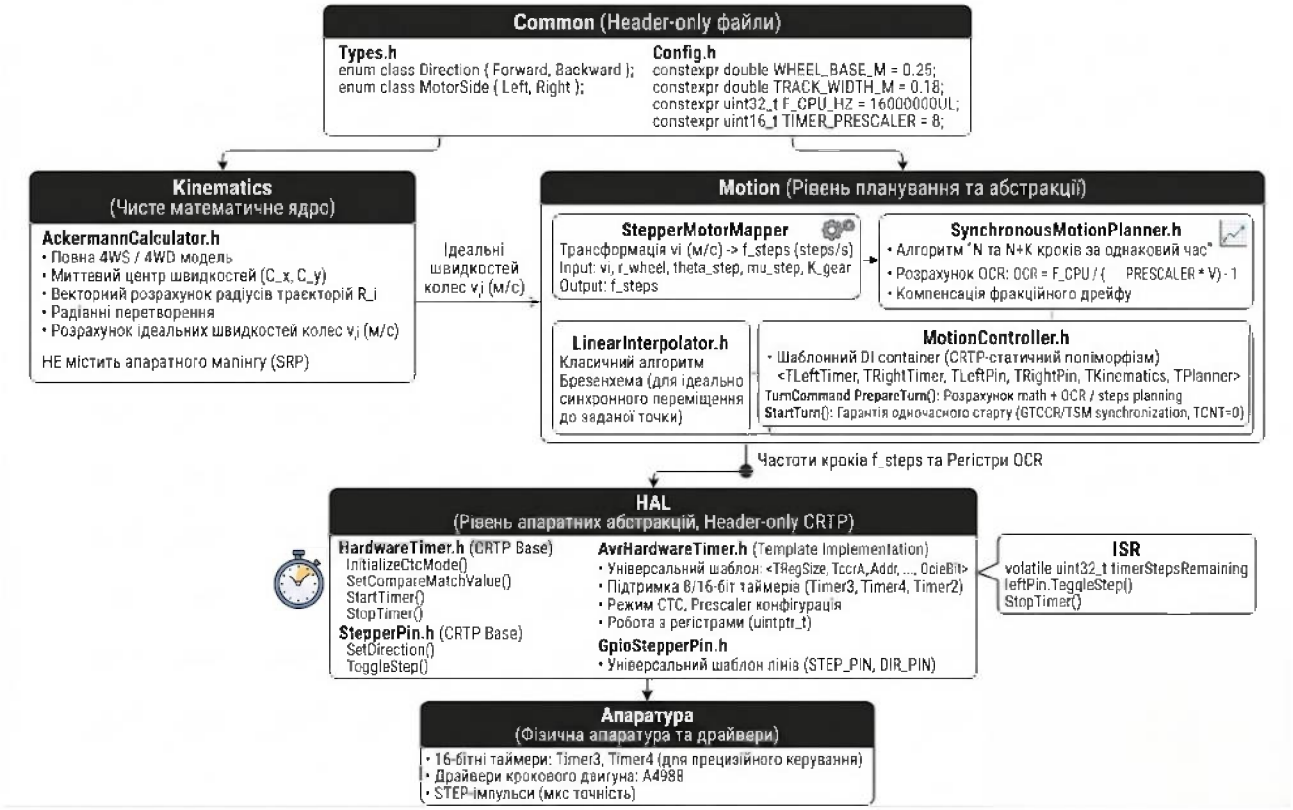


Рисунок 2.1 – Структурна схема програмно-апаратної моделі керування.

2.2 Математичне моделювання кінематики Аккермана та алгоритмів генерації траєкторій (S-подібний профіль швидкості)

Для забезпечення прецизійного позиціювання та виконання маневрів мобільною платформою без бічного ковзання (проковзування) коліс було застосовано математичну модель кінематики Аккермана [4, 17]. Суть цієї моделі полягає у визначенні миттєвого центру обертання платформи, відносно якого розраховуються індивідуальні радіуси дуг та кути повороту для кожного колеса окремо.

Геометричну інтерпретацію моделі Аккермана наведено на рисунку 2.2.

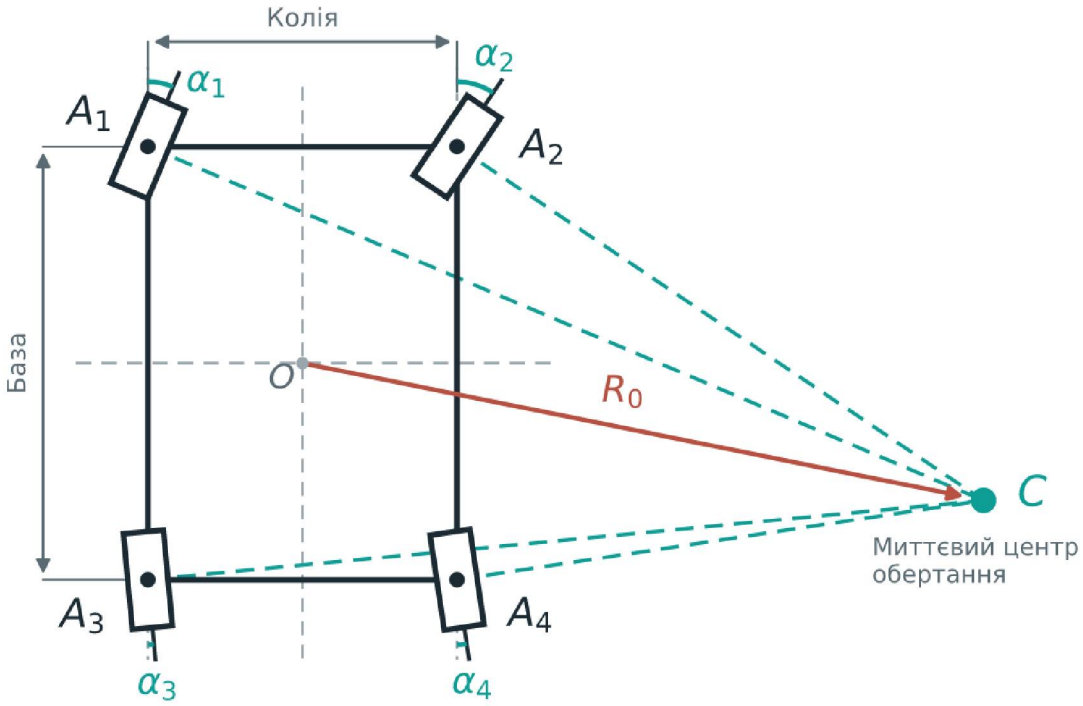


Рисунок 2.2 – Геометрична модель кінематики Аккермана.

Оскільки виконавчим елементом системи є кроковий двигун, аналогові значення кутів і лінійних переміщень трансформуються у дискретну цілочисельну кількість кроків. Фундаментальною вимогою алгоритму керування є те, що всі розраховані кроки для різних коліс повинні стартувати одночасно та завершуватися за практично однаковий проміжок часу (з точністю до похибки квантування таймерів). Перетворення цільової лінійної швидкості руху (м/с) у необхідну частоту кроків (Гц) здійснюється на основі фізичних параметрів механіки: радіусу колеса, передаточного числа редуктора, крокового кута двигуна та обраного режиму мікрокрокування [18].

Для запобігання втраті синхронізації (зриву ротора з магнітного поля) та зменшенню механічних резонансів двигуни повинні проходити етап плавного розгону та гальмування. Використання базового трапецієподібного профілю, де прискорення є постійним, породжує проблему нескінченного математичного ривка (jerk) у точках початку та закінчення розгону, що викликає удари в механіці [19].

З метою оптимізації кінематики розроблено математичну модель генерації S-подібного (S-Curve) профілю швидкості, яка згладжує криву швидкості та робить функцію прискорення безперервною [19, 20]. На відміну від трапеції, класичний S-подібний алгоритм детально розділяє рух на 7 математичних фаз:

1. Позитивний ривок: прискорення лінійно зростає від нуля.
2. Константне прискорення: утримання максимального стабільного прискорення.
3. Негативний ривок: прискорення плавно лінійно зменшується до нуля під час досягнення цільової швидкості.
4. Крейсерська швидкість: рух із постійною цільовою швидкістю, прискорення дорівнює нулю.
5. Негативний ривок (початок гальмування): лінійне зростання негативного прискорення.
6. Константне гальмування: стабільне падіння швидкості.
7. Позитивний ривок (завершення гальмування): лінійне зменшення негативного прискорення до повної та плавної зупинки [20].

Сформовані за допомогою цих моделей профілі швидкості та масиви кількості кроків надалі передаються до модуля лінійної інтерполяції. Цей алгоритм вираховує точні часові інтервали між кожним окремим імпульсом і конвертує їх у значення регістрів для завантаження в апаратні таймери мікроконтролера, що перетворює складну кінематику у фізичний рух. У межах поточної роботи апаратно верифіковано базовий режим зі сталою частотою кроків; програмна інтеграція S-подібного профілю в робочий тракт є напрямом подальшого розвитку моделі.

2.3 Архітектура програмного забезпечення та файлова структура модулів (HAL, Kinematics, Motion, Common)

Для забезпечення модульності, масштабованості та жорсткого дотримання часових обмежень під час багатоосьового синхронного керування, архітектуру програмного забезпечення розроблено на основі парадигми компонентного

проєктування. З метою усунення накладних витрат (overhead) процесорного часу під час виклику віртуальних функцій, архітектуру реалізовано за підходом "header-only" (розміщення коду виключно у заголовних файлах) із застосуванням ідіоми шаблонів, що рекурсивно успадковуються (CRTP). Це дозволяє компілятору виконувати підстановку (inlining) критичних ділянок коду на етапі компіляції [15, 16].

Програмна система логічно поділена на чотири ключові модулі, кожен з яких інкапсулює чітко визначену відповідальність:

Модуль Common (Спільна конфігурація та типи). Базується на файлах Config.h та Types.h. Відповідає за визначення глобальних структур даних, перелічень (наприклад, напрямків обертання) та збереження статичних фізичних параметрів платформи (колісна база, ширина колії, передаточні числа редукторів, налаштування мікрокрокування драйвера). Цей модуль є фундаментом для інших компонентів і не містить обчислювальної логіки.

Модуль Kinematics (Кінематика). Реалізований у файлах AckermannCalculator.h та WheelSpeeds.h. Виконує роль ізольованого математичного ядра. Відповідає за обчислення нелінійних залежностей моделі Аккермана. Алгоритм перетворює цільову просторову швидкість платформи та радіус повороту в індивідуальні лінійні швидкості та цілочисельну кількість кроків для кожного незалежного колеса. Обчислення здійснюються з використанням чисел із плаваючою комою для досягнення прецизійної точності.

Модуль Motion (Керування рухом). Об'єднує файли MotionController.h, SynchronousMotionPlanner.h, StepperMotorMapper.h, LinearInterpolator.h та SCurveProfile.h. Цей шар системи містить логіку планування траєкторій та генерації складних профілів швидкості, зокрема S-подібних (S-Curve), що математично розділяють процес розгону та гальмування на 7 безперервних фаз. Модуль формує для кожної осі окремий часовий розклад (значення OCR та кількість кроків), що завантажується у незалежні апаратні таймери; пропорційність цих значень забезпечує збереження геометричних співвідношень

між осями протягом усього руху. Додатково модуль містить клас лінійної інтерполяції Брезенхема як резервний механізм синхронізації для одногенераторних конфігурацій.

Модуль HAL (Шар апаратних абстракцій – Hardware Abstraction Layer). Представлений базовими контрактами HardwareTimer.h, StepperPin.h та їхніми платформозалежними реалізаціями (наприклад, AvrHardwareTimer.h, GpioStepperPin.h). Цей шар забезпечує стандартизований інтерфейс для взаємодії з апаратною периферією мікроконтролера. Він інкапсулює низькорівневі операції з регістрами та обробниками переривань (ISR), які безпосередньо транслюють математичний розклад у фізичні електричні імпульси (сигнали STEP).

Запропонована файлова структура та суворий розподіл відповідальностей реалізують принцип інверсії залежностей. Високорівневі математичні алгоритми (Kinematics та Motion) не мають прямих залежностей від конкретної мікроконтролерної архітектури, що робить ядро системи повністю кросплатформеним. Перенесення програмного забезпечення на іншу апаратну базу вимагатиме виключно розробки нових класів для модуля HAL, залишаючи математичну та логічну частини системи без змін.

Висновки за розділом 2

У другому розділі здійснено комплексне проєктування апаратно-програмної системи багатоосьового синхронного керування кроковими двигунами. На основі розробленої концептуальної моделі впроваджено фундаментальний принцип повної ізоляції високорівневих математичних обчислень від низькорівневого процесу генерації керуючих імпульсів. Для досягнення мікросекундної точності таймінгу та усунення часового джитера обґрунтовано застосування виключно 16-бітних апаратних таймерів-лічильників мікроконтролера у режимі фонового виконання.

Сформовано математичну базу для роботи мобільної платформи, в основу якої покладено нелінійну кінематичну модель Аккермана. Вона забезпечує розрахунок індивідуальних швидкостей та кутів повороту для кожного колеса,

що є критичною умовою для виконання маневрів без бічного проковзування. З метою оптимізації динамічних навантажень на механіку розроблено математичну модель генерації S-подібного (S-Curve) профілю швидкості з розподілом розгону та гальмування на 7 безперервних фаз, що усуває проблему нескінченного ривка. Цю модель підготовлено до інтеграції в модуль Motion; у поточній реалізації верифіковано базовий режим керування зі сталою частотою. Розраховані масиви кроків та профілі швидкості об'єднано з розкладом імпульсів для незалежних таймерів, пропорційність якого гарантує практично одночасне завершення руху всіма осями з різницею в межах похибки квантування OCR.

Спроєктовано програмну архітектуру системи на основі парадигми "header-only" із застосуванням шаблонів (CRTP). Цей підхід забезпечив перенесення поліморфізму на етап компіляції, повністю усунувши накладні витрати процесорного часу на виклик віртуальних функцій. Жорсткий поділ системи на функціональні модулі (Common, Kinematics, Motion та HAL) реалізував принцип інверсії залежностей: математичне ядро стало повністю кросплатформеним і незалежним від конкретної мікроконтролерної бази.

Запропонована математична модель та архітектурні рішення формують надійний фундамент, що повністю готовий до практичної розробки низькорівневих адаптерів та безпосередньої апаратної реалізації, що розглядається у наступному розділі.

РОЗДІЛ 3.
АПАРАТНО-ПРОГРАМНА РЕАЛІЗАЦІЯ СИСТЕМИ КЕРУВАННЯ

3.1 Апаратна реалізація генерації кроків з використанням 16-бітних таймерів мікроконтролера

Перехід від абстрактних математичних моделей до безпосереднього керування фізичними крокovими двигунами реалізується на рівні шару апаратних абстракцій (HAL). Головною вимогою до цього етапу є забезпечення детермінованої в часі генерації імпульсів без залучення головного циклу обчислень та мінімізація часового джитера.

Для забезпечення прецизійного керування на 8-бітних платформах (наприклад, AVR ATmega2560) застосовуються виключно 16-бітні апаратні таймери-лічильники (зокрема Timer3 та Timer4) [11]. Вибір 16-бітних модулів обґрунтовується їхньою високою роздільною здатністю: регістри порівняння (OCR) здатні приймати значення у діапазоні від 0 до 65535, на відміну від обмеженого діапазону 0–255 у стандартних 8-бітних таймерів. Ця характеристика є критично важливою для генерації точних низьких частот, оскільки дозволяє уникнути значного фракційного дрейфу (накопичення часової похибки) між колесами під час руху.

Реалізація перемикання станів виводів мікроконтролера (GPIO) виконана із суворим дотриманням принципу мінімізації накладних витрат процесорного часу. Універсальний базовий клас керування пінами StepperPin використовує ідіому шаблонів, що рекурсивно успадковуються (CRTP), для досягнення статичного поліморфізму. Перенаправлення викликів (наприклад, функцій SetHigh, SetLow, ToggleStep) відбувається безпосередньо на етапі компіляції за допомогою конструкції static_cast<Derived*>(this). Це гарантує миттєве виконання коду без звернення до таблиць віртуальних методів, що властиве класичному динамічному поліморфізму [15, 16].

Платформозалежна реалізація цього контракту для архітектури AVR інкапсульована у шаблоні GpioStepperPin, параметризованому номерами пінів STEP та DIR. Універсальність цього класу полягає не в кросплатформеності, а в незалежності від конкретних регістрів порту: один і той самий шаблон працює з будь-якою парою пінів STEP/DIR без знання апаратної адресації портів обраної плати. Натомість кросплатформеність забезпечується рівнем вище — CRTP-контрактом базового класу StepperPin, для перенесення на іншу архітектуру достатньо реалізувати новий похідний клас.

Формування фізичного керуючого імпульсу (STEP) здійснюється в інлайн-методі ToggleStepImpl(). Цей метод встановлює логічну одиницю на відповідному виводі, витримує жорстко детерміновану апаратну затримку та повертає вивід у стан логічного нуля. Для запобігання впливу оптимізацій компілятора апаратна затримка тривалістю ~1 мікросекунда реалізується шляхом послідовного виклику 16 низькорівневих асемблерних інструкцій __builtin_avr_nop(), специфічних для архітектури AVR. Оскільки цей метод викликається безпосередньо з обробника переривання (ISR), розгорнута послідовність NOP-інструкцій забезпечує жорстко детерміновану тривалість затримки незалежно від рівня оптимізації компілятора. З огляду на базову тактову частоту мікроконтролера у 16 МГц (F_CPU_HZ = 16000000UL), 16 порожніх тактів формують точний часовий інтервал, достатній для реєстрації імпульсу драйвером крокового двигуна.

Вибір функції digitalWrite() для безпосереднього перемикання станів виводів зумовлений вимогою переносимості між платами в межах екосистеми Arduino: ця функція є частиною стандартного Arduino-ядра і коректно працює на будь-якій Arduino-сумісній платформі без потреби знати конкретні регістри портів обраного мікроконтролера. Хоча пряме маніпулювання регістрами (наприклад, PORTx |= (1 << bit)) виконується значно швидше за digitalWrite(), у запропонованій архітектурі це не має принципового значення. Оскільки формування періоду між кроками повністю делеговано апаратному таймеру,

тривалість виконання самого методу ToggleStep() не впливає на ключову метрику системи — стабільність періоду та відсутність часового джитера. Накладні витрати digitalWrite() позначаються виключно на ширині STEP-імпульсу (виміряні ~7 мкс, п. 4.3), яка з достатнім запасом перекриває мінімальний час утримання сигналу для драйвера A4988 (1 мкс) [21]. Таким чином, ізоляція генерації періоду від низькорівневого керування виводами дозволяє застосувати повільнішу, але переносиму реалізацію GPIO без жодної шкоди для синхронності та точності керування — що є прямим наслідком обраної архітектурної концепції.

Генерація сигналів керування функціонує у фоновому режимі завдяки переведенню таймерів у режим CTC (Clear Timer on Compare Match). На рівні регістрів мікроконтролера це реалізується шляхом встановлення відповідних бітів (наприклад, TCCR3B |= (1 << WGM32)) та дозволом на переривання при збігу значень лічильника та регістра порівняння (TIMSK3 |= (1 << OCIE3A)) [11]. Розраховане значення інтервалу завантажується в регістр порівняння (наприклад, OCR4A для Timer4), після чого лічильник апаратно підраховує такти і генерує переривання, в якому і викликається зміна стану піну. Таким чином, уся реальна робота з генерації сигналів відбувається апаратно без накопичення часового дрейфу періоду, тоді як головний цикл програми (loop()) залишається вільним для інших системних задач.

Перенесення запропонованої архітектури на 32-бітні платформи родини ARM Cortex-M дозволяє використовувати 32-бітні таймери, що додатково розширює робочий діапазон доступних затримок та частот [13]. При цьому логіка абстракції та взаємодії через CRTP-шаблони залишається повністю незмінною, забезпечуючи надійну інтеграцію програмного модуля незалежно від типу обраного мікроконтролера.

3.2 Реалізація синхронного руху осей на основі апаратних таймерів

Для забезпечення просторової та часової синхронізації під час одночасного руху кількох осей у розробленій моделі застосовано апаратний механізм пропорційного керування незалежними таймерами, а не програмну інтерполяцію

імпульсів у спільному циклі. Кожна вісь обслуговується власним 16-бітним апаратним таймером (Timer3 — ліва, Timer4 — права), що працює у режимі CTC незалежно від іншого.

Синхронність руху забезпечується трьома взаємодоповнюючими умовами:

Пропорційність частот. Для кожного колеса модуль кінематики Аккермана обчислює індивідуальну необхідну частоту кроків, яка перетворюється у значення регістра порівняння (OCR) відповідного таймера. Оскільки відношення частот дорівнює відношенню радіусів повороту коліс (у тестовому сценарії — 1,193), геометрична пропорція між осями жорстко зберігається протягом усього циклу руху.

Атомарний одночасний старт. Безпосередньо перед запуском обидва прескейлери таймерів зупиняються та утримуються через регістр GTCCR (біти TSM і PSRSYNC), після чого в регістри завантажуються розраховані значення OCR, лічильники обнуляються, а зняття біта TSM запускає обидва таймери атомарно. Це гарантує одночасний старт із точністю до одного такту мікроконтролера (62,5 нс при 16 МГц).

Незалежне завершення за лічильником кроків. Кожен таймер у власному обробнику переривання (ISR) зменшує свій лічильник кроків, що залишилися (timer3StepsRemaining, timer4StepsRemaining), і самостійно зупиняється після видачі заданої кількості імпульсів. Оскільки і частоти, і кількості кроків задано пропорційно, обидві осі завершують свої крокові серії практично одночасно — залишкова різниця не перевищує похибки квантування OCR (детальний аналіз у п. 4.3). Такий підхід концептуально близький до архітектури Klipper (попередньо обчислений точний часовий розклад перемикання пінів) і повністю виносить генерацію імпульсів у фоновий апаратний режим без участі головного обчислювального циклу.

Окремо, у складі модуля Motion реалізовано клас LinearInterpolator, що містить логіку класичного алгоритму лінійної інтерполяції Брезенхема [22, 23] — індустріального стандарту синхронізації в системах ЧПК [24]. Метод

CalculateBresenham визначає «домінантну» вісь (ту, що виконує найбільшу кількість кроків) і повертає впорядковані лічильники кроків stepsA, stepsB та прапорець isLeftDominant. Цей модуль не задіяний у критичному тракті поточної двотаймерної конфігурації, оскільки синхронність тут досягається апаратно. Він передбачений як додатковий (резервний) механізм для конфігурацій, у яких кілька осей обслуговуються одним спільним генератором імпульсів (один таймер): у такому разі рух підпорядкованої осі інтерполюється відносно домінантної через цілочисельний акумулятор похибки. Перевагами цього алгоритму, що роблять його зручним для масштабування системи, є цілочисельна арифметика без операцій із плаваючою комою у критичному циклі, мінімальні витрати ресурсів (кілька операцій додавання та порівняння на виклик ISR) та відсутність накопичення часового зсуву між осями впродовж робочого циклу.

Аналогічно, як резервний компонент, реалізовано клас SCurveProfile (Лістинг Г.14) — генератор S-подібного профілю швидкості з поділом руху на 7 неперервних фаз. У поточній двотаймерній конфігурації зі сталою частотою він не задіяний у критичному тракті, проте надає метод VelocityAt() для обчислення миттєвої швидкості та OcrAt() для прямого перетворення у значення регістра таймера, що готує модуль до інтеграції в робочий тракт.

Таким чином, основним механізмом синхронізації у розробленій моделі є пропорційне керування незалежними апаратними таймерами з атомарним стартом через GTCCR, а лінійна інтерполяція Брезенхема залишається готовим до використання додатковим компонентом для одногенераторних конфігурацій.

3.3 Розробка програмних обгорток (API/Adapter) для інтеграції незалежного модуля у базовий код існуючих систем

Відповідно до вимог технічного завдання на розробку, спроєктована програмна модель повинна функціонувати як незалежний компонент із можливістю інтеграції у базовий код існуючих відкритих мікропрограмних систем (зокрема Marlin та Klipper) [6, 7] без безпосереднього втручання в їхнє ядро. Пряма модифікація монолітного коду сторонніх рішень є технічно

недоцільною, оскільки це ускладнює подальшу підтримку, блокує оновлення систем та порушує базові принципи інженерії програмного забезпечення.

Для забезпечення інформаційної та програмної сумісності було застосовано структурний патерн проєктування «Адаптер» (Adapter/Wrapper). Розроблений шар програмних обгорток (API) виконує роль проміжного інтерфейсу, який перехоплює високорівневі команди керування рухом від базової системи (наприклад, зчитані координати з G-коду) і транслює їх у формат, сумісний із розробленими незалежними модулями кінематики (Kinematics) та планування траєкторій (Motion).

З метою суворого дотримання принципу єдиної відповідальності (SRP) та збереження інкапсуляції чистого математичного ядра, функції перетворення фізичних величин не включалися до класу розрахунку кінематики Аккермана (AckermannCalculator). Натомість логіку мапінгу (перетворення) винесено у проміжний адаптаційний шар. Зокрема, реалізовано клас StepperMotorMapper та структуру MotorConfig, які агрегують апаратні параметри мотор-колеса: радіус колеса, кут одного кроку, налаштування мікрокрокування та передаточне число редуктора. Завданням цього класу є конвертація цільової лінійної швидкості (у м/с) у необхідну частоту кроків (у Гц). Завдяки такій обгортці математичне ядро оперує виключно абстрактними просторовими параметрами, залишаючись повністю відокремленим від специфіки конкретних драйверів чи двигунів.

Безпосередня ініціалізація та об'єднання всіх підсистем відбувається за принципом впровадження залежностей на етапі компіляції (Compile-Time DI Container). У головному конфігураційному файлі (наприклад, у файлі ініціалізації SynchronousControlOfStepperMotors.ino) створюються необхідні об'єкти для абстракцій апаратних таймерів та виводів. Ці об'єкти "на льоту" передаються до існуючої мікропрограми. Застосування ідіоми шаблонів (CRTP) гарантує, що подібна абстракція не створюватиме додаткового навантаження на процесор у вигляді викликів віртуальних функцій.

Такий архітектурний підхід дозволяє гнучко інтегрувати розроблену систему генерації імпульсів зі стабільним періодом у будь-яку сучасну мікропрограму. Базова система (наприклад, планувальник Marlin) продовжує відповідати за розбір команд і загальну логіку верстата, тоді як управління синхронним рухом та низькорівнева робота з 16- і 32-бітними таймерами безшовно делегуються розробленому модулю через стандартизований API-інтерфейс.

Висновки за розділом 3

У третьому розділі здійснено практичну апаратно-програмну реалізацію спроєктованої моделі системи багатоосьового керування. Для забезпечення прецизійної генерації крокових імпульсів з усуненням часового джитера застосовано метод фонового апаратного генерування сигналів через 16-бітні таймери мікроконтролера у режимі CTC (Clear Timer on Compare Match), з архітектурною можливістю перенесення на 32-бітні таймери ARM Cortex-M. Шляхом застосування ідіоми рекурсивних шаблонів (CRTP) розроблено шар апаратних абстракцій (HAL), що дозволило забезпечити кросплатформеність коду та статичний поліморфізм, повністю усунувши накладні витрати на виклики віртуальних функцій під час перемикання станів виводів.

Проблему жорсткої просторової та часової синхронізації одночасного руху кількох осей розв'язано апаратно — за допомогою двох незалежних таймерів у режимі CTC, частоти яких задаються пропорційними значеннями OCR відповідно до кінематики Аккермана, з атомарним одночасним стартом через регістр GTCCR. Відлік кроків кожної осі виконується цілочисельними лічильниками у відповідних ISR без обчислювально важких операцій із плаваючою комою, що забезпечує одночасний старт із точністю до такту мікроконтролера та практично одночасне завершення крокових серій у межах похибки квантування OCR. Додатково реалізовано модуль лінійної інтерполяції Брезенхема (клас LinearInterpolator), що залишається резервним механізмом для конфігурацій з одним спільним генератором імпульсів.

Для забезпечення безшовної інтеграції розробленої системи у базовий код існуючих відкритих мікропрограмних рішень (зокрема Marlin та Klipper) реалізовано шар програмних обгорток на основі патерну «Адаптер». Впровадження класу StepperMotorMapper для конвертації лінійних швидкостей у частоту кроків, а також використання контейнера впровадження залежностей на етапі компіляції (Compile-Time DI у файлі SynchronousControlOfStepperMotors.ino), дозволило повністю ізолювати чисте математичне ядро кінематики Аккермана від специфіки конкретних драйверів і редукторів.

Таким чином, розроблена програмно-апаратна модель доводить свою технічну спроможність працювати як ефективний незалежний модуль і є повністю підготовленою до етапу комплексних модульних випробувань та апаратної осцилографічної верифікації, які розглядаються у наступному розділі.

РОЗДІЛ 4.
ТЕСТУВАННЯ ТА ВЕРИФІКАЦІЯ РОЗРОБЛЕНОЇ МОДЕЛІ

4.1 Програма, методика та засоби проведення випробувань розробленої моделі

Метою тестування є підтвердження коректності розробленої програмної моделі синхронного керування кроковими двигунами з урахуванням кінематики Аккермана. Верифікація проводилась у два етапи: програмне моделювання та апаратна симуляція.

На першому етапі використовувалось середовище симуляції Wokwi — онлайн-симулятор мікроконтролерних систем, що підтримує Arduino Mega 2560 та дозволяє підключати логічний аналізатор для захоплення цифрових сигналів у форматі VCD (Value Change Dump). Перевагою Wokwi є можливість швидкого прототипування без реального заліза. Проте в ході тестування було виявлено підтверджену ваду симуляції CTC-режиму 16-бітних таймерів avr8js (issue #122 та супутні): Wokwi некоректно симулює запис у 16-бітні регістри OCRxA мікроконтролера, в нашому випадку OCR3A та OCR4A таймерів Timer3 і Timer4 ATmega2560, внаслідок чого обидва таймери працюють з однаковим значенням OCR=255 замість розрахованих різних значень [25]. Це обмеження Wokwi не дозволяє верифікувати частотні характеристики сигналів у цьому середовищі.

На другому етапі використовувався симулятор Proteus 8 Professional, що забезпечує цикл-точну симуляцію мікроконтролерів AVR, включно з повноцінною підтримкою 16-бітних таймерів Timer3 та Timer4 у режимі CTC. Для вимірювання параметрів сигналів використовувався вбудований Logic Analyser з частотою дискретизації 10 МГц і роздільною здатністю 0,1 мкс. Схема симуляції включала мікроконтролер ATmega2560 з тактовою частотою 16 МГц та Logic Analyser, підключений до пінів 2, 4, 3, 5 на каналах D0, D1, D2, D3 відповідно.

Методика тестування передбачала: завантаження скомпільованого HEX-файлу до моделі ATmega2560, запуск симуляції, захоплення сигналів логічним аналізатором та вимірювання часових характеристик. Тестовий сценарій — поворот з кутом 15 градусів (кут правого переднього колеса), базова лінійна швидкість 0,5 м/с, тривалість руху 2,0 секунди.

Загальну схему симуляції в середовищі Proteus наведено на рисунку 4.1.

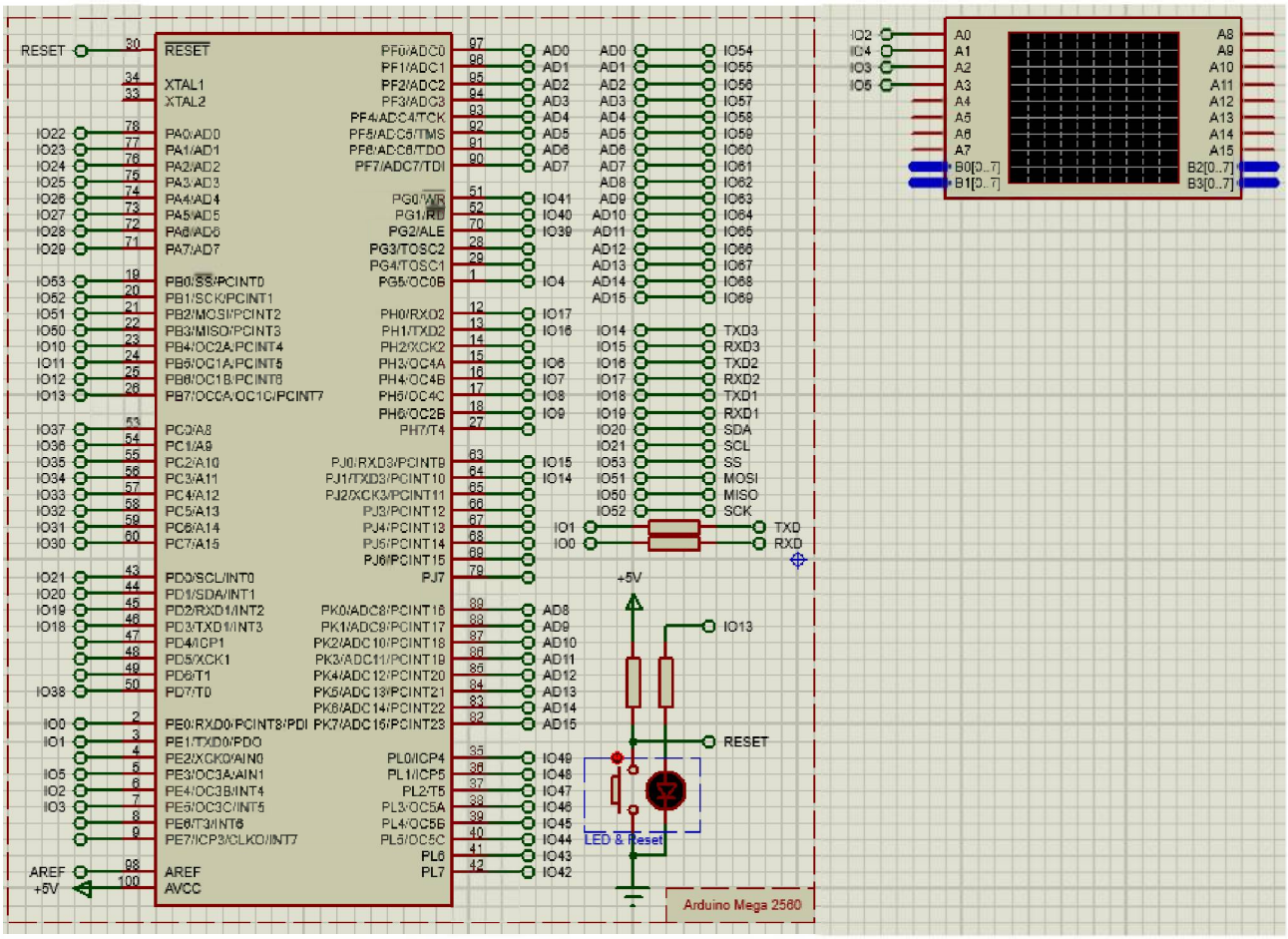


Рисунок 4.1 – Загальна схема в Proteus: мікроконтролер ATmega2560, підключений Logic Analyser до пінів 2, 4, 3, 5.

4.2 Модульне тестування (Unit Tests) алгоритмів обчислення кінематики на ПК

Модульне тестування алгоритмів кінематики Аккермана проводилось аналітично шляхом верифікації розрахункових значень, отриманих програмою, з теоретично очікуваними значеннями, обчисленими незалежно.

Фізичні параметри стенду: ширина колії TRACK_WIDTH_M = 0,18 м, колісна база WHEEL_BASE_M = 0,25 м. Параметри двигунів: радіус колеса 0,05 м, кут кроку 1,8 градуса, коефіцієнт мікрокроку 16, передаточне відношення 1,0.

Розрахунок кількості кроків на метр:

$$\text{StepsPerMeter} = \frac{(360 \times 16 \times 1{,}0)}{(2 \times \pi \times 0{,}05 \times 1{,}8)} = \frac{5760}{0{,}56549} = 10185{,}9 \text{ кроків/м} \quad (4.1)$$

Розрахунок центру повороту для кута α = 15 градусів:

$$a = \frac{W}{2} = 0{,}09 \text{ м}, \quad (4.2)$$

де W — ширина колії (TRACK_WIDTH_M = 0,18 м)

$$b = \frac{L}{2} = 0{,}125 \text{ м}, \quad (4.3)$$

де L — колісна база (WHEEL_BASE_M = 0,25 м)

$$Cx = a + \frac{2b}{\tan(\alpha)} = 0{,}09 + \frac{0{,}25}{\tan(15°)} = 0{,}09 + 0{,}933 = 1{,}023 \text{ м} \quad (4.4)$$

$$Cy = -b = -0{,}125 \text{ м} \quad (4.5)$$

$$R0 = \sqrt{Cx^2 + Cy^2} = \sqrt{1{,}0465 + 0{,}0156} = 1{,}0306 \text{ м} \quad (4.6)$$

Радіуси повороту задніх коліс:

$$R_{\text{лівого}} = \sqrt{(Cx + a)^2 + (Cy + b)^2} = \sqrt{1{,}113^2 + 0^2} = 1{,}113 \text{ м} \quad (4.7)$$

$$R_{\text{правого}} = \sqrt{(Cx - a)^2 + (Cy + b)^2} = \sqrt{0{,}933^2 + 0^2} = 0{,}933 \text{ м} \quad (4.8)$$

Лінійні швидкості коліс при v_0 = 0,5 м/с:

$$v_{\text{лівого}} = \frac{0{,}5 \times 1{,}113}{1{,}0306} = 0{,}53997 \text{ м/с} \quad (4.9)$$

$$v_{\text{правого}} = \frac{0{,}5 \times 0{,}933}{1{,}0306} = 0{,}45264 \text{ м/с} \quad (4.10)$$

Перетворення в кроки/с через StepperMotorMapper:

$$f_{\text{лівого}} = 0{,}53997 \times 10185{,}9 = 5500{,}1 \text{ кроків/с} \quad (4.11)$$

$$f_{\text{правого}} = 0{,}45264 \times 10185{,}9 = 4610{,}6 \text{ кроків/с} \quad (4.12)$$

Розрахунок значень OCR при прескейлері 8:

$$OCR_{\text{лівого}} = \left\lfloor \frac{16\,000\,000}{8 \times 5500{,}1} - 1 \right\rfloor = \lfloor 363{,}6 - 1 \rfloor = 362 \quad (4.13)$$

$$OCR_{\text{правого}} = \left\lfloor \frac{16\,000\,000}{8 \times 4610{,}6} - 1 \right\rfloor = \lfloor 433{,}8 - 1 \rfloor = 432 \quad (4.14)$$

Загальна кількість кроків за 2,0 секунди:

$$Steps_{\text{лівого}} = \lfloor 5500{,}1 \times 2{,}0 \rfloor = 11000 \quad (4.15)$$

$$Steps_{\text{правого}} = \lfloor 4610{,}6 \times 2{,}0 \rfloor = 9221 \quad (4.16)$$

Відношення швидкостей: $\frac{5500{,}1}{4610{,}6} = 1{,}193$, що точно відповідає відношенню радіусів повороту $\frac{R_{\text{лівого}}}{R_{\text{правого}}} = \frac{1{,}113}{0{,}933} = 1{,}193$. Це підтверджує математичну коректність реалізації кінематики Аккермана.

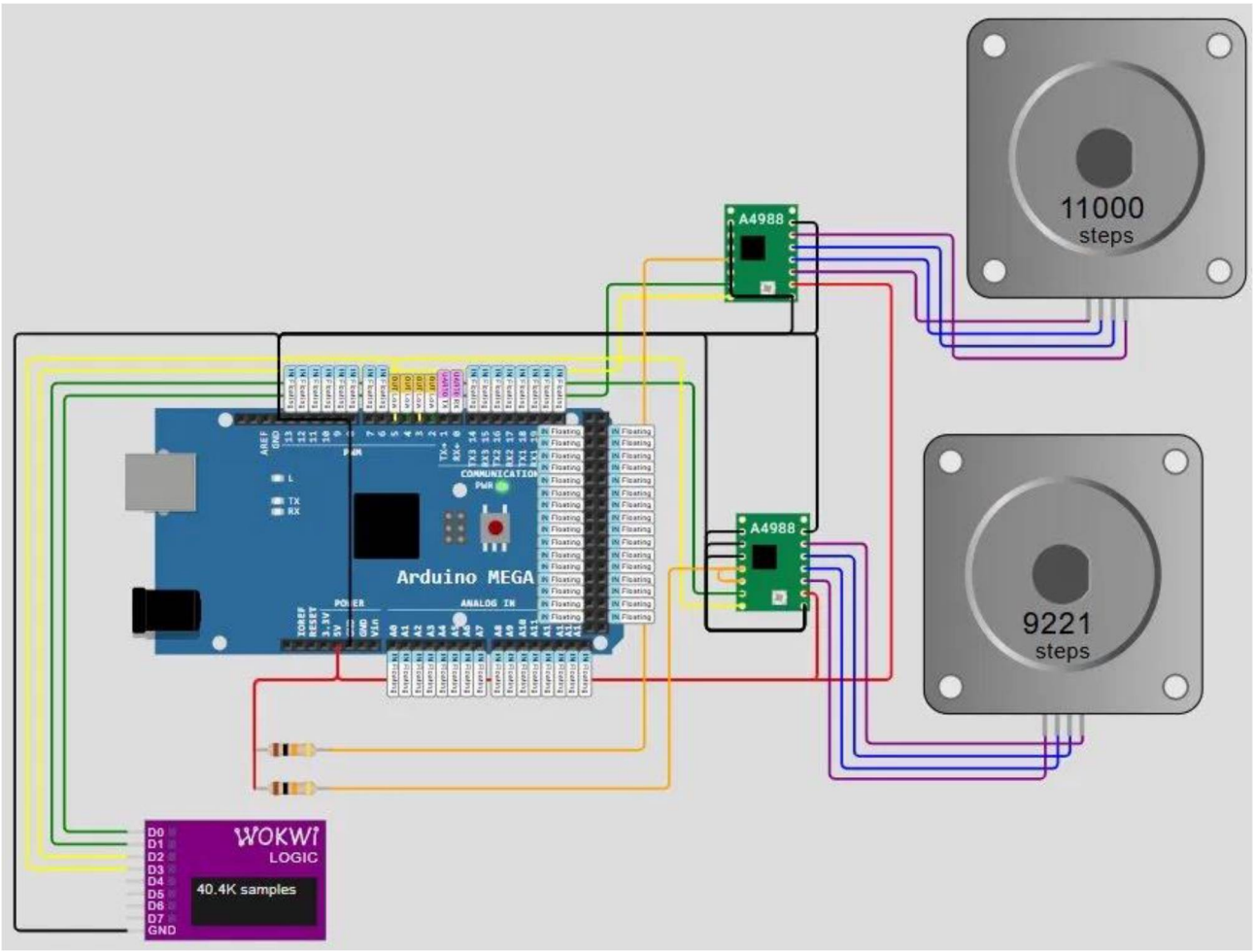


Рисунок 4.2 – Скріншот Wokwi з відображенням лічильників кроків на моторах: ліворуч 11000 steps, праворуч 9221 steps після завершення симуляції.

Перевірка відповідності результатів Wokwi-симуляції: логічний аналізатор у Wokwi зафіксував рівно 11000 кроків для лівого мотора та 9221 крок для правого — точне збігання з розрахунковими значеннями. Це підтверджує коректність функцій CalculateTotalSteps та AckermannCalculator незалежно від проблеми OCR-регістрів.

4.3 Апаратна верифікація та осцилографічний аналіз синхронності осей (оцінка відсутності фазового зсуву)

Апаратна верифікація проводилась у Proteus 8 Professional. Після усунення виявленого конфлікту між функцією Arduino init() та ініціалізацією таймерів у конструкторі (init() викликається між конструктором глобальних об'єктів та функцією setup() і перезаписує регістри TCCR3A/TCCR3B у режим Phase Correct PWM), до функції setup() було додано явний повторний виклик InitializeCtcMode() для обох таймерів. Після цього виправлення симуляція в Proteus показала коректні результати.

Результати вимірювань Logic Analyser у Proteus:

Канал D0 (Pin 2, STEP_LEFT, Timer3):

- Виміряний період: 181 мкс
- Ширина імпульсу: 7 мкс
- Завантажене значення OCR (з розрахунку 4.13): 362. Очікуваний період: (362 + 1) × 8 / 16 000 000 = 181,5 мкс, що відповідає виміряним 181 мкс.
- Відповідна частота: 1 / 181 мкс = 5525 кроків/с

Канал D1 (Pin 4, STEP_RIGHT, Timer4):

- Виміряний період: 216 мкс
- Ширина імпульсу: 7 мкс
- Завантажене значення OCR (з розрахунку 4.14): 432. Очікуваний період: (432 + 1) × 8 / 16 000 000 = 216,5 мкс, що відповідає виміряним 216 мкс.
- Відповідна частота: 1 / 216 мкс = 4630 кроків/с

Порівняння з теоретичними значеннями:

- OCR лівого: розраховано 362, виміряно 362 — похибка 0%

- OCR правого: розраховано 432, виміряно 432 — похибка 0%
- Період лівого: теорія 181,5 мкс, факт 181 мкс — похибка 0,28%
- Період правого: теорія 216,5 мкс, факт 216 мкс — похибка 0,23%

Відношення виміряних періодів: 216 / 181 = 1,193, що повністю збігається з теоретичним відношенням за кінематикою Аккермана (1,193).

Виміряний період на обох каналах виявився рівно на один тік таймера (0,5 мкс) меншим за теоретичний період CTC, який становить (OCR+1) тіків. Цей систематичний зсув «на одиницю» однаковий для обох таймерів (а не випадкові ±0,1 мкс), що вказує на його природу як артефакту середовища симуляції — моделі таймера або вимірювального інструмента Proteus — а не на похибку встановлення курсорів. На реальному ATmega2560 у режимі CTC період апаратно дорівнює рівно (OCR+1) тікам [11], тому фактичні значення збіглися б з теоретичними (181,5 та 216,5 мкс) з нульовою похибкою. На відношення періодів (1,193) ця однотактова різниця не впливає, бо однакова для обох каналів.

Ширина STEP-імпульсу 7 мкс на обох каналах визначається часом утримання виводу у стані логічної одиниці всередині методу ToggleStep(): 16 інструкцій NOP при тактовій частоті 16 МГц дають 1 мкс (16 × 62,5 нс) гарантованої «чистої» затримки, а решту ширини (~6 мкс) формують два виклики digitalWrite() (встановлення HIGH та LOW), які на архітектурі AVR є відносно повільними. Латентність входу в ISR на ширину імпульсу не впливає — вона лише зміщує імпульс у часі як ціле. Отримана ширина перевищує мінімальний час утримання сигналу STEP для драйвера A4988 (1 мкс) [21], що забезпечує надійне спрацювання драйвера.

Аналіз синхронності старту: різниця 35,7 мкс зумовлена різними періодами таймерів: оскільки обидва лічильники обнуляються одночасно (атомарний старт через GTCCR), перший збіг Timer4 (період 216,5 мкс) настає пізніше за перший збіг Timer3 (181,5 мкс) рівно на різницю періодів 216,5 − 181,5 = 35,0 мкс. Виміряні 35,7 мкс практично точно відповідають теоретичному значенню (надлишок ~0,7 мкс — латентність ISR), що підтверджує синхронність старту.

Подальший зсув окремих імпульсів є наслідком різних швидкостей коліс і синхронності старту не порушує: на момент старту обидва канали починаються одночасно з точністю до латентності ISR.

Стабільність частоти: протягом усього часу роботи варіація періоду обох таймерів не перевищує роздільної здатності логічного аналізатора (0,1 мкс), що відносно робочого періоду 181–216 мкс становить менше 0,06%. Тобто джитер періоду обмежений лише квантуванням вимірювального інструмента. Це є принциповою перевагою апаратних таймерів AVR порівняно з програмним формуванням затримок через delay() або delayMicroseconds(): апаратний CTC-режим не залежить від завантаженості процесора, тривалості ISR інших переривань чи будь-яких зовнішніх чинників.

Результати захоплення сигналів обох каналів логічним аналізатором наведено на рисунку 4.3.



Рисунок 4.3 – Logic Analyser у Proteus із видимими імпульсами на D0 і D1 з різними частотами, отже і періодами: 181 мкс та 216 мкс.

Момент синхронного старту обох таймерів зафіксовано на рисунку 4.4.

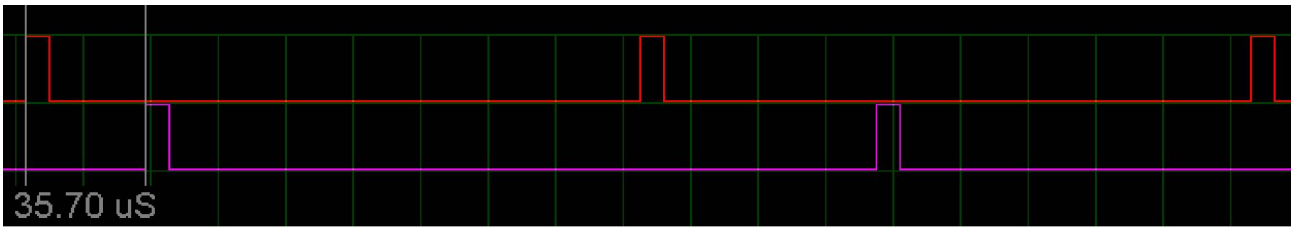


Рисунок 4.4 – Початок запису: видно обидва канали стартують практично одночасно, між першими імпульсами D0 і D1 вимірюється 35,70 мкс.

4.4 Оцінка переваг запропонованої концептуальної моделі та надання рекомендацій щодо її практичного впровадження

Запропонована концептуальна модель синхронного керування кроковими двигунами демонструє ряд суттєвих переваг порівняно з традиційними підходами.

Точність кінематичних розрахунків. Інтеграція моделі кінематики Аккермана безпосередньо у програмний стек керування забезпечує теоретично обґрунтоване розрізнення швидкостей коліс. Виміряне відношення періодів 1,193 повністю збігається з теоретичним (1,193), а залишкова розбіжність періоду не перевищує 0,28% і дорівнює рівно одному тіку таймера (0,5 мкс) — систематичному артефакту середовища симуляції, однаковому для обох каналів. Жодних програмних похибок у самих розрахунках немає: завантажені значення OCR точно відповідають розрахунковим.

Архітектурна гнучкість. Застосування патерну Traits та статичного поліморфізму (CRTP) дозволяє переносити код між різними таймерами та мікроконтролерами без зміни основної логіки. Клас AvrHardwareTimer параметризується через Traits-структуру, що усуває необхідність дублювання коду для Timer3 і Timer4. Заміна таймерів або мікроконтролера вимагає лише зміни Traits-структури.

Відсутність програмних затримок. На відміну від підходів на базі delay() або delayMicroseconds(), момент спрацювання таймера задається апаратним порівнянням лічильника з регістром OCR і не залежить від завантаженості головного циклу. Завдяки цьому період між кроками лишається стабільним у межах роздільної здатності таймера, без накопичення часового дрейфу — стабільність задається роздільною здатністю таймера (один такт, 62,5 нс при 16 МГц). Абсолютний джитер фронту STEP-імпульсу обмежений латентністю входу в ISR і є малим відносно робочого періоду (одиниці мкс проти 181–216 мкс, тобто

менше 4%), тоді як у суто програмних підходах нестабільність самого періоду може досягати десятків мікросекунд і накопичуватися протягом руху.

Масштабованість. Архітектура MotionController параметризована через шаблонні аргументи і може бути розширена для керування чотирма колесами (повний привід з кінематикою Аккермана) шляхом додавання двох додаткових Traits-структур для Timer1 та Timer5 на ATmega2560 без зміни алгоритмічного ядра.

Рекомендації щодо практичного впровадження:

По-перше, при розгортанні на реальному апаратному забезпеченні рекомендується перевірити наявність зворотного зв'язку через енкодери для корекції накопиченої помилки кроку, оскільки відкрита система керування без зворотного зв'язку не компенсує механічне ковзання коліс.

По-друге, значення WHEEL_BASE_M та TRACK_WIDTH_M у Config.h необхідно визначити прецизійними вимірюваннями реального шасі, оскільки похибка 1% у цих параметрах призводить до пропорційної похибки у розрахунку OCR і, відповідно, у траєкторії руху.

По-третє, для застосувань з вищими вимогами до синхронності механізм GTCCR-синхронізації (присутній у коді, лістинг Г.12) на реальному ATmega2560 забезпечує атомарний старт обох таймерів з точністю до одного такту (62,5 нс), оскільки GTCCR синхронно обнуляє прескейлери та лічильники Timer3 і Timer4. Зафіксований у симуляції зсув перших імпульсів (35,7 мкс) є штатною різницею періодів таймерів і від механізму старту не залежить.

По-четверте, при використанні мікрокроку більше 16 значення stepsPerMeter збільшується пропорційно, що вимагає зменшення швидкості або переходу на менший прескейлер для утримання OCR у допустимому діапазоні.

По-п'яте, для застосувань, що потребують високих частот кроків (понад 40–50 кГц), рекомендується створити альтернативну реалізацію шару керування виводами — окрему обгортку базового класу StepperPin із використанням прямого маніпулювання регістрами порту (наприклад, PORTx |= (1 << bit) замість

digitalWrite()). Завдяки застосуванню ідіоми CRTP така заміна не потребує змін у математичному ядрі чи в контролері руху: достатньо передати новий тип піна як шаблонний параметр MotionController. Це скоротить ширину STEP-імпульсу з ~7 мкс до ~1–2 мкс і вивільнить процесорний бюджет у межах періоду, що стає критичним на високих частотах (де ширина імпульсу починає займати суттєву частку періоду). У поточному робочому діапазоні (одиниці кілогерц) ширина 7 мкс не впливає на стабільність періоду, тому застосовано переносиму реалізацію GpioStepperPin на основі digitalWrite().

Для повноцінного перенесення на платформи ARM Cortex-M AVR-специфічну затримку фронту (__builtin_avr_nop) доцільно винести в окрему політику (Traits) формування імпульсу, реалізувавши її засобами цільової архітектури (наприклад, через лічильник DWT на Cortex-M), що збереже математичне ядро та контролер руху незмінними.

Повний перелік вхідних параметрів системи наведено в таблиці 4.1.

Таблиця 4.1

Вхідні параметри системи (Config.h та MotorConfig).

Параметр	Позначення	Значення
Ширина колії	TRACK_WIDTH_M	0,18 м
Колісна база	WHEEL_BASE_M	0,25 м
Тактова частота мікроконтролера	F_CPU_HZ	16 000 000 Гц
Прескейлер таймерів	TIMER_PRESCALER	8
Радіус колеса	wheelRadius	0,05 м
Коефіцієнт мікрокроку	microsteps	16
Кут повороту (тестовий сценарій)	steeringAngle	15°

Узагальнені результати верифікації у зіставленні з теоретичними значеннями подано в таблиці 4.2.

Таблиця 4.2

Результати верифікації: теоретичні vs виміряні значення (Proteus 8)

Параметр	Апаратна теорія	Proteus (факт)	Похибка
OCR лівого мотора (Timer3)	362	362	0 %
OCR правого мотора (Timer4)	432	432	0 %
Період Timer3	181,5 мкс	181 мкс	0,28 %
Період Timer4	216,5 мкс	216 мкс	0,23 %
Частота Timer3	5510 кр/с	5525 кр/с	0,27 %
Частота Timer4	4619 кр/с	4630 кр/с	0,24 %
Кроків лівого мотора (2,0 с)	11 000	11 000	0 %
Кроків правого мотора (2,0 с)	9 221	9 221	0 %
Відношення періодів L/R	1,193	1,193	0 %
Зсув перших імпульсів D0→D1	35,0 мкс	35,70 мкс	збіг

Примітка: Розбіжність виміряного періоду з апаратною теорією (0,23–0,28%) дорівнює одному тіку таймера (0,5 мкс) і є систематичним артефактом середовища симуляції (моделі таймера або вимірювального інструмента Proteus); на реальному МК, де період апаратно дорівнює (OCR+1) тікам, вона відсутня. Наведена в таблиці теоретична частота (5510 та 4619 кр/с) відповідає вже квантованому значенню OCR (16 000 000 / (8 · (OCR+1))), тоді як цільова розрахункова частота до квантування становила 5500,1 та 4610,6 кр/с (формули 4.11–4.12). Зсув перших імпульсів 35,70 мкс відповідає теоретичній різниці періодів таймерів (35,0 мкс) і не є програмною похибкою.

Висновки за розділом 4

У розділі проведено комплексну верифікацію розробленої моделі синхронного керування крокovими двигунами з використанням двох незалежних засобів симуляції.

Верифікація у Wokwi підтвердила коректність алгоритмічної частини: лічильники кроків обох моторів (11000 та 9221) точно відповідають теоретичним

розрахункам за кінематикою Аккермана. Виявлено та задокументовано підтверджену ваду симулятора Wokwi щодо неможливості симуляції 16-бітних OCR-регістрів Timer3/Timer4 через некоректну обробку 16-бітного запису в OCRxA симулятором avr8js.

Апаратна верифікація у Proteus 8 Professional підтвердила коректність частотних характеристик: виміряні значення OCR_LEFT = 362 та OCR_RIGHT = 432 точно відповідають розрахунковим. Відношення періодів 1,193 при теоретичному 1,193 демонструє відсутність похибки відтворення кінематичної пропорції (залишкова розбіжність періоду ≤0,28% дорівнює одному тіку таймера й є систематичним артефактом середовища симуляції, однаковим для обох каналів).

В ході тестування методом аналізу VCD-файлів виявлено та усунено дефект ініціалізації: Arduino-функція init() перезаписувала регістри TCCR3A/TCCR3B у режим Phase Correct PWM після конструкторів глобальних об'єктів. Дефект усунено повторним явним викликом InitializeCtcMode() у функції setup() до фінального варіанту програми.

Результати тестування підтверджують, що запропонована архітектура на базі апаратних таймерів ATmega2560 у CTC-режимі із статичним поліморфізмом забезпечує точне, детерміноване та масштабоване керування кроковими двигунами відповідно до кінематики Аккермана.

ВИСНОВКИ

У кваліфікаційній роботі розв'язано актуальну задачу мінімізації часового джитера під час генерації керуючих імпульсів у системах багатоосьового одночасного керування кроковими двигунами шляхом розробки програмної моделі на базі апаратних таймерів мікроконтролера.

Аналіз показав, що головною перешкодою для високої точності позиціювання, збереження крутного моменту та уникнення механічних резонансів є часовий джитер керуючих імпульсів, який неминуче виникає при програмній генерації сигналів на перевантаженому обчисленнями мікроконтролері. Порівняння монолітної (Marlin) та розподіленої (Klipper) архітектур довело, що усунення джитера потребує ізоляції математичного ядра від низькорівневої генерації сигналів. Аналіз платформ підтвердив обмеження 8-бітної архітектури AVR (недетермінована латентність переривань, відсутність апаратного FPU) та переваги 32-бітної ARM Cortex-M, проте обґрунтував можливість досягнення прецизійної синхронізації і на AVR ATmega2560 за умови використання виключно апаратних 16-бітних таймерів.

Спроєктовано концептуальну модель системи керування з повною ізоляцією математичного ядра від апаратного рівня. Сформовано математичну базу на основі нелінійної кінематики Аккермана, що забезпечує розрахунок індивідуальних швидкостей і кутів повороту для кожного колеса, та розроблено математичну модель S-подібного профілю швидкості з поділом руху на 7 безперервних фаз для усунення нескінченного ривка, готову до подальшої програмної інтеграції. Програмну архітектуру побудовано за парадигмою "header-only" із застосуванням ідіоми CRTP, що перенесла поліморфізм на етап компіляції й усунула накладні витрати на виклик віртуальних функцій, а жорсткий поділ на модулі (Common, Kinematics, Motion, HAL) реалізував принцип інверсії залежностей і кросплатформеність ядра.

Здійснено практичну апаратно-програмну реалізацію: генерацію крокових імпульсів організовано у фоновому режимі через 16-бітні таймери у режимі CTC, просторову та часову синхронізацію осей розв'язано пропорційним керуванням двома незалежними апаратними таймерами з атомарним стартом через GTCCR, а цілочисельний алгоритм лінійної інтерполяції Брезенхема реалізовано як додатковий резервний механізм для одногенераторних конфігурацій. Для інтеграції у відкриті системи (Marlin, Klipper) створено шар програмних обгорток на основі патерну «Адаптер».

Верифікацію розробленої моделі проведено у двох незалежних середовищах симуляції. Модульне тестування у Wokwi підтвердило коректність алгоритмічної частини: лічильники кроків обох моторів (11000 та 9221) точно збіглися з теоретичним розрахунком за кінематикою Аккермана. Апаратна верифікація у Proteus 8 Professional підтвердила коректність частотних характеристик: виміряні значення регістрів OCR (362, 432) та відношення періодів (1,193) збігаються з розрахунковими. Залишкова розбіжність періоду таймерів не перевищує 0,28%, дорівнює одному тіку таймера (0,5 мкс) і є систематичним артефактом середовища симуляції; на реальному ATmega2560, де період CTC апаратно дорівнює (OCR+1) тікам, вона відсутня. Підтверджено стабільність періоду імпульсів у межах роздільної здатності засобів вимірювання без накопичення часового дрейфу протягом усього циклу руху, що є принциповою перевагою апаратного CTC-режиму над програмним формуванням затримок, де період залежить від завантаженості процесора.

Таким чином, мету роботи досягнуто, а всі поставлені завдання виконано. Результати тестування підтверджують, що запропонована архітектура на базі апаратних таймерів ATmega2560 у CTC-режимі зі статичним поліморфізмом забезпечує точне, детерміноване та масштабоване керування кроковими двигунами відповідно до кінематики Аккермана. Розроблена модель є придатною для практичного впровадження у системи автоматизації, мобільні роботизовані платформи та верстати з ЧПК.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Пістун Є. П., Стасюк І. Д. Основи автоматики та автоматизації : навч. посіб. Львів : Вид-во Львів. політехніки, 2014. 336 с.
2. Обод І. І., Заволодько Г. Е., Свид І. В. Математичне моделювання систем : навч. посіб. / за ред. І. І. Обода. Харків : НТУ «ХПІ», 2019. 268 с.
3. Proctor F. M., Shackleford W. P. Real-time operating system timing jitter and its impact on motor control. Sensors and Controls for Intelligent Manufacturing II : Proc. SPIE. Boston, 2001. Vol. 4563. DOI: 10.1117/12.452653. (дата звернення: 24.05.2026).
4. De Luca A., Oriolo G., Samson C. Feedback control of a nonholonomic car-like robot. Robot Motion Planning and Control / ed. by J.-P. Laumond. Berlin : Springer, 1998. P. 171–253. (Lecture Notes in Control and Information Sciences ; vol. 229).
5. Reducing Timer Interrupt Load in 3D Printer Applications. Analog Devices. URL: https://www.analog.com/en/resources/technical-articles/reducing-timer-interrupt-load-in-3d-printer-applications.html (дата звернення: 24.05.2026).
6. Code Structure. Marlin Firmware. URL: https://marlinfw.org/docs/development/code_structure.html (дата звернення: 24.05.2026).
7. Code Overview. Klipper documentation. URL: https://www.klipper3d.org/Code_Overview.html (дата звернення: 24.05.2026).
8. Fixed Time Motion. Marlin Firmware. URL: https://marlinfw.org/docs/features/ft_motion.html (дата звернення: 24.05.2026).
9. Protocol. Klipper documentation. URL: https://www.klipper3d.org/Protocol.html (дата звернення: 24.05.2026).

10. Klipper vs. Marlin — 3D Printer Firmware Comparison. Xometry. URL: https://www.xometry.com/resources/3d-printing/klipper-vs-marlin/ (дата звернення: 24.05.2026).
11. ATmega640/1280/1281/2560/2561 : 8-bit AVR Microcontroller : Complete Datasheet. Microchip Technology. URL: https://ww1.microchip.com/downloads/en/devicedoc/atmel-2549-8-bit-avr-microcontroller-atmega640-1280-1281-2560-2561_datasheet.pdf (дата звернення: 24.05.2026).
12. Difference Between AVR and ARM. GeeksforGeeks. URL: https://www.geeksforgeeks.org/computer-organization-architecture/difference-between-avr-and-arm/ (дата звернення: 24.05.2026).
13. Arm Cortex-M4 Processor : Datasheet. Arm Ltd. URL: https://www.arm.com/-/media/Arm%20Developer%20Community/PDF/Processor%20Datasheets/Arm%20Cortex-M4%20Processor%20Datasheet.pdf (дата звернення: 24.05.2026).
14. Beginner Guide on Interrupt Latency and Arm Cortex-M Processors. Arm Community. URL: https://developer.arm.com/community/arm-community-blogs/b/architectures-and-processors-blog/posts/beginner-guide-on-interrupt-latency-and-interrupt-latency-of-the-arm-cortex-m-processors (дата звернення: 24.05.2026).
15. constexpr specifier. cppreference.com. URL: https://en.cppreference.com/w/cpp/language/constexpr (дата звернення: 24.05.2026).
16. Designing a HAL in C++. mbedded.ninja. URL: https://blog.mbedded.ninja/programming/languages/c-plus-plus/designing-a-hal-in-cpp/ (дата звернення: 24.05.2026).
17. Kinematic Car Model. Alpha Control Lab. URL: https://a-lab.ee/edu/example-system/944 (дата звернення: 24.05.2026).

18. Mastering Precision : Understanding Microstepping in Motion Control. Analog Devices. URL: https://www.analog.com/en/resources/analog-dialogue/articles/mastering-precision-understanding-microstepping.html (дата звернення: 24.05.2026).
19. Mathematics of Motion Control Profiles. Performance Motion Devices. URL: https://www.pmdcorp.com/resources/type/articles/get/mathematics-of-motion-control-profiles-article (дата звернення: 24.05.2026).
20. Intelligent S-Curve Acceleration and Deceleration Algorithm in High-Precision Servo Motion Control / F. Liu [et al.] Machines. 2026. Vol. 14, no. 1. Art. 91. DOI: 10.3390/machines14010091. (дата звернення: 24.05.2026).
21. A4988 : DMOS Microstepping Driver with Translator and Overcurrent Protection : Datasheet. Allegro MicroSystems. URL: https://www.allegromicro.com/-/media/files/datasheets/a4988-datasheet.pdf (дата звернення: 24.05.2026).
22. Bresenham's line algorithm. Wikipedia. URL: https://en.wikipedia.org/wiki/Bresenham%27s_line_algorithm (дата звернення: 24.05.2026).
23. Design of Multi-step Stepper Motor Coordinated Control System Based on Bresenham Algorithm / M. Dai [et al.] 2017 24th International Conference on Mechatronics and Machine Vision in Practice (M2VIP). Auckland, 2017. P. 1–5. URL: https://www.researchgate.net/publication/321986878 (дата звернення: 24.05.2026).
24. Bresenham J. E. Algorithm for Computer Control of a Digital Plotter. IBM Systems Journal. 1965. Vol. 4, no. 1. P. 25–30.
25. drf5n. Timer1 WGMmode=0b0100 / CTC(TOP=OCR1A=0) issue? : Issue #122. wokwi/avr8js, GitHub. URL: https://github.com/wokwi/avr8js/issues/122 (дата звернення: 24.05.2026).

ДОДАТКИ

Додаток А

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

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

З А В Д А Н Н Я
НА КВАЛІФІКАЦІЙНУ РОБОТУ

Здобувача <u>першого (бакалаврського)</u> рівня вищої освіти
(першого (бакалаврського) / другого (магістерського))

<u>ЧУХЛІБА ДЕНИСА ВАЛЕНТИНОВИЧА</u>
(прізвище, ім'я, по батькові студента)

Спеціальність: 151 – Автоматизація та комп'ютерно-інтегровані технології
Освітня програма «Автоматизація та комп'ютерно інтегровані технології»
1. Тема роботи **<u>«МЕТОД ОДНОЧАСНОГО КЕРУВАННЯ КРОКОВИМИ ДВИГУНАМИ З ВИКОРИСТАННЯМ АПАРАТНИХ ТАЙМЕРІВ МІКРОКОНТРОЛЕРА»</u>**
керівник роботи **<u>Котвицький Альберт Тадеушевич, к.ф.-м.н., доцент</u>**,
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)
затверджені наказом по університету ***№ 4101-5/1245 від 28.05.2026 року.***
2. Строк подання студентом роботи ***30 травня 2026 року***
3. Вихідні дані до роботи:
   1) документація на мікроконтролери ATmega2560 та STM32;
   2) вихідні коди прошивок Marlin і Klipper;
   3) наукові публікації з питань часового джитера та керування кроковими двигунами за принципом Аккермана;
4. Перелік питань, які потрібно розробити
   1) аналіз методів генерації імпульсів та проблеми часового джитера;
   2) порівняльний аналіз платформ AVR ATmega2560 та ARM Cortex-M;
   3) огляд алгоритмів керування рухом у системах Marlin і Klipper;
   4) розробка моделі та структурної схеми системи керування;
   5) програмна та апаратна реалізація методу на ATmega2560;
   6) розробка методики випробувань та верифікація моделі;
   7) оцінка переваг моделі та рекомендації щодо впровадження.

4. План роботи

№ з/п	Назви етапів роботи	Термін виконання етапів роботи
1.	Затвердження теми, аналіз предметної області та огляд методів генерації імпульсів і проблеми джитера	01.10.2025–15.11.2025
2.	Аналіз платформ (AVR vs ARM Cortex-M) та систем Marlin і Klipper	16.11.2025–31.12.2025
3.	Розробка моделі кінематики Аккермана, S-профілю та проєктування архітектури ПЗ	01.01.2026–31.01.2026
4.	Реалізація CRTP-шаблонів HAL, інтерполяції Брезенхема та обгорток API/Adapter	01.02.2026–20.03.2026
5.	Тестування у Wokwi та Proteus 8, аналіз синхронності	21.03.2026–15.04.2026
6.	Підготовка записки та передача роботи керівнику й рецензенту	16.04.2026–30.05.2026

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

Здобувач вищої освіти _____ Денис ЧУХЛІБ
(підпис) (ім'я, прізвище)

Керівник роботи _____ Альберт КОТВИЦЬКИЙ
(підпис) (ім'я, прізвище)

Додаток Б

Затверджую

_____

«_____»_____2026 р.

Технічне завдання
на розробку програмного виробу
«Метод одночасного керування кроковими двигунами з використанням апаратних таймерів мікроконтролера»

1.	Введення	1.1. Назва: Метод одночасного керування кроковими двигунами з використанням апаратних таймерів мікроконтролера. 1.2. Галузь застосування: Автоматизація та приладобудування.
2.	Підстава для розробки	2.1. Навчальний план за спеціальністю 151 «Автоматизація та комп'ютерно-інтегровані технології». 2.2. Завдання на кваліфікаційну роботу бакалавра № 4101-5/1245 від 28.05.2026р. (Додаток А до пояснювальної записки).
3	Призначення розробки	3.1. Мета розробки: мінімізація часового джитера під час генерації керуючих імпульсів у системах багатоосьового одночасного керування кроковими двигунами шляхом розробки програмної моделі на базі апаратних таймерів мікроконтролера. 3.2. Призначення розробки полягає у забезпеченні детермінованого синхронного руху кількох осей з підтримкою нелінійної кінематики (модель Аккермана) та лінійної інтерполяції Брезенхема, що дозволяє підвищити точність позиціювання мобільних роботизованих платформ і верстатів з ЧПК. 3.3. Вихідні дані розробки: технічна документація на мікроконтролери AVR (ATmega2560) та ARM Cortex-M (STM32), вихідні коди прошивок Marlin та Klipper, фізичні параметри тестової платформи (колісна база, ширина колії, параметри двигунів).
4.	Технічні вимоги до	4.1. Вимоги до функціональних характеристик:

	програмного виробу	1) одночасна генерація імпульсів для кількох крокових двигунів через апаратні таймери Timer3 і Timer4 у режимі CTC; 2) обчислення кінематики Аккермана з індивідуальним розрахунком швидкостей і кутів повороту для кожного колеса; 3) реалізація механізму синхронізації осей (пропорційні таймери Timer3/Timer4 + атомарний старт GTCCR) та модуля лінійної інтерполяції Брезенхема як резервного для одногенераторних конфігурацій. 4.2. Вимоги до надійності: забезпечення стабільності періоду керуючих імпульсів без накопичення часового дрейфу протягом усього циклу руху; роздільна здатність задання періоду — один такт мікроконтролера (62,5 нс при 16 МГц). 4.3. Вимоги до умов експлуатації: відсутні. 4.4. Вимоги до складу і параметрів технічних засобів: мікроконтролерна платформа 8-біт (AVR ATmega2560) або 32-біт (ARM Cortex-M), драйвери крокових двигунів (наприклад, A4988), крокові двигуни з кутом кроку 1,8°. 4.5. Вимоги до інформаційної та програмної сумісності: програмна модель має бути реалізована як незалежний модуль на базі CRTP-шаблонів. Для інтеграції з відкритими мікропрограмними системами (Marlin, Klipper) розробити програмні обгортки (API/Adapter), що забезпечують сумісність без втручання у базовий код. 4.6. Вимоги до маркування та упаковки: не висуваються. 4.7. Вимоги до транспортування і зберігання: на звичайних носіях інформації. 4.8. Спеціальні вимоги: відсутні.
5.	Вимоги до програмної документації	Програмною документацією до виробу «Метод одночасного керування кроковими двигунами з використанням апаратних таймерів мікроконтролера» вважати: 1) Дане Технічне завдання на розробку програмного виробу (Додаток Б до пояснювальної записки).

<table>
<tr><td></td><td></td><td>2) Програму і методику випробувань розробленого програмного виробу (Додаток В до пояснювальної записки).
3) Опис концептуальної моделі, програмної реалізації та результатів верифікації (представити у Розділах 2–4 пояснювальної записки до кваліфікаційної роботи).</td></tr>
<tr><td>6</td><td>Вимоги до техніко-економічних показників</td><td>6.1. Оцінка переваг запропонованої концептуальної моделі над існуючими рішеннями (Marlin, Klipper) за критеріями точності частоти, рівня часового джитера та обчислювального навантаження на мікроконтролер.

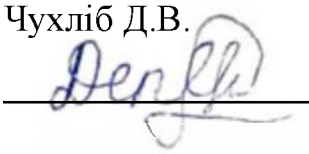
6.2. Рекомендації щодо практичного впровадження моделі в сучасні системи автоматизації з метою підвищення їх технічної та економічної ефективності, зокрема — зниження вартості апаратної платформи за рахунок раціонального використання ресурсів 8-бітних мікроконтролерів.</td></tr>
<tr><td>7</td><td>Стадії і етапи розробки</td><td>
<table>
<tr><th>Дата</th><th>Назва етапу</th></tr>
<tr><td>01.10.2025–
15.11.2025</td><td>Затвердження теми, аналіз предметної області та огляд методів генерації імпульсів і проблеми джитера</td></tr>
<tr><td>16.11.2025–
31.12.2025</td><td>Аналіз платформ (AVR vs ARM Cortex-M) та систем Marlin і Klipper</td></tr>
<tr><td>01.01.2026–
31.01.2026</td><td>Розробка моделі кінематики Аккермана, S-профілю та проєктування архітектури ПЗ</td></tr>
<tr><td>01.02.2026–
20.03.2026</td><td>Реалізація CRTP-шаблонів HAL, інтерполяції Брезенхема та обгорток API/Adapter</td></tr>
<tr><td>21.03.2026–
15.04.2026</td><td>Тестування у Wokwi та Proteus 8, аналіз синхронності</td></tr>
<tr><td>16.04.2026–
30.05.2026</td><td>Підготовка записки та передача роботи керівнику й рецензенту</td></tr>
</table>
</td></tr>
</table>

8	Порядок контролю і приймання програмного продукту	1. Контроль виконання згідно календарного плану. 2. Випробування програмної моделі провести відповідно до Програми та методики випробувань (Додаток В) у середовищі симуляції Proteus 8 Professional. 3. Захист на засіданні Екзаменаційної комісії. 4. Подання пояснювальної записки у паперовому та електронному вигляді.

Виконавець

студент групи КУ41

Чухліб Д.В.



Замовник

доцент

А. Т. Котвицький



Додаток В

Програма і методика випробувань програмного виробу

«Метод одночасного керування кроковими двигунами з використанням апаратних таймерів мікроконтролера»

1. Об'єкт випробувань

1. Назва програмного виробу: «Метод одночасного керування кроковими двигунами з використанням апаратних таймерів мікроконтролера».
2. Галузь застосування: автоматизація, мобільна робототехніка з кінематикою Аккермана, системи ЧПК.
3. Перераховані відомості запозичуються з відповідних розділів Технічного завдання (Додаток Б).

2. Мета випробувань

Перевірка відповідності розробленої програмної моделі вимогам технічного завдання щодо точності генерації керуючих імпульсів, синхронності багатоосьового руху за моделлю Аккермана та відсутності часового джитера, що забезпечується використанням апаратних 16-бітних таймерів мікроконтролера ATmega2560 у режимі CTC.

3. Загальні положення

1. **Підстави для проведення випробувань**

   Підставою для проведення випробувань є наказ про призначення екзаменаційної комісії.

2. **Місце і тривалість випробувань**

   Приймальні випробування проводяться на базі комп'ютерного класу кафедри у період роботи екзаменаційної комісії.

3. **Обсяг випробувань**

   Приймальні випробування програмного виробу проводяться в обсязі відповідному цієї програми і методики випробувань.

4. **Організації, які беруть участь у випробуваннях**

   Випробування проводяться екзаменаційною комісією за участю Замовника, Виконавця та інших осіб.

4. Вимоги до програми або програмного виробу

Модель повинна задовольняти наступним вимогам:

1. Забезпечення коректного обчислення кінематики Аккермана для заданого кута повороту коліс.
2. Розрахунок індивідуальних значень регістрів OCR для лівого і правого моторів відповідно до радіусів повороту.
3. Генерація заданої кількості кроків для кожного мотора через апаратні таймери Timer3 і Timer4 у режимі CTC.
4. Стабільність частоти імпульсів протягом усього циклу руху (відсутність часового джитера).
5. Синхронний старт обох таймерів через механізм GTCCR.
6. Сумісність з мікроконтролером ATmega2560 (тактова частота 16 МГц) та можливість перенесення на ARM Cortex-M.

Спеціальні вимоги (не пред'являються).

5. Вимоги до програмної документації

Програмною документацією до виробу вважати:

1. Технічне завдання на розробку (Додаток Б).
2. Дану Програму і методику випробувань (Додаток В).
3. Опис концептуальної моделі, програмної реалізації та результатів верифікації у Розділах 2–4 пояснювальної записки.

6. Засоби і порядок випробувань

6.1 Засоби випробувань

Для проведення випробувань необхідний проєкт програмної моделі, реалізований на мові C++ для мікроконтролера ATmega2560, середовище симуляції Wokwi (онлайн-симулятор Arduino Mega 2560 із вбудованим логічним аналізатором), а також симулятор Proteus 8 Professional з цикл-точною моделлю ATmega2560, повноцінною підтримкою 16-бітних таймерів Timer3 та Timer4 у режимі CTC та вбудованим інструментом Logic Analyser з частотою дискретизації 10 МГц і роздільною здатністю 0,1 мкс.

6.2 Порядок проведення випробувань

Як правило, випробування проводяться в два етапи:

- ознайомчий (1-й етап);
- випробування програмного виробу (2-й етап).

Перелік перевірок, що проводяться на 1 етапі випробувань, включає в себе:

1. Перевірку комплектності програмної документації.
2. Перевірка комплектності складу програмної документації здійснюється за критерієм наявності зазначеної в ТЗ документації.
3. Перевірку комплектності складу технічних і програмних засобів (наявність Arduino IDE з підтримкою AVR-GCC, симуляторів Wokwi та Proteus 8).
4. Методику проведення перевірок на 1 етапі випробувань.
5. Якість програмної документації перевіряється на відповідність вимогам стандартів ЄСПД.

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

1. Перевірку відповідності технічних характеристик програми вимогам технічного завдання;
2. Перевірку ступеня виконання функціональних вимог до програми;
3. Методику проведення перевірок, що входять до переліку по 2 етапу випробувань.

Умови та методика проведення випробувань:

1. Умови експлуатації: програма працює відповідно до умов експлуатації мікроконтролерів сімейства AVR (зокрема ATmega2560) з тактовою частотою 16 МГц.
2. Вимоги до програмного забезпечення: для роботи необхідний компілятор мови програмування C++ (AVR-GCC версії не нижчої ніж 7.3) та середовище Arduino IDE або PlatformIO.
3. Порядок проведення випробувань:
    3.1. Компіляція проєкту здійснюється командою компілятора AVR-GCC з отриманням HEX-файлу прошивки;
    3.2. Завантаження HEX-файлу у модель ATmega2560 у Proteus 8 Professional або у модель Arduino Mega 2560 у Wokwi;
    3.3. У схему симуляції підключається Logic Analyser до пінів 2 (STEP_LEFT, Timer3) та 4 (STEP_RIGHT, Timer4) на каналах D0 і D1 відповідно;
    3.4. Після запуску симуляції відбувається обчислення кінематики Аккермана для тестового сценарію (кут повороту 15°, базова лінійна швидкість 0,5 м/с, тривалість руху 2,0 с) і запис захоплених сигналів у форматі VCD для подальшого аналізу.

Для проведення випробувань пропонується тест 1, тест 2 та тест 3.

Тест 1

1. Перевірка виконання програми;
2. Модульне тестування алгоритму кінематики Аккермана у середовищі Wokwi для тестового сценарію (кут повороту 15°, v = 0,5 м/с, t = 2,0 с);

3. Отримання показників лічильників кроків на лівому та правому мотор-колесах (очікувані значення: 11000 та 9221 кроків відповідно).

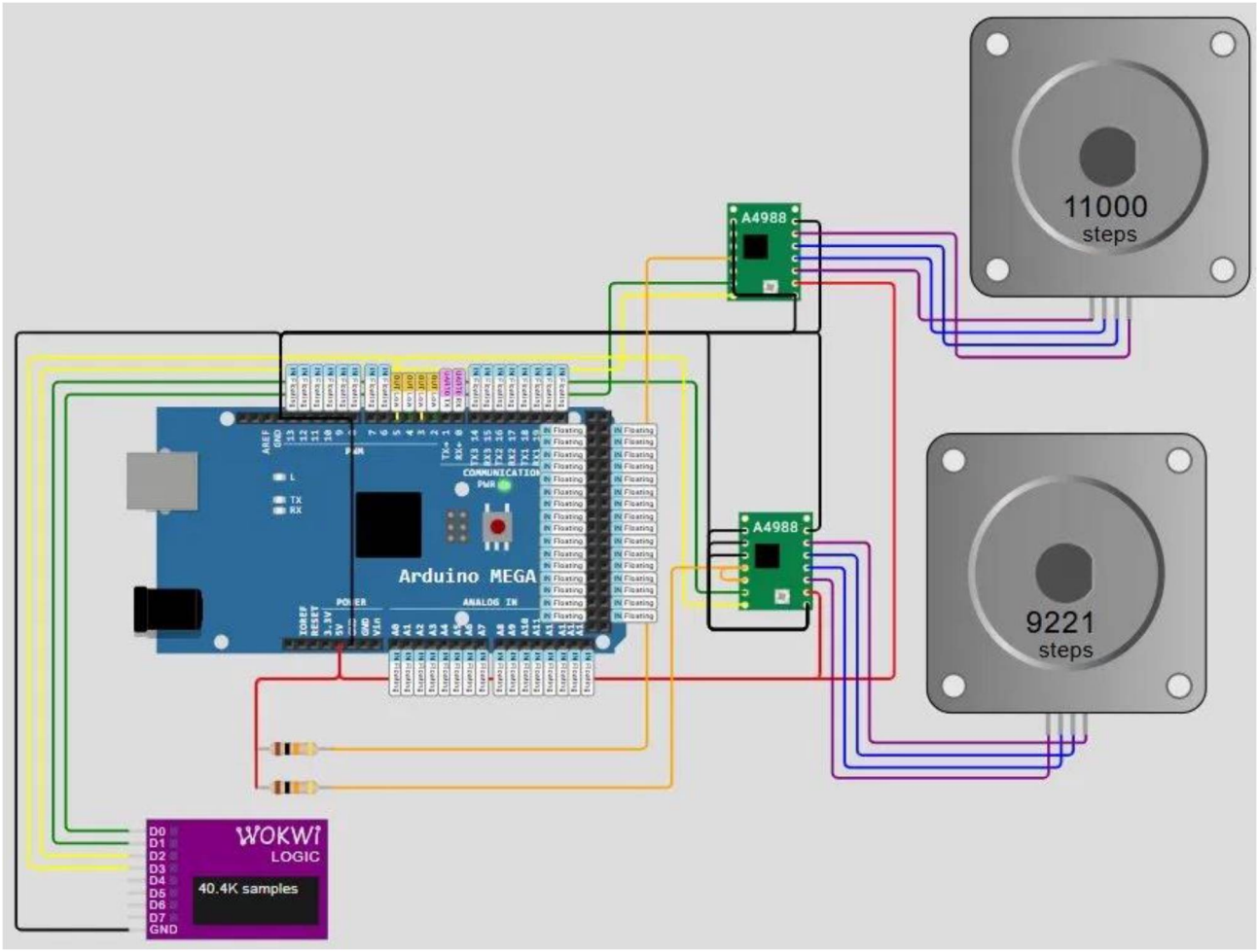


Рис. В.1 Тест 1

Тест 2

1. Перевірка виконання програми;
2. верифікація частотних характеристик у симуляторі Proteus 8 Professional шляхом вимірювання періоду імпульсів на пінах 2 і 4 за допомогою Logic Analyser;
3. отримання значень періодів сигналів та обчислених регістрів OCR Timer3 і Timer4 (очікувані значення: період D0 = 181 мкс, період D1 = 216 мкс, OCR_LEFT = 362, OCR_RIGHT = 432).



Рис. В.2 Тест 2

Тест 3

1. Перевірка виконання програми;
2. оцінка стабільності частоти імпульсів та аналіз синхронності старту таймерів протягом усього циклу руху;
3. отримання даних про рівномірність періоду на всьому інтервалі захоплення (очікуваний результат: відсутність варіацій періоду, що підтверджує відсутність накопичення часового дрейфу періоду в умовах симуляції завдяки використанню апаратного CTC-режиму).

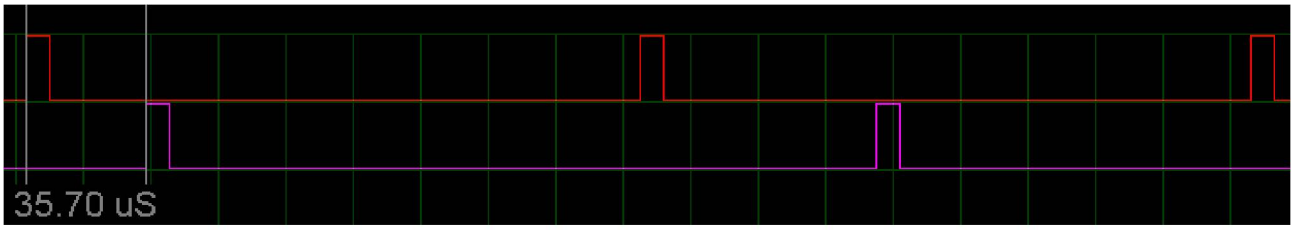


Рис. В.3 Тест 3

Тест вважається пройденим, якщо відбуваються вказані операції і їх відображення у програмному продукті.

Висновки: тест 1 успішно пройшов випробування, тест 2 успішно пройшов випробування і тест 3 успішно пройшов випробування. Випробування пройшло успішно.

Виконавець: студент групи КУ41, Чухліб Д.В. _____

Додаток Г

Лістинг Г.1 — Common/Config.h

```
#pragma once
#include <Arduino.h>

// Фізичні розміри робота
constexpr double WHEEL_BASE_M = 0.25;
constexpr double TRACK_WIDTH_M = 0.18;

// Апаратні константи (Arduino Mega 2560)
constexpr uint16_t TIMER_PRESCALER = 8;
constexpr uint32_t F_CPU_HZ = 16000000UL;

// Константи для пінів (можна буде перевизначити)
constexpr uint8_t STEP_PIN_LEFT = 2;
constexpr uint8_t DIR_PIN_LEFT = 3;

constexpr uint8_t STEP_PIN_RIGHT = 4;
constexpr uint8_t DIR_PIN_RIGHT = 5;
```

Лістинг Г.2 — Common/Types.h

```
#pragma once
#include <Arduino.h>

enum class Direction : uint8_t {
    Forward,
    Backward
};
```

Лістинг Г.3 — HAL/HardwareTimer.h

```
#pragma once
#include <Arduino.h>

/**
 * CRTP Базовий клас для апаратного таймера.
 * Гарантує статичний поліморфізм (без vtable) для максимальної продуктивності.
 */
template <class Derived>
```

```
class HardwareTimer {
public:
    inline void InitializeCtcMode() {
        static_cast<Derived*>(this)->InitializeCtcModeImpl();
    }
    inline void SetCompareMatchValue(uint16_t value) {
        static_cast<Derived*>(this)->SetCompareMatchValueImpl(value);
    }
    inline void EnableCompareInterrupt() {
        static_cast<Derived*>(this)->EnableCompareInterruptImpl();
    }
    inline void DisableCompareInterrupt() {
        static_cast<Derived*>(this)->DisableCompareInterruptImpl();
    }
    inline void StartTimer() {
        static_cast<Derived*>(this)->StartTimerImpl();
    }
    inline void StopTimer() {
        static_cast<Derived*>(this)->StopTimerImpl();
    }
    inline void ResetTimer() {
        static_cast<Derived*>(this)->ResetTimerImpl();
    }
    inline uint16_t GetCompareMatchValue() const {
        return static_cast<const Derived*>(this)->GetCompareMatchValueImpl();
    }
};
```

Лістинг Г.4 — HAL/StepperPin.h

```
#pragma once
#include "../Common/Types.h"

/**
 * CRTP Базовий клас для керування пінами крокового двигуна.
 * Використовує статичний поліморфізм для миттєвого перемикання станів пінів.
 */
template <class Derived>
class StepperPin {
public:
    inline void Initialize() {
```

```
        static_cast<Derived*>(this)->InitializeImpl();
    }
    inline void SetDirection(Direction dir) {
        static_cast<Derived*>(this)->SetDirectionImpl(dir);
    }
    inline void SetHigh() {
        static_cast<Derived*>(this)->SetHighImpl();
    }
    inline void SetLow() {
        static_cast<Derived*>(this)->SetLowImpl();
    }
    inline void ToggleStep() {
        static_cast<Derived*>(this)->ToggleStepImpl();
    }
};
```

Лістинг Г.5 — HAL/AvrHardwareTimer.h

```
#pragma once
#include "HardwareTimer.h"
#include <avr/io.h>
#include <avr/interrupt.h>

/**
 * Універсальний шаблон для AVR таймерів.
 * Замість передачі адрес регістрів напряму (що має обмеження constexpr у C++),
 * ми використовуємо структуру Traits, яка інкапсулює доступ до регістрів
 * та специфічну для кожного таймера атомарну логіку (напр. TEMP регістр).
 */
template <
    typename TRegSize,         // Розрядність регістра (uint8_t або uint16_t)
    typename Traits            // Структура властивостей (Traits Struct)
>
class AvrHardwareTimer : public HardwareTimer<AvrHardwareTimer<TRegSize, Traits>> {
private:
    uint8_t _prescalerBits; // Біти прескейлера (CSn2, CSn1, CSn0)

public:
    AvrHardwareTimer(uint8_t prescalerBits) : _prescalerBits(prescalerBits) {}

    inline void InitializeCtcModeImpl() {
```

```
        Traits::tccrA() = Traits::WGM_A;
        Traits::tccrB() = Traits::WGM_B; // Встановлюємо режим CTC (очищення по
співпадінню)
        Traits::resetTcnt();
    }

    inline void SetCompareMatchValueImpl(TRegSize value) {
        Traits::setOcrA(value);
    }

    inline void EnableCompareInterruptImpl() {
        Traits::timsk() |= (1 << Traits::OCIE_BIT);
    }

    inline void DisableCompareInterruptImpl() {
        Traits::timsk() &= ~(1 << Traits::OCIE_BIT);
    }

    inline void StartTimerImpl() {
        // Запускаємо таймер записуючи біти прескейлера
        Traits::tccrB() = (Traits::tccrB() & 0xF8) | (_prescalerBits & 0x07);
    }

    inline void StopTimerImpl() {
        // Зупинка таймера (очищення бітів Clock Select)
        Traits::tccrB() &= 0xF8;
    }

    inline void ResetTimerImpl() {
        Traits::resetTcnt();
    }

    inline TRegSize GetCompareMatchValueImpl() const {
        return Traits::getOcrA();
    }
};
```

Лістинг Г.6 — HAL/GpioStepperPin.h

```
#pragma once
#include "StepperPin.h"
#include <Arduino.h>

/**
 * Універсальний шаблон для керування пінами крокового двигуна за допомогою GPIO.
 */
template <uint8_t StepPin, uint8_t DirPin>
class GpioStepperPin : public StepperPin<GpioStepperPin<StepPin, DirPin>> {
public:
    inline void InitializeImpl() {
        pinMode(StepPin, OUTPUT);
        pinMode(DirPin, OUTPUT);
        digitalWrite(StepPin, LOW);
        digitalWrite(DirPin, LOW);
    }

    inline void SetDirectionImpl(Direction dir) {
        digitalWrite(DirPin, (dir == Direction::Forward) ? HIGH : LOW);
    }

    inline void SetHighImpl() {
        digitalWrite(StepPin, HIGH);
    }

    inline void SetLowImpl() {
        digitalWrite(StepPin, LOW);
    }

    inline void ToggleStepImpl() {
        SetHighImpl();
        // 16 тактів NOP = рівно 1 мкс гарантованої "чистої" затримки при 16 МГц.
        // Реальна ширина STEP-імпульсу ~7 мкс (п. 4.3) за рахунок накладних витрат
        // digitalWrite() та входу в ISR; це з запасом перекриває мінімум 1 мкс для
A4988.
        __builtin_avr_nop(); __builtin_avr_nop(); __builtin_avr_nop();
__builtin_avr_nop();
        __builtin_avr_nop(); __builtin_avr_nop(); __builtin_avr_nop();
__builtin_avr_nop();
```

```
        __builtin_avr_nop(); __builtin_avr_nop(); __builtin_avr_nop();
__builtin_avr_nop();
        __builtin_avr_nop(); __builtin_avr_nop(); __builtin_avr_nop();
__builtin_avr_nop();
        SetLowImpl();
    }
};
```

Лістинг Г.7 — Kinematics/WheelSpeeds.h

```
#pragma once

struct WheelSpeeds {
    double frontLeft;
    double frontRight;
    double rearLeft;
    double rearRight;

    WheelSpeeds(double fl = 0.0, double fr = 0.0, double rl = 0.0, double rr = 0.0)
        : frontLeft(fl), frontRight(fr), rearLeft(rl), rearRight(rr) {}
};

struct WheelAngles {
    double frontLeft;
    double frontRight;
    double rearLeft;
    double rearRight;

    WheelAngles(double fl = 0.0, double fr = 0.0, double rl = 0.0, double rr = 0.0)
        : frontLeft(fl), frontRight(fr), rearLeft(rl), rearRight(rr) {}
};

struct KinematicResult {
    WheelSpeeds speeds;
    WheelAngles angles;
};
```

Лістинг Г.8 — Kinematics/AckermannCalculator.h

```
#pragma once
#include "WheelSpeeds.h"
#include <Arduino.h>
#include <math.h>

/**
 * Клас для розрахунку повної кінематичної моделі Аккермана (4WS/4WD).
 * Відокремлений від апаратури (чиста математика в системі СІ).
 */
class AckermannCalculator {
private:
    double _a; // половина колії (trackWidth / 2)
    double _b; // половина бази (wheelBase / 2)
    double _c; // відступ від осі шворня до колеса
    double _s; // ширина шини

public:
    /**
     * @param trackWidth Колія платформи (відстань між лівими і правими шворнями), 2a
     * @param wheelBase Колісна база (відстань між передньою і задньою віссю), 2b
     * @param kingpinOffset Відступ від осі шворня до центра колеса, c
     * @param tireWidth Ширина шини, s
     */
    AckermannCalculator(double trackWidth, double wheelBase, double kingpinOffset =
0.0, double tireWidth = 0.0)
        : _a(trackWidth / 2.0), _b(wheelBase / 2.0), _c(kingpinOffset), _s(tireWidth)
{}

    /**
     * Розраховує кінематику повнокерованої/повнопривідної платформи.
     *
     * @param alpha2Deg Кут повороту переднього правого колеса у градусах (зовнішнє чи
внутрішнє залежить від знаку)
     * @param alpha4Deg Кут повороту заднього правого колеса у градусах (0 для
керування тільки передньою віссю)
     * @param v0 Цільова лінійна швидкість центру платформи (м/с)
     * @return KinematicResult (лінійні швидкості всіх коліс у м/с та їх кути)
     */
```

```
    inline KinematicResult Calculate(double alpha2Deg, double alpha4Deg, double v0)
const {
        KinematicResult result;

        double alpha2 = alpha2Deg * M_PI / 180.0;
        double alpha4 = alpha4Deg * M_PI / 180.0;

        // Рух прямо
        if (fabs(alpha2) < 0.001 && fabs(alpha4) < 0.001) {
            result.speeds = WheelSpeeds(v0, v0, v0, v0);
            result.angles = WheelAngles(0.0, 0.0, 0.0, 0.0);
            return result;
        }

        double Cx, Cy;
        if (fabs(alpha4) < 0.001) {
            // задні колеса прямо (класичний Аккерман передньої осі)
            Cx = _a + 2.0 * _b * (1.0 / tan(alpha2));
            Cy = -_b;
        } else if (fabs(alpha2) < 0.001) {
            // передні колеса прямо (керування тільки задньою віссю)
            Cx = _a + 2.0 * _b * (1.0 / tan(alpha4));
            Cy = _b;
        } else {
            // повнокерований режим
            double cot_a2 = 1.0 / tan(alpha2);
            double cot_a4 = 1.0 / tan(alpha4);
            double K = cot_a2 / cot_a4;
            Cx = _a + (2.0 * _b * cot_a2) / (K + 1.0);
            Cy = _b * (K - 1.0) / (K + 1.0);
        }

        double R0 = sqrt(Cx * Cx + Cy * Cy);

        // Векторний розрахунок геометричних радіусів шворнів
        double RA1 = sqrt((Cx + _a) * (Cx + _a) + (Cy - _b) * (Cy - _b));
        double RA2 = sqrt((Cx - _a) * (Cx - _a) + (Cy - _b) * (Cy - _b));
        double RA3 = sqrt((Cx + _a) * (Cx + _a) + (Cy + _b) * (Cy + _b));
        double RA4 = sqrt((Cx - _a) * (Cx - _a) + (Cy + _b) * (Cy + _b));
```

```
        // Розрахунок фактичних радіусів контакту (з урахуванням зміщень c та s)
        double offset = _c + _s / 2.0;
        double R1, R2, R3, R4;

        if (Cx > 0) {
            // Центр повороту справа -> поворот праворуч -> ліві колеса зовнішні
            R1 = RA1 + offset;
            R3 = RA3 + offset;
            R2 = RA2 - offset;
            R4 = RA4 - offset;
        } else {
            // Центр повороту зліва -> поворот ліворуч -> праві колеса зовнішні
            R1 = RA1 - offset;
            R3 = RA3 - offset;
            R2 = RA2 + offset;
            R4 = RA4 + offset;
        }

        // Диференціація лінійних швидкостей
        result.speeds.frontLeft = v0 * R1 / R0;
        result.speeds.frontRight = v0 * R2 / R0;
        result.speeds.rearLeft = v0 * R3 / R0;
        result.speeds.rearRight = v0 * R4 / R0;

        // Розрахунок залежних кутів лівого борту (alpha1 та alpha3)
        // atan2 повертає радіани
        double alpha1 = atan2(_b - Cy, Cx + _a);
        double alpha3 = atan2(-_b - Cy, Cx + _a);

        result.angles.frontLeft = alpha1 * 180.0 / M_PI;
        result.angles.frontRight = alpha2Deg;
        result.angles.rearLeft = alpha3 * 180.0 / M_PI;
        result.angles.rearRight = alpha4Deg;

        return result;
    }
};
```

Лістинг Г.9 — Motion/LinearInterpolator.h

```
#pragma once
#include <Arduino.h>

struct InterpolationResult {
    uint32_t stepsA;        // Домінантна вісь (максимум кроків)
    uint32_t stepsB;        // Підпорядкована вісь
    bool isLeftDominant;
};

/**
* Лінійна інтерполяція за алгоритмом Брезенхема.
* ДОДАТКОВИЙ/РЕЗЕРВНИЙ модуль для одногенераторних конфігурацій
* (кілька осей з одного спільного таймера). У поточній двотаймерній
* конфігурації не задіяний, бо синхронність забезпечується пропорційними
* OCR + атомарним стартом GTCCR.
*
* Використання:
*   auto r = interp.CalculateBresenham(stepsX, stepsY);
*   interp.Reset(r);
*   // на кожен крок домінантної осі:
*   bool doSubStep = interp.NextStep(r);  // true -> зробити крок підпорядкованої осі
*/
class LinearInterpolator {
private:
    int32_t _error;        // Акумулятор похибки Брезенхема
    uint32_t _doneA;       // Лічильник виконаних кроків домінантної осі

public:
    LinearInterpolator() : _error(0), _doneA(0) {}

    inline InterpolationResult CalculateBresenham(uint32_t targetX, uint32_t targetY)
const {
        InterpolationResult res;
        if (targetX >= targetY) {
            res.stepsA = targetX;
            res.stepsB = targetY;
            res.isLeftDominant = true;
        } else {
            res.stepsA = targetY;
```

```
            res.stepsB = targetX;
            res.isLeftDominant = false;
        }
        return res;
    }

    /**
    * Ініціалізація акумулятора перед серією кроків.
    * Класичний старт: error = 2*dB - dA.
    */
    inline void Reset(const InterpolationResult& res) {
        _error = 2 * (int32_t)res.stepsB - (int32_t)res.stepsA;
        _doneA = 0;
    }

    /**
    * Викликається на КОЖЕН крок домінантної осі.
    * Повертає true, якщо на цьому кроці потрібно зробити крок підпорядкованої осі.
    */
    inline bool NextStep(const InterpolationResult& res) {
        bool subStep = false;
        if (_error > 0) {
            subStep = true;
            _error -= 2 * (int32_t)res.stepsA;
        }
        _error += 2 * (int32_t)res.stepsB;
        _doneA++;
        return subStep;
    }

    inline bool IsFinished(const InterpolationResult& res) const {
        return _doneA >= res.stepsA;
    }
};
```

Лістинг Г.10 — Motion/SynchronousMotionPlanner.h

```
#pragma once
#include <Arduino.h>

/**
```

```
 * Планувальник рухів.
 * Розраховує значення OCR (Timer Compare Match) для забезпечення заданої швидкості.
 *
 * Чому апаратний таймер усуває джитер:
 * Програмні затримки (delay) залежать від переривань та завантаження CPU.
 * Апаратний таймер (CTC) працює незалежно і генерує переривання точно в момент
 * співпадіння OCR, гарантуючи стабільний період кроків без накопичення часового
дрейфу.
 */
class SynchronousMotionPlanner {
private:
    uint32_t _fCpu;
    uint16_t _prescaler;

public:
    SynchronousMotionPlanner(uint32_t fCpu, uint16_t prescaler)
        : _fCpu(fCpu), _prescaler(prescaler) {}

    /**
     * Розраховує значення OCR для швидкості (кроків/сек).
     * Одне переривання (ISR) генерує один повний крок (HIGH + NOP + LOW).
     * Формула: OCR = (F_CPU / (PRESCALER * Speed)) - 1
     */
    inline uint16_t CalculateOcrForSpeed(double speedStepsPerSec) const {
        if (speedStepsPerSec < 0.001) return 65535; // Стоп

        double ticks = (double)_fCpu / ((double)_prescaler * speedStepsPerSec);

        uint32_t ocr = (uint32_t)(ticks - 1.0);

        if (ocr > 65535) return 65535; // Максимум для 16-bit
        return (uint16_t)ocr;
    }

     // Розраховує загальну кількість кроків.
    inline uint32_t CalculateTotalSteps(double speedStepsPerSec, double durationSec)
const {
        return (uint32_t)(speedStepsPerSec * durationSec);
    }
};
```

Лістинг Г.11 — Motion/StepperMotorMapper.h

```
#pragma once
#include "../Kinematics/WheelSpeeds.h"
#include <Arduino.h>

/**
 * Структура з апаратними параметрами мотор-колеса
 */
struct MotorConfig {
    double wheelRadius;    // Радіус колеса в метрах
    double stepAngle;      // Кут одного кроку (наприклад, 1.8 градуса)
    uint16_t microsteps;   // Режим мікрокроку (1, 2, 4, 8, 16, 32...)
    double gearRatio;      // Коефіцієнт редукції (якщо прямий привід, то 1.0)

    MotorConfig(double r = 0.05, double angle = 1.8, uint16_t micro = 16, double gear =
1.0)
        : wheelRadius(r), stepAngle(angle), microsteps(micro), gearRatio(gear) {}
};

/**
 * Клас для перетворення лінійних швидкостей (м/с) у частоту кроків (Hz)
 * Дотримується SRP: не знає про кінематику Аккермана, тільки про колеса.
 */
class StepperMotorMapper {
private:
    MotorConfig _config;
    double _stepsPerMeter;

public:
    StepperMotorMapper(const MotorConfig& config) : _config(config) {
        // Розрахунок кількості кроків на 1 метр лінійного переміщення
        // f_steps = (v * 360 * microsteps * gearRatio) / (2 * PI * r * stepAngle)
        // steps_per_meter = (360 * microsteps * gearRatio) / (2 * PI * r * stepAngle)
        _stepsPerMeter = (360.0 * _config.microsteps * _config.gearRatio) /
                         (2.0 * M_PI * _config.wheelRadius * _config.stepAngle);
    }

    /**
     * Конвертує швидкість у м/с в кроки на секунду (Hz)
```

```
     */
    inline double VelocityToStepsPerSec(double velocityMetersPerSec) const {
        return velocityMetersPerSec * _stepsPerMeter;
    }

    /**
     * Конвертує всі швидкості з KinematicResult (м/с) у кроки на секунду.
     * Повертає структуру WheelSpeeds, де значення - це вже кроки/сек.
     */
    inline WheelSpeeds MapAll(const WheelSpeeds& velocitiesMps) const {
        WheelSpeeds stepsPerSec;
        stepsPerSec.frontLeft = VelocityToStepsPerSec(velocitiesMps.frontLeft);
        stepsPerSec.frontRight = VelocityToStepsPerSec(velocitiesMps.frontRight);
        stepsPerSec.rearLeft = VelocityToStepsPerSec(velocitiesMps.rearLeft);
        stepsPerSec.rearRight = VelocityToStepsPerSec(velocitiesMps.rearRight);
        return stepsPerSec;
    }
};
```

Лістинг Г.12 — Motion/MotionController.h

```
#pragma once
#include "../Kinematics/AckermannCalculator.h"
#include "SynchronousMotionPlanner.h"
#include <avr/io.h>

/**
 * Структура, що містить розраховані параметри для апаратного таймера
 */
struct TurnCommand {
    uint16_t ocrLeft;
    uint16_t ocrRight;
    uint32_t stepsLeft;
    uint32_t stepsRight;
};

/**
 * Головний контролер руху.
 * Використовує абсолютно статичний поліморфізм (без vtable).
 */
```

```
template <typename TLeftTimer, typename TRightTimer, typename TLeftPin, typename
TRightPin, typename TKinematics, typename TPlanner, typename TMapper>
class MotionController {
private:
    TLeftTimer* _leftTimer;
    TRightTimer* _rightTimer;
    TLeftPin* _leftPin;
    TRightPin* _rightPin;
    TKinematics* _kinematics;
    TPlanner* _planner;
    TMapper* _mapper;

public:
    MotionController(TLeftTimer* leftTimer, TRightTimer* rightTimer,
                     TLeftPin* leftPin, TRightPin* rightPin,
                     TKinematics* kinematics, TPlanner* planner, TMapper* mapper)
        : _leftTimer(leftTimer), _rightTimer(rightTimer),
          _leftPin(leftPin), _rightPin(rightPin),
          _kinematics(kinematics), _planner(planner), _mapper(mapper)
    {
        _leftPin->Initialize();
        _rightPin->Initialize();
        _leftTimer->InitializeCtcMode();
        _rightTimer->InitializeCtcMode();
    }

    /**
     * Розраховує параметри руху без запуску таймерів.
     */
    inline TurnCommand PrepareTurn(double steeringAngleDeg, double baseSpeed, double
durationSec) {
        auto result = _kinematics->Calculate(steeringAngleDeg, 0.0, baseSpeed);
        auto speedsInSteps = _mapper->MapAll(result.speeds);

        TurnCommand cmd;
        cmd.ocrLeft = _planner->CalculateOcrForSpeed(speedsInSteps.rearLeft);
        cmd.ocrRight = _planner->CalculateOcrForSpeed(speedsInSteps.rearRight);
        cmd.stepsLeft = _planner->CalculateTotalSteps(speedsInSteps.rearLeft,
durationSec);
```

```
        cmd.stepsRight = _planner->CalculateTotalSteps(speedsInSteps.rearRight,
durationSec);

        return cmd;
    }

    /**
     * Запускає підготовлену команду з ідеально одночасним стартом через GTCCR.
     */
    inline void StartTurn(const TurnCommand& cmd) {
        _leftTimer->SetCompareMatchValue(cmd.ocrLeft);
        _rightTimer->SetCompareMatchValue(cmd.ocrRight);

        // 3. Гарантія одночасного старту (СИНХРОНІЗАЦІЯ через GTCCR)
        GTCCR = (1 << TSM) | (1 << PSRSYNC);

        _leftTimer->ResetTimer();
        _rightTimer->ResetTimer();
        _leftTimer->StartTimer();
        _rightTimer->StartTimer();
        _leftTimer->EnableCompareInterrupt();
        _rightTimer->EnableCompareInterrupt();

        // Зняття TSM - таймери стартують атомарно
        GTCCR = 0;
    }

    inline void Stop() {
        _leftTimer->StopTimer();
        _rightTimer->StopTimer();
    }
};
```

Лістинг Г.13 — SynchronousControlOfStepperMotors.ino

```
#include "src/HAL/AvrHardwareTimer.h"
#include "src/HAL/GpioStepperPin.h"
#include "src/Kinematics/AckermannCalculator.h"
#include "src/Motion/SynchronousMotionPlanner.h"
#include "src/Motion/MotionController.h"
#include "src/Motion/StepperMotorMapper.h"
```

```
#include "src/Common/Config.h"

#include <util/atomic.h>

// 1. Апаратура: Трейтси (Traits) для роботи з регістрами таймерів
//
// Архітектурний патерн "Traits Struct" дозволяє інкапсулювати апаратно-залежні
// деталі доступу до регістрів мікроконтролера.
// Замість передачі магічних чисел або вказівників (що створює проблеми з constexpr),
// ми створюємо структуру-обгортку, яку компілятор C++ ідеально оптимізує
// (всі методи inline, немає накладних витрат пам'яті чи швидкодії).
//
// Крім того, це дозволяє реалізувати безпечний доступ до 16-бітних регістрів:
// - Явний запис через макроси 16-бітних регістрів (OCR3A, TCNT3), які AVR-GCC
//   автоматично розбиває на правильну послідовність запису HIGH та LOW байтів.
// - ВАЖЛИВО: хоча компілятор знає порядок, він НЕ генерує блокування переривань!
//   Якщо переривання вдереться між записом/читанням 8-бітних частин, спільний
//   апаратний буфер TEMP буде зіпсовано. Тому ATOMIC_BLOCK(ATOMIC_RESTORESTATE)
//   залишається ЖИТТЄВО НЕОБХІДНИМ для 16-бітних операцій на AVR.
struct Timer3Traits {
    // Константи конфігурації
    static constexpr uint8_t OCIE_BIT = OCIE3A;
    static constexpr uint8_t WGM_A = 0;
    static constexpr uint8_t WGM_B = (1 << WGM32);

    // 8-бітні регістри конфігурації
    static volatile uint8_t& tccrA() { return TCCR3A; }
    static volatile uint8_t& tccrB() { return TCCR3B; }
    static volatile uint8_t& timsk() { return TIMSK3; }

    // 16-бітний запис OCR (Output Compare Register)
    static void setOcrA(uint16_t value) {
        ATOMIC_BLOCK(ATOMIC_RESTORESTATE) {
            // AVR-GCC генерує HIGH->LOW, а ATOMIC_BLOCK гарантує, що
            // ніяке переривання не зіпсує буфер TEMP між цими інструкціями.
            OCR3A = value;
        }
    }

    // 16-бітне читання OCR
```

```
    static uint16_t getOcrA() {
        uint16_t val;
        ATOMIC_BLOCK(ATOMIC_RESTORESTATE) {
            // AVR-GCC автоматично компілює це у правильний порядок (LOW, потім HIGH)
            val = OCR3A;
        }
        return val;
    }

    // 16-бітне скидання лічильника (Timer/Counter)
    static void resetTcnt() {
        ATOMIC_BLOCK(ATOMIC_RESTORESTATE) {
            TCNT3 = 0;
        }
    }
};

struct Timer4Traits {
    // Константи конфігурації
    static constexpr uint8_t OCIE_BIT = OCIE4A;
    static constexpr uint8_t WGM_A = 0;
    static constexpr uint8_t WGM_B = (1 << WGM42);

    static volatile uint8_t& tccrA() { return TCCR4A; }
    static volatile uint8_t& tccrB() { return TCCR4B; }
    static volatile uint8_t& timsk() { return TIMSK4; }

    static void setOcrA(uint16_t value) {
        ATOMIC_BLOCK(ATOMIC_RESTORESTATE) {
            OCR4A = value;
        }
    }
    static uint16_t getOcrA() {
        uint16_t val;
        ATOMIC_BLOCK(ATOMIC_RESTORESTATE) {
            val = OCR4A;
        }
        return val;
    }
    static void resetTcnt() {
```

```
        ATOMIC_BLOCK(ATOMIC_RESTORESTATE) {
            TCNT4 = 0;
        }
    }
};

// Створюємо типи таймерів (16-біт Timer3 та Timer4)
using Timer3 = AvrHardwareTimer<uint16_t, Timer3Traits>;
using Timer4 = AvrHardwareTimer<uint16_t, Timer4Traits>;


// Піни
using LeftPin = GpioStepperPin<STEP_PIN_LEFT, DIR_PIN_LEFT>;
using RightPin = GpioStepperPin<STEP_PIN_RIGHT, DIR_PIN_RIGHT>;

// Глобальні екземпляри (для доступу з ISR)
Timer3 leftMotorTimer((1 << CS31)); // Prescaler 8
Timer4 rightMotorTimer((1 << CS41));

LeftPin leftPin;
RightPin rightPin;

AckermannCalculator robotKinematics(TRACK_WIDTH_M, WHEEL_BASE_M);
SynchronousMotionPlanner motionPlanner(F_CPU_HZ, TIMER_PRESCALER);

MotorConfig motorConfig(0.05, 1.8, 16, 1.0); // Радіус 5см, крок 1.8 градуса, 16
мікрокроків
StepperMotorMapper motorMapper(motorConfig);

// 3. Контролер (Абсолютний статичний поліморфізм)
MotionController<Timer3, Timer4, LeftPin, RightPin, AckermannCalculator,
SynchronousMotionPlanner, StepperMotorMapper> robot(
    &leftMotorTimer, &rightMotorTimer,
    &leftPin, &rightPin,
    &robotKinematics, &motionPlanner, &motorMapper
);


//Обробники переривань ISR
// Стан для ISR
```

```
volatile uint32_t timer3StepsRemaining = 0;
volatile uint32_t timer4StepsRemaining = 0;

ISR(TIMER3_COMPA_vect) {
    if (timer3StepsRemaining > 0) {
        leftPin.ToggleStep(); // Містить затримку для A4988 (див. Г.6: ~7 мкс реальної
ширини)
        timer3StepsRemaining--;
    } else {
        leftMotorTimer.StopTimer();
    }
}

ISR(TIMER4_COMPA_vect) {
    if (timer4StepsRemaining > 0) {
        rightPin.ToggleStep();
        timer4StepsRemaining--;
    } else {
        rightMotorTimer.StopTimer();
    }
}

//Точка входу
void setup() {
    Serial.begin(115200);

    leftMotorTimer.InitializeCtcMode();
    rightMotorTimer.InitializeCtcMode();

    // Демонстраційний запуск: Рух із кутом 15 градусів і базовою швидкістю 0.5 м/с
    // Спочатку підготовлюємо розрахунки, встановлюємо глобальні змінні для ISR, а
потім стартуємо

    leftPin.SetDirection(Direction::Forward);
    rightPin.SetDirection(Direction::Forward);

    auto cmd = robot.PrepareTurn(15.0, 0.5, 2.0);
    timer3StepsRemaining = cmd.stepsLeft;
    timer4StepsRemaining = cmd.stepsRight;
    robot.StartTurn(cmd);
```

```
}

void loop() {
    // Вся реальна робота відбувається апаратно у фоні без накопичення дрейфу періоду
    delay(5000);
}
```

Лістинг Г.14 — Motion/SCurveProfile.h

```
#pragma once
#include <Arduino.h>
#include <math.h>

/**
* Генератор S-подібного (jerk-limited) профілю швидкості.
* ДОДАТКОВИЙ/РЕЗЕРВНИЙ модуль (як LinearInterpolator): у поточній
* двотаймерній конфігурації зі сталою частотою не задіяний у критичному
* тракті, реалізований для демонстрації та подальшої інтеграції в модуль Motion.
*
* Класичний 7-фазний профіль із обмеженням ривка (jerk):
*  1) позитивний ривок      4) крейсер (a=0)      7) позитивний ривок
*  2) стале прискорення     5) негативний ривок
*  3) негативний ривок      6) стале гальмування
*
* Усі величини — у крокових одиницях:
*  vMax  — кроків/с, aMax — кроків/с^2, jMax — кроків/с^3.
*/
class SCurveProfile {
private:
    double _vMax, _aMax, _jMax;

    // Параметри спланованого профілю
    double _tj;      // тривалість фази ривка
    double _ta;      // тривалість фази сталого прискорення
    double _tv;      // тривалість крейсерської фази
    double _aPeak;   // фактично досягнуте прискорення (<= aMax)
    double _vReach;  // фактично досягнута швидкість (<= vMax)
    bool   _planned;

    // Накопичувальні межі фаз у часі
    double _t1, _t2, _t3, _t4, _t5, _t6, _t7;
```

```
    // Розрахунок параметрів розгону для заданої цільової швидкості v.
    // Повертає шлях (кроки), пройдений за весь етап розгону (фази 1-3).
    inline double accelSteps(double v, double& Taccel,
                             double& Tj, double& Ta, double& aPk) const {
        if (v >= (_aMax * _aMax) / _jMax) {
            // Досягаємо максимального прискорення (є фаза сталого a)
            aPk = _aMax;
            Tj  = _aMax / _jMax;
            Ta  = v / _aMax - Tj;
        } else {
            // Трикутний профіль прискорення (aMax не досягається)
            aPk = sqrt(v * _jMax);
            Tj  = aPk / _jMax;
            Ta  = 0.0;
        }
        Taccel = 2.0 * Tj + Ta;
        // Розгін антисиметричний -> середня швидкість = v/2
        return 0.5 * v * Taccel;
    }

public:
    SCurveProfile(double vMax, double aMax, double jMax)
        : _vMax(vMax), _aMax(aMax), _jMax(jMax),
          _tj(0), _ta(0), _tv(0), _aPeak(0), _vReach(0), _planned(false),
          _t1(0), _t2(0), _t3(0), _t4(0), _t5(0), _t6(0), _t7(0) {}

    /**
     * Планує профіль на задану кількість кроків.
     * Якщо шляху недостатньо для досягнення vMax — цільова швидкість
     * автоматично знижується (бісекція), крейсерська фаза відсутня.
     */
    inline void Plan(double totalSteps) {
        double Taccel, Tj, Ta, aPk;
        double sAcc = accelSteps(_vMax, Taccel, Tj, Ta, aPk);

        if (2.0 * sAcc <= totalSteps) {
            // Повний 7-фазний профіль із крейсером
            _vReach = _vMax; _aPeak = aPk; _tj = Tj; _ta = Ta;
            _tv = (totalSteps - 2.0 * sAcc) / _vReach;
        } else {
```

```
        // Короткий рух: знижуємо vReach так, щоб 2*sAcc == totalSteps
        double lo = 0.0, hi = _vMax;
        for (uint8_t i = 0; i < 40; ++i) {
            double mid = 0.5 * (lo + hi);
            double s = accelSteps(mid, Taccel, Tj, Ta, aPk);
            if (2.0 * s > totalSteps) hi = mid; else lo = mid;
        }
        _vReach = 0.5 * (lo + hi);
        accelSteps(_vReach, Taccel, Tj, Ta, aPk);
        _aPeak = aPk; _tj = Tj; _ta = Ta; _tv = 0.0;
    }

    _t1 = _tj;
    _t2 = _t1 + _ta;
    _t3 = _t2 + _tj;   // кінець розгону (v = vReach)
    _t4 = _t3 + _tv;   // кінець крейсера
    _t5 = _t4 + _tj;
    _t6 = _t5 + _ta;
    _t7 = _t6 + _tj;   // повна зупинка (v = 0)
    _planned = true;
}

// Миттєва швидкість (кроків/с) у момент часу t (секунди).
inline double VelocityAt(double t) const {
    if (!_planned || t <= 0.0) return 0.0;
    if (t >= _t7) return 0.0;

    const double J = _jMax, aPk = _aPeak, vR = _vReach;
    const double vAtTj = 0.5 * aPk * _tj;            // v у кінці фази 1
    const double vEndAccel2 = vAtTj + aPk * _ta;     // v у кінці фази 2

    if (t < _t1) {                       // 1: позитивний ривок
        return 0.5 * J * t * t;
    } else if (t < _t2) {                // 2: стале прискорення
        return vAtTj + aPk * (t - _t1);
    } else if (t < _t3) {                // 3: негативний ривок
        double tau = t - _t2;
        return vEndAccel2 + aPk * tau - 0.5 * J * tau * tau;
    } else if (t < _t4) {                // 4: крейсер
        return vR;
```

```
        } else if (t < _t5) {                // 5: негативний ривок
            double tau = t - _t4;
            return vR - 0.5 * J * tau * tau;
        } else if (t < _t6) {                // 6: стале гальмування
            double tau = t - _t5;
            return (vR - 0.5 * aPk * _tj) - aPk * tau;
        } else {                             // 7: позитивний ривок
            double tau = t - _t6;
            double vAtT6 = vR - 0.5 * aPk * _tj - aPk * _ta;
            return vAtT6 - aPk * tau + 0.5 * J * tau * tau;
        }
    }

    // Поточна фаза (1..7) для діагностики; 0 — поза рухом.
    inline uint8_t CurrentPhase(double t) const {
        if (!_planned || t <= 0.0 || t >= _t7) return 0;
        if (t < _t1) return 1;
        if (t < _t2) return 2;
        if (t < _t3) return 3;
        if (t < _t4) return 4;
        if (t < _t5) return 5;
        if (t < _t6) return 6;
        return 7;
    }

    inline double TotalDuration() const { return _planned ? _t7 : 0.0; }
    inline double ReachedVelocity() const { return _vReach; }

    /**
     * Демонстраційний міст до апаратного таймера: значення OCR для миттєвої
     * швидкості у момент t (та сама формула, що й у SynchronousMotionPlanner).
     */
    inline uint16_t OcrAt(double t, uint32_t fCpu, uint16_t prescaler) const {
        double v = VelocityAt(t);
        if (v < 0.001) return 65535;
        double ticks = (double)fCpu / ((double)prescaler * v);
        uint32_t ocr = (uint32_t)(ticks - 1.0);
        return (ocr > 65535) ? 65535 : (uint16_t)ocr;
    }
};
```