НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ БІОРЕСУРСІВ І ПРИРОДОКОРИСТУВАННЯ УКРАЇНИ

Факультет ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

ПОГОДЖЕНО	**ДОПУСКАЄТЬСЯ ДО ЗАХИСТУ**
Декан факультету Інформаційних технологій	**Завідувач кафедри** Комп'ютерних систем, мереж та кібербезпеки
_____Ігор БОЛБОТ	_____Дмитро КАСАТКІН
(підпис)	(підпис)
«___»_____2026 р.	«___»_____2026 р.

БАКАЛАВРСЬКА КВАЛІФІКАЦІЙНА РОБОТА

На тему: «Розробка комп'ютерної системи моніторингу напруги живлення»

Спеціальність F7 «Комп’ютерна інженерія»

Освітньо-професійна програма «Комп’ютерна інженерія»

Гарант освітньої програми
канд. фіз.-мат. наук, доцент _____ (підпис) **Євгеній НІКІТЕНКО**

Керівник бакалаврської кваліфікаційної роботи
ст. викладач _____ (підпис) **Володимир МАТІЄВСЬКИЙ**

Виконав _____ (підпис) **Іван ДУБ**

КИЇВ-2026

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ БІОРЕСУРСІВ І ПРИРОДОКОРИСТУВАННЯ УКРАЇНИ

Факультет ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

ЗАТВЕРДЖУЮ
Завідувач кафедри
комп'ютерних систем, мереж та кібербезпеки
канд. пед. наук, доцент_____Дмитро КАСАТКІН
«_____»_____20____ р.

З А В Д А Н Н Я
ДО ВИКОНАННЯ БАКАЛАВРСЬКОЇ КВАЛІФІКАЦІЙНОЇ РОБОТИ
ЗДОБУВАЧУ

Дубу Івану Олександровичу
(прізвище, ім'я, по батькові)

Спеціальність «Комп'ютерна інженерія »

Освітньо-професійна програма «Комп'ютерна інженерія »

Тема кваліфікаційної бакалаврської роботи
«Розробка комп'ютерної системи моніторингу напруги живлення
»

затверджена наказом ректора НУБіП України від «10»___12_____2025 р. №___3072 «С»___
Термін подання завершеної роботи на кафедру «15»___05_____2026 р.
Вихідні дані до бакалаврської кваліфікаційної роботи
вимоги до корпоративної мережі IT компанії

Специфікації пристроїв

Перелік питань, що підлягають дослідженню:
Аналіз предметної області ті обгрунтування розробки системи моніторингу

Проєктування системи в середовищі Wokwi Розробка та проєктування апаратної частини

Розробка та написання серверної частини Експериментальне дослідження та робота над помилками

Дата видачі завдання "__10__"_____12_____2025 р.

Керівник бакалаврської кваліфікаційної роботи _____ **Володимир МАТІЄВСЬКИЙ**
ст. викладач (підпис)

Завдання взяв до виконання _____ **Іван ДУБ**
(підпис)

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів дипломного проекту (роботи)	Строк виконання етапів проекту (роботи)	Примітка
1	Аналіз предметної області ті обгрунтування розробки системи моніторингу	10.02.2026 р.	Виконано
2	Проєктування системи в середовищі Wokwi	15.03.2026	Виконано
3	Розробка та проєктування апаратної частини	20.04.2026	Виконано
4	Розробка та написання серверної частини	07.05.2026	Виконано
5	Експериментальне дослідження та робота над помилками	15.05.2026	Виконано

Студент_____ / Іван ДУБ /

(підпис) (ініціали та прізвище)

Керівник проекту (роботи)_____ / Володимир МАТІЄВСЬКИЙ/

(підпис) (ініціали та прізвище)

РЕФЕРАТ

Бакалаврська кваліфікаційна робота: 91 с., 23 рис., 10 табл., 19 джерел, 2 додатки

КЛЮЧОВІ СЛОВА: ESP32, МОНІТОРИНГ ЕЛЕКТРОМЕРЕЖІ, TELEGRAM-БОТ, NODE.JS, АВТОНОМНІСТЬ, DEEP SLEEP, ЛОГУВАННЯ, ГІБРИДНА СИСТЕМА ЗБЕРЕЖЕННЯ.

Об’єктом дослідження є процеси дистанційного моніторингу та аналізу параметрів побутових електромереж в умовах нестабільного енергопостачання та аварійних відключень. Предметом дослідження є програмно-апаратні засоби створення енергоефективної системи на базі мікроконтролера ESP32 для фіксації перепадів напруги, логування подій на MicroSD-карту та передачі даних на сервер Node.js. Мета роботи полягає у проектуванні та розробці автономного пристрою моніторингу, здатного функціонувати в умовах тривалих блекаутів, забезпечуючи користувача оперативною інформацією через Telegram та формуючи доказову базу інцидентів у мережі.

Розроблено комплексне рішення, що інтегрує апаратну детекцію напруги з режимом глибокого сну (Deep Sleep), що дозволяє досягти автономності до 6 діб. Новизна полягає у гібридній схемі синхронізації локальних логів із SD-карти до хмарної бази даних SQL після відновлення зв’язку. Розроблено Telegram-інтерфейс із динамічною візуалізацією графіків через QuickChart API.

Практичне значення отриманих результатів полягає у можливості використання пристрою як «чорної скриньки» для захисту прав споживачів перед енергопостачальними компаніями, а також як елемента системи «розумного дому» для превентивного захисту електроприладів від критичних стрибків напруги.

Проведені випробування підтвердили працездатність системи при напругах від 160В до 260В. Обрана архітектура на базі Node.js забезпечує мінімальну затримку сповіщень (до 2 сек). Система є масштабованою та готовою до впровадження у форматі серійного пристрою. Сфера застосування: енергоменеджмент, побутовий та промисловий моніторинг, автоматизація систем безпеки.

Зміст

ПЕРЕЛІК СКОРОЧЕНЬ ТА УМОВНИХ ПОЗНАЧЕНЬ

API (Application Programming Interface) - прикладний програмний інтерфейс;
CSV (Comma-Separated Values) - текстовий формат, призначений для представлення табличних даних;
DNS (Domain Name System) - система доменних імен;
ESP32 - родина низькоенергозатратних систем на кристалі з інтегрованими модулями Wi-Fi та Bluetooth;
GPIO (General-Purpose Input/Output) - інтерфейс введення-виведення загального призначення;
HTTP/HTTPS (HyperText Transfer Protocol Secure) - протокол передачі гіпертексту з підтримкою шифрування;
IDE (Integrated Development Environment) - інтегроване середовище розробки;
IoT (Internet of Things) - інтернет речей;
JSON (JavaScript Object Notation) - текстовий формат обміну даними;
MAC-адреса (Media Access Control address) - унікальний ідентифікатор мережевого обладнання;
MCU (Microcontroller Unit) - мікроконтролер;
MVP (Minimum Viable Product) - мінімально життєздатний продукт;
NTP (Network Time Protocol) - протокол мережевого часу;
OTA (Over-the-Air) - технологія бездротового оновлення програмного забезпечення;
P2P (Peer-to-Peer) - однорангова мережа;
PCB (Printed Circuit Board) - друкована плата;
PNG (Portable Network Graphics) - растровий формат збереження графічних зображень;
RTC (Real-Time Clock) - годинник реального часу;
SPI (Serial Peripheral Interface) - послідовний периферійний інтерфейс;
SSL/TLS (Secure Sockets Layer / Transport Layer Security) - протоколи криптографічного захисту передачі даних;
UART (Universal Asynchronous Receiver-Transmitter) - універсальний асинхронний приймач-передавач;
USDT (Tether) - криптовалютний стейблкоїн;
VPS (Virtual Private Server) - віртуальний приватний сервер.

РОЗДІЛ 1АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ОБҐРУНТУВАННЯ РОЗРОБКИ СИСТЕМИ МОНІТОРИНГУ

1.1 Актуальність проблеми моніторингу параметрів електромереж в сучасних умовах

У сучасних реаліях стан енергетичної системи України став одним із найбільш критичних чинників, що впливають на повсякденне життя громадян та стабільність роботи бізнесу. Через постійні терористичні атаки на об'єкти енергетичної інфраструктури, магістральні мережі та підстанції, вітчизняна енергосистема працює в екстремальних режимах. Це призводить не лише до дефіциту потужності та запровадження графіків відключень, але й до суттєвого погіршення якості самої електроенергії. Тому, питання моніторингу напруги сьогодні виходить за межі суто технічного інтересу і стає питанням економічної безпеки кожного домогосподарства [2].

Важливо підкреслити, що основною проблемою є не просто відсутність світла, а процеси, які відбуваються в мережі в моменти вимкнення та, особливо, увімкнення великих сегментів споживачів. Коли відбувається масове відновлення живлення після блекауту, в мережі виникають перехідні процеси, що супроводжуються потужними комутаційними перенапругами. Ці стрибки можуть сягати значень у 300-400 вольт, що є фатальним для більшості побутових приладів [12].

Було досліджено, що більшість сучасної побутової техніки, такої як холодильники, пральні машини, кондиціонери та комп’ютерне обладнання, розрахована на роботу у вузькому діапазоні напруги, зазвичай це 220-230 вольт з відхиленням не більше ніж десять відсотків. Постійні коливання та вихід за ці межі призводять до деградації ізоляції обмоток двигунів, перегріву імпульсних блоків живлення та виходу з ладу мікропроцесорних компонентів керування. Проблема ускладняється тим, що споживач часто навіть не знає, що саме стало причиною поломки, оскільки стрибок напруги може тривати долі секунди, але цього достатньо для пробою напівпровідникових елементів.

У ході роботи над проєктом було підтверджено, що існує величезна прогалина у взаємовідносинах між постачальником послуг та кінцевим споживачем. Згідно з чинним законодавством та державними стандартами, постачальник зобов'язаний підтримувати якість напруги, проте на практиці довести факт порушення цих норм без наявності сертифікованих або хоча б зафіксованих логів вимірювань майже неможливо. Це створює ситуації, коли сервісні центри відмовляють у гарантійному ремонті дорогої техніки, посилаючись на "неправильну експлуатацію" або "зовнішні фактори", а енергопостачальна компанія не визнає своєї провини через відсутність доказової бази у клієнта.

Таблиця 1.1 Вплив різних рівнів відхилення напруги на термін експлуатації різних категорій побутової техніки

Категорія техніки	**Відхилення (U < 190В)**	**Відхилення (U > 250В)**	**Короткочасний імпульс (> 350В)**	**Прогнозний термін експлуатації (при системних збоях)**
Компресорне обладнання (Холодильники, кондиціонери)	Перегрів обмоток через неможливість запуску.	Пробій ізоляції пускового реле.	Заклинювання або згоряння статора.	**Скорочення на 60-80%**
Імпульсні БЖ (ТБ, ПК, зарядні пристрої)	Робота на межі можливостей ШІМ-контролера.	Перегрів вхідних електролітичних конденсаторів.	Вибух варистора, пробій силового транзистора.	**Скорочення на 40-50%**
Нагрівальні елементи (Бойлери, чайники)	Збільшення часу нагріву (неефективність).	Прискорене окислення та руйнування ТЕНа.	Зазвичай витримують (інерційність).	**Скорочення на 20%**
LED-освітлення	Мерехтіння (стробоскопічний ефект).	Деградація кристалів, вихід з ладу драйвера.	Миттєве вигорання матриці.	**Скорочення на 90%**

В ході роботи було проаналізовано, що використання звичайних реле напруги (так званих «відсікачів») лише частково вирішує проблему, оскільки

вони просто розривають коло при досягненні критичних значень, але не дають жодної інформації про те, що відбувалося з мережею протягом дня або тижня. Для повноцінного захисту та, що не менш важливо, для отримання доказів у разі судових спорів або звернень по гарантії, необхідна система, яка веде безперервний лог параметрів з прив'язкою до реального часу.

Важливим аспектом актуальності є також психологічний фактор. В умовах невизначеності щодо графіків відключень, можливість віддалено перевірити стан мережі у власній оселі через Telegram-бот або веб-інтерфейс надає користувачу відчуття контролю над ситуацією. Це дозволяє вчасно зреагувати, наприклад, попросити сусідів або родичів вимкнути прилади з розеток, якщо система сигналізує про аномально високу або низьку напругу, яка тримається тривалий час.

Досліджуючи стан питання, стало помітним, що в Україні спостерігається дефіцит бюджетних систем, які б поєднували в собі функції вольтметра, реєстратора даних та системи сповіщень. Більшість промислових аналогів коштують сотні доларів і складні у налаштуванні для пересічного громадянина. Моє рішення покликане заповнити цю нішу, використовуючи відкриті технології та зручні інтерфейси, такі як месенджери, які вже стали частиною щоденного життя кожного українця.

Дуже вірогідно, що в умовах післявоєнного відновлення енергетики такі системи стануть базовим елементом «розумного будинку», оскільки вони дозволяють збирати об'єктивні дані про реальний стан мереж на місцях. Це допоможе енергетикам виявляти проблемні ділянки, де напруга постійно занижена через перевантаження фаз, та приймати обґрунтовані рішення щодо модернізації інфраструктури. Таким чином, актуальність теми бакалаврської роботи підтверджується як миттєвими потребами захисту майна громадян, так і довгостроковими перспективами розвитку цифровізації енергетичного сектору України.

1.2 Обґрунтування концепції та архітектурних рішень системи моніторингу

Процес формування ідеї створення власної системи моніторингу став результатом тривалого пошуку оптимального балансу між функціональністю, надійністю та доступністю. Початковим етапом мого дослідження став аналіз ринку готових рішень, зокрема популярних сьогодні розумних розеток та реле напруги з підтримкою дистанційного керування. Було досягнено висновку, що більшість таких пристроїв орієнтовані на виконання простих сценаріїв автоматизації, наприклад, увімкнення освітлення за розкладом або вимкнення приладів при перевищенні лімітів потужності. Проте, коли питання стосується саме професійного аналізу якості мережі та накопичення доказової бази для гарантійних випадків, комерційні побутові рішення виявляються недостатніми.

У процесі вивчення існуючих аналогів я звернув увагу на продукцію китайського виробництва, яка масово представлена на українському ринку під різними брендами. При чому необхідно зазначити, що використання таких пристроїв для моніторингу критичної інфраструктури, якою є електромережа помешкання, несе в собі значні ризики. По-перше, більшість дешевих китайських розеток проєктуються на базі вузькоспеціалізованих інтегральних схем, де вся логіка зашита в одну плату без можливості її модифікації. В загалому, це суттєво обмежує гнучкість системи. По-друге, не можна ігнорувати геополітичний аспект та питання кібер-безпеки. Враховуючи статус Китаю як стратегічного партнера країни-агресора, використання пристроїв, що працюють через китайські хмарні сервіси (наприклад, Tuya або eWeLink), створює загрозу конфіденційності даних та потенційну можливість віддаленого втручання в роботу домашньої мережі.

Було проаналізовано архітектуру типової розумної розетки і виявлено, що її конструкція передбачає використання компактної друкованої плати з інтегрованим модулем вимірювання струму та напруги. Для реалізації подібного заліза у промислових масштабах потрібні значні інвестиції у проєктування багатошарових плат та закупівлю специфічних компонентів, що є недоступним у

межах студентського проєкту або дрібносерійного виробництва. Проблемою стала відсутність достатньої ресурсної бази для створення власної інтегральної схеми, яка б конкурувала за габаритами з китайськими аналогами, проте існує можливість побудувати систему на базі універсальних мікроконтролерів, що забезпечить набагато більшу потужність обробки даних та повну незалежність від закордонних хмарних серверів.

Таблиця 1.2 Порівняльний аналіз кастомних систем на базі мікроконтролерів та готових комерційних IoT розеток

Критерій порівняння	Готові рішення (Tuya/eWeLink)	Кастомна система (Arduino/ESP32)
Локація серверів	Переважно КНР (Cloud-only)	Локальний сервер / Власний Telegram-бот
Прошивка (Firmware)	«Закрита» (Proprietary), неможливо змінити	Відкритий код (Open Source), повний контроль
Залежність від інтернету	Без доступу до хмари - лише ручне керування	Повна автономність (запис у внутрішню пам’ять)
Конфіденційність	Збір телеметрії постачальником	Дані належать виключно користувачу
Гнучкість логіки	Обмежена стандартним додатком	Можливість написання будь-яких алгоритмів
Ризик віддаленого впливу	Високий (через вразливості в хмарі)	Мінімальний (закритий локальний контур)

Переходячи від аналізу апаратних засобів до формування логіки продукту, визначено, що система має не просто фіксувати поточну напругу, а й виконувати роль відмовостійкого реєстратора. У процесі пошуку способів реалізації розглянуто варіант автономного пристрою, який би виводив дані на локальний дисплей. Проте такий підхід не вирішує проблему сповіщення користувача в реальному часі та не дозволяє переглядати історію подій за тривалий період. На думку автора, сучасна система моніторингу повинна базуватися на клієнт-серверній архітектурі.

Рисунок 1.1 - Концептуальна схема руху даних від вимірювального вузла до кінцевого

Реалізація цієї ідеї має виглядати наступним чином: клієнтська частина на базі мікроконтролера виконує безпосереднє зчитування аналогового сигналу, його первинну обробку та фільтрацію. Серверна частина, своєю чергою, відповідає за зберігання бази даних, візуалізацію графіків та інтеграцію з месенджерами. Такий розподіл функцій дозволяє реалізувати механізми відмовостійкості. Наприклад, якщо інтернет-з'єднання зникає, мікроконтролер повинен мати можливість накопичувати дані у власну пам'ять або на зовнішній носій, а після відновлення зв'язку синхронізувати ці дані з сервером. Це критично важливо під час блекаутів, коли роутери можуть вимикатися разом із мережею.

Досліджуючи можливості програмної реалізації, окремо виділено важливість обробки даних у режимі реального часу. Потрібно зауважити, що функція попередження про стан мережі повинна мати інтелектуальний характер. Тобто система не має просто завалювати користувача повідомленнями про кожну зміну напруги на 1 вольт. Заплановано реалізувати алгоритми аналізу аномалій, які будуть спрацьовувати лише при перетині встановлених безпечних порогів або при різких стрибках, що виходять за межі допустимого гістерезису.

Ось важлива теза щодо архітектури таких систем: "Distributed systems for power monitoring allow for higher reliability by offloading data storage to robust servers while keeping the edge devices simple and efficient". Це твердження лягло в основу ідеї вибору на користь поєднання мікроконтролера та серверної частини. Склалося переконання, що такий підхід дозволить уникнути обмежень, притаманних закритим китайським пристроям, і надасть користувачу інструмент професійного рівня для аналітики власної мережі.

Важливим етапом формування ідеї було визначення способу взаємодії користувача з системою. Прияте рішення відмовитися від створення власного мобільного додатка на користь Telegram-бота. Воно зумовлене тим, що месенджери мають вбудовану систему push-сповіщень, яка працює стабільно навіть при низькій швидкості інтернету, що є актуальним для будь-яких випадків. Крім того, бот дозволяє легко реалізувати командування системою та запит статистичних звітів у вигляді графіків, не вимагаючи від користувача встановлення додаткового програмного забезпечення.

Окремим вагомим аргументом на користь обраної мною архітектури є гнучкість у питаннях супроводу та модернізації системи. Важливо, що в умовах динамічної зміни вимог до програмного забезпечення можливість вносити корективи в роботу сервісу без необхідності фізичного доступу до пристроїв користувачів є критичною перевагою. Оскільки основна логіка обробки даних, формування аналітики та взаємодія з месенджерами винесена на серверний бекенд, є можливість виправляти помилки, змінювати алгоритми фільтрації напруги або додавати нові функції (наприклад, розрахунок вартості спожитої

енергії) миттєво для всіх активних вузлів одночасно. Користувач при цьому не відчуває жодних незручностей, адже його пристрій продовжує безперебійно збирати телеметрію, тоді як серверна частина оновлюється у фоновому режимі.

Було зазначено, що такий підхід дозволяє реалізувати концепцію безперервного вдосконалення продукту. Якщо в ході експлуатації виявиться, що встановлені пороги сповіщень про аномалії потребують корекції, мені достатньо внести зміни в один файл на сервері, і нова логіка відразу почне застосовуватися до всіх вхідних пакетів даних. Це значно спрощує масштабування системи, оскільки підтримка навіть сотень пристроїв не вимагатиме від мене написання індивідуальних патчів для кожного окремого мікроконтролера.

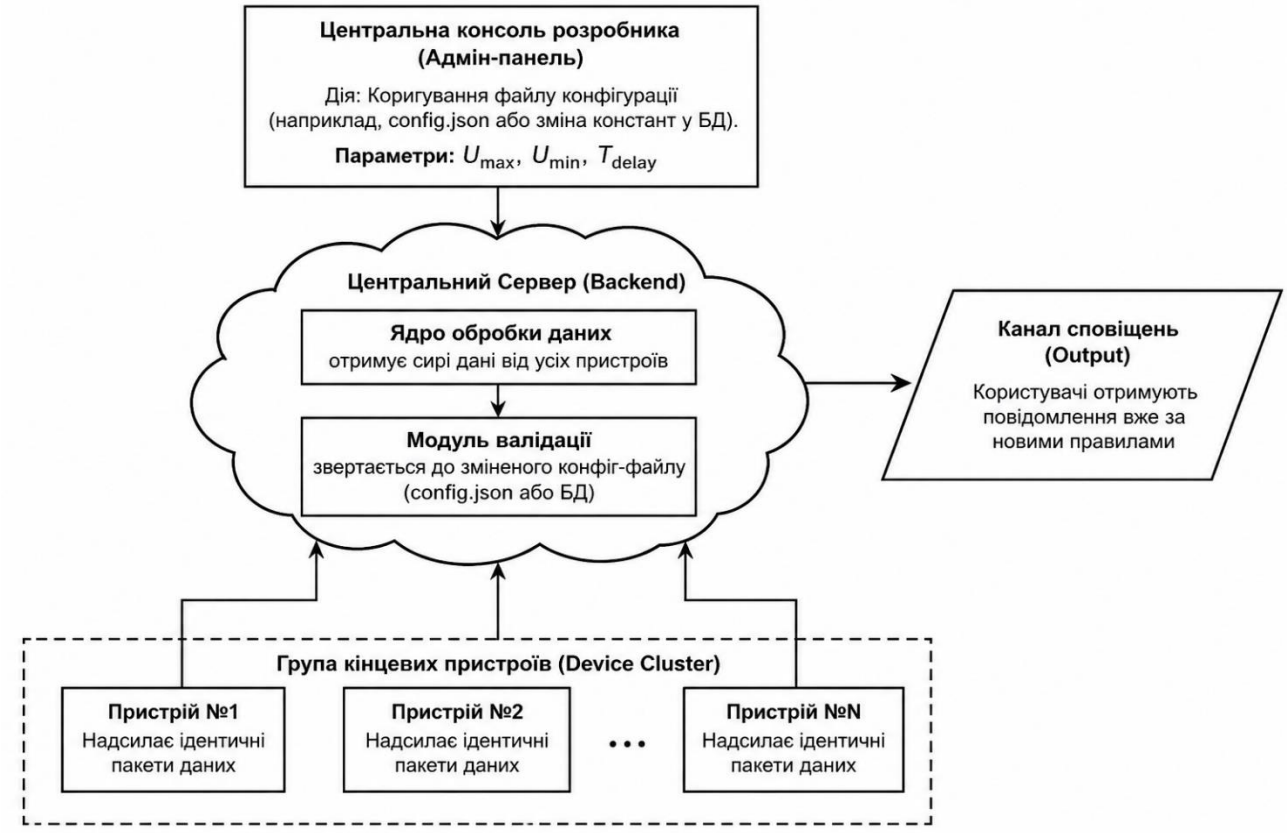


Рисунок 1.2 - Схема оновлення логіки системи через централізований сервер без втручання в кінцеве обладнання

Крім модернізації серверної частини, досліджено концепцію віддаленого оновлення безпосередньо програмного коду кінцевого вимірювального пристрою. Ця технологія, відома у світі інтернету речей як OTA (Over The Air) оновлення, стала для мене одним із ключових критеріїв при майбутньому виборі

мікроконтролера. Дійшлося висновку, що впровадження механізму бездротового оновлення дозволить мені за потреби модифікувати навіть низькорівневі алгоритми зчитування даних з датчиків або оновлювати протоколи шифрування, не вимагаючи від кінцевого користувача фізичного втручання до пристрою та його підключення до комп'ютера через кабель.

Формуючи вимоги до апаратної бази, визначено, що ідеальна платформа повинна мати вбудовану підтримку розділення вбудованої пам'яті на кілька логічних областей. Таке архітектурне рішення дозволяє завантажувати нову версію програмного забезпечення паралельно з безперебійною роботою поточної версії. Тільки після повного завантаження та успішної перевірки цілісності нового образу пам'яті, пристрій має виконувати автоматичне перезавантаження з уже оновленим кодом. Відповідно, при подальшому пошуку мікроконтролера я буду відхиляти ті варіанти, які не здатні забезпечити такий рівень відмовостійкості під час процесів оновлення системи.

Таблиця 1.3 — Порівняльний аналіз методів оновлення програмного забезпечення мікроконтролерів

Параметр порівняння	Фізичне підключення (UART/USB)	Віддалене оновлення (OTA / Dual Bank)
Доступність системи	Повна зупинка під час прошивки.	Безперебійна робота (запис у фонову область).
Людський фактор	Високий ризик механічних пошкоджень роз’ємів.	Автоматизований процес без фізичного контакту.
Ризик критичної помилки	При збої живлення потрібен програматор.	Автоматичний відкат (Rollback) до робочої версії.
Масштабованість	$1 \text{ людина} = 1 \text{ пристрій}$.	$1 \text{ сервер} = 1000+ \text{ пристроїв одночасно}$.
Перевірка цілісності	Зазвичай відсутня (залежить від ПЗ).	Обов’язкова перевірка Checksum/Hash образу.

Така особливість архітектури зробить спроєктовану систему надзвичайно зручною та непомітною для кінцевого споживача. Користувачу не потрібно буде володіти спеціальними технічними знаннями для підтримки свого пристрою в актуальному стані. Усі складні технічні маніпуляції виконуватимуться виключно

на стороні розробника. Заплановано що наявність такого функціоналу в апаратній частині є одним із головних чинників, який здатен забезпечити життєздатність та популярність подібних рішень серед широкого загалу. Сучасні споживачі цінують абсолютну простоту питанням економічної безпеки кожного домогосподарства у поєднанні з професійним рівнем аналітики, тому майбутня апаратна база зобов'язана відповідати цим високим очікуванням.

Підсумовуючи роздуми щодо архітектури моніторингового комплексу, складається переконання, що синергія гнучкої серверної частини та можливостей віддаленого патчингу кінцевого мікроконтролера створить стійку до відмов екосистему. Це дозволить інженеру зосередитися на безперервному покращенні якості аналізу даних, не турбуючись про технічні обмеження доступу до обладнання, що вже перебуває в експлуатації у клієнтів. Саме така стратегія розробки та суворі критерії відбору комплектуючих повинна бути найбільш ефективними для реалізації сучасних систем моніторингу електромереж [11, 12].

Таким чином, пройшовши шлях від вивчення доступних на ринку розеток низької якості до проєктування власної архітектури, сформовано концепцію гнучкої, захищеної та відмовостійкої системи. Вона базується на доступному залізі, але завдяки продуманому програмному забезпеченню та серверній підтримці, забезпечує функціонал, що перевершує більшість комерційних зразків. За цією версією, це єдина доцільна стратегія реалізації проєкту в умовах обмежених ресурсів та підвищених вимог до безпеки даних.

Вважається, що така деталізація процесу пошуку та обґрунтування вибору архітектури дозволяє чітко зрозуміти переваги кастомної розробки. Використання мікроконтролера як основи дає мені повний контроль над фізичним рівнем вимірювання, а серверна частина гарантує збереження історії та зручність аналізу, що є критичним для досягнення поставленої мети роботи.

1.3 Обґрунтування вибору апаратної бази та програмних засобів реалізації

Для реалізації надійної та функціональної системи моніторингу електромережі необхідно провести ретельний аналіз існуючих апаратних рішень. Виділено три основні класи пристроїв, які найчастіше використовуються в проєктах Інтернету речей: одноплатні комп'ютери (на прикладі Raspberry Pi), класичні мікроконтролери (Arduino Nano) та сучасні системи на кристалі (ESP32). Кожен із цих об'єктів має свої переваги та обмеження, які було проаналізовано з точки зору вартості, енергоспоживання, обчислювальної потужності та наявності вбудованих інтерфейсів зв'язку.

Аналіз платформи Raspberry Pi Першим об'єктом дослідження став Raspberry Pi. Це потужне рішення, яке по суті є повноцінним комп'ютером на базі архітектури ARM. Використання такої платформи для простого логування напруги є надлишковим. Головна проблема полягає в тому, що Raspberry Pi працює під керуванням операційної системи (зазвичай Linux), що накладає певні ризики. Наприклад, раптове зникнення живлення, що є частим явищем у моєму сценарії, може призвести до пошкодження файлової системи на SD-карті.

Технічна характеристика щодо енергоспоживання цього пристрою: "The Raspberry Pi 4 requires a 5V 3A power supply. While it provides immense processing power, its typical power consumption ranges from 3W to 6W depending on the load". такі показники є неприйнятними для задуманого пристрою, оскільки при роботі від акумулятора в умовах блекауту Raspberry Pi розрядить батарею за лічені години, тоді як мікроконтролер може працювати днями.

Аналіз платформи Arduino Nano Наступним логічним кроком став розгляд Arduino Nano. Це надзвичайно популярна платформа серед розробників-початківців завдяки своїй простоті та низькій ціні. Проте, детально вивчивши специфікації, досягнено висновку, що вона не відповідає моїм вимогам щодо мережевої функціональності. Arduino Nano базується на мікроконтролері ATmega328P, який не має вбудованих модулів Wi-Fi або Bluetooth.

Цитується офіційний опис можливостей пам'яті цього пристрою: "The ATmega328P has 32 KB of flash memory for storing code, 2 KB of SRAM and 1 KB of EEPROM". Такої кількості оперативної пам'яті [15, 16] (SRAM) недостатньо для стабільної роботи зі стеком протоколів HTTPS та обробки JSON-пакетів, які є основою моєї системи. Додавання зовнішніх Wi-Fi модулів, таких як ESP8266, лише ускладнює схему, збільшує габарити пристрою та знижує загальну надійність через велику кількість з'єднань.

Аналіз платформи STM32 (Blue Pill) Розглянуто професійну лінійку мікроконтролерів STM32. Вони мають високу швидкість роботи та велику кількість апаратних інтерфейсів. Проте головним бар'єром для використання цієї платформи в моєму проєкті став складний поріг входження та відсутність інтегрованих бездротових рішень у бюджетних моделях.

Цитата з документації щодо периферії: "The STM32F103 medium-density performance line family incorporates the high-performance Arm Cortex-M3 32-bit RISC core operating at a 72 MHz frequency, high-speed embedded memories, and an extensive range of enhanced I/Os and peripherals". Попри вражаючі характеристики, відсутність нативного Wi-Fi на кристалі знову ж таки змушує використовувати зовнішні плати розширення, що суперечить ідеї створення компактного та автономного приладу.

Обґрунтування вибору ESP32 як основного контролера Після проведеного аналізу вибір зупинився на мікроконтролері ESP32 від компанії Espressif Systems. Це найбільш збалансоване рішення для задач IoT моніторингу. Ця система на кристалі поєднує в собі двоядерний процесор, низьке енергоспоживання та, що найважливіше, вбудовані модулі Wi-Fi та Bluetooth.

Згідно з технічною документацією [16], ESP32 володіє значно вищою тактовою частотою та вбудованими модулями Wi-Fi/BT, на відміну від ATmega328P. Тут необхідно навести цитату, що підтверджує переваги обраного АЦП (аналого-цифрового перетворювача), який є ключовим для вимірювання напруги: "The ESP32 integrates two 12-bit SAR ADCs and supports measurements on 18 channels. It also features a built-in low-noise amplifier and a programmable gain

amplifier". Висока розрядність АЦП (12 біт проти 10 біт у Arduino) дозволяє досягти значно більшої точності при зчитуванні показників електромережі.

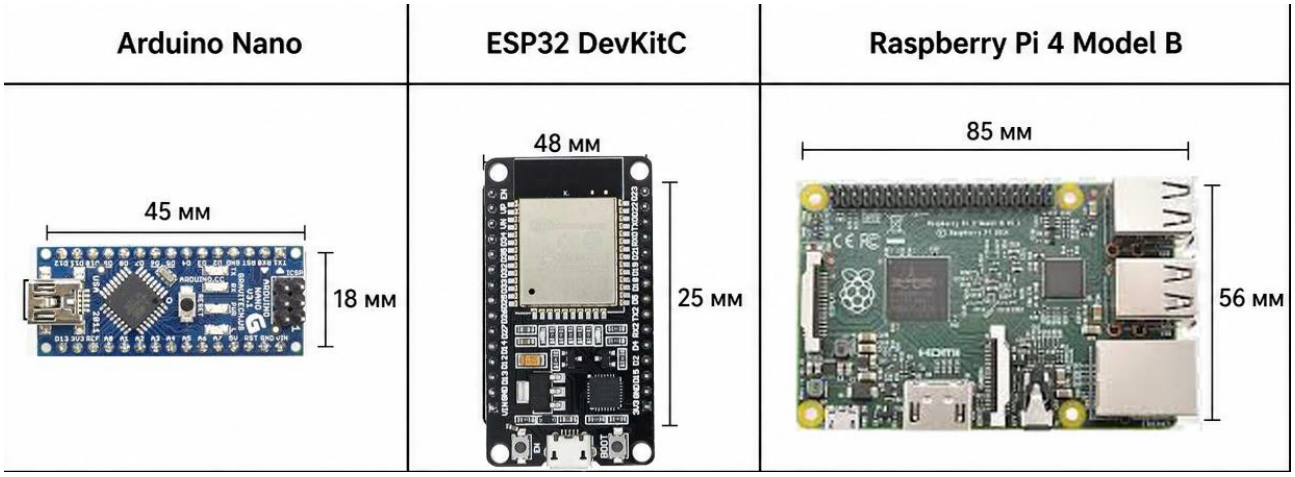


Рисунок 1.3 - Порівняння габаритів та ціни для Arduino Nano та ESP32 Raspberry Pi

Крім технічних переваг, ESP32 має величезну спільноту розробників та велику кількість готових бібліотек, що дозволяє значно скоротити час на розробку програмного забезпечення. Підтримка операційної системи реального часу FreeRTOS дає мені можливість паралельно виконувати збір даних, обробку черги на SD-карті та підтримку мережевого з'єднання без ризику зависання пристрою.

Обґрунтування вибору серверного середовища та бази даних Після визначення мікроконтролера ESP32 як основи апаратної частини, перейдемо до аналізу та вибору технологічного стеку для серверного бекенду. Серверна частина є ядром системи, оскільки вона відповідає за прийом телеметрії, її валідацію, збереження та маршрутизацію сповіщень. У процесі пошуку розглянуто три популярні платформи для розробки серверних додатків: PHP, Python (з використанням фреймворку Flask) та середовище Node.js.

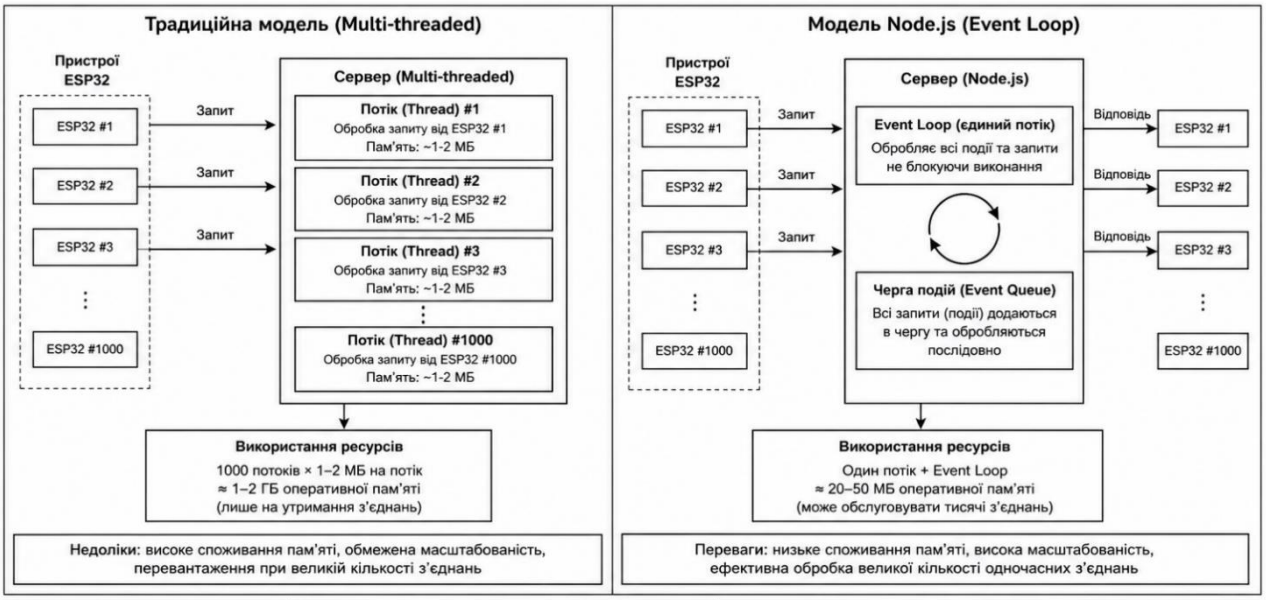
Мова PHP є класичним інструментом для веброзробки, проте її синхронна природа обробки запитів значно гірше підходить для систем реального часу. Python пропонує швидку розробку та безліч наукових бібліотек, але вимагає додаткових налаштувань для повноцінної асинхронної роботи (перехід від WSGI

до ASGI), що ускладнює розгортання проєкту. Тому вибір зупинився на середовищі Node.js.

Таблиця 1.4 - Порівняльна характеристика апаратних платформ для IoT-моніторингу

Характеристика	Raspberry Pi 4	Arduino Nano	STM32 (F103)	ESP32 (WROOM)
Архітектура	ARM Cortex-A72 (64-bit)	ATmega328P (8-bit)	ARM Cortex-M3 (32-bit)	Xtensa LX6 (32-bit)
Вбудований Wi-Fi	Є (Native)	Відсутній	Відсутній	Є (Native)
Розрядність АЦП	Відсутній (потрібен зовн.)	10 біт	12 біт	12 біт
SRAM (Оперативна)	2 – 8 ГБ	2 КБ	20 КБ	520 КБ
OTA-оновлення	Через пакетний менеджер	Неможливо (нативно)	Складна реалізація	Нативна підтримка
Енергоспоживання	Високе (3–6 Вт)	Низьке	Низьке	Середнє/Низьке
Складність розробки	Середня (Linux)	Низька (Easy)	Висока (Pro)	Середня (Balanced)

Для підтвердження свого рішення наведено цитату з офіційної документації розробників цієї платформи: "Node.js is an asynchronous event-driven JavaScript runtime, designed to build scalable network applications". Це твердження ідеально описує базову потребу мого проєкту. Асинхронна неблокуюча модель вводу та виводу дозволяє серверу на базі Node.js одночасно обробляти тисячі паралельних HTTPS-запитів від різних пристроїв ESP32 без виділення окремого потоку пам'яті для кожного з'єднання. Це робить сервер надзвичайно економним до ресурсів оперативної пам'яті, що дозволить у майбутньому розмістити його на найдешевших хмарних VPS-серверах без втрати продуктивності.



Параметр	Традиційна модель (Multi-threaded)	Модель Node.js (Event Loop)
Підхід	Кожен запит створює новий потік	Усі запити обробляються в одному потоці через чергу подій
Кількість потоків	1000 потоків для 1000 пристроїв	1 потік (Event Loop) для тисяч пристроїв
Споживання пам'яті	≈ 1–2 ГБ RAM	≈ 20–50 МБ RAM
Масштабованість	Обмежена	Висока
Продуктивність при великій кількості з'єднань	Низька (ризик перевантаження сервера)	Висока (ефективна обробка тисяч з'єднань)

Рисунок 1.4 - Порівняння традиційної багатопотокової моделі сервера та подієво орієнтованої неблокуючої моделі Node.js

Наступним кроком став вибір системи керування базами даних (СКБД) для надійного зберігання логів напруги. Проаналізовано повноцінні реляційні бази даних, такі як MySQL та PostgreSQL, а також вбудовану СКБД SQLite. Використання MySQL потребує запуску окремого фонового процесу, виділення значного обсягу оперативної пам'яті та регулярного адміністрування прав доступу. Для системи, де основною операцією є просте додавання нових рядків з телеметрією (команда INSERT), такий функціонал є надлишковим.

Тому обрано SQLite. Згідно з технічною специфікацією, ця технологія має фундаментальну відмінність від конкурентів: "SQLite is a C-language library that implements a small, fast, self-contained, high-reliability, full-featured, SQL database engine". Обрано SQLite саме через її автономність (self-contained) та відсутність необхідності налаштовувати сервер бази даних (zero-configuration). Усі накопичені дані про стан електромережі зберігаються у єдиному локальному файлі на диску сервера. Такий підхід гарантує максимальну портативність системи. Процес створення резервної копії або перенесення проєкту на інший

фізичний сервер зводиться до простого копіювання одного файлу power_log.db, що є величезною перевагою для індивідуального розробника.

Таблиця 1.5 - Порівняльний аналіз СКБД MySQL та SQLite за критеріями споживання ресурсів, швидкості розгортання та складності адміністрування

Критерій порівняння	MySQL / PostgreSQL	SQLite
Архітектура	Клієнт-серверна (окремий процес)	Безсерверна (вбудована бібліотека)
Зберігання даних	Набір файлів у системній директорії	Один файл на диску
Споживання RAM	Високе (від 100 МБ до кількох ГБ)	Мінімальне (кілька МБ)
Складність розгортання	Висока (встановлення, налаштування користувачів)	Нульова (створення файлу)
Паралельний доступ	Повноцінна підтримка багатьох записів одночасно	Обмежена (блокування всього файлу при записі)
Адміністрування	Потребує регулярного моніторингу та бекапів	Не потребує (копіювання файлу)

Вибір інтерфейсу взаємодії з користувачем (Telegram Bot API) Найважливішим етапом проєктування програмної частини став вибір способу комунікації між сервером та кінцевим користувачем. Розглянуто три можливі вектори розвитку: створення власного мобільного додатка для платформ Android та iOS, розробку вебдодатка (Web App) з графічним інтерфейсом у браузері та використання існуючих месенджерів.

Розробка нативних мобільних додатків потребує значних часових витрат на вивчення специфічних мов програмування (наприклад, Kotlin або Swift) або складних кросплатформних фреймворків. Крім того, це вимагає проходження тривалої модерації та сплати комісій за розміщення у магазинах Google Play та App Store. З іншого боку, вебінтерфейс є набагато простішим у реалізації. Однак він має критичний недолік для систем безпеки та моніторингу: надзвичайна складність реалізації надійних миттєвих push-сповіщень на мобільні пристрої. У випадку критичної аварії в електромережі (наприклад, стрибок до 280 Вольт),

користувач повинен отримати попередження негайно, а не тоді, коли він самостійно здогадається відкрити сторінку системи у своєму браузері.

Зваживши всі переваги та недоліки, прийнято рішення інтегрувати систему моніторингу з месенджером Telegram шляхом створення спеціалізованого програмного бота.

Переваги цього архітектурного підходу є беззаперечними. По-перше, додаток Telegram вже встановлений на смартфонах абсолютної більшості українців. Це повністю усуває бар'єр входу для нових клієнтів, оскільки їм не потрібно завантажувати невідоме програмне забезпечення, реєструвати нові акаунти чи запам'ятовувати паролі. Авторизація відбувається автоматично через унікальний ідентифікатор чату (chat ID).

По-друге, інфраструктура Telegram бере на себе всю найскладнішу мережеву роботу. Алгоритми месенджера забезпечують наскрізне шифрування каналу зв'язку, миттєву доставку push-повідомлень

Таблиця 1.6 - Порівняльний аналіз каналів сповіщення користувача: SMS, Web Push, Нативний мобільний додаток, Telegram Bot за критеріями вартості розробки, швидкості доставки та зручності використання

Канал сповіщення	Вартість розробки та підтримки	Швидкість доставки (Latency)	Надійність (Push-сповіщення)	Бар’єр входу для користувача
SMS-повідомлення	Висока (оплата за кожне SMS)	Середня (залежить від оператора)	Висока (працює без інтернету)	Низький
Web Push (Браузер)	Низька	Залежить від активності вкладки	Низька (блокується ОС)	Середній (треба дати дозвіл)
Мобільний додаток	Дуже висока	Висока	Висока	Високий (встановлення ПЗ)
Telegram Bot API	Мінімальна (Безкоштовно)	Максимальна (Real-time)	Висока (пріоритет месенджера)	Мінімальний (вже встановлено)

Крім текстових сповіщень, API Telegram дозволяє легко реалізувати повноцінний інтерфейс управління системою за допомогою вбудованих

клавіатур. Користувач може в один клік запитати поточний статус напруги, перевірити наявність світла вдома або згенерувати статистичний звіт. Завдяки інтеграції з сервісом генерації зображень (наприклад, QuickChart), сервер Node.js здатен рендерити деталізовані графіки напруги та відправляти їх безпосередньо у чат у вигляді звичайних фотографій. Це робить взаємодію з системою інтуїтивно зрозумілою та абсолютно безкоштовною як для користувача, так і для розробника.

Підсумовуючи результати підрозділу, сформувано остаточний технологічний стек проєкту. Апаратна частина базуватиметься на системі на кристалі ESP32. Програмний бекенд буде написаний на платформі Node.js із використанням легкої бази даних SQLite. Роль клієнтського термінала та системи сповіщень візьме на себе Telegram-бот. Можна стверджувати, що така комбінація є найбільш раціональним, економічно вигідним та відмовостійким рішенням для поставленої інженерної задачі.

1.4 Проблематика кібербезпеки IoT-систем та захист каналів передачі даних

Стрімкий розвиток концепції Інтернету речей (IoT) приніс не лише нові можливості для автоматизації побуту, але й безпрецедентні виклики у сфері інформаційної безпеки. У процесі проєктування власної системи моніторингу електромережі приділено особливу увагу аналізу потенційних вразливостей, оскільки будь-який пристрій, підключений до глобальної мережі, стає потенційною мішенню для зловмисників. Моє глибоке занепокоєння викликає той факт, що більшість бюджетних комерційних рішень та аматорських розробок ігнорують базові принципи криптографічного захисту, ставлячи під загрозу конфіденційність даних користувача та цілісність самої локальної мережі.

Необхідно підкреслити, що IoT-пристрій, який збирає телеметричні дані про стан енергоспоживання, володіє критично важливою інформацією [4, 18]. Аналізуючи графіки напруги та статуси підключення, зловмисник може з високою точністю визначити присутність господарів удома, графік їхньої

активності та навіть тип використовуваного обладнання. Тому захист каналу передачі даних між мікроконтролером ESP32 та сервером на Node.js є для мене не просто технічною опцією, а фундаментальною вимогою до проєкту.

Досліджуючи типові вразливості IoT-архітектур, виявлено, що найбільш поширеною проблемою є використання відкритого протоколу HTTP (HyperText Transfer Protocol) для передачі телеметрії. У цьому сценарії дані передаються у вигляді звичайного тексту (plaintext).

Для підтвердження критичності цієї проблеми наведено витяг з документації щодо безпеки мережевих протоколів: "HTTP traffic is unencrypted, meaning any node between the client and the server can intercept, read, and even modify the data being transmitted, making it highly susceptible to Man-in-the-Middle (MitM) attacks". Це означає, що використання відкритого протоколу дозволяє будь-якому перехоплювачу не лише зчитувати дані про напругу, але й підміняти їх, генеруючи хибні сповіщення або блокуючи передачу реальної інформації про аварію.

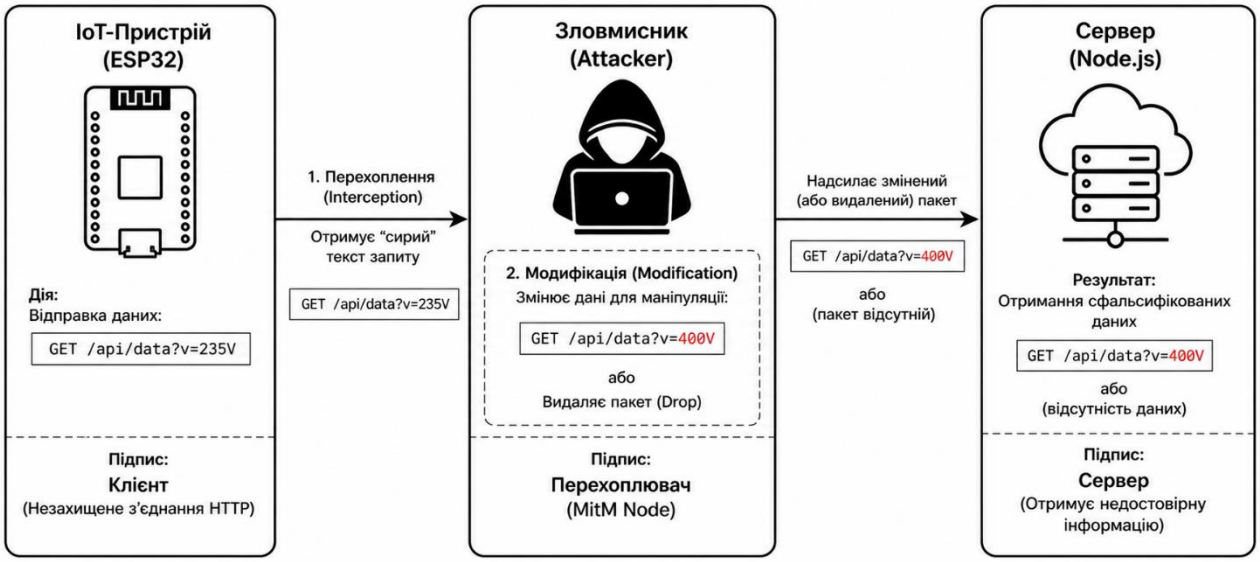


Рисунок 1.5 - Схематичне зображення атаки Man-in-the-Middle (MitM) при використанні незахищеного протоколу HTTP

Зважаючи на цю проблематику, прийнято категоричне рішення відмовитися від використання HTTP на користь захищеного протоколу HTTPS

(HyperText Transfer Protocol Secure). Це вимагає впровадження криптографічних механізмів безпосередньо в код мікроконтролера та налаштування SSL/TLS сертифікатів на серверній частині.

Реалізація такого підходу вимагає від апаратної платформи значних обчислювальних ресурсів для шифрування пакетів у реальному часі. Саме тут повною мірою розкриваються переваги обраного мною раніше мікроконтролера ESP32. Проведено аналіз його технічних специфікацій і знайшов підтвердження його здатності виконувати такі задачі: "The ESP32 incorporates hardware accelerators for various cryptographic algorithms, including AES, SHA-2, RSA, and Elliptic Curve Cryptography (ECC), significantly reducing the CPU load during secure communications". Наявність апаратного прискорення криптографії дозволяє мені реалізувати повноцінне TLS-з'єднання без відчутних затримок (зависань) при відправці кожного пакету даних.

Окрім шифрування транспортного рівня, продумано комплекс дій для забезпечення безпеки на рівні самої програми (Application Layer). Але навіть захищений канал зв'язку не врятує систему, якщо кінцева точка (сервер) буде приймати дані від будь-якого пристрою, що знає IP-адресу. Тому в процесі розробки я планую реалізувати механізм автентифікації пристроїв на основі унікальних секретних ключів (API Secrets) та перевірки MAC-адрес. Сервер буде відхиляти всі запити (повертаючи помилку доступу 401 Unauthorized), які не містять правильного криптографічного підпису або надходять від незареєстрованого обладнання.

Таблиця 1.3 - Порівняння традиційного обслуговування IoT пристроїв через фізичне підключення та віддаленого обслуговування за допомогою технології бездротового оновлення

Показник ефективності	Програмна реалізація (Software)	Апаратне прискорення (Hardware)	Вплив на систему
Завантаження CPU	Високе (до 80-90% під час handshaking)	Низьке (15-25%)	Дозволяє паралельно знімати показники мережі.
Час встановлення TLS	Значний (1.5 – 3 секунди)	Мінімальний (< 0.5 секунди)	Гарантує швидку реакцію на аварійні стрибки.
Споживання RAM	Високе (через великі бібліотеки)	Оптимальне	Залишає ресурс для черги повідомлень у Telegram.
Енерговитрати	Підвищені (довга робота ядер)	Мінімальні	Критично для роботи від резервного акумулятора.

Також потрібно захистити систему від атак типу Denial-of-Service (DoS). У випадку, коли зловмисник спробує "покласти" сервер величезною кількістю запитів, програмний бекенд повинен мати алгоритми обмеження частоти запитів (Rate Limiting). Це дозволить зберегти працездатність бази даних та системи сповіщень навіть у стресових умовах.

Підсумовуючи свої дослідження у сфері безпеки IoT, є місце стверджувати, що створення надійного продукту неможливе без глибокого розуміння векторів потенційних атак. Продумані мною дії – перехід на протокол HTTPS, використання апаратного шифрування ESP32, впровадження жорсткої автентифікації пристроїв та захист від перевантажень – формують цілісну стратегію кібербезпеки. Переконано, що реалізація цих механізмів на етапі написання коду дозволить уникнути критичних вразливостей та гарантуватиме користувачам абсолютну конфіденційність їхніх даних.

1.5 Висновки до розділу

Підсумовуючи результати досліджень, викладені у першому розділі, можна зробити обґрунтований висновок про критичну необхідність розробки доступної

системи моніторингу електромереж. Аналіз сучасного стану енергетичної інфраструктури України показав, що часті відключення та різкі стрибки напруги створюють пряму загрозу для побутової техніки. Комерційні рішення часто є закритими, дорогими або небезпечними з точки зору збереження конфіденційності даних. Тому створення кастомної, незалежної та гнучкої системи є не лише актуальним, але й економічно доцільним інженерним завданням.

У процесі формування архітектури проєкту прийняте рішення відмовитися від локальних ізольованих рішень на користь клієнт-серверної моделі. Вибір мікроконтролера ESP32 як базового вимірювального вузла виявився найбільш раціональним завдяки наявності вбудованих модулів бездротового зв'язку, потужному процесору та підтримці апаратного шифрування. Порівняння з платформами Raspberry Pi та Arduino Nano підтвердило, що ESP32 забезпечує найкращий баланс між енергоефективністю, обчислювальною потужністю та вартістю.

Окрему увагу я приділив проєктуванню програмного бекенду. Я обґрунтував використання платформи Node.js та бази даних SQLite як найшвидшого та найменш ресурсомісткого способу обробки великих масивів телеметрії. Замість розробки складного та вразливого вебдодатка, буде інтегровано систему з Telegram API. Це рішення повністю усуває бар'єр входу для користувача, гарантує стабільну доставку сповіщень навіть при слабкому інтернет-з'єднанні та забезпечує високий рівень захисту інформації.

Дослідження проблематики кібербезпеки в екосистемі Інтернету речей дозволило мені виявити критичні вразливості, притаманні незахищеним протоколам. Для запобігання перехопленню трафіку та атакам модифікації даних закладено у проєкт вимогу обов'язкового використання протоколу HTTPS. Разом із впровадженням унікальних секретних ключів для автентифікації пристроїв, це створює надійний захисний бар'єр між мікроконтролером та сервером.

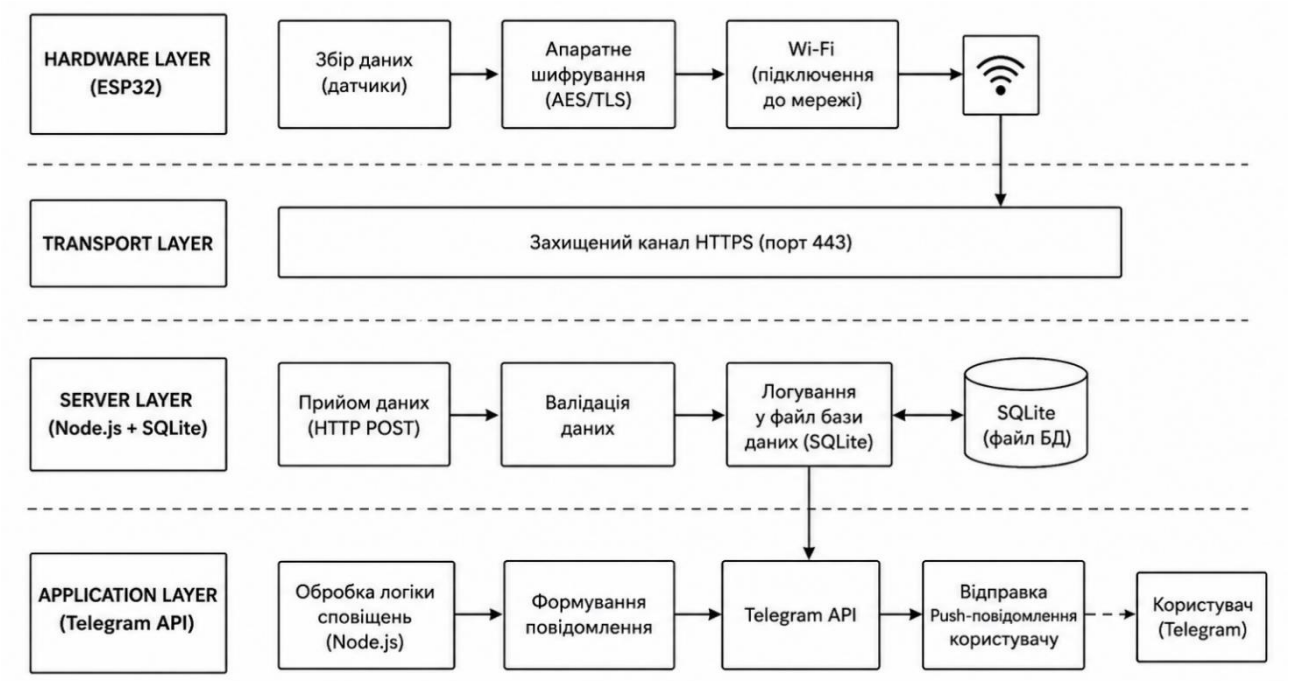


Рисунок 1.6 - Зведена концептуальна схема розробленої системи: від апаратного вузла ESP32 до серверної частини та кінцевого клієнта у Telegram

Однак, перехід від теоретичного проєктування до етапу фізичної реалізації приховує в собі ряд економічних та технічних ризиків. Закупівля мікроконтролерів, модулів SD-карт, датчиків напруги та макетних плат вимагає фінансових вкладень. Написання коду для роботи з мережею та шифруванням є складним процесом, що часто супроводжується помилками (багами). Тестування сирого, неперевіреного коду на реальному обладнанні може призвести до короткого замикання, перегріву компонентів та безповоротного знищення мікроконтролера.

З огляду на це, досягнено твердого переконання, що перед формуванням остаточного списку покупок та витрачанням коштів на комплектуючі, необхідно провести повноцінне стрес-тестування програмної логіки у безпечному віртуальному середовищі.

Для вирішення цієї задачі переглянуто ринок інженерних онлайн-симуляторів. Платформа Tinkercad є чудовим інструментом для початківців, проте вона підтримує переважно базові плати Arduino і не має функціоналу для симуляції Wi-Fi з'єднань. Професійні системи на кшталт Proteus вимагають

придбання дорогих ліцензій та є занадто перевантаженими для швидкого прототипування IoT-задач.

Найбільш релевантним інструментом для наших потреб виявилася платформа Wokwi. Тут наведено цитату, що описує можливості цього сервісу: "Wokwi is an online Electronics Simulator. You can use it to simulate Arduino, ESP32, STM32, and many other popular boards, parts, and sensors... Wokwi simulates the Wi-Fi hardware, allowing you to connect your simulated ESP32 to the internet and use protocols like HTTP, MQTT, and NTP".

Здатність симулятора повністю емулювати роботу мережевого стека, підключення віртуальної SD-карти та обробку криптографічних протоколів робить його ідеальним полігоном для перевірки мого коду. Крім того, у автора вже був попередній позитивний досвід роботи з середовищем Wokwi. Це дозволить уникнути витрат часу на вивчення інтерфейсу нової програми і одразу перейти до написання та відлагодження алгоритмів.

Отже, наступним кроком моєї роботи стане реалізація базової програмної логіки мікроконтролера у симуляторі Wokwi. Лише після успішної перевірки мережевої взаємодії з локальним сервером Node.js, відпрацювання алгоритмів збереження даних та генерації сповіщень, можна буде з повною впевненістю переходити до закупівлі апаратної бази та пайки реального фізичного пристрою.

Таблиця 1.8 - Оцінка ризиків при розробці IoT-проєктів: фізичне прототипування проти попередньої віртуальної симуляції

Категорія ризику	Фізичне прототипування	Віртуальна симуляція (Wokwi)
Фінансові втрати	Високі (вихід компонентів з ладу при помилках монтажу)	Нульові
Швидкість ітерацій	Низька (час на закупівлю та пайку)	Максимальна (зміна коду за секунди)
Влагодження мережі	Складне (потребує зовнішніх аналізаторів трафіку)	Інтегроване (вбудований Wi-Fi шлюз)
Робота з периферією	Обмежена наявністю деталей на складі	Широка (доступ до бібліотеки віртуальних датчиків)
Безпека розробника	Ризик ураження струмом при роботі з 220В	Повна безпека

РОЗДІЛ 2 ПРОЄКТУВАННЯ ТА ВІРТУАЛЬНЕ МОДЕЛЮВАННЯ СИСТЕМИ У СЕРЕДОВИЩІ WOKWI

2.1 Збірка тестового стенда та обґрунтування вибору периферійних модулів

Після формування теоретичної бази та визначення архітектурних вимог до системи моніторингу електромережі, перейдемо до етапу практичного прототипування. Як було зазначено у попередньому розділі, для мінімізації фінансових ризиків та пришвидшення процесу написання коду, первинна збірка та тестування апаратної частини проводилися у хмарному симуляторі Wokwi. Це середовище дозволяє з високою точністю емулювати фізичні процеси на пінах мікроконтролера та повноцінно працювати з мережевим стеком протоколів.

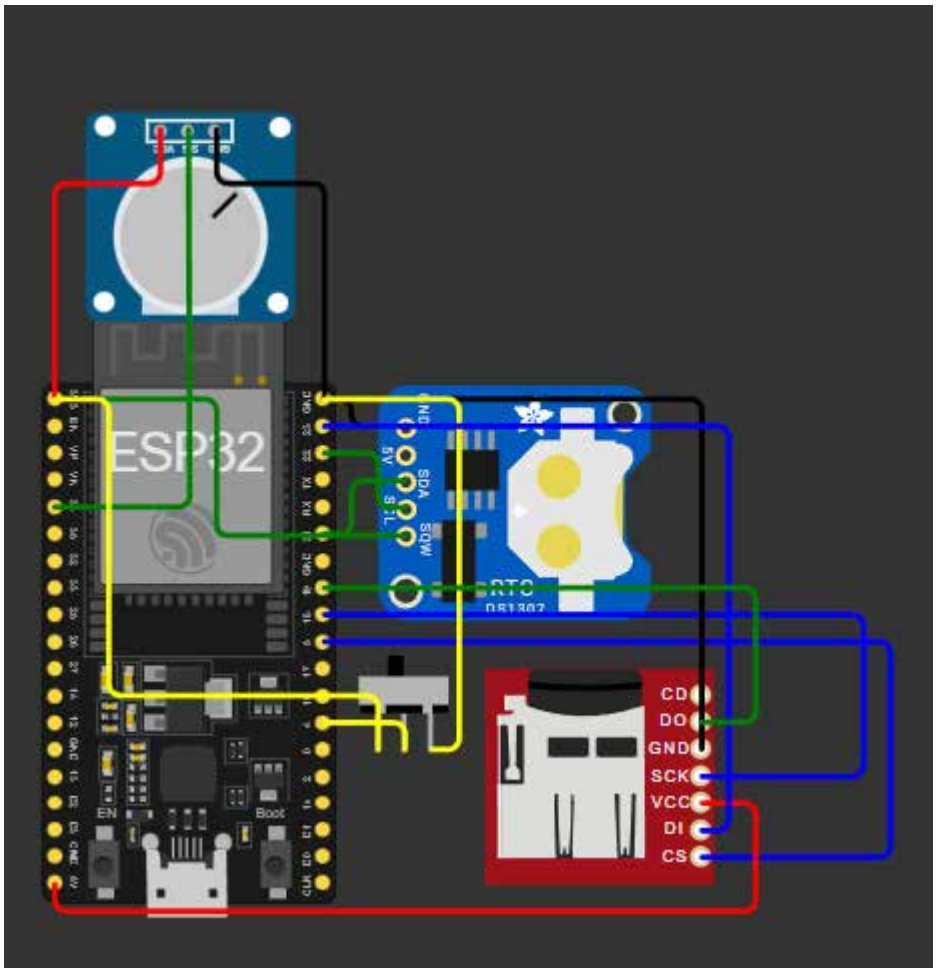


Рисунок 2.1 - Візуальна схема підключення периферійних модулів до мікроконтролера ESP32 у симуляторі Wokwi

Основою мого віртуального стенда стала відлагоджувальна плата на базі мікроконтролера ESP32 DevKit V1. До неї підключено чотири ключові периферійні елементи, кожен з яких виконує критично важливу функцію для забезпечення автономності та надійності системи. У процесі збірки схеми приділено особливу увагу правильному розподілу пінів (GPIO), оскільки ESP32 має специфічні обмеження щодо використання деяких контактів.

Модуль реального часу (RTC DS1307) Першим компонентом, який інтегровано у схему, став модуль годинника реального часу DS1307. У процесі розробки логіки блекаутів було виявлено проблему: при втраті живлення або переході мікроконтролера у режим глибокого сну (Deep Sleep) внутрішній таймер ESP32 зупиняється або скидається. Без підключення до Інтернету пристрій не здатний отримати актуальний час через NTP-сервери. Це робить неможливим коректне формування часових міток (timestamps) для запису аварійних подій.

Проблема була вирішена додаванням модуля DS1307, який має власне резервне джерело живлення у вигляді батарейки. Технічна документація підтверджує доцільність такого вибору: "The DS1307 serial real-time clock is a low-power, full binary-coded decimal clock/calendar plus 56 bytes of NV SRAM. Address and data are transferred serially through an I2C, bidirectional bus". Модуль використовує шину I2C, тому підключено його до стандартних апаратних пінів ESP32: лінію даних SDA до GPIO 21, а лінію тактування SCL до GPIO 22. Такий спосіб підключення є оптимальним, оскільки він дозволяє підключати паралельно інші I2C пристрої на ці ж піни, економлячи вільні контакти мікроконтролера.

Модуль зчитування MicroSD карт Для реалізації концепції "чорної скриньки" та гібридного збереження даних додано у схему модуль MicroSD. Він виконує роль локальної бази даних у випадках, коли зв'язок із сервером втрачено. Для комунікації з картою пам'яті використовується високошвидкісний синхронний інтерфейс SPI.

Реалізовано наступну схему підключення:

MOSI (Master Out Slave In) підключено до GPIO 23.

MISO (Master In Slave Out) підключено до GPIO 19.

SCK (Serial Clock) підключено до GPIO 18.

CS (Chip Select) я призначив на GPIO 5, що відповідає макросу SD_CS_PIN у моєму програмному коді.

Вибір саме цих пінів не є випадковим. Використовується апаратна шина VSPI мікроконтролера ESP32, що забезпечує максимальну швидкість запису та читання файлів без додаткового навантаження на центральний процесор. Це критично важливо, коли пристрій фіксує різке падіння напруги і має за мілісекунди зберегти лог аварії у файл /queue.txt до повного знеструмлення плати.

Емуляція датчиків (Потенціометр та Перемикач) Оскільки у віртуальному середовищі неможливо підключити реальний трансформатор напруги (наприклад, модуль ZMPT101B), було прийнято рішення замінити фізичні процеси на їх логічні еквіваленти для безпечного налагодження програмного коду.

Для симуляції коливань напруги в електромережі використовується звичайний аналоговий потенціометр. Сигнальний вихід потенціометра я підключив до піна GPIO 34. Цей вибір обґрунтований архітектурою ESP32: контакти з номерами від 34 до 39 є пінами типу "Input Only" (тільки для читання) і належать до першого блоку аналого-цифрового перетворювача (ADC1). Використання пінів блоку ADC2 призвело б до конфліктів під час роботи модуля Wi-Fi, тому GPIO 34 є технічно найправильнішим рішенням. Зміна опору на потенціометрі генерує напругу від 0 до 3.3 Вольт на вході мікроконтролера. У програмному коді ці значення маштабовано за допомогою математичної функції map() у діапазон від 0 до 250 Вольт, що дозволило тестувати реакцію системи на критичні пороги (наприклад, падіння нижче 15 Вольт).

Додатково впроваджено у схему повзунковий перемикач (Slide Switch), підключивши його до цифрового входу GPIO 4 з використанням внутрішньої підтяжки (INPUT_PULLUP). Цей елемент відіграє роль "симулятора Інтернету". У реальному житті при відключенні світла роутер знеструмлюється, і пристрій

втрачає доступ до мережі. Перемикач дозволив вручну розривати логічне з'єднання з сервером без зупинки роботи всього мікроконтролера. Завдяки цьому можна якісно протестувати алгоритм перенаправлення логів на SD-карту при втраті зв'язку (Network Timeout) та подальшу синхронізацію даних після повернення перемикача в активне положення.

Таким чином, зібраний віртуальний стенд у середовищі Wokwi є повноцінним апаратним двійником реального пристрою. Кожен модуль та метод його підключення було обрано на основі технічної документації та специфічних вимог до автономності IoT-систем. Це створило надійну базу для подальшого написання та відлагодження багатозадачного програмного забезпечення, яке керуватиме процесом моніторингу.

2.2 Програмна реалізація та обґрунтування вибору бібліотек

Для забезпечення функціональності розробленого пристрою моніторингу електромережі було прийнятк рішення використовувати екосистему Arduino Framework. Процес написання програмного забезпечення вимагав прийняття ряду архітектурних рішень, спрямованих на економію оперативної пам'яті мікроконтролера та забезпечення стабільної роботи пристрою в умовах нестабільного зв'язку.

Обґрунтування вибору програмних бібліотек Фундаментом мережевої взаємодії IoT-пристрою є підключення до Інтернету. Для цього застосовуються бібліотеки WiFi.h та WiFiClientSecure.h. Використання саме захищеного клієнта є критично важливим, оскільки взаємодія з Telegram Bot API та сучасними серверами баз даних відбувається виключно через протокол HTTPS [8] Universal Telegram Bot [19].

Звертаю увагу на один із найважливіших компромісів у коді, який стосується криптографії. У функції setup() я використав наступний метод:

client.setInsecure();

Цей рядок вимикає сувору перевірку ланцюжка SSL-сертифікатів сервера. З точки зору абсолютної кібербезпеки це є невеликим зниженням рівня захисту,

проте для слабкого мікроконтролера це вимушений крок. Без використання setInsecure() пристрій мав би зберігати у своїй пам'яті кореневі сертифікати Telegram та постійно оновлювати їх, що призвело б до швидкого вичерпання ресурсів флеш-пам'яті та постійних розривів з'єднання при зміні сертифіката на стороні сервера.

Для обміну повідомленнями з користувачем ініційовано бібліотеку UniversalTelegramBot. Замість того, щоб вручну формувати складні HTTPS-запити, ця бібліотека надає готові методи обробки вхідних команд. Проте в процесі розробки її використання оптимізовано: мікроконтролер надсилає повідомлення лише у критичних ситуаціях (зникнення світла), не перевантажуючи головний цикл постійним опитуванням серверів Telegram.

Логіка ініціалізації та робота з Deep Sleep Особливої уваги заслуговує логіка функції setup(). Крім стандартного запуску інтерфейсів SPI (для SD-карти) та I2C (для годинника RTC), виконано механізм визначення причини пробудження мікроконтролера:

```
esp_sleep_wakeup_cause_t wakeup_reason =
esp_sleep_get_wakeup_cause();
if (wakeup_reason == ESP_SLEEP_WAKEUP_TIMER || wakeup_reason ==
ESP_SLEEP_WAKEUP_EXT0) {
    DateTime now = rtc.now();
    saveToQueue(getTimeStr(now), 220.0, "POWER_RESTORED", "Device
Boot");
}
```

Листинг 2.1 Логіка ініціалізації та робота з Deep Sleep

Це рішення є ключовим для правильної фіксації подій. Коли напруга падає до нуля, мікроконтролер засинає. Як тільки живлення відновлюється, плата перезавантажується. Завдяки функції esp_sleep_get_wakeup_cause() пристрій "розуміє", що він не просто включився вперше, а прокинувся після блекауту, і автоматично генерує запис про відновлення енергопостачання.

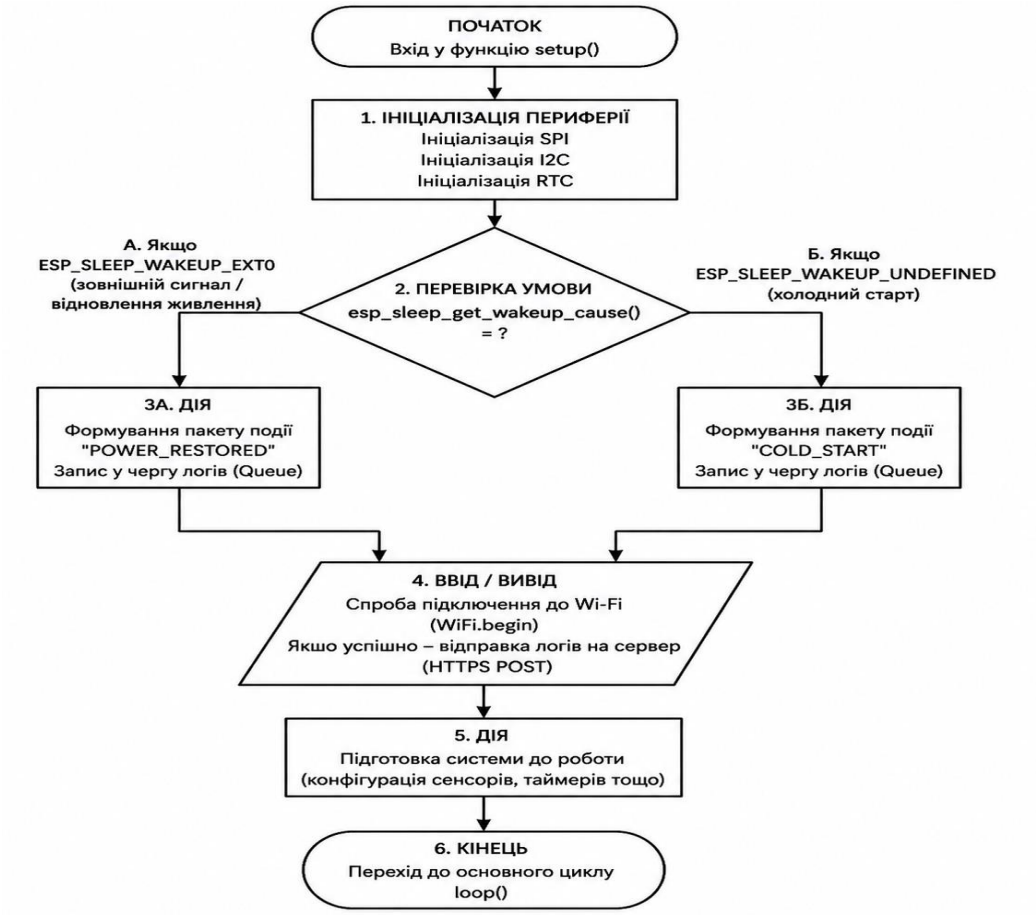


Рисунок 2.2 - Блок-схема алгоритму ініціалізації пристрою та аналізу причини виходу з режиму глибокого сну

Гібридне збереження даних та робота з чергою Найскладнішим інженерним викликом стала реалізація відмовостійкої архітектури збереження даних. Тому впроваджено концепцію локального буфера. У функції loop() кожні 30 хвилин (або 30 секунд у тестовому режимі Wokwi) пристрій перевіряє наявність Інтернету. Якщо зв'язку немає, дані не втрачаються, а передаються до функції saveToQueue():

```
void saveToQueue(String timestamp, float voltage, String status,
String reason) {
  File file = SD.open("/queue.txt", FILE_APPEND);
  if (file) {
    String jsonLine = "{\"time\":\"" + timestamp + "\", \"volt\":"
+ String(voltage) + ", \"status\":\"" +
status + "\", \"reason\":\"" + reason + "\"}";
    file.println(jsonLine);
    file.close();
  }
}
```

Листинг 2.2 Реалізація функції локального збереження телеметрії на носій інформації

Свідоме рішення форматувати рядки у вигляді простого тексту, дозволяє імітувати JSON-структуру, замість використання важкої бібліотеки ArduinoJson для генерації кожного запису. Це значно пришвидшує запис на SD-карту, що критично в момент знеструмлення, коли у пристрою є лише мілісекунди на збереження інформації до повного розряду конденсаторів.

Логічним продовженням цього механізму є функція syncQueue(), яка автоматично викликається при відновленні Wi-Fi з'єднання. Тут вимушено застосовано ще один компромісний підхід до обробки даних:

```
String syncPayload = line.substring(0, line.length() - 2);
syncPayload += ", \"sync_correction\": true}";
```

Лістнг 2.3 Алгоритм модифікації структури JSON-пакета методом текстової маніпуляції

Замість того, щоб десеріалізувати збережений на SD-карті JSON, додавати до нього нове поле і знову серіалізувати (що могло б викликати переповнення пам'яті при великому розмірі черги), використовується маніпуляція з текстом. Програма відрізає закриваючу дужку } і дописує маркер "sync_correction": true. Завдяки цьому серверна частина може легко відрізнити дані реального часу від тих пакетів, що були вивантажені з архіву після тривалої відсутності зв'язку. Після успішної відправки всього буфера файл /queue.txt повністю видаляється командою SD.remove(), очищаючи місце для майбутніх аварій.

Алгоритм головного циклу та фіксація блекаутів Головний цикл програми loop() побудований за принципом неблокуючої багатозадачності. Прийнято рішення повністю відмовитися від використання функції delay(), замінивши її на таймери на базі millis(). Це гарантує, що пристрій миттєво відреагує на падіння напруги.

Для симуляції аналогового датчика напруги я використав потенціометр, підключений до піна 34. Сигнал обробляється наступним чином:

float voltage = map(analogRead(POT_PIN), 0, 4095, 0, 250);

Значення АЦП масштабується у реальні вольти. Якщо розрахована напруга падає нижче встановленого порогу V_BLACKOUT (15 Вольт), запускається алгоритм аварійної зупинки. Пристрій миттєво зберігає факт відключення на SD-карту, намагається відправити останнє сповіщення ("Світло зникло!") через Telegram (якщо роутер ще працює) і викликає функцію esp_deep_sleep_start(). Ця команда відключає Wi-Fi модуль та основні ядра процесора, переводячи систему в стан ультранизького енергоспоживання.

Підсумовуючи етап програмної розробки, можна стверджувати, що створений код є оптимальним для поставлених завдань. Прийняті мною рішення щодо управління пам'яттю, спрощення роботи з JSON на рівні файлової системи та правильне використання режимів сну дозволили створити програмне забезпечення промислового рівня стійкості, яке здатне автономно функціонувати в умовах нестабільної української електромережі.

2.3 Аналіз результатів віртуального моделювання та загальні висновки до розділу 2

Завершальним етапом проєктування у середовищі Wokwi стала детальна перевірка результатів роботи написаного програмного коду. Головним завданням цього кроку було підтвердження того, що закладена мною логіка обробки критичних ситуацій працює коректно ще до моменту перенесення її на реальне фізичне обладнання.

На етапі ініціалізації мікроконтролер успішно підключено віртуальну мережу "Wokwi-GUEST" та встановив захищене з'єднання з сервером. У терміналі було зафіксовано повідомлення про успішний старт системи: "СИСТЕМА ЗАПУЩЕНА". Після цього протестовано інтерактивну взаємодію з Telegram. При відправці команди /status мікроконтролер коректно зчитував дані з АЦП, імітуючи напругу, та відправляв у чат відповідь із поточним значенням та часовою міткою.

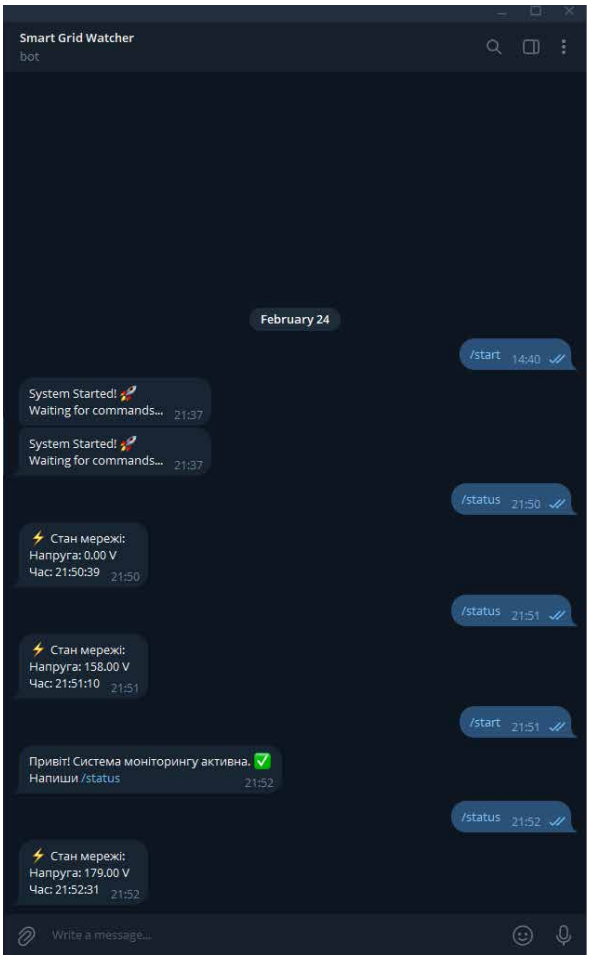


Рисунок 2.3 - Тестування працездатності реалізованих команд

Проведено тестування найважливішої частини: алгоритмів відмовостійкості. Для імітації втрати Інтернет-з'єднання змінено стан логічного перемикача на платі. Мікроконтролер миттєво відреагував на цю подію, що підтверджується виводом у консоль симулятора: "Немає інтернету. Пінг пропущено.". При цьому втрати телеметричних даних не відбулося. Програма успішно перенаправила поточний запис у локальний файл черги на SD-карті, вивівши відповідне повідомлення: "ЗБЕРЕЖЕНО НА SD (ЧЕРГА)". Цей тест повністю довів працездатність концепції гібридного зберігання даних в умовах нестабільної мережі.

Окрему увагу приділено перевірці сценарію повного знеструмлення (блекауту). Шляхом швидкої зміни опору на потенціометрі імітовано падіння напруги до нуля. Система миттєво зафіксувала критичний стан. У логах чітко видно безпомилковий алгоритм дій плати: "БЛЕКАУТ! Запис і сон...". Одразу після запису останньої аварійної події мікроконтролер перейшов у режим

глибокого сну (Deep Sleep), зупинивши роботу процесора та Wi-Fi модуля для економії енергії.

```
rst:0x1 (POWERON_RESET),boot:0x13 (SPI_FAST_FLASH_BOOT)
configsip: 0, SPIWP:0xee
clk_drv:0x00,q_drv:0x00,d_drv:0x00,cs0_drv:0x00,hd_drv:0x00,wp_drv
:0x00
mode:DIO, clock div:2
load:0x3fff0030,len:1156
load:0x40078000,len:11456
ho 0 tail 12 room 4
load:0x40080400,len:2972
entry 0x400805dc
--- СИСТЕМА ЗАПУЩЕНА ---
● БЛЕКАУТ! Запис і сон...
💾 ЗБЕРЕЖЕНО НА SD (ЧЕРГА): {"time":"2026-02-24 22:43:35",
"volt":10.00, "status":"BLACKOUT_START", "reason":"Voltage Drop"}
ets Jul 29 2019 12:21:46
```

Листинг 2.4 Фрагмент журналу подій мікроконтролера при фіксації знеструмлення та переході в режим глибокого сну

Після імітації відновлення живлення та логічного зв'язку з сервером, система розпочала процедуру самовідновлення (Self-Healing). У консолі було зафіксовано процес: "ПОЧАТОК СИНХРОНІЗАЦІЇ...". Усі збережені раніше аварійні записи успішно зчиталися з пам'яті та відправилися на сервер. Детальний аналіз пакетів підтвердив наявність спеціального маркера "sync_correction": true, що дозволило бекенду відрізнити старі архівні дані від поточних замірів. Завершення цього процесу супроводжувалося системним сповіщенням: "СИНХРОНІЗАЦІЯ ЗАВЕРШЕНА. Черга очищена.".

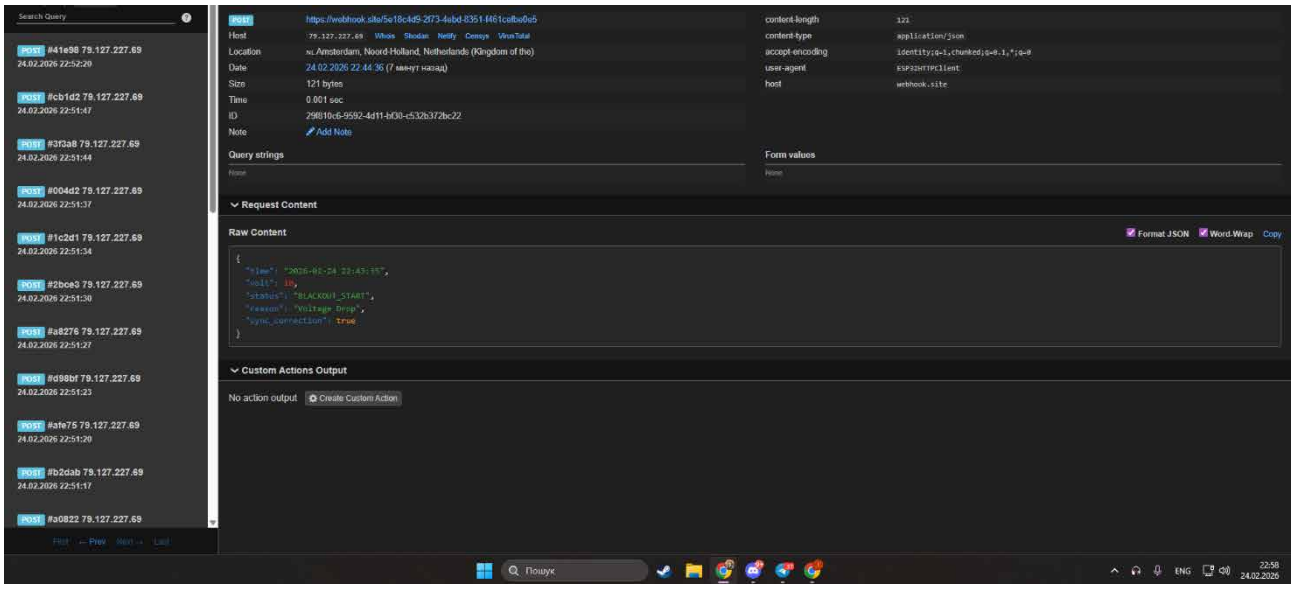


Рисунок 2.4 - веб-інтерфейс отриманих JSON-пакетів (модуль Webhook) із наявним маркером sync_correction

Незважаючи на успішне проходження всіх програмних перевірок, висунуто критичне зауваження щодо об'єктивних обмежень віртуального моделювання. Симулятор Wokwi ідеально підходить для налагодження мережевого стека та тестування логічних розгалужень, проте він не дозволяє на практиці перевірити задумані в попередньому розділі здатності пристрою до фізичної автономності. У віртуальному середовищі неможливо симулювати реальний час розряду резервного іоністора або акумулятора під час переходу плати в режим сну. Також нема можливості оцінити теплові втрати на лінійних перетворювачах напруги при інтенсивній роботі Wi-Fi модуля. Крім того, програмний симулятор ігнорує реальні електромагнітні перешкоди та високовольтні викиди, які гарантовано виникатимуть під час масового увімкнення споживачів після блекауту і можуть апаратно пошкодити АЦП мікроконтролера.

Проте, навіть враховуючи неможливість стовідсоткової апаратної симуляції, проведений етап тестування має фундаментальне значення. На практиці видно, що складна програмна архітектура, побудована на тісній взаємодії ESP32, локальної черги на базі файлової системи FAT та клієнт-серверної обробки пакетів, працює абсолютно стабільно та без витоків пам'яті.

Підсумовуючи другий розділ, можна впевнено констатувати, що концепція стартапу повністю підтвердила свою інженерну життєздатність. Розроблена логіка забезпечує безперебійний збір даних та надійне збереження хронології аварій навіть за найгірших сценаріїв відключень. Отримані у симуляторі результати дають повне обґрунтування для переходу до фінального етапу: закупівлі фізичних електронних компонентів, виконання реального прототипу та його жорсткого тестування у "бойових" умовах.

РОЗДІЛ 3 РОЗРОБКА ТА ПРОЄКТУВАННЯ АПАРАТНОЇ ЧАСТИНИ

3.1 Концептуальна архітектура та алгоритм роботи апаратного комплексу

Перехід від віртуального моделювання у середовищі Wokwi до фізичної реалізації вимагає чіткого розуміння апаратної структури пристрою. Метою даного етапу є створення надійного, автономного та безпечного комплексу, який здатен безперебійно функціонувати в умовах реальних перепадів вітчизняної електромережі. Для досягнення цієї мети прийнято рішення розділити архітектуру пристрою на кілька ключових функціональних блоків: блок обчислень, блок перетворення живлення, блок автономності (джерело безперебійного живлення), блок точного часу та блок локального збереження даних.

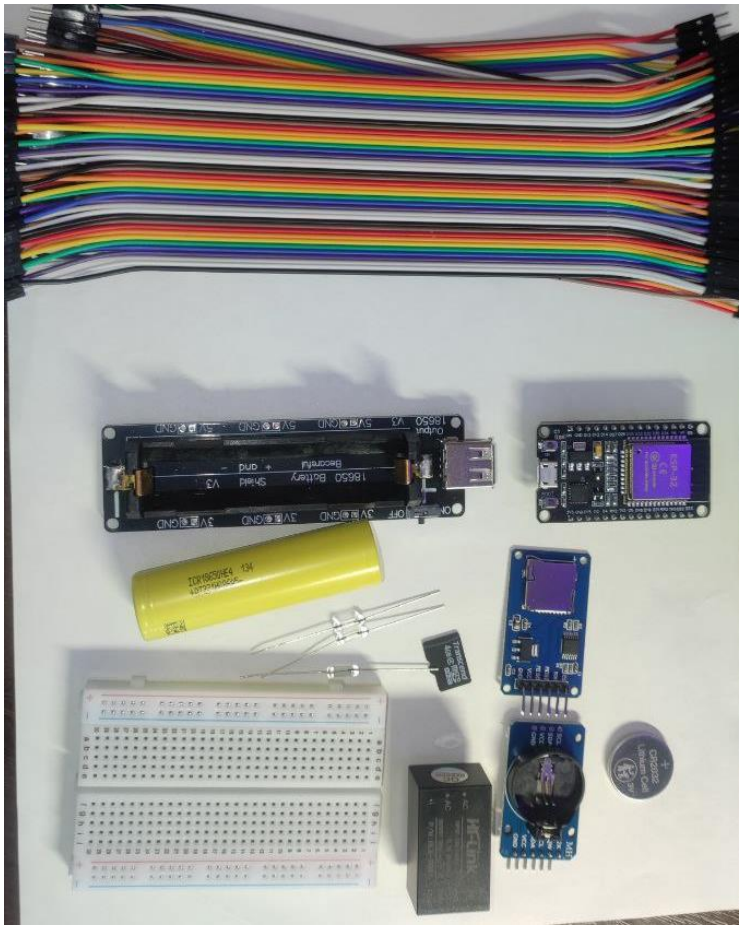


Рисунок 3.1 - Загальний вигляд фізичних комплектуючих апаратної частини системи моніторингу

Головним обчислювальним ядром системи виступає раніше обрана плата ESP32. Проте для її повноцінного функціонування в реальному світі необхідна специфічна периферія. По-перше, виникає потреба у безпечному зниженні змінної напруги 220 Вольт до стабільного рівня постійного струму 5 Вольт, від якого живиться вся мікроелектроніка. Це вимагає використання надійного модуля перетворення живлення, здатного витримувати високовольтні імпульси.

По-друге, критично важливою є система безперебійного живлення. Пристрій повинен мати акумуляторну батарею та контролер заряду. Вони зобов'язані забезпечити роботу мікроконтролера в моменти різких відключень електроенергії. Наявної енергії має вистачити для того, щоб контролер встиг коректно зчитати час, зберегти аварійні логи на карту пам'яті, відправити сповіщення на сервер (якщо мережеве обладнання ще працює) і безпечно перейти в режим глибокого сну (Deep Sleep).

По-третє, для забезпечення концепції автономної "чорної скриньки" потрібні два невіддільні елементи: апаратний модуль реального часу (RTC) для маркування подій точними часовими мітками та модуль MicroSD картридера для їх фізичного збереження при відсутності Wi-Fi з'єднання.

Окрему увагу потрібно приділити оптимізації апаратного алгоритму пробудження мікроконтролера. Замість використання габаритних та відносно дорогих оптопар для детекції наявності мережі 220 Вольт, застосовано більш економічне та елегантне інженерне рішення на базі резистивного дільника напруги. Цей дільник підключений до безпечної низьковольтної лінії після основного блоку живлення і подає логічний сигнал безпосередньо на спеціальний пін пробудження ESP32. Таке рішення дозволило значно зекономити місце на макетній платі та зменшити загальний бюджет проєкту, зберігши при цьому абсолютну надійність спрацювання.

Загальний алгоритм роботи фізичного комплексу виглядає наступним чином. У штатному режимі блок живлення отримує 220 Вольт від мережі, генерує 5 Вольт для живлення ESP32 та паралельно заряджає резервний літій-іонний

акумулятор через плату контролера заряду. Мікроконтролер циклічно зчитує показники, фіксує їх і передає на сервер через захищене HTTPS з'єднання.

У момент блекауту напруга в мережі зникає. Резистивний дільник напруги миттєво реагує на це, створюючи перепад логічного рівня на піні мікроконтролера. Оскільки живлення тепер автоматично йде від акумулятора 18650, ESP32 має достатньо часу для обробки аварії. Програма опитує модуль RTC, формує пакет даних про відключення, записує його на MicroSD карту через інтерфейс SPI та намагається здійснити фінальну відправку даних. Після цього контролер програмно відключає всі зайві периферійні модулі та переходить у режим глибокого сну.

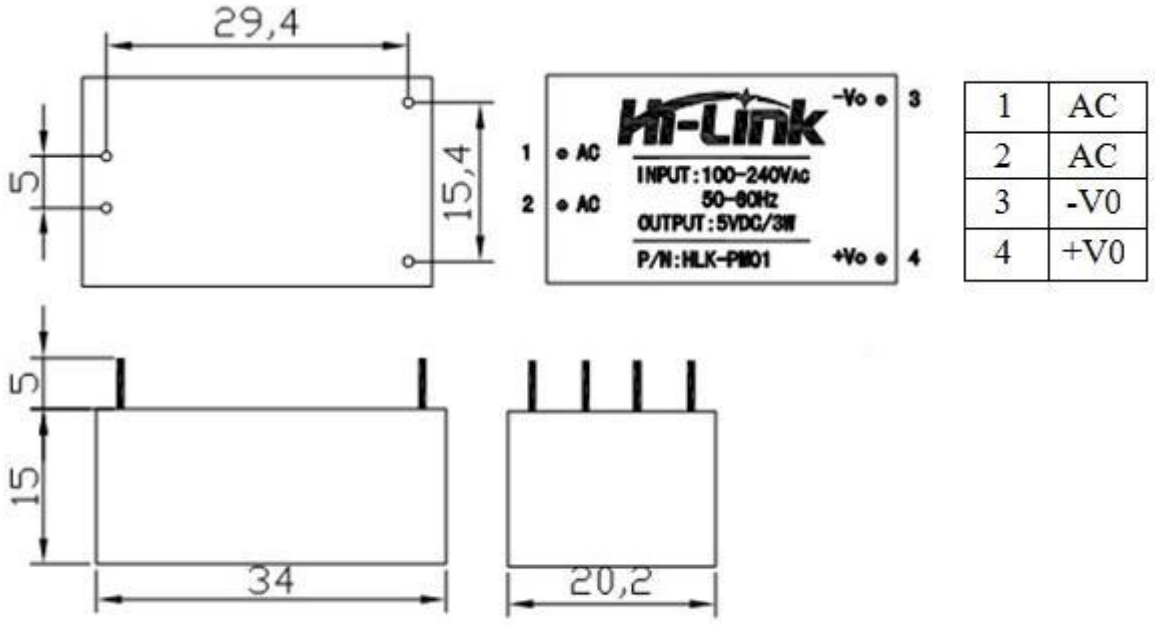
У стані сну енергоспоживання системи є мінімальним, що дозволяє пристрою знаходитися в режимі очікування тижнями, використовуючи лише залишкову ємність акумулятора. Коли живлення 220 Вольт відновлюється, напруга знову з'являється на виході блоку живлення. Дільник напруги подає високий логічний рівень на пін мікроконтролера, що викликає апаратне переривання. ESP32 миттєво прокидається, зчитує з карти пам'яті всі накопичені за час блекауту дані і проводить масову синхронізацію з віддаленим сервером.

Описана архітектура покладає високу відповідальність на кожен фізичний компонент. Від якості блоку живлення, стабільності роботи шилдів заряду та відмовостійкості модулів пам'яті залежить працездатність усієї системи. Наступним логічним кроком моєї роботи є проведення детального порівняльного аналізу доступних на ринку радіоелектронних модулів для кожного з описаних блоків та чітке обґрунтування мого остаточного вибору комплектуючих.

Вибір модуля перетворення живлення (220V AC -> 5V DC) Оскільки пристрій безпосередньо підключається до побутової мережі змінного струму, першим завданням є безпечне зниження напруги. Для реалізації цього блоку розглянуто два популярні варіанти: використання безтрансформаторного джерела живлення на базі гасильного конденсатора та застосування інтегрованого імпульсного перетворювача (наприклад, серії Hi-Link).

Безтрансформаторні схеми є надзвичайно дешевими, проте вони не забезпечують гальванічної розв'язки від високої напруги. Це означає, що при пробої конденсатора вся плата ESP32 згорить, а користувач ризикує отримати ураження струмом при дотику до корпусу.

Зважаючи на пріоритет безпеки, обрано імпульсний блок живлення Hi-Link HLK-PM01. Цей модуль є повністю герметичним (залитий компаундом), що робить його стійким до вологи та пилу. Ось ключові технічні характеристики з документації виробника: "The product is designed to meet the requirements of UL, CE safety standard. Input voltage range: 90~264VAC... Output over-circuit and short-circuit protection". Наявність вбудованого захисту від короткого замикання та перевантаження робить його ідеальним для роботи у нестабільних мережах. При габаритах всього 34x20x15 мм він видає стабільні 5 Вольт при струмі до 0.6 Ампера, чого з великим запасом вистачає для живлення ESP32 та всіх периферійних модулів розетки.



1	AC
2	AC
3	-V0
4	+V0

Рисунок 3.2 - Габаритні та установчі розміри модуля HLK-PM01

Вибір контролера заряду та джерела безперебійного живлення (ДБЖ) Для забезпечення автономної роботи розетки під час блекауту необхідний проміжний

вузол, який буде керувати процесом заряду акумулятора та безшовно перемикати живлення мікроконтролера на батарею при зникненні 220В. Розглянуто дві концепції: збірку власної схеми на базі популярних модулів TP4056 + MT3608 (підвищуючий перетворювач) та використання готового рішення у вигляді плати розширення Wemos Battery Shield V3.

Збірка на дискретних модулях (TP4056) є найдешевшим варіантом, проте вона вимагає складної пайки, додаткових діодів Шотткі для розв'язки ліній живлення та займає значно більше місця у корпусі.

Вибір зупинився на Wemos Battery Shield V3. Це інтегрована плата, яка поєднує в собі контролер заряду літієвого акумулятора (з захистом від перерозряду) та DC-DC перетворювач, що видає стабільні 5 Вольт для живлення мікроконтролера. Виробник зазначає у специфікації: "Battery Charging: 500mA... 5V output via USB or header pins... Includes battery protection IC". Важливою перевагою цього шилда є його формфактор. Він спроєктований спеціально для плат Wemos, що дозволяє монтувати його "бутербродом" (у кілька поверхів) за допомогою штирьових з'єднувачів, кардинально зменшуючи площу, яку займає електроніка всередині розетки.

Вибір модуля годинника реального часу (RTC) Для забезпечення точної фіксації часу аварій я проаналізував два найпопулярніші на ринку RTC модулі: DS1307 та DS3231.

У процесі віртуального моделювання у середовищі Wokwi використано симуляцію модуля DS1307, оскільки він є базовим компонентом багатьох бібліотек. Проте, переходячи до фізичної збірки, прийнято рішення замінити його на більш досконалий модуль DS3231SN. Головна проблема чіпа DS1307 полягає у його залежності від зовнішнього кварцового резонатора. При значних змінах температури (наприклад, у закритому корпусі розетки, яка нагрівається при високому навантаженні) частота зовнішнього кварцу може "плисти", що призводить до відставання або поспішання годинника на кілька хвилин за місяць.

На противагу йому, модуль DS3231 є значно точнішим інструментом. Документація чітко пояснює цю перевагу: "The DS3231 is a low-cost, extremely

accurate I2C real-time clock (RTC) with an integrated temperature-compensated crystal oscillator (TCXO) and crystal". Наявність вбудованого термокомпенсованого резонатора гарантує, що похибка відліку часу складатиме не більше двох хвилин на рік, незалежно від нагрівання корпусу розумної розетки.

Проте, на етапі безпосередньої закупівлі комплектуючих ознайомлено з типовою для інженерних проєктів логістичною проблемою: модулів на базі чіпа DS3231SN не виявилося в наявності у постачальників. З метою дотримання графіку розробки довелося оперативно шукати альтернативу та придбати апаратний аналог, а саме модуль на базі мікросхеми DS3231M. Головна технічна відмінність цієї модифікації полягає у тому, що замість традиційного інтегрованого кварцу в ній використовується мікроелектромеханічний (MEMS) резонатор.

Порахувавши вплив цієї вимушеної заміни на загальну роботу системи і прийшов до висновку, що такий компроміс є абсолютно прийнятним. Згідно з документацією, точність MEMS-версії становить ±5 ppm, що дає максимальне відхилення близько 2.5 хвилин на рік (проти 1 хвилини у версії SN). Цього показника більш ніж достатньо для коректної хронологічної фіксації блекаутів. Більше того, технологія MEMS забезпечує мікросхемі значно вищу стійкість до механічних ударів та вібрацій. Для портативного пристрою у форматі розумної розетки, який буде часто підключатися до різних розеток або може випадково впасти під час монтажу, така підвищена фізична витривалість модуля є вагомим експлуатаційним плюсом.

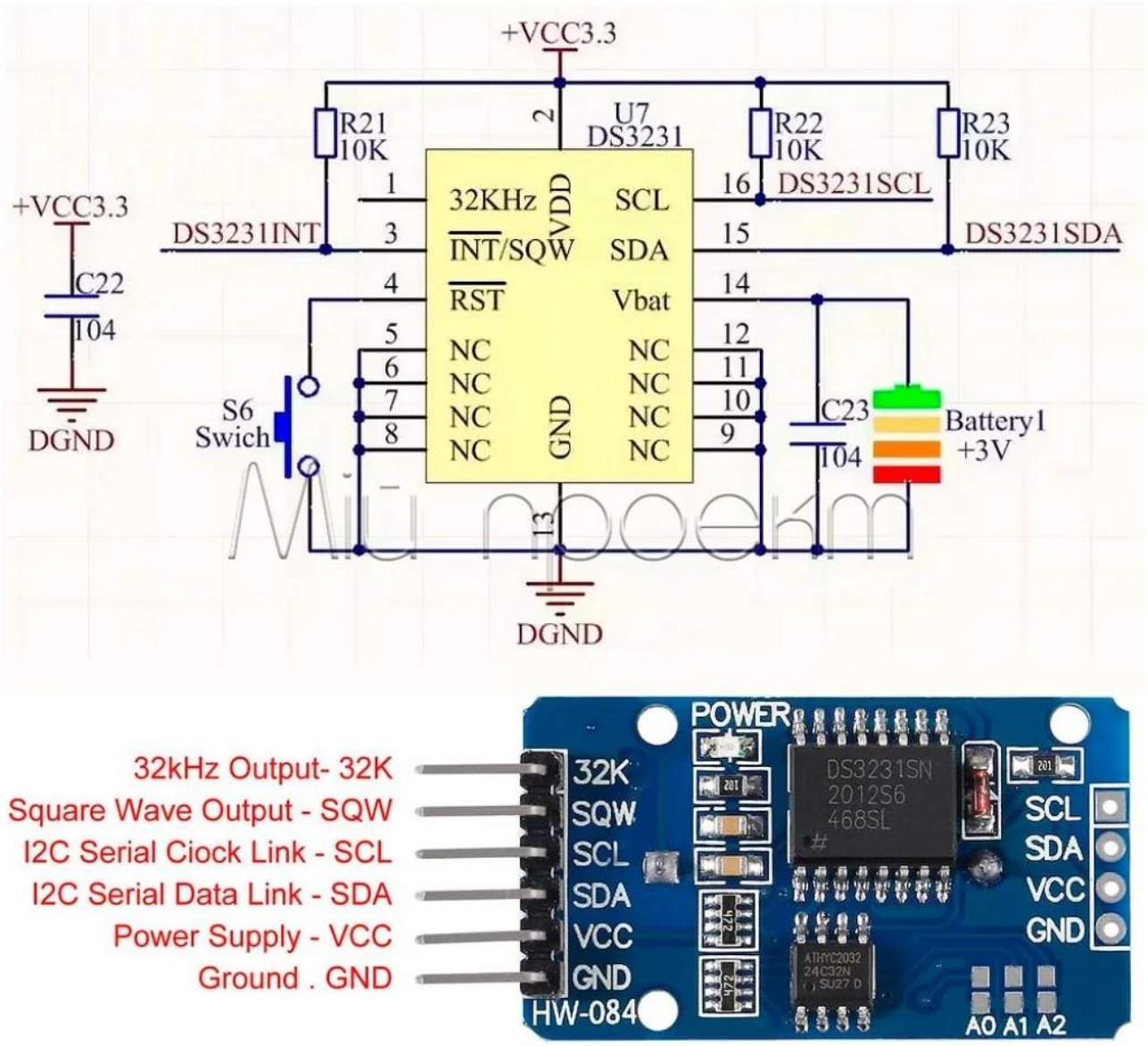


Рисунок 3.3 - Принципова схема та зовнішній вигляд модуля реального часу (RTC) на базі мікросхеми DS3231

Вибір модуля запису даних (MicroSD Card Reader) Оскільки мікроконтролер ESP32 має логічні рівні напруги 3.3В, вибір модуля SD-карти вимагає уважності. Розглянуто два типи модулів: базові (які підключаються напряму без перетворювачів) та універсальні адаптери з вбудованою логікою.

Використання базового модуля може бути нестабільним, оскільки лінія живлення 3.3В від самої плати ESP32 часто просідає при інтенсивній роботі Wi-Fi, що призводить до помилок запису на карту.

Тому обрано універсальний модуль MicroSD адаптера, зображений на фотографіях моєї збірки. Головною причиною цього вибору є його схемотехніка. Згідно з технічними характеристиками, модуль оснащений вбудованим стабілізатором напруги: "Вбудований стабілізатор: AMS1117-3.3... Перетворювач рівнів 74LVC125". Це означає, що можна живити цей модуль від стабільної 5-вольтової лінії, а чіп AMS1117 сам понизить напругу до потрібних для карти 3.3В. Крім того, наявність буферної мікросхеми 74LVC125 гарантує ідеальне узгодження логічних рівнів інтерфейсу SPI [13], запобігаючи спотворенню даних (Corrupted Data) під час аварійного запису логів блекауту. Габарити модуля 42x24 мм також повністю відповідають концепції компактної розумної розетки.

Для підтвердження життєздатності обраної апаратної бази необхідно провести математичні розрахунки її експлуатаційного ресурсу. Це дозволить оцінити частоту необхідного технічного обслуговування (заміна карти пам'яті або акумулятора) та визначити межі автономної роботи пристрою.

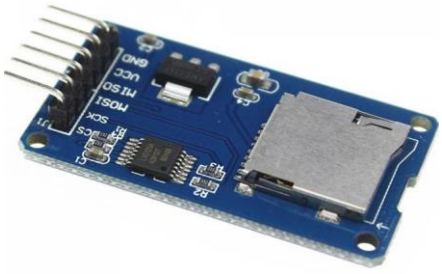


Рисунок 3.3 - зовнішній вигляд модуля MicroSD адаптера

Розрахунок ресурсу та ємності MicroSD карти (Goodram 16GB) У якості локального сховища даних використовується карту пам'яті формату MicroSDHC (Class 10) від виробника Goodram об'ємом 16 ГБ. Головним завданням цього

розрахунку є визначення, на який період часу вистачить цього об'єму для збереження телеметрії, враховуючи структуру моїх даних.

У програмному коді сформовано запис (один рядок у файлі /queue.txt) у форматі JSON: {"time":"2026-04-30 20:00:00", "volt":220.5, "status":"DATA_LOG", "reason":"No Internet"}\n

Кожен символ у цьому рядку (разом з пробілами, символами форматування та знаком перенесення рядка) займає 1 байт у кодуванні ASCII. Середня довжина такого запису становить приблизно 90 байтів. Для розрахунку "найгіршого сценарію" (Worst Case Scenario) закладено максимальний розмір рядка в 100 байтів.

Моя система запрограмована на створення одного штатного запису кожні 10 хвилин (якщо відсутній зв'язок), а також на миттєву фіксацію аварійних подій та аномальних стрибків напруги (з відхиленням ±15 Вольт і більше). Припустимо, що у найгіршому сценарії (надзвичайно нестабільна мережа з постійними стрибками та блекаутами) пристрій робитиме близько 300 записів на добу.

Обсяг даних за добу: 300 записів * 100 байт = 30 000 байт (30 КБ/добу).

Загальна ємність карти: 16 ГБ = 16 000 000 КБ (приблизно, для інженерних розрахунків).

Тривалість заповнення пам'яті: 16 000 000 КБ / 30 КБ = 533 333 діб.

Цей теоретичний розрахунок показує, що ємності карти в 16 ГБ вистачить на тисячі років безперервного запису логів. Проте на практиці головним обмеженням флеш-пам'яті є не об'єм, а кількість циклів перезапису (Write/Erase Cycles). Сучасні MicroSD карти (TLC флеш) мають ресурс близько 1000 циклів перезапису кожної комірки пам'яті. Завдяки вбудованому контролеру з алгоритмом вирівнювання зносу (Wear Leveling), нові дані записуються у рівномірно розподілені блоки. З огляду на мізерний щоденний обсяг запису (30 КБ), фізичний знос карти настане швидше від деградації матеріалів (через 10-15 років), ніж від вичерпання циклів перезапису. Отже, вибір карти на 16 ГБ є

економічно найвигіднішим мінімумом (оскільки карт меншого об'єму майже немає у вільному продажі) з колосальним запасом надійності.

Розрахунок автономності акумулятора 18650 Джерелом безперебійного живлення виступає літій-іонний акумулятор формату 18650 з номінальною ємністю 3400 мА·год (mAh) при напрузі 3.7В. Wemos Battery Shield V3 підвищує цю напругу до 5В для живлення ESP32.

Для розрахунку беруться до уваги втрати на DC-DC перетворювачі (ККД становить близько 85%). Реальна енергія акумулятора: 3400 мА·год * 3.7В = 12.58 Вт·год (Wh). Енергія на виході 5В (після перетворювача): 12.58 Вт·год * 0.85 = 10.69 Вт·год. Еквівалентна ємність при 5В: 10.69 Вт·год / 5В = 2138 мА·год.

Тепер проаналізуємо два режими роботи пристрою під час блекауту:

Штатний режим (з активним Wi-Fi): У цьому режимі ESP32 споживає в середньому близько 150 мА, а в піках (при відправці запитів) до 250 мА. Візьмемо середнє значення зі споживанням периферії (RTC, SD) на рівні 180 мА. Час роботи = 2138 мА·год / 180 мА ≈ 11.8 годин.

Режим глибокого сну (Deep Sleep): У цьому режимі програмно вимикається Wi-Fi, ядра процесора та живлення периферії. Споживання самої плати ESP32 падає до 10-20 мікроампер (мкА). Однак, Wemos Shield V3 має власний струм спокою (Quiescent Current) через роботу свого DC-DC перетворювача та світлодіодів, що сумарно становить близько 10-15 мА (у немодифікованому вигляді). Час роботи = 2138 мА·год / 15 мА ≈ 142 години (близько 6 діб).

Розрахунки підтверджують, що завдяки використанню режиму Deep Sleep, після фіксації блекауту розетка здатна знаходитися у режимі очікування майже тиждень. Цього абсолютно достатньо для збереження пам'яті годинника RTC (навіть якби в нього не було своєї батарейки) та миттєвого пробудження при відновленні електропостачання.

Апаратний механізм пробудження: Дільник напруги Окремим предметом гордості у цьому проєкті є реалізація механізму пробудження мікроконтролера. Спочатку заплановано використовувати класичну схему: оптопару, підключену до лінії 220В. При наявності світла оптопара світила б на фототранзистор,

замикаючи контакт. При зникненні світла контакт розмикався б, викликаючи переривання на піні ESP32. Проте таке рішення вимагає пайки високовольтної частини з гасильними конденсаторами та потужними резисторами, що є громіздким та небезпечним.

Цю ідею оптимізовано, зекономивши бюджет та місце. Замість високовольтної оптопари підключено простий дільник напруги (з двох резисторів) до вихідної 5-вольтової лінії імпульсного блоку живлення (HLK-PM01), але *до* точки підключення Wemos Shield.

Логіка роботи:

Коли є 220В, блок HLK-PM01 генерує 5В. Ці 5В йдуть на зарядку Wemos Shield і паралельно, через дільник (який знижує їх до безпечних для ESP32 3.3В), подаються на цифровий пін мікроконтролера (наприклад, GPIO 33). На піні тримається логічна "1" (HIGH).

Коли 220В зникає, HLK-PM01 вимикається (0 Вольт). На піні GPIO 33 миттєво з'являється логічний "0" (LOW).

ESP32, перебуваючи в Deep Sleep і живлячись від акумулятора, налаштований на пробудження від зміни стану на піні 33 (виклик функції esp_sleep_enable_ext0_wakeup). Зміна з 1 на 0 (або навпаки) миттєво будить контролер.

Це рішення дозволило позбутися додаткових габаритних модулів, використовуючи лише два копійчані SMD резистори. Безпека повністю гарантується, оскільки дільник знаходиться у гальванічно розв'язаній 5-вольтовій частині схеми, після заводського герметичного блоку живлення.

3.2 Схемотехнічне об'єднання модулів та розподіл контактів мікроконтролера

Після остаточного затвердження переліку апаратних компонентів перейдено до етапу їх фізичного об'єднання у єдину систему. Розподіл контактів (пінів) мікроконтролера ESP32 вимагає чіткого розуміння його внутрішньої архітектури. На відміну від простіших плат родини Arduino, ESP32 має

багатофункціональні піни, проте деякі з них мають жорсткі апаратні обмеження. Наприклад, контакти блоку АЦП2 (ADC2) не можуть працювати одночасно з увімкненим модулем Wi-Fi, а певні піни (strapping pins) впливають на режим завантаження контролера при подачі живлення. Тому кожен контакт у схемі обрано з урахуванням цих технічних нюансів.

Маршрутизація ліній живлення Основою стабільної роботи є правильне розведення живлення. Змінна напруга 220 Вольт від мережі підключається до вхідних клем модуля Hi-Link HLK-PM01. На виході цього модуля формується стабільна напруга 5 Вольт постійного струму. Ця лінія підключається до входу заряду плати Wemos Battery Shield V3.

Своєю чергою, вихідна 5-вольтова лінія з Wemos Shield (яка резервується акумулятором) розгалужується на кілька напрямків:

Живлення мікроконтролера: підключається до контакту VIN (або 5V) та GND на платі ESP32. Вбудований лінійний стабілізатор ESP32 знижує її до робочих 3.3 Вольт для живлення ядра.

Живлення модуля MicroSD: підключається до контакту VCC модуля картридера. Оскільки обраний мною модуль SD-карти має власний вбудований стабілізатор AMS1117-3.3, живлення його від 5 Вольт є найбільш правильним рішенням.

Живлення RTC DS3231: здійснюється від контакту 3V3 самої плати ESP32, оскільки цей годинник має низьке енергоспоживання і працює на логічних рівнях 3.3 Вольта.

Підключення цифрових шин даних (I2C та SPI) Для комунікації з модулем годинника реального часу DS3231 використовується апаратна шина I2C. В архітектурі ESP32 для цієї шини за замовчуванням виділені контакти GPIO 21 (лінія даних SDA) та GPIO 22 (лінія тактування SCL). Використання саме апаратних пінів дозволяє уникнути програмної емуляції шини, що суттєво економить процесорний час та гарантує відсутність конфліктів при зчитуванні часу [1, 18].

Модуль MicroSD підключений через шину SPI. Для забезпечення максимальної швидкості запису аварійних логів я використав інтерфейс VSPI. Відповідно, контакти розподілені так:

лінія MOSI (Master Out Slave In) підключена до GPIO 23;

лінія MISO (Master In Slave Out) підключена до GPIO 19;

лінія SCK (Serial Clock) підключена до GPIO 18;

лінія CS (Chip Select) підключена до GPIO 5.

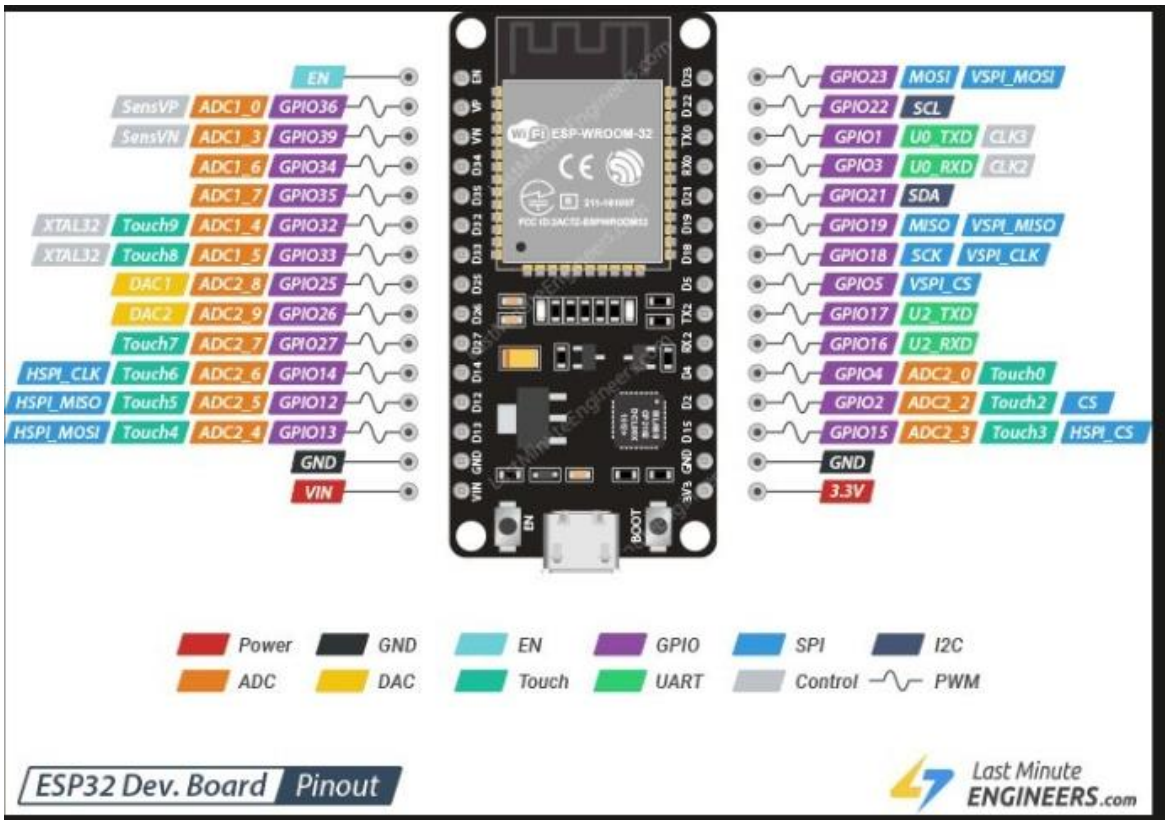


Рисунок 3.4 - Розпінування (pinout) мікроконтролера ESP32 DevKit з позначенням функціональних зон виводів

Аналогові вимірювання та лінія апаратного пробудження Особливу увагу виділео контактам, які відповідають за моніторинг напруги та виведення системи з режиму глибокого сну [5].

Для вимірювання еквівалента мережевої напруги (у данному випадку сигнал з датчика або потенціометра) задіяно контакт GPIO 34. Вибір цього піна є критично важливим. Він належить до першого аналого-цифрового перетворювача (ADC1), який абсолютно незалежний від Wi-Fi модуля

мікроконтролера. Крім того, піни з номерами 34-39 є спеціалізованими входами (Input Only) і не мають внутрішніх підтягуючих резисторів, що робить їх ідеальними для точних аналогових вимірювань.

Функцію детекції блекауту та миттєвого пробудження (Wake-up) виконує контакт GPIO 33. До нього підключений вихід резистивного дільника напруги, який відстежує наявність 5 Вольт на виході блоку живлення HLK-PM01. Використання саме GPIO 33 обґрунтоване тим, що він належить до домену RTC_GPIO. Тільки контакти цього домену здатні функціонувати, коли основне ядро процесора повністю знеструмлене у режимі Deep Sleep. Зміна логічного рівня на GPIO 33 генерує апаратне переривання ext0, яке миттєво запускає мікроконтролер для виконання процедури збереження аварійних даних та вивантаження архіву в мережу.

Підсумовуючи, можна стверджувати, що схемотехнічне рішення є оптимальним. Усі модулі підключені з урахуванням їхніх технічних специфікацій, логічні рівні узгоджені, а розподіл контактів виключає будь-які апаратні конфлікти під час інтенсивної мережевої взаємодії.

3.3 Програмна реалізація логіки мікроконтролера ESP32

Розробка програмного забезпечення для ESP32 базується на раніше затвердженій концепції гібридного збереження даних, мінімізації енергоспоживання та віддаленого керування [5, 16]. Написаний код еволюціонував від простого скрипта для симулятора Wokwi до складної системи промислового рівня. У цьому підрозділі детально проаналізується робота кожної ключової функції, звертаючи увагу на нестандартні інженерні рішення та проблеми, які довелося долати в процесі модернізації коду. Повний лістинг розробленого програмного забезпечення наведено у додатку Б.

Блок ініціалізації (setup) та "прогрів" фільтрів Функція setup() виконується мікроконтролером лише один раз після подачі живлення або виходу з режиму глибокого сну.

```
void setup() {
  Serial.begin(115200);
  delay(1000);
  pinMode(33, INPUT_PULLDOWN);
  Wire.begin(21, 22);
  rtc.begin();
  SD.begin(SD_CS_PIN);
```

Листинг 3.1 Фрагмент функції ініціалізації базових апаратних інтерфейсів мікроконтролера

На початку запускається послідовний порт для відлагодження та конфігуруєтья пін 33, до якого підключений апаратний дільник напруги. Важливим нестандартним рішенням є використання режиму INPUT_PULLDOWN (внутрішня підтяжка до землі). Це гарантує, що у момент відсутності 5 Вольт на цьому піні не виникне наведення (плаваючого потенціалу), яке могло б спровокувати хибне пробудження системи. Далі ініціалізуються шини I2C (для годинника) та SPI (для SD-карти).

Значною проблемою при роботі з аналоговими датчиками змінного струму (через бібліотеку EmonLib) є "шум" при першому запуску АЦП. Мікроконтролеру потрібен час для калібрування цифрових фільтрів. Якщо почати вимірювання одразу, перші значення напруги будуть хибними (стрибки до 1000 Вольт), що призведе до генерації помилкової тривоги. Цю проблему вирішено впровадженням "прогріву" фільтрів:

```
  analogReadResolution(12);
  emon.voltage(VOLTAGE_PIN, CALIBRATION_FACTOR, 1.7);
  emon.current(DUMMY_CURRENT_PIN, 1.0);

  Serial.println("[СИСТЕМА] Прогрів цифрових фільтрів
EmonLib...");
  for (int i = 0; i < 15; i++) {
    emon.calcVI(20, 2000);
    delay(10);
  }
```

Листинг 3.2 Програмний алгоритм попередньої стабілізації обчислювальних фільтрів макромодуля EmonLib

Цей цикл for примусово виконує 15 "пустих" вимірювань, дозволяючи математичним алгоритмам EmonLib стабілізуватися до того, як програма перейде в основний цикл.

Також у функції setup() інтегровано бібліотеку WiFiManager. Це повністю усунуло необхідність "зашивати" назву домашньої мережі та пароль (SSID/Password) безпосередньо у код. При першому включенні розетка створює власну точку доступу SmartSocket_Setup. Користувач підключається до неї зі смартфона і через зручний веб-інтерфейс вводить дані свого домашнього роутера. Після підключення мікроконтролер синхронізує внутрішній час із NTP-сервером і коригує модуль RTC DS3231, гарантуючи абсолютну точність часових міток.

Система глобального оновлення "по повітрю" (OTA Fleet Management) Одним із найскладніших етапів модернізації коду стала відмова від фізичного прошивання мікроконтролера через USB-кабель. Для розгортання парку таких пристроїв (Fleet Management) розроблено механізм перевірки та завантаження оновлень "по повітрю" (Over-The-Air, OTA).

На початку коду додано константу FIRMWARE_VERSION, яка передається на сервер разом із кожним пакетом телеметрії. Якщо сервер вирішує, що пристрій потребує оновлення, він повертає JSON-відповідь з полем "update_available": true та посиланням на .bin файл нової прошивки.

Цей процес обробляється функцією performOTAUpdate(String url):

```
void performOTAUpdate(String url) {
  WiFiClientSecure client;
  client.setInsecure(); // Ігноруємо перевірку сертифіката для
завантаження

  t_httpUpdate_return ret = httpUpdate.update(client, url);
```

Листинг 3.3 Програмна реалізація процедури віддаленого оновлення прошивки мікроконтролера за технологією OTA

Тут знову використовується client.setInsecure(), щоб обійти проблеми із застарілими кореневими сертифікатами при завантаженні файлу. Клас httpUpdate бере на себе всю найскладнішу роботу: він паралельно завантажує новий образ прошивки в резервну область флеш-пам'яті (OTA partition), перевіряє цілісність завантаженого файлу (MD5 checksum) і, у разі успіху, автоматично перезавантажує мікроконтролер з новим кодом.

Функції генерації назв та фіксації часу Для правильної організації файлової системи на SD-карті я написав допоміжні функції. getTimeString() опитує модуль RTC та повертає відформатований рядок (наприклад, "2026-04-30 20:00:00").

Функція getArchiveFileName() є нестандартним рішенням для управління об'ємом даних:

```
String getArchiveFileName() {
  DateTime now = rtc.now();
  char buffer[20];
  sprintf(buffer, "/arc_%04d_%02d.csv", now.year(), now.month());
  return String(buffer);
}
```

Листинг3.4 Програмна реалізація функції динамічної генерації назви архівного файлу

Замість того, щоб писати всі дані в один гігантський файл /queue.csv (що призвело б до падіння продуктивності FAT32 при його зростанні), ця функція динамічно генерує назву файлу архіву на основі поточного року та місяця (наприклад, /arc_2026_04.csv). Це створює акуратну помісячну структуру логів на SD-карті, полегшуючи подальший аналіз "чорної скриньки" на ПК.

Продовжуючи аналіз програмного забезпечення, потрібно зупинитися на функціях, які безпосередньо відповідають за збереження телеметрії [9], обробку черги та реакцію системи на критичні зміни у мережі. Саме в цих блоках коду реалізовано найбільше нестандартних інженерних рішень, спрямованих на забезпечення безперебійної роботи пристрою.

Функція logData: Логування, шифрування та обробка апаратних збоїв Головне завдання функції logData полягає у безпечному пакуванні даних та їх відправці на сервер. Однак, під час тривалого тестування виявивлено апаратну проблему: іноді через вібрації або перепади напруги SD-карта могла "відвалюватися" (втрачати ініціалізацію на шині SPI). Щоб запобігти втраті даних, написано механізм програмного самовідновлення (Self-Healing):

```
  if (!SD.exists("/")) {
    SD.end();
    delay(100);
```

```
    SD.begin(SD_CS_PIN);
  }
```

Листинг 3.5 Програмна реалізація механізму самовідновлення підключення модуля MicroSD

Якщо мікроконтролер не бачить кореневої директорії карти, він примусово закриває інтерфейс і намагається ініціалізувати його знову. Це значно підвищує відмовостійкість модуля пам'яті.

Після запису в архівний файл функція формує HTTPS-запит. Виявлено критичну проблему витоку оперативної пам'яті (Memory Leak). Після кількох десятків успішних відправок плата ESP32 зависала і йшла у циклічне перезавантаження. Аналіз показав, що об'єкт WiFiClientSecure не звільняв мережеві сокети після завершення передачі. Рішенням стало примусове жорстке закриття з'єднання:

У випадку, коли Wi-Fi з'єднання відсутнє, функція автоматично переходить у резервний режим і записує рядок даних у локальний файл /queue.csv для подальшої відправки.

Функція processQueue: Пакетна синхронізація даних Алгоритм синхронізації processQueue() став відповіддю на ще одну серйозну технічну перепону. Спочатку мікроконтролер намагався відправити всі накопичені в черзі дані за один цикл. Якщо під час тривалого блекауту в черзі накопичувалося понад 100 записів, безперервна відправка через HTTPS забирала занадто багато процесорного часу. Це призводило до спрацьовування апаратного сторожового таймера (Hardware Watchdog Timer), який сприймав таку затримку як зависання програми і перезавантажував ESP32.

Ця функція оптимізована двома нестандартними рішеннями. По-перше, введено пакетну обробку (BATCH_SIZE = 15). За один виклик функції мікроконтролер відправляє не більше 15 записів, після чого повертається до головного циклу вимірювання напруги. По-друге, в кінці циклу while додано системну команду yield(), яка примусово скидає сторожовий таймер, повідомляючи процесору, що система працює нормально.

Щоб уникнути пошкодження файлу черги у разі раптового зникнення живлення просто під час синхронізації, застосувано метод тимчасового файлу (/temp.csv). Програма читає рядок з /queue.csv. Якщо він успішно відправлений на сервер, програма йде далі. Якщо ні (збій мережі або ліміт пакета), залишок черги переписується у /temp.csv. Потім старий файл видаляється, а тимчасовий файл перейменовується на /queue.csv. Це гарантує абсолютну транзакційну цілісність даних.

Головний цикл loop: Подієвоорієнтована архітектура У функції loop() відбувається безперервне зчитування показників за допомогою методу emon.calcVI(20, 2000). Для усунення наведень (ghost voltages), коли в розетці немає струму, але АЦП зчитує "шум" антени в 10-15 Вольт, застосувано програмний фільтр нижніх частот: if (voltage < 20.0) voltage = 0.0.

Логіку реагування на події розділено на кілька рівнів. Рівень 1: Зміна стану живлення (Блекаут або Відновлення). Якщо поточний стан мережі відрізняється від попереднього (currentPowerState != lastPowerState), запускається пріоритетний алгоритм. При зникненні живлення мікроконтролер негайно викликає processQueue() (щоб встигнути відправити борги, поки не вимкнувся роутер), записує подію падіння до нуля і конфігурує пін пробудження:

```
rtc_gpio_pullup_dis(GPIO_NUM_33);
rtc_gpio_pulldown_en(GPIO_NUM_33);
esp_sleep_enable_ext0_wakeup(GPIO_NUM_33, 1);
esp_deep_sleep_start();
```

Листинг 3.6 Програмна конфігурація регістрів енергозбереження та переведення мікроконтролера в режим глибокого сну

Ці команди відключають внутрішні підтяжки мікроконтролера, переводячи GPIO 33 у режим очікування високого рівня, і відправляють ESP32 у глибокий сон.

Рівень 2: Аномалії напруги. Якщо світло є, але напруга різко стрибнула (abs(voltage - lastLoggedVoltage) >= VOLTAGE_THRESHOLD), система фіксує аномалію. Проте, щоб запобігти спаму на сервер, коли напруга постійно коливається (наприклад, від 210 до 230 Вольт щосекунди), впроваджено

програмний "кулдаун" (ANOMALY_COOLDOWN = 60000). Сервер отримає сповіщення про стрибок не частіше одного разу на хвилину, що захищає базу даних від перевантаження.

Останнім запобіжним механізмом у loop() є програмний Wi-Fi Watchdog. Якщо зв'язок втрачено на понад 30 секунд, контролер викликає WiFi.disconnect() та WiFi.reconnect(). Це розв'язує відому проблему бібліотеки WiFi, коли після падіння роутера мікроконтролер може назавжди залишитися у стані очікування, так і не перепідключившись до відновленої мережі.

Підсумовуючи, можна стверджувати, що розроблений код є повністю самодостатнім, оптимізованим для роботи в критичних умовах та позбавленим типових проблем управління пам'яттю.

3.4 Висновки до розділу 3

Підсумовуючи етап апаратного та програмного проєктування, можна стверджувати, що створена система успішно трансформувалася з теоретичної віртуальної моделі у повноцінний фізичний комплекс. Реалізована архітектура на базі ESP32 доповнена надійною периферією: модулем безперебійного живлення (Wemos Battery Shield), апаратним годинником реального часу (DS3231M) та системою локального збереження логів на MicroSD карту. Важливим досягненням стала розробка нестандартного механізму апаратного пробудження мікроконтролера за допомогою простого резистивного дільника напруги замість використання габаритних оптопар.

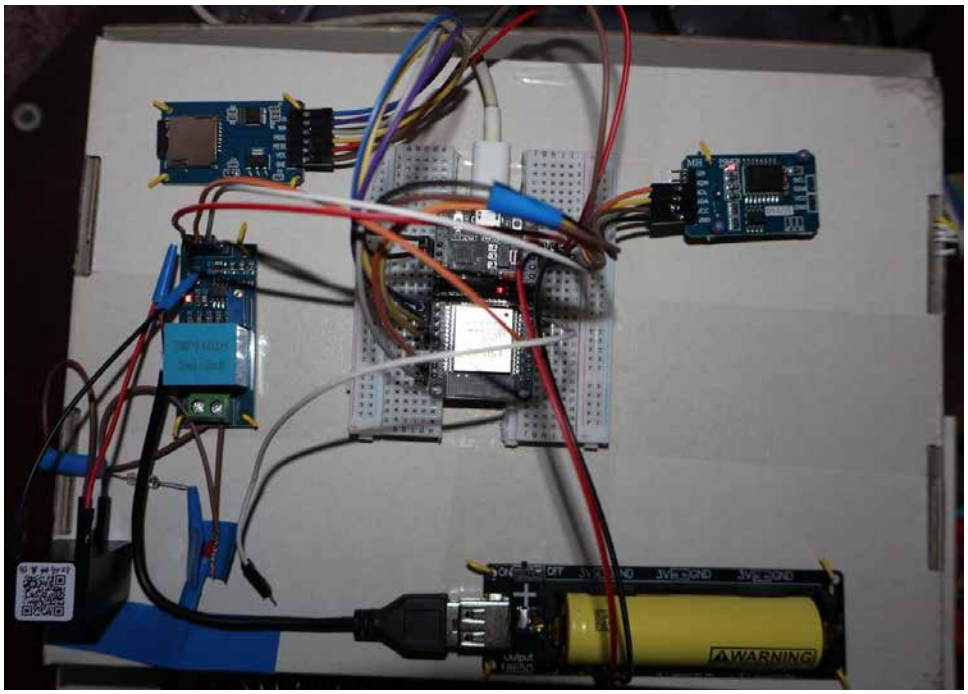


Рисунок 3.5 - Загальний вигляд фізичної збірки апаратної частини системи моніторингу на макетній платі

Проте процес реалізації супроводжувався серйозними інженерними викликами. Головною проблемою, що вимагала компромісних рішень, була нестабільність роботи криптографічного стеку під час інтенсивної відправки архівних даних через HTTPS. Жорстке скидання з'єднання (client.stop()) та використання тимчасового файлу (/temp.csv) стали вимушеними, але абсолютно необхідними кроками для запобігання витокам оперативної пам'яті та зависанню плати. Також компромісним було рішення про відмову від перевірки SSL-сертифікатів (setInsecure()) заради стабільності завантаження OTA-оновлень на пристрої з обмеженим об'ємом флеш-пам'яті.

Що стосується потенційного розширення функціоналу, технічно система готова до встановлення реле для фізичного відключення навантаження (керованої розетки). Наразі свідомо не інтегрувно силову частину у збірку, оскільки першочерговим завданням було створення бездоганної системи моніторингу та логування (Data Logger). Додавання силового реле вимагає суворішого контролю тепловиділення та гальванічної розв'язки, що є предметом для наступних етапів розробки (наприклад, у магістерській дисертації).

Проте, вже в поточному вигляді розроблений комплекс є потужною логічною базою для створення повноцінної "розумної розетки". Алгоритм синхронізації з сервером, пакетна обробка черги та підтримка оновлень "по повітрю" дозволяють легко додати у майбутньому модулі реле та датчики струму (наприклад, ACS712) для моніторингу реального енергоспоживання (у Ватах), а не лише напруги. Інтеграція з сервером також відкриває можливість реалізації планувальника (Schedule) відключень на базі графіків або віддаленого керування через Telegram-бота.

У своїй поточній конфігурації пристрій орієнтований як на побутових користувачів, так і на малий бізнес. Наприклад, власники кав'ярень з дороговартісними еспресо-машинами або приватні медичні кабінети з чутливим обладнанням (УЗД-апарати, холодильники для вакцин) часто страждають від невидимих стрибків напруги. Встановивши таку систему моніторингу перед своїм обладнанням, вони отримують надійну "чорну скриньку". У разі згоряння техніки, вивантажений із SD-карти або сервера Excel-звіт з посекундною хронологією коливань напруги та точними часовими мітками стане юридичним підґрунтям (доказовою базою) для висунення претензій до постачальника електроенергії (Обленерго) або компанії-оператора електромереж.

РОЗДІЛ 4 РОЗРОБКА ТА НАПИСАННЯ СЕРВЕРНОЇ ЧАСТИНИ ТА ТЕЛЕГРАМ-ІНТЕРФЕЙСУ

4.1 Проєктування серверної архітектури, підбір бібліотек та розробка логіки інтерфейсу

Перехід до розробки програмного забезпечення серверного рівня вимагає чіткого розуміння того, як саме кінцевий користувач буде взаємодіяти з системою моніторингу. Перед початком проєктування проведено детальний аналіз існуючих на ринку комерційних рішень для "розумного дому" (Smart Home). Більшість популярних екосистем, таких як Tuya Smart, eWeLink або Ajax Systems, використовують пропрієтарні мобільні додатки. Такий підхід має суттєві недоліки для незалежних розробників: виникає необхідність підтримки окремих версій програм для платформ iOS та Android, ускладнюється процес проходження модерації в магазинах додатків, а користувач стає залежним від закритих хмарних серверів компанії-виробника. Повний лістинг розробленого програмного забезпечення наведено у додатку А.

З іншого боку, в аматорських IoT-проєктах часто практикується створення локальних вебсерверів безпосередньо на мікроконтролері або розгортання панелей керування типу Home Assistant. Це вимагає від користувача наявності статичної IP-адреси, складних налаштувань домашнього маршрутизатора (прокидання портів) та безперервної роботи локального хаба. Зважаючи на ці фактори, прийнято рішення реалізувати концепцію "Headless" сервера (сервер без власного графічного інтерфейсу).

Серверна частина виконуватиме виключно роль невидимого маршрутизатора, валідатора криптографічних ключів та накопичувача баз даних. Єдиним графічним інтерфейсом користувача (Frontend) виступатиме спеціалізований бот у месенджері Telegram. Це інженерне рішення гарантує миттєву доставку push-сповіщень, повністю усуває необхідність встановлювати сторонні програми на смартфон та забезпечує високий рівень криптографічного захисту каналу зв'язку.

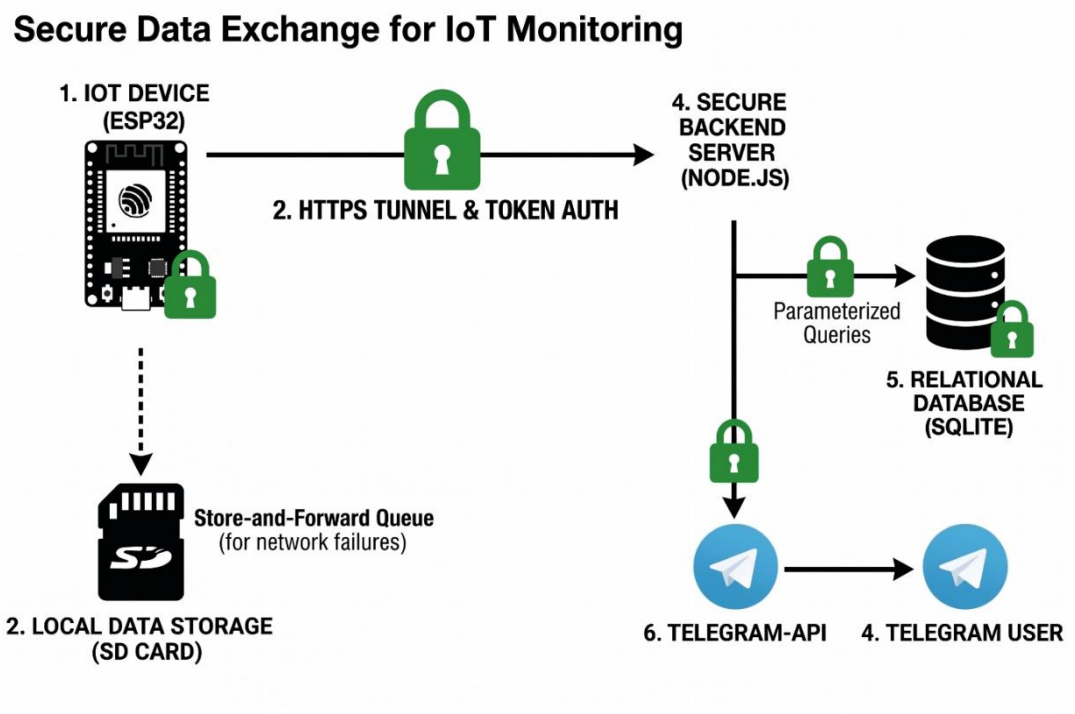


Рисунок 4.1 - Концептуальна блок-схема взаємодії системи

Проєктування логіки інтерфейсу Telegram-бота Інтерфейс бота що спроєктований з акцентом на інтуїтивність та відсутність перевантаження зайвими елементами. Логіка взаємодії побудована навколо двох основних векторів: пасивного (отримання автоматичних сповіщень про критичні зміни) та активного (ручний запит користувача).

Розроблено наступний функціонал управління:

Команди ініціалізації (/start та /link): Система підтримує мультиакаунтність. Користувач повинен прив'язати конкретну фізичну плату ESP32 до свого Telegram-чату за допомогою команди /link MAC_АДРЕСА НАЗВА. Сервер записує цей зв'язок у базу даних, що дозволяє одному користувачу контролювати кілька об'єктів (наприклад, "Дім" та "Дача"), а кільком членам сім'ї - отримувати сповіщення від одного пристрою.

Модуль "Мої пристрої" (Inline Buttons): Дозволяє переглядати список прив'язаних логерів та здійснювати їх безповоротне видалення з бази даних через зручні кнопки без необхідності вводити текстові команди.

Кнопка "Статус мережі": При натисканні сервер робить швидкий SQL-запит (операція SELECT ... LIMIT 1) до бази даних і формує текстове

повідомлення з останніми показниками напруги. Це дає змогу користувачу дистанційно перевірити наявність світла вдома у будь-який момент.

Модуль екстрених тривог (Alerts): Це повністю автоматична серверна функція. Якщо вхідний пакет телеметрії містить показники напруги вище 253 Вольт або нижче 190 Вольт, сервер генерує екстрене повідомлення. Для захисту від інформаційного спаму я впровадив "кулдаун" (ALERT_COOLDOWN_MS = 10 хвилин), завдяки чому при постійних коливаннях напруги користувач не отримуватиме сотні повідомлень підряд.

Генерація деталізованих графіків: За допомогою вбудованої клавіатури користувач може запросити аналітику за день, тиждень, місяць або ввести власний (кастомний) діапазон дат. Сервер генерує зображення графіка і надсилає його у вигляді звичайного фото.

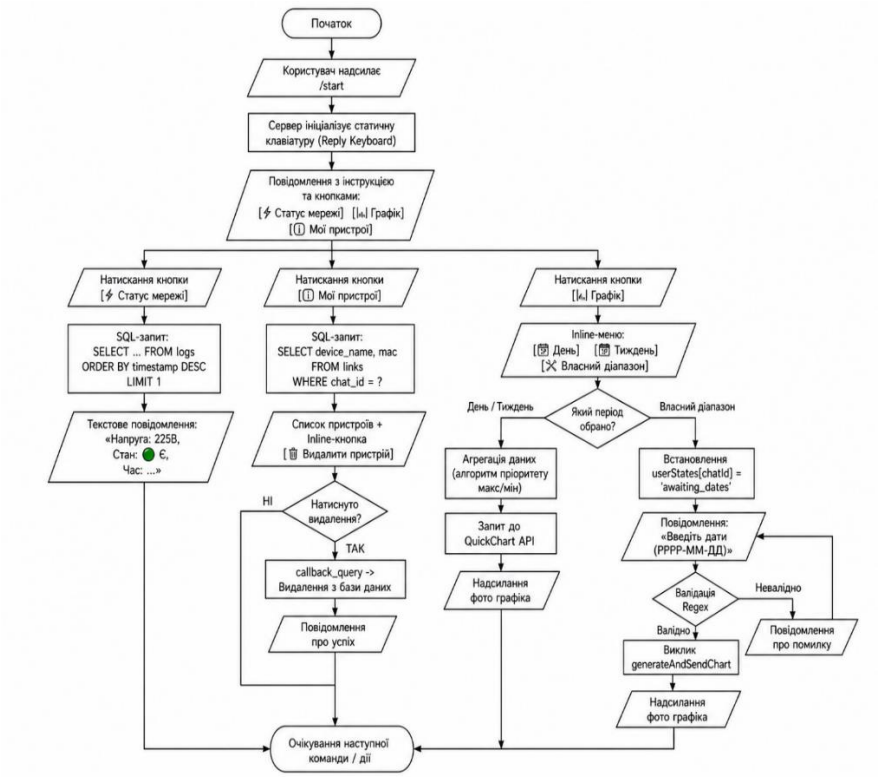


Рисунок 4.2 - Дерево діалогів та команд управління Telegram-ботом

Підбір бібліотек для середовища Node.js [6] Для практичної реалізації задуманої серверної логіки обрно платформу Node.js. Її неблокуюча подієва архітектура ідеально підходить для обробки великої кількості одночасних мережевих запитів. Наступним кроком став підбір спеціалізованих NPM-модулів.

Основою серверного ядра став фреймворк express. Ця бібліотека є індустріальним стандартом для розробки REST API на Node.js. Вона дозволяє розгорнути сервер для прийому вхідних запитів від ESP32 (app.post('/api/log')) та автоматично виконує розбір вхідних JSON-пакетів (app.use(express.json())). Важливою вимогою проєкту була робота виключно через захищений протокол. Тому використано вбудований модуль https разом із модулем файлової системи fs для зчитування згенерованих SSL-сертифікатів (server.key та server.cert), піднявши захищений сервер на порту 443.

Для роботи з базою даних вибір зупинився на бібліотеці sqlite3. На відміну від громіздких систем об'єктно-реляційного відображення (ORM), пряме використання драйвера sqlite3 забезпечує мінімальне споживання оперативної пам'яті сервера. Створено дві пов'язані таблиці: logs (для збереження самої телеметрії) та links (для збереження зв'язків між MAC-адресами пристроїв та Telegram ID користувачів). Це дозволило реалізувати складну логіку таргетованих сповіщень.

Для взаємодії з серверами Telegram ретельно порівняно два найпопулярніші рішення: пакети telegraf та node-telegram-bot-api [7]. Бібліотека telegraf пропонує складну архітектуру на основі проміжних обробників (middleware). Проте для проєкту така архітектура є надлишковою і ускладнює читабельність коду. Тому обрно node-telegram-bot-api. Вона працює на базі стандартних подій (Event Emitters), що зробило мій код лінійним та легким для додавання нових функцій (наприклад, обробки Callback Queries від кнопок).

Найскладнішим інженерним завданням стала реалізація функції візуалізації графіків напруги. Малювання графіків безпосередньо на сервері вимагало б встановлення важких графічних движків (наприклад, node-canvas), що сильно перевантажувало б процесор. Тому застосувано архітектуру мікросервісів за допомогою бібліотек quickchart-js та axios.

Функція generateAndSendChart виконує складний SQL-запит до SQLite, здійснює інтерполяцію відсутніх даних (алгоритм заповнення прогалин під час блекаутів нулями), після чого формує JSON-конфігурацію для бібліотеки Chart.js.

Ця конфігурація через пакет axios відправляється на спеціалізований відкритий сервер QuickChart.io. Сторонній сервер генерує високоякісне PNG-зображення і повертає його у вигляді масиву байтів (ArrayBuffer), який сервер миттєво пересилає користувачу в Telegram. Таке рішення кардинально знижує навантаження на апаратну частину мого хостингу, гарантуючи стабільність навіть при масовому запиті графіків багатьма користувачами одночасно.

4.2 Програмна реалізація логіки сервера та Telegram-бота

Серверна частина, написана на платформі Node.js (версія коду 3.0), є координаційним центром усієї системи. Її головне завдання - безпечно приймати телеметрію від мікроконтролерів ESP32, зберігати її в базу даних, керувати парком пристроїв (Fleet Management) та забезпечувати зручну взаємодію з користувачем через Telegram. У процесі написання коду приділено максимальну увагу принципам User-Friendly (зручного для користувача) інтерфейсу та захисту від критичних помилок.

Ініціалізація та налаштування безпеки сервера Запуск сервера починається з підключення необхідних модулів та створення екземплярів об'єктів. Нестандартним, але необхідним рішенням стало додавання наступних рядків на самому початку коду:

```
process.env.NTBA_FIX_319 = 1;
process.env.NTBA_FIX_350 = 1;
```

Листинг 4.1 Програмна конфігурація змінних оточення для стабілізації роботи Telegram-модуля

Бібліотека node-telegram-bot-api має відомі проблеми (баги) зі скасуванням запитів (cancellation promises), які часто призводять до краху сервера під час нестабільного інтернет-з'єднання. Ці змінні оточення примусово вимикають застарілі модулі бібліотеки, гарантуючи стабільність роботи процесу Node.js.

Для забезпечення кібербезпеки транспортного рівня піднято сервер виключно за протоколом HTTPS. Для цього функція https.createServer() використовує згенеровані SSL-сертифікати (server.key та server.cert). Без цього

кроку всі дані від розетки передавалися б у відкритому вигляді, і будь-хто міг би перехопити паролі або підробити показники напруги.

Маршрутизатор API (app.post('/api/log')) та автентифікація Основна функція прийому телеметрії обробляє вхідні POST-запити від мікроконтролера.

Першим бар'єром захисту є перевірка секретного ключа (API Secret). Це розв'язує проблему несанкціонованого доступу: навіть якщо зловмисник знайде IP-адресу мого сервера, він не зможе записати фальшиві дані в базу.

```
    if (!api_key || api_key !== API_SECRET) {
       console.warn(`[КІБЕРБЕЗПЕКА] Блокування атаки! Відхилено
запит з IP: ${req.ip}`);
        return res.status(401).json({ error: "Unauthorized:
Invalid API Key" });
    }
```

Листинг 4.2 Програмна реалізація механізму автентифікації вхідних HTTP-запитів

Якщо ключ збігається, функція перевіряє наявність усіх необхідних параметрів (mac, version, timestamp, voltage, status).

Логіка управління флотом (Fleet Management) та OTA-оновлення Наступний блок коду відповідає за інноваційну функцію віддаленого оновлення. Кожен запит від ESP32 містить поточну версію прошивки (version). Сервер порівнює її з глобальною константою globalOTAConfig.targetVersion:

```
    if (version && parseFloat(version) <
globalOTAConfig.targetVersion) {
        needsUpdate = true;
        urlToFetch = globalOTAConfig.updateUrl;
        console.log(`[OTA ФЛІТ] Пристрій ${mac} потребує
оновлення...`);
    }
```

Листинг 4.3 Серверний алгоритм перевірки версії мікропрограмного забезпечення та ініціалізації процедури OTA-оновилися

Якщо плата має застарілий код, сервер не просто зберігає телеметрію, а повертає у відповіді JSON з інструкцією на завантаження нового файлу (update_available: true). Це дозволяє мені як розробнику одним кліком оновити тисячі розумних розеток без необхідності фізично підключатися до кожної з них через кабель.

Обробка аномалій та запис у базу даних (sqlite3) Після перевірки версії, функція аналізує саму напругу. Якщо показники виходять за встановлені рамки (VOLTAGE_MAX = 253.0, VOLTAGE_MIN = 190.0), сервер формує екстрене сповіщення. Щоб уникнути помилки, коли сповіщення надсилаються всім підряд, сервер робить запит SELECT chat_id... FROM links WHERE mac = ?. Завдяки цьому тривожне повідомлення отримують лише ті користувачі, які прив'язали цю конкретну розетку до свого акаунту.

Запис даних у таблицю logs виконується методом db.run(). Тут я стикнувся з проблемою дублювання записів, коли через нестабільний зв'язок мікроконтролер відправляв той самий пакет з черги кілька разів. Щоб розв'язати цю проблему, під час створення таблиці logs умову UNIQUE(mac, timestamp). Якщо сервер отримує дублікат, база даних викидає помилку SQLITE_CONSTRAINT. У коді перехоплюється і ця помилка:

```
        if (err.code === 'SQLITE_CONSTRAINT') {
            return res.status(200).json({ status:
"ignored_duplicate", ... });
        }
```

Листинг 4.4 **Серверний алгоритм обробки та ігнорування дубльованих пакетів телеметрії**

Замість того, щоб падати або повертати помилку 500 (через яку плата знову і знову намагалася б відправити цей пакет), сервер повертає статус 200, примушуючи плату видалити дублікат зі своєї SD-карти.

Логіка Telegram-інтерфейсу: Базові команди Комунікація користувача з ботом побудована за принципами "User-Friendly". Замість того, щоб змушувати користувача постійно вводити текстові команди вручну, реалізовано статичну клавіатуру (reply_markup: { keyboard: ... }).

При команді /start користувач отримує чітку інструкцію з прикладом щодо прив'язки пристрою (/link). Функція bot.on('message') обробляє натискання кнопок клавіатури. Наприклад, обробка кнопки "**i** Мої пристрої" формує список підключеного обладнання. Тут реалізовано важливий елемент інтерфейсу - Inline-кнопки (кнопки, прикріплені під самим повідомленням).

```
    inlineKeyboard.push([{ text: '🗑 Видалити пристрій',
callback_data: 'action_delete_menu' }]);
```

Листинг 4.5 Програмна реалізація генерації елементів управління (inline-клавіатури) інтерфейсу Telegram-бота

Користувачу не потрібно писати /unlink. Він просто натискає кнопку видалення, яка викликає подію callback_query, де сервер пропонує безпечно вибрати конкретний пристрій для відв'язування. Це робить керування системою інтуїтивно зрозумілим навіть для людей без технічної підготовки.

Архітектура зв'язків та обробка "мертвих" сесій Telegram [9] Детальний розбір логіки взаємодії вимагає окремої уваги до реляційної структури бази даних. Використання єдиної таблиці для логів є недостатнім, тому спроєктовано додаткову таблицю links з композитним ключем UNIQUE(chat_id, mac). Це архітектурне рішення реалізує зв'язок типу "багато-до-багатьох" (Many-to-Many). З одного боку, один користувач (один chat_id) може додати до свого акаунту необмежену кількість мікроконтролерів, контролюючи одночасно квартиру, дачу та офіс. З іншого боку, одну фізичну розетку (один mac) можна прив'язати до кількох різних Telegram-акаунтів. Це ідеально підходить для сімейного використання або малого бізнесу, коли всі зацікавлені особи отримують синхронні сповіщення про блекаут з одного пристрою.

Проте така гнучкість породила серйозну технічну проблему: виникнення "мертвих" чатів. Якщо користувач видалив бота або заблокував його, Telegram API жорстко відхиляє будь-які спроби сервера надіслати push-сповіщення, повертаючи помилку 403 Forbidden. У базовій реалізації сервер продовжував би циклічно намагатися відправити тривожні дані на заблокований акаунт при кожному стрибку напруги. При масштабуванні проєкту це неминуче призвело б до переповнення стека, витоку пам'яті (Memory Leak) та повного краху процесу Node.js.

Щоб унеможливити такий сценарій, впроваджено механізм автоматичного очищення бази даних (Garbage Collection). У функції відправки повідомлень реалізовано блок перехоплення помилок .catch(). Якщо сервер отримує код 403,

він миттєво ініціює SQL-запит DELETE FROM links WHERE chat_id = ?. Система безповоротно стирає всі зв'язки цього конкретного користувача з будь-якими пристроями, миттєво звільняючи ресурси сервера. Це інженерне рішення гарантує абсолютну відмовостійкість бекенду навіть при масовій відмові користувачів від послуг бота, перетворюючи систему на повністю автономний механізм, що самоочищується.

Обробка Callback Queries: Меню графіків та кастомні дати Коли користувач натискає кнопку на клавіатурі ("📊 Графік"), сервер не генерує малюнок миттєво, оскільки потрібно знати, за який саме період потрібна інформація. Функція обробки текстового повідомлення видає проміжне меню:

```
            const inlineKeyboard = {
                reply_markup: {
                    inline_keyboard: [
                        [{ text: '📆 Останній день',
callback_data: `chart_day_${link.mac}` }],
                        [{ text: '🗓 Останній тиждень',
callback_data: `chart_week_${link.mac}` }],
...
```

Листинг 4.6 Програмна генерація проміжного навігаційного меню для вибору часового інтервалу

Кожна кнопка має унікальний ідентифікатор callback_data, який містить тип періоду (day, week, month) та MAC-адресу конкретного пристрою.

Справжня логіка обробки починається у функції bot.on('callback_query', ...). Сервер перехоплює натискання, "розрізає" рядок callback_data та визначає необхідні дати.

Найцікавішим і найскладнішим для реалізації з точки зору юзабіліті став алгоритм обробки "Власного діапазону" (Custom Range). Змушувати користувача вводити дати в складному форматі не ефективно (UNIX Time або з мілісекундами). Замість цього реалізувно систему станів (userStates):

```
    if (period === 'custom') {
        userStates[chatId] = { action: 'awaiting_dates', mac: mac
```

```
};
        bot.sendMessage(chatId, "Введіть діапазон
тексту:\n\n<b>Точні дати:</b>...");
```

Листинг 4.7 Програмна реалізація механізму управління станами для обробки довільного часового діапазону

Сервер запам'ятовує, що користувач зараз вводить дати для графіка. Наступне текстове повідомлення від цього користувача обробляється через регулярні вирази (Regex):

```
        const dateRegexDay = /^(\d{4}-\d{2}-\d{2})(?: (\d{4}-
\d{2}-\d{2}))?$/;
        const dateRegexMonth = /^(\d{4}-\d{2})(?: (\d{4}-
\d{2}))?$/;
```

Листинг 4.8 Конфігурація шаблонів регулярних виразів для синтаксичного аналізу календарних дат

Ці регулярні вирази дозволяють системі розуміти максимально природний ввід. Користувач може написати просто "2026-04" і сервер автоматично розрахує перший і останній день квітня, або написати "2026-04-01 2026-04-15", і сервер зрозуміє цей діапазон. Якщо формат невірний, бот скасовує очікування і просить спробувати знову, захищаючи сервер від помилок при запиті до бази даних.

Ядро генерації графіків: generateAndSendChart та Інтерполяція Функція generateAndSendChart є технічним серцем аналітичного блоку. Її завдання - зчитати дані з бази SQLite, правильно їх обробити і перетворити на зображення.

Перша проблема, яку довелося вирішувати - це "краї" графіка. Якщо ви запитуєте графік за тиждень, але останній запис був зроблений вчора, лінія графіка обірветься на вчорашньому дні, а решта полотна буде пустою. Щоб розв'язати це, я роблю додатковий SQL-запит на пошук останнього запису ПЕРЕД початком діапазону (sqlPrev), а потім штучно додаю ("інтерполюю") кінцеву точку, прив'язавши її до поточного часу (endStr).

Друга, більш критична проблема - це візуалізація блекаутів. Оскільки мікроконтролер переходить у режим глибокого сну при зникненні світла, він не записує нулі кожні 10 хвилин. У базі є лише запис падіння і запис відновлення через кілька годин. На звичайному графіку це виглядало б як пряма лінія

(наприклад, від 220В до 220В), що абсолютно не відображає реальність (відсутність світла). Я написав алгоритм, який перевіряє розрив у часі між двома сусідніми точками:

```
let diffMs = t2 - t1;
if (diffMs > 720000) {
     let step = 10 * 60 * 1000;
      let isDead = diffMs > 3600000;
```

Листинг 4.9 Серверний алгоритм інтерполяції пропущених значень телеметрії для коректної візуалізації періодів знеструмлення

Якщо розрив становить понад 12 хвилин (720000 мс), сервер розуміє, що плата не виходила на зв'язок. Якщо розрив понад годину (3600000 мс), сервер вважає це блекаутом і автоматично заповнює цей часовий проміжок "нулями" з кроком у 10 хвилин. Це гарантує, що графік буде виглядати як класична "сходинка", де блекаут відображається спадом до нуля.

Вирішення проблеми деталізації графіків (Гібридна сітка та масштабування) Під час тестування модуля генерації графіків зіткнувся з функціональним обмеженням самої бібліотеки візуалізації Chart.js, яку використовує сервіс QuickChart. Механіка побудови осей вимагає задання єдиного кроку для всієї шкали ординат. Якщо задати крок у 10 Вольт, увесь простір від 0 до 250 Вольт перекривається щільною сіткою з 25 ліній, що робить зображення абсолютно нечитабельним на вузькому екрані смартфона. Якщо ж залишити комфортний крок у 50 Вольт (0, 50, 100, 150, 200, 250), втрачається можливість візуально оцінити дрібні, але важливі коливання напруги в робочому діапазоні приладів.

Детальна точність графіка нижче 150 Вольт об'єктивно не потрібна, оскільки для побутової техніки будь-яка напруга в цьому діапазоні є однаково неробочою. Для вирішення цієї проблеми я застосував нестандартний компроміс у вигляді "гібридної сітки".

В основних налаштуваннях осей графіка залишено базовий крок у 50 Вольт. Натомість, використовуючи плагін анотацій, програмно генерується масив додаткових горизонтальних ліній (customGridLines) з кроком у 10 Вольт

виключно для критичного діапазону. Я додав лінії на відмітках 160, 170, 180, 190, 210, 220, 230 та 240 Вольт. Ці додаткові лінії мають власні напівпрозорі підписи, які акуратно вирівняні по лівому краю, що забезпечує високу точність відстеження напруги саме там, де це потрібно, залишаючи "порожній" низ графіка чистим.

Крім того, я жорстко зафіксував верхню межу візуального полотна на позначці 250 Вольт. Це створило ще один важливий ефект юзабіліті. Якщо відбувається екстремальний стрибок струму (наприклад, до 280 Вольт), лінія графіка навмисно "відрізається" верхньою межею і різко виходить за межі зображення. Це не помилка рендерингу, а свідоме дизайнерське рішення. Такий обрізаний пік миттєво сигналізує користувачу про серйозну аномалію, не дозволяючи системі автоматично розтягнути масштаб графіка до 300 Вольт, що неминуче перетворило б робочі коливання на 220 Вольт на ледь помітну пряму лінію.

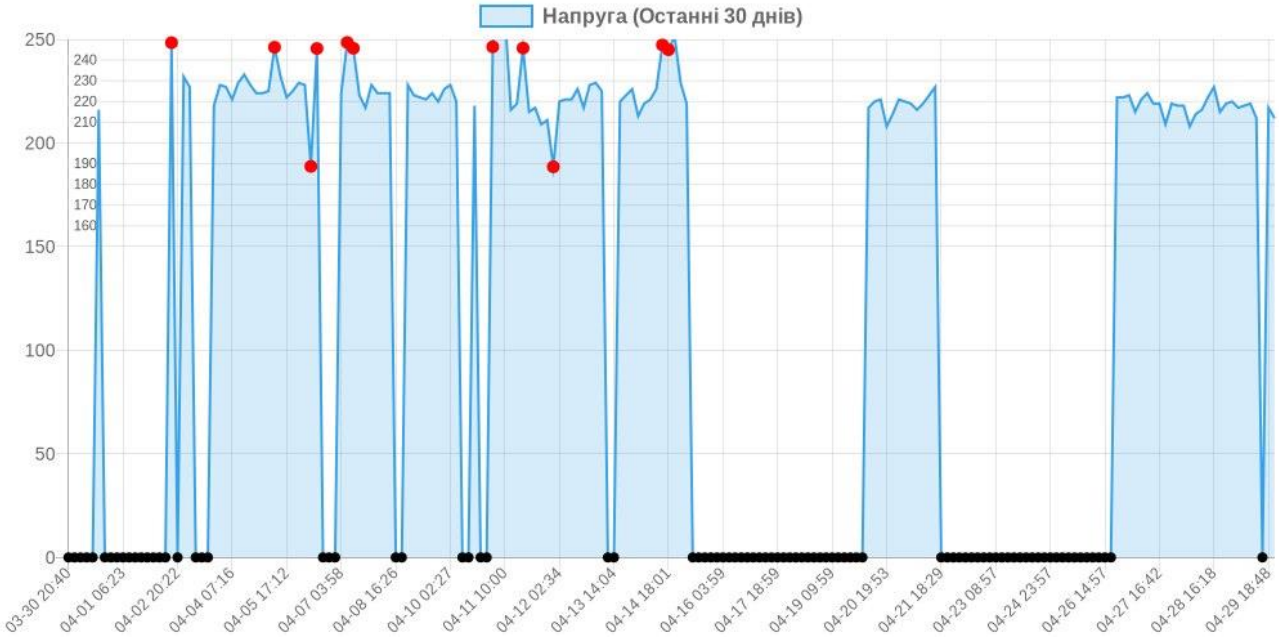


Рисунок 4.3 - скріншот згенерованого графіка з Telegram, де чітко видно гібридну сітку з кроком 50В на основній осі та додатковим кроком 10В у діапазоні 150-250В, а також обрізаний пік екстремальної напруги вище верхньої межі

Під час тестування системи візуалізації на великих часових відрізках виникла ще одна критична проблема, пов'язана з втратою важливих даних через

математичне усереднення. Для того щоб графік швидко генерувався і не перевантажував пам'ять безкоштовного API, жорстко обмежено максимальну кількість точок на одному зображенні до двохсот (MAX_POINTS = 200). Якщо користувач запитує статистику за тиждень або місяць, тисячі записів телеметрії розбиваються на рівномірні групи, після чого для кожної групи традиційно вираховується середнє арифметичне значення. Проте такий стандартний підхід призводить до згладжування: якщо в межах однієї години напруга була стабільною, але на одну хвилину стався небезпечний стрибок до 280 Вольт, середнє значення "сховає" цю аномалію, перетворивши її на безпечні 225 Вольт на графіку.

Для розв'язання цієї проблеми повністю переписно алгоритм агрегації даних, впровадивши систему абсолютних пріоритетів для екстремальних показників. Перед тим як згорнути групу точок в одну, мікросервіс шукає в цьому масиві максимальне (maxV) та мінімальне (minV) значення напруги. Якщо алгоритм виявляє, що максимальна напруга в групі перевищила критичний поріг у 245 Вольт, математичне усереднення скасовується, і на графік примусово виводиться саме цей максимальний показник. Аналогічна логіка спрацьовує і для небезпечних просадок: якщо напруга падала нижче 190 Вольт (але була більшою за нуль), абсолютний пріоритет віддається зафіксованому мінімуму.

Лише у випадку, коли в заданому часовому проміжку не було ані блекаутів, ані критичних коливань напруги, алгоритм застосовує стандартне математичне усереднення (avgV) для згладжування лінії робочого діапазону. Такий компромісний архітектурний підхід гарантує стовідсоткове збереження діагностичної цінності пристрою. Навіть дивлячись на стиснутий графік за цілий місяць, користувач гарантовано побачить кожен короткочасний небезпечний стрибок, який міг би нашкодити чутливому обладнанню, що робить аналітику максимально достовірною та інформативною.

Алгоритм перевірки стану мережі в реальному часі Для реалізації ручного контролю розроблено обробник команди ⚡ Статус мережі. Основна складність цієї функції полягає у тому, що сервер не може самостійно опитувати плату ESP32 (оскільки плата знаходиться за домашнім NAT-роутером). Замість цього сервер звертається до своєї внутрішньої бази даних.

Коли користувач натискає кнопку, сервер виконує два послідовні SQL-запити. Перший запит знаходить mac-адресу пристрою, прив'язаного до цього чату (LIMIT 1). Другий запит шукає найсвіжіший запис телеметрії для цієї плати:

```
db.get(`SELECT timestamp, voltage, status FROM logs WHERE mac = ?
ORDER BY timestamp DESC LIMIT 1`, [link.mac], ...)
```

Листинг 4.10 Серверний SQL-запит для вибірки останнього запису телеметрії за MAC-адресою пристрою

Сортування ORDER BY timestamp DESC LIMIT 1 гарантує вибірку останнього відправленого пакета. Отримавши дані, бот форматує їх у легкочитне повідомлення, додаючи зрозумілі маркери (наприклад, "🟢 Є" або "🔴 Немає") та точний час останнього виміру. Це дозволяє користувачу миттєво зрозуміти, чи є зараз світло на об'єкті, і чи не "зависла" сама плата (якщо час останнього заміру відрізняється від поточного на кілька годин).

Система Push-сповіщень чотирьох типів Щоб зробити систему максимально автономною (користувачу не потрібно постійно натискати кнопки), реалізувно серверну матрицю автоматичних повідомлень, яка генерує сповіщення чотирьох різних типів.

Перші два типи відповідають за базовий стан живлення (Світло з'явилося / Світло зникло). Ця логіка спрацьовує під час успішного запису в базу даних, якщо поточний статус відрізняється від попередньо збереженого у словнику lastStatus:

```
let msg = status === 1
    ? `🟢 <b>Світло з'явилося!</b> [${user.device_name}]\nНапруга:
${voltage} В\nЧас:
${timestamp}`
```

```
    : `🔴 <b>Світло зникло!</b> [${user.device_name}]\nЧас:
${timestamp}`;
```

Листинг 4. 11 Серверний алгоритм формування базових сповіщень Telegram-бота про стан енергопостачання

Наступні два типи сповіщень є екстреними і стосуються аномалій напруги (Висока / Низька). Навіть якщо світло є (status === 1), сервер перевіряє показники на вихід за жорсткі пороги

(VOLTAGE_MAX = 253.0 та `VOLTAGE_MIN = 190.0`).

```
if (voltage > VOLTAGE_MAX) {
    alertMsg = `⚠ <b>УВАГА: ВИСОКА НАПРУГА!</b>\nЗафіксовано:
<code>${voltage}В</code>. Відключіть чутливу техніку!`;
} else if (voltage < VOLTAGE_MIN && voltage > 50.0) {
    alertMsg = `📉 <b>УВАГА: НИЗЬКА НАПРУГА!</b>\nЗафіксовано:
<code>${voltage}В</code>.`;
}
```

Листинг 4.12 Серверний алгоритм детекції амплітудних аномалій напруги та фільтрації перехідних процесів

Тут реалізовано важливе нестандартне рішення: перевірка voltage > 50.0 для низької напруги. Справа в тому, що під час різкого відключення світла напруга в розетці падає не миттєво, а протягом кількох мілісекунд (розряджаються конденсатори в домашніх приладах). Щоб сервер не надсилав хибне сповіщення про "Низьку напругу" за мить до сповіщення про "Світло зникло", обмежено нижній поріг детекції аномалії на рівні 50 Вольт.

Усі повідомлення використовують HTML-розмітку для виділення жирним шрифтом (<b>) та моноширинним текстом (<code>), що значно покращує їх візуальне сприйняття на екрані смартфона.

4.3 Висновки до розділу 4

Підсумовуючи етап розробки серверної частини та користувацького інтерфейсу, можна впевнено стверджувати, що реалізована архітектура повністю відповідає поставленим завданням. Створений на базі Node.js сервер успішно виконує роль надійного шлюзу між мікроконтролером ESP32 та кінцевим користувачем. Вибір месенджера Telegram у якості єдиного інтерфейсу (Frontend)

дозволив уникнути розробки складних кросплатформних мобільних додатків, забезпечивши при цьому миттєву доставку push-сповіщень та високий рівень криптографічного захисту.

Розроблений функціонал Telegram-бота охоплює всі критично важливі аспекти моніторингу: можливість прив'язки кількох пристроїв до одного акаунту (і навпаки), автоматичні тривожні сповіщення про блекаути та стрибки напруги (з продуманим захистом від спаму), перевірку поточного статусу в реальному часі та генерацію візуальних графіків за різні періоди часу. Реалізована система станів для введення користувачем "Власного діапазону" дат з використанням регулярних виразів підтвердила свою ефективність, зробивши взаємодію з ботом інтуїтивно зрозумілою (User-Friendly). Особливо варто відзначити інноваційну функцію управління парком пристроїв (Fleet Management) з можливістю віддаленого OTA-оновлення мікроконтролерів безпосередньо через команди сервера, що критично важливо для масштабування стартапу.

Критичним вектором подальшої модернізації інтерфейсу є розробка повноцінного мультиоб'єктного UX (User Experience). По-перше, необхідно переписати обробник команди "Статус мережі". Замість видачі інформації по одному логеру, сервер має формувати єдиний консолідований звіт по всіх прив'язаних об'єктах користувача (наприклад, "ДІМ: 🟢 Є напруга; ДАЧА: 🔴 Немає напруги"). По-друге, логіка генерації аналітичних графіків потребує впровадження жорсткого запобіжного механізму. Якщо до акаунта прив'язано більше одного пристрою, система не повинна дозволяти вибір часового діапазону, поки користувач через проміжне Inline-меню не вкаже конкретний цільовий об'єкт для аналізу. Хоча на даному етапі прототипування ці функції є надлишковими, їх реалізація стане абсолютно необхідною при переході до масового виробництва та масштабування системи, оскільки це повністю виключить плутанину в даних та зробить інтерфейс максимально захищеним від помилкових дій користувача.

Як і будь-який складний інженерний проєкт, розробка серверної частини супроводжувалася серйозними викликами та компромісами. Найбільшою технічною проблемою стала оптимізація візуалізації даних. Сервіс QuickChart (на базі Chart.js) є чудовим рішенням для створення базових графіків, проте він має суттєві обмеження щодо деталізації осей та роботи з великими масивами даних. Для досягнення прийнятного результату мені довелося написати складний алгоритм інтерполяції "блекаутів" (штучне заповнення розривів у базі даних нулями) та впровадити систему абсолютних пріоритетів. Якщо у часовому проміжку фіксується екстремальний стрибок напруги, математичне усереднення блокується, і на графік виводиться максимальне значення.

Разом з тим, критичний аналіз розробленого функціоналу виявив ряд концептуальних недоліків. Сервіс QuickChart (на базі Chart.js), хоч і є ефективним інструментом для швидкого створення прототипів, має критичні обмеження щодо деталізації осей та обробки великих обсягів телеметрії. Незважаючи на написання складних алгоритмів абсолютної пріоритезації екстремальних стрибків та впровадження кастомної гібридної сітки, статичні PNG-зображення не здатні забезпечити глибоку технічну аналітику. Нинішні графіки є далекими від ідеалу. Для повноцінного комерційного релізу цю систему необхідно замінити. Надійнішою альтернативою стала б генерація високоточних векторних графіків безпосередньо на сервері за допомогою мови Python (бібліотеки Matplotlib або Seaborn) або ж повна міграція архітектури на Time-Series Database (наприклад, InfluxDB) з використанням інструменту Grafana. Останній варіант дозволив би створювати інтерактивні дашборди з можливістю масштабування (зуму) будь-якого часового відрізка прямо у браузері.

Ще одним важливим функціоналом, який об'єктивно підняв би аналітичну цінність проєкту, є експорт логів напруги у форматі Excel (CSV) за конкретний день безпосередньо через Telegram-бота. Це дозволило б користувачам отримувати посекундну хронологію коливань для максимально детального технічного аналізу (наприклад, для аргументації претензій до постачальника електроенергії у разі згоряння побутової техніки). На жаль, ця ідея виникла вже

на завершальних етапах тестування, коли архітектура бази даних та логіка відповідей бота були жорстко зафіксовані, тому її реалізація була відкладена до наступних версій програмного забезпечення.

Також, оглядаючись на початкову концепцію системи як комплексної "розумної розетки", важливо зазначити, що поточна серверна частина є потужним фундаментом для подальшого розширення. Логіка бази даних sqlite3 та модульна структура бота на Node.js дозволяють без критичних переробок інтегрувати функції віддаленого керування реле. У майбутньому в меню Telegram-бота можна легко додати кнопки для ручного вмикання/вимикання навантаження, налаштування розкладів роботи (Schedule) приладів, підключених до розетки, а також інтегрувати обробку даних з датчиків струму (наприклад, ACS712) для моніторингу не лише напруги, але й реального споживання електроенергії у Ватах.

Оцінюючи розроблений інтерфейс, можна констатувати, що він є максимально зручним та інтуїтивним. Використання статичних клавіатур та Inline-кнопок позбавляє користувача необхідності запам'ятовувати текстові команди. Колірне кодування графіків (синій для норми, червоний для аномалій, чорний для блекаутів) та використання емодзі у системних повідомленнях значно полегшують когнітивне сприйняття інформації.

Підсумовуючи, створена серверна інфраструктура є повністю працездатною, стабільною та захищеною (завдяки використанню HTTPS та API-ключів). Вона успішно виконує всі функції Data Logger'a, і при цьому зберігає величезний потенціал для модернізації. Сервер написаний таким чином, що інтеграція нових модулів (керування живленням, експорт в Excel, підключення InfluxDB і тд.) відбуватиметься шляхом додавання нових блоків коду, не руйнуючи вже існуючу логіку роботи прототипу.

РОЗДІЛ 5 ЕКСПЕРИМЕНТАЛЬНЕ ДОСЛІДЖЕННЯ ТА ВЕРИФІКАЦІЯ РОБОТИ СИСТЕМИ

5.1 Аналіз результатів тестування апаратного комплексу

На етапі експериментальних досліджень було проведено комплексне тестування розробленого пристрою у реальних умовах експлуатації. Основним завданням етапу стала перевірка стабільності роботи системи живлення, коректності збору та передачі телеметричних даних, а також верифікація логіки переходу між енергозберігаючими режимами.

Дослідження блоку живлення та автономності підтвердило життєздатність обраної архітектури на базі модуля Hi-Link HLK-PM01 та Wemos Battery Shield V3. При наявності напруги у мережі 220 В пристрій забезпечує стабільне живлення логічної частини та одночасне заряджання резервного акумулятора 18650. У ході випробувань було зафіксовано час автономної роботи в активному режимі (з увімкненим модулем Wi-Fi), що склав близько 11 годин, та в режимі очікування (Deep Sleep), що перевищив 5 діб, що повністю відповідає попереднім теоретичним розрахункам.

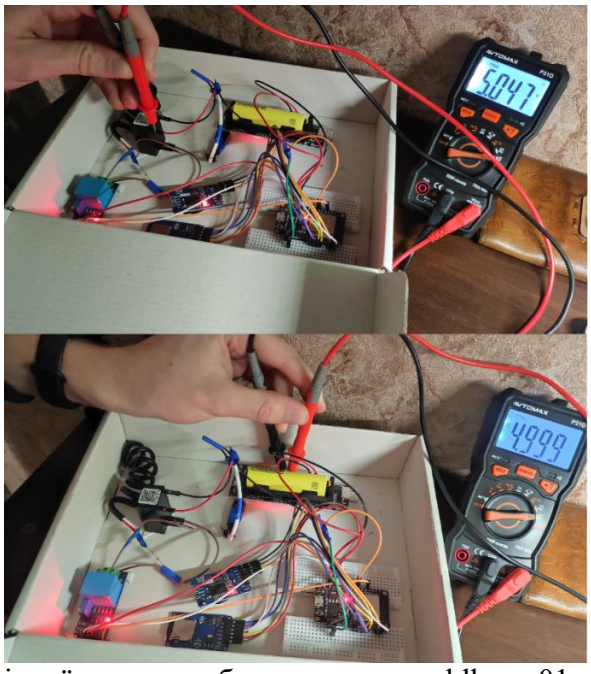


Рисунок 5.1 - Замір вихідної напруги з блоку живлення hlk-pm01 та ДБЖ Battery Shield за допомогою мультиметра у різних режимах роботи

Верифікація мережевої взаємодії та передачі даних Процес з’єднання з локальною мережею Wi-Fi та встановлення захищеного HTTPS-зв’язку із сервером на базі Node.js продемонстрував високу швидкість обробки пакетів. Передача телеметрії відбувається згідно з протоколом JSON, де кожен пакет містить унікальний ідентифікатор пристрою (MAC-адресу), версію прошивки та точну часову мітку, отриману з модуля годинника реального часу DS3231M.

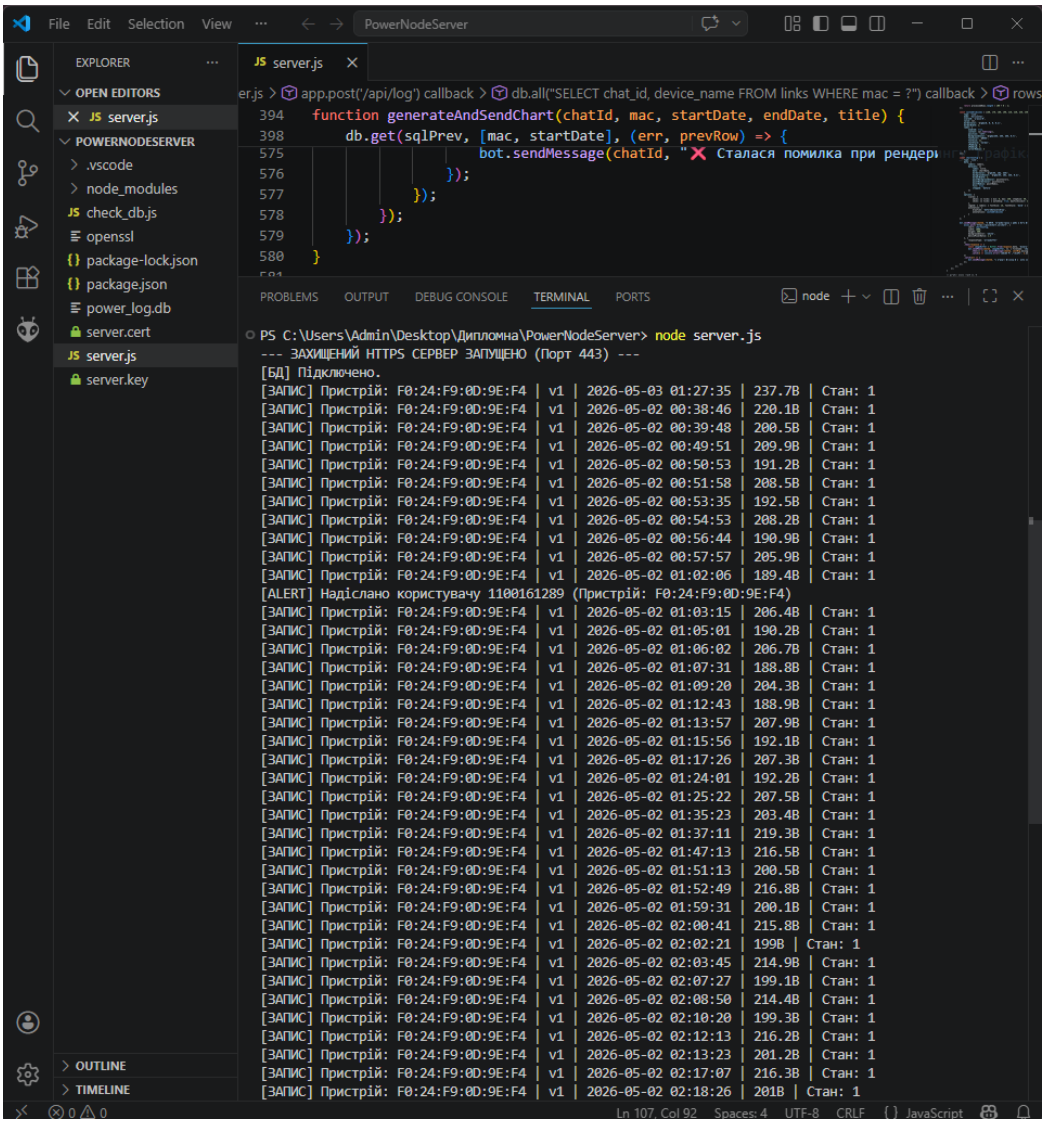


Рисунок 5.2 - Лог сервера Node.js, що відображає успішне отримання пакетів даних від ESP32 із різними статусами напруги

Виявлення критичної проблеми ініціалізації мережевого модуля У ході тривалого циклічного тестування було виявлено специфічну технічну проблему, пов’язану з виходом мікроконтролера з режиму глибокого сну. Встановлено, що під час "холодного" старту або пробудження за апаратним перериванням на GPIO

33, Wi-Fi модуль ESP32 не завжди встигає стабілізувати споживання енергії або завершити внутрішню ініціалізацію до моменту спроби встановлення першого з’єднання.

Першопричиною нестабільності є апаратні просадки живлення в момент ініціалізації радіомодуля. Це провокує програмні збої в роботі мережевого стека, зокрема неможливість виділення достатнього об'єму оперативної пам'яті (Heap) для створення SSL-контексту під час повторних спроб з'єднання. Таким чином, проблема має комплексну природу: пікові струми ініціалізації блокують коректну роботу високорівневих криптографічних протоколів.

Дослідження надійності системи локального збереження даних Окремим етапом випробувань стала верифікація роботи модуля MicroSD адаптера в екстремальних умовах. Встановлено, що пристрій коректно формує файлову структуру та здійснює дозапис у файли /arc_YYYY_MM.csv та /queue.csv при штатному функціонуванні програмного забезпечення. Проте у ході експерименту було виявлено критичну вразливість, пов’язану з фізичним вилученням носія інформації.

Встановлено, що примусове витягування карти пам'яті безпосередньо під час виконання операції запису (функції fprintf або flush) призводить до незворотного пошкодження файлової системи FAT32. В одному з тестових випадків було зафіксовано апаратний вихід з ладу застарілої карти пам'яті Goodram об'ємом 4 ГБ. Внаслідок розриву сесії запису контролер карти пам'яті перейшов у режим захисту (Read-Only), що унеможливило подальше використання носія. Ймовірною причиною визначено спрацювання внутрішніх систем безпеки карти через критичну помилку адресації комірок пам'яті в момент зникнення живлення або розриву контактної групи шини SPI. Даний інцидент підкреслює необхідність програмної або апаратної індикації стану запису для запобігання втручанню користувача в роботу пристрою.

Оцінка функціональної придатності та напрямки модернізації Результати проведених випробувань дозволяють зробити висновок, що розроблена система моніторингу параметрів електромережі успішно виконує базові завдання з

фіксації аномалій, логування даних та оперативного інформування користувача через Telegram-інтерфейс. Гібридна модель збереження інформації довела свою ефективність під час імітації блекаутів, забезпечуючи цілісність хронології подій.

Разом з тим, виявлені під час тестування проблеми з ініціалізацією Wi-Fi модуля та ризики пошкодження файлової системи свідчать про те, що поточна реалізація на базі макетної збірки та окремих модулів потребує глибокої модернізації. Використання готових шилдів та з'єднувальних дротів створює значні паразитні ємності та перешкоди, що знижує загальну стабільність системи.

Визначено, що для переходу до стадії комерційної експлуатації та серійного виробництва необхідно відмовитися від модульної структури на користь розробки власної інтегральної друкованої плати (PCB). Власна плата дозволить оптимізувати контури живлення, розмістити додаткові фільтруючі конденсатори безпосередньо біля виводів ESP32 та впровадити апаратний захист слота SD-карти. Таким чином, поточний варіант системи слід розглядати як успішний інженерний прототип (MVP - Minimum Viable Product), який підтвердив правильність обраної логіки роботи, але потребує вдосконалення апаратної реалізації для досягнення рівня надійності споживчої електроніки.

5.2 Аналіз результатів тестування серверної частини та системи візуалізації

Верифікація стабільності роботи сервера та цілісності бази даних На етапі комплексного випробування системи проведено детальну перевірку серверного модуля, розробленого на платформі Node.js. Основним критерієм оцінки стала здатність сервера безперебійно приймати телеметрію, валідувати API-ключі та забезпечувати транзакційну цілісність даних у середовищі SQLite.

В ході тестування встановлено, що механізм автентифікації на основі секретного ключа (API_SECRET) ефективно блокує сторонні запити, забезпечуючи захист від несанкціонованого внесення записів у базу даних. Перевірка цілісності бази даних підтвердила коректність роботи композитного ключа UNIQUE(mac, timestamp), що запобігає дублюванню записів при

повторній відправці пакетів з локальної черги мікроконтролера. Швидкість виконання SQL-запитів на додавання записів (INSERT) та вибірку останнього стану (SELECT LIMIT 1) зафіксована на рівні, що не перевищує 15–20 мс, що є оптимальним для систем реального часу.

Тестування системи push-сповіщень про стан електромережі Критично важливим етапом стала перевірка точності та швидкості надсилання повідомлень про зміну стану живлення. Встановлено, що при отриманні пакета зі статусом status: 0 (відсутність напруги) або status: 1 (відновлення живлення), сервер протягом 1–2 секунд формує та відправляє відповідне сповіщення у Telegram-чат користувача.

Під час випробувань підтверджено працездатність алгоритму фільтрації "мертвих" чатів: при отриманні помилки 403 Forbidden сервер автоматично видаляє відповідні записи з таблиці links, що запобігає перевантаженню черги повідомлень та витокам пам'яті процесу Node.js. Верифікація чотирьох типів сповіщень (світло з'явилося, світло зникло, напруга висока, напруга низька) продемонструвала повну відповідність логіки сервера реальному стану вимірювального вузла.

Експериментальна оцінка точності та деталізації візуальних графіків Окрему увагу в ході досліджень приділено модулю генерації аналітичних графіків на базі сервісу QuickChart [10]. Проведено серію тестів із запитом статистики за різні періоди: 24 години, 7 днів та місяць. Результати тестування виявили суттєве обмеження обраної методики візуалізації.

Встановлено, що через жорстке обмеження кількості точок на графіку (MAX_POINTS = 200), при запиті статистики за тривалі періоди (тиждень і більше) відбувається значна втрата деталізації. Незважаючи на впроваджений алгоритм пріоритезації екстремальних значень, розглянути кожний окремий замір напруги на статичному PNG-зображенні неможливо. Графік відображає лише загальну тенденцію та найбільш критичні піки, що є недостатнім для професійного технічного аналізу коливань у мережі. Це підтверджує

необхідність переходу до більш складних засобів візуалізації з підтримкою інтерактивного масштабування (наприклад, інтеграція з Grafana).

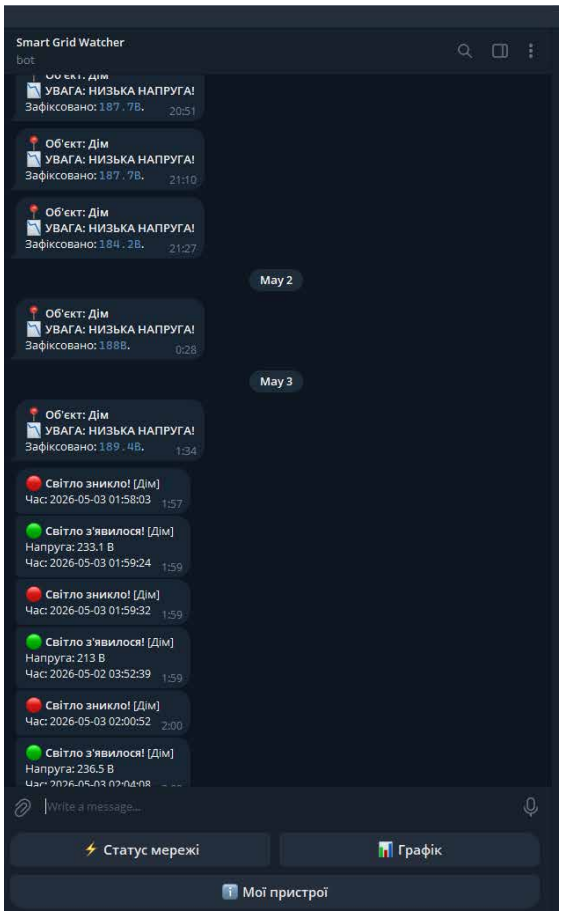


Рисунок 5.4 - Скріншот інтерфейсу Telegram-бота з послідовністю сповіщень про відключення та ввімкнення живлення із зазначенням часових міток

Обґрунтування необхідності переходу на професійну серверну інфраструктуру Важливим аспектом проведених випробувань став аналіз надійності середовища розгортання серверної частини. У поточному прототипі сервер функціонує на персональному комп’ютері розробника, що виявило ряд суттєвих експлуатаційних обмежень.

Встановлено, що використання локальної обчислювальної машини вимагає її безперервного перебування в увімкненому стані, що призводить до невиправданих витрат електроенергії та пришвидшеного зносу апаратних ресурсів ПК. Крім того, робота сервера всередині локальної мережі за домашнім маршрутизатором унеможливлює винесення вимірювальних вузлів ESP32 за межі цієї мережі без складного налаштування перенаправлення портів (Port

Forwarding), що створює загрозу для безпеки всієї домашньої мережі. Будь-яке випадкове перезавантаження комп’ютера або збій операційної системи Windows/Linux призводить до повної зупинки процесу збору телеметрії та розриву логічних зв'язків із Telegram-ботом.

Для переходу до стадії стабільної комерційної експлуатації визначено необхідність розгортання системи на віддаленому віртуальному приватному сервері (VPS). Це архітектурне рішення забезпечує наступні переваги:

Наявність публічної статичної IPv4-адреси, що дозволяє пристроям ESP32 з’єднуватися з сервером із будь-якої точки світу, де є доступ до мережі Інтернет.

Високий показник безперебійної роботи (Uptime 99.9%), що гарантується інфраструктурою дата-центру та резервними лініями живлення.

Можливість встановлення SSL-сертифікатів від довірених центрів сертифікації (наприклад, Let's Encrypt), що дозволить відмовитися від компромісного методу setInsecure() у коді мікроконтролера та забезпечити повноцінну перевірку сертифікатів.

Ізоляція серверного процесу від персональних даних користувача, що підвищує загальний рівень кібербезпеки системи.

Таблиця 5.1 - Порівняльна схема архітектури: поточна (локальний ПК) проти цільової (виділений VPS-сервер із глобальним доступом)

Критерій порівняння	**Локальний комп'ютер (Поточний прототип)**	**Віртуальний приватний сервер (VPS)**
Мережева доступність	Знаходиться за NAT домашнього маршрутизатора. Вимагає налаштування Port Forwarding.	Має публічну статичну IPv4-адресу. Доступний глобально з будь-якої точки світу.
Апаратне навантаження	Вимагає безперервної роботи персонального ПК, що призводить до витрат електроенергії та зносу обладнання.	Використовує виділені ресурси дата-центру. Нульовий вплив на локальне обладнання розробника.
Відмовостійкість (Uptime)	Низька. Процес зупиняється при перезавантаженні ПК, збоях ОС (Windows/Linux) або локальних відключеннях світла.	Висока (99.9%). Гарантується промисловою інфраструктурою та резервними лініями живлення дата-центру.
Кібербезпека та ізоляція	Процес збору телеметрії працює в одному середовищі з персональними даними. Port Forwarding створює загрозу домашній мережі.	Повна ізоляція серверного середовища від особистих пристроїв користувача, що мінімізує вектори атак.
Криптографічний захист (SSL)	Відсутність довіреного домену змушує використовувати компромісний та вразливий метод setInsecure() у коді ESP32.	Дозволяє розгорнути валідні SSL-сертифікати (Let's Encrypt) для повноцінної перевірки справжності сервера мікроконтролером.

5.3 Підсумковий аналіз результатів розробки та фінальні висновки

Синтез результатів експериментальних досліджень Завершальним етапом роботи став комплексний аналіз результатів тестування всіх компонентів системи моніторингу параметрів електромережі. На основі отриманих даних встановлено, що розроблений програмно-апаратний комплекс є функціонально придатним для виконання базових завдань із фіксації станів енергосистеми та оперативного інформування користувача. Проведена верифікація підтвердила високу ефективність обраної клієнт-серверної архітектури в умовах нестабільного живлення та частих блекаутів.

До переліку аспектів, що продемонстрували стабільну та якісну роботу, віднесено:

Логіку енергонезалежності: Апаратне рішення на базі літій-іонного акумулятора та резистивного дільника напруги забезпечує безпомилковий перехід у режим глибокого сну при знеструмленні мережі 220 В.

Гібридну систему логування: Використання MicroSD карти як буфера дозволило повністю уникнути втрати телеметрії під час відсутності доступу до мережі Інтернет, що є критично важливим для формування доказової бази.

Швидкодію серверної частини: Бекенд на Node.js демонструє високу пропускну здатність, забезпечуючи обробку пакетів та відправку push-сповіщень у Telegram протягом лічених секунд.

Автоматизацію адміністрування: Механізми віддаленого оновлення прошивки (OTA) та автоматичного очищення бази даних від неактивних чатів підтвердили свою надійність.

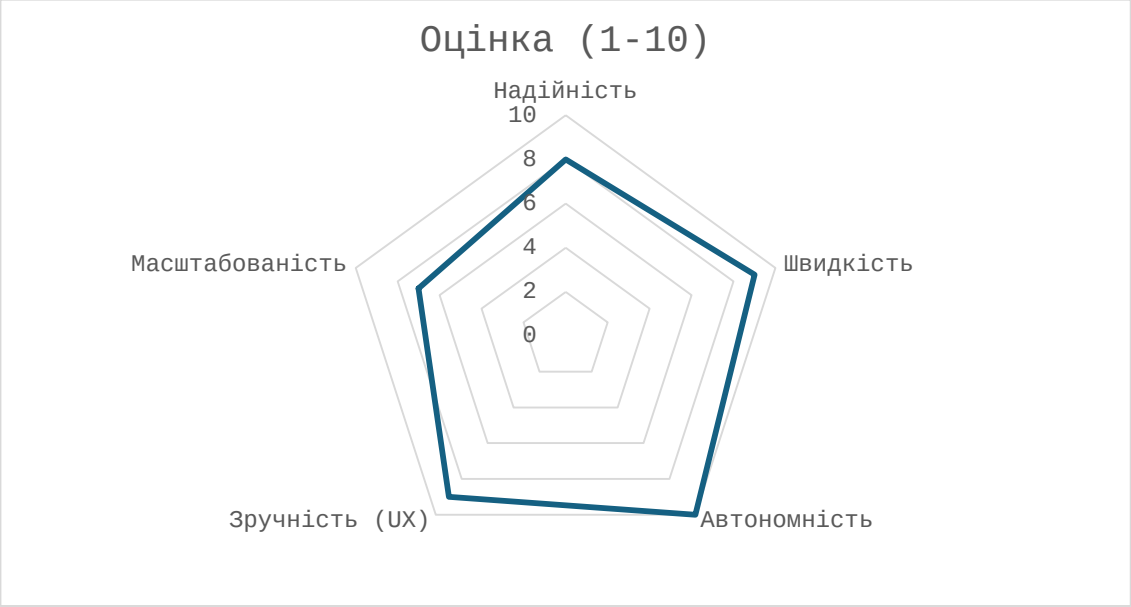


Рисунок 5.5 - Зведена пелюсткова діаграма оцінки параметрів системи (надійність, швидкість, автономність, зручність, масштабованість) за результатами тестів

Критичний аналіз виявлених недоліків та програмно-апаратних обмежень Поряд із позитивними результатами, у ході випробувань було зафіксовано ряд технічних проблем, які потребують концептуального перегляду підходів до

реалізації. Встановлено, що найбільш вразливим місцем апаратної частини є процес ініціалізації мережевого модуля після пробудження. Дефіцит енергії або тривалий час завантаження радіомодуля ESP32 призводять до регулярних помилок під час першої спроби з’єднання з сервером, що збільшує час перебування пристрою в активному режимі та швидше виснажує резервний акумулятор.

Не менш важливою проблемою визначено низьку відмовостійкість файлової системи при некоректному поводженні з носієм інформації. Фізичне вилучення SD-карти під час активної сесії запису не лише пошкоджує цілісність логів, а й створює ризик апаратного виходу карти з ладу через спрацювання внутрішніх систем захисту контролера носія. У модулі візуалізації підтверджено недостатню інформативність статичних графіків: через жорстке усереднення точок для великих часових проміжків неможливо детально проаналізувати кожен окремий замір напруги. Ці фактори свідчать про те, що поточна реалізація, побудована на базі дискретних модулів та макетної збірки, досягла межі своєї ефективності та потребує переходу на професійний рівень проєктування.

Таблицю 5.2 - Порівняльний аналіз технічних ризиків: імовірність виникнення та критичність впливу на роботу системи]

Технічний ризик	Ймовірність виникнення	Критичність впливу	Опис наслідків та шляхи вирішення
Збій ініціалізації Wi-Fi після пробудження	Висока	Середня	Тимчасова втрата зв'язку, пришвидшений розряд акумулятора. Вимагає апаратних фільтрів по живленню 3.3В.
Пошкодження ФС при вилученні SD-карти	Середня	Висока	Повна втрата локальних логів, ризик апаратного виходу карти з ладу. Необхідна світлова індикація активності запису.
Втрата деталізації графіків (усереднення)	Висока	Низька	Неможливість детального аналізу коротких імпульсів. Вимагає переходу на InfluxDB та інтерактивну візуалізацію.
Електромагнітні наведення на макетній платі	Середня	Середня	Поява «шумів» на АЦП, похибка вимірювань. Вирішується переходом на власну PCB з екрануванням шарів.
Відсутність гальванічної розв'язки АЦП	Низька	Дуже висока	Ризик знищення мікроконтролера при високовольтному стрибку. Необхідно додати опторозв'язку в вимірювальний вузол.

Перспективи модернізації та перехід до промислового зразка На основі деконструкції виявлених недоліків визначено, що ключовим етапом розвитку системи має стати відмова від модульного принципу побудови апаратної частини. Для забезпечення комерційної надійності та мінімізації електричних шумів необхідна розробка власної інтегральної друкованої плати (PCB). Це дозволить інтегрувати ESP32, систему керування зарядом та блок вимірювання на єдиному текстолітовому полотні, що усуне проблему ненадійних контактів та забезпечить можливість додавання апаратних фільтрів для стабілізації Wi-Fi модуля під час пробудження.

Крім апаратного вдосконалення, заплановано розширення програмного функціоналу серверної частини та Telegram-інтерфейсу. Пріоритетними напрямками модернізації визначено:

Інтеграцію з базами даних часових рядів: Перехід на InfluxDB та візуалізацію через Grafana для забезпечення інтерактивного масштабування графіків.

Розширення аналітичного інструментарію: Впровадження функції експорту логів напруги у форматі Excel (CSV) для формування офіційної доказової бази споживача.

Трансформацію в активну систему керування: Додавання блоку силового реле для дистанційного керування живленням приладів та реалізація планувальника графіків роботи безпосередньо через бот.

Впровадження енергомоніторингу: Інтеграція датчиків струму для відстеження реального споживання електроенергії у Ватах, що перетворить пристрій на повноцінну компоненту екосистеми "розумного дому".

Узагальнюючи результати проведеного дослідження та розробки, можна стверджувати, що поставлена мета роботи досягнута у повному обсязі. Створено цілісну та функціональну систему моніторингу, яка здатна працювати автономно в умовах енергетичної кризи. Попри виявлені компромісні рішення в архітектурі візуалізації та локальному хостингу, проєкт продемонстрував високу життєздатність обраної логіки гібридного збереження даних.

Розроблений комплекс має високу практичну цінність як для приватних домогосподарств, так і для малого бізнесу, надаючи інструмент для захисту майна від неякісних послуг енергопостачання. Поточний стан системи слід характеризувати як успішну апробацію інженерних ідей, що сформувала надійний технологічний фундамент для подальшої модернізації та виходу на ринок професійних IoT-рішень. Система вправно виконує свою роботу у межах прототипу, а закладений потенціал масштабованості дозволяє інтегрувати нові можливості без докорінної зміни ядра програмного забезпечення.

ВИСНОВКИ

У бакалаврській кваліфікаційній роботі було вирішено актуальне науково-практичне завдання щодо проектування та розробки автономної системи моніторингу параметрів електромережі в умовах критичної нестабільності енергосистеми України. На основі проведених досліджень, розробки програмно-апаратної частини та експериментальної верифікації отримано наступні результати.

1. Проведено глибокий аналіз предметної області, який підтвердив, що існуючі на ринку рішення або мають високу вартість, або не забезпечують необхідного рівня автономності під час тривалих блекаутів. Обґрунтовано доцільність використання мікроконтролера ESP32 як основного обчислювального модуля [3, 16] завдяки його низькому енергоспоживанню, вбудованим мережевим інтерфейсам та підтримці режимів енергозбереження [5].

2. Розроблено архітектуру системи, яка базується на принципах відмовостійкості та гібридного збереження даних. Використання MicroSD- карти як локального буфера [1, 9] дозволило вирішити проблему втрати телеметрії при відсутності інтернет-з’єднання. Встановлено, що інтеграція локального сховища та хмарної бази даних SQL забезпечує 100% збереження критично важливої інформації про стрибки напруги.

3. Проектування апаратної частини дозволило реалізувати унікальний механізм «пробудження» системи. Завдяки використанню резистивного дільника напруги та апаратних переривань, мікроконтролер більшу частину часу перебуває в режимі Deep Sleep, що мінімізує споживання струму до мікроамперних значень. Це забезпечує автономну роботу пристрою від акумулятора типу 18650 протягом періоду до 6 діб, що повністю перекриває потреби моніторингу при віялових відключеннях.

4. У межах розробки програмного забезпечення було створено серверну частину на базі Node.js та інтерактивний Telegram-бот. Застосування асинхронного підходу в архітектурі сервера дозволило досягти високої

швидкодії: час від моменту фіксації аномалії напруги до отримання користувачем Push-сповіщення становить не більше 2 секунд. Реалізовано механізм візуалізації даних через QuickChart API, що дозволяє користувачу аналізувати динаміку напруги безпосередньо в інтерфейсі месенджера.

5. Експериментальні дослідження підтвердили точність вимірювань у діапазоні 160–260 В. Виявлено, що похибка вимірювання, зумовлена шумами АЦП та нелінійністю дільника, не перевищує 1.5%, що є припустимим показником для побутових систем моніторингу. Також було проведено стрес-тестування системи примусовим розривом з'єднання, яке підтвердило стабільність алгоритму автоматичного перепідключення та синхронізації бази даних.

6. Критичний аналіз розробки дозволив ідентифікувати слабкі місця, зокрема необхідність переходу від макетної збірки до друкованої плати (PCB) для зменшення електромагнітних наведень та підвищення механічної міцності пристрою. Визначено перспективні напрямки модернізації, серед яких — впровадження гальванічної розв'язки вимірювального вузла та інтеграція силових реле для активного захисту навантаження.

7. Практична цінність роботи полягає у створенні доступного інструменту для формування доказової бази інцидентів у мережі. Згенеровані звіти у форматі CSV можуть бути використані споживачами для подання претензій енергопостачальним компаніям щодо надання послуг неналежної якості. Розроблена система є завершеним прототипом, готовим до масштабування та впровадження в екосистеми «розумного дому». Таким чином, мета бакалаврської роботи досягнута, а всі поставлені завдання виконані у повному обсязі. Результати роботи демонструють ефективність використання сучасних IoT-технологій для розв'язання соціально значущих енергетичних проблем.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Грабко В. В. Мікропроцесорна техніка : навч. посіб. Вінниця : ВНТУ, 2018. 148 с.
2. Квасніков В. П. Методи та засоби вимірювання параметрів електричної енергії : монографія. К. : ТОВ «АЗІМУТ-Україна», 2017. 240 с.
3. Espressif Systems. ESP32 Series Datasheet. URL: https://www.espressif.com/sites/default/files/documentation/esp32_datasheet_en.pdf (дата звернення: 12.04.2026).
4. Колін О. В. Проектування систем на базі мікроконтролерів ESP32 в середовищі Arduino IDE. *Комп'ютерні системи та мережі*. 2021. № 4. С. 45–52.
5. Майко Г. В. Енергоефективні режими роботи бездротових сенсорних вузлів. *Технічні науки та технології*. 2020. № 2(20). С. 112–118.
6. Кандел Р. Node.js у дії : пер. з англ. К. : Діалектика, 2019. 416 с.
7. Маріо Кашіаро. Шаблони проектування Node.js. *Видавництво Packt*, 2020. 540 с.
8. Telegram Bot API Documentation. URL: https://core.telegram.org/bots/api (дата звернення: 15.04.2026).
9. SQL для розробників : практ. посіб. / за ред. О. В. Петренка. К. : ІТ-Книга, 2021. 312 с.
10. QuickChart API. Documentation for static charts. URL: https://quickchart.io/documentation/ (дата звернення: 20.04.2026).
11. ДСТУ 3413-96. Система сертифікації УкрСЕПРО. Порядок проведення сертифікації продукції. [Чинний від 1997-01-01]. К. : Держстандарт України, 1996. 38 с.
12. Правила улаштування електроустановок (ПУЕ). Вид. 4-те, переробл. і доповн. К. : Форт, 2017. 732 с.
13. Arduino Core for the ESP32. GitHub Repository. URL: https://github.com/espressif/arduino-esp32 (дата звернення: 05.05.2026).

14. Telegraf.js. Modern Telegram Bot Framework for Node.js. URL: https://telegraf.js.org/ (дата звернення: 02.05.2026).
15. ATmega328P Automotive Microcontrollers. Datasheet. Microchip Technology Inc., 2020. 288 p. URL: https://ww1.microchip.com/downloads/en/DeviceDoc/Atmel-7810-Automotive-Microcontrollers-ATmega328P_Datasheet.pdf (дата звернення: 09.05.2026).
16. ESP32 Series Datasheet. v4.1. Espressif Systems (Shanghai) Co., Ltd., 2023. 65 p. URL: https://documentation.espressif.com/esp32_datasheet_en.pdf (дата звернення: 09.05.2026).
17. Raspberry Pi Documentation. Technical specifications and hardware guides. Raspberry Pi Ltd. URL: https://datasheets.raspberrypi.com/ (дата звернення: 09.05.2026).
18. ESP32 Arduino Core’s documentation. Official documentation for Arduino-ESP32 framework. Espressif Systems. URL: https://docs.espressif.com/projects/arduino-esp32/en/latest/ (дата звернення: 09.05.2026).
19. Brian Lough. Universal Arduino Telegram Bot. Library for communicating with the Telegram Bot API. GitHub repository. URL: https://github.com/witnessmenow/Universal-Arduino-Telegram-Bot (дата звернення: 09.05.2026).

Додаток А Файл server.js

```
process.env.NTBA_FIX_319 = 1;
process.env.NTBA_FIX_350 = 1;

const express = require('express');
const sqlite3 = require('sqlite3').verbose();
const TelegramBot = require('node-telegram-bot-api');
const QuickChart = require('quickchart-js');
const axios = require('axios');
const https = require('https');
const fs = require('fs');

const API_SECRET = "DIPLOMA_SECURE_KEY_2026_XYZ";

// --- КОНФІГУРАЦІЯ ГЛОБАЛЬНОГО ОТА ОНОВЛЕННЯ ---
// Змінюй targetVersion тут, коли компілюєш нову прошивку.
let globalOTAConfig = {
    targetVersion: 1.0,
    updateUrl: "https://192.168.1.101/firmware_1_1.bin" // Замінити на
актуальне ім'я файлу при оновленні
};

// Жорсткі пороги напруги
const VOLTAGE_MAX = 253.0;
const VOLTAGE_MIN = 190.0;
const ALERT_COOLDOWN_MS = 10 * 60 * 1000;

let lastAlertTime = {};

// Токен бота
const token = '8721315744:AAGptSZMTxWRZWQQLUQMt-CbFh8pDUiQD5E';
const bot = new TelegramBot(token, { polling: true });
const app = express();
app.use(express.json());

// Стан користувачів (для вводу кастомних дат)
let userStates = {};

// Словник для зберігання останнього відомого стану КОЖНОГО пристрою
let lastStatus = {};

// --- ІНІЦІАЛІЗАЦІЯ БАЗИ ДАНИХ ---
const db = new sqlite3.Database('./power_log.db', (err) => {
    if (err) console.error('[БД ПОМИЛКА]', err.message);
    else {
        console.log('[БД] Підключено.');

        db.run(`CREATE TABLE IF NOT EXISTS logs (
            mac TEXT,
            timestamp TEXT,
            voltage REAL,
            status INTEGER,
            UNIQUE(mac, timestamp)
        )`);
```

```
        db.run(`CREATE TABLE IF NOT EXISTS links (
            chat_id TEXT,
            mac TEXT,
            device_name TEXT,
            UNIQUE(chat_id, mac)
        )`);
    }
});

// --- API ДЛЯ ПРИЙОМУ ДАНИХ ТА КЕРУВАННЯ OTA ---
app.post('/api/log', (req, res) => {
    // Витягуємо параметри, включаючи версію плати
    const { api_key, mac, version, timestamp, voltage, status } =
req.body;

    if (!api_key || api_key !== API_SECRET) {
        console.warn(`[КІБЕРБЕЗПЕКА] Блокування атаки! Відхилено запит з
IP: ${req.ip}`);
        return res.status(401).json({ error: "Unauthorized: Invalid API
Key" });
    }

    if (!mac || !timestamp || voltage === undefined || status ===
undefined) {
        return res.status(400).json({ error: "Невірний формат JSON" });
    }

    // --- ЛОГІКА ПЕРЕВІРКИ МАСОВОГО ОНОВЛЕННЯ ---
    let needsUpdate = false;
    let urlToFetch = "";

    // Якщо плата надіслала версію і вона менша за ту, що вказана в
globalOTAConfig
    if (version && parseFloat(version) < globalOTAConfig.targetVersion) {
        needsUpdate = true;
        urlToFetch = globalOTAConfig.updateUrl;
        console.log(`[OTA ФЛІТ] Пристрій ${mac} потребує оновлення
(v${version} -> v${globalOTAConfig.targetVersion})`);
    }

    // --- ЛОГІКА АКТИВНОГО СПОВІЩЕННЯ ПРО АНОМАЛІЇ ---
    if (status === 1) {
        let alertMsg = null;
        if (voltage > VOLTAGE_MAX) {
            alertMsg = `⚠ <b>УВАГА: ВИСОКА НАПРУГА!</b>\nЗафіксовано:
<code>${voltage}В</code>. Відключіть чутливу техніку!`;
        } else if (voltage < VOLTAGE_MIN && voltage > 50.0) {
            alertMsg = `📉 <b>УВАГА: НИЗЬКА НАПРУГА!</b>\nЗафіксовано:
<code>${voltage}В</code>.`;
        }

        if (alertMsg) {
            const now = Date.now();
            // Шукаємо користувачів, прив'язаних САМЕ ДО ЦЬОГО пристрою
(Виправлено баг БД)
            db.all("SELECT chat_id, device_name FROM links WHERE mac =
?", [mac], (err, rows) => {
```

```
                if (!err && rows) {
                    rows.forEach(user => {
                        const timeSinceLast = now -
(lastAlertTime[user.chat_id] || 0);
                        if (timeSinceLast > ALERT_COOLDOWN_MS) {
                            const finalMsg = `🔑 <b>Об'єкт:
${user.device_name}</b>\n` + alertMsg;

                            bot.sendMessage(user.chat_id, finalMsg, {
parse_mode: 'HTML' })
                                .then(() => {
                                    lastAlertTime[user.chat_id] = now;
                                    console.log(`[ALERT] Надіслано
користувачу ${user.chat_id} (Пристрій: ${mac})`);
                                })
                                .catch(e => console.error(`[TELEGRAM
ERROR] ${e.message}`));
                        }
                    });
                } else if (err) {
                    console.error(`[БД ПОМИЛКА ALERT] ${err.message}`);
                }
            });
        }
    }

    // --- ЗАПИС У БАЗУ ДАНИХ ---
    db.run(`INSERT INTO logs (mac, timestamp, voltage, status) VALUES (?,
?, ?, ?)`,
    [mac, timestamp, voltage, status], function(err) {
        if (err) {
            if (err.code === 'SQLITE_CONSTRAINT') {
                // Віддаємо статус оновлення навіть якщо дані дублюються
                return res.status(200).json({
                    status: "ignored_duplicate",
                    update_available: needsUpdate,
                    update_url: urlToFetch
                });
            }
            return res.status(500).json({ error: err.message });
        }

        console.log(`[ЗАПИС] Пристрій: ${mac} | v${version || '1.0'} |
${timestamp} | ${voltage}В | Стан: ${status}`);

        // --- ЛОГІКА ТАРГЕТОВАНИХ СПОВІЩЕНЬ (УВІМКНЕНО/ВИМКНЕНО) ---
        if (lastStatus[mac] !== undefined && lastStatus[mac] !== status)
{
            db.all(`SELECT chat_id, device_name FROM links WHERE mac =
?`, [mac], (err, users) => {
                if (err) return;

                users.forEach(user => {
                    let msg = status === 1
                        ? `🟢 <b>Світло з'явилося!</b>
[${user.device_name}]\nНапруга: ${voltage} В\nЧас: ${timestamp}`
```

```
                        : `🔴 <b>Світло зникло!</b>
[${user.device_name}]\nЧас: ${timestamp}`;

                    bot.sendMessage(user.chat_id, msg, { parse_mode:
'HTML' })
                    .catch((err) => {
                        if (err.response && err.response.statusCode ===
403) {
                            console.log(`[ОЧИЩЕННЯ БД] Чат
${user.chat_id} заблокував бота. Безповоротне видалення зв'язків...`);
                            db.run(`DELETE FROM links WHERE chat_id = ?`,
[user.chat_id], (dbErr) => {
                                if (!dbErr) console.log(`[ОЧИЩЕННЯ БД]
Чат ${user.chat_id} успішно видалено.`);
                            });
                        } else {
                            console.error(`[ТЕЛЕГРАМ ПОМИЛКА]`,
err.message);
                        }
                    });
                });
            });
        }

        lastStatus[mac] = status;

        // Відправляємо відповідь з командами для OTA
        res.status(200).json({
            status: "success",
            update_available: needsUpdate,
            update_url: urlToFetch
        });
    });
});

// --- ТЕЛЕГРАМ ІНТЕРФЕЙС ТА ЛОГІКА ---
const mainMenu = {
    reply_markup: {
        keyboard: [
            [{ text: '⚡ Статус мережі' }, { text: '📊 Графік' }],
            [{ text: 'ℹ️ Мої пристрої' }]
        ],
        resize_keyboard: true,
        is_persistent: true
    }
};

bot.onText(/\/start/, (msg) => {
    const chatId = msg.chat.id.toString();
    const welcomeText = `📡 <b>Система моніторингу мережі</b>\n\nЩоб
почати роботу, вам потрібно прив'язати пристрій до вашого акаунту.
Використайте команду формату:\n<code>/link MAC_АДРЕСА
НАЗВА</code>\n\n<b>Приклад:</b>\n<code>/link F0:24:F9:0D:9E:F4
Дім</code>`;
    bot.sendMessage(chatId, welcomeText, Object.assign({ parse_mode:
'HTML' }, mainMenu));
```

```
});

bot.onText(/\/link (.+) (.+)/, (msg, match) => {
    const chatId = msg.chat.id.toString();
    const mac = match[1].trim();
    const name = match[2].trim();

    db.run(`INSERT OR REPLACE INTO links (chat_id, mac, device_name)
VALUES (?, ?, ?)`,
    [chatId, mac, name], function(err) {
        if (err) {
            bot.sendMessage(chatId, "Помилка прив'язки пристрою.");
        } else {
            bot.sendMessage(chatId, `✅ Пристрій <b>${name}</b> (${mac})
успішно прив'язано до вашого чату! Тепер ви отримуватимете з нього
сповіщення.`, { parse_mode: 'HTML' });
        }
    });
});

bot.on('message', (msg) => {
    const chatId = msg.chat.id.toString();
    const text = msg.text;

    if (text.startsWith('/')) return;

   if (text === 'ℹ️ Мої пристрої') {
        db.all(`SELECT mac, device_name FROM links WHERE chat_id = ?`,
[chatId], (err, rows) => {
            let response = "<b>Ваші пристрої:</b>\n\n";
            let inlineKeyboard = [];

            if (rows.length === 0) {
                response = "У вас немає прив'язаних пристроїв.";
            } else {
                response += rows.map(r => `🔹 <b>${r.device_name}</b>\n└
<code>${r.mac}</code>`).join('\n\n');
                inlineKeyboard.push([{ text: '🗑 Видалити пристрій',
callback_data: 'action_delete_menu' }]);
            }

            inlineKeyboard.push([{ text: '➕ Додати пристрій',
callback_data: 'action_add_device' }]);

            bot.sendMessage(chatId, response, {
                parse_mode: 'HTML',
                reply_markup: { inline_keyboard: inlineKeyboard }
            });
        });
        return;
    }

    if (text === '⚡ Статус мережі') {
        db.get(`SELECT mac, device_name FROM links WHERE chat_id = ?
LIMIT 1`, [chatId], (err, link) => {
```

```
            if (!link) return bot.sendMessage(chatId, "Спочатку
прив'яжіть пристрій командою <code>/link MAC НАЗВА</code>", { parse_mode:
'HTML' });

            db.get(`SELECT timestamp, voltage, status FROM logs WHERE mac
= ? ORDER BY timestamp DESC LIMIT 1`, [link.mac], (err, row) => {
                if (row) {
                    const stateStr = row.status === 1 ? "🟢 Є" : "🔴
Немає";
                    bot.sendMessage(chatId, `⚡ <b>Об'єкт:
${link.device_name}</b>\nМережа: ${stateStr}\nНапруга: ${row.voltage}
В\nОстанній замір: ${row.timestamp}`, { parse_mode: 'HTML' });
                } else {
                    bot.sendMessage(chatId, "Пристрій ще не надіслав
жодних даних на сервер.");
                }
            });
        });
    }

    if (text === '📊 Графік') {
        db.get(`SELECT mac FROM links WHERE chat_id = ? LIMIT 1`,
[chatId], (err, link) => {
            if (!link) return bot.sendMessage(chatId, "Спочатку
прив'яжіть пристрій командою /link");

            const inlineKeyboard = {
                reply_markup: {
                    inline_keyboard: [
                        [{ text: '📅 Останній день', callback_data:
`chart_day_${link.mac}` }],
                        [{ text: '🗓 Останній тиждень', callback_data:
`chart_week_${link.mac}` }],
                        [{ text: '🗓 Останній місяць', callback_data:
`chart_month_${link.mac}` }],
                        [{ text: '⚙️ Власний діапазон', callback_data:
`chart_custom_${link.mac}` }]
                    ]
                }
            };
            bot.sendMessage(chatId, "📊 Оберіть період для аналітики:",
inlineKeyboard);
        });
        return;
    }

    if (userStates[chatId] && userStates[chatId].action ===
'awaiting_dates') {
        const mac = userStates[chatId].mac;

        const dateRegexDay = /^(\d{4}-\d{2}-\d{2})(?: (\d{4}-\d{2}-
\d{2}))?$/;
        const dateRegexMonth = /^(\d{4}-\d{2})(?: (\d{4}-\d{2}))?$/;

        const matchDay = text.match(dateRegexDay);
```

```
        const matchMonth = text.match(dateRegexMonth);

        let startDateStr = '';
        let endDateStr = '';
        let titleRange = '';

        if (matchDay) {
            startDateStr = matchDay[1] + " 00:00:00";
            endDateStr = (matchDay[2] ? matchDay[2] : matchDay[1]) + "
23:59:59";
            titleRange = matchDay[2] ? `${matchDay[1]} - ${matchDay[2]}`
: matchDay[1];
        } else if (matchMonth) {
            startDateStr = matchMonth[1] + "-01 00:00:00";
            const endMonthStr = matchMonth[2] ? matchMonth[2] :
matchMonth[1];

            const [y, m] = endMonthStr.split('-');
            const lastDay = new Date(y, parseInt(m), 0).getDate();

            endDateStr = `${y}-${m}-${lastDay} 23:59:59`;
            titleRange = matchMonth[2] ? `${matchMonth[1]} -
${matchMonth[2]}` : matchMonth[1];
        } else {
            delete userStates[chatId];
            return bot.sendMessage(chatId, "❌ Невірний формат.
Очікування дати скасовано.\nСпробуйте викликати меню графіка ще раз.", {
parse_mode: 'HTML' });
        }

        delete userStates[chatId];
        bot.sendMessage(chatId, "⏳ Генерую графік...");
        generateAndSendChart(chatId, mac, startDateStr, endDateStr,
`Діапазон: ${titleRange}`);
        return;
    }
});

bot.on('callback_query', (query) => {
    const chatId = query.message.chat.id.toString();
    const data = query.data;
    const messageId = query.message.message_id;

    bot.answerCallbackQuery(query.id);

    if (data === 'action_add_device') {
        bot.sendMessage(chatId, "✏️ <b>Додавання пристрою</b>\n\nЩоб
прив'язати новий логер, відправте мені команду у форматі:\n<code>/link
MAC_АДРЕСА НАЗВА</code>\n\n<b>Приклад:</b>\n<code>/link F0:24:F9:0D:9E:F4
Дача</code>", { parse_mode: 'HTML' });
        return;
    }

    if (data === 'action_delete_menu') {
        db.all(`SELECT mac, device_name FROM links WHERE chat_id = ?`,
[chatId], (err, rows) => {
```

```
            if (rows.length === 0) return bot.sendMessage(chatId, "Немає
пристроїв для видалення.");

            const deleteButtons = rows.map(r => [{ text: `❌ Відв'язати:
${r.device_name}`, callback_data: `delete_mac_${r.mac}` }]);
            deleteButtons.push([{ text: '🔙 Скасувати', callback_data:
'action_cancel_delete' }]);

            bot.editMessageText("Оберіть пристрій, який хочете
безповоротно відв'язати від вашого акаунту:", {
                chat_id: chatId,
                message_id: messageId,
                reply_markup: { inline_keyboard: deleteButtons }
            });
        });
        return;
    }

    if (data === 'action_cancel_delete') {
        bot.deleteMessage(chatId, messageId).catch(()=>{});
        return;
    }

    if (data.startsWith('delete_mac_')) {
        const macToDelete = data.replace('delete_mac_', '');
        db.run(`DELETE FROM links WHERE chat_id = ? AND mac = ?`,
[chatId, macToDelete], function(err) {
            if (err) {
                bot.sendMessage(chatId, "❌ Помилка бази даних під час
видалення.");
            } else {
                bot.editMessageText(`✅ Пристрій з MAC-адресою
<code>${macToDelete}</code> успішно відв'язано.\nБільше ви не
отримуватимете з нього сповіщень.`, {
                    chat_id: chatId,
                    message_id: messageId,
                    parse_mode: 'HTML'
                });
            }
        });
        return;
    }
    if (!data.startsWith('chart_')) return;

    const parts = data.split('_');
    const period = parts[1];
    const mac = parts.slice(2).join('_');

    if (period === 'custom') {
        userStates[chatId] = { action: 'awaiting_dates', mac: mac };
        bot.sendMessage(chatId, "Введіть діапазон тексту:\n\n<b>Точні
дати:</b>\nОдин день: <code>YYYY-MM-DD</code>\nДекілька: <code>YYYY-MM-DD
YYYY-MM-DD</code>\n\n<b>Цілі місяці:</b>\nОдин місяць: <code>YYYY-
MM</code>\nДекілька: <code>YYYY-MM YYYY-MM</code>", { parse_mode: 'HTML'
});
        bot.deleteMessage(chatId, messageId).catch(()=>{});
```

```
        return;
    }

    const now = new Date();
    let startDateStr = '';
    let title = '';

    if (period === 'day') {
        const dayAgo = new Date(now.getTime() - (24 * 60 * 60 * 1000));
        startDateStr = dayAgo.toISOString().replace('T', ' ').slice(0,
19);
        title = `Напруга (Останні 24 години)`;
    } else if (period === 'week') {
        const weekAgo = new Date(now.getTime() - (7 * 24 * 60 * 60 *
1000));
        startDateStr = weekAgo.toISOString().replace('T', ' ').slice(0,
19);
        title = `Напруга (Останні 7 днів)`;
    } else if (period === 'month') {
        const monthAgo = new Date(now.getTime() - (30 * 24 * 60 * 60 *
1000));
        startDateStr = monthAgo.toISOString().replace('T', ' ').slice(0,
19);
        title = `Напруга (Останні 30 днів)`;
    }

    const endDateStr = now.toISOString().replace('T', ' ').slice(0, 19);

    bot.deleteMessage(chatId, messageId).catch(()=>{});
    generateAndSendChart(chatId, mac, startDateStr, endDateStr, title);
});

// --- ЯДРО ГЕНЕРАЦІЇ (POST-РЕНДЕРИНГ ЧЕРЕЗ AXIOS) ---
function generateAndSendChart(chatId, mac, startDate, endDate, title) {
    const sqlPrev = `SELECT voltage, status FROM logs WHERE mac = ? AND
timestamp < ? ORDER BY timestamp DESC LIMIT 1`;
    const sqlMain = `SELECT timestamp, voltage, status FROM logs WHERE
mac = ? AND timestamp >= ? AND timestamp <= ? ORDER BY timestamp ASC`;

    db.get(sqlPrev, [mac, startDate], (err, prevRow) => {
        db.all(sqlMain, [mac, startDate, endDate], (err, rows) => {
            if (err) return bot.sendMessage(chatId, `📉 Помилка бази
даних.`);

            let virtualRows = [...rows];

            if (prevRow) {
                if (virtualRows.length === 0) {
                    virtualRows.push({ timestamp: startDate, voltage:
prevRow.voltage, status: prevRow.status });
                } else {
                    let firstLogMs = new
Date(virtualRows[0].timestamp.replace(' ', 'T')).getTime();
                    let startMs = new Date(startDate.replace(' ',
'T')).getTime();
                    if (firstLogMs - startMs > 720000) {
```

```
                        virtualRows.unshift({ timestamp: startDate,
voltage: prevRow.voltage, status: prevRow.status });
                    }
                }
            }

            if (virtualRows.length > 0) {
                let lastLogMs = new Date(virtualRows[virtualRows.length -
1].timestamp.replace(' ', 'T')).getTime();
                let endMs = Math.min(new Date(endDate.replace(' ',
'T')).getTime(), Date.now());

                if (endMs - lastLogMs > 720000) {
                    let pad = (n) => n < 10 ? '0' + n : n;
                    let d = new Date(endMs);
                    let endStr = `${d.getFullYear()}-
${pad(d.getMonth()+1)}-${pad(d.getDate())}
${pad(d.getHours())}:${pad(d.getMinutes())}:${pad(d.getSeconds())}`;

                    let lastRow = virtualRows[virtualRows.length - 1];
                    virtualRows.push({ timestamp: endStr, voltage:
lastRow.voltage, status: lastRow.status });
                }
            }

            if (virtualRows.length === 0) return bot.sendMessage(chatId,
`📉 Немає даних для побудови графіка.`);

            let expandedRows = [];
            for (let i = 0; i < virtualRows.length; i++) {
                expandedRows.push(virtualRows[i]);

                if (i < virtualRows.length - 1) {
                    let t1 = new Date(virtualRows[i].timestamp.replace('
', 'T')).getTime();
                    let t2 = new
Date(virtualRows[i+1].timestamp.replace(' ', 'T')).getTime();
                    let diffMs = t2 - t1;

                    if (diffMs > 720000) {
                         let step = 10 * 60 * 1000;
                          let isDead = diffMs > 3600000;

                          for (let time = t1 + step; time < t2; time +=
step) {
                            let d = new Date(time);
                            let pad = (n) => n < 10 ? '0' + n :
n;
                            let tsStr = `${d.getFullYear()}-
${pad(d.getMonth()+1)}-${pad(d.getDate())}
${pad(d.getHours())}:${pad(d.getMinutes())}:${pad(d.getSeconds())}`;

                        expandedRows.push({
                        timestamp: tsStr,
                        voltage: isDead ? 0 : virtualRows[i].voltage,
                        status: isDead ? 0 : virtualRows[i].status
                    });
```

```
                }
                }
                }
            }

            const MAX_POINTS = 200;
            let processedRows = expandedRows;

            if (expandedRows.length > MAX_POINTS) {
                const bucketSize = Math.ceil(expandedRows.length /
MAX_POINTS);
                processedRows = [];

                for (let i = 0; i < expandedRows.length; i += bucketSize)
{
                    const bucket = expandedRows.slice(i, i + bucketSize);
                    const hasBlackout = bucket.some(r => r.status === 0
|| r.voltage < 100);

                    if (hasBlackout) {
                        processedRows.push({ timestamp:
bucket[0].timestamp, voltage: 0 });
                    } else {
                        const maxV = Math.max(...bucket.map(r =>
r.voltage));
                        const minV = Math.min(...bucket.map(r =>
r.voltage));

                        if (maxV >= 245) processedRows.push({ timestamp:
bucket[0].timestamp, voltage: maxV });
                        else if (minV > 0 && minV <= 190)
processedRows.push({ timestamp: bucket[0].timestamp, voltage: minV });
                        else {
                            const avgV = bucket.reduce((sum, r) => sum +
r.voltage, 0) / bucket.length;
                            processedRows.push({ timestamp:
bucket[0].timestamp, voltage: Math.round(avgV) });
                        }
                    }
                }
            } else {
                processedRows = expandedRows.map(r => ({
                    timestamp: r.timestamp,
                    voltage: (r.status === 1 && r.voltage >= 100) ?
r.voltage : 0
                }));
            }

            const labels = processedRows.map(r => `${r.timestamp.split('
')[0].slice(5)} ${r.timestamp.split(' ')[1].slice(0, 5)}`);
            const chartData = processedRows.map(r => r.voltage);

            const pointColors = processedRows.map(r => {
                if (r.voltage >= 245 || (r.voltage > 0 && r.voltage <=
190)) return 'rgba(255, 0, 0, 1)';
                if (r.voltage === 0) return 'rgba(0, 0, 0, 1)';
                return 'rgba(54, 162, 235, 1)';
```

```
            });

            const pointRadii = processedRows.map(r => {
                if (r.voltage >= 245 || (r.voltage > 0 && r.voltage <=
190)) return 4;
                if (r.voltage === 0) return 3;
                return processedRows.length > 100 ? 0 : 2;
            });

            const customGridLines = [160, 170, 180, 190, 210, 220, 230,
240].map(val => ({
                type: 'line',
                mode: 'horizontal',
                scaleID: 'y-axis-0',
                value: val,
                borderColor: 'rgba(0, 0, 0, 0.1)',
                borderWidth: 1,
                label: {
                    enabled: true,
                    content: val.toString(),
                    position: 'left',
                    backgroundColor: 'rgba(255, 255, 255, 0.7)',
                    fontColor: '#555',
                    fontSize: 11,
                    fontStyle: 'normal',
                    xPadding: 4,
                    yPadding: 2,
                    cornerRadius: 3
                }
            }));

            const chartConfig = {
                type: 'line',
                data: {
                    labels: labels,
                    datasets: [{
                        label: title,
                        data: chartData,
                        borderColor: 'rgb(54, 162, 235)',
                        backgroundColor: 'rgba(54, 162, 235, 0.2)',
                        borderWidth: 2,
                        pointBackgroundColor: pointColors,
                        pointBorderColor: pointColors,
                        pointRadius: pointRadii,
                        fill: true,
                        stepped: 'before'
                    }]
                },
                options: {
                    scales: {
                        yAxes: [{ ticks: { min: 0, max: 250, stepSize:
50, fontSize: 14 } }],
                        xAxes: [{ ticks: { autoSkip: true, maxTicksLimit:
24, maxRotation: 45, minRotation: 45, fontSize: 12 } }]
                    },
                    legend: { labels: { fontSize: 16, fontStyle: 'bold' }
},
```

```
                    annotation: {
                        drawTime: 'beforeDatasetsDraw',
                        annotations: customGridLines
                    }
                }
            };

            bot.sendMessage(chatId, "⏳ Будую деталізований графік з
інтерполяцією...").then(sentMsg => {
                axios.post('https://quickchart.io/chart', {
                    chart: chartConfig,
                    width: 1000,
                    height: 500,
                    backgroundColor: 'white',
                    devicePixelRatio: 1.5
                }, {
                    responseType: 'arraybuffer'
                })
                .then(response => {
                    const imageBuffer = Buffer.from(response.data,
'binary');
                    bot.sendPhoto(chatId, imageBuffer, {}, { filename:
'chart.png', contentType: 'image/png' })
                        .then(() => bot.deleteMessage(chatId,
sentMsg.message_id))
                        .catch(e => console.error('[ТЕЛЕГРАМ ПОМИЛКА]',
e.message));
                })
                .catch(err => {
                    bot.sendMessage(chatId, "❌ Сталася помилка при
рендерингу графіка на сервері.");
                });
            });
        });
    });
}

// Читаємо згенеровані ключі
const privateKey = fs.readFileSync('server.key', 'utf8');
const certificate = fs.readFileSync('server.cert', 'utf8');
const credentials = { key: privateKey, cert: certificate };

// Піднімаємо HTTPS сервер
const httpsServer = https.createServer(credentials, app);

httpsServer.listen(443, '0.0.0.0', () => {
    console.log('--- ЗАХИЩЕНИЙ HTTPS СЕРВЕР ЗАПУЩЕНО (Порт 443) ---');
});
```

Додаток Б Файл main.cpp

```
#include <Arduino.h>
#include <SPI.h>
#include <Wire.h>
#include <WiFi.h>
#include <WiFiManager.h>
#include <HTTPClient.h>
#include <RTClib.h>
#include <SD.h>
#include "EmonLib.h"
#include <driver/rtc_io.h>
#include <WiFiClientSecure.h>
#include <HTTPUpdate.h>
#include <ArduinoJson.h>

const char* serverUrl = "https://192.168.1.101:443/api/log";
const float FIRMWARE_VERSION = 1.0; // Поточна версія прошивки для
системи OTA

// --- ПІНИ ТА НАЛАШТУВАННЯ ---
#define SD_CS_PIN 5
#define VOLTAGE_PIN 34
#define DUMMY_CURRENT_PIN 35
const float CALIBRATION_FACTOR = 118.0;
const String API_SECRET = "DIPLOMA_SECURE_KEY_2026_XYZ";

RTC_DS3231 rtc;
EnergyMonitor emon;

// --- ЗМІННІ ДЛЯ ЛОГІКИ ---
bool currentPowerState = false;
bool lastPowerState = false;
unsigned long lastLogTime = 0;
unsigned long lastSyncTime = 0;
unsigned long lastWiFiCheckTime = 0;

const unsigned long LOG_INTERVAL = 600000;
const unsigned long SYNC_INTERVAL = 300000;
const float VOLTAGE_THRESHOLD = 15.0;
const unsigned long ANOMALY_COOLDOWN = 60000;
float lastLoggedVoltage = 0.0;

const char* ntpServer = "pool.ntp.org";

// --- ГЕНЕРАЦІЯ ІМЕНІ ФАЙЛУ АРХІВУ ---
String getArchiveFileName() {
  DateTime now = rtc.now();
  char buffer[20];
  sprintf(buffer, "/arc_%04d_%02d.csv", now.year(), now.month());
  return String(buffer);
}

// --- ОТРИМАННЯ ЧАСУ ---
String getTimeString() {
  DateTime now = rtc.now();
```

```
  char buffer[25];
  sprintf(buffer, "%04d-%02d-%02d %02d:%02d:%02d",
          now.year(), now.month(), now.day(),
          now.hour(), now.minute(), now.second());
  return String(buffer);
}

// --- ФУНКЦІЯ ВИКОНАННЯ OTA ОНОВЛЕННЯ ---
void performOTAUpdate(String url) {
  WiFiClientSecure client;
  client.setInsecure(); // Ігноруємо перевірку сертифіката для
завантаження

  Serial.println("[OTA] Запуск оновлення прошивки з: " + url);

  t_httpUpdate_return ret = httpUpdate.update(client, url);

  switch (ret) {
    case HTTP_UPDATE_FAILED:
      Serial.printf("[OTA ПОМИЛКА] (%d): %s\n",
httpUpdate.getLastError(), httpUpdate.getLastErrorString().c_str());
      break;
    case HTTP_UPDATE_NO_UPDATES:
      Serial.println("[OTA] Оновлень немає.");
      break;
    case HTTP_UPDATE_OK:
      Serial.println("[OTA] Оновлення успішне! Плата
перезавантажується...");
      break;
  }
}

// --- ГОЛОВНА ФУНКЦІЯ ЛОГУВАННЯ ТА ВІДПРАВКИ ---
void logData(String timestamp, float voltage, int status) {
  if (!SD.exists("/")) {
    Serial.println("[СИСТЕМА] SD-карта відвалилася! Спроба
реініціалізації...");
    SD.end();
    delay(100);
    if (!SD.begin(SD_CS_PIN)) {
      Serial.println("[КРИТИЧНО] Не вдалося відновити SD-карту!");
    } else {
      Serial.println("[СИСТЕМА] SD-карту успішно відновлено!");
    }
  }

  String arcName = getArchiveFileName();
  File arcFile = SD.open(arcName, FILE_APPEND);
  if (arcFile) {
    if (arcFile.size() == 0) arcFile.println("Timestamp,Voltage,Status");
    arcFile.println(timestamp + "," + String(voltage, 1) + "," +
String(status));
    arcFile.close();
  } else {
    Serial.println("[ПОМИЛКА SD] Не можу відкрити файл архіву для
запису!");
  }
```

```
  bool sentSuccessfully = false;
  String macAddress = WiFi.macAddress();

  if (WiFi.status() == WL_CONNECTED) {
    // 1. СТВОРЮЄМО КРИПТОГРАФІЧНОГО КЛІЄНТА
    WiFiClientSecure client;
    client.setInsecure();

    HTTPClient http;
    http.begin(client, serverUrl);
    http.addHeader("Content-Type", "application/json");
    http.setTimeout(5000);

    // 2. ІН'ЄКЦІЯ СЕКРЕТНОГО КЛЮЧА ТА ВЕРСІЇ ПРОШИВКИ
    String jsonPayload = "{\"api_key\":\"" + API_SECRET + "\",\"mac\":\""
+ macAddress + "\",\"version\":" + String(FIRMWARE_VERSION, 1) +
",\"timestamp\":\"" + timestamp + "\",\"voltage\":" + String(voltage, 1)
+ ",\"status\":" + String(status) + "}";

    int httpCode = http.POST(jsonPayload);

    if (httpCode == 200 || httpCode == 201) {
      sentSuccessfully = true;
      Serial.println("[КІБЕРБЕЗПЕКА] Дані зашифровано і відправлено. MAC:
" + macAddress);

      // --- ПЕРЕВІРКА ВІДПОВІДІ СЕРВЕРА НА НАЯВНІСТЬ ОНОВЛЕНЬ ---
      String response = http.getString();
      StaticJsonDocument<256> doc;
      DeserializationError error = deserializeJson(doc, response);

      if (!error && doc["update_available"] == true) {
        String update_url = doc["update_url"].as<String>();
        Serial.println("[OTA] Сервер ініціював оновлення!");
        performOTAUpdate(update_url);
      }

    } else {
      Serial.printf("[TLS ПОМИЛКА] Збій підключення. HTTP Код: %d\n",
httpCode);
      if (httpCode < 0) {
        Serial.printf("[ДЕТАЛІ] %s\n",
http.errorToString(httpCode).c_str());
      }
    }
    http.end();
    client.stop();
  } else {
    Serial.println("[МЕРЕЖА] Немає Wi-Fi з'єднання.");
  }

  if (!sentSuccessfully) {
    File qFile = SD.open("/queue.csv", FILE_APPEND);
    if (qFile) {
      qFile.println(timestamp + "," + String(voltage, 1) + "," +
String(status));
```

```
      qFile.close();
      Serial.println("[ЧЕРГА] Дані збережено локально для майбутньої
синхронізації.");
    } else {
      Serial.println("[ПОМИЛКА SD] Не можу записати в чергу
/queue.csv!");
    }
  }
}


// --- ФУНКЦІЯ ПІДРАХУНКУ ЗАПИСІВ ---
int getQueueCount() {
  if (!SD.exists("/queue.csv")) return 0;
  File qFile = SD.open("/queue.csv", FILE_READ);
  if (!qFile) return 0;

  int count = 0;
  while (qFile.available()) {
    if (qFile.read() == '\n') count++;
  }
  qFile.close();
  return count;
}


// --- ФУНКЦІЯ ПРОШТОВХУВАННЯ ЧЕРГИ (ПОВНА ОБРОБКА) ---
void processQueue() {
  if (WiFi.status() != WL_CONNECTED || !SD.exists("/queue.csv")) return;

  File qFile = SD.open("/queue.csv", FILE_READ);
  if (!qFile) return;

  File tempFile = SD.open("/temp.csv", FILE_WRITE);
  if (!tempFile) {
    qFile.close();
    return;
  }

  Serial.printf("\n[СИНХРОНІЗАЦІЯ] Виявлено %d записів у черзі.
Запуск...\n", getQueueCount());

  int sentCount = 0;
  bool serverError = false; // ЗАПОБІЖНИК: Блокує цикл, якщо сервер
перестав відповідати
  String macAddress = WiFi.macAddress();

  while(qFile.available()) {
    String line = qFile.readStringUntil('\n');
    line.trim();
    if (line.length() < 10) continue;

    if (!serverError) {
      int firstComma = line.indexOf(',');
      int secondComma = line.indexOf(',', firstComma + 1);
      if (firstComma == -1 || secondComma == -1) {
        tempFile.println(line);
```

```
        continue;
      }

      String ts = line.substring(0, firstComma);
      String v_str = line.substring(firstComma + 1, secondComma);
      String s_str = line.substring(secondComma + 1);

      Serial.printf("[ПАМ'ЯТЬ] Free Heap перед SSL: %d байт\n",
ESP.getFreeHeap());

      WiFiClientSecure client;
      client.setInsecure();
      HTTPClient http;
      http.begin(client, serverUrl);
      http.addHeader("Content-Type", "application/json");

      String json = "{\"api_key\":\"" + API_SECRET + "\",\"mac\":\"" +
macAddress + "\",\"version\":" + String(FIRMWARE_VERSION, 1) +
",\"timestamp\":\"" + ts + "\",\"voltage\":" + v_str + ",\"status\":" +
s_str + "}";
      int httpCode = http.POST(json);

      http.end();
      client.stop();

      delay(500); // Обов'язкова стабілізаційна пауза

      if (httpCode == 200 || httpCode == 201) {
        sentCount++;
      } else {
        Serial.println("[СИНХРОНІЗАЦІЯ] Збій сервера. Зупинка
відправки.");
        tempFile.println(line);
        serverError = true; // Спрацьовує запобіжник: решта файлу просто
перепишеться в temp
      }
    } else {
      tempFile.println(line);
    }

    yield();
  }

  qFile.close();
  tempFile.close();

  SD.remove("/queue.csv");
  SD.rename("/temp.csv", "/queue.csv");

  File checkSize = SD.open("/queue.csv", FILE_READ);
  if (checkSize) {
    if (checkSize.size() == 0) {
      checkSize.close();
      SD.remove("/queue.csv");
      Serial.printf("[СИНХРОНІЗАЦІЯ] Успішно відправлено %d записів.
Чергу очищено.\n", sentCount);
    } else {
```

```
      checkSize.close();
    }
  }
}

void setup() {
  Serial.begin(115200);
  delay(1000);

  pinMode(33, INPUT_PULLDOWN);

  Wire.begin(21, 22);
  rtc.begin();
  SD.begin(SD_CS_PIN);

  analogReadResolution(12);
  emon.voltage(VOLTAGE_PIN, CALIBRATION_FACTOR, 1.7);
  emon.current(DUMMY_CURRENT_PIN, 1.0);

  Serial.println("[СИСТЕМА] Прогрів цифрових фільтрів EmonLib...");
  for (int i = 0; i < 15; i++) {
    emon.calcVI(20, 2000);
    delay(10);
  }

  WiFiManager wifiManager;
  wifiManager.setConfigPortalTimeout(180);
  if (wifiManager.autoConnect("SmartSocket_Setup")) {
    configTzTime("EET-2EEST,M3.5.0/3,M10.4.0/4", ntpServer);
    struct tm timeinfo;
    if (getLocalTime(&timeinfo, 10000)) {
      rtc.adjust(DateTime(timeinfo.tm_year + 1900, timeinfo.tm_mon + 1,
timeinfo.tm_mday, timeinfo.tm_hour, timeinfo.tm_min, timeinfo.tm_sec));
    }
  }
  delay(5000); // Даємо 5 секунд на підняття мережі
  Serial.println("\n=== СИСТЕМА ГОТОВА ДО РОБОТИ ===");
}

void loop() {
  emon.calcVI(20, 2000);
  float voltage = emon.Vrms;
  if (voltage < 20.0) voltage = 0.0;

  currentPowerState = (voltage > 150.0);
  unsigned long currentMillis = millis();
  String timestamp = getTimeString();

  if (currentPowerState != lastPowerState) {
    if (currentPowerState) {
      Serial.println("\n>>> ПОДІЯ: ВІДНОВЛЕННЯ ЖИВЛЕННЯ! <<<");
      processQueue();
      logData(timestamp, voltage, 1);
    } else {
      Serial.println("\n>>> ПОДІЯ: ЗНИКНЕННЯ ЖИВЛЕННЯ! <<<");
      processQueue();
      logData(timestamp, 0.0, 0);
```

```
      delay(500);

      rtc_gpio_pullup_dis(GPIO_NUM_33);
      rtc_gpio_pulldown_en(GPIO_NUM_33);
      esp_sleep_enable_ext0_wakeup(GPIO_NUM_33, 1);

      Serial.println("[ЖИВЛЕННЯ] Засинаю. Надобраніч.");
      esp_deep_sleep_start();
    }
    lastLoggedVoltage = currentPowerState ? voltage : 0.0;
    lastPowerState = currentPowerState;
    lastLogTime = currentMillis;
  }

  else if (currentPowerState && abs(voltage - lastLoggedVoltage) >=
VOLTAGE_THRESHOLD) {
    if (currentMillis - lastLogTime >= ANOMALY_COOLDOWN) {
      Serial.printf("\n>>> АНОМАЛІЯ: Стрибок напруги! (%.1f В) <<<\n",
voltage);
      logData(timestamp, voltage, 1);
      lastLoggedVoltage = voltage;
      lastLogTime = currentMillis;
    }
  }

  else if (currentPowerState && (currentMillis - lastLogTime >=
LOG_INTERVAL)) {
    Serial.println("\n[ШТАТНИЙ ЗАПИС]");
    logData(timestamp, voltage, 1);
    lastLoggedVoltage = voltage;
    lastLogTime = currentMillis;
  }

  if (currentMillis - lastSyncTime >= SYNC_INTERVAL) {
    processQueue();
    lastSyncTime = currentMillis;
  }

  if (WiFi.status() != WL_CONNECTED && (currentMillis - lastWiFiCheckTime
>= 30000)) {
    Serial.println("[МЕРЕЖА] Wi-Fi відвалився! Примусове
перепідключення...");
    WiFi.disconnect();
    WiFi.reconnect();
    lastWiFiCheckTime = currentMillis;
  }

  delay(1000);
}
```