НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ БІОРЕСУРСІВ І ПРИРОДОКОРИСТУВАННЯ УКРАЇНИ

ФАКУЛЬТЕТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

ПОГОДЖЕНО

ДОПУСКАЄТЬСЯ ДО ЗАХИСТУ

Завідувач кафедри

Комп'ютерних систем, мереж та кібербезпеки

_____ Касаткін Д.Ю., к.пед.н., доц.

(підпис) (ПІБ, вчене звання і ступінь)

«___»_____2025 р.

КВАЛІФІКАЦІЙНА БАКАЛАВРСЬКА РОБОТА

На тему: «Розроблення комп'ютерної системи опрацювання і візуалізації даних з периферійних пристроїв»

Спеціальність F7 «Комп'ютерна інженерія»

Гарант освітньої програми:_____Нікітенко Є.В.__

Керівник дипломного проекту:_____ / Шкарупило В.В. /

підпис ПІБ

Виконав:_____ / Левченко М.С. /

підпис ПІБ

КИЇВ-2025

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ БІОРЕСУРСІВ І ПРИРОДОКОРИСТУВАННЯ УКРАЇНИ
ФАКУЛЬТЕТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

«ЗАТВЕРДЖУЮ»
завідувач кафедри
комп'ютерних систем, мереж та кібербезпеки
_____/ Касаткін Д.Ю., к.п.н., доц. /
підпис ПІБ, вчене звання і ступінь
«__» _____ 2025_ р.

З А В Д А Н Н Я

ДО ВИКОНАННЯ КВАЛІФІКАЦІЙНОЇ БАКАЛАВРСЬКОЇ СТУДЕНТУ

Максиму Сергійовичу Левченко
(прізвище, ім'я, по батькові)
Спеціальність (напрям підготовки): комп'ютерна інженерія_____

Тема кваліфікаційної бакалаврської роботи: «Розроблення комп'ютерної системи опрацювання і візуалізації даних з периферійних пристроїв»_____
_____
_____
затверджена наказом ректора НУБіП України від "___" _____2025_р. №____
Термін подання завершеної роботи на кафедру _____
Вихідні дані до кваліфікаційної бакалаврської роботи _____
_____
_____
_____
_____
_____
Перелік питань, що підлягають розробці:
1. _____
2. _____
3. _____
Перелік графічного матеріалу (за потреби) _____
_____

Дата видачі завдання "_____"_____2025 р.

Керівник бакалаврської роботи _____ _Шкарупило В.В., д.т.н., професор_
(підпис) (прізвище та ініціали)
Завдання прийняв до виконання _____ ___Левченко М.С.
(підпис) (прізвище та ініціали студента)

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів дипломного проекту (роботи)	Строк виконання етапів проекту (роботи)	Примітка
1	Аналіз предметної області	14.02.2025 р - 16.02.2025 р	Виконано
2	Проєктування комп’ютерної системи	01.03.2025 р - 19.03.2025 р	Виконано
3	Реалізація комп’ютерної системи	20.04.2025 р - 15.05.2025 р	Виконано
4	Тестування комп’ютерної системи	11.05.2025 р - 20.05.2025 р	Виконано
5	Оформлення пояснювальної записки	13.06.2025 р.	Виконано
6	Оформлення графічного матеріалу	13.06.2025 р.	Виконано

Студент _____ Левченко М.С.

(підпис) (ініціали та прізвище)

Керівник проекту (роботи) _____ Шкарупило В.В

(підпис) (ініціали та прізвище)

РЕФЕРАТ

Пояснювальна записка: 64 сторінки, 12 рисунків, 0 таблиць, 19 лістингів, 1додаток, 20 джерел.

КОМП'ЮТЕРНА СИСТЕМА, LabVIEW, GraphRenderer, OpenHAB, MONGO DB, TAILWIND CSS, JS, REACT, NEXT JS

Об'єкт аналізу – процес розроблення комп'ютерної системи для збору, обробки та візуалізації даних від периферійних пристроїв

Мета роботи – створення комп'ютерної системи, яка забезпечує збирання, аналіз і візуалізацію даних, отриманих від периферійних пристроїв.

Проєкт складається з чотирьох основних розділів.

У першому розділі проведено аналіз предметної області, розглянуто принцип роботи периферійних пристроїв, проаналізовано існуючі аналоги подібних систем та сформульовано основні задачі розробки.

Другий розділ присвячено проєктуванню системи: описано її архітектуру, обґрунтовано вибір інструментальних засобів та бази даних, наведено основні алгоритми функціонування.

У третьому розділі детально розглянуто реалізацію комп'ютерної системи, зокрема збирання та обробку даних, реалізацію алгоритмів аналізу, а також створення засобів для візуалізації результатів.

Четвертий розділ містить тестування комп'ютерної системи: описано користувацький інтерфейс, перевірено функціональність системи та її адаптивність до різних пристроїв.

У результаті роботи було розроблено повноцінну комп'ютерну систему, здатну ефективно обробляти та візуалізувати дані, отримані від периферійних пристроїв, відповідно до поставлених вимог.

ЗМІСТ

ВСТУП

У сучасних умовах стрімкого розвитку інформаційних технологій зростає потреба у створенні ефективних комп’ютерних систем, здатних працювати з великим обсягом даних, що надходять від різноманітних периферійних пристроїв. Зокрема, актуальним є завдання розроблення систем, які забезпечують не лише стабільне зчитування інформації з таких пристроїв, а й її обробку та подальшу візуалізацію у зручному для користувача вигляді. Подібні системи знаходять широке застосування у наукових дослідженнях, інженерії, промисловості, а також в освітніх і побутових сферах. У зв’язку з цим виникає потреба в глибокому аналізі існуючих методів опрацювання даних та підходів до побудови відповідного програмного забезпечення, яке може взаємодіяти з різними типами периферійних пристроїв.

Обрана тема потребує комплексного підходу, що охоплює як вивчення технічних характеристик зовнішніх пристроїв, так і розроблення програмної логіки для прийому, обробки та візуального представлення даних. Розроблення такої системи вимагає поєднання знань у галузях архітектури комп’ютерних систем, мікроелектроніки, програмування та дизайну інтерфейсів. Під час підготовки до реалізації проєкту здійснюється аналіз технологій, що можуть бути використані для побудови гнучкої, масштабованої та надійної системи, здатної до адаптації під різні технічні умови. Особливу увагу приділено вибору засобів взаємодії з апаратною частиною, ефективним методам обробки потокових даних та інструментам візуалізації, які дозволяють зробити результат зрозумілим та придатним для подальшого аналізу.

Таким чином, у межах даної роботи розглядається процес розроблення комп’ютерної системи, що виконує функції збору, обробки та візуалізації даних з периферійних пристроїв.

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 Принцип роботи переферійних пристроїв

Периферійні пристрої становлять ключовий компонент інформаційної інфраструктури, що забезпечує адаптацію цифрових процесів до потреб користувача та до фізичного світу в цілому. Їхнє значення виходить за межі простого технічного доповнення до центрального обчислювального ядра, оскільки вони фактично реалізують функціональну спроможність комп'ютерної системи до сприймання, перетворення, відображення й збереження інформаційних потоків, що циркулюють між суб'єктом, даними та виконавчими компонентами. До пристроїв введення, як правило, належать клавіатури, миші, сканери, мікрофони, камери — тобто все, що забезпечує надходження інформації до комп'ютера. У свою чергу, пристрої виведення (монітори, принтери, акустичні системи) транслюють оброблену інформацію у форму, зручну для сприйняття людиною або інших систем. Подвійні пристрої — такі як сенсорні екрани, мережеві адаптери, жорсткі диски, флеш-накопичувачі — можуть виконувати функції як введення, так і виведення, що дозволяє оптимізувати інформаційний обмін у двонаправленому режимі.

Роль периферійних пристроїв не обмежується лише забезпеченням базових функцій. У контексті сучасного програмного забезпечення вони активно використовуються для реалізації адаптивних, інтелектуальних та контекстно-залежних інтерфейсів, що реагують на дії користувача у реальному часі. Наприклад, графічні планшети та контролери віртуальної реальності слугують не просто засобами введення, а й механізмами просторової навігації та моделювання, які змінюють парадигму взаємодії між людиною й машиною. Крім того, із розвитком мобільних технологій периферійні пристрої стали базою для побудови комп'ютеризованих систем у межах інтернету речей, біометричного і сенсорного моніторингу, керування промисловими процесами, дистанційного навчання тощо.

З архітектурної точки зору, периферія виконує не лише роль фізичного інтерфейсу, а й формується у вигляді абстрактних рівнів апаратної віртуалізації, драйверного обслуговування та логічного керування ресурсами, що забезпечує її інтеграцію у програмне середовище. Це дозволяє створювати модульні та масштабовані системи, в яких будь-який пристрій може бути динамічно підключений, розпізнаний та використаний без необхідності зміни ядра операційної системи. Таким чином, периферійні пристрої становлять не лише інтерфейс взаємодії, але й платформу для розширення функціональності, гнучкої адаптації та підвищення ефективності комп'ютерних систем у різних сферах застосування.

Описана класифікація периферійних пристроїв за функціональним призначенням ґрунтується на принципах організації інформаційного обміну між користувачем і комп'ютерною системою. Пристрої введення є засобами перетворення фізичних або аналогових сигналів у цифрові дані, які можуть бути інтерпретовані програмним забезпеченням. Вони становлять початкову ланку у процесі взаємодії, оскільки забезпечують надходження інформації від людини або середовища до центрального процесора [1]. Саме через ці пристрої здійснюється не лише введення команд, а й зчитування контексту навколишнього середовища, що є критичним у задачах автоматизованого керування, адаптивних систем, біометричного розпізнавання, обробки мовлення та зображень. Технологічне вдосконалення сенсорів, мікрофонних матриць або сканувальних пристроїв значно розширює можливості комп'ютерного зору, слуху, а отже й загальну здатність системи до інтерпретації вхідних даних.

Пристрої виведення, у свою чергу, реалізують протилежний напрям обміну: вони забезпечують матеріалізацію результатів цифрової обробки у вигляді, сприйнятному для користувача або іншої системи [2]. Через дисплеї візуалізуються графічні інтерфейси, за допомогою принтерів здійснюється перенесення інформації на папір, а динаміки транслюють акустичні сигнали. Вивід може бути також оптичним, механічним, вібраційним або комбінованим, залежно від вимог до зворотного зв'язку. Важливим аспектом функціонування

пристроїв виведення є затримка, роздільна здатність, точність передачі кольору або звуку, що критично для таких сфер, як дизайн, телемедицина, керування складними технічними об’єктами.

Комбіновані пристрої відіграють особливу роль у побудові інтерактивних або автономних систем. Завдяки здатності одночасно фіксувати й видавати дані, вони забезпечують повноцінну двосторонню комунікацію. Такі пристрої не тільки підтримують адаптивність взаємодії, а й дозволяють будувати гнучкі інформаційні моделі з розподіленою обробкою. Сенсорні екрани, наприклад, дають змогу реалізовувати прямий тактильний вплив на інтерфейс, уникаючи необхідності використання окремих пристроїв введення. Мережеві адаптери функціонують на рівні передачі даних між системами, уможливлюючи колективне обчислення, доступ до віддалених ресурсів, синхронізацію й балансування навантажень.

Необхідність чіткої класифікації периферійних пристроїв пояснюється як концептуальними, так і практичними міркуваннями, оскільки структура інформаційних систем дедалі більше тяжіє до ускладнення, динамічного масштабування та інтеграції з багатокомпонентним середовищем. Зокрема, кожна з функціональних груп пристроїв формує окремий тип інформаційного потоку, який має специфічні характеристики щодо швидкості, обсягу, формату та часової чутливості, що зумовлює різні вимоги до каналів передачі, обчислювальних ресурсів та алгоритмів попередньої обробки. Пристрої введення, наприклад, генерують нерегулярні за обсягом, але критичні за часовими параметрами потоки, обробка яких повинна відбуватися із мінімально можливою затримкою, особливо у випадках інтерактивної взаємодії чи контролю фізичних об’єктів. Це вимагає використання механізмів пріоритетного доступу до ресурсів, асинхронної обробки подій, буферизації й фільтрації шумів, що інтегрується на рівні драйверів та прикладного програмного забезпечення [3].

З іншого боку, пристрої виведення мають переважно регулярну й контрольовану модель роботи, де значення набуває не стільки швидкодія, скільки стабільність, точність відтворення, та мінімізація навантаження на

користувача. У випадках візуалізації графіки, відтворення звуку або друку документів надзвичайно важливими стають параметри роздільної здатності, глибини кольору, частоти оновлення, динамічного діапазону. Відповідно, для таких пристроїв розробляються спеціалізовані протоколи, стандарти інтерфейсів (HDMI, DisplayPort, PCL тощо) та механізми обробки команд, що гарантують узгодженість між системними модулями.

Особливу складність становлять комбіновані пристрої, які одночасно генерують і приймають дані. Їхнє використання передбачає наявність двостороннього каналу зв'язку, підтримку дуплексної передачі та синхронізації з боку операційної системи й апаратної платформи. Це вимагає реалізації узгоджених стеків протоколів, таких як TCP/IP, USB OTG, Bluetooth HID, а також контролю за цілісністю та послідовністю обміну. Наприклад, при використанні зовнішнього мережевого адаптера система повинна постійно здійснювати контроль стану підключення, пакування даних, обробку втрат, дублювання та перевірку контрольних сум. Таким чином, поділ периферійних пристроїв не є формальністю, а функціонує як базовий механізм оптимізації архітектури системного обслуговування, забезпечення сумісності компонентів та підвищення надійності інтеграційного середовища в умовах високої варіативності та постійного оновлення технічної бази.

Структура взаємодії між периферійними пристроями та центральною обчислювальною частиною комп'ютера побудована за принципом модульної архітектури, що передбачає чітке розмежування між фізичним апаратним рівнем і логікою керування, реалізованою на програмному рівні. На апаратному рівні ключовими елементами такої архітектури виступають шини, контролери й порти, які формують основу для електричної та логічної комунікації. Шини (зокрема, USB, PCI Express, SATA, Thunderbolt) забезпечують високошвидкісний обмін даними між системною платою та периферією, водночас формуючи уніфіковане середовище для передавання як адресної, так і керівної інформації. Їхня пропускна здатність, енергоспоживання, топологія

з’єднань та підтримка протоколів визначають можливість масштабування й продуктивність системи загалом.

Ключову функцію виконують контролери периферійних пристроїв, які виступають посередницькою ланкою між низькорівневою фізичною передачею сигналів та високорівневою логікою операційної системи. Контролер зчитує сигнали, які надходять через інтерфейсні порти, перетворює їх у структури, сумісні з внутрішньою шиною даних, і передає до центрального процесора або оперативної пам’яті. У зворотному напрямі він здійснює інтерпретацію команд і перетворення їх у сигнали, зрозумілі для виконавчої частини пристрою. Завдяки цьому можлива реалізація безперервної, синхронної або асинхронної передачі даних відповідно до характеристик периферії. Наприклад, інтерфейс USB підтримує гаряче підключення, автоматичне визначення пристрою, а також живлення через той самий порт, що значно спрощує експлуатацію. PCI Express, своєю чергою, орієнтований на високошвидкісні внутрішні пристрої, що вимагають мінімальних затримок, наприклад, відеокарти або твердотільні накопичувачі [4].

В умовах розвитку бездротових технологій дедалі більшого поширення набувають інтерфейси, що не потребують фізичного з’єднання. Протоколи Bluetooth, Wi-Fi, NFC дозволяють здійснювати повноцінну інтеграцію периферійних пристроїв, зберігаючи мобільність та мінімізуючи обмеження, пов’язані з портами вводу/виводу. У цьому випадку архітектура передбачає додаткові шари взаємодії, пов’язані з аутентифікацією, шифруванням, управлінням частотним діапазоном та уникненням конфліктів каналів, що ускладнює реалізацію, проте значно розширює функціональні можливості системи. Таким чином, взаємодія периферійних пристроїв із комп’ютерною системою базується на ієрархії апаратних і програмних компонентів, де кожен елемент має своє чітко визначене місце й функцію, а ефективність системи залежить від узгодженості всіх складових на кожному з рівнів.

На програмному рівні архітектура взаємодії периферійних пристроїв із комп’ютерною системою реалізується шляхом впровадження спеціалізованих

низькорівневих компонентів — драйверів, що виконують функцію проміжної ланки між операційною системою та апаратною частиною пристрою. Драйвери інкапсулюють апаратно-залежну логіку, транслюючи універсальні інструкції, сформульовані ядром операційної системи або прикладними програмами, у специфічні сигнали, які розпізнає контролер пристрою. Саме цей процес дозволяє забезпечити модульність і масштабованість системного програмного забезпечення, уможливлюючи підтримку широкого спектру пристроїв без потреби змінювати архітектуру ядра або користувацького програмного середовища [5].

У більшості сучасних операційних систем використовується концепція драйверів, що функціонують як частина ядра (kernel-mode drivers) або у вигляді модулів користувацького рівня (user-mode drivers), що дозволяє досягти балансу між продуктивністю й стабільністю системи. Драйвер ініціалізується при першому підключенні пристрою, зчитує ідентифікатори обладнання (наприклад, Vendor ID та Product ID), проводить процедуру завантаження налаштувань, резервує необхідні ресурси (канали введення/виведення, переривання, буфери), та після цього забезпечує готовність до передачі даних. Крім того, драйвери відповідають за моніторинг функціонального стану пристрою, реагування на виняткові ситуації, обробку помилок з урахуванням таблиць помилок, і за потреби ініціюють оновлення мікропрограмного забезпечення (firmware update), якщо це передбачено виробником.

Завдяки драйверам можливе абстрагування апаратної реалізації від логіки прикладного програмного забезпечення: розробники застосунків можуть оперувати високорівневими інтерфейсами API, не занурюючись у деталі сигналізації або електричних параметрів. Це забезпечує кросплатформену сумісність, спрощує розгортання систем у гетерогенному середовищі та підтримує масштабованість на рівні обслуговування обладнання. У складних випадках — наприклад, при роботі з пристроями реального часу, апаратними прискорювачами або мережевими інтерфейсами з QoS — драйвери також виконують динамічну адаптацію режимів роботи пристрою до поточного

навантаження, пріоритетів процесів або енергоспоживання. Таким чином, драйвери виступають не лише засобом забезпечення доступу до периферії, а й критично важливим компонентом програмної архітектури, що формує фундамент стабільної, безпечної й ефективної взаємодії апаратної й програмної частин комп'ютерної системи.

Архітектурний підхід до взаємодії з периферійними пристроями в сучасних комп'ютерних системах ґрунтується на принципах розподіленої обробки та автономності окремих компонентів, що дозволяє значно знизити навантаження на центральний процесор і водночас забезпечити високу швидкодію та стійкість до помилок. Центральний процесор, будучи основним виконавцем логіки програм, передає частину рутинних або специфічних функцій контролерам периферії, які здійснюють попередню обробку, буферизацію та фільтрацію даних без участі основного обчислювального ядра. Наприклад, контролери вводу/виводу, мережеві адаптери, графічні процесори та накопичувачі на базі NVMe можуть мати власні процесорні ядра або мікропрограмне забезпечення, що дозволяє їм діяти як незалежні вузли обробки даних.

Такий розподіл дозволяє зменшити затримки, пов'язані з обробкою великих обсягів інформації в реальному часі, а також запобігає виникненню вузьких місць у передачі даних між різними компонентами. Зокрема, це важливо для мультимедійних задач, де необхідно забезпечити паралельну обробку відео, аудіо та сенсорної інформації з високою частотою оновлення. Також така архітектура мінімізує ризики, пов'язані з деградацією продуктивності: збій або перевантаження одного периферійного модуля не викликає каскадного падіння всієї системи, оскільки відсутній централізований вузол керування всіма пристроями.

Додатково, широке впровадження технологій автоматичного розпізнавання пристроїв, зокрема "plug-and-play", стало каталізатором зростання гнучкості комп'ютерних систем. Вказана технологія забезпечує ідентифікацію нового обладнання на рівні системної шини, ініціює пошук відповідного

драйвера в базі операційної системи або в інтернет-репозиторії, а також проводить автоматичну ініціалізацію та конфігурацію ресурсоємних параметрів. У результаті, користувач отримує готовий до використання пристрій без необхідності здійснювати ручне налаштування адрес, переривань або конфігурацій шин, що було типовим у ранніх етапах розвитку ПК.

У сукупності всі зазначені механізми формують архітектурну основу для побудови динамічних, адаптивних та масштабованих систем, які здатні не лише реагувати на зміну конфігурації, а й проактивно оптимізувати свої ресурси з урахуванням поточних умов експлуатації. Це забезпечує безперервність роботи, знижує поріг технічної підготовки користувача та створює передумови для формування екосистем інтелектуального середовища, що самоорганізовується — як у персональних комп'ютерах, так і в промислових, медичних чи мобільних обчислювальних платформах.

Таким чином, архітектура взаємодії між периферійними пристроями та комп'ютерною системою є складним, багаторівневим механізмом, в основі якого лежить тісна інтеграція апаратної та програмної складових. Вона дозволяє не лише забезпечити стабільний обмін даними, а й створити підґрунтя для подальшого розвитку систем автоматизованого керування, інтернету речей та інших високотехнологічних рішень, де роль периферії стає дедалі важливішою [6].

Процес обміну даними між периферійними пристроями та комп'ютерною системою є не лише базовим елементом архітектури обчислювального комплексу, а й визначальним чинником ефективності його функціонування в умовах інтенсивної обробки інформації. Концептуально такий обмін ґрунтується на моделі вводу/виводу, яка формалізує двосторонню передачу даних: з одного боку — надходження інформації від пристрою до центрального процесора, з іншого — передача керуючих команд або результатів обчислень у зворотному напрямку. Ця модель реалізується шляхом залучення як фізичних інтерфейсів, так і логічних протоколів, що спільно формують цілісний канал комунікації між апаратним та програмним середовищем.

На практичному рівні обмін даними здійснюється через багаторівневу архітектуру, у якій кожен шар виконує чітко окреслену функцію: починаючи від фізичного рівня, що передбачає електричну передачу сигналів через шини та порти, і завершуючи рівнем операційної системи, яка інтерпретує ці сигнали як команди або повідомлення. У межах цього процесу важливу роль відіграє буферизація, яка дає змогу згладжувати різницю в швидкості обробки між повільнішими периферійними пристроями та високошвидкісним процесором, запобігаючи втраті або спотворенню даних. Буфери, зазвичай реалізовані як частина оперативної пам’яті або внутрішніх регістрів контролерів, тимчасово зберігають інформацію, дозволяючи системі асинхронно обробляти вхідні та вихідні потоки без потреби в безперервному доступі до ресурсу.

Обмін даними супроводжується також системою сигналізації, яка включає механізми переривань, прямого доступу до пам’яті (DMA) та циклічного опитування. Переривання дозволяють пристроям повідомляти центральний процесор про події, що вимагають обробки, з мінімальною затратою ресурсів; DMA — передавати великі блоки даних без участі процесора, що знижує його завантаження; а опитування — забезпечує контроль стану пристрою в регулярному ритмі, хоча й менш ефективне в багатозадачних середовищах. Вибір механізму обміну залежить від типу пристрою, обсягу даних, критичності часу реакції та загальної архітектури системи.

Таким чином, процес обміну між комп’ютером і периферійними пристроями є не лише технічним каналом передавання інформації, а й складним механізмом, що поєднує апаратну реалізацію з програмною логікою. Він вимагає ретельної синхронізації, забезпечення сумісності інтерфейсів, управління чергами даних і підтримки узгоджених протоколів обміну. Від ефективності цього процесу залежить загальна продуктивність обчислювальної системи, стабільність її функціонування та можливість інтеграції нових пристроїв у мінливому технічному середовищі.Передача даних може здійснюватися в кількох режимах, залежно від типу пристрою, характеру інформації та потреб конкретного застосування. Найпоширенішими є синхронний та асинхронний

режими, кожен з яких має свої технічні переваги. У синхронному обміні дані передаються із фіксованим тактовим сигналом, що забезпечує передбачуваність і злагодженість у роботі системи. Водночас, асинхронний режим дозволяє здійснювати обмін у довільний момент, що знижує затримки при обробці непостійних потоків інформації, зокрема при роботі з сенсорами або подієво орієнтованими пристроями.

Буфери, як структурний елемент архітектури вводу/виводу, відіграють критичну роль у забезпеченні стабільності та ефективності обміну даними між периферійними пристроями та центральною обчислювальною частиною системи. Їх основне призначення полягає в тимчасовому накопиченні інформації, яка надходить від пристрою або призначена для нього, з метою компенсації різниці у швидкодії між джерелом і споживачем даних. Такий підхід дозволяє уникнути ситуацій, за яких менш продуктивний компонент системи стає «вузьким місцем», що обмежує загальну пропускну здатність і призводить до втрати даних або збоїв у роботі.

Буферизація особливо актуальна в середовищах, де використовується асинхронний або псевдоасинхронний обмін — наприклад, у випадках зчитування аудіопотоку з мікрофона, обробки відеосигналу з камери, передавання пакетів у мережевому адаптері або виведення зображення на дисплей з частотою оновлення, що перевищує внутрішню частоту обробки системи. У таких випадках дані накопичуються у буфері — спеціальному регістрі або сегменті оперативної пам’яті, — з якого вони згодом, у контрольованому режимі, передаються до адресата. Завдяки цьому забезпечується рівномірне навантаження на процесор, зменшується ризик перевантаження шин, і створюються умови для багатопоточності в обробці потоків.

Окрім вирівнювання темпу обміну, буфери також виконують функцію захисту від пошкоджень даних, які можуть виникнути внаслідок конфліктів доступу або нестабільності передавання. Сучасні реалізації драйверів передбачають механізми контролю заповнення буферів, зокрема використання

циклічних черг, семафорів та пріоритетів обробки. Це дозволяє організувати динамічне керування доступом до буферизованої інформації відповідно до вимог до часу реакції, обсягу даних або типу пристрою. Наприклад, у мережевих стекових протоколах дані буферизуються до моменту повного складання пакету, після чого здійснюється перевірка контрольної суми, і лише тоді інформація передається до прикладного рівня.

З боку операційної системи обробка вводу/виводу реалізується через спеціальні програмні інтерфейси та черги обробки запитів. Кожен запит на читання або запис ставиться в чергу й обробляється у порядку надходження, або згідно з пріоритетами, визначеними системними політиками [7]. Це дозволяє забезпечити справедливий розподіл ресурсів між усіма активними процесами та уникнути конфліктів при одночасному зверненні кількох програм до одного й того самого пристрою. У контексті багатозадачних систем також застосовуються механізми переривань, які дозволяють пристрою повідомити процесору про готовність до обміну, мінімізуючи витрати часу на опитування стану пристрою вручну.

Таким чином, принципи вводу/виводу формують технічну і програмну основу для функціональної взаємодії всіх компонентів обчислювальної системи з периферійним середовищем. Їх правильна реалізація є критично важливою для забезпечення стабільності, ефективності та передбачуваності роботи комп’ютера в умовах взаємодії з великою кількістю зовнішніх пристроїв, що генерують або приймають дані у режимі реального часу.

Функціонування периферійних пристроїв у складі комп’ютерної системи неможливе без наявності відповідного програмного забезпечення, що забезпечує коректну взаємодію апаратної частини з операційним середовищем. Центральну роль у цьому процесі відіграють драйвери — спеціалізовані програмні компоненти, які виступають у ролі посередника між обладнанням і програмним забезпеченням вищого рівня. Їх основним завданням є інтерпретація команд операційної системи у форму, яку розуміє конкретний пристрій, а також

зворотна передача результатів взаємодії у стандартизованому вигляді, придатному для обробки на рівні системних процесів [8].

Драйвери забезпечують ініціалізацію пристрою після підключення, встановлюють параметри взаємодії, забезпечують контроль за його станом і виконують обробку помилок, які можуть виникати в процесі обміну даними. Крім того, сучасні драйвери часто мають розширений функціонал, що включає автоматичне оновлення, моніторинг продуктивності, підтримку різних режимів роботи та адаптацію під конфігурацію системи. У випадку зі складними пристроями, такими як графічні прискорювачі, мережеві адаптери або багатофункціональні принтери, драйвер може містити додаткові модулі або навіть графічний інтерфейс користувача для зміни налаштувань і діагностики.

Особливе значення має програмне забезпечення в контексті стабільності та безпеки системи. Невірно встановлений або несумісний драйвер може спричинити системні збої, конфлікти ресурсів або повну втрату працездатності пристрою. Саме тому сучасні операційні системи реалізують механізми цифрового підпису драйверів, перевірки їх сумісності та ізоляції віртуального простору, щоб запобігти впливу потенційно небезпечних компонентів на основне ядро системи [9]. Також програмна підтримка включає модулі діагностики, які дозволяють визначити причини несправності та відновити стабільну роботу пристрою без фізичного втручання.

Важливо підкреслити, що роль програмного забезпечення не обмежується лише базовою підтримкою функціонування. Воно також формує інтерфейс між пристроєм і користувачем, дозволяючи адаптувати поведінку периферії до конкретних потреб. У випадку сенсорів, наприклад, драйвер може фільтрувати шум або інтерпретувати вхідні сигнали згідно з заданими алгоритмами. Для пристроїв з високим ступенем варіативності налаштувань, таких як аудіоінтерфейси чи відеокамери, програмне забезпечення стає ключовим елементом, що визначає не лише стабільність, а й функціональні можливості всієї системи загалом.

Отже, драйвери та допоміжне програмне забезпечення виконують критично важливу функцію у контексті інтеграції периферійних пристроїв у комп’ютерне середовище [10]. Вони забезпечують надійний зв’язок між рівнями апаратної та програмної архітектури, створюючи умови для ефективного, безпечного й адаптивного функціонування пристроїв у різних умовах експлуатації.

Побудова ефективної взаємодії між периферійними пристроями та комп’ютерною системою значною мірою залежить від правильно організованих інтерфейсів та протоколів зв’язку, які визначають формат, швидкість, послідовність і спосіб передавання даних. Під інтерфейсом розуміється апаратна або програмна система засобів, через яку здійснюється фізичне або логічне з'єднання пристроїв з комп’ютером. Протокол зв’язку, у свою чергу, є формалізованим набором правил, що регулює спосіб передавання, прийому та підтвердження інформації. Спільне функціонування інтерфейсу та протоколу забезпечує надійність обміну даними та узгодженість у роботі пристроїв.

У сучасних комп’ютерних системах застосовується велика кількість інтерфейсів, які поділяються за способом передавання сигналу (паралельний або послідовний), типом середовища (дротове чи бездротове), а також за швидкісними характеристиками. Найбільш універсальним прикладом є USB-інтерфейс, що завдяки широкій підтримці, високій швидкості обміну та зручності використання став основним стандартом для підключення більшості зовнішніх пристроїв. У сфері передачі аудіо- та відеосигналів використовуються такі інтерфейси, як HDMI та DisplayPort, які дозволяють передавати інформацію високої чіткості з мінімальними втратами. Для високошвидкісної взаємодії з накопичувачами даних активно використовується протокол SATA або новіші стандарти NVMe через шину PCIe.

Протоколи зв’язку становлять фундаментальну складову механізмів обміну даними між комп’ютерною системою та периферійними пристроями, оскільки саме вони регламентують спосіб організації передачі інформації на рівні електричних, логічних та семантичних сигналів. В основі будь-якого

протоколу лежить чітко визначена послідовність дій, що охоплює встановлення з’єднання, обмін службовими сигналами, формування пакету даних із заголовком і корисним навантаженням, а також механізми перевірки достовірності й повторної передачі в разі виявлення помилок. Таке структуроване середовище взаємодії забезпечує не лише коректність обміну, а й дозволяє адаптуватися до різноманітних характеристик пристроїв — від швидкості передачі до обмежень по енергоспоживанню та відстані.

Серед найбільш поширених у контексті вбудованих систем і сенсорних мереж протоколів варто виокремити I²C (Inter-Integrated Circuit), що є синхронним серійним протоколом з мульти-мастерною архітектурою. Його особливістю є використання лише двох ліній — SDA (Serial Data) та SCL (Serial Clock), які обслуговують як передавання даних, так і керування передачею, що дозволяє реалізовувати компактні схеми з великою кількістю підключених модулів. Його типове застосування охоплює зчитування значень температурних сенсорів, акселерометрів, годинників реального часу тощо. SPI (Serial Peripheral Interface), навпаки, є швидшим, але вимагає додаткових сигнальних ліній (часто чотирьох), що робить його менш зручним для реалізації в багатокомпонентних системах, однак незамінним у випадках, де критичними є швидкодія та низька затримка — наприклад, у цифрових аудіопристроях або дисплейних контролерах.

У сфері бездротових технологій протоколи Bluetooth, Wi-Fi та ZigBee реалізують стандарти передачі на різні дистанції з урахуванням енергетичних і топологічних характеристик мережі. Bluetooth є оптимальним для персональних пристроїв із низьким енергоспоживанням і обмеженим радіусом дії, зокрема у сфері носимих технологій або бездротової периферії. Wi-Fi орієнтований на високошвидкісну передачу даних у локальних мережах і широко застосовується в інтерактивних пристроях із вимогами до мультимедійної синхронізації. ZigBee, у свою чергу, призначений для побудови розподілених сенсорних мереж із мінімальним споживанням енергії, що робить його ефективним у системах

автоматизації, моніторингу середовища та побудові енергоефективних IoT-рішень.

Таким чином, вибір протоколу зв’язку визначає функціональну придатність та архітектурну доцільність конкретного пристрою в межах комп’ютерної системи. Він обумовлює не лише фізичні характеристики інтерфейсу, а й логіку обміну, тип доступу до спільного каналу, процедури обробки помилок та забезпечення цілісності даних, що в сукупності формує основу надійного й продуктивного інформаційного середовища.

Вибір інтерфейсу та протоколу здійснюється з урахуванням технічних вимог до системи, обмежень апаратної реалізації, а також специфіки експлуатації. Наприклад, у реальному часі особливо важливими є мінімальні затримки, тому пріоритет надається інтерфейсам із гарантованою пропускною здатністю та протоколам з низькою затримкою. Для систем, де важлива масштабованість або гнучкість, доцільніше використовувати ті протоколи, що підтримують динамічне підключення нових пристроїв без зупинки основного процесу. Крім того, сучасні рішення все частіше включають підтримку шифрування та автентифікації, що дозволяє забезпечити захист переданої інформації в умовах потенційних загроз.

Таким чином, інтерфейси та протоколи зв’язку є критично важливими елементами архітектури обміну даними, які визначають функціональність, стабільність та продуктивність усієї комп’ютерної системи. Їх правильне поєднання забезпечує не лише фізичну можливість підключення пристрою, а й логічну узгодженість усіх етапів комунікації — від ініціалізації до завершення передачі, що є необхідною умовою для ефективної роботи сучасних периферійних пристроїв.

Живлення периферійних пристроїв є важливим компонентом функціонування комп’ютерної системи, що безпосередньо впливає на стабільність, надійність та ефективність обміну даними. Незважаючи на те, що увага часто приділяється переважно логіці взаємодії та протоколам передачі, забезпечення адекватного живлення є умовою, без якої жоден пристрій не може

функціонувати навіть на базовому рівні. У межах сучасної архітектури ПК та вбудованих систем живлення периферії може здійснюватися як безпосередньо від комп’ютера, так і за допомогою зовнішніх джерел — залежно від типу, потужності та призначення пристрою.

Для більшості пристроїв, що підключаються через стандартні порти, зокрема USB, передбачено живлення напряму від комп’ютерного джерела. Це дозволяє зменшити кількість зовнішніх адаптерів і забезпечити мобільність системи. Проте така архітектура має обмеження: максимальна сила струму, яку може надати порт, обмежена технічними характеристиками контролера, зазвичай не перевищуючи 500 мА для USB 2.0 або 900 мА для USB 3.0. У випадку одночасної роботи кількох енергоємних пристроїв, або ж при підключенні складних модулів, таких як зовнішні жорсткі диски, високоточні сенсори чи плати розширення, може виникнути дефіцит живлення, що призводить до збоїв, переривань у передачі даних або повного відключення периферії.

Окремі класи пристроїв, особливо ті, що мають активні елементи або високе енергоспоживання, потребують автономного живлення. Це стосується, зокрема, принтерів, сканерів, деяких медичних сенсорів або промислових пристроїв. Вони обладнані власними блоками живлення, які часто виконують функцію стабілізації напруги та захисту від перенавантажень. У таких системах важливо не лише наявність джерела живлення, а й правильна організація заземлення, екранування та захисту від імпульсних завад, особливо в умовах, де паралельно працює чутлива електроніка або здійснюється обробка аналогових сигналів.

З технічного погляду, у проєктуванні комп’ютерних систем з підключенням периферійних пристроїв необхідно враховувати розподіл споживання енергії, пускові струми, особливості терморегуляції та потенційне зниження ефективності при підвищенні навантаження. У вбудованих системах додатково враховується акумуляторне живлення або живлення з відновлюваних джерел, що вимагає оптимізації споживання на рівні програмного забезпечення

— наприклад, шляхом реалізації режимів сну, циклічного опитування або динамічного керування частотою обміну.

Таким чином, організація живлення є не лише енергетичною задачею, а й комплексним інженерним викликом, який охоплює аспекти електроніки, програмування та системного аналізу. Від правильного підходу до цього питання залежить не лише працездатність окремих пристроїв, а й загальна надійність та довговічність усієї системи. Зневажання вимогами до живлення може призвести до непередбачуваних збоїв, зниження продуктивності або навіть фізичного пошкодження компонентів, що особливо критично в умовах промислового чи наукового застосування.

У процесі передавання даних між периферійними пристроями та комп’ютерною системою можуть виникати різного роду помилки, спричинені як фізичними обмеженнями середовища передачі, так і логічними порушеннями протоколів комунікації. З огляду на це, обробка помилок та забезпечення безпеки передачі є ключовими аспектами при проєктуванні систем взаємодії з периферією. Їх реалізація спрямована на гарантування цілісності, достовірності та конфіденційності інформації, що надходить або передається зовнішніми пристроями.

На рівні фізичної передачі основними причинами помилок можуть бути електромагнітні завади, нестабільна напруга живлення, неспівпадіння швидкості обміну або дефекти контактів. Для виявлення таких помилок широко застосовуються алгоритми контролю чітності та цілісності даних. Зокрема, у більшості протоколів передбачено використання контрольних сум (наприклад, CRC — циклічний надлишковий код), які дозволяють виявити спотворення даних під час пересилання. У разі фіксації розбіжності система може ініціювати повторну передачу пакета або повідомити про збій. Також поширеними є механізми перевірки на рівні кадру, коли кожна одиниця переданої інформації супроводжується службовими байтами, що дозволяють контролювати її структуру та коректність.

У випадках, коли обробка даних відбувається в режимі реального часу або коли повторна передача неможлива через затримки або обмеження ресурсу, застосовуються методи виправлення помилок, зокрема коди з вбудованою надлишковістю, що дозволяють не лише виявити, а й частково відновити втрачену інформацію. Такий підхід актуальний при роботі з сенсорами, які передають великі обсяги неперервного потоку даних, наприклад, у системах моніторингу або дистанційного управління.

Окрему увагу заслуговує питання безпеки переданої інформації. У контексті використання бездротових технологій, а також підключення до публічних або напіввідкритих середовищ, існує загроза перехоплення, модифікації або несанкціонованого доступу до даних. Для запобігання подібним інцидентам впроваджуються алгоритми шифрування, автентифікації та керування правами доступу. Сучасні протоколи передбачають використання асиметричних схем захисту, цифрових підписів, а також систем виявлення підозрілої активності, що дозволяє своєчасно реагувати на потенційні загрози. У ряді випадків реалізується апаратне шифрування на рівні самого пристрою, що мінімізує ризики витоку даних навіть при компрометації комп’ютерної системи.

Загалом, реалізація надійних механізмів обробки помилок та безпеки передачі є необхідною умовою функціонування сучасних систем, особливо в умовах критичних застосувань, де втрата або спотворення інформації може мати серйозні наслідки. Система, здатна не лише виявити помилку, а й адаптивно відреагувати на неї, демонструє вищий рівень стійкості, що є важливим показником якості у сучасному інженерному середовищі.

1.2 Аналіз аналогічних розробок

Однією з найвідоміших і найбільш використовуваних систем для побудови програмно-апаратних комплексів збору та обробки даних з периферійних

пристроїв є середовище LabVIEW (Laboratory Virtual Instrument Engineering Workbench), розроблене компанією National Instruments [11] (рис.1.1).

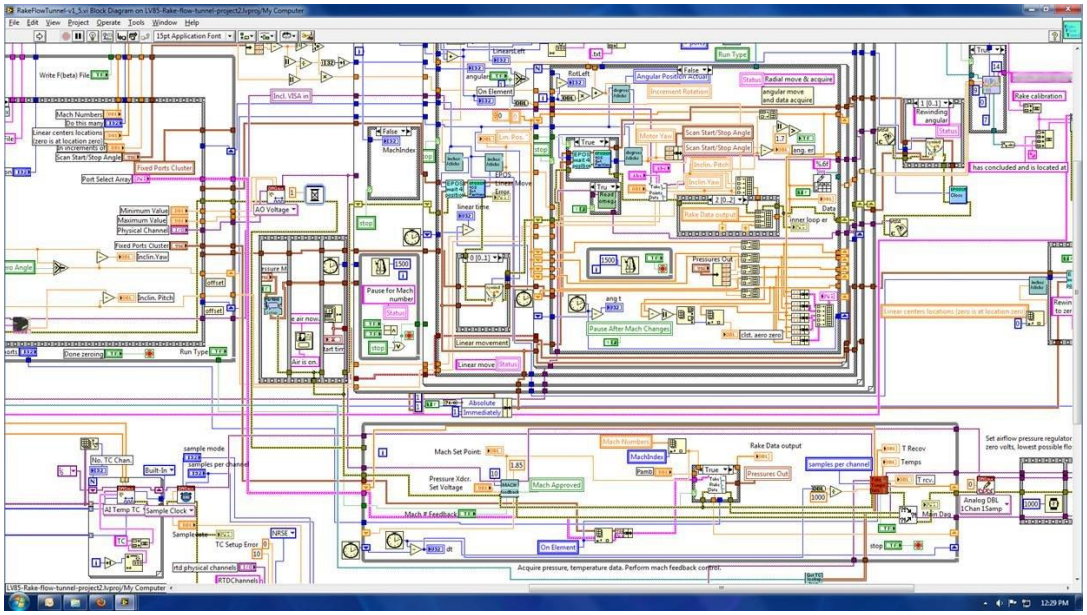

Рисунок 1.1 – Середовище LabVIEW

Це програмне забезпечення являє собою платформу графічного програмування, орієнтовану на інженерні та наукові задачі, зокрема ті, що потребують інтенсивної взаємодії з апаратними засобами. Характерною особливістю даної системи є використання візуального підходу до розробки — замість традиційного кодування застосовується побудова блок-схем, які моделюють логіку роботи системи, що істотно полегшує проектування складних структур і прискорює розгортання систем у промислових або лабораторних умовах.

LabVIEW підтримує широкий спектр периферійних інтерфейсів, включаючи USB, RS-232, GPIB, Ethernet, PCI, а також роботу з різноманітними модулями збору даних та сенсорами. За рахунок інтеграції з апаратною платформою NI DAQ системи, забезпечується стабільна взаємодія на фізичному рівні, а також можливість високоточної синхронізації сигналів. Програмне середовище надає великі можливості для візуалізації — від простих цифрових індикаторів до побудови складних графіків, таблиць, тривимірних моделей та панелей керування, що є надзвичайно корисним у завданнях оперативного контролю та аналізу.

Важливо зазначити, що середовище має високий ступінь модульності: функціональні блоки можна зберігати, повторно використовувати або змінювати, адаптуючи під різні проєкти. Крім того, LabVIEW підтримує паралельне виконання процесів, що особливо актуально при одночасній роботі з кількома каналами введення/виведення. У багатьох випадках реалізація проєкту у цьому середовищі дозволяє обійтися без написання низькорівневого коду, що значно зменшує час розробки і знижує вимоги до досвіду програміста. Водночас, така абстрагованість створює певні обмеження — LabVIEW досить вимогливий до ресурсів системи, а її функціональність сильно залежить від придбаних ліцензій і додаткових модулів.

Таким чином, середовище LabVIEW є потужним інструментом для побудови складних систем збору та візуалізації даних з периферійних пристроїв, яке має великий набір функціональних можливостей, проте не є універсальним рішенням для всіх випадків. Його доцільно використовувати у високотехнологічних проєктах із наявністю апаратної бази компанії National Instruments, але для індивідуальних або малобюджетних розробок існують більш доступні альтернативи.

Однією з помітних сучасних розробок, яка демонструє ефективну реалізацію збору та обробки даних з периферійних пристроїв, є OpenHAB (Open Home Automation Bus) — програмне забезпечення з відкритим кодом, створене для побудови гнучких систем автоматизації, здатних інтегрувати широкий спектр апаратних платформ та протоколів зв’язку (рис.1.2).

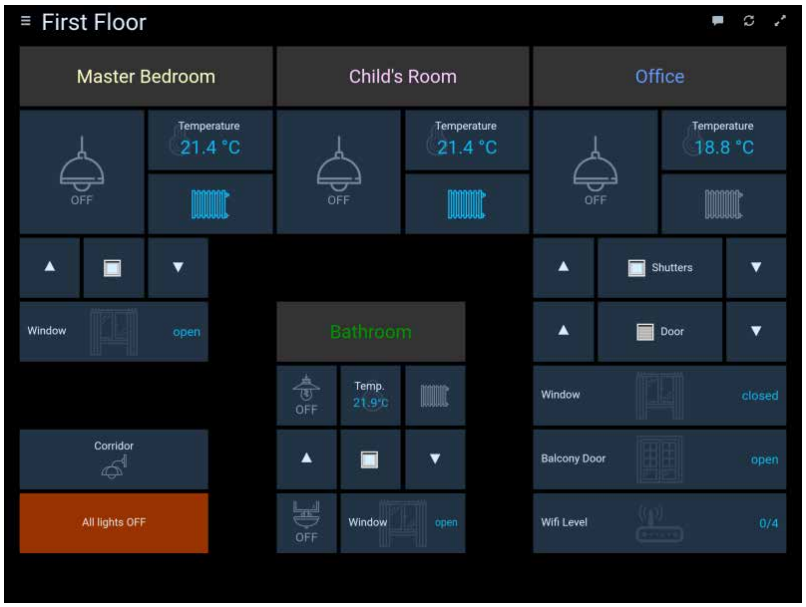

Рисунок 1.2 – Середовище OpenHAB

Попри свою первинну орієнтацію на сферу “розумного дому”, ця система знайшла застосування в задачах моніторингу, аналізу даних та створення користувацьких інтерфейсів, що робить її релевантною для порівняння у контексті розробки комп’ютерних систем взаємодії з периферійними пристроями [12].

OpenHAB реалізовано на мові Java, що забезпечує її кросплатформеність та можливість розгортання як на персональних комп’ютерах, так і на вбудованих системах, зокрема на базі Raspberry Pi. Ключовою архітектурною особливістю цієї системи є підтримка так званих “біндінгів” — модулів, що забезпечують інтеграцію з конкретними пристроями або протоколами (наприклад, MQTT, Modbus, Bluetooth, ZigBee, Z-Wave, HTTP тощо). Завдяки цьому OpenHAB здатен одночасно взаємодіяти з великою кількістю фізичних пристроїв, збирати від них дані, зберігати в базі, обробляти та виводити у вигляді графіків, індикаторів або повідомлень.

Особливістю OpenHAB є можливість створення адаптивного веб-інтерфейсу, який дозволяє в режимі реального часу керувати пристроями та слідкувати за показниками з будь-якого браузера. Для зберігання даних система використовує стандартні реляційні або часові бази даних (наприклад, InfluxDB), а для візуалізації часто інтегрується з такими інструментами як Grafana. Таким

чином, забезпечується гнучкий і розширюваний механізм роботи з інформацією, який може бути адаптований під потреби конкретного застосування.

Серед переваг OpenHAB варто виокремити відкритість коду, активну спільноту розробників, велику кількість доступних розширень і модулів, а також можливість повної кастомізації під індивідуальні потреби. Водночас, певною складністю для початкового користувача є досить високий поріг входження: система потребує налаштування конфігураційних файлів, глибокого розуміння логіки подій, правил автоматизації та структури взаємодії між компонентами. Проте, у випадку належної реалізації, вона демонструє високу стабільність, продуктивність та адаптивність до змін у середовищі.

Отже, OpenHAB є яскравим прикладом гнучкої програмної розробки, яка дозволяє будувати масштабовані, розподілені системи збору та обробки даних з периферійних пристроїв. Вона може бути ефективно використана як у побутовому, так і в напівпромисловому середовищі, забезпечуючи користувачам повний контроль над пристроями та доступ до інформації в зручному, візуалізованому форматі.

Ще однією прикладною розробкою, що демонструє практичне використання периферійних пристроїв у контексті збору та візуалізації даних, є середовище Processing (рис.1.3).

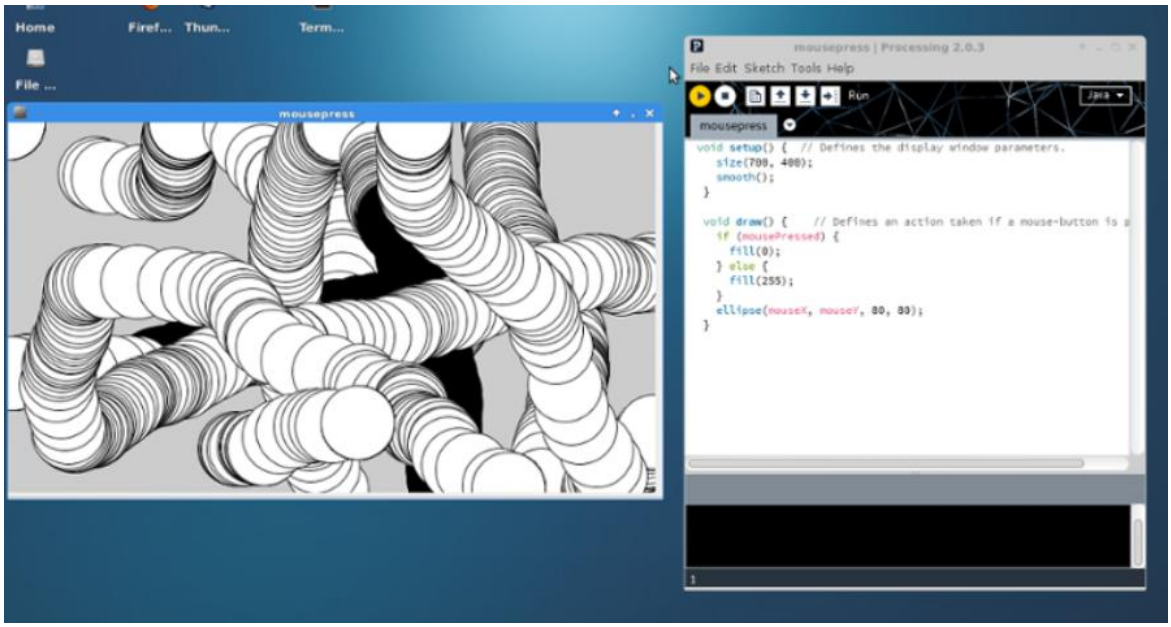

Рисунок 1.3 – Середовище Processing

Це вільно поширюване програмне середовище, створене для візуального програмування, яке спочатку розроблялося як інструмент для дизайнерів та митців, однак з часом набуло широкого застосування у технічних, освітніх та дослідницьких проєктах. Processing надає користувачеві зручний спосіб створення інтерактивної графіки, а також дозволяє зчитувати, обробляти та відображати дані з різноманітних зовнішніх пристроїв [13].

Перевагою цієї системи є простота синтаксису, яка базується на мові Java, що дозволяє швидко реалізовувати проєкти без необхідності глибокого занурення в складні архітектурні рішення. Через підтримку бібліотек, таких як Serial, Firmata або ControlP5, забезпечується взаємодія з периферійними пристроями — зокрема, з мікроконтролерами Arduino, сенсорами температури, вологості, ультразвуковими модулями та іншими фізичними датчиками. Таким чином, середовище Processing дозволяє організувати повноцінний цикл збору інформації — від її прийому до візуального подання в режимі реального часу.

Візуалізаційні можливості Processing охоплюють побудову графіків, анімацій, векторних об'єктів, а також тривимірних сцен, що робить його придатним для розробки адаптивних інтерфейсів моніторингу. Враховуючи це, середовище часто використовується у демонстраційних або навчальних проєктах, де важливо не лише отримати дані, а й зробити їх доступними для сприйняття користувачем. До того ж Processing має широку спільноту користувачів, велику кількість прикладів реалізації схожих проєктів та потужну документацію, що полегшує процес навчання та адаптації.

Основними обмеженнями цієї розробки є обмежена підтримка складних апаратних протоколів, невисока продуктивність при роботі з великими обсягами даних, а також відсутність вбудованих засобів обробки даних високого рівня. Проте в рамках невеликих систем збору з базовою фільтрацією та виведенням інформації Processing показує себе як надійне, гнучке та легко адаптовуване рішення.

У підсумку, середовище Processing є ефективним інструментом для реалізації систем збору та візуалізації даних з периферійних пристроїв у межах прототипування, навчальних проєктів та інтерактивних експериментів. Його відкритість, візуальна орієнтованість і низький поріг входу роблять його доступним вибором для широкого кола розробників, які прагнуть швидко втілити ідею у функціональний програмний продукт.

1.3 Постановка задачі

У рамках даного проєкту основний акцент зроблено на розробці програмного забезпечення, яке забезпечує повноцінне функціонування системи збору, обробки та візуалізації даних з периферійних пристроїв. Апаратна частина розглядається лише як джерело сирих вхідних сигналів, тоді як саме програмне забезпечення відповідає за реалізацію ключових функцій системи — починаючи від ініціалізації з’єднання з пристроєм і завершуючи побудовою зручного для сприйняття користувача інтерфейсу з виводом оброблених даних.

Основною задачею є створення модульного, масштабованого та стабільного програмного продукту, здатного працювати з різними типами периферійних пристроїв, незалежно від їх конкретної фізичної реалізації. Програмне забезпечення повинно забезпечувати універсальний механізм зчитування даних через відповідні інтерфейси зв’язку (наприклад, послідовний порт, USB або TCP-з’єднання), а також їх подальшу обробку, фільтрацію та структурування відповідно до заданих параметрів.

Окрему увагу приділено реалізації адаптивного інтерфейсу користувача з елементами динамічної візуалізації — графіками, індикаторами, цифровими шкалами тощо. Це дозволяє не лише представити дані у наочній формі, а й забезпечити оперативне реагування на зміни вхідних сигналів. Забезпечення інтерактивності інтерфейсу, можливості масштабування, зміни параметрів

обробки “на льоту” та підтримка кількох підключень є складовими частинами поставленого завдання.

Окрім візуального відображення, реалізується модуль збереження оброблених даних до локального сховища або бази даних для подальшого аналізу. Забезпечується також обробка помилок, виявлення розривів з’єднання, спроби повторного підключення та інші механізми, необхідні для підтримки надійної роботи системи в умовах нестабільного апаратного середовища. Таким чином, програмна частина виступає як єдиний повноцінний інтелектуальний центр системи, який формує логіку її функціонування, перетворюючи сировинні сигнали від пристроїв у структуровану, придатну для використання інформацію.

2 ПРОЄКТУВАННЯ КОМП'ЮТЕРНОЇ СИСТЕМИ

2.1 Архітектура системи

Архітектура системи представлена на рисунку 2.1.

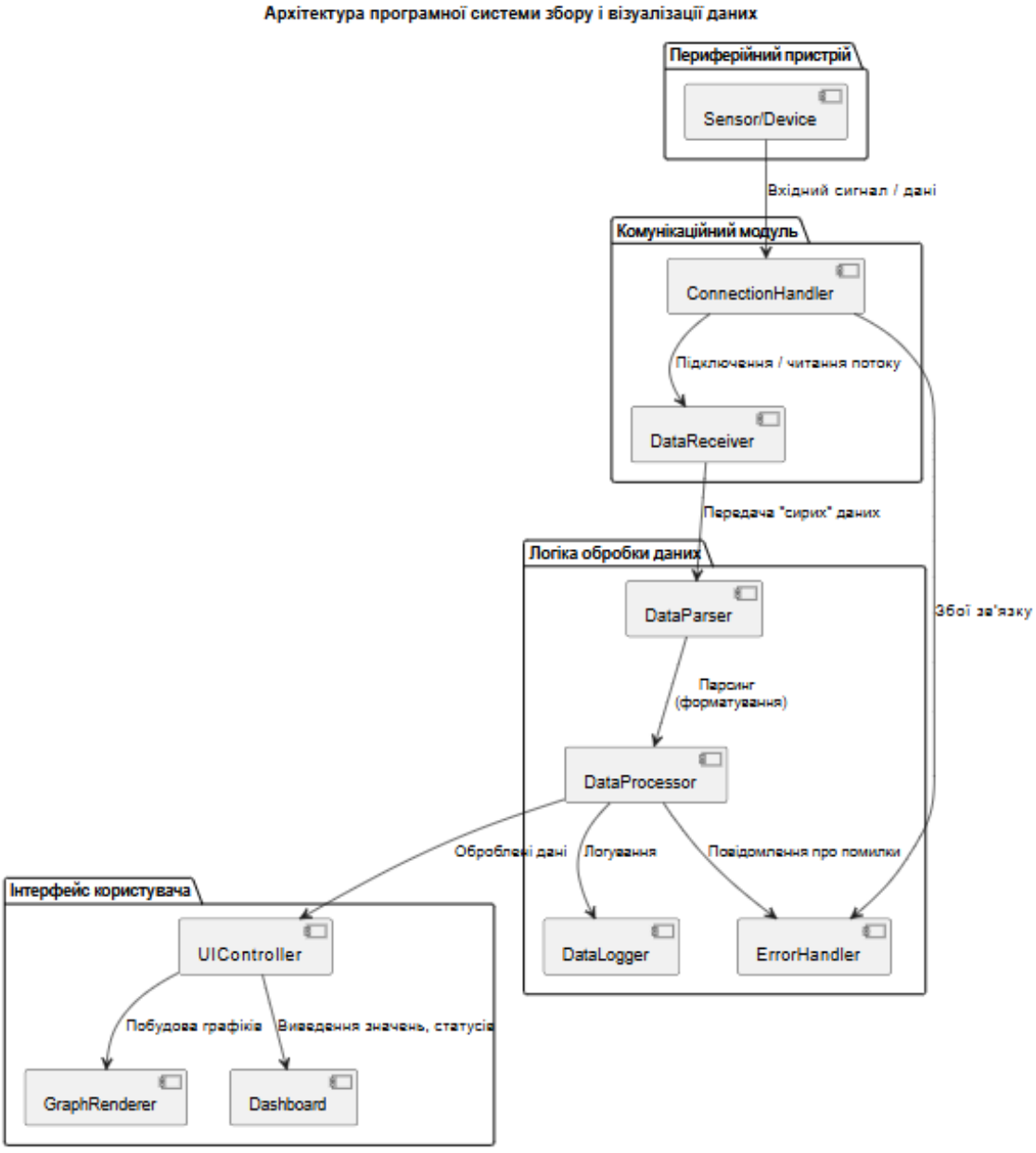

Рисунок 2.1 – Архітектура системи

У контексті представленої архітектури програмної системи периферійний пристрій виконує функцію джерела первинних даних. Він є фізичним модулем, що безпосередньо взаємодіє із зовнішнім середовищем та фіксує певні

параметри — наприклад, температуру, вологість, рівень освітлення, рух, електричні сигнали тощо. Його завдання обмежується збором аналогових або цифрових сигналів, перетворенням їх у машинозчитуваний формат (зазвичай через вбудований контролер або мікропроцесор) та передачею цих даних до комп’ютерної системи через визначений канал зв’язку.

Периферійний пристрій в архітектурі розглядається як пасивний учасник, що не має власної логіки обробки, не зберігає історичних даних і не здійснює аналізу. Його вихідні дані передаються до комунікаційного модуля, де вже на програмному рівні відбувається встановлення з’єднання, зчитування та подальша обробка інформації. Залежно від специфіки застосування, пристрій може функціонувати у режимі постійного надсилання даних (streaming), або працювати в подієво-орієнтованому режимі — реагуючи на зміну параметрів або отримання зовнішньої команди на вимірювання.

Таким чином, периферійний пристрій є ініціатором запуску програмної обробки, але самостійно не бере участі у процесах візуалізації, фільтрації чи зберігання даних. Його роль у системі є критично важливою для запуску повного програмного циклу, проте логіка функціонування зосереджена винятково у програмних модулях, що взаємодіють з пристроєм через визначений протокол комунікації.

Комунікаційний модуль у структурі програмної системи виконує роль посередника між периферійним пристроєм та основними внутрішніми компонентами, відповідальними за обробку й візуалізацію даних. Основне його призначення — забезпечення стабільного, безпечного та контрольованого каналу зв’язку з джерелом даних. У модулі виділяються два ключові функціональні компоненти: ConnectionHandler та DataReceiver.

ConnectionHandler відповідає за встановлення та підтримку з’єднання з периферійним пристроєм. Він реалізує логіку ініціалізації інтерфейсу (наприклад, відкриття серійного порту, TCP-з'єднання або іншого каналу передачі), перевірку доступності пристрою, конфігурацію параметрів з’єднання (швидкість, таймаути, формат даних) та обробку критичних ситуацій — таких як

втрата зв’язку, помилки передачі або зміна параметрів підключення під час виконання. В окремих випадках може також реалізовувати процедури автентифікації чи верифікації, якщо пристрій вимагає захищеного підключення.

DataReceiver безпосередньо обробляє потік вхідних даних після встановлення з’єднання. Він реалізує механізми прийому "сирих" сигналів або пакетів, фрагментації великих повідомлень, контроль їхньої цілісності, та передачу даних на наступний рівень — до DataParser. DataReceiver також може виконувати базову перевірку формату повідомлень, відкидаючи явно некоректні або неповні дані, з метою зниження навантаження на оброблювальний блок. У разі багатопоточності чи роботи з кількома пристроями одночасно, цей компонент керує чергами обробки та синхронізацією даних у відповідності до встановленого порядку пріоритетів.

Взаємодія між цими двома компонентами дозволяє забезпечити ефективну роботу всієї системи навіть за умови нестабільного середовища або змін у режимі роботи пристрою. При належному проєктуванні комунікаційний модуль здатен динамічно адаптуватися до змін умов передачі, виконуючи повторні спроби підключення, перепідключення, або перемикання на резервні канали зв’язку. Таким чином, комунікаційний модуль формує фундамент для подальшої обробки, гарантує надійність отримання інформації та захищає інші частини системи від впливу нестабільних зовнішніх факторів.

Блок логіки обробки даних виступає центральним функціональним елементом програмної архітектури системи. Саме в ньому реалізується ключовий цикл трансформації вхідної інформації, що надходить із периферійних пристроїв, у структуровані, аналітично значущі об'єкти, готові до візуалізації або збереження. Логіка складається з кількох взаємопов’язаних модулів, кожен із яких виконує визначену частину функціоналу.

Модуль DataParser відповідає за первинну обробку "сирих" вхідних даних, які надходять у вигляді потоку байтів або текстових фрагментів. Його завдання полягає в розпізнаванні структури повідомлення, виділенні окремих параметрів (наприклад, значення сенсорів, часових міток, службової інформації) та

приведенні їх до внутрішнього уніфікованого формату, що далі використовується всіма компонентами системи.

Наступний етап — модуль DataProcessor, який реалізує основну логіку обчислень, фільтрації, нормалізації та перетворення отриманих даних. У цьому модулі можуть бути застосовані алгоритми згладжування, порогової обробки, виявлення аномалій або обчислення похідних величин (наприклад, середніх значень, тенденцій, градієнтів). Він також формує підготовлені дані для виведення в UI та збереження в лог-файл або базу даних.

Модуль ErrorHandler виконує моніторинг коректності потоку, виявлення помилок формату, втрат даних, повторів або нестабільного з'єднання. За необхідності, він генерує повідомлення про помилки, які потрапляють до системи візуалізації або реєструються для подальшого аналізу. Реакції системи можуть бути адаптивними: повторний запит, реконекція або перехід у захищений режим.

Модуль DataLogger відповідає за збереження оброблених даних у визначене сховище. Це може бути як звичайний лог-файл, так і база даних. Дані зберігаються у форматі, що підтримує подальший аналіз, пошук, експорт або агрегацію. При необхідності реалізується ротація логів, контроль за обсягом даних, що накопичуються, або шифрування.

Таким чином, логічний блок системи є ядром, яке забезпечує узгоджену та стійку роботу всієї програми. Саме він забезпечує абстрагування від нестабільностей апаратного рівня, адаптує дані для потреб користувача, реалізує інтелектуальну обробку та підтримує стабільний цикл життєдіяльності інформації в межах програмної системи.

Інтерфейс користувача в межах даної архітектури виконує роль центрального візуального середовища, через яке здійснюється безпосередня взаємодія між системою та користувачем. Основна його функція полягає не лише у відображенні оброблених даних, але й у забезпеченні доступу до ключових параметрів, управління режимами роботи, а також у моніторингу поточного стану системи.

Згідно з розробленою структурою, інтерфейс реалізується через декілька взаємопов’язаних модулів. Компонент UIController відповідає за логіку інтерфейсу — ініціалізацію елементів, обробку подій, оновлення віджетів, а також координацію взаємодії між підсистемами. Саме через нього відбувається передача оброблених даних із ядра системи до візуального шару.

Візуалізаційна частина представлена модулем GraphRenderer, який забезпечує графічне відображення динамічних даних у формі графіків, гістограм або інших типів інфографіки. Цей модуль адаптовано до роботи в реальному часі та підтримує оновлення графічних компонентів без перезавантаження всієї сцени, що дозволяє зберігати безперервність візуального потоку.

Допоміжним елементом виступає Dashboard — віртуальна панель індикаторів, яка об’єднує текстові поля, цифрові лічильники, кольорові статуси, повідомлення про помилки та інші інтерактивні елементи. Завдяки цьому користувач може отримати оперативну інформацію про поточний стан системи, змінити налаштування, а також проаналізувати динаміку показників у зручному форматі.

Інтерфейс розроблено з урахуванням принципів адаптивності та мінімалістичного дизайну: компоненти мають чітку структуру, забезпечують інтуїтивне розміщення інформації та мінімізують ризик помилкових дій з боку користувача. Передбачено також можливість масштабування — підтримка різних роздільностей екрана, робота у віконному або повноекранному режимі, а також зміна теми оформлення залежно від умов експлуатації (наприклад, нічний режим для роботи в темному середовищі).

Таким чином, інтерфейс користувача є не лише оболонкою для виводу інформації, а повноцінним інтерактивним середовищем, яке виконує функції візуалізації, контролю та забезпечення зворотного зв’язку між користувачем і внутрішніми модулями системи. Його реалізація є критичним етапом проєкту, оскільки саме через нього оцінюється зручність, зрозумілість та ефективність усієї програмної розробки.

2.2 Вибір та обгрунтування інструментальних засобів

Для реалізації програмного забезпечення системи збору, обробки та візуалізації даних з периферійних пристроїв було обрано сучасний стек технологій, що базується на використанні React для побудови клієнтської частини інтерфейсу користувача та Node.js для реалізації серверної логіки та обробки даних [14]. Такий вибір зумовлений потребою у створенні гнучкої, масштабованої та інтерактивної системи, здатної до обробки даних у режимі реального часу з одночасним забезпеченням зручного візуального середовища для користувача.

React є однією з найпопулярніших бібліотек для розробки фронтенду, яка дозволяє реалізувати динамічний інтерфейс із високим ступенем інтерактивності [15]. Його компонентна структура забезпечує зручну організацію коду, повторне використання елементів та ефективне оновлення інтерфейсу без повного перезавантаження сторінки [16]. В контексті реалізації системи візуалізації це дозволяє оперативно оновлювати графіки, індикатори, панелі стану та інші візуальні елементи у відповідь на зміну даних, що надходять з периферійних пристроїв. Крім того, React має велику кількість додаткових бібліотек (наприклад, для побудови графіків, маршрутизації, стилізації), що дає змогу значно прискорити розробку та забезпечити високий рівень користувацького досвіду [17].

Node.js, у свою чергу, обрано як серверне середовище виконання через його здатність ефективно обробляти великі обсяги асинхронних операцій, які притаманні системам збору даних [18]. Його неблокуюча архітектура дозволяє обробляти запити з периферійних пристроїв у реальному часі без зниження продуктивності, що є критично важливим у системах, де постійно надходять сигнали, які потрібно зчитувати, обробляти та передавати далі. Завдяки великій екосистемі npm-модулів, було реалізовано зручну структуру взаємодії з

фізичними пристроями (через серійні порти, TCP/UDP-з’єднання тощо), а також організовано API для передачі оброблених даних на фронтенд [19].

Крім того, перевагою обраної зв’язки є використання однієї мови програмування — JavaScript — на обох рівнях системи, що значно спрощує розробку, налагодження та супровід проєкту [20]. Такий підхід дозволяє уникнути дублювання логіки, зменшити технічний борг та підтримувати єдиний стиль написання коду в усіх компонентах програмного забезпечення.

У результаті вибір React і Node.js дозволяє створити масштабовану, стабільну і легко модифіковану систему, що відповідає вимогам сучасної розробки, при цьому забезпечуючи високу продуктивність, зручність у використанні та адаптивність до зміни зовнішніх умов або підключених пристроїв.

2.3 Вибір та обгрунтування бази даних

Для збереження та подальшого аналізу даних, що надходять із периферійних пристроїв, у межах даного проєкту було обрано MongoDB — документно-орієнтовану нереляційну базу даних, яка оптимально підходить для роботи з динамічно структурованими даними та високочастотними потоками інформації. Основною причиною такого вибору стала її природна сумісність із сучасними веб-технологіями, зокрема з Node.js, а також гнучка структура зберігання, яка не потребує жорсткого визначення схем, як у випадку класичних реляційних систем.

Однією з ключових переваг MongoDB є використання формату BSON (Binary JSON), який дозволяє зберігати складні вкладені структури даних, що особливо зручно у випадках, коли кожен запис має різну структуру або містить додаткові метадані. У контексті системи збору даних з периферійних пристроїв це означає, що кожен сигнал, замір або повідомлення може бути збережений у

зручному вигляді без необхідності попереднього нормалізування або створення допоміжних таблиць.

Також MongoDB має вбудовані можливості горизонтального масштабування, що дозволяє легко адаптувати систему до збільшення обсягу даних без необхідності кардинального перепроектування архітектури. Завдяки асинхронному доступу до бази та підтримці індексації по різних полях, забезпечується швидкий доступ до історичних записів, фільтрація за часовими мітками, значеннями сенсорів або іншими ключовими параметрами. Це дає змогу реалізувати гнучкі механізми запитів і ефективно формувати звіти або графіки на основі накопичених даних.

З технічного боку, MongoDB також добре інтегрується з інструментами екосистеми Node.js. Використання бібліотеки Mongoose як ORM-рішення дозволяє організувати структуру моделі, додати валідацію, обробку подій та взаємодію з даними у зручному для розробника вигляді. Це особливо корисно при побудові серверної частини, яка одночасно приймає потік даних, обробляє запити від клієнтської частини та взаємодіє з базою в режимі реального часу.

Таким чином, використання MongoDB як основного засобу зберігання інформації обумовлено її гнучкістю, продуктивністю, високим рівнем сумісності з обраним програмним стеком та здатністю ефективно працювати в умовах динамічного середовища з нерегламентованою структурою вхідних даних. Обрана база даних відповідає усім критеріям, необхідним для успішної реалізації функціональної частини програмного забезпечення в межах поставленого завдання.

Структура бази даних представлена на рисунку 2.2.

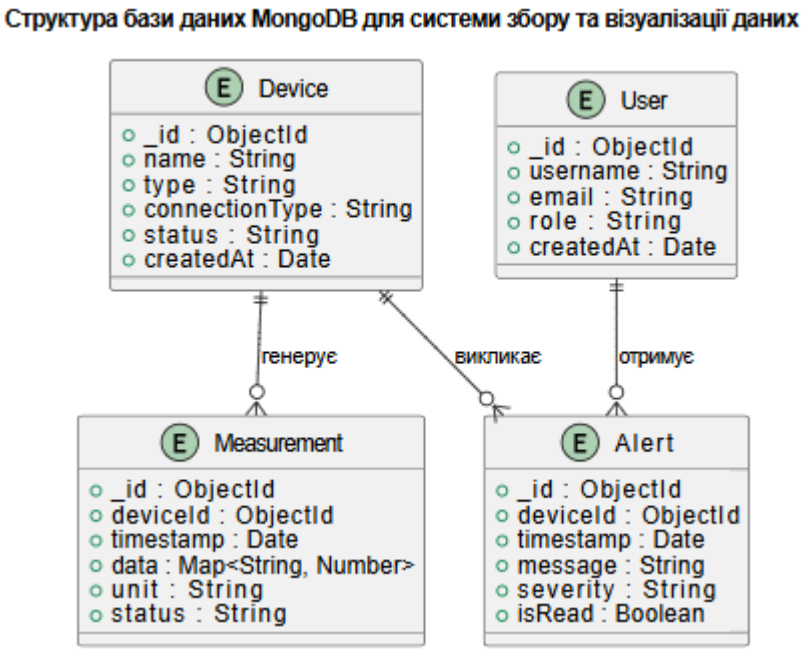

Рисунок 2.2 – Структура бази даних системи

Колекція Device виконує роль базового елементу структури даних і слугує для збереження інформації про всі периферійні пристрої, що беруть участь у зборі даних у системі. Кожен документ у цій колекції представляє окремий фізичний пристрій або логічний канал отримання інформації, який реєструється в системі з метою подальшого моніторингу, візуалізації та зв'язку з іншими сутностями бази даних, зокрема з колекціями Measurement та Alert.

У структурі документа передбачено унікальний ідентифікатор _id, який автоматично генерується системою MongoDB для забезпечення унікальності запису. Поле name містить назву пристрою, призначену для зручності ідентифікації в інтерфейсі користувача. Поле type визначає категорію або специфікацію пристрою (наприклад, "температурний сенсор", "вологомір", "датчик тиску"), що дозволяє системі адаптувати формат поданих даних. Атрибут connectionType вказує спосіб фізичного або логічного підключення пристрою до системи — наприклад, "USB", "UART", "Wi-Fi", "MQTT" тощо.

Поле status відображає поточний стан пристрою і може містити значення на кшталт "active", "offline", "error", що дає змогу оперативно реагувати на зміни в працездатності периферії. Нарешті, createdAt — це мітка часу, яка фіксує

момент реєстрації пристрою в системі, що є корисним для ведення журналу подій, історії та адміністративного аудиту.

Загалом, структура колекції Device є достатньо простою для швидкої реєстрації нового пристрою, але водночас гнучкою для подальшого масштабування — при потребі вона може бути доповнена додатковими параметрами, такими як місце розташування пристрою, калібрувальні значення або специфікації датчика, без необхідності зміни схеми всієї бази.

Колекція Measurement у структурі бази даних виступає центральним логічним елементом, відповідальним за зберігання всіх отриманих вимірювань із периферійних пристроїв. Кожен документ у цій колекції містить в собі не лише числові значення, а й усю необхідну метаінформацію для коректного аналізу, фільтрації, інтерпретації та подальшого відображення даних у візуальній формі. У контексті обраної архітектури система орієнтується на гнучке зберігання інформації, тому структура кожного запису передбачає використання поліморфної структури поля data, що дозволяє зберігати як одновимірні, так і багатовимірні значення з різних сенсорів. Всі вимірювання прив’язані до конкретного пристрою через унікальний ідентифікатор, що дає змогу здійснювати запити з урахуванням джерела даних, а також виконувати угрупування записів за типами або географічною прив’язкою. Час надходження фіксується у полі timestamp, що забезпечує коректне сортування, фільтрацію за часовими інтервалами та можливість побудови графіків динаміки значень. Додатково, для збереження контексту інтерпретації, передбачено поле unit, яке дозволяє визначити одиниці вимірювання та забезпечити правильне форматування при візуалізації. Усі документи можуть супроводжуватися службовим статусом, що відображає технічний або логічний стан вимірювання, зокрема інформацію про валідність, позаштатні ситуації чи діагностичні повідомлення. Завдяки гнучкій структурі та підтримці вкладених об’єктів MongoDB, таблиця Measurement не обмежується жорсткою схемою, але при цьому зберігає логічну цілісність і може бути ефективно використана в запитах,

агрегаціях та для подальшої аналітичної обробки у межах клієнтської частини системи.

Колекція User відіграє допоміжну, проте структурно важливу роль у програмному забезпеченні, оскільки забезпечує облік ідентифікованих суб'єктів взаємодії з системою та формує основу для реалізації базових механізмів контролю доступу, персоналізації та аудиту подій. Зберігання даних про користувачів у вигляді окремих документів дозволяє забезпечити централізовану реєстрацію подій, фільтрацію доступу до обробленої інформації та створення логічного контексту для відображення даних, пов'язаних із конкретними діями.

Кожен документ у колекції User включає унікальний ідентифікатор (_id), який автоматично генерується системою, та низку полів, необхідних для авторизації, автентифікації та визначення рівня доступу. Зокрема, поле username слугує для зручної ідентифікації користувача у межах інтерфейсу та може використовуватися як ключ під час входу в систему. Поле email містить зареєстровану адресу електронної пошти, яка, окрім ідентифікаційної функції, може бути використана для нотифікацій або відновлення доступу. Значення поля role визначає права користувача в системі — наприклад, оператор, адміністратор або гість — і дозволяє контролювати доступ до критичних операцій, зміни налаштувань або перегляду даних.

Також у структурі документа передбачено поле createdAt, яке фіксує дату та час створення облікового запису. Це забезпечує можливість відстеження хронології реєстрації та може бути використано при аналізі активності користувачів, побудові звітів або в рамках розширеної функції аудиту. При необхідності структура User може бути доповнена іншими параметрами, такими як токени автентифікації, історія входів або персональні налаштування інтерфейсу, однак базовий склад полів уже дозволяє ефективно управляти доступом і забезпечити зв'язок з іншими сутностями системи, такими як повідомлення, сповіщення чи збережені профілі налаштувань.

Загалом, колекція User створює основу для реалізації багатокористувацької взаємодії в межах програмної системи та дозволяє

формувати логічну ідентичність для кожного суб’єкта, що здійснює запити, переглядає дані або керує параметрами системи.

У структурі бази даних особливе місце займає колекція Alert, яка виконує функцію фіксації критичних або нестандартних подій, пов’язаних із роботою підключених периферійних пристроїв. Її призначення полягає в забезпеченні оперативного інформування користувача або системного модуля про потенційні збої, порушення нормальних параметрів або нестабільну поведінку окремих компонентів. Вона виконує роль цифрового дзвону тривоги, що фіксує — *щось пішло не так*, і тепер це офіційно задокументовано.

Кожен запис у колекції Alert містить обов’язковий унікальний ідентифікатор, поле deviceId, яке встановлює зв’язок з конкретним пристроєм, що став джерелом події, а також мітку часу timestamp, яка дозволяє точно відслідкувати момент виникнення інциденту. Це критично важливо при аналізі логів, побудові звітності або візуалізації історії змін у системі.

Поле message виконує функцію короткого текстового опису суті події. Його вміст може бути сформовано автоматично — наприклад, “Температура перевищила 70°C”, або вручну, якщо передбачено втручання користувача чи адміністратора. Це поле має бути достатньо інформативним для того, щоб користувач міг швидко зрозуміти характер проблеми без необхідності переглядати необроблені значення сенсорів.

Поле severity визначає рівень критичності сповіщення. Його значення можуть мати умовну класифікацію на кшталт “info”, “warning”, “critical”, що дозволяє сортувати або фільтрувати події у візуальному інтерфейсі. Такий підхід дозволяє не лише відображати тривожні повідомлення в порядку важливості, а й адаптувати реакцію системи — від простого запису в лог до активації аварійного сценарію або надсилання повідомлення адміністратору.

Окрему роль відіграє поле isRead, що має булевий тип і фіксує, чи було повідомлення переглянуто користувачем. Це дозволяє організувати інтерфейс сповіщень, в якому не лише видно, що саме сталося, але й які з подій залишаються без уваги. Реалізація цієї функції дозволяє покращити контроль над

станом системи, зменшити ризик пропущених критичних подій, а також підвищити прозорість журналу системної активності.

Таким чином, колекція Alert не лише виконує роль звичайного журналу подій, а є повноцінним інформаційним механізмом взаємодії між системою та користувачем. Вона забезпечує своєчасне виявлення нестабільної роботи, формує підстави для подальшого аналізу, а також дозволяє реалізувати реактивну поведінку системи відповідно до заданих правил. Структура даної колекції є простою, але логічно завершеною, що робить її придатною як для поточного моніторингу, так і для довгострокового аудиту подій.

2.4 Алгоритми роботи системи

Алгоритм агрегації даних представлено на рисунку 2.3.

Рисунок 2.3 – Алгоритм агрегації даних

Агрегація даних у межах даної програмної системи реалізується як окремий логічний процес, покликаний зменшити обсяг необробленої інформації, яка накопичується з часом, та надати можливість ефективної побудови аналітичних візуалізацій. Оскільки система працює з постійним потоком вхідних значень, збережених у базі даних з точністю до мілісекунди, виникає необхідність перетворення цих сирих записів на більш узагальнені, придатні для відображення у вигляді графіків або звітів.

Основна ідея алгоритму полягає у групуванні даних за фіксованими часовими інтервалами, наприклад, по одній хвилині, годині або добі, залежно від типу аналізу. У межах кожного інтервалу виконується обчислення агрегованого значення, зокрема середнього, мінімального, максимального або медіанного — залежно від задачі. Такий підхід дозволяє не лише зменшити кількість точок для візуалізації, а й виділити загальні тренди без втрати критичного змісту.

Алгоритм працює автономно, без участі користувача, заздалегідь визначаючи межі часових інтервалів відповідно до параметрів. Після вибірки записів з бази, що відповідають цим межам, виконується групування по часових мітках. Для кожної групи обчислюється відповідна агрегована метрика, а результат записується до окремої колекції — із зазначенням інтервалу, типу агрегату та вихідних параметрів. Цей процес може повторюватись циклічно, як за запитом користувача, так і за допомогою системного планувальника.

Такий підхід дозволяє розділити систему на два рівні роботи з даними: первинний, де зберігається повний потік, і аналітичний, де дані спрощено до рівня, зручного для вивчення. У цьому сенсі агрегатор виступає як фільтр, що відсіює шум і залишає суть. Його реалізація не лише оптимізує навантаження на систему, а й закладає основу для глибшого аналітичного модулю — від прогнозування до порівняння змін у динаміці.

Алгоритм фільтрації аномалій представлено на рисунку 2.4.

Рисунок 2.4 – Алгоритм фільтрації аномалій

У межах програмної частини системи реалізовано алгоритм фільтрації аномальних значень, який виконує функцію автоматичного виявлення потенційно некоректних або підозрілих даних у потоці. Його основна мета — відокремити показники, що не відповідають очікуваним статистичним характеристикам, знизити ризик спотворення графіків, неправильних висновків і взагалі — зберегти гідність користувацького інтерфейсу.

Алгоритм працює в реальному часі та застосовується до кожного нового запису, що надходить у систему. При надходженні нового значення воно порівнюється з динамічно обчислюваним середнім значенням, отриманим на основі останніх N збережених записів. У разі, якщо відхилення перевищує визначений допустимий поріг (який може задаватися вручну або адаптуватися автоматично), значення класифікується як аномальне.

Подальша поведінка системи залежить від обраної політики: аномальні значення можуть бути або виключені з подальшої обробки, або відзначені тегом

для подальшого аналізу. При цьому зберігається їхній оригінальний вигляд, а інформація про виявлену аномалію фіксується у вигляді окремого повідомлення або запису з додатковими метаданими: величина відхилення, час виявлення, статус "очікує перевірки" тощо. Це дозволяє не лише автоматично очищати потік даних, але й зберігати повну інформацію для можливого аудиту або ручної верифікації.

Окрему увагу приділено гнучкості: алгоритм дозволяє налаштувати кількість записів, що використовуються для розрахунку статистики, змінити порогові значення або повністю відключити фільтрацію у разі потреби. Це дає змогу використовувати його як у високоточних системах, де кожен відхилений запис — це втрата даних, так і в більш гнучких сценаріях, де важливіше уникнути візуального шуму або хибних спрацювань.

Таким чином, алгоритм фільтрації аномалій виконує не лише роль «цифрового санітара» потоку даних, а й формується як розумна логічна надбудова, що підвищує довіру до всієї системи загалом. Він дозволяє підтримувати якість даних, зменшує навантаження на інтерфейс і забезпечує базовий рівень аналітичної поведінки, який відрізняє справжню програмну систему від простої трансляції сигналів.

Алгоритм керування статусом алертів представлено на рисунку 2.5.

Рисунок 2.5 – Алгоритм керування статусом алертів

Керування статусом алертів у системі є не стільки технічною задачею, скільки механізмом контролю інформаційного порядку у світі, де потік повідомлень — нескінченний, а користувач — забудькуватий. Це процес, що забезпечує осмислений життєвий цикл повідомлень: від моменту, коли система

панікує й створює запис у базі, до миті, коли користувач нарешті зволить прочитати його, натиснувши кудись у UI.

На рівні логіки, алгоритм реалізує ітераційне проходження по колекції актуальних алертів, кожен з яких перевіряється на предмет ознаки isRead. Якщо значення цього поля дорівнює false, повідомлення виводиться у графічному інтерфейсі (використовуючи якийсь елегантний, бажано темний, компонент), після чого негайно позначається як прочитане. Це не тільки прибирає його зі списку “нових”, а й символічно визнає: “Ми бачили цю загрозу. Ми її обробили. Ми не ігноруємо свій код”.

Алерти, які вже мають статус isRead = true, не підлягають повторному відображенню. Вони зберігаються в базі як частина історії — як дим на кухні після того, як спрацювала сигналізація. У реальному світі вони вже не викликають дій, але залишаються доступними в історії на випадок постапокаліптичного аудиту або дуже прискіпливого викладача.

3 РЕАЛІЗАЦІЯ КОМП'ЮТЕРНОЇ СИСТЕМИ

3.1 Реалізація збору та опрацювання даних

У рамках реалізації системи основну увагу було зосереджено на програмному процесі збору та подальшого опрацювання даних, що надходять до серверної частини застосунку. Весь процес реалізовано на платформі Node.js, із використанням асинхронної моделі обробки та мінімізації блокуючих викликів. Це дало змогу досягти стабільної роботи навіть у випадках надходження великої кількості даних за короткий проміжок часу.

Дані надходять у вигляді HTTP-запитів або через WebSocket, залежно від конфігурації клієнта. Для обробки вхідного потоку на сервері створено окремий модуль, що відповідає за прийом, первинну перевірку, парсинг та запис у базу даних. Нижче представлено лістинг 3.1, що реалізує обробку POST-запиту з даними.

Лістинг 3.1 – Обробка POST-запиту.

```
const Measurement = require('../models/Measurement');
async function handleIncomingData(req, res) {
try {
const { deviceId, value, unit } = req.body;
if (!deviceId || value === undefined) {
return res.status(400).json({ error: 'Invalid payload' });
}
const measurement = new Measurement({
deviceId,
timestamp: new Date(),
data: { value },
unit,
status: 'ok',
});
```

```
await measurement.save();
res.status(201).json({ message: 'Data stored successfully' });
} catch (err) {
console.error('[Data Error]', err);
res.status(500).json({ error: 'Internal server error' });
}
}
```

Після надходження дані проходять перевірку на відповідність мінімальним вимогам (структура, наявність ключових полів). У разі успішної перевірки — створюється новий запис типу Measurement та зберігається до MongoDB. Для підвищення надійності система логування фіксує кожну подію, яка не відповідає базовій структурі, і надає розробнику повний контекст для налагодження.

Окремо реалізовано модуль, який здійснює обробку даних у фоновому режимі. Зокрема, при збереженні кожного нового запису автоматично перевіряється, чи потрапляє значення у допустимий діапазон. Якщо воно виходить за межі — створюється відповідне повідомлення типу Alert (лістинг 3.2).

Лістинг 3.2 – Обробка даних.

```
const Alert = require('../models/Alert');
async function validateMeasurement(measurement) {
  const threshold = 70; // допустима межа
  const actualValue = measurement.data.value;
  if (actualValue > threshold) {
    const alert = new Alert({
      deviceId: measurement.deviceId,
      timestamp: new Date(),
      message: `Value exceeded: ${actualValue}`,
      severity: 'warning',
      isRead: false,
    });
    await alert.save();
  }
}
```

Цей механізм дозволяє автоматизувати контроль за даними без необхідності ручної перевірки або складної логіки на клієнтському боці. Усі опрацьовані записи одразу відображаються у користувацькому інтерфейсі через WebSocket-з’єднання, що забезпечує оновлення у реальному часі без додаткових запитів до сервера.

Для ефективної роботи системи також реалізовано механізм періодичного очищення тимчасових буферів, індексацію за часовими мітками та оптимізацію запитів до бази даних. Усе це дозволяє зберігати високу продуктивність системи навіть при зростанні кількості джерел даних.

3.2 Реалізація алгоритмів аналізу та опрацювання даних

Процес аналізу та опрацювання даних у розробленій системі реалізовано як послідовність модулів, що виконують низку логічно взаємопов’язаних операцій — від отримання сирих значень до фільтрації, нормалізації та, за потреби, створення супутніх записів (наприклад, алертів або логів). В основі лежить подієво-орієнтований підхід, при якому кожна нова порція даних ініціює каскад обробки в серверній частині системи.

На першому етапі обробки отримані дані (зазвичай у вигляді JSON-структури) проходять перевірку на коректність — відсутність обов’язкових ключів, перевірка типів, значень, наявність дублювань (лістинг 3.3).

Лістинг 3.3 – Перевірка на коректність.

```
function validateIncomingData(data) {
  if (!data || typeof data !== 'object') return false;
  return data.hasOwnProperty('value') && typeof data.value === 'number';
}
```

Це найпростіший, але обов’язковий фільтр. Він дозволяє не запускати решту обробки на випадковому тексті, який приїхав через WebSocket.

Наступним етапом є фільтрація аномалій. На основі визначеного допустимого діапазону значень або простого статистичного аналізу останніх N значень виконується оцінка відхилення. Якщо значення виходить за межі, воно або відсікається, або маркується як потенційна аномалія (лістинг 3.4).

Лістинг 3.4 – Фільтрація аномалій.

```
function isAnomalous(value, average, threshold = 0.3) {
  return Math.abs(value - average) / average > threshold;
}
```

Після фільтрації нормальні значення зберігаються у базі даних у структурованому вигляді разом із міткою часу, аномальні — передаються в систему алертів. Усе це відбувається асинхронно, щоб не блокувати основний цикл прийому даних (лістинг 3.5).

Лістинг 3.5 – Асинхронне збереження даних.

```
const measurement = new MeasurementModel({
  deviceId: data.deviceId,
  value: data.value,
  timestamp: new Date(),
  isAnomaly: isAnomalous(data.value, avg)
});
await measurement.save();
```

І нарешті, після збереження, дані транслюються на клієнт за допомогою WebSocket або іншого push-каналу, щоб оновити графіки та індикатори в інтерфейсі без затримок (лістинг 3.6).

Лістинг 3.6 – Асинхронне збереження даних

```
io.emit('data:update', {
  value: data.value,
  timestamp: measurement.timestamp,
  isAnomaly: measurement.isAnomaly
});
```

Таким чином, реалізований процес аналізу та опрацювання даних складається з чітко виокремлених етапів: валідація → аналіз → збереження → візуалізація, що дозволяє зберігати стабільність роботи системи, мінімізувати ризики накопичення помилкових даних та забезпечити своєчасну реакцію на критичні ситуації.

3.3 Реалізація візуалізації даних

Реалізація візуалізації даних у межах програмної частини системи передбачає побудову інтерфейсу, здатного оперативно відображати значення, що надходять з бази даних або у реальному часі через серверний канал. Основна мета цього етапу — трансформувати числові або текстові показники в зрозумілу, наочну й адаптивну форму, яку користувач здатен інтерпретувати за лічені секунди.

На клієнтській частині візуалізація реалізована з використанням бібліотеки Recharts — компонентної бібліотеки для React, яка дозволяє легко будувати інтерактивні графіки з гнучкими параметрами налаштування (лістинг 3.7).

Лістинг 3.7 – Побудова графіку

```
<LineChart width={800} height={400} data={sensorData}>
  <CartesianGrid stroke="#ccc" />
  <XAxis dataKey="timestamp" />
  <YAxis />
  <Tooltip />
  <Line type="monotone" dataKey="value" stroke="#82ca9d" dot={false} />
</LineChart>
```

У наведеному прикладі sensorData — це масив об’єктів, який формується з бази даних або отримується через WebSocket-з’єднання (лістинг 3.8).

Лістинг 3.8 – SensorData.

```
[
  { timestamp: '2025-04-04T12:00:00Z', value: 42 },
  { timestamp: '2025-04-04T12:01:00Z', value: 43 },
  ...
]
```

Дані автоматично оновлюються завдяки реалізації підписки на серверні події, що дозволяє досягти ефекту роботи в режимі реального часу. На серверній частині використовується WebSocket (через Socket.IO), який транслює нові значення клієнту одразу після їх появи (лістинг 3.9).

Лістинг 3.9 – Обробка події на клієнті

```
useEffect(() => {
  socket.on('new-data', (data) => {
    setSensorData(prev => [...prev.slice(-99), data]); // обмеження до 100 останніх записів
  });
}, []);
```

Цей механізм дозволяє не лише виводити графік, що автоматично оновлюється, але й реалізувати м’яку анімацію, яка створює відчуття живої, динамічної системи. Графіки можуть супроводжуватися цифровими індикаторами, що виводять останнє значення (лістинг 3.10).

Лістинг 3.10 – Цифрові індикатори.

```
<h2>Поточне значення: {sensorData.at(-1)?.value ?? '—'}</h2>
```

Інтерфейс також передбачає можливість перемикання діапазонів відображення: останні 10 хвилин, година, доба, а також підтримує режим паузи, коли оновлення призупиняється для аналізу вже отриманих даних. Це реалізується через просте фільтрування перед візуалізацією (лістинг 3.11).

Лістинг 3.11 – Фільтрування перед візуалізацією

```
const visibleData = sensorData.filter(d => withinTimeRange(d.timestamp, selectedRange));
```

Таким чином, візуалізація даних реалізована як декларативний, реактивний процес, у якому інтерфейс динамічно реагує на зміни стану без

необхідності ручного оновлення. Це не лише покращує UX, а й дозволяє зменшити кількість потенційних помилок у відображенні, оскільки всі оновлення централізовано обробляються через state-менеджмент React.

4 ТЕСТУВАННЯ КОМП'ЮТЕРНОЇ СИСТЕМИ

4.1 Опис інтерфейсу користувача

Інтерфейс користувача є одним із ключових компонентів програмного забезпечення, оскільки саме через нього здійснюється доступ до всіх основних функцій системи. Його побудовано за принципами односторінкового вебзастосунку, що дозволяє забезпечити високу швидкість взаємодії, відсутність повного перезавантаження сторінок і зручність навігації.

Загальна структура інтерфейсу складається з кількох логічних секцій. Центральне місце займає панель візуалізації, яка виконує функцію основного інформаційного простору. У ній у режимі реального часу відображаються графіки, індикатори та цифрові значення, що дозволяє оперативно відслідковувати зміну параметрів, отриманих у ході обробки вхідних даних. Користувач має змогу перемикатися між часовими діапазонами, фільтрувати інформацію та адаптувати вигляд графічного подання під власні потреби.

Окремим елементом інтерфейсу є панель керування, яка містить навігаційні елементи для переходу між розділами програми, зокрема такими як "Живі дані", "Історія", "Алерти" та "Налаштування". Взаємодія з кожним розділом здійснюється без виходу з поточного сеансу, що сприяє безперервності користувацького досвіду.

Розділ повідомлень (алертів) виконує функцію інформаційного контролю стану системи. У ньому відображаються сповіщення про критичні або нестандартні події, які можуть вплинути на роботу програми. Непрочитані алерти позначаються окремими маркерами та фіксуються до моменту взаємодії користувача. Це дозволяє уникнути втрати важливої інформації внаслідок неуважності або надмірного навантаження.

З точки зору дизайну інтерфейс витримано у мінімалістичному стилі з акцентом на функціональність, контрастність та зручність сприйняття.

Кольорова схема нейтральна, із чіткими акцентами для ключових елементів — кнопок, попереджень, навігаційних блоків. Усі компоненти адаптивні, що дозволяє використовувати інтерфейс на різних типах пристроїв, включно з мобільними.

Загалом інтерфейс користувача розроблений як цілісна система, що поєднує в собі візуальну інформативність, логічну впорядкованість і технічну гнучкість. Його структура дозволяє забезпечити не лише зручність взаємодії, а й точність, ефективність та своєчасність прийняття рішень на основі виведених даних.

4.2 Тестування функціоналу

Тестування виведення сповіщення з пристроїв представлено на рисунку 4.1.

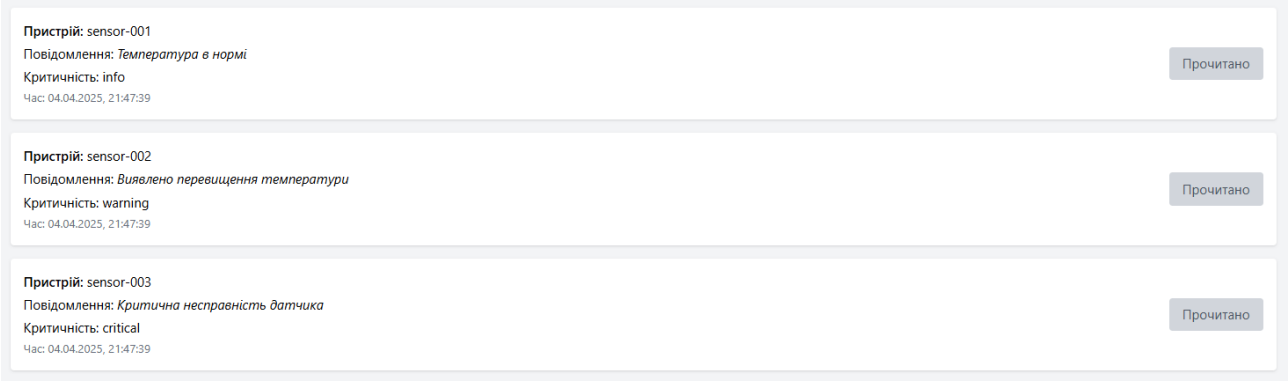

Рисунок 4.1 – Виведені сповіщення з пристроїв

У процесі тестування системи моделюється ситуація, коли певні сенсори або пристрої надсилають сповіщення (алерти) до центрального сервера, і ці алерти відображаються у вебінтерфейсі в реальному часі. Спочатку, при запуску сервера (Node.js + Express) та клієнтської частини (React) на різних портах, у базі даних (MongoDB) з’являється початковий (seed) набір даних — наприклад, один інформаційний алерт, один з попередженням та один критичний. Це дозволяє

переконатися, що сама система коректно зберігає та віддає алерти, а також що клієнтський інтерфейс уміє їх отримувати і правильно виводити. У браузері відкривається сторінка з простим списком цих сповіщень: на ній заголовком вказано «Останні Алерти», під яким кожен алерт з’являється у вигляді карточки з назвою пристрою, повідомленням, рівнем критичності та часовою міткою. Для кожного елемента є кнопка «Позначити як прочитане», натискання якої оновлює стан сповіщення у базі, і на екрані алерт отримує індикацію, що він уже неактивний (наприклад, кнопка стає сірою й недоступною). Паралельно у правій або нижній частині сторінки (залежно від макета) відображається лінійний графік, який отримує (або імітує) поточні значення з пристроїв — зазвичай це можуть бути дані про температуру, вологість чи будь-який інший показник. У ході тестування додаються нові алерти через REST-запит: наприклад, якщо датчик фіксує відхилення вищі за норму, то в тестовому режимі вноситься запис із рівнем критичності «warning» або «critical» (через POST-запит до /api/alerts). У результаті, на інтерфейсі майже миттєво (залежить від частоти оновлення) з’являється новий пункт зі свіжим повідомленням, що дає змогу користувачеві одразу оцінити рівень загрози й виконати дії з підтвердження чи ігнорування. Якщо ж у процесі тесту виникає помилка на сервері чи проблеми із підключенням до бази, інтерфейс зазвичай відображає або порожній список, або повідомлення про помилку в консолі браузера, однак за нормальних умов система впевнено приймає нові алерти, оновлює статуси вже наявних і синхронізує ці зміни у базі без перезавантаження сторінки. Таким чином, процес тестування підтверджує, що функціонал обробки, збереження й відображення алертів працює належним чином, а також демонструє динамічну побудову графіка для реальних або тестових показників.

Тестування візуалізації даних з пристроїв представлено на рисунку 4.2.

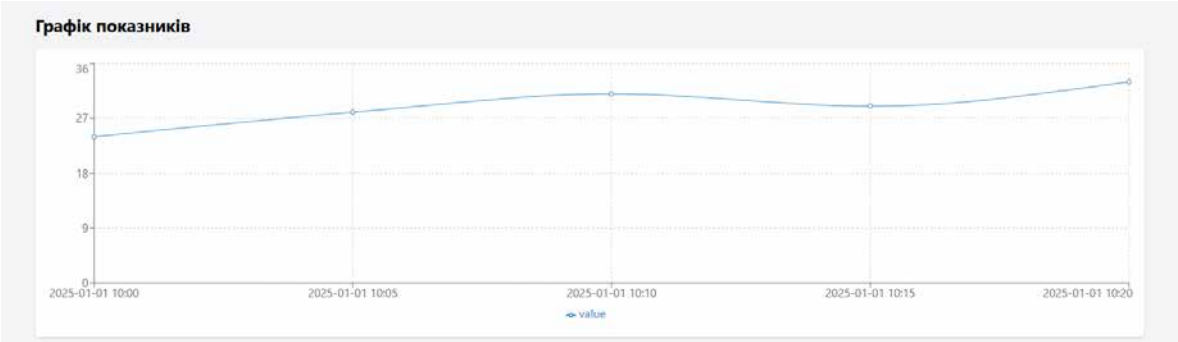

Рисунок 4.2 – Візуалізація показів з пристроїв

У типовому сценарії використання система відстежує набір пристроїв (наприклад, сенсорів), що надсилають до сервера дані про стан чи події у вигляді алертів і числових показників. Після запуску застосунку, вебінтерфейс відображає на екрані список останніх алертів та побудований на основі реальних або тестових даних графік (лінійний чи інший тип, залежно від специфікації). При відкритті вебсторінки програма виконує запит до серверної частини, щоб отримати найактуальніші події й оновлені показники.

Поряд зі списком подій на сторінці присутній інтерактивний графік показників від сенсорів чи інших пристроїв. Цей графік побудовано з використанням бібліотеки Recharts, що дає змогу відображати часові ряди у вигляді лінійної діаграми. Вісь абсцис містить часові позначки (timestamp), а вісь ординат показує числові значення датчиків (наприклад, температуру, вологість або інші вимірювані параметри). Кожна нова порція даних, отриманих від сервера, автоматично додається до поточної вибірки і відображається на діаграмі, завдяки чому формується ефект реального часу (або майже реального, залежно від інтервалу опитування чи механізму оновлення). Графік дозволяє візуально оцінити тренди, виявити аномалії та швидко реагувати на критичні відхилення показників.

Таким чином, весь процес виглядає так, ніби у «динамічному» вікні відбувається поява червоних, жовтих чи синіх оповіщень (відповідно до заданих рівнів severity), а поруч можна переглянути числові тенденції: якщо температурний датчик перевищив встановлений поріг, користувач побачить графічний зріст значення і паралельне повідомлення у списку з відповідним рівнем критичності. Це надає можливість не лише читати повідомлення, а й

відразу аналізувати, як змінювалися вимірювання перед чи після алерта. Водночас інтерфейс лишається інтерактивним і адаптивним, тож на мобільних пристроях чи різних розмірах екранів усі елементи зберігають зручне розташування та читабельність.

4.3 Тестування адаптивності

Тестування адаптивності вебзастосунку починається з перевірки поведінки інтерфейсу на різних розмірах вікна та різних типах пристроїв. Спочатку за допомогою інструментів розробника (наприклад, Google Chrome DevTools) обираються кілька типових розширень екрана: смартфон (320–480 px), планшет (768–1024 px), ноутбук чи настільний монітор (1366 px і більше). Проводиться поступове зменшення та збільшення ширини вікна браузера, спостерігається, як елементи інтерфейсу змінюють своє розташування, а також як працюють точки перелому (breakpoints). Перевіряється відсутність горизонтальної прокрутки, збереження коректних відступів та читабельності тексту, а також правильне відображення динамічних компонентів, як-от списки, таблиці чи графіки. У той самий час звертається увага на функціонал кнопок, форм і випадаючих меню: чи легко та зручно взаємодіяти з ними при різній щільності пікселів та різних пропорціях екрана.

Після перевірки в симуляції через інструменти розробника додатково тестуються реальні пристрої, коли це можливо: відкривається вебзастосунок на смартфоні, планшеті чи ноутбуці, звертається увага на час завантаження, масштаби інтерфейсу та наскільки інтуїтивним лишається переміщення між розділами. Виконується поворот екрану (portrait/landscape), щоб упевнитися, що компоненти не розташовуються довільно чи не накладаються один на одного. Перевіряється відповідність кількості відображуваної інформації на екрані можливостям пристрою: все, що має бути видиме, повинно легко

прокручуватися та зберігати зручність взаємодії. Також за потреби виконується аудит за допомогою Lighthouse або схожих інструментів, щоб проаналізувати зручність перегляду й навігації з різних точок зору, включно з розміром шрифтів, контрастністю та структурою компонентів. Результати тестування дозволяють переконатися, що кінцевий користувач матиме коректний досвід використання незалежно від того, яке пристрій чи роздільну здатність він застосовує.

Вигляд інтерфейсу на пристрої Pixel 7 представлено на рисунку 4.3.

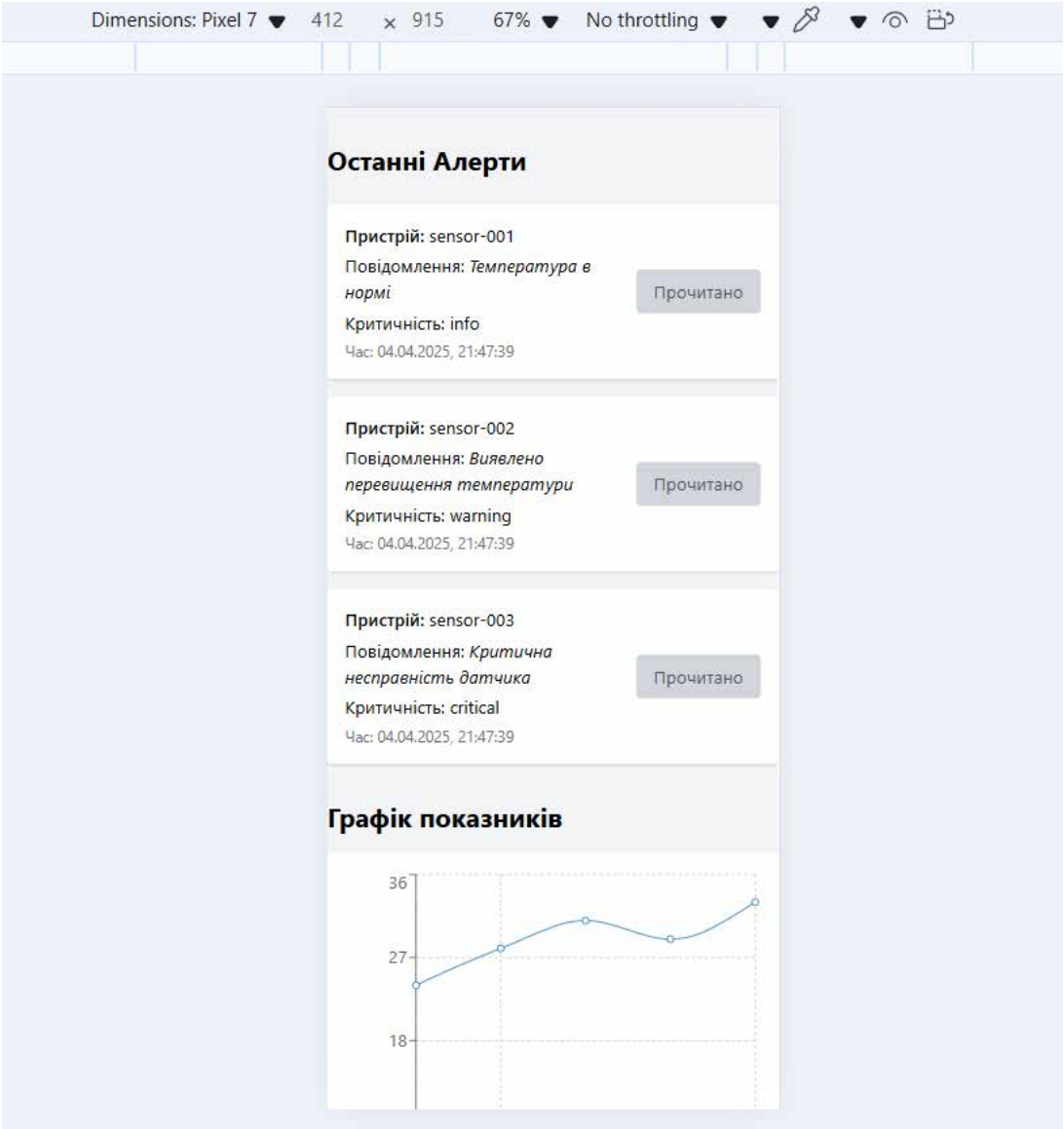

Рисунок 4.3 – Вигляд інтерфейсу на пристрої Pixel 7

На рисунку 4.3 представлено приклад візуального відображення клієнтського інтерфейсу системи на мобільному пристрої Google Pixel 7.

Система успішно адаптується до обмеженої ширини екрану, що підтверджує коректну реалізацію принципів респонсивного дизайну.

Інтерфейс автоматично перебудовується у вертикальний лінійний макет, у якому основні компоненти — список алертів і графік показників — розміщено один під одним. Кожен блок алерта представлений у вигляді окремої картки з чітким відділенням від сусідніх елементів, завдяки чому зберігається візуальна структурованість навіть на вузьких екранах.

Текстові поля мають відповідну адаптацію за розміром шрифту: заголовки відображаються жирним шрифтом для акценту, тоді як допоміжна інформація, зокрема дата і час події, — меншим розміром, але без втрати читабельності. Кнопка “Прочитано” також підлаштована під мобільне відображення: вона не виходить за межі контейнера, зберігає пропорційність і не перекриває текст.

В нижній частині розміщується графік показників, який автоматично масштабується до ширини екрану, зберігаючи достатню деталізацію та доступність для взаємодії. Графік не обрізається, а лінії побудови залишаються чіткими й зрозумілими завдяки динамічному рендерингу.

Таким чином, інтерфейс системи повністю адаптований для використання на сучасних мобільних пристроях. Адаптивність досягається завдяки використанню фреймворку Tailwind CSS, який забезпечує автоматичну адаптацію компонентів без потреби у створенні окремих мобільних версій. Це підвищує універсальність застосунку та гарантує зручність користувача незалежно від розміру екрану пристрою.

Вигляд інтерфейсу на пристрої IPad Mini представлено на рисунку 4.4.

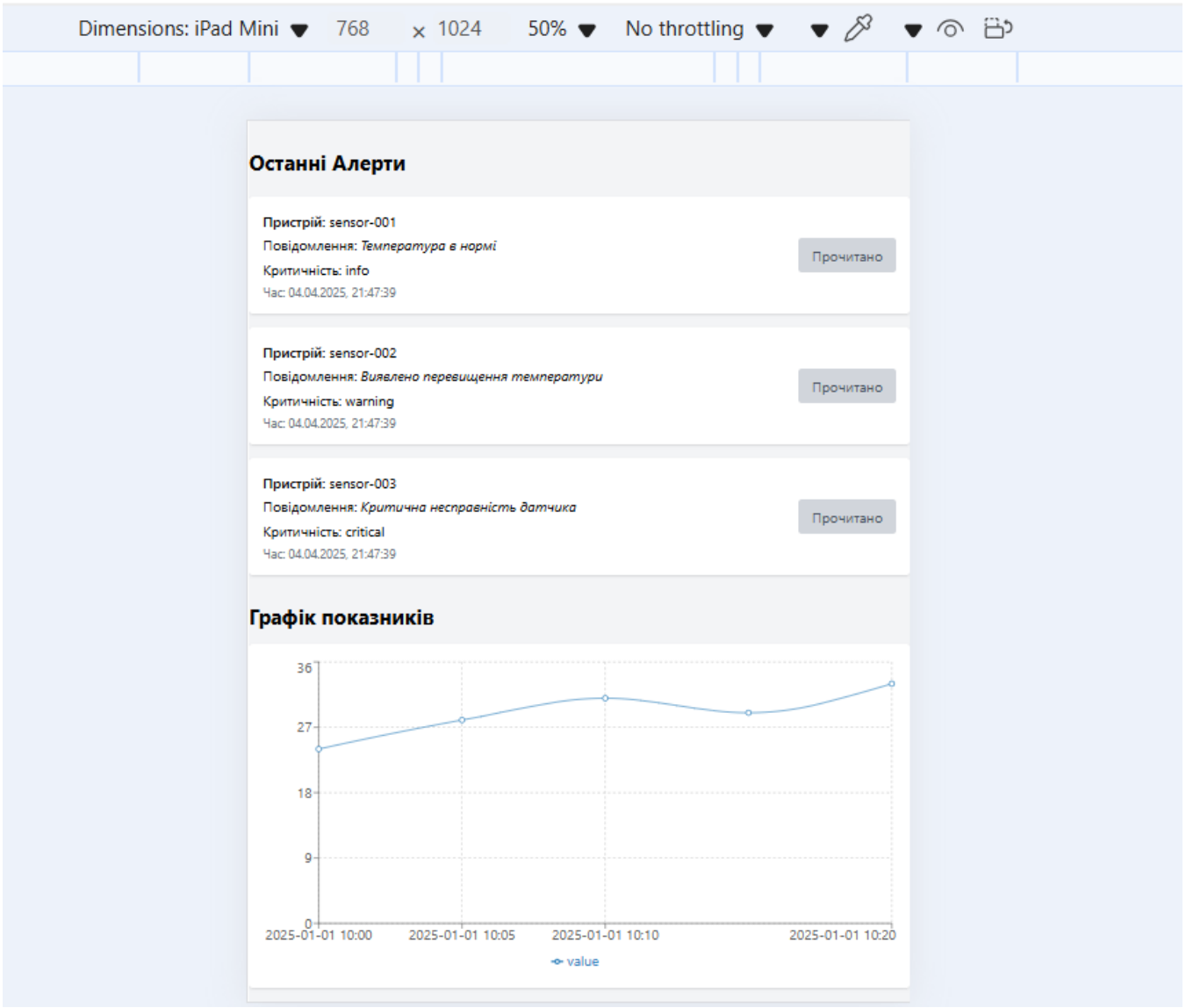

Рисунок 4.4 – Вигляд інтерфейсу на пристрої IPad Mini

На рисунку 4.4 наведено вигляд інтерфейсу на планшетному пристрої iPad Mini з роздільною здатністю 768×1024 пікселів. Інтерфейс коректно адаптується до розширеної ширини екрана порівняно з мобільним режимом, що підтверджує повну відповідність принципам адаптивного дизайну.

Завдяки середньому розміру дисплея компоненти інтерфейсу мають змогу розширюватися в горизонтальному напрямку, забезпечуючи кращу читабельність і розташування елементів. Кожна картка алерта використовує повну доступну ширину контейнера, що дозволяє уникнути візуального перевантаження і водночас зберігати логічну структуру інформації.

Кнопка “Прочитано” зберігає фіксовану позицію в межах кожної картки, не зміщується та залишається інтуїтивно зрозумілою для взаємодії. Всі текстові поля мають збалансовані відступи та розміри шрифтів, що покращує загальне візуальне сприйняття на планшетах.

Графік показників займає повну ширину доступного простору і адаптується до планшетного формату без спотворень. Він дозволяє відобразити більше проміжків часу або більше даних без необхідності горизонтального прокручування, що особливо зручно для перегляду трендів у вимірах.

Загалом, інтерфейс системи демонструє високий ступінь адаптивності до планшетного середовища. Завдяки гнучкому компонентному дизайну на базі Tailwind CSS, усі функціональні елементи залишаються однаково доступними та зручними у використанні як для мобільного, так і для планшетного користувача.

ВИСНОВКИ

У ході виконання роботи було здійснено повний цикл розробки програмної системи для обробки, збереження та візуалізації подій, що надходять від переферійних пристроїв. На першому етапі було проведено аналіз аналогічних рішень, а також сформульовано функціональні вимоги до системи, зокрема вимоги до інтерфейсу, структури бази даних та логіки взаємодії між компонентами.

Після цього було спроєктовано архітектуру клієнт-серверного застосунку, визначено основні інструментальні засоби, серед яких React для реалізації інтерфейсу користувача, Node.js і Express для побудови серверної логіки, а також MongoDB як документоорієнтована база даних для зберігання повідомлень про події (алертів) та вимірювань. Структура бази даних була адаптована під потреби системи та протестована із застосуванням тестових даних.

На етапі реалізації було розроблено інтерфейс користувача з урахуванням принципів адаптивності та зручності взаємодії. В інтерфейсі було реалізовано відображення алертів з можливістю позначення їх як прочитаних, а також вбудовано блок візуалізації у вигляді графіка, що динамічно відображає зміни показників. Було також перевірено вигляд інтерфейсу на різних пристроях, зокрема мобільних та планшетах, що засвідчило відповідність адаптивного дизайну практичним вимогам.

Серверна частина системи забезпечувала коректну роботу REST API, обробку запитів, збереження й модифікацію даних у базі. Було здійснено тестування окремих функцій та загальної взаємодії клієнта з сервером, що підтвердило стабільність і працездатність системи.

У результаті виконаної роботи було створено повноцінну програмну систему, що задовольняє заданим вимогам, має адаптивний і зручний інтерфейс, забезпечує візуалізацію даних та може бути розширена за потреби.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Мороз В. І. Комп’ютерна периферія: підручник / В. І. Мороз. – Київ : Ліра-К, 2016. – 256 с.

2. Волков П. М. Пристрої введення-виведення: теорія та практика : навч. посіб. / П. М. Волков. – Харків : Основа, 2017. – 312 с.

3. Власов О. М., Руденко Д. Д. Архітектура ПК. Переферійні пристрої : навч. посіб. / О. М. Власов, Д. Д. Руденко. – Одеса : Чорномор’я, 2018. – 224 с.

4. Таненбаум Е. С. Архітектура комп’ютера : [пер. з англ.] / Е. С. Таненбаум. – Київ : BHV, 2015. – 608 с.

5. Сміт Р. Низькорівневе програмування та робота з периферією : монографія / Р. Сміт. – Львів : ЛНУ, 2019. – 346 с.

6. Пономаренко В. Б. Основи комп'ютерної техніки та периферійних пристроїв : метод. рек. / В. Б. Пономаренко. – Дніпро : Вид-во ДНУ, 2020. – 158 с.

7. Петров Г. Апаратне забезпечення та інтеграція периферійних модулів : монографія / Г. Петров. – Київ : Наукова думка, 2021. – 289 с.

8. Семенюк І. В., Дмитрук Л. О. Синтез та аналіз зовнішніх пристроїв ЕОМ : навч. посіб. / І. В. Семенюк, Л. О. Дмитрук. – Тернопіль : ТНТУ, 2018. – 193 с.

9. ДСТУ ISO/IEC 2382:2015. Інформаційні технології — Словник термінів. – Київ : ДП «УкрНДНЦ», 2015. – 88 с.

10. Zubkov A., Krasnova O. Peripheral Interface Controllers in Embedded Systems [Electronic resource] / A. Zubkov, O. Krasnova // International Journal of Computer Engineering. – 2019. – Vol. 17, no. 3.

11. LabVIEW [Електронний ресурс]. – Режим доступу: https://www.ni.com/en/shop/labview.html?srsltid=AfmBOoqT0JQCFEBj4ByM1P8q92BzxQuk3aTmKiMLTBt8vZkTX6zSRpvb (дата звернення: 25.03.2025).

12. OpenHAB [Електронний ресурс]. – Режим доступу: https://www.openhab.org/ (дата звернення: 25.03.2025).

13. Processing [Електронний ресурс]. – Режим доступу: https://processing.org/ (дата звернення: 25.03.2025).14. MongoDB, Express, React & Node.js Full Project [Електронний ресурс]. – Режим доступу: https://www.youtube.com/watch?v=korRfKTDoxE (дата звернення: 28.03.2025).

15. MERN Part I: Building RESTful APIs with Node.js and Express [Електронний ресурс]. – Режим доступу: https://medium.com/weekly-webtips/building-restful-apis-with-node-js-and-express-a9f648219f5b (дата звернення: 28.03.2025).

16. React POST requests with Express/Node and MongoDB [Електронний ресурс]. – Режим доступу: https://stackoverflow.com/questions/50617351/react-post-requests-with-express-node-and-mongodb (дата звернення: 28.03.2025).

17. How To Use MERN Stack: A Complete Guide [Електронний ресурс]. – Режим доступу: https://www.mongodb.com/en-us/resources/languages/mern-stack-tutorial (дата звернення: 28.03.2025).

18. MERN stack tutorial: The Complete Guide with Examples [Електронний ресурс]. – Режим доступу: https://deadsimplechat.com/blog/mern-stack-the-complete-guide/ (дата звернення: 28.03.2025).

19. Building a Simple CRUD Application with MERN Stack [Електронний ресурс]. – Режим доступу: https://www.digitalocean.com/community/tutorials/build-a-to-do-application-using-docker-and-mern-stack (дата звернення: 28.03.2025).

20. Secure Your MERN Stack Application: Best Practices [Електронний ресурс]. – Режим доступу: https://www.mongodb.com/blog/post/secure-your-mern-stack-application-best-practices (дата звернення: 28.03.2025).

ДОДАТКИ

Додаток А «Лістинг програмного коду»

Лістинг А.1 – server/package.json

```
{
  "name": "server",
  "version": "1.0.0",
  "description": "Node.js сервер для обробки і зберігання
алертів",
  "main": "index.js",
  "scripts": {
    "start": "node index.js",
    "seed": "node scripts/seedAlerts.js"
  },
  "dependencies": {
    "cors": "^2.8.5",
    "express": "^4.18.2",
    "mongoose": "^6.6.5"
  }
}
```

Лістинг А.2 – server/index.js

```
// server/index.js
const express = require('express');
const mongoose = require('mongoose');
const cors = require('cors');
const alertsRouter = require('./routes/alerts');

const app = express();

// Дозвіл на використання JSON у запитах
app.use(express.json());

// Увімкнення CORS, щоб клієнт міг звертатися до сервера
app.use(cors());

// Підключення до MongoDB (замість 'alertsdb' за потреби можна
іншу назву БД)
mongoose.connect('mongodb://127.0.0.1:27017/alertsdb', {
  useNewUrlParser: true,
  useUnifiedTopology: true
```

```
}).then(() => {
  console.log('Успішне підключення до MongoDB');
}).catch((err) => {
  console.error('Помилка підключення до MongoDB:', err);
});

// Підключення маршрутизатора для /api/alerts
app.use('/api/alerts', alertsRouter);

// Запуск сервера на порту 3001 (якщо PORT не визначено через
змінні середовища)
const PORT = process.env.PORT || 3001;
app.listen(PORT, () => {
  console.log(`Сервер запущено на порту ${PORT}`);
});
```

Лістинг А.3 – server/models/Alert.js

```
// server/models/Alert.js
const mongoose = require('mongoose');

const alertSchema = new mongoose.Schema({
  deviceId: { type: String, required: true },
  message: { type: String, required: true },
  severity: { type: String, enum: ['info', 'warning', 'critical'],
required: true },
  timestamp: { type: Date, default: Date.now },
  isRead: { type: Boolean, default: false }
});

module.exports = mongoose.model('Alert', alertSchema);

server/routes/alerts.js

// server/routes/alerts.js
const express = require('express');
const router = express.Router();
const Alert = require('../models/Alert');

// Отримання останніх 100 алертів (найновіші — попереду)
router.get('/', async (req, res) => {
  try {
    const alerts = await Alert.find().sort({ timestamp: -1
}).limit(100);
    res.json(alerts);
  } catch (err) {
    res.status(500).json({ error: 'Не вдалося отримати алерти' });
  }
});

// Додавання нового алерта (для тестування або для реальних подій)
```

```
router.post('/', async (req, res) => {
  try {
    const { deviceId, message, severity } = req.body;
    const newAlert = new Alert({ deviceId, message, severity });
    const savedAlert = await newAlert.save();
    res.status(201).json(savedAlert);
  } catch (err) {
    res.status(400).json({ error: 'Неможливо створити алерт' });
  }
});

// Оновлення статусу алерта на прочитаний (isRead = true)
router.patch('/:id/read', async (req, res) => {
  try {
    const alertId = req.params.id;
    const updatedAlert = await Alert.findByIdAndUpdate(
      alertId,
      { $set: { isRead: true } },
      { new: true }
    );
    if (!updatedAlert) {
      return res.status(404).json({ error: 'Алерт не знайдено' });
    }
    res.json(updatedAlert);
  } catch (err) {
    res.status(500).json({ error: 'Помилка оновлення алерта' });
  }
});

module.exports = router;
```

Лістинг А.4 – server/scripts/seedAlerts.js

```
// server/scripts/seedAlerts.js
const mongoose = require('mongoose');
const Alert = require('../models/Alert');

mongoose.connect('mongodb://127.0.0.1:27017/alertsdb', {
  useNewUrlParser: true,
  useUnifiedTopology: true
})
  .then(async () => {
    console.log('Підключено до MongoDB. Початок наповнення
тестовими даними...');

    // Видаляються всі записи, щоб почати з чистої колекції
    await Alert.deleteMany({});

    // Створюється масив тестових алертів
    const sampleAlerts = [
      {
```

```
        deviceId: 'sensor-001',
        message: 'Температура в нормі',
        severity: 'info',
        timestamp: new Date(),
        isRead: false
      },
      {
        deviceId: 'sensor-002',
        message: 'Виявлено перевищення температури',
        severity: 'warning',
        timestamp: new Date(),
        isRead: false
      },
      {
        deviceId: 'sensor-003',
        message: 'Критична несправність датчика',
        severity: 'critical',
        timestamp: new Date(),
        isRead: false
      }
    ];

    // Збереження тестових даних у базі
    await Alert.insertMany(sampleAlerts);
    console.log('Тестові дані успішно додані');

    // Закриття підключення до бази
    mongoose.connection.close();
  })
  .catch((err) => console.error('Помилка:', err));
```

Лістинг А.5 – client/package.json

```
{
  "name": "client",
  "version": "1.0.0",
  "description": "React-клієнт для відображення алертів та
візуалізації даних",
  "main": "index.js",
  "scripts": {
    "start": "react-scripts start",
    "build": "react-scripts build"
  },
  "dependencies": {
    "react": "^18.2.0",
    "react-dom": "^18.2.0",
    "react-scripts": "5.0.1",
    "recharts": "^2.4.2"
  },
  "devDependencies": {
    "tailwindcss": "^3.2.0",
```

```
    "autoprefixer": "^10.4.2",
    "postcss": "^8.4.6"
  }
}
```

client/tailwind.config.js

```
module.exports = {
  content: [
    "./src//*.{js,jsx,ts,tsx}",
    "./public/index.html"
  ],
  theme: {
    extend: {},
  },
  plugins: [],
}
```

Лістинг А.6 – client/public/index.html

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width,initial-
scale=1" />
    <title>Alert Client</title>
  </head>
  <body>
    <div id="root"></div>
  </body>
</html>
```

Лістинг А.7 – client/src/index.css

```
@tailwind base;
@tailwind components;
@tailwind utilities;

/* Додаткові стилі за потреби */
body {
  margin: 0;
  padding: 0;
}
```

Лістинг А.8 – client/src/index.js

```
import React from 'react';
import ReactDOM from 'react-dom/client';
import './index.css';
import App from './App';

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<App />);
```

client/src/App.jsx

```
import React from 'react';
import HomePage from './pages/HomePage';

function App() {
  return (
    <div className="min-h-screen bg-gray-100">
      <HomePage />
    </div>
  );
}

export default App;
```

client/src/pages/HomePage.jsx

```
import React from 'react';
import AlertList from '../components/AlertList';
import ChartComponent from '../components/ChartComponent';

function HomePage() {
  return (
    <div className="container mx-auto py-8">
      <h1 className="text-2xl font-bold mb-6">Останні Алерти</h1>
      <AlertList />

      <h2 className="text-2xl font-bold mt-8 mb-4">Графік
показників</h2>
      <ChartComponent />
    </div>
  );
}

export default HomePage;
```

client/src/components/AlertList.jsx

```
import React, { useEffect, useState } from 'react';
import AlertItem from './AlertItem';

function AlertList() {
  const [alerts, setAlerts] = useState([]);

  useEffect(() => {
```

```
    fetchAlerts();
  }, []);

  const fetchAlerts = async () => {
    try {
      // Звернення до сервера, який, ймовірно, працює на 3001
порту
      const response = await
fetch('http://localhost:3001/api/alerts');
      const data = await response.json();
      setAlerts(data);
    } catch (error) {
      console.error('Помилка отримання алертів:', error);
    }
  };

  const markAsRead = async (alertId) => {
    try {
      const response = await
fetch(`http://localhost:3001/api/alerts/${alertId}/read`, {
        method: 'PATCH'
      });
      const updatedAlert = await response.json();
      setAlerts(prevAlerts =>
        prevAlerts.map(a => a._id === updatedAlert._id ?
updatedAlert : a)
      );
    } catch (error) {
      console.error('Помилка оновлення алерта:', error);
    }
  };

  return (
    <div className="space-y-4">
      {alerts.map(alert => (
        <AlertItem
          key={alert._id}
          alert={alert}
          onMarkAsRead={markAsRead}
        />
      ))}
    </div>
  );
}

export default AlertList;
```

client/src/components/AlertItem.jsx

```
import React from 'react';

function AlertItem({ alert, onMarkAsRead }) {
```

```
  const { _id, deviceId, message, severity, timestamp, isRead } =
alert;

  const handleMarkAsRead = () => {
    if (!isRead) {
      onMarkAsRead(_id);
    }
  };

  return (
    <div className="bg-white p-4 shadow rounded flex justify-
between items-center">
      <div>
        <p className="font-semibold">
          Пристрій: <span className="font-
normal">{deviceId}</span>
        </p>
        <p className="mt-1">
          Повідомлення: <span className="italic">{message}</span>
        </p>
        <p className="mt-1">Критичність: {severity}</p>
        <p className="text-sm text-gray-500 mt-1">
          Час: {new Date(timestamp).toLocaleString()}
        </p>
      </div>
      <button
        onClick={handleMarkAsRead}
        className={`px-4 py-2 rounded ml-4 ${
          isRead ? 'bg-gray-300 text-gray-600' : 'bg-blue-500
text-white'
        }`}
        disabled={isRead}
      >
        {isRead ? 'Прочитано' : 'Позначити як прочитане'}
      </button>
    </div>
  );
}

export default AlertItem;
```

client/src/components/ChartComponent.jsx

```
import React, { useEffect, useState } from 'react';
import {
  LineChart,
  Line,
  XAxis,
  YAxis,
  CartesianGrid,
  Tooltip,
  Legend,
  ResponsiveContainer
```

```
} from 'recharts';

function ChartComponent() {
  const [data, setData] = useState([]);

  useEffect(() => {
    // Імітація тестових даних; можна замінити на звернення до API
    const testData = [
      { timestamp: '2025-01-01 10:00', value: 24 },
      { timestamp: '2025-01-01 10:05', value: 28 },
      { timestamp: '2025-01-01 10:10', value: 31 },
      { timestamp: '2025-01-01 10:15', value: 29 },
      { timestamp: '2025-01-01 10:20', value: 33 }
    ];
    setData(testData);
  }, []);

  return (
    <div className="bg-white p-4 shadow rounded" style={{ width:
'100%', height: 400 }}>
      <ResponsiveContainer>
        <LineChart data={data}>
          <CartesianGrid strokeDasharray="3 3" />
          <XAxis dataKey="timestamp" />
          <YAxis />
          <Tooltip />
          <Legend />
          <Line type="monotone" dataKey="value" />
        </LineChart>
      </ResponsiveContainer>
    </div>
  );
}

export default ChartComponent;
```