КИЇВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ
БУДІВНИЦТВА І АРХІТЕКТУРИ
Автоматизації і інформаційних технологій
(факультет)
Кібербезпеки та комп'ютерної інженерії
(назва випускової кафедри)

КВАЛІФІКАЦІЙНА РОБОТА

ЗДОБУВАЧА СТУПЕНЯ ВИЩОЇ ОСВІТИ БАКАЛАВР

на тему:

Розробка відмовостійкого embedded–контролера для роботизованих систем (C/C++ + RTOS)

Корнійчук Леонід Миколайович
(прізвище, ім'я та по батькові здобувача повністю)

Київ 2026

КИЇВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ
БУДІВНИЦТВА І АРХІТЕКТУРИ

Автоматизації і інформаційних технологій
(факультет)

Кібербезпеки та комп'ютерної інженерії
(назва випускової кафедри)

ЗАТВЕРДЖУЮ

Завідувач кафедри

Делембовський М.М.

"__"_____2026 року

КВАЛІФІКАЦІЙНА РОБОТА

ЗДОБУВАЧА СТУПЕНЯ ВИЩОЇ ОСВІТИ БАКАЛАВР

Розробка відмовостійкого embedded–контролера для роботизованих систем (C/C++ + RTOS)
(назва)

Я, як здобувач вищої освіти КНУБА, розумію і підтримую політику закладу з академічної доброчесності. Я не надавав(–ла) і не одержував(–ла) недозволену допомогу під час підготовки цієї роботи. Використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело

Здобувач Корнійчук Леонід Миколайович
(прізвище, ім'я та по батькові повністю)

123 Комп'ютерна інженерія
(спеціальність)

Комп'ютерні системи і мережі
(освітня програма)

Група КСМс–23

Керівник Гуменний Д. О.
(прізвище та ініціали)

доцент, кандидат технічних наук
(вчене звання, науковий ступінь)

Рецензент Варава І.А.

Ідентичність підтверджую

Київ 2026 р.

КИЇВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ
БУДІВНИЦТВА І АРХІТЕКТУРИ

Факультет: автоматизації і інформаційних технологій

Випускова кафедра: кібербезпеки та комп'ютерної інженерії

Освітній рівень: бакалавр

Спеціальність: 123 Комп'ютерна інженерія

Освітня програма: Комп'ютерні системи і мережі

ЗАТВЕРДЖУЮ

Завідувач кафедри

Делембовський М.М.

"__"_____2026 року

ЗАВДАННЯ
ДО ВИКОНАННЯ КВАЛІФІКАЦІЙНОЇ РОБОТИ
ЗДОБУВАЧА СТУПЕНЯ ВИЩОЇ ОСВІТИ БАКАЛАВР

Корнійчук Леонід Миколайович

(прізвище, ім'я та по батькові здобувача)

1. Тема роботи Розробка відмовостійкого embedded–контролера для роботизованих систем (C/C++ + RTOS)

затверджена наказом ректора КНУБА № 577/52–15/13.2/26 від 14.05.2026

2. Керівник роботи Гуменний Дмитро Олександрович, кандидат технічних наук, доцент.

(прізвище, ім'я та по батькові, науковий ступінь, вчене звання)

3. Термін подання здобувачем роботи до захисту _____

4. Зміст пояснювальної записки за розділами:

Р1. 1. Аналіз предметної області та постановка задачі

Р2. 2. Обґрунтування та розроблення рішення

Р3. 3. Технічна реалізація та тестування

5. Графічні матеріали:

Р1. Візуалізація можливостей мікроконтролера і таблиця порівнянь

Р2. Рисунки елементів модулів та блоксхеми

Р3. Рисунки результатів тестувань, виведення в консоль та веб–інтерфейс

6. Календарний план виконання роботи:

Види робіт та їх зміст	Дата виконання
Розділ 1	06.04.2026
Розділ 2	27.04.2026
Розділ 3	11.05.2026
Остаточне оформлення роботи	29.05.2026
Направлення роботи для перевірки на плагіат	09.06.2026
Попередній захист роботи	12.06.2026
Направлення роботи на рецензування	09.06.2026

7. Дата видачі завдання: 17 лютого 2026

Керівник _____ Гуменний Д.О.
(підпис) (прізвище та ініціали)

Здобувач _____ Корнійчук Л.М.
(підпис) (прізвище та ініціали)

РЕЗЮМЕ (SUMMARY) до кваліфікаційної випускової роботи здобувача	Корнійчук Леонід Миколайович Korniichuk Leonid Mukolaiovuch		
ЗВО	Київський національний університет будівництва і архітектури		
Тема (українською та англійською)	Розробка відмовостійкого embedded–контролера для роботизованих систем (C/C++ + RTOS) Development of a fault-tolerant embedded controller for robotic systems (C/C++ + RTOS)		
Освітній ступінь	Бакалавр		
Факультет	Автоматизації і інформаційних технологій		
Випускова кафедра	Кібербезпеки та комп'ютерної інженерії		
Спеціальність	123(F7) - Комп'ютерна інженерія		
Освітня програма	Комп`ютерні системи і мережі		
Керівник	Гуменний Дмитро		
Обсяг роботи:	Поснювальна записка, стор.	Розділів	Презентація, кількість слайдів
	126	3	15
Розділ 1	Аналіз предметної галузі та постановка задачі		
Розділ 2	Обґрунтування та розроблення рішення		
Розділ 3	Реалізація та тестування		
Висновки по роботі	Розроблено відмовостійкий вбудований контролер для роботизованих систем за архітектурою Master–Slave на базі мікроконтролера ESP32 із резервним каналом телеметрії LoRa SX1280. Реалізовано ешелоновану систему захисту від відмов: скінченний автомат станів Green–Yellow–Red, програмні тайм-аути, апаратний сторожовий таймер та механізм автономного відновлення tryRecovery без перезавантаження контролера. Працездатність системи підтверджено інструментальними вимірюваннями (Rohde & Schwarz), польовими тестами радіоканалу на дистанції понад 100 м та верифікацією у режимі імітації відмов. Автономність системи становить 37–40 годин від акумулятора 18650 (2S 1P).		
Ключові слова: Keywords:	відмовостійкість, вбудований контролер, ESP32, LoRa, SX1280, FreeRTOS, скінченний автомат станів, Master-Slave, телеметрія, роботизовані системи, CRC, сторожовий таймер, автономне відновлення fault tolerance, embedded controller, ESP32, LoRa, SX1280, FreeRTOS, finite state machine, Master-Slave, telemetry, robotic systems, CRC, watchdog timer, autonomous recovery		

АНОТАЦІЯ

Роботу присвячено розробці відмовостійкого вбудованого контролера для роботизованих систем на базі мікроконтролера ESP32 із резервним каналом передачі телеметрії через радіоінтерфейс LoRa. Система реалізована за архітектурою Master–Slave із двома фізично незалежними вузлами: майстер-вузол виконує керування сервоприводом та зчитування даних датчика температури і вологості AHT10 по шині I2C, слейв-вузол забезпечує незалежний канал діагностики через триколірну світлодіодну індикацію та веб-інтерфейс. Радіоканал між вузлами побудований на модулях Ebyte із трансивером SX1280 у діапазоні 2.4 ГГц. У роботі реалізовано ешелоновану систему захисту від відмов, що включає скінченний автомат станів із трирівневою градацією Green–Yellow–Red, програмні тайм-аути при зверненні до периферії, апаратний сторожовий таймер та механізм автономного відновлення, який виконує переініціалізацію несправної підсистеми без перезавантаження контролера.

Ключові слова: відмовостійкість, вбудований контролер, ESP32, LoRa, SX1280, FreeRTOS, скінченний автомат станів, Master-Slave, телеметрія, роботизовані системи, CRC, сторожовий таймер, автономне відновлення

ABSRACT

The thesis is dedicated to the development of a fault-tolerant embedded controller for robotic systems based on the ESP32 microcontroller with a redundant telemetry channel via the LoRa radio interface. The system is implemented following a Master–Slave architecture with two physically independent nodes: the master node controls a servo motor and reads temperature and humidity data from an AHT10 sensor over the I2C bus, while the slave node provides an independent diagnostic channel through three-color LED indication and a web interface. The radio channel between the nodes is built on Ebyte modules with the SX1280 transceiver operating in the 2.4 GHz band. The work implements a layered fault protection system comprising a finite state machine with three-level Green–Yellow–Red gradation, software timeouts on peripheral access, a hardware watchdog timer, and an autonomous recovery mechanism that performs reinitialization of the failed subsystem without restarting the controller.

Key words: fault tolerance, embedded controller, ESP32, LoRa, SX1280, FreeRTOS, finite state machine, Master-Slave, telemetry, robotic systems, CRC, watchdog timer, autonomous recovery

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ

АЦП – Аналого–цифровий перетворювач

ЦАП – Цифро–аналоговий перетворювач

ШІМ (PWM) – Pulse Width Modulation — широтно–імпульсна модуляція

ДСКВ – Децентралізована система керування версіями

ADC – Analog–to–Digital Converter — аналого–цифровий перетворювач

AES – Advanced Encryption Standard — стандарт симетричного шифрування

AP – Access Point — точка доступу Wi–Fi

API – Application Programming Interface — інтерфейс прикладного програмування

BLE – Bluetooth Low Energy — Bluetooth малого енергоспоживання

BW – Bandwidth — смуга пропускання

CAN – Controller Area Network — послідовна шина для мікроконтролерів

CCITT – Consultative Committee for International Telephony and Telegraphy — Міжнародний консультативний комітет з телефонії та телеграфії

CPU – Central Processing Unit — центральний процесор

CR – Code Rate — коефіцієнт завадостійкого кодування

CRC – Cyclic Redundancy Check — циклічний надлишковий код

CSS – Chirp Spread Spectrum — модуляція на основі чирп–сигналів

DC – Direct Current — постійний струм

DHCP – Dynamic Host Configuration Protocol — протокол динамічного призначення IP–адрес

DOM – Document Object Model — об'єктна модель документа

DSSS – Direct Sequence Spread Spectrum — розширення спектру прямою послідовністю

ECU – Electronic Control Unit — електронний блок керування

FHSS – Frequency Hopping Spread Spectrum — розширення спектру зі стрибками частоти

FSM – Finite State Machine — скінченний автомат станів

GCC – GNU Compiler Collection — набір компіляторів GNU

GF(2) – Galois Field — поле Галуа з двома елементами

GND – Ground — спільна земля (нульовий потенціал)

GPIO – General Purpose Input/Output — порт введення–виведення загального призначення

GPS – Global Positioning System — глобальна система позиціонування

HTML – HyperText Markup Language — мова розмітки гіпертексту

HTTP – HyperText Transfer Protocol — протокол передачі гіпертексту

I2C – Inter–Integrated Circuit — двопровідна послідовна шина даних

IDE – Integrated Development Environment — інтегроване середовище розробки

IEC – International Electrotechnical Commission — Міжнародна електротехнічна комісія

IEEE – Institute of Electrical and Electronics Engineers — Інститут інженерів електротехніки та електроніки

IPEX – Мікрохвильовий роз'єм малого форм–фактору (U.FL сумісний)

IRQ – Interrupt Request — запит на переривання

ISM – Industrial, Scientific and Medical — діапазон радіочастот вільного використання

ISO – International Organization for Standardization — Міжнародна організація зі стандартизації

JSON – JavaScript Object Notation — текстовий формат обміну даними

LED – Light Emitting Diode — світлодіод

LoRa – Long Range — технологія радіозв'язку на великі відстані

LPWAN – Low Power Wide Area Network — мережа з малим енергоспоживанням та великим радіусом дії

MCU – Microcontroller Unit — мікроконтролер

MTBF – Mean Time Between Failures — середній час між відмовами

PSRAM – Pseudo Static RAM — псевдостатична оперативна пам'ять

RAM – Random Access Memory — оперативна пам'ять

RF – Radio Frequency — радіочастота / радіочастотний тракт

ROM – Read Only Memory — постійна пам'ять

RSA – Rivest–Shamir–Adleman — алгоритм асиметричного шифрування

RSSI – Received Signal Strength Indicator — індикатор рівня прийнятого сигналу

RTC – Real–Time Clock — годинник реального часу

RTOS – Real–Time Operating System — операційна система реального часу

RTToF – Round–Trip Time of Flight — час подвійного пробігу сигналу

RX – Receive / Receiver — приймання / приймач

SCL – Serial Clock Line — лінія тактового сигналу шини I2C

SDA – Serial Data Line — лінія даних шини I2C

SF – Spreading Factor — коефіцієнт розширення спектру

SHA – Secure Hash Algorithm — алгоритм захищеного хешування

SIL – Safety Integrity Level — рівень цілісності безпеки

SMA – SubMiniature version A — коаксіальний радіочастотний роз'єм

SNR – Signal–to–Noise Ratio — відношення сигнал–шум

SPI – Serial Peripheral Interface — послідовний периферійний інтерфейс

SPIFFS – Serial Peripheral Interface Flash File System — файлова система для SPI Flash

SRAM – Static Random Access Memory — статична оперативна пам'ять

SSID – Service Set Identifier — ідентифікатор бездротової мережі Wi–Fi

TCP/IP – Transmission Control Protocol / Internet Protocol — стек мережевих протоколів

TX – Transmit / Transmitter — передавання / передавач

UART – Universal Asynchronous Receiver–Transmitter — універсальний асинхронний приймач–передавач

UDP – User Datagram Protocol — протокол датаграм користувача

USB – Universal Serial Bus — універсальна послідовна шина

WDT – Watchdog Timer — сторожовий таймер

Wi–Fi – Wireless Fidelity — стандарт бездротового зв'язку IEEE 802.11

XOR – eXclusive OR — операція виключного АБО

ВСТУП

Розвиток сучасної робототехніки характеризується стрімким зростанням складності та автономності керованих систем, що висуває якісно нові вимоги до надійності вбудованої електроніки. Якщо ще кілька десятиліть тому відмова контролера у промисловому роботі означала зупинку технологічного процесу, то сьогодні некеровані роботизовані платформи здатні завдавати матеріальної шкоди або створювати небезпечні ситуації для людей у своєму робочому просторі. Ця реальність принципово змінює підхід до проєктування керуючої електроніки: надійність перестає бути бажаною властивістю і стає обов'язковою архітектурною вимогою.

Незважаючи на наявність розвинених промислових стандартів функціональної безпеки, зокрема IEC 61508 та ISO 26262, їх практична реалізація залишається прерогативою дорогих спеціалізованих платформ і фактично недоступна для широкого класу роботизованих систем початкового та середнього рівня, що будуються на основі доступних мікроконтролерів загального призначення. Між промисловими рішеннями із сертифікованою відмовостійкістю та типовими аматорськими або навчальними роботизованими платформами існує значний розрив. Останні здебільшого або повністю ігнорують питання відмовостійкості, або вирішують його локально через окремі ізольовані механізми на кшталт сторожового таймера без системного підходу до діагностики та відновлення.

Окремою практичною проблемою є відсутність у більшості доступних рішень незалежного каналу сповіщення оператора про аварійний стан системи. Якщо відмова зачіпає саму керуючу платформу або основний канал зв'язку, інформація про збій може просто не дійти до оператора, що унеможливлює своєчасне втручання. Вирішення цієї проблеми потребує архітектурного розділення рівня керування та рівня моніторингу із забезпеченням їх взаємної незалежності.

Метою даної дипломної роботи є розробка відмовостійкого вбудованого контролера для роботизованих систем, у якому відмовостійкість реалізована як архітектурний принцип, а не набір ізольованих захисних заходів. Для досягнення

поставленої мети визначено такі задачі: аналіз предметної галузі та існуючих підходів до побудови відмовостійких вбудованих систем; розробка архітектури системи з апаратною та програмною ізоляцією рівня керування від рівня моніторингу; реалізація ешелонованої системи захисту від відмов що включає скінченний автомат станів, програмні тайм–аути, апаратний сторожовий таймер та механізм автономного відновлення; організація резервного каналу передачі телеметрії через радіоінтерфейс LoRa; розробка діагностичного веб–інтерфейсу для дистанційного моніторингу стану системи; практична реалізація та експериментальна верифікація розроблених механізмів відмовостійкості в умовах імітації реальних аварійних сценаріїв.

Об'єктом дослідження є процес забезпечення відмовостійкості вбудованих контролерів для роботизованих систем. Предметом дослідження є методи та архітектурні рішення для виявлення відмов, збереження функціональності та автономного відновлення у вбудованих системах реального часу на базі мікроконтролерів загального призначення.

Практична реалізація виконана на базі мікроконтролера ESP32 із використанням операційної системи реального часу FreeRTOS. Система побудована за архітектурою Master–Slave із двома фізично незалежними вузлами, з'єднаними радіоканалом на основі модулів Ebyte із трансивером SX1280 у діапазоні 2.4 ГГц із модуляцією LoRa. Майстер–вузол виконує керування сервоприводом та зчитування даних датчика температури і вологості, моделюючи типову роботизовану платформу з виконавчими механізмами та сенсорами. Слейв–вузол реалізує незалежний канал діагностики через триколірну індикацію та веб–інтерфейс.

Результати роботи підтверджені експериментально: серія випробувань у режимі імітації відмов, вимірювання параметрів радіоканалу апаратурою Rohde & Schwarz та польові тести зв'язку підтвердили коректність реалізованих механізмів і відповідність системи поставленим вимогам.

1 АНАЛІЗ ПРЕДМЕТНОЇ ГАЛУЗІ ТА ПОСТАНОВКА ЗАДАЧІ

Сучасні роботизовані системи висувають дедалі жорсткіші вимоги до надійності керуючої електроніки, оскільки відмова вбудованого контролера у відповідальний момент може призвести не лише до втрати функціональності, а й до фізичного пошкодження обладнання або виникнення небезпечної ситуації. Саме тому аналіз предметної галузі в даній роботі зосереджений не стільки на огляді існуючих рішень заради самого огляду, скільки на виявленні конкретних технічних протиріч і обмежень, що визначили архітектурні рішення розроблюваної системи.

1.1 Аналіз сучасного стану та тенденцій розвитку вбудованих комп'ютерних систем у робототехніці

Мікроконтролер у своїй основі є повноцінним комп’ютером, розміщеним на одному напівпровідниковому кристалі, що містить у собі процесорне ядро, оперативну та постійну пам'ять, а також набір периферійних інтерфейсів для взаємодії із зовнішнім світом. Формально, мікроконтролер (MCU) – це компактна інтегральна схема, призначена для керування електронними пристроями. Це не просто процесор; це цілий комп’ютер на одному чіпі. Його основна функція – отримувати дані від навколишнього середовища (за допомогою різних сенсорів), обробляти їх за запрограмованим алгоритмом і виконувати дії: вмикати двигуни, світлодіоди, приймати та відправляти інформацію та відображати її на екрані тощо. На відміну від домашнього ПК, який виконує загальні задачі, мікроконтролер заточений під виконання однієї конкретної задачі дуже ефективно та автономно. Всередині корпусу мікроконтролера міститься цілий комплект: Центральний процесор (CPU): Виконує інструкції програми. Пам’ять (RAM та Flash): Оперативна пам’ять для тимчасових даних і постійна пам’ять для зберігання коду або параметрів для автозавантаження при старті. Периферійні пристрої: це компоненти під’єднані до портів вводу/виводу (GPIO), таймери, аналого–цифрові перетворювачі (ADC), інтерфейси зв’язку як I2C, SPI, UART. З візуалізацією можливостей мікроконтролера можна ознайомитися на рисунку 1.1.

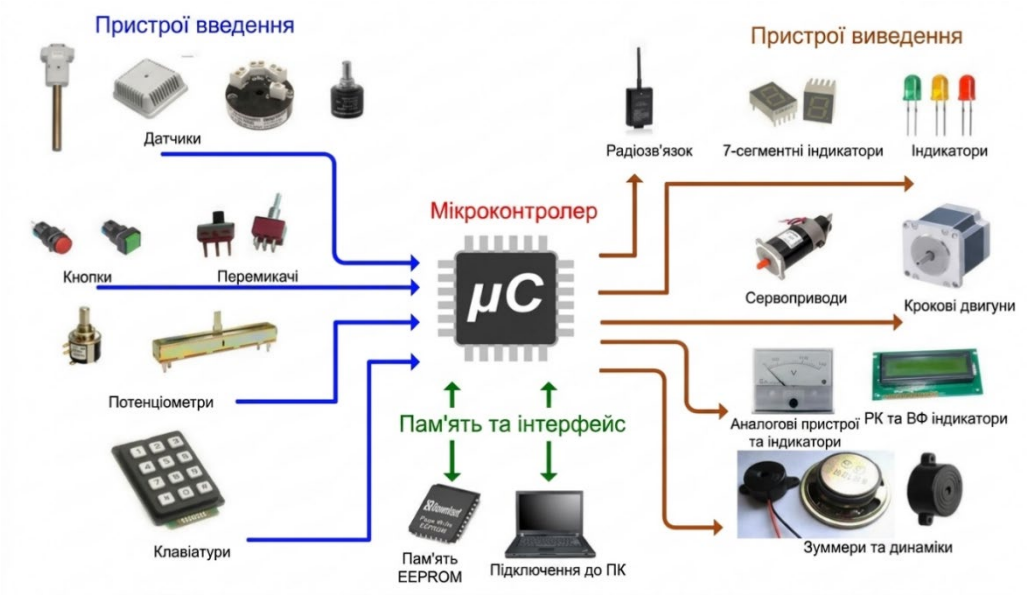

Рисунок 1.1 – Відображення можливих компонентів для підключення до мікроконтролера від Robocraze [22]

Саме наявність усіх цих компонентів на одному чіпі робить мікроконтролер незалежним, енергоощадним і дешевим рішенням для вбудованих систем. На відміну від персональних комп'ютерів, де процесор працює в парі з окремими модулями пам'яті та відеокартами, мікроконтролер спроектований для виконання конкретних, вузькоспеціалізованих задач в умовах обмежених енергетичних та обчислювальних ресурсів. В архітектурі сучасного мікроконтролера ключову роль відіграють порти введення–виведення загального призначення (GPIO), які дозволяють програмно маніпулювати електричними рівнями на виводах мікросхеми, перетворюючи бінарний код у фізичні дії, такі як увімкнення реле чи генерація складних сигналів керування. Розвиток мікроконтролерної техніки став каталізатором для сучасної робототехніки, оскільки саме поява потужних та дешевих 32–бітних обчислювачів дозволила перенести складні математичні розрахунки безпосередньо у самі автономні пристрої. У контексті роботизованих платформ мікроконтролер виконує роль центрального вузла, який одночасно обробляє потоки даних від різноманітних сенсорів, таких як ультразвукові далекоміри, акселерометри та цифрові термометри, і на основі цих даних формує команди для драйверів двигунів та сервоприводів. Використання мікроконтролерів у робототехніці вимагає особливого підходу до програмування, де критично

важливим стає поняття детермінізму – здатності системи гарантовано виконувати певний код за чітко визначений проміжок часу. Це особливо актуально для стабілізації польоту квадрокоптерів або балансування крокуючих роботів, де затримка в обчисленнях навіть на кілька мілісекунд може призвести до втрати рівноваги та фізичного руйнування конструкції. З розвитком IoT використання мікроконтролерів набуло надзвичайного поширення. Їхні застосування практично нескінченні:

– побутова електроніка: мультиварка, яка підтримує точну температуру, пральна машина з десятком програм, пульт дистанційного керування та навіть зубна щітка з таймером;

– автомобілі: Сучасний автомобіль може містити сотні мікроконтролерів. Вони керують двигуном (ECU), антиблокувальною гальмівною системою (ABS), подушками безпеки, клімат–контролем, навігаційною системою, тощо;

– промисловість: Керування конвеєрними лініями, роботами–маніпуляторами, моніторинг тиску та температури в технологічних процесах.

З набуваючих останнім часом популярності, через події в країні, тем є дрони. Вони теж базуються на STM мікропроцесорах обробляючи дані з сенсорів GPS, компаса, регулятора обертів та використовують зв’язкові вузли на основі ESP32 мікроконтролерів.

У сучасній ієрархії керування роботами мікроконтролери часто поділяють за їхньою обчислювальною потужністю та спеціалізацією. Найпростіші 8–бітні рішення, як–от архітектура AVR, що лежить в основі платформ Arduino, зазвичай використовуються для базових задач логічного керування, де не потрібна висока швидкість обробки даних. Проте для сучасних систем, які вимагають підтримки бездротових протоколів зв'язку та роботи в середовищі операційних систем реального часу (RTOS), стандартом стали контролери на базі архітектур ARM Cortex та Xtensa. Саме ці системи дозволяють реалізувати багаторівневу відмовостійкість, коли одне ядро процесора може бути повністю виділене під критично важливі задачі стабілізації та безпеки, тоді як інше ядро забезпечує

взаємодію з користувачем через веб–інтерфейс або хмарні сервіси. Такий розподіл ресурсів є фундаментальним для проектування надійних вбудованих систем, оскільки він мінімізує ризик того, що програмна помилка в мережевому стеку призведе до зупинки основного алгоритму керування роботом.

Вибір конкретного сімейства мікроконтролерів для роботизованого проекту завжди є компромісом між споживаною потужністю, кількістю апаратних інтерфейсів та складністю розробки. Наприклад, контролери серії STM32 цінуються за їхню неймовірну гнучкість у налаштуванні таймерів та периферії, що робить їх ідеальними для високоточних промислових маніпуляторів. Водночас платформи на базі ESP32 стали лідерами в галузі сервісної та мобільної робототехніки завдяки вбудованим засобам радіозв'язку, що дозволяє створювати розподілені мережі роботів, які обмінюються даними через Wi–Fi або LoRa без використання додаткових зовнішніх модулів. У результаті, сучасний мікроконтролер у робототехніці перетворився з простого логічного автомата на складну систему–на–кристалі (SoC), здатну не лише виконувати базові команди, а й проводити первинну обробку сигналів, фільтрацію шумів та навіть виконувати спрощені алгоритми штучного інтелекту для розпізнавання патернів у даних сенсорів.

Фундаментальна відмінність між роботизованою системою та звичайним споживчим гаджетом, наприклад смартфоном чи планшетом, полягає в характері взаємодії з фізичним світом. У той час як звичайний гаджет зорієнтований на споживання та обробку інформації для користувача, робот є виконавчим пристроєм, чиї дії мають прямі фізичні наслідки. Якщо мобільний додаток зависне на кілька секунд під час завантаження стрічки новин, користувач відчує лише легке роздратування. Однак, якщо контролер безпілотного літального апарата або промислового маніпулятора затримає обробку сигналу від сенсора на ті ж кілька секунд, це неминуче призведе до втрати стійкості, і як наслідок потенційного механічного пошкодження об'єкта або падіння.

Центральним поняттям, що розділяє ці два світи, є детермінізм і робота в режимі реального часу. Більшість звичайних гаджетів працюють під управлінням операційних систем загального призначення, де планувальник завдань намагається забезпечити плавність інтерфейсу та максимальну пропускну здатність, але не гарантує точний час виконання конкретної операції. Роботизована система вимагає операційної системи реального часу (RTOS), де кожен цикл обчислень має суворі часові рамки. В даному випадку правильний результат, отриманий із запізненням, вважається помилковим. Наприклад, якщо алгоритм стабілізації не отримає дані про нахил вчасно через те, що процесор був зайнятий обробкою веб–запиту, робот просто не зможе втримати рівновагу. Критичність збоїв у робототехніці перетворює питання надійності з програмної площини в площину безпеки життєдіяльності та збереження майна. У звичайних гаджетах архітектура розрахована на м’які відмови: перезавантаження програми зазвичай вирішує проблему без наслідків. Роботизовані комплекси проектуються за принципом ізоляції відмов. Це означає, що архітектура має передбачати сценарії, коли один вузол виходить з ладу, але система залишається керованою або переходить у безпечний стан. Саме тому впроваджується спеціальний Slave–контролер із діодною індикацією – він створює незалежний канал сповіщення, який працює навіть тоді, коли основний інтерфейс взаємодії з користувачем став недоступним.

Ще одна відмінність полягає в енергетичному та завадовому профілі середовища експлуатації. Гаджети зазвичай працюють у відносно стабільних умовах офісів чи житлових приміщень. Роботизовані системи часто функціонують у середовищі з високим рівнем електромагнітних завад від електродвигунів, значними вібраціями та коливаннями напруги живлення. Ці фактори можуть викликати випадкові помилки в пам'яті або збої на шинах передачі даних, таких як I2C, UART, CAN. Тому, на відміну від побутової електроніки, вбудоване ПЗ робота повинно містити надлишкові механізми перевірки цілісності даних, такі як контрольні суми та сторожові таймери, які постійно моніторять адекватність поведінки системи.

Таким чином, проектування роботизованої системи – це завжди пошук балансу між продуктивністю та передбачуваністю. У той час як ринок гаджетів женеться за гігагерцами та гігабайтами, вбудована інженерія фокусується на гарантованому часі відгуку та живучості. Робот – це не просто гаджет із моторами, це замкнена система автоматичного керування, де кожен програмний цикл є критичною ланкою в ланцюгу фізичної безпеки, а надійність зв'язку між Master та Slave вузлами визначає, чи залишиться машина під контролем у критичній ситуації.

Зі збільшенням функціональних можливостей сучасних вбудованих систем, програмне забезпечення перестає бути просто набором лінійних інструкцій і перетворюється на багатошарову екосистему, де складність зростає експоненційно. У проєкті, де одночасно працюють стеки протоколів LoRa, Wi–Fi, шина I2C та алгоритми керування сервоприводами, кількість можливих станів системи стає настільки великою, що їх неможливо повністю протестувати на етапі розробки. Кожна нова функція або бібліотека додає приховані залежності, які при певних комбінаціях зовнішніх чинників можуть призвести до непередбачуваної поведінки або повного зависання мікроконтролера.

Одним із головних чинників зростання ймовірності помилок є перехід від однопотокового виконання коду до багатозадачності в середовищі операційних систем реального часу (RTOS). Коли декілька завдань, так звані Tasks одночасно змагаються за спільні ресурси процесора або периферійні пристрої, виникає ризик появи специфічних помилок синхронізації, таких як стан гонитви, коли результат залежить від того в якому порядку і як швидко різноконтекстні задачі виконуються, або взаємне блокування. Наприклад, якщо завдання діагностики намагається зчитати дані з датчика I2C саме в той момент, коли завдання керування сервоприводами ініціює обмін даними, без належного використання семафорів система може зайти в тупик, що призведе до логічного збою всієї роботизованої платформи.

Також треба враховувати, що система постійно отримує непередбачувані дані з реального світу (шум сенсорів, механічні перешкоди), програмне забезпечення повинно обробляти тисячі сценаріїв.

1.2 Класифікація дестабілізуючих факторів та видів відмов

Для того, щоб забезпечити стабільну роботу роботизованої системи, необхідно враховувати весь спектр потенційних загроз, які можуть виникнути під час її експлуатації. Всі дестабілізуючі фактори можна класифікувати за їхнім походженням на кілька основних груп, що дозволяє розробити специфічні методи протидії для кожного виду збою:

– апаратні збої: Просадки живлення, перегрів, вплив вібрації на контакти;

– програмні збої: Deadlocks у RTOS, переповнення стека, витік пам'яті;

– зовнішні завади: Електромагнітний шум у промислових умовах або міській забудові, який дає завади стандартний зв’язок;

– комунікаційні завади: Сюди відносяться втрата пакетів даних у радіоефірі, електромагнітні наводки на цифрові шини зв'язку та конфлікти ідентифікаторів у мережі.

Почнемо з апаратних збоїв:

1.2.1 Апаратні збої

Апаратні збої є критичним фактором дестабілізації, оскільки вони виникають на фізичному рівні та часто не можуть бути виправлені лише програмними методами. У вбудованих системах, що працюють у реальних умовах, такі відмови стають причиною непередбачуваної поведінки мікроконтролера, навіть якщо логіка коду є бездоганною. Основними джерелами таких проблем у роботизованих комплексах виступають нестабільність живлення, температурний дріфт та механічний вплив на контактні групи;

– просадки живлення (Brown–out) є однією з найпідступніших проблем для мікроконтролерів серії ESP32. У моменти пікових струмів при використанні сервоприводів, напруга на шині живлення може короткочасно впасти нижче критичного порогу. Це не завжди призводить до повного перезавантаження

системи, але може викликати спотворення даних у регістрах процесора або зупинку окремих периферійних модулів. У таких випадках внутрішній детектор просадки напруги може ініціювати апаратне переривання. Для стабілізації живлення в такому випадку краще встановити конденсатор;

– перегрів компонентів безпосередньо впливає на надійність напівпровідникових переходів та стабільність тактової частоти. У вбудованих системах, де ESP32 виконує інтенсивні обчислення в RTOS паралельно з роботою веб–сервера, виділення тепла може стати критичним, особливо в закритих корпусах без активного охолодження. Підвищення температури призводить до зростання витоків струму та може спричинити деградацію точності АЦП, що зробить дані з датчиків недостовірними;

– вплив вібрації на контакти є специфічним викликом для мобільної робототехніки. Постійні механічні коливання, викликані роботою двигунів або рухом платформи, призводять до виникнення мікротріщин у паяних з'єднаннях та окислення роз'ємів. Найбільш вразливою в цьому контексті є шина I2C, де навіть короткочасна втрата контакту на лініях SDA або SCL може призвести до зупинки роботи всієї шини, оскільки Master–пристрій буде нескінченно очікувати на підтвердження (ACK) від датчика, якщо зв’язок не буде відновлено;

1.2.2 Програмні помилки

Програмні помилки є найбільш підступними дестабілізуючими факторами, оскільки вони часто мають випадковий характер і проявляються лише за певних умов навантаження на систему. У відмовостійких контролерах, що працюють під управлінням операційних систем реального часу (RTOS), такі збої можуть миттєво паралізувати роботу всієї роботизованої платформи, перетворюючи її на некерований об'єкт.

Однією з найскладніших проблем багатозадачних систем є виникнення ситуацій взаємного блокування, відомих як Deadlocks. Це явище виникає, коли два або більше завдань утримують певні ресурси (наприклад, семафори для доступу до шини I2C або модуля LoRa) і водночас намагаються отримати доступ до ресурсів,

які вже зайняті іншими завданнями. У результаті процеси потрапляють у замкнене коло очікування, жоден із них не може продовжити виконання, а система фактично зупиняється, попри те, що процесор продовжує отримувати живлення. У проекті на базі ESP32 такий сценарій можливий, якщо завдання обробки веб–інтерфейсу та завдання передачі аварійного сигналу не погодять пріоритети доступу до спільних апаратних інтерфейсів.

Кожне завдання в RTOS має виділену йому обмежену ділянку оперативної пам'яті – стек, де зберігаються локальні змінні, адреси повернення функцій та стани регістрів. Переповнення стека відбувається, коли обсяг даних, що записуються в цю ділянку, перевищує її розмір. Наслідки переповнення стека є катастрофічними: дані виходять за межі відведеної пам'яті, затираючи критично важливі змінні сусідніх завдань або пошкоджуючи службові структури ядра RTOS. Це призводить до непередбачуваної поведінки коду, коли система може продовжувати працювати, але виконувати абсолютно безглузді операції, що є неприпустимим для відмовостійкого контролера.

Витік пам'яті є накопичувальною помилкою, яка зазвичай виникає при динамічному розподілі ресурсів. Якщо програма виділяє пам'ять для певного об'єкта, наприклад для формування тексту веб–сторінки на ESP32, і не звільняє її після використання, обсяг доступної оперативної пам'яті поступово зменшується. У короткостроковій перспективі це може не впливати на роботу робота, проте під час тривалої експлуатації вільна пам'ять вичерпається повністю. У цей момент черговий запит на виділення ресурсів для передачі даних через LoRa або обробку показань датчика температури закінчиться невдачею, що призведе до аварійної зупинки всієї системи. Для вбудованих систем, що претендують на високу надійність, витік пам'яті вважається критичним дефектом проектування, оскільки він робить час безвідмовної роботи системи обмеженим і непередбачуваним.

1.2.3 Зовнішні завади

Зовнішні завади в радіоефірі є одним із найскладніших факторів дестабілізації, оскільки вони не залежать від якості коду чи надійності заліза, а визначаються

середовищем, у якому працює роботизована система. У сучасних умовах міської забудови або промислових зон діапазон 2.4 ГГц, у якому працює більшість гаджетів, є дуже перевантаженим, що створює ефект радіошуму, здатного повністю заблокувати передачу критично важливих пакетів даних. У міській забудові головним джерелом завад є висока щільність мереж Wi–Fi та пристроїв Bluetooth.

1.2.4 Комунікаційні завади

Оскільки більшість побутових роутерів працюють на тих самих частотах, що й стандартні модулі керування роботами, виникає явище інтерференції. Коли десятки пристроїв одночасно передають дані, рівень фонового шуму піднімається достатньо високо, що необхідний корисний сигнал від контролера просто втрачається в ньому. Це призводить до різкого зниження відношення сигналу до шуму (SNR), через що приймач не може розпізнати початок пакета або отримує дані з великою кількістю бітових помилок, які неможливо виправити навіть за допомогою контрольних сум. У промислових умовах ситуація ускладнюється наявністю потужного електрообладнання, такого як електродвигуни, частотні перетворювачі, зварювальні апарати та високовольтні лінії. Ці пристрої генерують імпульсні завади широкого спектра, які створюють короткочасні, але дуже потужні сплески електромагнітного випромінювання. Такі сплески можуть не тільки заглушити радіоефір, а й навести паразитні струми безпосередньо на антени або доріжки друкованої плати пристрою. Це викликає збої в роботі трансивера на фізичному рівні, що вимагає від системи здатності до швидкої повторної ініціалізації радіомодуля без перезавантаження всього мікроконтролера.

1.3 Характеристика та порівняльний аналіз існуючих протоколів бездротового зв’язку

Для вибору оптимального каналу передачі даних необхідно оцінити існуючі стандарти зв’язку за критеріями дальності, енергоспоживання та стійкості до перешкод.

Короткодистанційні протоколи зв'язку є базовим рівнем для більшості сучасних вбудованих систем, проте в контексті відмовостійкої робототехніки вони

мають як суттєві переваги, так і критичні недоліки, що вимагають детального розгляду.

Wi–Fi (IEEE 802.11) – вважається найбільш універсальним рішенням завдяки своїй здатності передавати величезні масиви даних на високих швидкостях. У робототехніці це дозволяє транслювати відеопотік з камери робота в реальному часі, що і використовується в цифровому відео зв’язку для дронів, але там компенсують проблему дальності направленими антенами та багатоватними підсилювачами, або підтримувати складний веб–інтерфейс керування безпосередньо на мікроконтролері ESP32. Однак рівень енергоспоживання доволі високий, що є критичним для автономних платформ з живленням від акумуляторів, але ж головною проблемою Wi–Fi є його чутливість до завантаженості ефіру. Оскільки Wi–Fi використовує механізм прослуховування каналу перед передачею, у середовищі з великою кількістю інших мереж виникають значні та непередбачувані затримки, що робить неможливим гарантований час відгуку, необхідний для систем реального часу, або необхідно додавати підсилювачі сигналу які знову ж зменшують час автономності і при достатньо сильному сигналі буде погіршувати зв’язок всіх інших пристроїв, забиваючи ефір. Крім того, складна структура стека протоколів Wi–Fi робить його більш вразливим до програмних збоїв у порівнянні з простішими методами радіозв'язку.

Bluetooth – займає нішу енергоефективного зв'язку на невеликих відстанях. Його головна перевага – мінімальне споживання струму та здатність швидко встановлювати з'єднання з мобільними пристроями. У робототехніці BLE часто використовується для конфігурування параметрів або як сервісний інтерфейс. Проте для керування відмовостійким контролером Bluetooth має серйозне обмеження – малу дальність дії, яка зазвичай не перевищує 10–20 метрів у реальних умовах з перешкодами. Також Bluetooth працює за принципом стрибкоподібної перебудови частоти FHSS, або ж ППРЧ, що допомагає боротися із завадами, але не рятує від загального високого рівня шуму в діапазоні 2.4 ГГц, де сигнал може просто загубитися через низьку потужність передавача.

Промислові протоколи зв’язку та мережі класу LPWAN розроблялися спеціально для вирішення задач, з якими не справляються побутові інтерфейси: робота на великих відстанях, висока енергоефективність та здатність об’єднувати сотні пристроїв у єдину інфраструктуру.

ZigBee (стандарт IEEE 802.15.4) є одним із найбільш розповсюджених рішень для автоматизації промислових об’єктів та систем IoT. Його головна особливість – підтримка комірчастої топології Mesh–мережі, де кожен вузол може виступати ретранслятором для іншого. Це дозволяє покривати великі площі зі складним плануванням, оминаючи залізобетонні стіни та металеві конструкції. Для робототехніки ZigBee зручний своєю низькою затримкою при передачі коротких команд. Однак цей протокол працює переважно в тому ж зашумленому діапазоні 2.4 ГГц, що й Wi–Fi, і використовує модуляцію DSSS, яка хоч і є завадостійкою, але але потужних промислових наводок вистачає цю завадостійкість нівелюють.

LoRaWAN (діапазони 868/915 МГц) є еталоном дальності в світі IoT. Завдяки роботі на нижчих частотах та використанню фірмової модуляції LoRa, цей стандарт дозволяє передавати дані на відстані до 15–20 км на відкритій місцевості. В умовах промислового підприємства це забезпечує стабільний зв’язок крізь будь–які перешкоди. Проте LoRaWAN має критичний недолік для задач реального часу – дуже низьку пропускну здатність та відносно великі затримки. Класична мережа LoRaWAN працює за принципом UDP, відправник надсилає не чекаючи відповіді потік з даних, а цикл передачі одного пакета може тривати від кількох сотень мілісекунд до секунд в залежності від кількості чирпів для передачі сигналу та ширини каналу.

У контексті локальних роботизованих систем технологія LoRa 2.4 GHz на базі чіпа SX1280 виступає хорошим вибором, оскільки вона ліквідує розрив між надійністю промислових рішень та швидкістю побутових інтерфейсів. На відміну від класичної LoRaWAN, яка працює на низьких частотах і призначена для повільної передачі даних на кілометри, версія 2.4 ГГц оптимізована для динамічного керування та високої частоти оновлення даних. LoRa компенсує

мінуси LoRaWAN по частоті надсилання, затримкам та тільки покращеною завадостійкістю, оминаючи проблеми з завантаженністю радіоефіру. В таблиці 1.1 було наведено порівняльні характеристики.

Таблиця 1.1 – Порівняльна характеристика технологій

Параметр	**Wi–Fi (802.11n)**	**Bluetooth (BLE)**	**LoRaWAN (868 MHz)**	**LoRa (2.4 GHz/SX1280)**
Дальність (пряма видимість)	Середня (до 50–100 м)	Низька (до 10–30 м)	Дуже висока (до 15 км)	**Висока (до 2–3 км)**
Пропускна здатність	Дуже висока (до 150 Мбіт/с)	Низька (до 2 Мбіт/с)	Дуже низька (до 50 кбіт/с)	**Середня (до 1.3 Мбіт/с)**
Завадостійкість	Низька (чутливий до шуму 2.4GHz)	Середня (використовує FHSS)	Висока (модуляція CSS)	**Дуже висока (CSS + висока швидкість)**
Енергоспоживання	Дуже високе (200–400 мА)	Дуже низьке (5–15 мА)	Низьке (10–40 мА)	**Низьке (30–50 мА)**
Затримка	Варіативна (залежить від ефіру)	Середня	Дуже висока (секунди)	**Низька (мілісекунди)**
Складність стека ПЗ	Висока (потребує ОС/RTOS)	Середня	Висока (потребує Gateway)	**Низька (Point–to–Point)**

1.4 Опис об'єкта дослідження: відмовостійкий контролер роботизованої платформи

Типова структура сучасного робота являє собою складну ієрархічну систему, де кожен компонент виконує роль певної функції системи, а їхня злагоджена взаємодія забезпечує автономність та адекватну реакцію на зовнішні події. У центрі цієї архітектури знаходиться мікроконтролер, такий як ESP32, який виконує роль

центрального вузла обробки інформації. Завдяки наявності двох обчислювальних ядер, такий процесор здатний паралельно вирішувати завдання високого рівня, такі як: підтримку мережевого стека та взаємодію з користувачем, і критично важливі завдання реального часу, пов’язані з безпосереднім керуванням фізичними процесами. Висока тактова частота та значний обсяг оперативної пам'яті дозволяють йому не просто виконувати лінійний код, а працювати в середовищі операційної системи реального часу, що є фундаментом для створення відмовостійких систем.

Інформаційний вхід системи забезпечується через мережу сенсорів. Найчастіше для взаємодії з цифровими датчиками використовується шини I2C, UART, CAN, SPI які дозволяють підключати велику кількість різноманітної периферії – акселерометрів, гіроскопів, термометрів чи далекомірів – всього за допомогою двох сигнальних ліній. Це значно спрощує апаратну частину, проте накладає особливі вимоги до програмної стабільності, оскільки збій на цих шинах може вивести робота з рівноваги не залишивши йому даних на які можна спертися. Отримані дані з датчиків проходять первинну обробку, фільтрацію шумів та математичне перетворення всередині мікроконтролера, можливо фільтрацію, після чого алгоритм керування приймає рішення про необхідні фізичні дії.

Виконавчий рівень структури робота представлений сервоприводами, моторами та іншими актуаторами, які перетворюють електричні сигнали керування в механічний рух. Керування сервоприводами та моторами зазвичай відбувається за допомогою широтно–імпульсної модуляції, де тривалість імпульсу визначає точний кут повороту вала, або швидкість обертання визначений ТТХ моторів та регулятором обертів відносно певного PWM. Це завершує цикл зворотного зв’язку: процесор зчитує стан середовища через сенсори з шин, обчислює необхідну корекцію та відправляє сигнал на виконавчі механізми. Важливою особливістю професійної архітектури є те, що силова частина живлення сервоприводів завжди відокремлена від живлення мікроконтролера, щоб уникнути електричних завад та просадок напруги, які розглядались як один із головних дестабілізуючих факторів.

Якщо в стандартному роботі вихід з ладу центрального процесора призводить до повної зупинки або неконтрольованої поведінки, то у відмовостійкій системі структура передбачає наявність моніторингової системи. Вона постійно отримує данні стосовно станів головного контролера через спеціальні сторожові сигнали та має власну індикацію. Таким чином, навіть якщо основний інтелект робота буде паралізований програмною помилкою чи перегрівом, апаратна структура дозволить ідентифікувати збій та безпечно завершити роботу, що і відрізняє професійний роботизований комплекс від простої електронної іграшки.

Визначення критичних точок у робототехніці – це аналіз найгірших сценаріїв, де збій однієї ланки призводить до незворотних наслідків. Якщо зв'язок із роботом зникає саме під час активного руху, система опиняється в стані динамічного хаосу. Найбільша небезпека полягає в тому, що без спеціальних запобіжних механізмів виконавчі органи робота зберігають останню отриману команду. Це означає, що якщо робот їхав вперед на повній швидкості або маніпулятор рухався в бік перешкоди, вони продовжуватимуть цей рух до моменту фізичного зіткнення або повного розряду акумулятора, оскільки головний контролер не отримує сигналу на зупинку.

У цей момент виникає критичний конфлікт між механічною інерцією та програмним вакуумом. У звичайних системах без відмовостійкості мікроконтролер може просто продовжувати нескінченно чекати на наступний пакет даних, поки сервоприводи виходять з ладу, намагаючись подолати опір перешкоди. Втрата зв'язку під час руху також унеможливлює отримання телеметрії: оператор не знає, де знаходиться робот, у якому стані його батарея та чи не стався перегрів драйверів двигунів. Це створює керування, яке в промислових умовах може призвести до травмування персоналу або пошкодження дорогого обладнання.

Для нейтралізації цієї критичної точки в архітектуру впроваджується концепція Fail–Safe або безпечного стану. Програмне забезпечення відмовостійкого контролера повинно містити сторожовий таймер зв'язку, який постійно відраховує час з моменту отримання останнього коректного пакета. Якщо інтервал очікування

перевищує встановлений поріг, система повинна автоматично ініціювати протокол екстреної зупинки: зняти живлення з двигунів, заблокувати сервоприводи в нейтральному положенні та активувати світлову або звукову сигналізацію про втрату контролю.

У той час як головний процесор може бути зайнятий спробами відновити складне Wi–Fi з'єднання, незалежний Slave–контролер через виділений канал продовжує моніторити наявність сигналу. Якщо він фіксує радіотишу, він може примусово перехопити керування або подати сигнал на апаратне відключення виконавчих механізмів. Таким чином, критична точка втрати зв'язку перетворюється з імовірної катастрофи на штатну аварійну ситуацію, з якої система виходить автоматично і безпечно.

До прикладу в прошивках на дрони таких як Betaflight та Ardupilot концепція FailSafe як раз присутня та заготовлена під певні сценарії: втрата зв’язку або критичний рівень батареї. Обирається алгоритм який буде виконуватися при тому чи іншому сценарію: вимкнення моторів і падіння борту, алгоритм повернення додому на основі даних з сенсорів(GPS, магнітометр, барометр, акселерометр), м’яка посадка з плавним зниженням висоти.[18][19]

1.5 Формулювання задачі проєктування

Проведений аналіз дестабілізуючих факторів та порівняння існуючих протоколів зв’язку підтверджують, що стандартні архітектури на базі Wi–Fi або Bluetooth не здатні забезпечити необхідний рівень живучості роботизованої системи в умовах критичних програмних збоїв або інтенсивних зовнішніх завад. Виявлено, що втрата зв’язку під час активного руху робота створює критичні ризики для обладнання та оточення через інерційність виконавчих механізмів та відсутність автоматизованих протоколів безпеки в типових контролерах. Виходячи з цього, метою даної роботи є розробка відмовостійкого контролера, архітектура якого передбачає розділення функціональних обов’язків між основним та діагностичним вузлами.

Вибір мікроконтролера ESP32 як центрального вузла відмовостійкої системи зумовлений його унікальним балансом між обчислювальною потужністю, енергоефективністю та інтегрованими комунікаційними можливостями. На відміну від класичних 8–бітних рішень, ця платформа забезпечує необхідний технологічний стек для реалізації сучасних роботизованих комплексів, залишаючись при цьому одним із найдоступніших рішень на ринку. Енергонезалежність та пам'ять, підтримка FreeRTOS, двох ядерна архітектура та доступність на ринку цих чіпів, або наборів підготовлених для прототипування певного роду проектів.

Ключовим технічним рішенням є впровадження резервного каналу передачі даних на базі технології LoRa SX1280 2.4 GHz. Це дозволить створити незалежну магістраль для моніторингу стану системи та примусового переведення робота в безпечний режим Fail–Safe, навіть у разі повної відмови основної операційної системи або блокування стандартних каналів зв’язку. Такий підхід гарантує керованість об’єкта в позаштатних ситуаціях, що є необхідною умовою для експлуатації сучасних роботизованих комплексів у складних умовах. У відмовостійких системах час реакції – це інтервал між виникненням критичної події та переходом системи у безпечний стан. Використання LoRa 2.4 GHz дозволяє мінімізувати цей показник завдяки майже відсутній мережевій затримці, бо пакет передається на фізичному рівні, замість того щоб проходити через весь мережевий стек TCP/IP, та високій частоті оновлення показуючи результат на швидкості рівня декількох мілісекунд. Також слід згадати, що LoRa цілком і повністю виправдовує свою абривіатуру Long Range, забезпечуючи зв’язок на відстань до двох кілометрів без підсилення сигналу, при умовах прямої видимості.

1.6 Стандарти надійності

Під час розробки відмовостійких ембеддед–систем орієнтація на міжнародні стандарти є невід'ємною частиною надійності, які визначають підходи до проєктування, оцінки та підтвердження здатності системи виконувати свої функції в заданих умовах протягом певного часу. Надійність у цьому контексті

розглядається як комплексна властивість, що включає безвідмовність, довговічність, ремонтопридатність і збережуваність, і саме стандарти задають формалізовані критерії для її оцінювання.

Одним із базових стандартів у цій сфері є IEC 61508, який регламентує функціональну безпеку електричних, електронних та програмованих електронних систем. У цьому стандарті вводиться поняття рівнів повноти безпеки SIL, які визначають ймовірність відмови системи та допустимий рівень ризику. Для ембеддед–контролерів це означає необхідність застосування таких підходів, як резервування, діагностика, самоконтроль і контроль помилок, що безпосередньо пов’язано з реалізацією запасних каналів зв’язку, зокрема через LoRa. Іншим важливим документом є ISO 26262, який є адаптацією IEC 61508 для автомобільної галузі. Хоча він орієнтований на транспортні системи, його підходи широко застосовуються і в інших сферах, де важлива висока надійність. У цьому стандарті визначено рівні ASIL, а також описані методи аналізу небезпек, оцінки ризиків і проєктування архітектур із підвищеною відмовостійкістю. Для оцінювання власне надійності електронних компонентів і систем широко використовується стандарт MIL–HDBK–217, який містить моделі прогнозування інтенсивності відмов failure rate для різних типів компонентів. Він дозволяє на етапі проєктування оцінити середній час напрацювання на відмову (MTBF) і порівняти різні варіанти реалізації системи. Хоча цей стандарт має військове походження, він досі використовується в інженерній практиці як базовий орієнтир. Також важливим є стандарт IEC 60068, який визначає методи випробувань на вплив зовнішніх факторів, таких як температура, вологість, вібрації та механічні навантаження. Для ембеддед–пристроїв це критично, оскільки їх робота часто відбувається в нестабільних або агресивних умовах. Проведення таких випробувань дозволяє підтвердити, що система зберігає працездатність у заданому діапазоні впливів. У сфері програмного забезпечення стандарт ISO/IEC 25010 описує модель якості програмних систем, включаючи такі характеристики, як надійність, відмовостійкість, відновлюваність і здатність до обробки помилок. Для систем на базі мікроконтролерів це означає

необхідність реалізації механізмів обробки виключень, watchdog–таймерів, журналювання подій та відновлення після збоїв. Крім того, варто згадати стандарт IEC 61000, який регламентує електромагнітну сумісність. У контексті використання бездротових технологій, таких як LoRa, це особливо актуально, оскільки система повинна бути стійкою до електромагнітних завад і водночас не створювати надмірних перешкод для інших пристроїв.

Таким чином, застосування міжнародних стандартів надійності дозволяє сформувати системний підхід до проєктування відмовостійкого контролера. Вони забезпечують не лише формальні критерії оцінки, але й практичні рекомендації щодо архітектури, вибору компонентів, методів тестування та забезпечення стабільної роботи системи в реальних умовах експлуатації.

Висновок до розділу 1

За результатами аналізу предметної галузі встановлено, що сучасні роботизовані системи висувають до керуючої електроніки вимоги що суттєво виходять за межі простої функціональності та обчислювальної продуктивності. Ключовою проблемою є відсутність у більшості доступних рішень на базі мікроконтролерів загального призначення систематичного підходу до відмовостійкості: механізми захисту або відсутні взагалі, або реалізовані як ізольовані заходи без урахування взаємозалежності підсистем та необхідності незалежного каналу сповіщення оператора.

Аналіз існуючих протоколів бездротового зв'язку показав, що технологія LoRa у діапазоні 2.4 ГГц є обґрунтованим вибором для резервного телеметричного каналу завдяки поєднанню високої завадостійкості, прийнятної дальності та низького енергоспоживання. Огляд апаратної бази підтвердив, що мікроконтролер ESP32 із двоядерною архітектурою та вбудованою підтримкою FreeRTOS надає достатній технологічний стек для реалізації відмовостійкої системи на доступній елементній базі.

На підставі проведеного аналізу сформульовано задачу розробки: створити відмовостійкий вбудований контролер за архітектурою Master–Slave, у якому

апаратна та програмна ізоляція рівня керування від рівня моніторингу, ешелонована система захисту від відмов та механізм автономного відновлення є невід'ємними архітектурними властивостями, а не додатковими надбудовами.

2 ОБҐРУНТУВАННЯ ТА РОЗРОБЛЕННЯ РІШЕННЯ

2.1 Обґрунтування вибору апаратної платформи та компонентної бази

2.1.1 Мікроконтролер ESP32

Мікроконтролер ESP32 є одним із найбільш популярних рішень для побудови сучасних ембеддед–систем завдяки своїй двоядерній архітектурі, що базується на процесорних ядрах Tensilica Xtensa LX6. Така архітектура передбачає наявність двох незалежних обчислювальних ядер, які можуть працювати паралельно, що відкриває значні можливості для підвищення продуктивності, ефективності та, що особливо важливо в контексті цієї роботи, відмовостійкості системи.

Ключовою перевагою двоядерної архітектури є можливість розподілу функціональних задач між ядрами таким чином, щоб критично важливі процеси не заважали один одному. У випадку розробки відмовостійкого контролера з резервним каналом телеметрії через LoRa доцільно виділити одне ядро для обробки задач зв’язку, включаючи роботу з бездротовими інтерфейсами, стеком протоколів, обробкою переривань від радіомодуля та підтримкою з’єднання. Друге ядро при цьому може виконувати основні алгоритмічні задачі системи, такі як обробка даних сенсорів, прийняття рішень, фільтрація сигналів або виконання керуючої логіки. Такий підхід дозволяє ізолювати комунікаційні процеси від обчислювальних. Це означає, що навіть у випадку перевантаження або часткового збою алгоритмічної частини системи, підсистема зв’язку продовжує працювати стабільно. У контексті резервного каналу телеметрії через LoRa це має критичне значення, оскільки забезпечує можливість передачі діагностичної інформації навіть при деградації основної функціональності контролера. Іншими словами, система не втрачає здатність сповістити про проблему, що є одним із базових принципів відмовостійких систем.

Крім того, розподіл задач між ядрами знижує ризик виникнення блокувань та зменшує вплив довготривалих операцій на загальну стабільність системи. Наприклад, якщо обробка даних на Core 1 потребує значного часу або потрапляє у некоректний стан, це не блокує обробку переривань і передачу даних на Core 0. У

класичних однопроцесорних системах подібна ситуація могла б призвести до повної втрати керованості або зависання пристрою. Ще одним важливим аспектом є інтеграція операційної системи реального часу FreeRTOS, яка використовується в ESP32. Вона дозволяє гнучко закріплювати задачі за конкретними ядрами, встановлювати пріоритети та контролювати розподіл ресурсів. Завдяки цьому можна гарантувати, що задачі зв’язку матимуть вищий пріоритет і виконуватимуться навіть при пікових навантаженнях на систему. Це безпосередньо впливає на надійність передачі телеметрії, особливо у критичних умовах експлуатації.

З точки зору відмовостійкості, двоядерна архітектура також дозволяє реалізувати механізми взаємного контролю між ядрами. Наприклад, одне ядро може періодично перевіряти стан іншого, використовуючи watchdog–таймери або міжпроцесорну взаємодію. У разі виявлення аномалії можлива ініціація перезапуску окремої задачі або навіть усього ядра без повного перезавантаження системи. Це значно підвищує загальну надійність у порівнянні з однопроцесорними рішеннями.

Таким чином, використання двоядерної архітектури в ESP32 дозволяє не лише підвищити продуктивність системи, але й створити ефективну основу для побудови відмовостійких рішень. Розділення задач між ядрами, ізоляція критичних процесів зв’язку, можливість гнучкого управління задачами через RTOS та реалізація механізмів самоконтролю формують комплексний підхід до забезпечення безперервної роботи контролера навіть у випадку часткових відмов. Це робить ESP32 оптимальним вибором для систем із підвищеними вимогами до надійності, зокрема тих, що використовують резервні канали телеметрії, такі як LoRa.

Мікроконтролер ESP32 виробництва Espressif Systems побудований на базі двох ядер архітектури Xtensa LX6 із тактовою частотою до 240 МГц і є центральним обчислювальним елементом обох вузлів розроблюваної системи. Пам'ять організована у вигляді трьох банків: ROM обсягом 448 кілобайт із вбудованим завантажувачем, основна SRAM обсягом 520 кілобайт для розміщення задач

FreeRTOS та буферів периферії, а також RTC SRAM обсягом 16 кілобайт що зберігає свій вміст у режимах глибокого сну. За необхідності мікроконтролер підтримує підключення зовнішньої флеш–пам'яті та PSRAM.

Бездротова підсистема включає Wi–Fi стандарту 802.11 b/g/n із пропускною здатністю до 150 мегабіт за секунду та Bluetooth 4.2 з підтримкою BLE, причому обидва тракти використовують вбудований підсилювач потужності, малошумний підсилювач та балун без необхідності у зовнішніх узгоджувальних елементах. Периферійна підсистема охоплює до 34 GPIO з підтримкою SPI, I2C, UART, ШІМ, 12–бітного АЦП на 18 каналів та двоканального ЦАП, що повністю перекриває потреби проєкту в частині керування сервоприводом, зчитування датчика температури та взаємодії з LoRa–модулем. Споживання у режимі глибокого сну становить близько 10 мікроампер, що є важливим для автономних конфігурацій системи. Апаратні прискорювачі AES, SHA та RSA разом із механізмами Secure Boot і Flash Encryption забезпечують криптографічний захист прошивки. В таблиці 2.1 можна ознайомитися з характеристиками та з виглядом на рисунку 2.1.

Таблиця 2.1 – Технічні характеристики[2][23]

Характеристика	**DOIT ESP32 DEVKIT v1**
Призначення	Мікроконтролер + Wi–Fi/BT
Архітектура	Xtensa LX6, 2 ядра, 240 МГц
Діапазон частот	2.4 ГГц (Wi–Fi/BT)
ROM / Flash	448 КБ ROM
SRAM	520 КБ + 16 КБ RTC
Модуляція	Wi–Fi 802.11 b/g/n, BT 4.2
Чутливість приймача	–94 dBm (BLE)
Живлення	Vin – 5 В, 3.3 В, deep sleep ~10 мкА
Периферія	SPI, I2C, UART, PWM, ADC, DAC
GPIO	до 34
Додаткові функції	Secure Boot, AES/SHA/RSA, Ethernet MAC
Робоча температура	–40°C … +125°C
Орієнтація	Вбудовані системи, IoT

Рисунок 2.1 – Мікроконтроллер ESP32

2.1.2 Трансивер LoRa SX1280

Радіо модуль SX1280 є представником сімейства LoRa–трансиверів, що працюють у діапазоні 2.4 ГГц, на відміну від більш поширених рішень у субгігагерцових діапазонах (433/868/915 МГц). Вибір саме частоти 2.4 ГГц зумовлений рядом інженерних компромісів, які особливо актуальні для сучасних ембеддед–систем з підвищеними вимогами до глобальної сумісності та пропускної здатності. Діапазон 2.4 ГГц є ISM–діапазоном, що дозволений для використання практично в усьому світі без необхідності ліцензування, що спрощує розгортання пристроїв у різних країнах. Крім того, на цій частоті доступна більша ширина смуги, що дозволяє досягати вищих швидкостей передачі даних порівняно з класичними LoRa–рішеннями. Також важливою перевагою є менші розміри антен, що прямо пропорційні довжині хвилі і є критичними для компактних пристроїв. Разом з тим, фізичні властивості радіохвиль у діапазоні 2.4 ГГц передбачають більші втрати при поширенні та гіршу проникність через перешкоди порівняно з нижчими частотами. Саме тут ключову роль відіграє використання модуляції Chirp Spread Spectrum (CSS), яка лежить в основі технології LoRa. CSS базується на використанні так званих “чирпів” – сигналів, частота яких змінюється з часом у

межах заданої смуги. Замість передачі бітів як фіксованих частот або фаз, кожен символ кодується зміщенням початкової частоти чирпу. Іншими словами, інформація передається не просто сигналом, а його часово–частотною структурою. Фізично це означає, що сигнал розприділяється по широкій смузі частот, що робить його значно стійкішим до вузькосмугових завад та шумів. Якщо частина спектра зашумлена або втрачена, приймач все одно може відновити сигнал, оскільки інформація розподілена по всій ширині каналу. Додатково використовується кореляційна обробка сигналу, що дозволяє отримати корисний сигнал навіть тоді, коли його рівень знаходиться нижче рівня шуму. Це одна з ключових особливостей LoRa, яка забезпечує надвисоку чутливість приймача. Як виглядає модуль можна побачити на рисунку 2.2.

Рисунок 2.2 – Трансивер LoRa від Ebyte

У цьому контексті важливим є поняття бюджету лінії зв’язку, яке описує баланс між потужністю передавача, втратами сигналу в каналі та чутливістю приймача. У найпростішому вигляді бюджет лінії можна представити як різницю між потужністю переданого сигналу і мінімальним рівнем сигналу, який здатен прийняти приймач. Чим більший цей запас, тим більша дальність зв’язку або стійкість до перешкод. Висока чутливість приймача в LoRa–модулях, таких як SX1280, може досягати значень порядку –130 дБм і нижче, що означає здатність приймати надзвичайно слабкі сигнали. У поєднанні з CSS це дозволяє системі працювати в умовах значного зашумлення ефіру, наприклад у насиченому діапазоні 2.4 ГГц, де одночасно функціонують Wi–Fi, Bluetooth та інші бездротові технології.

Навіть якщо сигнал частково перекривається іншими джерелами випромінювання, механізми розширеного спектру та кореляційного прийому дозволяють відновити передані дані. Таким чином, висока ефективність LoRa у складних умовах досягається не за рахунок збільшення потужності передавання, а завдяки оптимізації всієї системи передачі сигналу: використанню широкосмугової модуляції, підвищеній чутливості приймача та грамотному формуванню бюджету лінії зв'язку. Це особливо важливо для відмовостійких систем, де необхідно гарантувати передачу телеметрії навіть при несприятливих умовах радіоканалу, що робить SX1280 ефективним рішенням для резервних каналів зв'язку. Таблицю 2.1 створено згідно з офіційною документацією Semtech:

Таблиця 2.1 – Технічні характеристики LoRa SX1280[4][9]

Характеристика	**SX1280/Ebyte**
Призначення	LoRa трансивер
Діапазон частот	2.1–2.7 ГГц
Модуляція	LoRa (CSS), FLRC, (G)FSK, BLE PHY
Чутливість приймача	до –132 dBm
Вихідна потужність	13 dBm
Інтерфейс керування	SPI
Живлення	3.3В
Додаткові функції	Ranging (ToF), завадостійкість
Робоча температура	–40°C … +85°C
Орієнтація	IoT, логістика, wearable

SX1280 є високочутливим 2.4 ГГц трансивером із підтримкою LoRa та альтернативних модуляцій, здатним працювати при рівнях сигналу до –132 dBm та забезпечувати дальній зв'язок навіть у зашумленому середовищі. ESP32, у свою чергу, є високопродуктивною двоядерною SoC з інтегрованими Wi–Fi та Bluetooth, значним набором периферії та низьким енергоспоживанням, що робить його придатним для складних IoT та телеметричних систем.

2.1.3 Переваги платформи ESP32 для розробки вбудованих систем

У контексті розробки ембеддед–систем важливим фактором вибору мікроконтролера є не лише його апаратні можливості, але й складність програмування, швидкість розробки та поріг входження. У цьому аспекті мікроконтролер ESP32 має суттєві переваги над класичними рішеннями на базі STM32, що особливо помітно при розробці прототипів, IoT–пристроїв та систем телеметрії.

Однією з ключових причин простоти програмування ESP32 є наявність добре інтегрованого середовища розробки та програмного стеку від Espressif Systems. Офіційний фреймворк ESP–IDF вже включає в себе драйвери для більшості периферії, мережеві стеки (Wi–Fi, TCP/IP), підтримку Bluetooth, файлові системи та засоби роботи з RTOS. Це означає, що розробнику не потрібно самостійно інтегрувати або налаштовувати значну частину базової функціональності – вона вже готова до використання. Для порівняння, у випадку STM32, хоча екосистема від STMicroelectronics також є потужною, вона часто вимагає більш глибокого налаштування, розуміння регістрів та взаємодії між модулями, особливо при роботі без високорівневих бібліотек.

Ще одним важливим аспектом є підтримка високорівневих середовищ та мов програмування. ESP32 широко використовується разом із платформами, такими як Arduino IDE або MicroPython, що дозволяє значно спростити написання коду, особливо для задач комунікації та швидкого прототипування. У таких середовищах багато складних речей, пов’язаних із ініціалізацією периферії чи налаштуванням стеків зв’язку, приховані за простими API. STM32 також може працювати з Arduino, але це не є основним сценарієм, і часто потребує додаткової конфігурації.

Важливою перевагою ESP32 є також більш “готова до використання” мережева функціональність. Інтегрований Wi–Fi та Bluetooth мають повністю реалізовані програмні стеки, які легко викликаються з коду. У випадку STM32, навіть якщо використовується мікроконтролер із вбудованим Ethernet або зовнішнім Wi–Fi–

модулем, розробнику часто доводиться самостійно інтегрувати стек, налаштовувати драйвери та вирішувати питання сумісності.

Крім того, ESP32 має добре документовану систему прикладів, бібліотек і готових рішень, що значно скорочує час розробки. Велика спільнота користувачів забезпечує швидкий доступ до типових рішень, що зменшує кількість помилок і спрощує налагодження. STM32 також має широку документацію, але вона часто є більш низькорівневою і орієнтованою на інженерів із досвідом роботи з апаратною частиною.

Ще один фактор – це інтеграція з операційною системою реального часу. В ESP32 використовується FreeRTOS від старту, причому вона вже оптимізована під двоядерну архітектуру та має зручні механізми створення задач. У STM32 використання RTOS є опціональним і зазвичай потребує додаткової інтеграції та налаштування, що ускладнює початковий етап розробки.

Таким чином, простота програмування ESP32 пояснюється високим рівнем інтеграції апаратних і програмних компонентів, наявністю готових стеків зв’язку, підтримкою високорівневих інструментів та великою кількістю прикладів. У порівнянні зі STM32, де часто потрібна більш глибока робота з апаратним рівнем і конфігурацією, ESP32 дозволяє швидше реалізовувати функціональність і зосередитися безпосередньо на логіці системи, що є важливим фактором при розробці складних, але гнучких телеметричних рішень.

2.1.4 Периферія

У межах розробки відмовостійкого ембеддед контролера вибір периферійних пристроїв визначається не лише функціональністю, але й вимогами до надійності, керованості та передбачуваності поведінки системи. Як об’єкти керування доцільно розглядати сервоприводи, оскільки вони є типовими виконавчими механізмами, що широко застосовуються в автоматизованих системах, робототехніці та телеметричних рішеннях. Управління сервоприводами здійснюється за допомогою широтно–імпульсної модуляції, що реалізується апаратними засобами мікроконтролера ESP32 і не потребує складних обчислювальних ресурсів.

Сервопривід працює на основі періодичного PWM–сигналу, де ключовим параметром є тривалість імпульсу в межах фіксованого періоду. Типово використовується частота близько 50 Гц, а ширина імпульсу в межах приблизно 1–2 мс визначає кут повороту валу. Такий спосіб керування є надзвичайно простим з точки зору реалізації, але водночас достатньо точним і повторюваним. Для відмовостійких систем це має принципове значення, оскільки дозволяє легко відстежувати коректність сигналу: відхилення у тривалості імпульсів або їх зникнення можуть однозначно інтерпретуватися як помилка або збій. Крім того, сервоприводи мають вбудований зворотний зв’язок, як наслідок позиційний контроль, що підвищує стабільність виконання команд без необхідності реалізації складних алгоритмів на стороні контролера. Рисунок 2.3 демонструє вигляд сервопривода

Рисунок 2.3 – Сервопривід SG90[20]

Застосування PWM також добре узгоджується з апаратними можливостями ESP32, який має спеціалізовані таймери та модулі генерації сигналів, що дозволяє формувати стабільний сигнал без навантаження на центральні ядра. Це особливо важливо при розподілі задач між ядрами, коли одне ядро обслуговує зв’язок, а інше – алгоритмічну частину. Генерація PWM відбувається незалежно, що підвищує загальну детермінованість системи.

Додатково до вже розглянутої периферії доцільно включити цифровий датчик температури та вологості AHT10, який працює по інтерфейсу I2C і широко

застосовується в ембеддед–системах завдяки простоті підключення та достатній точності вимірювань. Використання такого датчика дозволяє реалізувати базовий моніторинг навколишнього середовища або внутрішніх умов роботи пристрою, що є важливим елементом телеметрії у відмовостійких системах.

AHT10 є повністю цифровим сенсором, який виконує внутрішню обробку сигналів і передає вже готові значення температури та відносної вологості через I2C–шину. Це значно спрощує програмну реалізацію, оскільки відсутня необхідність у калібруванні або складній обробці аналогових сигналів, як у випадку з деякими іншими типами датчиків. Інтерфейс I2C дозволяє підключати декілька пристроїв до однієї шини, що мінімізує кількість необхідних виводів мікроконтролера ESP32 і спрощує масштабування системи. Рисунок 2.4 демонструє сенсор температури та вологості.

Рисунок 2.4 – Цифровий датчик температури та вологості AHT10[21]

Типові характеристики AHT10 включають діапазон вимірювання температури приблизно від –40°C до +85°C та вологості від 0 до 100%, з точністю порядку ±0.3°C для температури та близько ±2% для вологості. Живлення самого сенсора зазвичай становить 3.3 В, однак багато готових модулів оснащені стабілізатором та рівневими перетворювачами, що дозволяє використовувати їх у системах із живленням 5 В, що спрощує інтеграцію з різними платформами.

З точки зору відмовостійкості, використання такого датчика є доцільним не лише для збору корисних даних, але й як елемент діагностики. Наприклад, різке підвищення температури може свідчити про перегрів компонентів або аварійний режим роботи, а зміна вологості – про вплив зовнішнього середовища чи

порушення герметичності корпусу. Ці дані можуть передаватися через основний або резервний канал телеметрії (зокрема LoRa), що дозволяє віддалено оцінювати стан системи навіть у випадку часткової відмови інших підсистем.

Крім того, I2C–інтерфейс підтримує механізми підтвердження передачі ACK/NACK, що дозволяє виявляти помилки зв’язку на рівні протоколу. У разі відсутності відповіді від датчика система може ініціювати повторний запит або зафіксувати несправність периферійного вузла. Це добре узгоджується із загальною концепцією побудови відмовостійкої системи, де важливо не лише виконання функцій, але й контроль їх коректності.

Включення датчика AHT10 у склад периферії є обґрунтованим рішенням, що підвищує інформативність телеметрії, спрощує реалізацію вимірювань та додає додатковий рівень контролю стану системи без значного ускладнення апаратної чи програмної частини.

2.2 Розробка архітектури відмовостійкої вбудованої системи

Архітектура розроблюваної системи побудована за принципом Master–Slave із двома фізично незалежними вузлами на базі мікроконтролера ESP32, що взаємодіють через радіоканал LoRa на базі модуля Ebyte з трансивером SX1280. Такий розподіл обов'язків між вузлами є усвідомленим архітектурним рішенням, що відображає концепцію функціональної декомпозиції: кожен вузол несе відповідальність за чітко визначену підмножину задач і не має прямої залежності від внутрішнього стану іншого вузла окрім тих даних, що передаються через телеметричний канал. Майстер–вузол є активним виконавчим елементом системи і зосереджує в собі всю логіку взаємодії з фізичним середовищем: він здійснює керування сервоприводом Tower Pro SG90 через ШІМ–сигнал та зчитування температури і відносної вологості з датчика AHT10 через апаратний інтерфейс I2C. Слейв–вузол, натомість, є пасивним спостерігачем що не має безпосереднього зв'язку з виконавчими механізмами, а лише приймає телеметричні пакети від майстра, інтерпретує закодований у них статус системи і відображає його через триколірну світлодіодну індикацію та веб–інтерфейс. Така архітектура забезпечує

природну ізоляцію між рівнем керування та рівнем моніторингу, що є важливою властивістю відмовостійких систем: відмова слейв–вузла або втрата радіоканалу жодним чином не впливає на здатність майстра продовжувати виконання своїх основних функцій.

Принципово важливим організаційним рішенням є розміщення прошивок обох вузлів в єдиному репозиторії з одним конфігураційним файлом PlatformIO, де розподіл між ролями здійснюється на етапі компіляції через препроцесорний прапор ROLE_MASTER або ROLE_SLAVE, визначений у секціях відповідних середовищ збірки. Команда компіляції для конкретного вузла зводиться до вибору відповідного середовища: pio run –e master або pio run –e slave. Завдяки цьому спільний код, що описує структуру телеметричного пакету, константи протоколу LoRa та визначення станів FSM, компілюється в обох випадках, тоді як специфічний для кожної ролі код огороджується умовними директивами препроцесора і потрапляє у прошивку лише при компіляції відповідного середовища. Така організація усуває проблему розсинхронізації форматів пакетів між вузлами, яка є типовою помилкою при роздільній розробці передавального та приймального кінців протоколу, і гарантує що будь–яка зміна у структурі пакету автоматично враховується при наступній компіляції обох прошивок.

Програмна архітектура кожного вузла організована у вигляді трирівневої ієрархії що відображає класичну модель розділення відповідальності у вбудованих системах. Нижній рівень є рівнем апаратної абстракції і містить драйвери периферійних інтерфейсів що надають уніфікований програмний інтерфейс для роботи з конкретними апаратними блоками мікроконтролера. До цього рівня належать функції ініціалізації та циклічного зчитування датчика AHT10 через апаратний I2C контролер ESP32 за фіксованою адресою 0x38, функції генерації ШІМ–сигналу з керованою шпаруватістю для позиціонування сервоприводу SG90 засобами бібліотеки ESP32Servo, а також функції обміну даними з модулем Ebyte через чотирипровідний SPI–інтерфейс із окремими лініями вибору мікросхеми та переривання. Ключова властивість цього рівня полягає у тому що решта системи не

має прямого доступу до апаратних регістрів і взаємодіє з периферією виключно через визначені інтерфейсні функції, що дозволяє замінювати або модифікувати апаратну реалізацію без жодних змін у логіці вищих рівнів. Практичним наслідком є те що заміна датчика AHT10 на інший I2C–сумісний пристрій або зміна пінів підключення LoRa–модуля потребуватиме правок лише у файлах драйверного рівня і не зачепить логіку FSM або задачі FreeRTOS.

Середній рівень є сервісним і реалізований на основі FreeRTOS що постачається у складі фреймворку Arduino для ESP32 від Espressif. Саме цей рівень є серцем відмовостійкості системи з точки зору програмної архітектури. На ньому визначаються всі задачі операційної системи реального часу із відповідними пріоритетами та розмірами стеків, черги міжзадачної комунікації для передачі вимірювань датчика та команд сервоприводу між задачами, м'ютекси для захисту спільних структур даних від одночасного доступу з різних контекстів виконання, а також програмні таймери FreeRTOS що керують інтервалами відправки телеметрії та перевірки тайм–ауту на слейв–вузлі. Двоядерна природа ESP32 використовується свідомо: задачі з жорсткими часовими вимогами, зокрема зчитування датчика та обробка переривань від LoRa–модуля, прив'язуються до першого ядра, тоді як менш критичні задачі на кшталт обслуговування веб–сервера або формування JSON–відповідей можуть виконуватись на другому ядрі, що усуває взаємний вплив між підсистемами з різними часовими характеристиками.

Верхній прикладний рівень містить логіку скінченного автомата станів системи, алгоритми прийняття рішень щодо переходів між станами Green, Yellow та Red на підставі агрегованих діагностичних даних, формування та розбір телеметричних пакетів засобами бібліотеки ArduinoJson, а також логіку HTTP–сервера що обслуговує GET–запити на отримання поточного стану системи та POST–запити для надсилання керуючих команд у зворотному напрямку через LoRa–канал. Цей рівень оперує виключно абстракціями – станами, подіями, командами та вимірюваннями – і не містить жодного прямого звернення до

апаратних ресурсів, що робить його повністю незалежним від конкретної апаратної реалізації нижніх рівнів.

Механізми ізоляції помилок є центральним елементом всієї архітектури і реалізовані на кількох рівнях одночасно, утворюючи ешелоновану систему захисту. Першим і найглибшим рубежем є апаратний сторожовий таймер ESP32, що налаштовується на інтервал спрацювання який перевищує найдовший очікуваний цикл виконання будь–якої критичної задачі. Головна задача моніторингу скидає лічильник сторожового таймера у кожній своїй ітерації, і якщо це скидання не відбулось у межах встановленого інтервалу через зависання будь–якої задачі, блокування на м'ютексі, переповнення стеку або будь–яку іншу аномалію, таймер спрацьовує і ініціює апаратне перезавантаження мікроконтролера після якого система відновлює роботу з початкового стану ініціалізації. Це є останнім рубежем захисту що забезпечує відновлення системи навіть у тих аварійних сценаріях які не були явно передбачені при розробці логіки FSM, і саме наявність цього механізму дозволяє стверджувати, що система є відмовостійкною, а не продовжує працювати тільки в межах заздалегідь відомих видів відмов.

Другим рубежем є програмні тайм–аути при зверненні до периферії. Зчитування датчика AHT10 через I2C виконується з контролем часу очікування відповіді від пристрою, і якщо за визначений інтервал коректне вимірювання не отримано – через фізичну несправність датчика, порушення цілісності ліній SDA або SCL, або помилку ініціалізації – функція драйверного рівня негайно повертає код помилки замість того щоб блокувати викликаючу задачу у нескінченному очікуванні шини. Це принципово важливо для RTOS–середовища де зависання однієї задачі у заблокованому стані порушує часові гарантії для всіх задач з нижчим пріоритетом. Задача моніторингу на прикладному рівні аналізує повернені коди помилок та інкрементує відповідні лічильники, і лише після перевищення визначеного порогу послідовних помилок FSM виконує перехід у стан Yellow або Red, що запобігає реагуванню на поодинокі випадкові збої характерні для будь–якого реального середовища. Аналогічний підхід із тайм–аутами та лічильниками

помилок застосовується при взаємодії з LoRa–модулем через SPI, при перевірці цілісності отриманих пакетів на слейв–вузлі та при контролі часу між послідовними успішними прийомами телеметрії. Такий захисний стиль програмування де кожна операція з зовнішньою периферією має явно визначений граничний час виконання, чіткий шлях обробки помилки та гістерезис реакції на нестабільні умови є фундаментальним принципом побудови надійних вбудованих систем і пронизує всі рівні програмної архітектури розроблюваного проєкту від драйверів периферії до логіки прикладного рівня.

2.3 Проєктування логічної структури та протоколу передачі даних

Розробка формату пакету телеметрії є одним із найбільш відповідальних етапів проєктування розподіленої вбудованої системи, оскільки від цього рішення залежить не лише коректність передачі даних, а й стійкість системи до помилок каналу, ефективність використання обмеженої пропускної здатності LoRa–радіоканалу та складність реалізації приймальної сторони. Вибір між текстовим і бінарним форматом представлення даних у пакеті є принциповим питанням що потребує окремого обґрунтування, оскільки обидва підходи мають свої переваги і типові сценарії застосування.

Текстовий формат на кшталт JSON або CSV є природним вибором для систем де зручність налагодження важливіше за компактність. Однак для радіоканалу з модуляцією LoRa та обраним параметром SF9 час передачі одного байту є невід'ємною частиною загального часу перебування у ефірі, що безпосередньо впливає на споживання енергії та ймовірність колізій із іншими пристроями у тому самому частотному діапазоні. Якщо представити значення температури 23.4 у текстовому вигляді, воно займатиме п'ять байтів ASCII–символів плюс роздільники, тоді як те саме значення у бінарному форматі у вигляді знакового цілого числа з фіксованою точкою займає рівно два байти із масштабним коефіцієнтом 100 тобто значення 234 у типі int16_t. Економія у цьому конкретному випадку складає більше ніж вдвічі, і ця різниця стає ще більш відчутною якщо врахувати що текстовий пакет потребує також службових символів дужок, лапок та

імен полів. Бінарний формат також повністю детермінований за розміром: кожен пакет має однакову довжину незалежно від поточних значень полів, що суттєво спрощує синхронізацію на приймальній стороні та дозволяє завчасно розрахувати точний час передачі.

Структура бінарного пакету телеметрії розроблена таким чином щоб мінімальним обсягом даних передати максимально повну картину стану майстер–вузла. Загальний розмір пакету складає 10 байтів, що є компромісом між інформаційною насиченістю та часом передачі. Структура пакету ServoPacket містить такі поля: магічне число PKT_MAGIC для валідації на приймальній стороні, ідентифікатор типу PKT_SERVO, поточний кут сервоприводу у градусах як знакове ціле int16_t, ширину ШІМ–імпульсу у мікросекундах як uint16_t, код стану FSM masterState типу uint8_t, значення температури tempC10 масштабоване з коефіцієнтом 10 як int16_t, та значення вологості humPct10 масштабоване з коефіцієнтом 10 як uint16_t. Контроль цілісності пакету забезпечується вбудованим механізмом CRC трансивера SX1280 на рівні фізичного рівня LoRa, що звільняє прикладний рівень від необхідності ручного розрахунку контрольної суми. Алгоритм контролю цілісності базується на методі циклічного надлишкового коду, відомого під абревіатурою CRC, що є одним із найбільш поширених і математично обґрунтованих підходів до виявлення помилок у каналах зв'язку. Математична основа CRC полягає у представленні бітової послідовності даних як коефіцієнтів полінома над полем GF(2), тобто полем Галуа з двома елементами де додавання відповідає операції XOR, а множення виконується за модулем незвідного породжуючого полінома.

Для алгоритму CRC–16/CCITT що застосовується у даному проєкті породжуючий поліном має вигляд:

$$G(x) = x^{16} + x^{12} + x^5 + 1, \quad (1)$$

де x – фіктивна змінна (базис полінома).

Що у шістнадцятковому представленні відповідає значенню 0x1021. Процедура обчислення контрольної суми розгортається наступним чином: регістр

CRC ініціалізується початковим значенням 0xFFFF, після чого для кожного байту вхідних даних виконується операція XOR старшого байту регістру із поточним байтом даних, а потім регістр зсувається на вісім позицій вліво із одночасним виконанням XOR із значенням полінома якщо старший біт що вийшов за межі регістру дорівнює одиниці. Після обробки всіх байтів пакету у регістрі залишається двобайтове значення контрольної суми що передається у складі пакету і обчислюється повторно на приймальній стороні для порівняння.

Здатність CRC виявляти помилки визначається його теоретичними властивостями що випливають із алгебраїчної структури породжуючого полінома. CRC–16 з поліномом 0x1021 гарантовано виявляє всі одиночні та подвійні бітові помилки у пакеті будь–якої довжини, всі пакети помилок довжиною не більше 16 біт, а також всі непарні кількості бітових помилок. Для пакету розміром 10 байтів імовірність невиявленої помилки при рівномірному розподілі помилкових патернів складає приблизно 2^{-16} тобто близько 0,0015 відсотка, що є цілком прийнятним показником для телеметричної системи некритичного рівня безпеки.

У розробленій системі обчислення CRC делеговано апаратному блоку трансивера SX1280, який автоматично обчислює та перевіряє контрольну суму на рівні фізичного рівня LoRa. Це усуває необхідність програмної реалізації алгоритму та знижує обчислювальне навантаження на мікроконтролер.

Часовий аналіз передачі пакету є важливим для оцінки реальної затримки між виникненням події на майстрі та її відображенням на слейві. Час передачі одного символу LoRa при параметрах SF7 та смузі пропускання 800 кГц визначається як відношення кількості чіпів на символ до смуги пропускання: $T_s = 2^7 / 800000 = 128 / 800000 \approx 0.16$ мілісекунди на символ. Кожен байт корисних даних кодується як один символ при коефіцієнті кодування CR рівному одиниці, а до корисного навантаження додається преамбула що складається з восьми символів та службовий заголовок PHY–рівня довжиною п'ять байтів. Таким чином загальна кількість символів для передачі пакету із десятьма байтами корисного навантаження складає $\text{ceil}((8\times10 - 4\times7 + 28 + 16) / (4\times7)) \times (1 + 4) = \text{ceil}((80 - 28 + 28 + 16) / 28) \times 5 =$

ceil(96/28) × 5 = ceil(3.43) × 5 = 4 × 5 ≈ 20 символи, ще враховуємо преамбулу 8, та заголовок 4.25, виходить 33 символи, а час передачі усього пакету становить приблизно 33 × 0,16 ≈ 5.3 мілісекунд. З урахуванням інтервалу між пакетами що встановлений на рівні двох секунд та часу обробки пакету на слейві що не перевищує одного–двох мілісекунд, сукупна затримка між зміною стану на майстрі та оновленням індикації на слейві складає від 5.3 мілісекунд у найкращому випадку коли подія сталася безпосередньо перед відправкою пакету до приблизно двох секунд у найгіршому випадку коли подія сталася одразу після відправки попереднього пакету. Для задач оперативної діагностики роботизованої системи така затримка є цілком прийнятною і не впливає на безпеку або керованість системи.

Аналіз часових характеристик передачі аварійних повідомлень є принципово важливим для будь–якої системи що претендує на відповідність вимогам реального часу. У роботизованих системах поняття детермінізму означає не просто швидку реакцію, а гарантовану верхню межу затримки що не може бути перевищена ні за яких умов нормальної роботи системи. Саме ця гарантованість, а не абсолютна швидкість, є визначальною характеристикою системи реального часу відповідно до класичної теорії планування задач.

Для коректного розрахунку часу доставки повідомлення про помилку необхідно розглянути повний ланцюжок подій від моменту виникнення аномалії у фізичному середовищі до моменту зміни індикації на слейв–вузлі. Цей ланцюжок складається з кількох послідовних етапів кожен із яких вносить свою складову у загальну затримку і має власні часові характеристики що визначаються архітектурою системи.

Перший етап є етапом виявлення помилки на рівні драйвера периферії. Задача зчитування датчика AHT10 виконується з фіксованим періодом 100 мілісекунд відповідно до налаштувань планувальника FreeRTOS. Після надсилання команди вимірювання датчику AHT10 потребує не менше 75 мілісекунд для завершення перетворення, що є апаратним обмеженням зафіксованим у документації на

компонент. Тайм–аут очікування відповіді по I2C встановлений на рівні 10 мілісекунд понад цей час, тобто 85 мілісекунд від моменту надсилання команди. Якщо датчик не відповів у межах цього вікна, драйверна функція повертає код помилки і задача сенсора інкрементує лічильник послідовних збоїв. Затримка виявлення на цьому рівні у найгіршому випадку складає повний період задачі плюс тайм–аут: $T_{detect} = 100 + 85 = 185$ мілісекунд.

Другий етап є етапом реакції FSM на накопичені помилки. Задача моніторингу перевіряє лічильники помилок з власним періодом 50 мілісекунд. Для переходу у стан Yellow встановлено поріг трьох послідовних помилок що є свідомим гістерезисом для відсіювання одиночних випадкових збоїв характерних для реального I2C–середовища. Таким чином час від першої помилки до зміни стану FSM у найгіршому випадку складає три повних цикли задачі сенсора плюс один цикл задачі моніторингу: $T_{fsm} = 3 \times 100 + 50 = 350$ мілісекунд. Для переходу у стан Red при критичній помилці, наприклад при перевищенні температурного порогу що виявляється за одним вимірюванням, поріг становить одну подію і затримка реакції FSM скорочується до одного циклу задачі сенсора плюс один цикл моніторингу: $T_{fsm\_critical} = 100 + 50 = 150$ мілісекунд.

Третій етап є етапом формування та відправки телеметричного пакету. Задача LoRaTxTask виконується з інтервалом в 1 секунду і не синхронізована із задачами сенсора та моніторингу навмисно, щоб уникнути одночасного пробудження кількох задач що погіршує детермінізм планувальника. У найгіршому випадку зміна стану FSM відбувається одразу після того як задача передачі надіслала попередній пакет, і тоді до наступної відправки залишається повні дві секунди. У найкращому випадку зміна стану збігається у часі із моментом активації задачі передачі і пакет із оновленим статусом надсилається негайно. Математичне очікування затримки на цьому етапі становить рівно одну секунду як половину від максимального інтервалу: $E[T_{tx}] = 1000$ мілісекунд. Сам процес передачі пакету займає близько 5.3 мілісекунд як розраховано раніше.

Четвертий етап є етапом обробки пакету на слейві. Після отримання переривання від LoRa–модуля через лінію DIO1 задача прийому пробуджується із заблокованого стану що займає не більше одного тіку планувальника FreeRTOS, тобто одну мілісекунду при стандартному налаштуванні частоти тіку 1000 Гц. Зчитування пакету через SPI, перевірка CRC та оновлення внутрішнього стану займають ще приблизно два–три мілісекунди залежно від поточного завантаження шини. Таким чином затримка обробки на слейві: T_slave = 1 + 3 = 4 мілісекунди.

Зводячи всі складові разом можна отримати повну картину часових характеристик доставки аварійного повідомлення. Для некритичної помилки що вимагає підтвердження трьома послідовними збоями затримка у найгіршому випадку складає суму всіх етапів:

T_worst = 185 + 350 + 1000 + 5.3 + 4 ≈ 1544 мс ≈ 1.5 с T_best = 185 + 150 + 0 + 5.3 + 4 ≈ 344 мс

Тобто приблизно 2.6 секунди. Для критичної помилки що вимагає негайної реакції без підтвердження:

T_worst_critical = 185 + 150 + 1000 + 5.3 + 4 ≈ 1344 мс ≈ 1.3 с

У найкращому випадку коли помилка виявлена і FSM відреагувала безпосередньо перед плановою відправкою пакету:

T_best = 185 + 150 + 0 + 5.3 + 4 ≈ 344 мс

Ці цифри потребують інтерпретації у контексті вимог до реального часу роботизованих систем. З точки зору класифікації систем реального часу розроблювана система є системою м'якого реального часу відносно каналу телеметрії: перевищення розрахованого часового вікна призводить до деградації якості моніторингу але не до катастрофічних наслідків для керованого об'єкту. Це є принциповою відмінністю від жорстких систем реального часу де перевищення дедлайну є відмовою системи за визначенням. Водночас внутрішня реакція майстер–вузла на аварійну подію, а саме переведення сервоприводу у безпечне положення та зміна стану FSM, підпорядковується жорстким часовим вимогам і відбувається повністю у межах локального вузла без залежності від LoRa–каналу.

Затримка цієї внутрішньої реакції визначається виключно часом виявлення помилки та одним циклом задачі моніторингу і складає не більше 235 мілісекунд, що є достатнім для безпечного керування сервоприводом класу SG90 із механічною постійною часу порядку кількох сотень мілісекунд. Таким чином архітектурний розподіл між локальною реакцією на відмову та дистанційним сповіщенням через LoRa–канал забезпечує виконання жорстких часових вимог для критично важливих функцій безпеки незалежно від характеристик радіоканалу.

2.4 Розробка алгоритмічного забезпечення моніторингу та діагностики

Під час розробки з додаванням вузлів для роботи з’явилася потреба в тестуванні та діагностиці роботи модулів, з метою полегшення розробки та для кращої архітектури було створено блок–схеми описуючі стани та успіх чи не успіх операції

2.4.1 Алгоритм master частини

Скінченний автомат станів є центральним архітектурним елементом програмного забезпечення майстер–вузла і визначає поведінку системи в цілому залежно від сукупності діагностичних показників що збираються у кожному циклі виконання. Реалізація FSM у даному проєкті суттєво відрізняється від канонічної академічної моделі де переходи між станами відбуваються за чітко визначеними дискретними подіями: натомість тут стан системи є результатом безперервного аналізу накопичених лічильників помилок що оновлюються у кожному циклі передачі телеметрії з інтервалом одна секунда. Такий підхід реалізує природний гістерезис переходів між станами без необхідності явного кодування логіки затримки, оскільки сам механізм накопичення помилок автоматично фільтрує поодинокі випадкові збої та реагує лише на стійкі аномалії.

Перш ніж розглядати окремі стани варто описати загальну структуру даних що лежить в основі FSM. Система оперує двома незалежними лічильниками: sensorFails що відраховує кількість послідовних невдалих зчитувань датчика AHT10 та txFails що відраховує кількість послідовних невдалих передач через LoRa–канал. Обидва лічильники скидаються до нуля при першому успішному

виконанні відповідної операції, що означає що система здатна повернутись до кращого стану вже після однієї успішної операції за умови що лічильник ще не перевищив критичний поріг. На додаток до цих лічильників FSM враховує два бінарних прапори що встановлюються під час ініціалізації: ahtOk що вказує на фізичну доступність датчика та servo.attached() що підтверджує успішне підключення сервоприводу. Ці прапори мають вищий пріоритет над лічильниками помилок і здатні самостійно визначити стан системи незалежно від накопичених значень. З спрощеною схемою можна ознайомитися на рисунку 2.5.

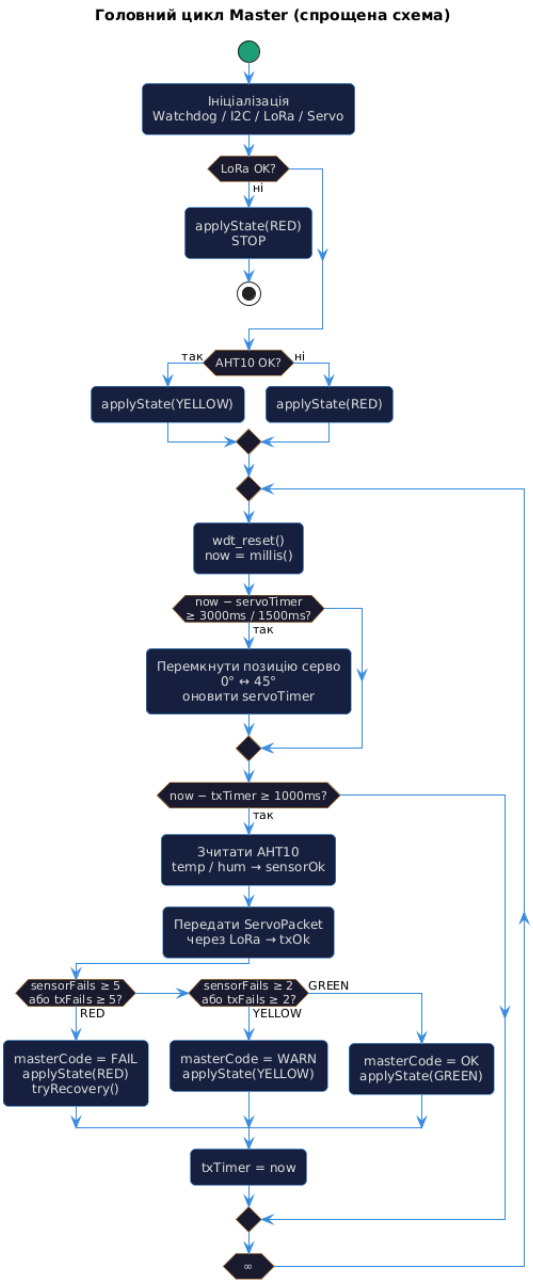

Рисунок 2.5 – Алгоритм Master

Стан GREEN є цільовим робочим станом системи і означає що всі підсистеми функціонують коректно у межах встановлених параметрів. З точки зору коду цей стан досягається коли masterCode набуває значення MASTER_OK, що відбувається лише за одночасного виконання кількох умов. По–перше датчик AHT10 має бути фізично доступний, тобто прапор ahtOk встановлений у true за результатами ініціалізації або успішного відновлення. По–друге сервопривід має бути підключений і відповідати на запити через servo.attached(). По–третє обидва лічильники помилок sensorFails та txFails мають бути строго меншими за поріг YELLOW що дорівнює двом, тобто фактично дорівнювати нулю або одиниці. Остання умова є принципово важливою: одна помилка підряд не переводить систему навіть у стан YELLOW, що відображає реалістичне ставлення до поодиноких збоїв на I2C–шині або в радіоканалі які є цілком нормальним явищем у реальному середовищі експлуатації. Перебування у стані GREEN супроводжується світінням зеленого світлодіода на слейв–вузлі та відповідним записом у JSON–відповіді веб–сервера зі стрічкою стану "All systems OK" що формується через таблицю MASTER_MAP на стороні слейва.

Стан YELLOW є проміжним попереджувальним станом що сигналізує про наявність нестабільності у роботі однієї або кількох підсистем без повної втрати функціональності. Система переходить у цей стан коли masterCode набуває одного з двох значень: MASTER_SENSOR_WARN або MASTER_TX_WARN. Перший код встановлюється коли лічильник sensorFails досягає порогу YELLOW рівного двом але ще не перевищує поріг RED рівний п'яти, тобто датчик не відповів або повернув некоректні дані двічі або більше поспіль але менш ніж п'ять разів. Другий код аналогічно встановлюється при досягненні лічильником txFails того самого порогу. Важливо що у стані YELLOW система продовжує виконувати всі свої основні функції в повному обсязі: сервопривід продовжує циклічно переміщатись між позиціями, спроби зчитування датчика та передачі телеметрії виконуються з тим самим інтервалом одна секунда, і у разі успіху відповідного лічильника скидається до нуля що призводить до автоматичного повернення у стан GREEN. Таким чином

стан YELLOW є не лише сигналом про проблему, а й активним станом самовідновлення де система продовжує спроби нормалізувати роботу несправної підсистеми без будь–якого зовнішнього втручання. На слейв–вузлі цей стан відображається світінням жовтого світлодіода та відповідним описом у веб–інтерфейсі: "Sensor intermittent" при проблемах з датчиком або "TX intermittent" при проблемах з передачею.

Окремим тригером переходу у стан YELLOW є ситуація що виникає на стороні слейва незалежно від стану майстра: деградація якості радіоканалу що виражається у збільшенні інтервалу між отриманими пакетами. Слейв–вузол відстежує час від моменту останнього успішного прийому через змінну lastRxTime і якщо цей інтервал перевищує три секунди але залишається меншим за вісім секунд, слейв самостійно переходить у стан YELLOW із повідомленням "Link degraded" незалежно від того який стан передає майстер у своїх пакетах. Це є важливою властивістю розподіленої архітектури: кожен вузол здатний самостійно оцінювати якість зв'язку і відображати її у своєму стані без покладання на діагностику протилежного вузла.

Стан RED є аварійним станом що сигналізує про критичну несправність яка унеможливлює нормальне функціонування системи або суттєво знижує рівень довіри до її вихідних даних. Перехід у цей стан може бути викликаний кількома категоріями тригерів що мають різну природу та різний механізм виявлення. Перша категорія охоплює структурні несправності що виявляються під час ініціалізації: якщо датчик AHT10 не відповів на I2C–адресі 0x38 під час aht10Init, прапор ahtOk залишається false і masterCode негайно набуває значення MASTER_NO_SENSOR без необхідності накопичення будь–яких лічильників помилок. Аналогічно якщо servo.attached() повертає false що може статись при апаратній несправності або некоректній ініціалізації, masterCode набуває значення MASTER_NO_SERVO. Друга категорія охоплює накопичені помилки під час роботи: досягнення лічильником sensorFails значення п'ять встановлює код MASTER_SENSOR_FAIL, а аналогічне досягнення txFails встановлює MASTER_TX_FAIL. П'ять послідовних

секундних циклів із невдалими зчитуваннями або передачами відповідають п'яти секундам реального часу що є достатнім вікном для однозначного висновку про стійку несправність на противагу тимчасовій нестабільності. Третя категорія стосується виключно слейв–вузла: якщо інтервал від останнього отриманого пакету перевищує вісім секунд, слейв переходить у стан RED із повідомленням "Link lost" що означає повну втрату радіозв'язку з майстром.

Поведінка системи у стані RED принципово відрізняється від поведінки у стані YELLOW у одному ключовому аспекті: активується механізм автономного відновлення tryRecovery що запускається у кінці кожного виклику updateState при виявленні RED–стану. Цей механізм із інтервалом 10 секунд повторює спроби ініціалізації саме тієї підсистеми що спричинила перехід у RED: при кодах MASTER_SENSOR_FAIL або MASTER_NO_SENSOR повторюється повна процедура aht10Init із скиданням і калібровкою датчика, при коді MASTER_TX_FAIL повторюється loraInit із повторною ініціалізацією SPI та трансивера SX1280. Якщо відновлення пройшло успішно відповідний лічильник помилок скидається до нуля і вже у наступному циклі updateState система отримує шанс перейти у кращий стан без перезавантаження мікроконтролера. Це є ключовим архітектурним рішенням що відрізняє справжню відмовостійку систему від простої системи з watchdog–перезавантаженням: замість сліпого перезапуску всього програмного забезпечення система хірургічно відновлює лише ту підсистему що відмовила, зберігаючи при цьому накопичений контекст виконання, стан сервоприводу та усі часові лічильники.

Логіка об'єднання стану на слейв–вузлі реалізована за принципом найгіршого результату через функцію getOverallState де стан каналу зв'язку та стан що повідомляє майстер комбінуються таким чином що гірший з двох завжди визначає результуючий стан відображення. При цьому проблеми з каналом зв'язку мають пріоритет відображення над проблемами майстра: якщо і канал і майстер одночасно перебувають у RED–стані, у веб–інтерфейсі відображається причина пов'язана саме з каналом, оскільки вона є більш критичною з точки зору оперативного реагування

і вказує на проблему що може приховувати будь–яку інформацію про реальний стан майстра.

2.4.2 Алгоритм slave частини

Слейв–вузол реалізує якісно складнішу логіку індикації ніж може здатись на перший погляд, оскільки він змушений одночасно враховувати два незалежних джерела діагностичної інформації: стан що повідомляє майстер у телеметричних пакетах та власну оцінку якості радіоканалу що базується на аналізі часових інтервалів між отриманими пакетами. Саме поєднання цих двох джерел через алгоритм вибору найгіршого результату і формує остаточне рішення щодо індикації, що відображається як на світлодіодах так і у веб–інтерфейсі.

Фізично прийом пакетів та оновлення індикації розведені на два різних ядра ESP32 що є принциповим архітектурним рішенням. Задача loraRxTask що виконується на Core 0 займається виключно прийомом пакетів через блокуючий виклик LT.receive із тайм–аутом п'ять секунд і записом отриманих даних у volatile–поля спільної пам'яті. Головний цикл slaveLoop на Core 1 натомість займається читанням цих полів, обчисленням поточного стану системи та оновленням індикації і веб–сервера. Такий розподіл гарантує що навіть під час тривалого очікування пакету на Core 0 веб–сервер і світлодіоди на Core 1 продовжують реагувати на запити без жодних затримок.

Центральним елементом логіки виявлення втрати сигналу є змінна lastRxTime що зберігає мітку часу отримання останнього валідного пакету від майстра. Вона оновлюється виключно у задачі loraRxTask після успішної перевірки магічного числа та типу пакету, що означає що випадковий радіошум або фрагменти чужих пакетів не можуть скинути лічильник і штучно продовжити час видимості майстра. При кожній ітерації slaveLoop обчислюється різниця між поточним millis() та lastRxTime, і саме це значення age є первинним показником стану радіоканалу. Особливим випадком є момент одразу після завантаження слейва коли жодного пакету ще не було отримано і lastRxTime дорівнює нулю. Для коректної обробки цього сценарію введена змінна startTime що ініціалізується під час slaveSetup і

використовується як базова точка відліку замість lastRxTime якщо той ще не оновлювався. Завдяки цьому система не переходить одразу у аварійний стан при першому ввімкненні а натомість починає відлік від моменту завантаження що дає майстру розумний час для встановлення зв'язку.

Порогові значення для класифікації стану каналу вибрані з урахуванням реальних часових характеристик системи. Інтервал відправки пакетів майстром становить одну секунду, а тайм–аут прийому на слейві встановлений на п'ять секунд, що означає що у нормальних умовах age ніколи не перевищує трохи більше однієї секунди. Поріг переходу у стан YELLOW встановлений на три секунди що відповідає пропуску двох–трьох пакетів підряд і є достатнім для відсіювання природних варіацій затримки LoRa–каналу. Поріг переходу у стан RED встановлений на вісім секунд що відповідає пропуску семи–восьми пакетів підряд і однозначно свідчить про стійку втрату зв'язку а не про тимчасову деградацію каналу. Такий дворівневий гістерезис дозволяє оператору розрізняти якісно різні ситуації: нестабільний але присутній зв'язок при якому пакети іноді доходять та повна відсутність зв'язку при якій жодних пакетів не надходить протягом тривалого часу.

Результуючий стан для індикації формується у функції getOverallState через об'єднання стану каналу та стану що повідомляє майстер за принципом найгіршого результату. Якщо обидва джерела показують GREEN система перебуває у GREEN–стані і зелений світлодіод вмикається через applyState. Якщо хоча б одне джерело показує YELLOW а інше не гірше за YELLOW результуючий стан є YELLOW і вмикається жовтий світлодіод. Якщо хоча б одне джерело показує RED результуючий стан є RED незалежно від іншого джерела. При рівності тяжкості станів пріоритет відображення причини надається стану каналу зв'язку над станом майстра, оскільки проблема з каналом є більш критичною з точки зору інформативності: при втраченому зв'язку будь–який стан що повідомляє майстер є застарілим і не заслуговує на довіру.

Оновлення світлодіодної індикації відбувається у кожній ітерації slaveLoop через виклик applyState із результатом getOverallState. Функція applyState є спільною для обох вузлів і реалізована у файлі system_state.cpp де вона керує трьома GPIO до яких підключені світлодіоди через струмообмежувальні резистори. При переході у стан GREEN вмикається лише зелений світлодіод і вимикаються жовтий та червоний. При переході у YELLOW вмикається жовтий і вимикаються інші два. При переході у RED вмикається червоний і вимикаються решта. Важливою властивістю є те що applyState викликається також і у обробнику HTTP–запиту handleData що означає що навіть якщо браузер клієнта надіслав GET–запит на /data між двома ітераціями головного циклу, світлодіоди все одно отримають актуальне оновлення без необхідності чекати наступної ітерації slaveLoop. Це усуває потенційну розсинхронізацію між станом що відображається у веб–інтерфейсі та фізичною індикацією на світлодіодах.

Веб–інтерфейс доповнює світлодіодну індикацію детальною текстовою інформацією про причину поточного стану через поле reason у JSON–відповіді. При стані GREEN причина відображається як "All systems OK" що береться з таблиці MASTER_MAP відповідно до коду MASTER_OK отриманого від майстра. При стані YELLOW причиною може бути або "Link degraded" якщо проблема у каналі зв'язку або "Sensor intermittent" чи "TX intermittent" якщо проблема на стороні майстра. При стані RED причиною може бути "Link lost" при тайм–ауті каналу або одне з повідомлень таблиці MASTER_MAP при критичній несправності на стороні майстра: "Sensor not found", "Servo detached", "Sensor failed" або "TX failed". Така деталізація причини стану є суттєвою перевагою веб–інтерфейсу над простою триколірною індикацією, оскільки дозволяє оператору одразу зрозуміти не лише факт наявності проблеми а й її природу та місце виникнення без необхідності підключення до серійного порту або іншого діагностичного інтерфейсу. Це дає оператору змогу розрізнити два якісно різні сценарії: майстер працює, але повідомляє про проблему, і майстер або канал зв'язку недоступні взагалі. Реалізація мигання світлодіода в середовищі FreeRTOS виконується без використання

затримок типу vTaskDelay у задачі індикації в блокуючому режимі для інших задач, натомість застосовується програмний таймер FreeRTOS, який з визначеним інтервалом інвертує стан відповідного GPIO. Це дозволяє задачі веб–сервера та задачі прийому продовжувати роботу без жодних пауз навіть під час активної аварійної індикації. Веб–інтерфейс слейва також реагує на тайм–аут окремим чином. Якщо в нормальному режимі сторінка відображає актуальні дані телеметрії з останнього пакету разом із міткою часу його отримання, то при детектуванні втрати зв'язку інтерфейс виводить візуальне попередження про відсутність даних і час, що минув з моменту останнього успішного прийому. Вигляд циклу на рисунку 2.6.

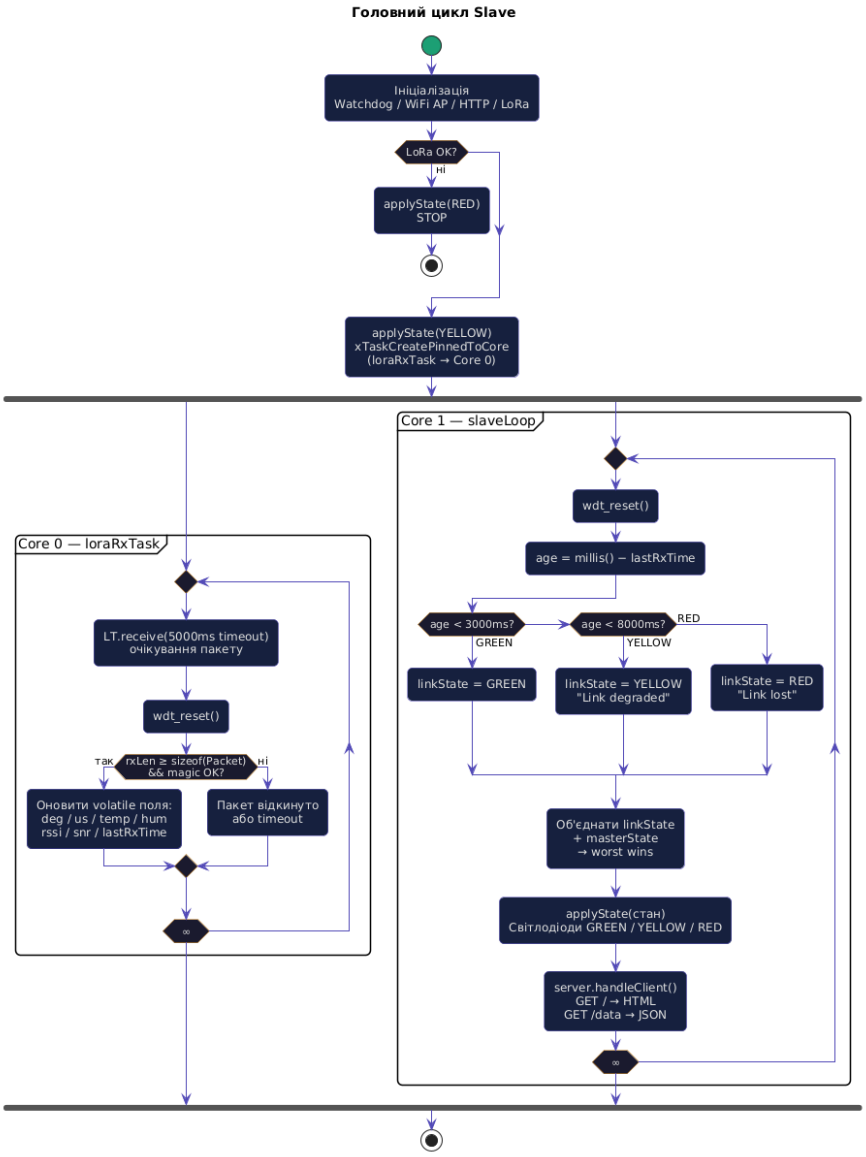

Рисунок 2.6 – Алгоритм Slave

Така інформація є критично важливою для оператора, оскільки дозволяє оцінити тривалість простою і прийняти рішення про необхідність втручання. Оновлення веб–сторінки реалізовано через механізм періодичних запитів з боку браузера, що не вимагає підтримки постійного з'єднання і добре узгоджується з обмеженими ресурсами ESP32 у ролі HTTP–сервера.

2.4.3 Розрахунок завадостійкості

Вибір LoRa як фізичного рівня передачі телеметрії в розроблюваній системі обумовлений не лише простотою інтеграції, а й фундаментальними властивостями модуляції на основі chirp–сигналів, які забезпечують виняткову стійкість до завад у порівнянні з традиційними вузькосмуговими протоколами. Для кількісної оцінки пропускної здатності каналу зв'язку між майстер та слейв вузлами доцільно застосувати теорему Шеннона–Хартлі, яка встановлює теоретичну межу швидкості передачі інформації через канал із адитивним білим гауссівським шумом. Теорема формулюється таким чином: максимальна пропускна здатність каналу C вимірювана в бітах за секунду дорівнює добутку смуги пропускання B на двійковий логарифм від суми одиниці та відношення сигнал–шум SNR, тобто :

$$C = B \cdot \log_2(1 + SNR), \quad (2)$$

де B — смуга пропускання каналу в герцах

SNR — відношення сигнал–шум

$\log_2(1 + SNR)$ – інформаційна ємність одного герца смуги

Для LoRa–модуля SX1280, який типово використовується разом із ESP32, типова смуга пропускання складає 125 кГц, а чутливість приймача при SF12 досягає мінус 137 дБм. Якщо прийняти рівень потужності передавача 17 дБм та врахувати втрати в тракті, відношення сигнал–шум на вході приймача в умовах прямої видимості на відстані кількох сотень метрів складатиме приблизно 10–15 дБ у лінійній шкалі від 10 до 31. Підставляючи ці значення, отримуємо теоретичну пропускну здатність у діапазоні від 415 кбіт/с до 500 кбіт/с. Важливо розуміти, що реальна швидкість передачі даних у LoRa є значно нижчою через особливості самої модуляції та механізм розширення спектру, тому теорема Шеннона–Хартлі тут

задає лише верхню теоретичну межу, яку практична система апріорі не може перевищити.

Ключовим параметром, що визначає реальний компроміс між швидкістю передачі та дальністю зв'язку у LoRa, є коефіцієнт розширення спектру, відомий під назвою Spreading Factor. Цей параметр приймає цілочисельні значення від 7 до 12 і визначає кількість чірпів, на яку кодується кожен символ: при SF7 один символ кодується 128 чіпами, а при SF12 – вже 4096 чіпами. Збільшення кількості чіпів на символ означає, що енергія одного біту інформації розподіляється на значно більший часовий інтервал, що дозволяє приймачу виділяти сигнал навіть при рівні потужності нижче рівня шуму. Саме ця властивість є основою стійкості LoRa до завад і принципово відрізняє цю технологію від звичайних вузькосмугових систем, де зниження відношення сигнал–шум нижче певного порогу призводить до повної втрати зв'язку. Швидкість передачі символів Rs визначається як відношення смуги пропускання B до кількості чіпів на символ 2^SF, тобто

$$Rs = B / 2^{SF}, \quad (3)$$

де B – смуга пропускання в герцах(Bandwidth);

2^SF – кількість чіпів на один символ(Spreading Factor).

При смузі 125 кГц та SF7 швидкість складе приблизно 977 символів за секунду, тоді як при SF12 вона знижується до 30 символів за секунду. Ефективна бітова швидкість при цьому розраховується з урахуванням коефіцієнта завадостійкого кодування CR за формулою

$$Rb = SF \cdot (4 / (4 + CR)) \cdot Rs \quad (4)$$

де SF — Spreading Factor, кількість біт що несе один символ;

4 / (4 + CR) — коефіцієнт завадостійкого кодування.

При SF7 та CR рівному одиниці реальна швидкість становить близько 5470 біт за секунду, а при SF12 вона падає до 293 біт за секунду. Для задач телеметрії в розроблюваній системі, де розмір одного пакету не перевищує кількох десятків байт і надсилається з інтервалом у кілька секунд, навіть найнижча швидкість є цілком

достатньою, що відкриває можливість використовувати високі значення SF для максимізації дальності та завадостійкості.

Залежність між SF та ефективною чутливістю приймача є лінійною у логарифмічній шкалі: кожне збільшення SF на одиницю покращує чутливість приблизно на 2.5 дБ та збільшує час передачі пакету вдвічі. Так, перехід від SF7 до SF12 дає виграш у чутливості близько 12.5 дБ, що в умовах вільного простору відповідає збільшенню дальності зв'язку приблизно у 4.2 рази згідно з моделлю втрат у вільному просторі. Практично це означає, що система, яка при SF7 стабільно працює на 200 метрах, при SF12 зберігатиме надійний зв'язок на відстані до 800–900 метрів в умовах міської забудови. Для конкретної конфігурації розроблюваної системи обрано SF9 як компромісне значення: воно забезпечує ефективну швидкість близько 1758 біт за секунду при чутливості мінус 129 дБм, що з запасом перекриває потреби телеметричного каналу і водночас залишає достатній енергетичний резерв для стабільної роботи в умовах можливих перешкод. Час передачі одного пакету розміром 20 байт при цьому складає приблизно 330 мілісекунд, що є прийнятним з точки зору затримки відображення статусу на слейв–вузлі та не створює надмірного навантаження на радіоканал. Таким чином, аналітичний вибір параметрів LoRa–модуляції підтверджує обґрунтованість використання цього протоколу для передачі телеметрії між вузлами системи в умовах реального розгортання.

2.4.4 Електрична схема(структурна)

Структурна електрична схема майстер–вузла відображає логічну організацію підсистем живлення, розподіл напруг між компонентами та взаємозв'язки між функціональними блоками без деталізації схемотехніки окремих каскадів. Архітектура живлення побудована з урахуванням реальних вимог до стабільності напруги, імпульсного характеру споживання сервоприводу та необхідності забезпечення кількох альтернативних джерел вхідного живлення.

Основним джерелом живлення майстер–вузла є літій–полімерний акумулятор у конфігурації 2S, що забезпечує номінальну напругу 7.4 вольта при двох

послідовно з'єднаних комірках та максимальну напругу заряду 8.4 вольта. Діапазон вхідної напруги від 6.4 до 8.4 вольта відповідає робочому діапазону обраного імпульсного знижувального перетворювача що є ключовою вимогою до узгодженості компонентів силової частини. Використання акумулятора 2S є обґрунтованим рішенням з точки зору енергетичного балансу: вхідна напруга достатньо висока щоб знижувальний перетворювач працював у режимі з високим ККД, і водночас не настільки велика щоб вимагати спеціальних заходів ізоляції між силовими та сигнальними ланцюгами.

Імпульсний знижувальний перетворювач є центральним вузлом підсистеми живлення і виконує перетворення вхідної напруги 2S акумулятора у стабілізовані 5 вольт що є основною шиною живлення системи. Імпульсна топологія обрана замість лінійного стабілізатора насамперед через значну різницю між вхідною та вихідною напругою: при лінійній стабілізації різниця у 2.4–3.4 вольта при струмі споживання до одного ампера означала б розсіювання потужності від 2.4 до 3.4 ват безпосередньо на корпусі стабілізатора, що є неприйнятним для компактної системи без активного охолодження. Імпульсний перетворювач з ККД порядку 85–92 відсотків вирішує цю проблему і дозволяє значно ефективніше використовувати ємність акумулятора.

На виході п'яти вольт лінії після перетворювача встановлено захисний діод що виконує кілька функцій одночасно. По–перше він захищає перетворювач від зворотного струму який може виникати при одночасному підключенні кількох джерел живлення, що є принципово важливим в архітектурі з кількома альтернативними джерелами. По–друге він забезпечує автоматичне перемикання між джерелами за принципом діодного або: якщо одночасно підключені і акумулятор через перетворювач і живлення через USB, активним джерелом стає те що забезпечує вищу напругу після діодного падіння. Падіння напруги на діоді Шотткі складає приблизно 0.3–0.4 вольта що знижує шину з 5 вольт до 4.6–4.7 вольта, що є прийнятним для всіх компонентів системи оскільки ESP32,

сервопривід SG90 та датчик AHT10 зберігають коректну роботу при напрузі живлення від 3.3 до 5 вольт і від 4.8 до 6 вольт відповідно.

Електролітичний конденсатор ємністю 220 мікрофарад із робочою напругою 16 вольт встановлений безпосередньо на шині живлення після діода і виконує функцію локального енергетичного резервуару що компенсує імпульсні провали напруги. Фізична природа цієї проблеми полягає у наступному: сервопривід SG90 під час зміни позиції споживає імпульсний струм амплітудою до 500–600 міліампер тривалістю кілька мілісекунд, тоді як середнє споживання у стані утримання позиції складає лише 150–200 міліампер. Цей імпульс струму протікає через внутрішній опір акумулятора, провідники та перетворювач і спричиняє миттєвий провал напруги на шині живлення. Без накопичувального конденсатора такий провал може досягати кількох десятих вольта і тривати достатньо довго щоб спричинити перезавантаження ESP32 або збій у роботі I2C–шини. Конденсатор 220 мікрофарад здатний підтримати струм 500 міліампер протягом приблизно 0.9 мілісекунди при допустимому провалі напруги 2 вольти, що перекриває тривалість типового пускового струмового піку сервоприводу.

Керамічний конденсатор що встановлений поряд з електролітичним виконує принципово іншу функцію попри зовнішню схожість призначення. Електролітичний конденсатор завдяки великій ємності ефективно фільтрує низькочастотні пульсації та компенсує повільні провали напруги, однак через значну власну індуктивність він погано справляється із високочастотними завадами у діапазоні від кількох мегагерц і вище. Керамічний конденсатор ємністю зазвичай від 100 нанофарад до одного мікрофарада має на кілька порядків меншу власну індуктивність і ефективно шунтує саме ці високочастотні завади що генеруються імпульсним перетворювачем на частоті комутації та поширюються по шинах живлення до чутливих аналогових і цифрових вузлів. Для ESP32 з його чутливим радіочастотним трактом Wi–Fi наявність керамічного конденсатора безпосередньо біля виводів живлення є особливо важливою оскільки високочастотні пульсації на

шині живлення безпосередньо погіршують параметри передавача і приймача вбудованого радіомодуля.

Розподіл живлення між компонентами організований таким чином що п'ятивольтова шина після діода і конденсаторів є спільною точкою живлення для сервоприводу SG90, датчика AHT10 та мікроконтролера ESP32. Сервопривід підключається безпосередньо до цієї шини без додаткових стабілізаторів оскільки його споживання вже враховане при виборі параметрів накопичувального конденсатора. Датчик AHT10 також живиться від п'ятивольтової шини, проте його споживання є незначним і не перевищує кількох міліампер, тому будь–які пульсації що виникають від датчика є несуттєвими. Мікроконтролер ESP32 живиться від тієї самої п'ятивольтової шини через вбудований лінійний стабілізатор на платі розробника DevKit що формує внутрішнє живлення 3.3 вольта для ядра процесора та периферії. Модуль Ebyte із трансивером SX1280 живиться безпосередньо від виводів ESP32 що забезпечують напругу 3.3 вольта, оскільки модуль розрахований саме на цей рівень живлення і має власний вбудований DC–DC перетворювач для живлення радіочастотного тракту.

Архітектура з кількома альтернативними джерелами живлення є важливою практичною особливістю системи. Окрім основного живлення від акумулятора 2S через перетворювач система допускає живлення через роз'єм USB Type–C безпосередньо на плату ESP32 DevKit, що є зручним під час розробки та налагодження коли підключення акумулятора небажане. При одночасному підключенні обох джерел діод на виході перетворювача забезпечує що активним є джерело з вищою напругою, і перемикання між джерелами відбувається автоматично без будь–якого програмного втручання. Можливість підключення додаткового зовнішнього джерела живлення постійного струму через відповідний роз'єм розглядається як третій варіант що може бути корисним при стаціонарному розгортанні системи де стабільність мережевого живлення є пріоритетом над мобільністю акумуляторного живлення.

2.4.5 Стек протоколів веб–інтерфейсу

Веб–інтерфейс слейв–вузла реалізований на основі вбудованого HTTP–сервера що функціонує безпосередньо на мікроконтролері ESP32 без залучення будь–яких зовнішніх обчислювальних ресурсів або хмарних сервісів. Вибір саме такого підходу до організації діагностичного інтерфейсу є результатом свідомого аналізу альтернативних технологій з урахуванням обмежень апаратної платформи, вимог до доступності системи моніторингу та складності реалізації в умовах обмеженого обсягу оперативної пам'яті ESP32.

З точки зору мережевого стеку взаємодія організована за класичною багаторівневою моделлю. На фізичному рівні та рівні канального доступу функціонує вбудований модуль Wi–Fi мікроконтролера ESP32 що підтримує стандарт IEEE 802.11 b/g/n у діапазоні 2.4 гігагерц. Слейв–вузол налаштований у режимі програмної точки доступу через виклик WiFi.softAP із визначеними у config.h ідентифікатором мережі та паролем. Такий режим роботи є принциповим для автономного розгортання системи: він не вимагає наявності зовнішньої мережевої інфраструктури у вигляді маршрутизатора або точки доступу, що робить систему повністю незалежною від навколишнього мережевого середовища. Оператор може підключитись до слейва з будь–якого пристрою з підтримкою Wi–Fi безпосередньо у польових умовах. Мережева адреса слейва у режимі точки доступу автоматично встановлюється на 192.168.4.1 і вбудований DHCP–сервер ESP32 автоматично призначає адреси підключеним клієнтам у підмережі 192.168.4.0/24.

На транспортному рівні використовується протокол TCP що забезпечує надійну доставку даних між браузером клієнта та HTTP–сервером на ESP32. На прикладному рівні функціонує HTTP версії 1.1 що обслуговується бібліотекою WebServer у складі Arduino–фреймворку для ESP32. Сервер зареєстрований на порту 80 що є стандартним портом для HTTP і дозволяє відкривати веб–інтерфейс просто введенням IP–адреси без зазначення номера порту. Реєстрація обробників виконується через server.on для двох маршрутів: кореневий шлях / що

обслуговується функцією handleRoot та шлях /data що обслуговується функцією handleData.

При першому зверненні браузера до кореневого шляху обробник handleRoot повертає статичну HTML–сторінку що повністю вбудована у прошивку у вигляді рядкової константи HTML у секції даних флеш–пам'яті мікроконтролера. Сторінка містить розмітку інтерфейсу із сіткою карток для відображення телеметричних даних, вбудовані CSS–стилі із темною кольоровою схемою та вбудований JavaScript–код що автоматично виконується після завантаження сторінки. Розміщення всього HTML, CSS та JavaScript в одному рядковому літералі у коді прошивки усуває необхідність у файловій системі SPIFFS або LittleFS і спрощує процес збірки та завантаження прошивки, хоча й обмежує гнучкість у модифікації інтерфейсу без перекомпіляції.

Після завантаження HTML–сторінки браузер починає автоматично надсилати GET–запити на маршрут /data з інтервалом одна секунда через механізм setInterval у JavaScript. Це є реалізацією техніки короткого опитування відомої як short polling, де клієнт з фіксованою periodичністю запитує сервер про оновлення даних. Обробник handleData при кожному такому запиті викликає getOverallState для отримання актуального стану системи, виконує applyState для синхронізації світлодіодної індикації, обчислює оцінку відстані через rssiToDistance та формує JSON–об'єкт що містить усі поточні телеметричні дані: кут сервоприводу у градусах, ширину ШІМ–імпульсу у мікросекундах, температуру та вологість від датчика AHT10 масштабовані з коефіцієнтом 10, рівень сигналу RSSI у дБм, відношення сигнал–шум SNR у дБ, оцінку відстані у метрах, рядковий ідентифікатор стану та текстову причину поточного стану. Формування JSON виконується через snprintf безпосередньо у символьний буфер розміром 260 байтів без використання бібліотеки ArduinoJson на стороні слейва, що є свідомим рішенням на користь економії оперативної пам'яті: фіксований формат відповіді дозволяє обійтись прямим форматуванням рядка замість побудови JSON–дерева у динамічній пам'яті.

Отримавши JSON–відповідь браузер через JavaScript розбирає її та оновлює вміст відповідних DOM–елементів сторінки без перезавантаження: поля із кутом, мікросекундами, температурою, вологістю, RSSI, SNR та відстанню оновлюються текстовим вмістом, кольоровий індикатор стану змінює клас CSS між значеннями GREEN, YELLOW та RED що через таблицю стилів відповідають зеленому, жовтому та червоному кольорам, а текстовий рядок причини стану відображає деталізований опис поточної ситуації. Інтервал опитування одна секунда узгоджений з інтервалом відправки телеметрії майстром також одна секунда, що означає що кожне оновлення сторінки теоретично відображає дані з останнього отриманого пакету за умови відсутності затримок у мережі Wi–Fi.

Вибір між технологією короткого опитування та більш сучасними альтернативами є принциповим питанням що потребує детального обґрунтування. Першою альтернативою є WebSocket що забезпечує повнодуплексний канал між клієнтом та сервером із мінімальними накладними витратами після встановлення з'єднання і теоретично дозволив би серверу самостійно надсилати оновлення клієнту одразу після отримання нового пакету від майстра без необхідності чекати наступного запиту від браузера. Однак реалізація WebSocket–сервера на ESP32 вимагає значно більших ресурсів оперативної пам'яті для підтримки стану з'єднання, буферизації фреймів та обробки протоколу рукостискання, а також суттєво ускладнює обробку аварійних ситуацій коли клієнт несподівано розриває з'єднання. Враховуючи що Core 1 слейва вже виконує задачу системного моніторингу та обслуговування Wi–Fi стека, додаткове навантаження від підтримки WebSocket–з'єднань могло б негативно вплинути на стабільність роботи. Другою альтернативою є Server–Sent Events що є однонаправленим варіантом постійного з'єднання де сервер може надсилати події клієнту у будь–який момент, однак ця технологія також вимагає підтримки тривалих TCP–з'єднань що є навантажувальним для вбудованого стека ESP32 при одночасній роботі Wi–Fi AP та HTTP–сервера.

Підхід із простим HTTP–опитуванням на противагу цим альтернативам споживає пам'ять лише під час обробки конкретного запиту після якої всі ресурси звільняються, кожен запит є незалежною транзакцією що не залишає ніякого стану на сервері, і відмова або перезавантаження клієнтського браузера не вимагає жодної обробки на стороні сервера. Для діагностичного інтерфейсу із даними що оновлюються з частотою одна секунда різниця у затримці між опитуванням та WebSocket є незначною і практично непомітною для оператора, тоді як виграш у простоті реалізації та стабільності роботи є суттєвим.

Обґрунтування вибору саме веб–інтерфейсу як засобу діагностики над альтернативними підходами є окремим важливим питанням. Нативний мобільний додаток вимагав би розробки та встановлення окремого програмного забезпечення на пристрій оператора що суттєво ускладнює розгортання системи та обмежує коло можливих пристроїв для моніторингу. Спеціалізований протокол на кшталт MQTT вимагає наявності брокера та окремого клієнтського застосунку і є надлишково складним для задачі відображення телеметрії від одного вузла. Виведення даних через серійний порт UART є зручним для розробника але вимагає фізичного підключення кабелю та встановлення термінальної програми що унеможливлює дистанційний моніторинг. Веб–інтерфейс у цьому контексті є оптимальним рішенням що поєднує універсальну доступність з будь–якого пристрою з браузером, відсутність необхідності у встановленні додаткового програмного забезпечення, наочне графічне представлення даних із кольоровою індикацією стану та можливість одночасного перегляду з кількох пристроїв що підключились до точки доступу слейва.

2.4.6 Перевірка вихідної потужності сигналу

Окремим етапом експериментальної перевірки розробленої системи стало вимірювання реальної вихідної потужності LoRa–модуля, оскільки паспортне значення 13 дБм, заявлене виробником, відображає лише номінальні умови і може відрізнятися від фактичного рівня випромінювання у складі зібраного пристрою. На реальну потужність впливає низка чинників: якість узгодження антенного

тракту, втрати у пігтейлі та роз'ємах, а також стабільність живлення модуля під час передачі. Тому було прийняте рішення перевірити вихідну потужність для підтвердження відповідності системи розрахунковому енергетичному бюджету каналу.

Вимірювання проводилось за допомогою професійної контрольно–вимірювальної апаратури компанії Rohde & Schwarz[16] на базі проведення переддипломної практики, що забезпечує високу точність аналізу спектра в радіочастотному діапазоні. Модуль було переведено у режим безперервної передачі телеметричних пакетів, після чого аналізатор спектра фіксував рівень сигналу у частотному діапазоні від 2.1 до 2.7 ГГц.

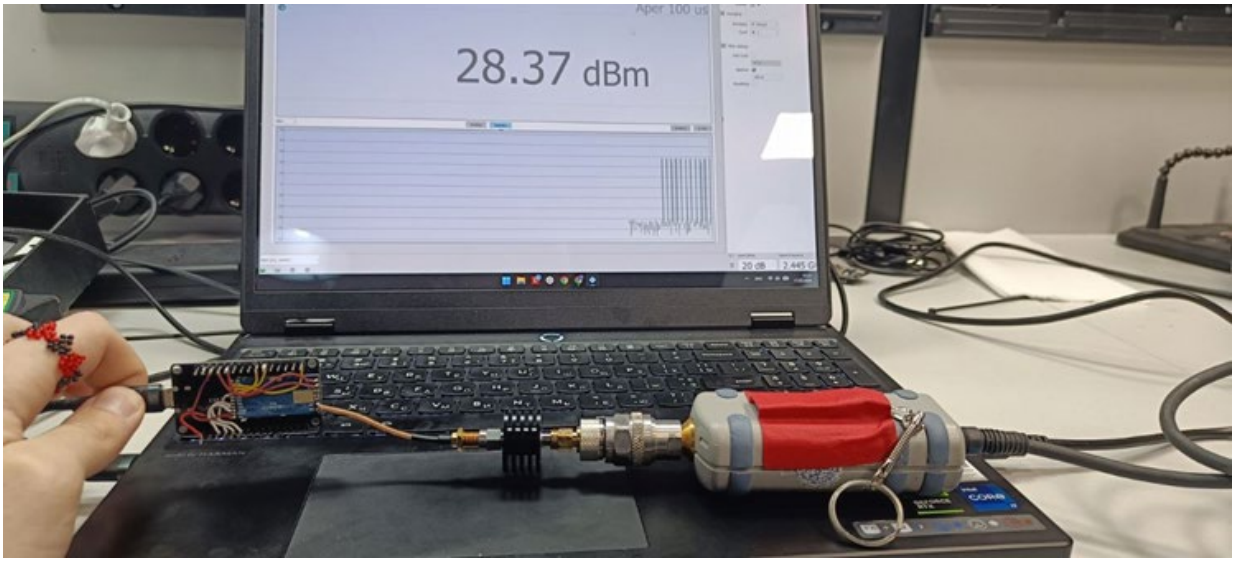

Рисунок 2.7 – Стенд для перевірки потужності сигналу на практиці

З отриманих даних можна сказати, що заводська вихідна потужність та фактичні потужність відрізнаються між собою. Через вище описані чинники результати можуть відрізнятися, але різниця в 3 дБм доволі серйозна, враховуючи що 13 дБм і так доволі не велика потужність і в випадку радіо це вже різниця в 2 рази. Так як 10 дБм це 10 міліват, а 13 дБм вже – 20. Припускаю, що мною був обраний або бракований чіп, або попала певна партія заміри якої виходять за заводські межі, але проводити розробку можна.

8.9(2100), 9.3(2150)	9.6(2200), 10.1(2250)	10.1(2300), 10.0(2350)	9.98(2400), 9.94(2450)	9.96(2500), 9.97(2550)	10.0(2600), 10.05(2650)	10.15(2700), 9.3(2750)

Рисунок 2.8 – Таблиця замірів

Висновок до розділу 2:

За результатами проєктування та розроблення системи обґрунтовано всі ключові архітектурні та технічні рішення, що забезпечують реалізацію відмовостійкого вбудованого контролера. Архітектура Master–Slave із двома фізично незалежними вузлами на базі ESP32 підтвердила свою доцільність як засіб природної ізоляції рівня керування від рівня моніторингу: жоден з вузлів не має критичної залежності від внутрішнього стану іншого, що усуває єдину точку відмови характерну для класичних однопроцесорних рішень.

Розроблена трирівнева система станів Green–Yellow–Red із гістерезисом переходів забезпечує градуйовану реакцію на діагностичні події та відсіює поодинокі випадкові збої без хибних спрацювань. Механізм автономного відновлення tryRecovery реалізує хірургічну переініціалізацію несправної підсистеми без перезавантаження контролера, що якісно відрізняє розроблене рішення від типового підходу із watchdog–перезапуском. Апаратний сторожовий таймер замикає ешелоновану систему захисту як останній рубіж проти непередбачуваних аварійних сценаріїв.

Обґрунтовано вибір бінарного формату телеметричного пакету із контролем цілісності за алгоритмом CRC–16, параметрів модуляції LoRa, схеми живлення з розділенням силової та цифрової гілок, інструментарію розробки на базі VS Code та PlatformIO, а також стратегії гілкування Git із виділеними fault–гілками для верифікації аварійних сценаріїв. Сукупність прийнятих рішень формує цілісну архітектуру, у якій відмовостійкість реалізована на всіх рівнях — від схемотехніки живлення до логіки прикладного програмного забезпечення.

3 РЕАЛІЗАЦІЯ ТА ТЕСТУВАННЯ

3.1 Середовище розробки та засоби керування версіями

Середовище розробки є фундаментальною складовою будь–якого вбудованого проєкту, оскільки від його можливостей безпосередньо залежить зручність написання коду, відтворюваність збірки між різними машинами та керованість зовнішніми залежностями.

3.1.1 Комбінація IDE VS Code та плагін Platform IO

Для розробки програмного забезпечення майстер– та слейв–вузлів обрано зв'язку Visual Studio Code із розширенням PlatformIO – одне з найзріліших рішень для професійної розробки під мікроконтролери. Сам по собі VS Code є універсальним редактором коду з системою розширень, підсвічуванням синтаксису, вбудованим терміналом та інтеграцією з Git, проте для розробки під ESP32 він виконує роль лише оболонки. Усю реальну роботу – компіляцію, завантаження прошивки, моніторинг послідовного порту та налагодження – забезпечує розширення PlatformIO, яке перетворює редактор на повноцінне середовище розробки вбудованих систем і підтримує сотні апаратних платформ, зокрема обидва вузли проєкту на базі ESP32.

Ключова перевага PlatformIO над Arduino IDE полягає в підході до керування проєктом та залежностями. В Arduino IDE бібліотеки встановлюються глобально, а конфігурація плати зберігається у налаштуваннях середовища, тому перенесення проєкту на іншу машину чи відтворення збірки через певний час ускладнюється через відмінності у версіях бібліотек. PlatformIO натомість використовує центральний файл platformio.ini у корені проєкту, що повністю описує ціль збірки, параметри завантаження та перелік залежностей із зафіксованими версіями. Для розроблюваного проєкту цей файл містить два середовища збірки: для майстер–вузла з бібліотеками роботи з LoRa–модулем, I2C–датчиком і сервоприводом та для слейв–вузла з бібліотеками прийому LoRa–пакетів, веб–сервера й світлодіодної індикації. Підключення бібліотеки зводиться до одного рядка у секції lib_deps, після чого PlatformIO автоматично завантажує її разом з усіма транзитивними

залежностями, зберігаючи їх у каталозі з прив’язкою до конкретного проєкту, що виключає конфлікти версій між проєктами.

Окремою перевагою є підтримка системи збірки CMake та компілятора Xtensa GCC у конфігурації для ESP32, що дозволяє налаштовувати рівень оптимізації, розмір стеку задач FreeRTOS та параметри розподілу пам’яті безпосередньо у конфігураційному файлі. Для відмовостійкої системи реального часу це принципово, оскільки некоректний розмір стеку задачі є поширеною причиною важко відтворюваних збоїв у RTOS–застосунках. Вбудований монітор послідовного порту з фільтрацією виводу суттєво спрощує налагодження там, де апаратний відладчик недоступний: повідомлення з різних задач FreeRTOS перемежовуються, і фільтрація за ключовими словами дозволяє зосередитися на поведінці конкретної підсистеми. Під час верифікації роботи FSM та механізму тайм–ауту на слейв–вузлі цей інструмент виявився практично незамінним, даючи змогу спостерігати за переходами між станами у реальному часі.

3.1.2 ДСКВ Git та його використання під час розробки

Система контролю версій давно вийшла за межі суто командної розробки і стала невід’ємним інструментом навіть для одноосібних проєктів. Для відмовостійких систем реального часу, де некоректна зміна логіки FSM чи параметрів планувальника може спричинити важко відтворювану аварійну поведінку, повна історія змін коду перетворюється з зручності на вимогу до процесу розробки. Git як децентралізована система зберігає історію кожної зміни у локальному репозиторії, не потребуючи мережевого з’єднання для щоденних операцій. Кожен коміт фіксує не лише стан файлів, а й метадані про автора, час та опис змін, що дозволяє у будь–який момент відновити стан коду, який відповідав певній поведінці системи – наприклад, повернутися до версії з коректним переходом FSM зі стану Yellow у Red, якщо подальші зміни порушили цю логіку.

Особливо цінною є можливість паралельної роботи у гілках. Основна гілка main завжди містить прошивку, що пройшла перевірку всіх підсистем, тоді як експериментальні зміни – налаштування порогів FSM чи модифікація структури

телеметричного пакету – вносяться в окрему гілку і зливаються лише після підтвердження коректності. Така дисципліна є прямим аналогом принципу відмовостійкості на рівні процесу розробки: подібно до того як апаратна система має безпечний стан, репозиторій завжди має стабільну точку відліку. Механізм тегування дозволяє маркувати окремі коміти як релізні версії прошивки та відтворювати точний бінарний образ у будь–який момент. У поєднанні з файлом конфігурації PlatformIO репозиторій утворює самодостатній пакет, що описує і програмну логіку, і середовище збірки: будь–хто, хто клонує його, отримує вихідний код, точні версії бібліотек та конфігурацію платформи.

Використання віддаленого репозиторію на GitHub додає рівень зовнішнього резервного копіювання, що захищає від втрати результатів у разі апаратного збою робочої машини. Оскільки розробка ведеться паралельно з написанням пояснювальної записки протягом кількох місяців, така страховка є практично значущою, а змістовні коментарі до комітів формують документований слід процесу розробки.

Організація робочого процесу у Git для вбудованого ПЗ відрізняється від веб–розробки, оскільки некоректна прошивка здатна вивести апаратуру у невизначений стан або пошкодити периферію. Тому основна гілка main містить виключно прошивку, що пройшла повний цикл перевірки: компіляцію без попереджень, коректну ініціалізацію периферії, підтверджену роботу FSM у всіх трьох станах та стабільну передачу телеметрії протягом кількох хвилин. Кожен коміт супроводжується описом, що чітко розрізняє тип зміни – виправлення помилки, нову функціональність чи зміну конфігурації. Для розробки нової функціональності використовується гілка develop, що відгалужується від main і слугує інтеграційним середовищем перед перенесенням змін у стабільну гілку.

Принципово важливим і специфічним для даного проєкту є використання окремої категорії гілок для розробки прошивки із задуманим відхиленням від нормальної поведінки системи. Ці гілки мають префікс fault та призначені для навмисного відтворення аварійних сценаріїв з метою верифікації механізмів

відмовостійкості. Ідея полягає у наступному: щоб переконатися що система коректно переходить у стан Red при недоступності I2C–шини, недостатньо просто відключити датчик фізично під час тестування, оскільки такий підхід не є відтворюваним і не документує поведінку системи. Натомість у гілці “fault / i2c – timeout – simulation” вноситься навмисна зміна у код даних отриманих по I2C шині, що змушує його завжди повертати помилку тайм–ауту незалежно від реального стану шини. Прошивка зібрана з цієї гілки за визначенням є непрацюючою з точки зору нормального функціонування системи, проте вона є цілком коректним інструментом для верифікації того, що FSM реагує на цей сценарій саме так, як передбачено архітектурним рішенням.

Окрім локального зберігання історії змін, Git надає можливість синхронізації репозиторію із віддаленим сервером, що у контексті розробки дипломного проєкту виконує насамперед функцію зовнішнього резервного копіювання. Для розроблюваної системи використовується платформа GitHub, на якій розміщено приватний репозиторій із повною історією розробки обох вузлів системи. Практична цінність такого підходу виявляється у захисті від апаратних збоїв робочої машини, випадкового видалення файлів або пошкодження локального репозиторію. Оскільки розробка ведеться паралельно із написанням пояснювальної записки протягом кількох місяців, втрата навіть тижневого прогресу була б суттєвою. Операція push після кожної завершеної сесії розробки забезпечує що актуальний стан коду завжди присутній у двох фізично розділених місцях незалежно від стану локального середовища.

Додатковою перевагою є можливість продовжувати розробку з будь–якого пристрою через просту операцію clone або pull, що знімає прив'язку до конкретної робочої машини. Інтерфейс GitHub також надає зручний веб–перегляд історії комітів, порівняння гілок та різниці між версіями файлів, що є зручним доповненням до локальних інструментів VS Code при аналізі того, коли і чому була внесена конкретна зміна у логіку FSM або параметри LoRa–каналу.

3.2 Практична реалізація апаратної частини

В практичній частині слід відмітити час затрачений на практиці на створення фізичних модулів. Завдяки місцю практики з підходящим мені профілем під тему диплому я мав змогу отримати доступ до необхідних ресурсів, інструментів та знань, які дуже знадобилися мені для формування диплому.

3.2.1 Вигляд модулів master та slave

Змонтував модуль master на основі ESP32 DOIT DEV KIT V1 та LoRa SX1280. На вхідні піни Vin та GND припаяв живлення з DC–DC перетворювача через діод. Припаяв до DC–DC перетворювача на вхід відповідний конектор яким можна було б під’єднатися до 2S 1P батареї. На рисунку 3.1 можна ознайомитися з виглядом модулів.

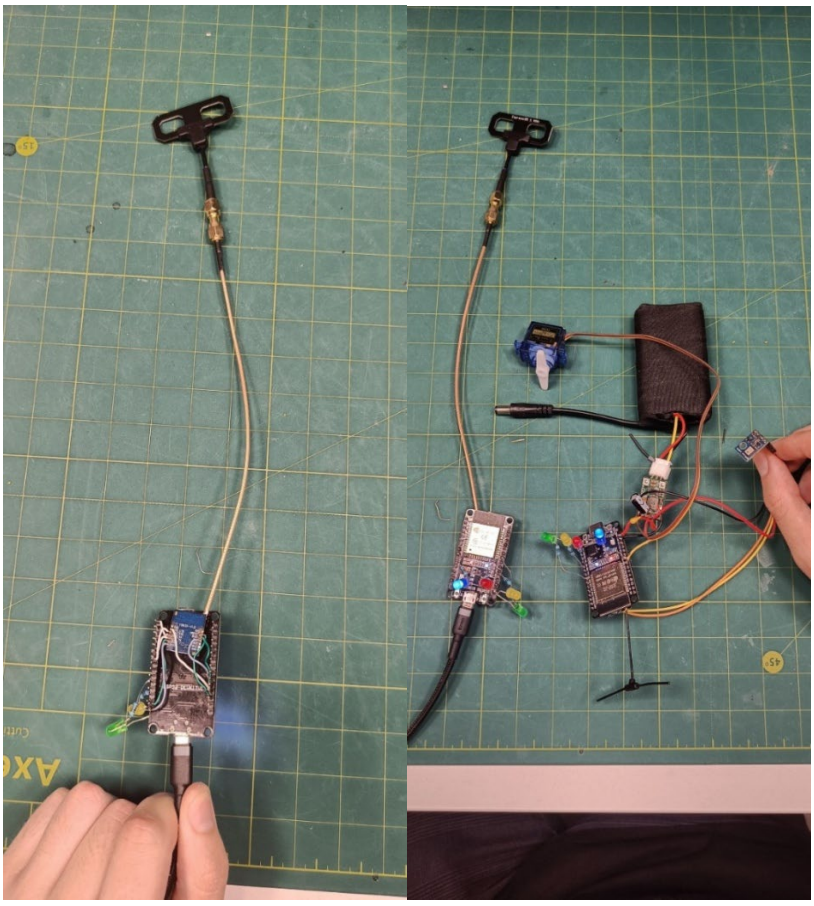

Рисунок 3.1 – Вигляд модулів

Далі по периферії був сервопривід живлення якого береться з перетворювача, щоб обійти підключення живлення сервопривода через ESP, щоб при пікових споживаннях при роботі сервопривода 3.3 вольт лінія ESP не просідала, через це

живлення сервопривода окреме. Перед останній був AHT10 датчик який живлення теж брав з перетворювача і підключався до ESP тільки SCK та SDA каналами I^2C шини. Заключними були світлодіоди які будуть звітувати користувачу про стан справ з master, або стану зв’язку з ним.Модуль slave складався за цим же принципом, але живлення розраховується брати з USB, оскільки slave модуль сам по собі менш енергоспоживаючий і має бути портативним.

При пайці елементів між собою приділялася увага до її якості, оскільки потрібно щоб елементи трималися міцно, провода не відривалися і загалом збірка була надійною.

3.2.2 Схеми розпайки модуля та розміщення антени

Питання розміщення антени є одним із найбільш недооцінених аспектів при розробці систем із радіоканалом, оскільки навіть найкращий трансивер із оптимально підібраними параметрами модуляції не здатен компенсувати втрати що виникають через некоректне розташування або орієнтацію антени. У розроблюваній системі це питання вирішене через використання пігтейлю – короткого гнучкого коаксіального кабелю що з'єднує роз'єм IPEX на модулі Ebyte із зовнішньою антеною, забезпечуючи механічну незалежність між антеною та електронікою. На рисунках 3.2 та 3.3 показано схеми розпайки модуля Master та трансивера LoRa.

Використання пігтейлю є практично виправданим рішенням одразу з кількох причин. По–перше електронна частина вузла може розміщуватись у корпусі або на шасі роботизованої системи у довільній позиції не зважаючи на вимоги до орієнтації антени, тоді як антена виводиться у заздалегідь обране оптимальне місце через гнучкий кабель. По–друге це дозволяє у будь–який момент змінити положення антени без переробки монтажу електроніки, що є суттєвою перевагою на етапі налагодження та оптимізації зв'язку між вузлами. Ключовим параметром що визначає якість зв'язку між майстром та слейвом у даній конфігурації є узгодження поляризації антен. Діпольна антена, що типово використовується із модулями Ebyte має лінійну вертикальну, або горизонтальну поляризацію, і

максимальна потужність передається та приймається коли обидві антени орієнтовані паралельно одна одній у вертикальному чи горизонтальному положенні відповідно.[24]

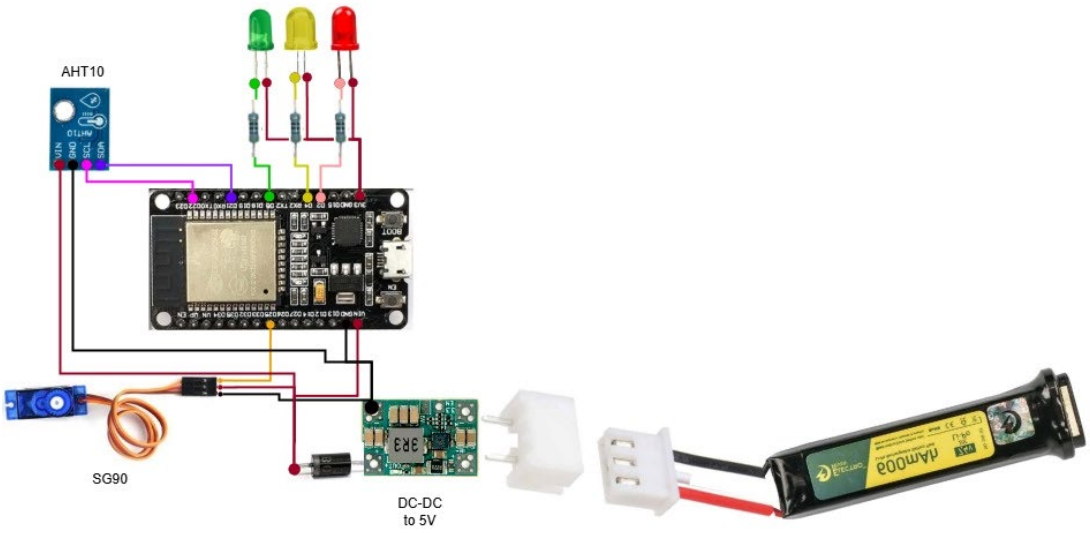

Рисунок 3.2 – Схема розпайки модуля

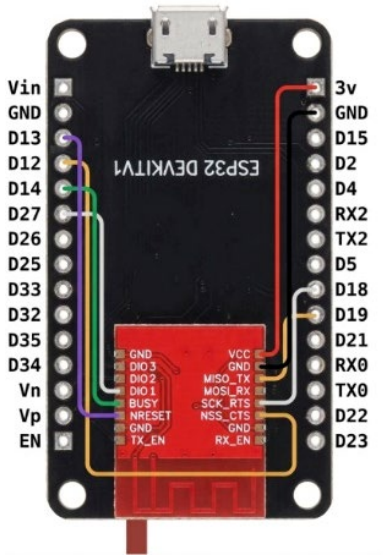

Рисунок 3.3 – Схема розпайки LoRa модуля для приєднання по SPI

Невідповідність поляризації на 90 градусів тобто одна антена вертикальна, а інша горизонтальна теоретично призводить до втрат близько 20 dB що є катастрофічним для бюджету каналу і може повністю унеможливити зв'язок навіть на мінімальній відстані. Саме для вирішення цієї проблеми і передбачено пігтейль: він дозволяє виставити антену у потрібну орієнтацію незалежно від того як розташована плата з модулем всередині корпусу системи. На практиці обидві

антени майстра та слейва орієнтуються вертикально у паралельних площинах що є стандартною рекомендацією для штирових антен і забезпечує максимальний рівень прийнятого сигналу при заданій відстані та потужності передавача 13 дБм модуля Ebyte.

3.2.3 Реалізація програмних модулів та веб інтерфейсу

Процедура ініціалізації майстер–вузла реалізована у функції masterSetup і є послідовністю кроків де кожен наступний виконується лише за умови успішного завершення попереднього або з явною фіксацією результату у відповідному прапорі стану. Першим кроком є налаштування апаратного сторожового таймера із порогом 15 секунд та підпискою поточної задачі, що гарантує перезавантаження системи навіть якщо подальша ініціалізація зависне на очікуванні відповіді від периферії.

Перевірка наявності датчика AHT10 реалізована через функцію aht10Init що починається з найпростішої можливої діагностики I2C–шини: спроби встановити з'єднання з пристроєм за адресою 0x38 та отримати підтвердження. Ненульове повернене значення Wire.endTransmission однозначно свідчить про фізичну відсутність датчика або несправність шини і функція повертає false без подальших спроб. При успішному виявленні виконується повна процедура ініціалізації із програмним скиданням та калібруванням датчика відповідно до документації AHT10. Результат зберігається у прапорі ahtOk і виводиться у серійний порт як [AHT10] Ready або [AHT10] Not found.

Ініціалізація LoRa–модуля є більш критичною: при її невдачі masterSetup викликає applyState(RED) і негайно завершується без активації сервоприводу, оскільки система без радіоканалу не здатна виконувати свою основну функцію. Перевірка наявності SX1280 відбувається через LT.begin що виконує апаратне скидання модуля та перевіряє коректність відповіді на SPI–запити – єдиний надійний спосіб підтвердити справність трансивера, оскільки SPI на відміну від I2C не має вбудованого механізму виявлення пристроїв. При успіху налаштовуються параметри модуляції SF7, BW800, CR 4/5 що мають бути ідентичними на обох вузлах через спільний config.h. Сервопривід підключається останнім і переводиться

у початкову позицію 0 градусів, після чого система встановлює стан YELLOW при наявному датчику або RED при його відсутності, відображаючи філософію що GREEN є результатом підтвердженої роботи, а не просто успішного завантаження.

Веб–сервер слейв–вузла є кінцевою точкою взаємодії між системою та оператором і надає зведену діагностичну картину стану майстра у зручному для сприйняття форматі без необхідності підключення до серійного порту або використання спеціалізованих інструментів. Всі дані що відображаються у веб–інтерфейсі є актуальними на момент останнього отриманого телеметричного пакету і оновлюються автоматично кожну секунду через механізм JavaScript–опитування ендпоінту /data. На рисунку 3.4 показано вигляд веб–інтерфейсу.

Рисунок 3.4 – Вигляд веб–інтерфейсу

Інтерфейс організований у вигляді сітки карток де кожна картка відображає окремий параметр системи. Перша група даних стосується стану сервоприводу: відображається поточний кут у градусах що може набувати значень 0 або 45 відповідно до поточної фази циклу руху, а також ширина ШІМ–імпульсу у мікросекундах що є більш точним низькорівневим показником реального сигналу керування. Наявність обох параметрів дозволяє верифікувати коректність роботи бібліотеки ESP32Servo: кут є інтерпретованим значенням а мікросекунди є сирим

апаратним параметром і їх відповідність підтверджує правильність калібровки сервоприводу.

Друга група даних містить показники датчика AHT10: температура у градусах Цельсія та відносна вологість у відсотках, обидва параметри передаються у пакеті масштабованими з коефіцієнтом 10 як цілі числа і відновлюються до дійсних значень на стороні слейва діленням на 10 перед відображенням. При недоступності датчика обидва поля відображають нульові значення що є індикатором проблеми на стороні майстра навіть без аналізу поля стану.

Третя група містить параметри радіоканалу: рівень сигналу RSSI у дБм та відношення сигнал–шум SNR у дБ що зчитуються безпосередньо із регістрів трансивера SX1280 після кожного успішного прийому через LT.readPacketRSSI() та LT.readPacketSNR().

Четверта група складається з показу дистанції між модулями яка обраховується внутрішнім заміром часу пересилання пакету та отримання відповіді – RTToF. Працює це так, що при надсилання пакета від одного лора модуля до іншого стартує внутрішній таймер який зупинить рахувати час після отримання відповіді, знаючи приблизний час обробки макету модулем який має прийняти пакет і віддати відповідь. Тут все зводиться до звичайної математики, а саме до трикутника: швидкість, відстань та час. час визначається модулем на основі вимірювання тривалості прольоту пакету; швидкість поширення електромагнітного випромінювання є фізичною константою, а саме швидкість розсіяння електромагнітного випромінювання, вона ж є швидкість світла $3 \cdot 10^8$ м/с. Маючи такий точний спосіб вирахувати відстань на нього можна доволі спокійно покладатися. Все ще потрібно враховувати особливості кругової та лінійної поляризації антен, перевідбиття сигналів можуть розцінитися як необхідний пакет та внести дані які вже є хибні, але це вирішується в основному виставленням вертикальної лінійної поляризації, або ж програмними фільтрами які згладжують ці піки, або використання кругової поляризації, яка, підтверджено компанією Semtech, знатно стійкіше сприймає перевідбиті сигнали.

Найважливішим елементом інтерфейсу є індикатор стану що складається із кольорового кружка та текстового опису причини. Кружок змінює колір між зеленим, жовтим та червоним відповідно до результату функції getOverallState що комбінує стан каналу зв'язку та код стану отриманий від майстра. Текстове поле поряд із кружком відображає конкретну причину поточного стану що береться або з таблиці MASTER_MAP при проблемах на стороні майстра, наприклад "Sensor not found", "Sensor failed", "TX failed", "Servo detached", або формується безпосередньо на слейві при проблемах із каналом: "Link degraded" при інтервалі між пакетами від трьох до восьми секунд та "Link lost" при перевищенні восьми секунд. Саме це текстове поле виконує функцію журналу помилок у спрощеному вигляді: воно не зберігає історію подій але завжди відображає актуальну причину відхилення від норми що є достатнім для оперативної діагностики у більшості практичних сценаріїв. Повноцінний журнал подій із хронологією доступний через серійний порт де обидва вузли виводять детальні повідомлення із префіксами [AHT10], [LORA], [SERVO], [STATE] та [RECOVERY] що дозволяють відстежити повну послідовність подій від ініціалізації до поточного моменту.

Прийом пакетів відбувається у задачі loraRxTask що виконується на Core 0 у нескінченному циклі із блокуючим викликом LT.receive та тайм–аутом п'ять секунд. Після повернення з LT.receive незалежно від результату негайно викликається esp_task_wdt_reset що підтверджує нормальну роботу задачі сторожовому таймеру. Отриманий буфер перевіряється за двома критеріями: довжина має бути не меншою за розмір структури ServoPacket, і поля type та magic мають відповідати очікуваним константам PKT_SERVO та PKT_MAGIC. Ця двоетапна валідація захищає від двох різних видів некоректних даних: перша перевірка відсіює обрізані або порожні пакети що могли виникнути через помилки SPI або часткову втрату даних у каналі, а друга відсіює пакети від сторонніх пристроїв що можуть працювати на тій самій частоті з тими самими параметрами модуляції. Лише при проходженні обох перевірок дані розпаковуються у відповідні volatile–поля і що принципово важливо оновлюється мітка часу lastRxTime = millis(). Якщо пакет не пройшов валідацію

lastRxTime не оновлюється це означає що некоректний пакет з точки зору логіки тайм–ауту відсутній взагалі.

Логіка виявлення тайм–ауту та формування результуючого стану реалізована у функції getOverallState що викликається у кожній ітерації slaveLoop на Core 1. Функція обчислює вік останнього валідного пакету як різницю між поточним millis() та lastRxTime, із спеціальною обробкою початкового стану коли жодного пакету ще не отримано: у цьому випадку замість lastRxTime використовується startTime що дорівнює значенню millis() на момент завантаження слейва. Завдяки цьому система не переходить у RED одразу після ввімкнення а надає майстру розумний час для встановлення зв'язку.

Класифікація стану каналу за віком пакету реалізована через три порогових рівні. При значенні age меншому за три секунди канал вважається справним і linkState встановлюється у GREEN що відповідає нормальній роботі з інтервалом пакетів одна секунда та невеликим запасом на природні варіації затримки LoRa. При значенні від трьох до восьми секунд що відповідає пропуску від двох до семи пакетів підряд linkState встановлюється у YELLOW із повідомленням "Link degraded" що сигналізує про деградацію каналу без повної його втрати. При перевищенні восьми секунд linkState переходить у RED із повідомленням "Link lost" що є однозначним індикатором стійкої відсутності зв'язку.

Результуючий стан для індикації формується через об'єднання linkState та masterState що отримується з таблиці MASTER_MAP за кодом lastMasterState із останнього валідного пакету. Принцип об'єднання є найгіршим результатом: RED від будь–якого джерела дає результуючий RED, YELLOW від будь–якого джерела при відсутності RED дає результуючий YELLOW, і лише при GREEN від обох джерел результат є GREEN. При рівності тяжкості станів пріоритет текстового опису причини надається стану каналу над станом майстра оскільки при проблемному каналі будь–яка інформація від майстра є потенційно застарілою і менш достовірною. Результуючий стан передається у applyState що вмикає відповідний світлодіод та оновлює JSON–відповідь веб–сервера включаючи

текстову причину стану через поле reason. В таблиці 3.1 показано значення станів, їх показ сигналами діодів та помилки які вони означають.

Таблиця 3.1 – Таблиця стані контролера

Стан	Світлодіод	Причина
GREEN	Зелений	MASTER_OK – всі підсистеми справні, канал стабільний
YELLOW	Жовтий	MASTER_SENSOR_WARN – датчик AHT10 нестабільний (2–4 збої)
YELLOW	Жовтий	MASTER_TX_WARN – нестабільна передача LoRa (2–4 збої підряд)
YELLOW	Жовтий	Link degraded – пакет не надходив 3–8 секунд
RED	Червоний	MASTER_NO_SENSOR – датчик AHT10 не виявлено при ініціалізації
RED	Червоний	MASTER_NO_SERVO – сервопривід не підключено
RED	Червоний	MASTER_SENSOR_FAIL – датчик AHT10 відмовив (5+ збоїв підряд)
RED	Червоний	MASTER_TX_FAIL – передача LoRa відмовила (5+ збоїв підряд)
RED	Червоний	Link lost – пакет не надходив понад 8 секунд

Одночасно може бути активним лише один світлодіод. Якщо кілька умов RED або YELLOW активні одночасно − відображається найгірший стан, а пріоритет текстового опису причини у веб–інтерфейсі надається проблемі каналу над проблемою майстра.

3.3 Тестування системи в режимі імітації відмов

Тестування проводилося з двома цілями: перевірити як себе веде система не в лабораторних умовах та на скільки на неї можна покладатися відносно точності вимірювання відстані.

3.3.1 Тест 1: Обрив лінії I2C

Після завантаження прошивок на обидва вузли та подачі живлення перший крок перевірки полягав у підключенні до точки доступу слейва із назначеним SSID та відкритті веб–інтерфейсу за адресою 192.168.4.1 у браузері. На момент першого підключення інтерфейс відображав жовтий індикатор стану із повідомленням що відповідає початковому YELLOW–стану системи до отримання підтверджених даних, і жовтий світлодіод на платі слейва світився відповідно.

Встановивши стабільний зв'язок між вузлами веб–інтерфейс почав оновлюватись кожну секунду: у полях температури та вологості з'явились реальні

показники датчика AHT10, поля кута сервоприводу циклічно змінювались між 0 та 45 градусами відповідно до трисекундного інтервалу руху, а поля RSSI та SNR відображали актуальний стан радіоканалу. Індикатор стану перейшов у зелений і зелений світлодіод на платі слейва підтвердив це фізично. На рисунках 3.5 та 3.6 показано вигляд штатної роботи модулів та вигляд веб–інтерфейсу

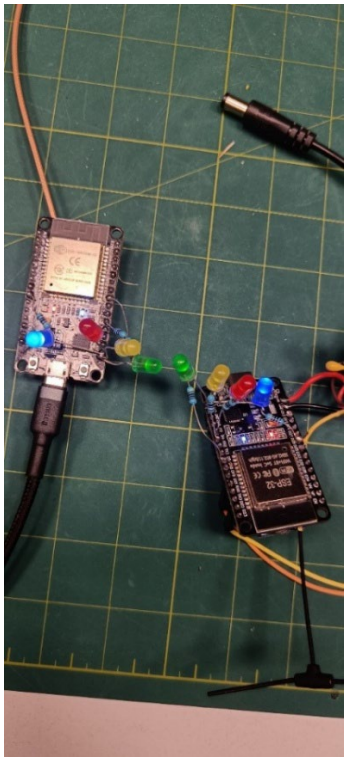

Рисунок 3.5 – Модуль з підключеною i2c шиною в вигляді жовтого та оранжевого проводів

Рисунок 3.6 – Вигляд інтерфейсу з станом «Зеленим» та описом

Для перевірки механізму тайм–ауту живлення майстра було вимкнено без попередження. Протягом перших трьох секунд інтерфейс продовжував відображати останні отримані дані із жовтим індикатором та повідомленням "Link degraded", а жовтий світлодіод замінив зелений. Після восьми секунд без пакетів індикатор змінився на червоний із повідомленням "Link lost" і червоний світлодіод підтвердив перехід у аварійний стан. При повторному вмиканні майстра система автоматично відновила зв'язок і повернулась до зеленого стану без будь–якого втручання оператора. На рисунках 3.7 та 3.8 показано стадії деградації системи.

Рисунок 3.7 – Вигляд інтерфейсу зі станом «Жовтий» та описом перебоїв у зв’язку

Рисунок 3.8 – Вигляд інтерфейсу зі станом «Червоний» та описом відсутності зв’язку

Додатково була перевірена реакція на відключення датчика AHT10 під час роботи: після двох секунд із нульовими показниками температури та вологості веб–інтерфейс відобразив жовтий стан із повідомленням "Sensor intermittent", а після п'яти секунд стан змінився на червоний із повідомленням "Sensor failed". Після повторного підключення датчика механізм відновлення tryRecovery автоматично переініціалізував AHT10 протягом наступних десяти секунд і система повернулась до зеленого стану. На рисунках 3.9-3.12 показано вигляд інтерфейсу та модуля коли статус змінюється на жовтий «перервано з’єднання з сенсором» та його втрата.

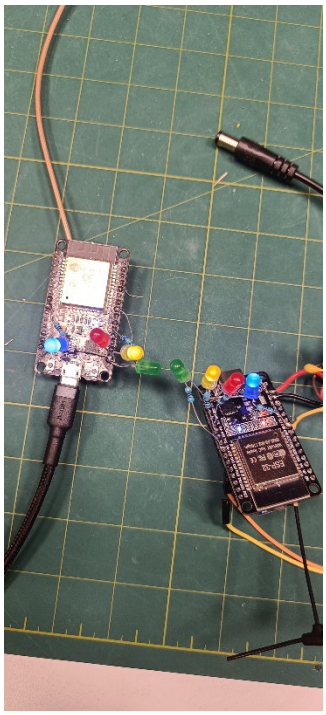

Рисунок 3.9 – Модуль з від’єднаним провідком SCK

Рисунок 3.10 – Вигляд інтерфейсу з станом «Жовтий» та описом про сенсор

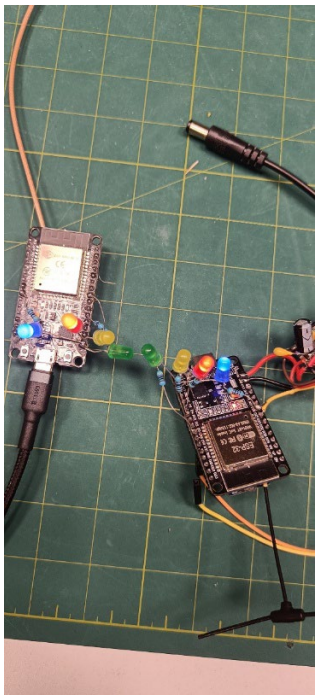

Рисунок 3.11 – Модуль сигналізуючий про відсутність даних з сенсора

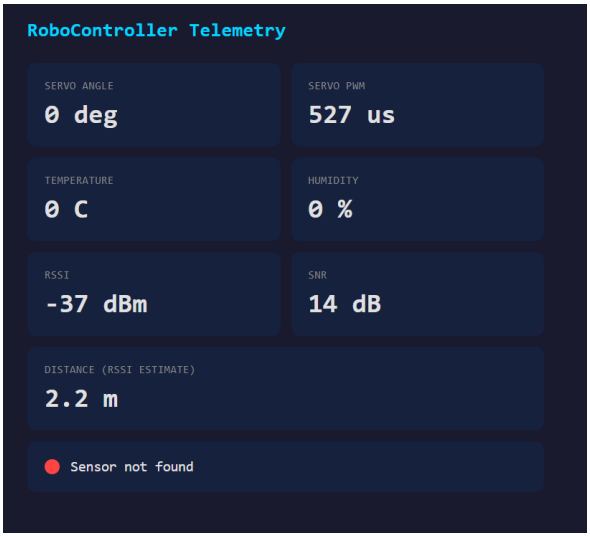

Рисунок 3.12 – Вигляд інтерфейсу з станом «Червоний» та описом про сенсор

Після приєднання SCK до ESP моментально відновлюється зв’язок з датчиком температури про, що сигналізують діоди та опис на веб–інтерфейсі описані вище.

3.3.2 Тест 2: Критичний поріг температури

У розробленій системі датчик AHT10 підключений до майстер–вузла через шину I2C і виконує безперервне вимірювання температури та відносної вологості навколишнього середовища. Отримані значення масштабуються з коефіцієнтом 10 та записуються у поля tempC10 і humPct10 структури ServoPacket що передається

слейву через LoRa–канал із інтервалом одна секунда. Таким чином показники температури є постійно присутніми у телеметричному потоці і відображаються у веб–інтерфейсі слейва у реальному часі.

Програмна логіка майстра передбачає два порогових рівні температури що визначають поведінку скінченного автомата станів. На рисунках 3.13 та 3.14 показано стан коли модуль попереджає про високу температуру.

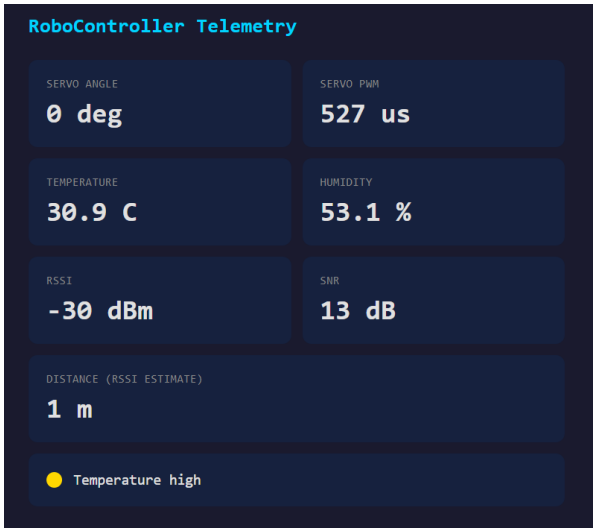

Рисунок 3.13 – Вигляд інтерфейсу з станом «Жовтий» та описом попередження підвищеної температури

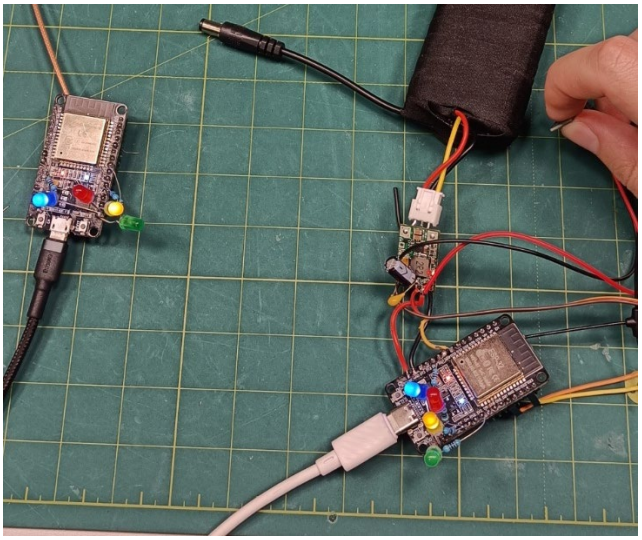

Рисунок 3.14 – Модуль сигналізуючий про попередження підвищеної температури

Попереджувальний поріг встановлений на рівні 30 градусів Цельсія і відповідає ситуації підвищеного теплового навантаження на електронні

компоненти, яке ще не є аварійним, але потребує уваги. При перевищенні цього порогу функція updateState встановлює код MASTER_SENSOR_WARN і FSM переходить у стан YELLOW.

Критичний поріг встановлений на рівні 32 градусів Цельсія і відповідає умовам за яких подальша нормальна робота є небезпечною для напівпровідникових компонентів системи. При його перевищенні встановлюється код MASTER_SENSOR_FAIL і система переходить у стан RED. На рисунках 3.13 та 3.14 показано стан коли модуль попереджає про критичну температуру.

Рисунок 3.15 – Вигляд інтерфейсу з станом «Червоним» та описом критичної температури

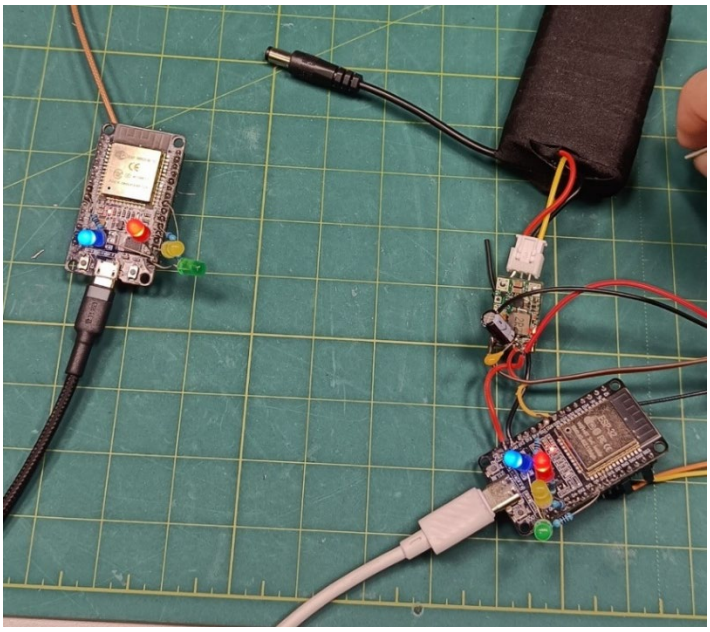

Рисунок 3.16 – Модуль сигналізуючий про попередження критичної температури

Принципова відмінність температурного тригера від тригера за недоступністю датчика полягає у миттєвості реакції. Якщо збій зчитування по I2C накопичується через лічильник sensorFails із порогами два та п'ять послідовних помилок, то перевищення критичного температурного порогу виявляється за одним вимірюванням і одразу призводить до переходу у стан RED без очікування наступних циклів. Це архітектурно обґрунтоване рішення: одиничний збій зчитування може бути наслідком випадкової перешкоди на шині, тоді як реальний перегрів є стійким фізичним процесом який не вирішиться між двома секундними циклами і вимагає негайної реакції.

У стані RED спричиненому критичною температурою система виконує захисну послідовність дій. Сервопривід SG90 переводиться у безпечне положення спокою на позиції 0 градусів і його циклічний рух призупиняється, оскільки активне навантаження на обмотки серводвигуна є додатковим джерелом тепловиділення що погіршує ситуацію. Телеметричний пакет із відповідним кодом стану відправляється слейву позачергово поза плановим інтервалом що є пріоритетним відхиленням від штатного розкладу передачі. Слейв отримує пакет, розпізнає RED–стан через таблицю MASTER_MAP, вмикає червоний світлодіод та відображає у веб–інтерфейсі причину із текстовим описом "Sensor failed" разом із поточними числовими значеннями температури що дозволяє оператору одразу оцінити масштаб відхилення від норми.

Механізм відновлення після температурного RED–стану реалізований із гістерезисом щоб уникнути небажаного ефекту осциляції стану на межі порогового значення. Система не повертається до стану YELLOW одразу при падінні температури нижче критичного порогу, натомість вимагається щоб значення опустилось щонайменше на п'ять градусів нижче порогу і утримувалось там протягом кількох послідовних вимірювань. Лише за цієї умови функція updateState вважає теплове навантаження нормалізованим і дозволяє FSM перейти у кращий стан. Механізм tryRecovery що спрацьовує кожні десять секунд у стані RED контролює цю умову і при її виконанні скидає відповідні прапори що відкриває

шлях до автоматичного відновлення нормальної роботи без перезавантаження мікроконтролера.

3.3.3 Тест 3: Перевірка зв’язку на супротив перешкодам, чіткість дистанції

– Тест зв’язку через перешкоди 1:

Тест проводився у доволі складних умовах – через дерева, між будівлями та з різницею висот. За таких умов розраховувати на точні дані про дальність не було сенсу, оскільки потужності чипа недостатньо, щоб пробитися крізь перешкоди для точних замірів. Водночас для передачі даних модуль цілком справлявся: іноді спостерігалася нестабільність зв’язку на рівні –100…–109 dBm, але дані оновлювалися стабільно. Дальність у цьому випадку становила 118 метрів. На рисунку 3.17 показано умови вимірювання та реальна позиція на карті модулів першого тесту з перешкодами.

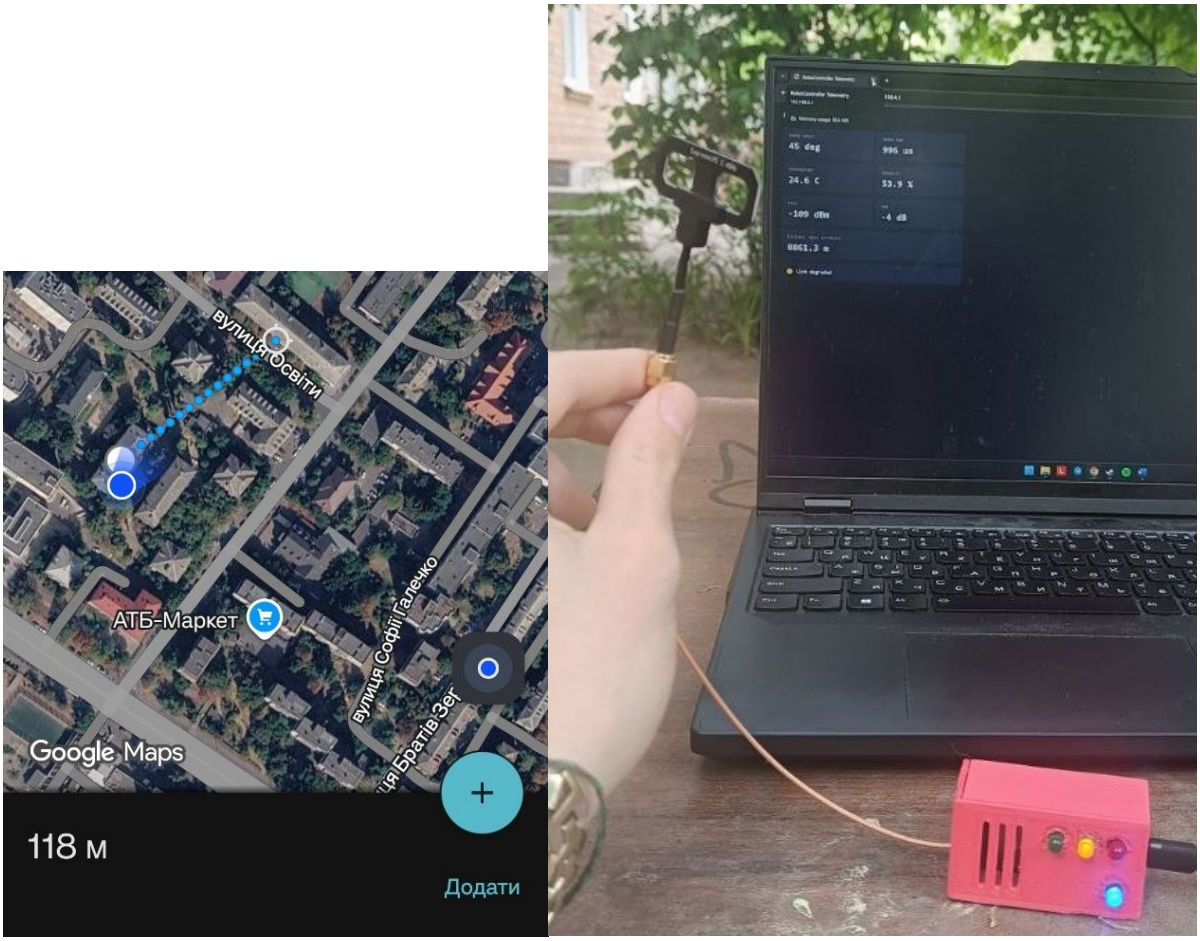

Рисунок 3.17 – Умови проведення тесту

– Тест зв’язку через перешкоди 2:

Для перевірки працездатності модулів та можливого покращення замірів було обрано ближчу дистанцію. Зв’язок та дані залишалися стабільними, проте через нестачу потужності модуля і відсутність прямої видимості (невідповідність радіогоризонту) стверджувати щось певне про правдивість даних дистанції, на жаль, не можна. На рисунку 3.18 показано умови вимірювання та реальна позиція на карті модулів другого тесту з перешкодами.

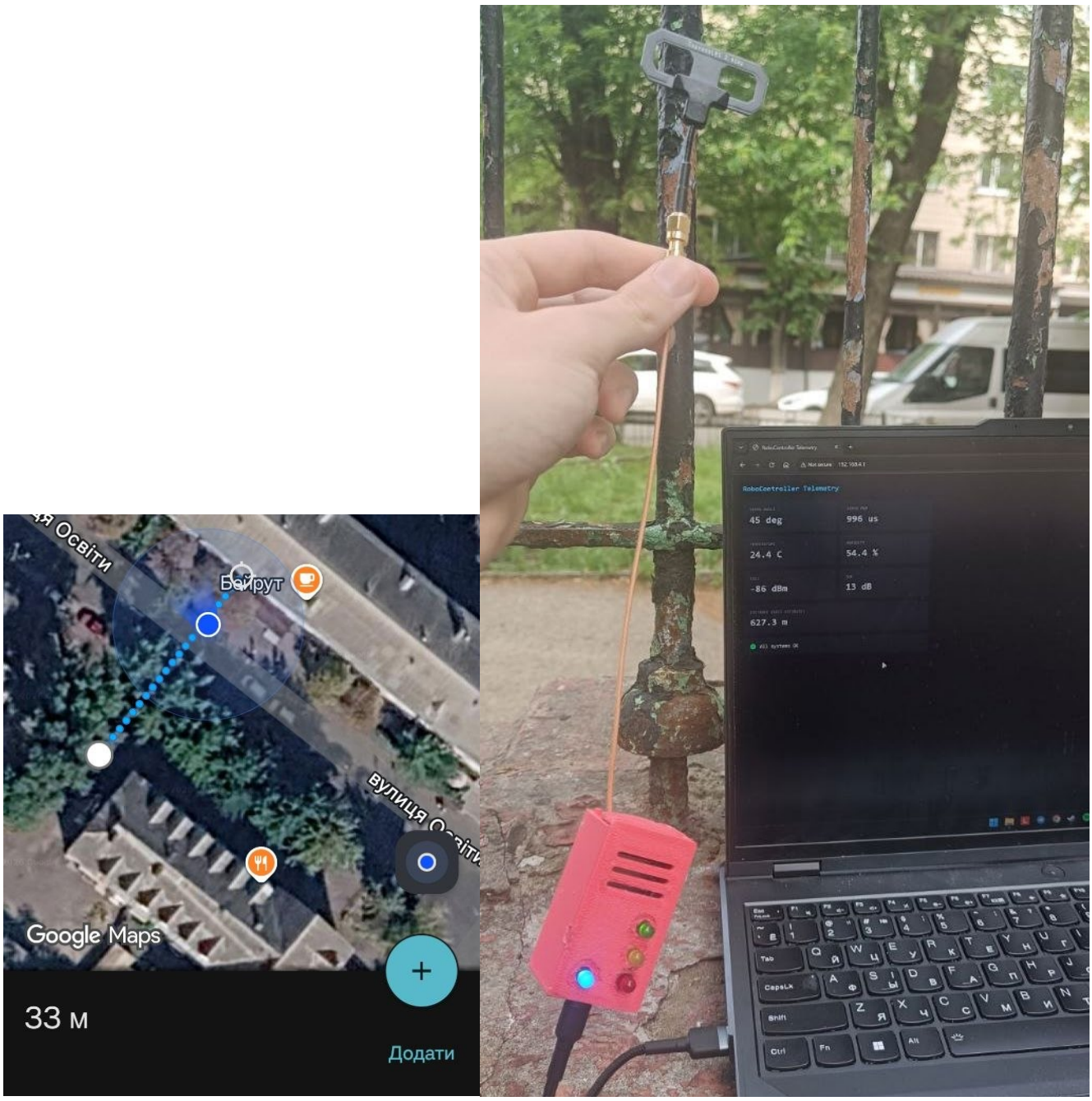

Рисунок 3.18 – Умови проведення тесту

– Тест на чіткість дистанції 1:

Через перешкоди у вигляді людей на шляху радіосигналу та невиконану калібровку під конкретну антену дані помітно різняться. За рекомендацією Semtech її слід проводити для кожної використовуваної антени. Похибка приблизно 50 метрів, наглядно видно як не вистачає потужності сигналу від передавача. З підсилювачем, або модулем більшої потужності можна отримати доволі чітку дистанції навіть на відстанях в десятки кілометрів. На рисунку 3.19 показано умови вимірювання та реальна позиція на карті модулів першого тесту на чіткість дистанції.

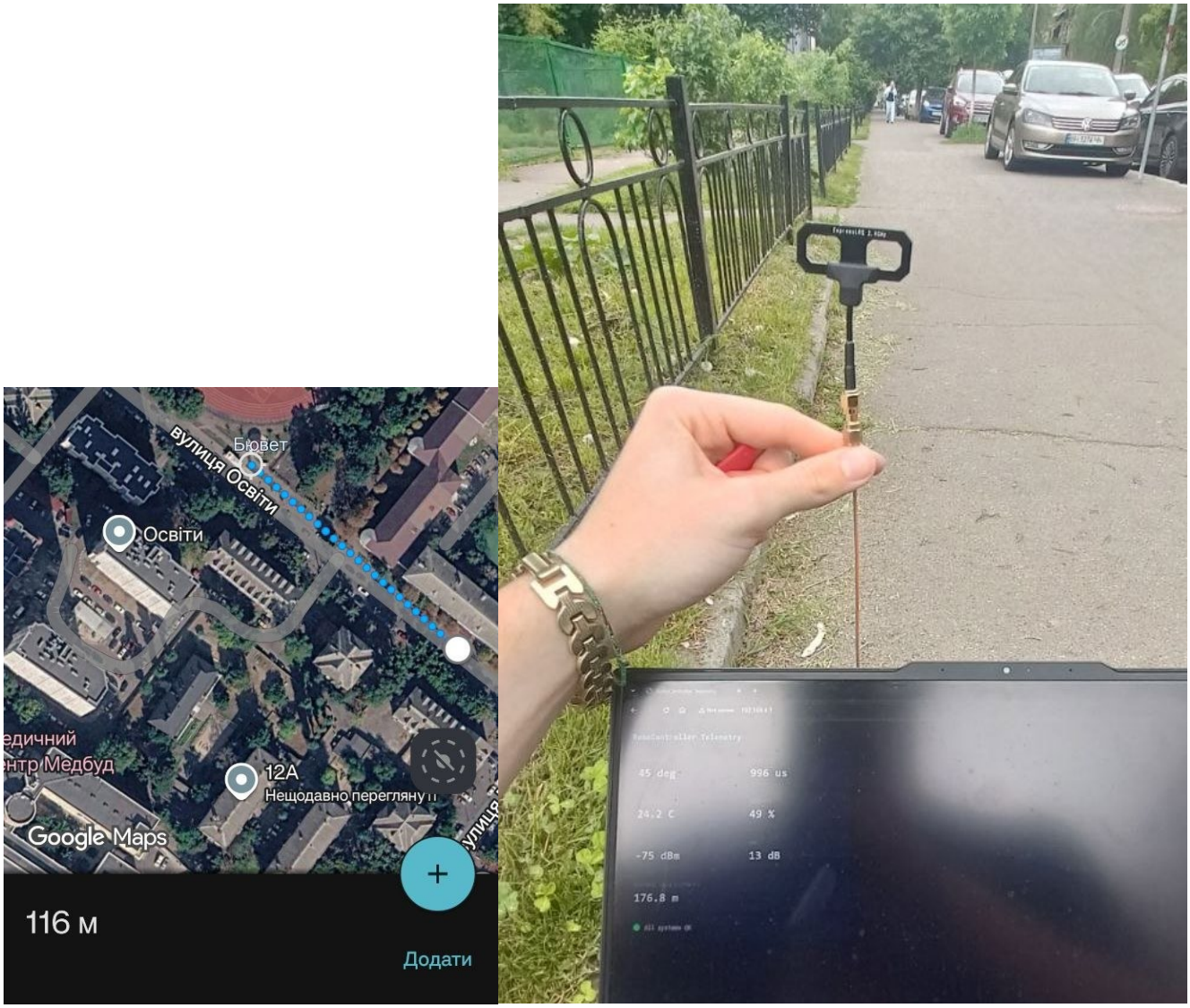

Рисунок 3.19 – Умови проведення тесту

– Тест на чіткість дистанції 2:

На дистанції 40 метрів цей модуль дозволяє отримувати доволі точні дані про відстань. Значення дещо стрибають через відсутність фільтрів, які б згладжували різкі скачки виміряної дистанції, проте на модулі потужністю 13 dBm у радіусі близько 50 метрів на дані дистанції можна покладатися. На рисунку 3.20 показано умови вимірювання та реальна позиція на карті модулів другого тесту з чіткістю дистанції.

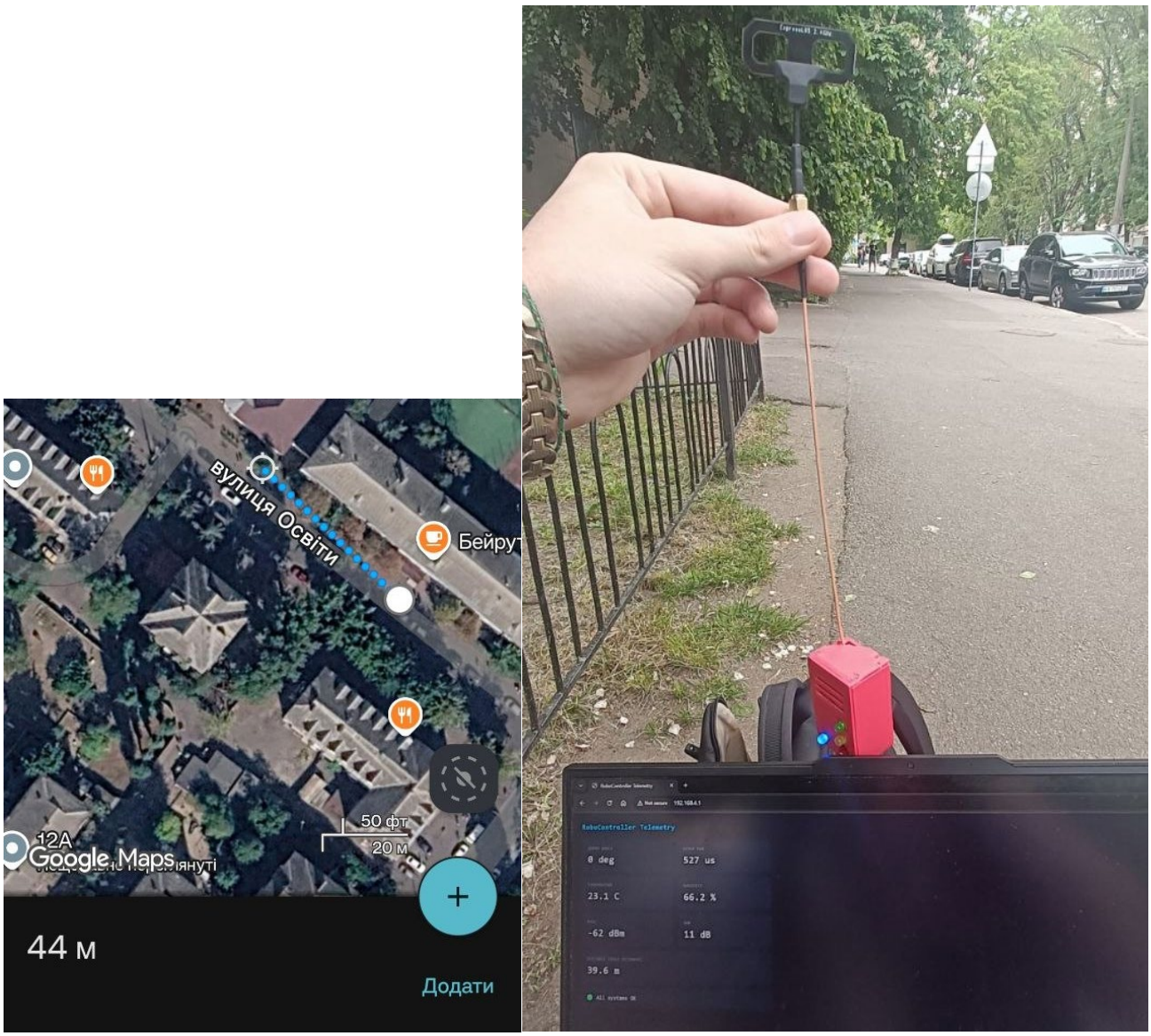

Рисунок 3.20 – Умови проведення тесту

3.3.4 Використання Git та аналіз даних на serial порту під час розробки

Під час розробки активно використовувався ДСКВ Git. Кожен коміт фіксує конкретний етап розробки із описом внесених змін, що дозволяє відстежити

еволюцію проєкту від початкової ініціалізації структури репозиторію до фінальної версії прошивки обох вузлів. Рисунок 3.21 демонструє дерево комітів проекту на гітхаб.

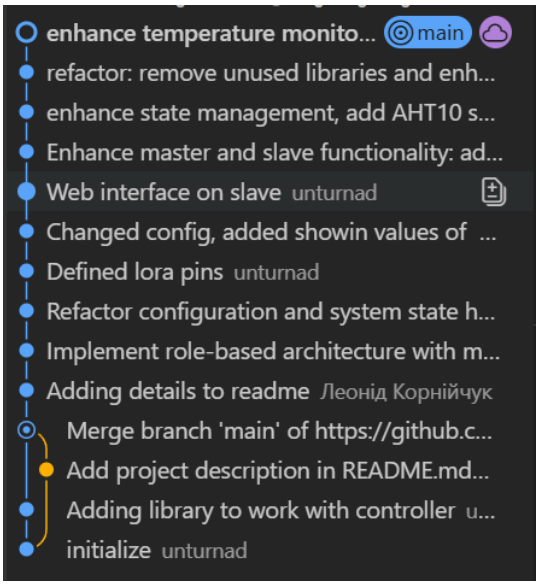

Рисунок 3.21 – Дерево комітів

Також слід показати вікно діагностики до поки не з’явився веб–інтерфейс, який на справді є доволі банальним інструментом. Рисунок 3.22 демонструє виведення журналу повідомлень.

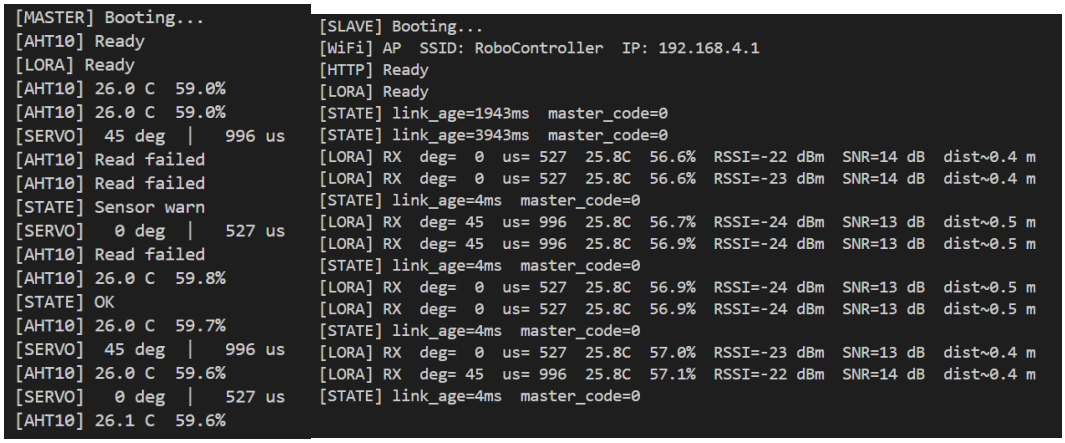

Рисунок 3.22 – Виведення інформації на Serial–порт

3.3.5 Аналіз споживання та вигляд PWM сигналу на осцилографі

При розводці та додаванні елементів проводилося діагностування споживання модуля при різних сценаріях. В середньому, якщо оцінювати, що використовуються банки 18650 в яких ємність 3000 мАг, то розрахувати час роботи можна наступним чином:

3 секунди він знаходиться в стані спокою та споживає 0.052 А, 1.5 секунд серва працює споживання зростає і стає 0.076 А. На рисунку 3.23 демонструються показники споживання модуля Master, в свою чергу 3.24 показує вигляд тестового стенду для дослідження ШІМ-сигналу. Повний цикл виходить 4.5 секунди, тобто приблизний час роботи в спокої 67%, та 33% з цього часу працює серва. Обрахуємо середнє споживання:

$$0.052 \times 0.67 + 0.076 \times 0.33 = 0.0348 + 0.0251 \approx 0.060 \text{ A} \quad (2)$$

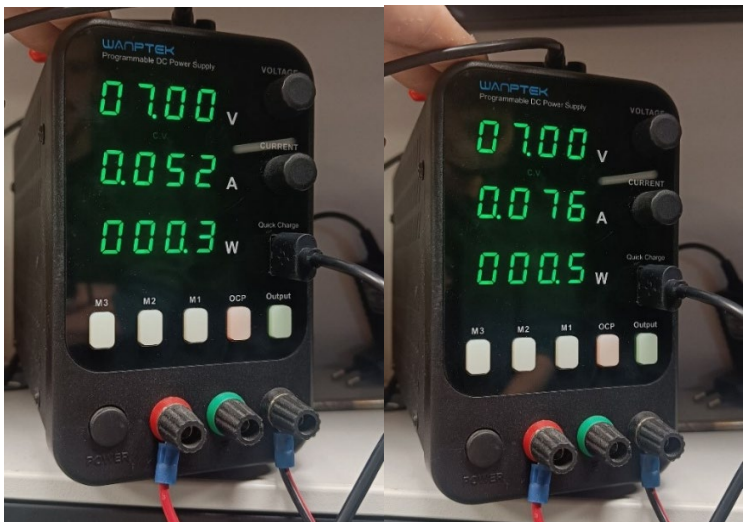

Рисунок 3.23 – Споживання у ватах та струм при номінальній напрузі у стані спокою і при роботі серви

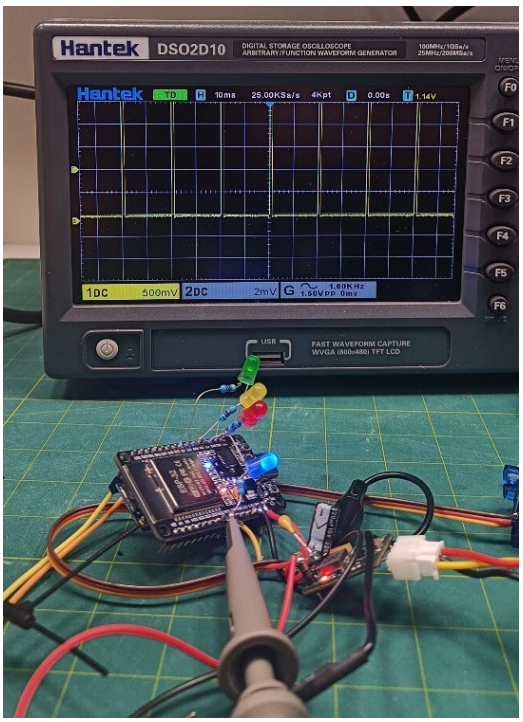

Рисунок 3.24 – Вигляд тестового стенда

Тестовий стенд для аналізу роботи PWM контакта виглядав наступним чином, під’єднаним сигнальний щупом підключився до контакта який генерує ШИМ сигнал і під’єднав землю:

Наступним чином виглядає різниця сигналу для сервина рисунках 3.25 та 3.26, яка повертає її на 45°:

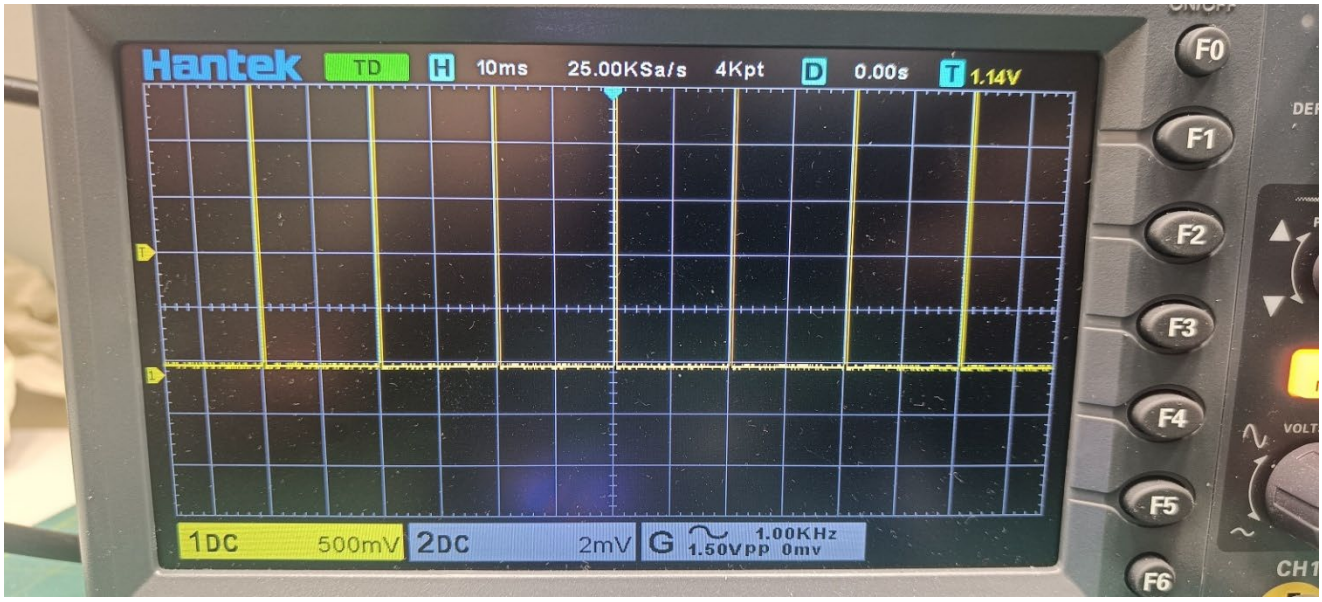

Рисунок 3.25 – Вигляд сигналу при подачі 527 PWM

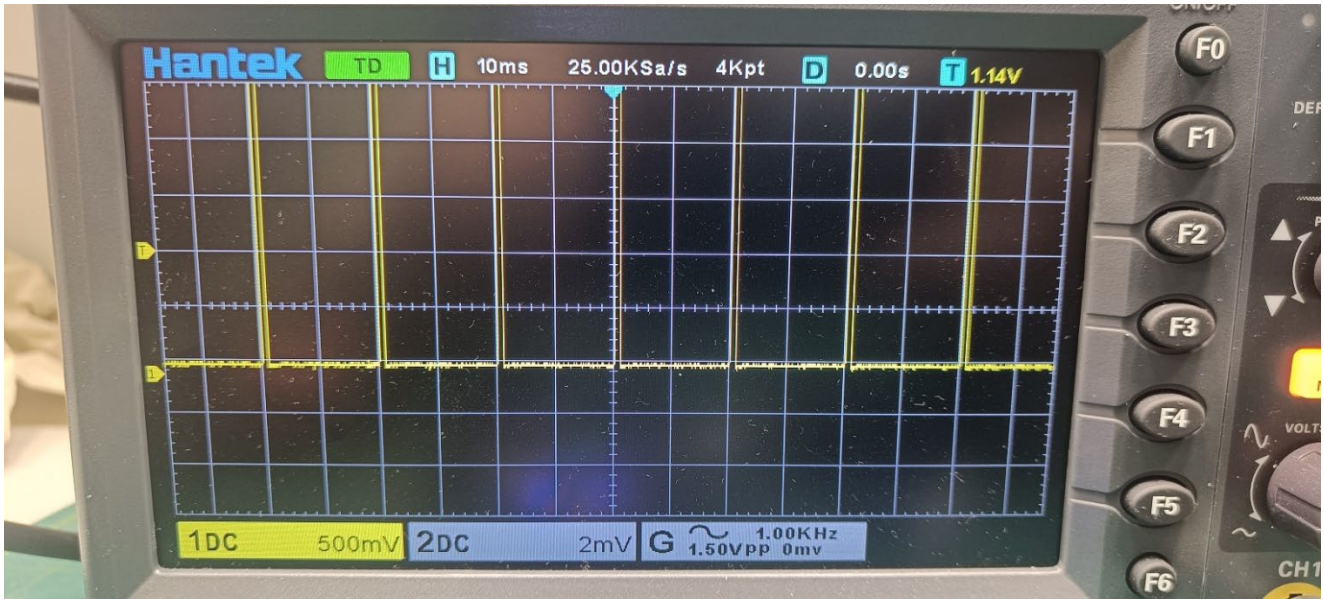

Рисунок 3.26 – Вигляд сигналу при подачі 996 PWM

Для полегшення проведення вуличних тестувань для модуля slave було створено корпус для транспортування.

Під час польових тестувань було знайдено декілька багів та слабкі місця у фіксації модуля в корпусі. Після проведення випробувань всі проблеми були виправлено.

Висновок до розділу 3

За результатами практичної реалізації та тестування підтверджено працездатність розробленої системи та коректність усіх реалізованих механізмів відмовостійкості в умовах наближених до реальних сценаріїв експлуатації.

Монтаж обох вузлів виконано з дотриманням вимог до розділення силової та цифрової частин живлення, якості антенного тракту та орієнтації антен для узгодження поляризації. Програмні модулі майстра та слейва реалізовані в єдиному репозиторії з розподілом ролей на етапі компіляції, що забезпечило відтворюваність збірки та унеможливило розсинхронізацію формату телеметричних пакетів між вузлами.

Серія випробувань у режимі імітації відмов підтвердила, що система коректно реагує на обрив лінії I2C, перевищення температурних порогів та повну втрату радіозв'язку: у кожному сценарії FSM переходила у відповідний стан, індикація та веб–інтерфейс відображали актуальну причину відхилення, а після усунення несправності система автономно відновлювала нормальну роботу. Вимірювання вихідної потужності радіомодуля апаратурою Rohde & Schwarz підтвердило відповідність радіочастотного тракту розрахунковому енергетичному бюджету. Польові випробування радіоканалу показали стабільну передачу телеметрії на дистанції понад 100 метрів за відсутності прямої видимості, а розрахунок енергоспоживання підтвердив здатність системи до автономної роботи близько 37–40 годин від акумулятора 2S 1P на базі елементів 18650.

Окремо верифіковано поведінку системи при одночасній наявності кількох активних помилок: у разі деградації як I2C-підсистеми так і радіоканалу FSM коректно визначала результуючий стан за принципом найгіршого результату, а механізм tryRecovery послідовно намагався відновити кожну несправну підсистему

незалежно. Це підтвердило що ешелонована система захисту працює як єдиний злагоджений контур, а не як набір ізольованих перевірок.

Отримані результати свідчать про те, що поставлені задачі виконано в повному обсязі, а розроблена система може слугувати практичною основою для побудови відмовостійких керуючих вузлів у ширшому класі роботизованих та вбудованих застосувань.

ВИСНОВКИ

У ході виконання дипломної роботи було розроблено та практично реалізовано відмовостійкий контролер для роботизованих систем на базі мікроконтролера ESP32 підтримуючим RTOS, додавши до нього резервний канал передачі телеметрії через радіоінтерфейс LoRa.

Центральною метою роботи було не просто створення працездатного контролера, а забезпечення його стійкості до відмов – здатності зберігати доступ до сенсорів та керуванням органами, ще й безперервно інформувати оператора про власний стан навіть за умов часткового виходу з ладу окремих підсистем.

Центром відмовостійкості розробленої системи є архітектура Master–Slave із двома фізично незалежними вузлами. Такий поділ забезпечує природну ізоляцію рівня керування від рівня моніторингу: майстер–вузол виконує керування сервоприводом та зчитування датчика, тоді як слейв–вузол виступає незалежним засобом діагностики. Принциповим є те, що відмова слейв–вузла, втрата радіоканалу або вихід з ладу веб–інтерфейсу жодним чином не впливають на здатність майстра продовжувати виконання основних функцій, а збій майстра, навпаки, не позбавляє оператора каналу сповіщення. Жоден з вузлів не має критичної залежності від внутрішнього стану іншого, що усуває єдину точку відмови, характерну для класичних однопроцесорних рішень.

У роботі реалізовано ешелоновану систему захисту від відмов, де кожен механізм перекриває певний клас аварійних сценаріїв. Скінченний автомат станів із трьома рівнями Green, Yellow та Red забезпечує градуйовану реакцію на діагностичні події: система розрізняє нормальну роботу, деградацію та критичну несправність. Програмні тайм–аути при кожному зверненні до периферії унеможливлюють блокування задач реального часу у разі зависання шини або несправності пристрою, що є критичним для збереження детермінованості системи. Апаратний сторожовий таймер виступає останнім рубежем захисту, гарантуючи відновлення роботи навіть за тих непередбачених сценаріїв, які не були явно враховані при проєктуванні логіки. Особливе значення має механізм

автономного відновлення: замість сліпого перезавантаження всього контролера він виконує переініціалізацію саме тієї підсистеми, що відмовила, зберігаючи накопичений контекст роботи. Це відрізняє розроблену систему від простих рішень із watchdog–перезапуском і наближає її до промислового розуміння відмовостійкості. Цілісність телеметричних пакетів захищена алгоритмом CRC, що дозволяє слейву відрізняти достовірні дані від спотворених і трактувати пошкоджений пакет як відсутність зв'язку, а не як хибну інформацію.

Перевірка механізмів захисту проводилася не епізодично, а систематично – через навмисне відтворення аварійних сценаріїв у спеціально виділених гілках репозиторію з префіксом fault, що дозволило документовано та відтворювано підтвердити поведінку системи за кожного класу відмов. Серія випробувань у режимі імітації відмов – обрив лінії I2C, перевищення попереджувального і критичного температурних порогів, повна втрата радіозв'язку, відключення датчика під час роботи – підтвердила, що система у кожному випадку коректно переходить у відповідний стан, своєчасно та однозначно сповіщає оператора через індикацію і веб–інтерфейс, а після усунення причини відмови автономно повертається до нормальної роботи без втручання людини. Тести радіоканалу в реальних умовах із перешкодами показали, що резервний канал телеметрії зберігає стабільність передачі на дистанції понад 100 метрів навіть за відсутності прямої видимості, а вимірювання вихідної потужності модуля апаратурою Rohde & Schwarz підтвердило відповідність радіочастотного тракту розрахунковому енергетичному бюджету. Аналіз залежності похибки вимірювання дистанції від параметрів SF та BW дозволив обґрунтувати вибір робочих параметрів модуляції, а розрахунок енергоспоживання показав здатність системи до автономної роботи протягом близько 37–40 годин від акумулятора на основі елементів 18650.

Випробування виявили й обмеження поточної реалізації, що не стосуються власне механізмів відмовостійкості, але визначають напрями подальшого вдосконалення. Точність вимірювання дистанції методом RTToF суттєво залежить від потужності модуля, поляризації антен та наявності перешкод, а відсутність

програмної фільтрації призводить до помітних коливань виміряних значень. Калібрування антенного тракту під конкретну антену, рекомендоване виробником, у межах роботи не виконувалось, що також вплинуло на похибку.

З огляду на отримані результати розроблену систему доцільно рекомендувати до використання як основу контролера для навчальних та дослідницьких роботизованих платформ, а також як практичний приклад реалізації принципів відмовостійкості у вбудованих системах реального часу. Перспективними напрямами розвитку є розширення набору контрольованих діагностичних параметрів, упровадження програмної фільтрації даних дистанції, калібрування антенного тракту та застосування вбудованого механізму ranging трансивера SX1280 для задач локалізації вузлів. Реалізована система демонструє, що принципи відмовостійкості можуть бути ефективно втілені навіть на доступній елементній базі, що робить її практично цінною для широкого кола застосувань

ПЕРЕЛІК ПОСИЛАНЬ

1. ДСТУ 3008:2015. Інформація та документація. Звіти у сфері науки і техніки. Структура та правила оформлювання. – Київ : ДП «УкрНДНЦ», 2016. – 26 с.

2. ESP32 Series Datasheet. Version 4.4 / Espressif Systems. – Shanghai : Espressif Systems, 2023. – 65 с.

3. ESP32 Technical Reference Manual. Version 5.1 / Espressif Systems. – Shanghai : Espressif Systems, 2023. – 1198 с.

4. SX1280/SX1281 Datasheet. Rev. 3.2 / Semtech Corporation. – Camarillo : Semtech Corporation, 2021. – 130 с.

5. FreeRTOS Kernel Reference Manual. Rev. 1.0 / Real Time Engineers Ltd. – 2022. – 213 с.

6. I2C–bus specification and user manual. Rev. 7.0 / NXP Semiconductors. – Eindhoven : NXP Semiconductors, 2021. – 64 с.

7. SG90 Micro Servo Datasheet / Tower Pro Pte Ltd. – Singapore : Tower Pro, 2010. – 2 с.

8. Espressif IoT Development Framework (ESP–IDF) Documentation [Електронний ресурс] / Espressif Systems. – Режим доступу : https://docs.espressif.com/projects/esp–idf/en/latest/ (дата звернення: 10.05.2026).

9. Semtech LoRa Developer Portal [Електронний ресурс] / Semtech Corporation. – Режим доступу : https://lora–developers.semtech.com (дата звернення: 10.05.2026).

10. Polak L. On the Interference between LoRa and Bluetooth in the 2.4 GHz Unlicensed Band / L. Polak, F. Paul, M. Simka, R. Zedka, J. Kufa, R. Sotner // 2022 32nd International Conference Radioelektronika (RADIOELEKTRONIKA). – Kosice : IEEE, 2022. – С. 1–4. – DOI: 10.1109/RADIOELEKTRONIKA54537.2022.9764912.

11. Tofterup V. K. A Methodology for Evaluating Commercial Off The Shelf Parachutes Designed for sUAS Failsafe Systems / V. K. Tofterup, K. Jensen // 2019 International Conference on Unmanned Aircraft Systems (ICUAS). – Atlanta : IEEE, 2019. – С. 129–137. – DOI: 10.1109/ICUAS.2019.8798368.

12. Shao C. Toward Resilient LoRa Communication under Wi–Fi Interference / C. Shao, K. Tsukamoto, Y. W. Ma // 2024 International Conference on Consumer Electronics–Taiwan (ICCE–Taiwan). – Taiwan : IEEE, 2024. – С. 249–250. – DOI: 10.1109/ICCE–Taiwan62264.2024.10674150.

13. Enhanced I2C Architecture With Integrated CRC for Error Detection in Multi–Slave Systems // IEEE Conference Publication. – IEEE, 2025. – DOI: 10.1109/11010016.

14. Zuo B. Robotised Tower Cranes with Hardware–in–The–Loop: Advancing Construction Safety and Efficiency / B. Zuo, T. K. X. Ku, W. T. Ang // SSRN Electronic Journal. – 2024. – DOI: 10.2139/ssrn.4751401.

15. Hochet Derévianckine G. Opportunities and Challenges of LoRa 2.4 GHz / G. Hochet Derévianckine, A. Guitton, O. Iova, B. Ning, F. Valois // IEEE Communications Magazine. – 2023. – DOI: 10.1109/MCOM.010.2200566.

16. Rohde & Schwarz. Internet of Things (IoT) Device Testing [Електронний ресурс] / Rohde & Schwarz GmbH & Co. KG. – Режим доступу : https://www.rohde–schwarz.com/us/solutions/wireless–communications–testing/iot–m2m–testing/iot–device–certification–and–qualification/iot–device–certification–and–qualification_233860.html (дата звернення: 20.05.2026).

17. Архітектури мікроконтролерів: порівняльний огляд [Електронний ресурс] / techmaker.ua. – Режим доступу : https://blog.techmaker.ua/posts/mcu–architectures/ (дата звернення: 08.05.2026).

18. Betaflight Failsafe Configuration Guide [Електронний ресурс] / Betaflight Project. – Режим доступу : https://betaflight.com/docs/wiki/guides/current/Failsafe (дата звернення: 09.05.2026).

19. ArduPilot Failsafe Documentation [Електронний ресурс] / ArduPilot Dev Team. – Режим доступу : https://ardupilot.org/copter/docs/failsafe–landing–page.html (дата звернення: 09.05.2026).

20. SG90 Micro Servo Motor Datasheet [Електронний ресурс] / Imperial College London. – Режим доступу :

http://www.ee.ic.ac.uk/pcheung/teaching/DE1_EE/stores/sg90_datasheet.pdf (дата звернення: 07.05.2026).

21. AHT10 Temperature and Humidity Sensor Datasheet [Електронний ресурс] / ASAIR. – Режим доступу : https://altronics.cl/uploads/AHT10.pdf (дата звернення: 07.05.2025).

22. Top 10 Applications of Microcontrollers in Daily Life, Industry & Technology [Електронний ресурс] / RoboCraze. – Режим доступу : https://robocraze.com/blogs/post/top–10–applications–of–microcontrollers–in–daily–life–industry–technology (дата звернення: 08.05.2025).

23. ESP32 DOIT DevKit V1 – PlatformIO Board Documentation [Електронний ресурс] / PlatformIO. – Режим доступу : https://docs.platformio.org/en/latest/boards/espressif32/esp32doit–devkit–v1.html (дата звернення: 10.05.2025).

24. Ротхаммель К. Антени / К. Ротхаммель, А. Крішке. – 13–те вид., переробл. та доп. – Штутгарт : Franckh–Kosmos Verlag, 2001. – 1248 с.

25. Horowitz P. The Art of Electronics / P. Horowitz, W. Hill. – 3rd ed. – Cambridge : Cambridge University Press, 2015. – 1220 с.

ДОДАТОК А

ПОВНИЙ КОД MASTER–ВУЗЛА

Код яким зашивається модуль master який має в собі відмовостійку логіку та працює з сенсорами, сервоприводом та LoRa трансивером

```
#ifdef ROLE_MASTER
#include "master.h"
#include "config.h"
#include "system_state.h"
#include "lora_packet.h"
#include <SPI.h>
#include <Wire.h>
#include <SX128XLT.h>
#include <ESP32Servo.h>
#include <esp_task_wdt.h>
#define WDT_TIMEOUT_S  15

static SX128XLT LT;
static Servo    servo;
static bool     ahtOk = false;

static unsigned long servoTimer    = 0;
static unsigned long txTimer       = 0;
static unsigned long recoveryTimer = 0;
static bool servoAtRest            = true;
static uint8_t sensorFails         = 0;
static uint8_t txFails             = 0;
static uint8_t masterCode          = MASTER_OK;
static SystemState currentState    = SystemState::YELLOW;

static const int  SERVO_REST_DEG            = 0;
static const int  SERVO_MOVE_DEG            = 45;
static const unsigned long SERVO_INTERVAL_MS = 3000;
static const unsigned long SERVO_HOLD_MS     = 1500;
static const unsigned long TX_INTERVAL_MS    = 1000;
static const unsigned long RECOVERY_INTERVAL_MS = 10000;

static const uint8_t FAILS_YELLOW = 2;
static const uint8_t FAILS_RED    = 5;
```

```
// ——— AHT10 bare I2C ————————————————————————————————————————————————————
#define AHT10_ADDR 0x38

static bool aht10Init() {
  Wire.beginTransmission(AHT10_ADDR);
  if (Wire.endTransmission() != 0) return false;
  Wire.beginTransmission(AHT10_ADDR);
  Wire.write(0xBA);
  Wire.endTransmission();
  delay(20);
  Wire.beginTransmission(AHT10_ADDR);
  Wire.write(0xBE); Wire.write(0x08); Wire.write(0x00);
  Wire.endTransmission();
  delay(300);

  return true;
}
static bool aht10Read(float &temp, float &hum) {
  Wire.beginTransmission(AHT10_ADDR);
  Wire.write(0xAC); Wire.write(0x33); Wire.write(0x00);
  if (Wire.endTransmission() != 0) return false;
  delay(80);
  if (Wire.requestFrom((uint8_t)AHT10_ADDR, (uint8_t)6) != 6) return false;
  uint8_t d[6];
  for (auto &b : d) b = Wire.read();
  if (d[0] & 0x80) return false;
  uint32_t rawHum  = ((uint32_t)d[1] << 12) | ((uint32_t)d[2] << 4) | (d[3] >> 4);
  uint32_t rawTemp = ((uint32_t)(d[3] & 0x0F) << 16) | ((uint32_t)d[4] << 8) | d[5];
  hum  = (rawHum  / 1048576.0f) * 100.0f;
  temp = (rawTemp / 1048576.0f) * 200.0f – 50.0f;
  return true;
}
static bool loraInit() {
  SPI.begin(LORA_SCK, LORA_MISO, LORA_MOSI, LORA_NSS);
  if (!LT.begin(LORA_NSS, LORA_NRESET, LORA_RFBUSY, LORA_DIO1, DEVICE_SX1280))
    return false;
  LT.setupLoRa(LORA_FREQ_HZ, 0, LORA_SF7, LORA_BW_0800, LORA_CR_4_5);
  return true;
}
```

```
// ——— Recovery
————————————————————————————————————————————————————————————————————————————

static void tryRecovery() {
    unsigned long now = millis();
    if (now – recoveryTimer < RECOVERY_INTERVAL_MS) return;
    recoveryTimer = now;

    if (masterCode == MASTER_SENSOR_FAIL || masterCode == MASTER_NO_SENSOR) {
        Serial.println("[RECOVERY] Retrying AHT10...");
        ahtOk = aht10Init();
        if (ahtOk) { sensorFails = 0; Serial.println("[RECOVERY] AHT10 restored"); }
        else       Serial.println("[RECOVERY] AHT10 still unavailable");
    }
    if (masterCode == MASTER_TX_FAIL) {
        Serial.println("[RECOVERY] Retrying LoRa TX...");
        if (loraInit()) { txFails = 0; Serial.println("[RECOVERY] LoRa TX restored"); }
        else            Serial.println("[RECOVERY] LoRa TX still unavailable");
    }
}
// ——— State machine
————————————————————————————————————————————————————————————————————————

static void updateState(bool sensorOk, bool txOk, float temp) {
    uint8_t prev = masterCode;
    if (!ahtOk) {
        masterCode = MASTER_NO_SENSOR;
    } else if (!servo.attached()) {
        masterCode = MASTER_NO_SERVO;
    } else {
        sensorFails = sensorOk ? 0 : (uint8_t)(sensorFails + 1);
        txFails     = txOk     ? 0 : (uint8_t)(txFails     + 1);
        if      (sensorFails >= FAILS_RED)            masterCode = MASTER_SENSOR_FAIL;
        else if (txFails     >= FAILS_RED)            masterCode = MASTER_TX_FAIL;
        else if (sensorOk && temp >= TEMP_CRIT_C)     masterCode = MASTER_TEMP_CRIT;
        else if (sensorFails >= FAILS_YELLOW)         masterCode = MASTER_SENSOR_WARN;
        else if (txFails     >= FAILS_YELLOW)         masterCode = MASTER_TX_WARN;
        else if (sensorOk && temp >= TEMP_WARN_C)     masterCode = MASTER_TEMP_WARN;
        else                                          masterCode = MASTER_OK;
    }
    currentState = (masterCode == MASTER_OK)                  ? SystemState::GREEN
                 : (masterCode == MASTER_SENSOR_WARN ||
                    masterCode == MASTER_TX_WARN    ||
```

```
                masterCode == MASTER_TEMP_WARN)              ? SystemState::YELLOW
              : SystemState::RED;
    if (masterCode != prev) {
        static const char* names[] = {
            "OK", "Sensor warn", "TX warn",
            "No sensor", "No servo", "Sensor fail", "TX fail",
            "Temp high", "Temp critical"
        };
        Serial.printf("[STATE] %s\n", names[masterCode < 9 ? masterCode : 0]);
    }
    applyState(currentState);
    if (currentState == SystemState::RED) tryRecovery();
}
//
```

```
static void logServo(int deg) {
    Serial.printf("[SERVO] %3d deg  |  %4d us\n", deg, servo.readMicroseconds());
}
static void sendTelemetry() {
    ServoPacket pkt;
    pkt.type        = PKT_SERVO;
    pkt.magic       = PKT_MAGIC;
    pkt.deg         = (int16_t)(servoAtRest ? SERVO_REST_DEG : SERVO_MOVE_DEG);
    pkt.us          = (uint16_t)servo.readMicroseconds();
    pkt.masterState = masterCode;
    float temp = 0, hum = 0;
    bool sensorOk = ahtOk && aht10Read(temp, hum);
    if (sensorOk) {
        pkt.tempC10  = (int16_t)(temp * 10.0f);
        pkt.humPct10 = (uint16_t)(hum  * 10.0f);
        Serial.printf("[AHT10] %.1f C  %.1f%%\n", temp, hum);
    } else {
        pkt.tempC10  = 0;
        pkt.humPct10 = 0;
        if (ahtOk) Serial.println("[AHT10] Read failed");
    }
    uint8_t len = LT.transmit((uint8_t*)&pkt, sizeof(pkt), 1000, LORA_TX_POWER, WAIT_TX);
    bool txOk = (len > 0);
    if (!txOk) Serial.println("[LORA] TX error");
    updateState(sensorOk, txOk, temp);
```

```
}
// ——— Public API
————————————————————————————————————————————————————————————————————————————
void masterSetup() {
   esp_task_wdt_init(WDT_TIMEOUT_S, true);  // panic–reset if loop stalls
   esp_task_wdt_add(NULL);                  // subscribe Arduino loop task
   Wire.begin(I2C_SDA, I2C_SCL);
   Wire.setClock(100000);
   ahtOk = aht10Init();
   Serial.printf("[AHT10] %s\n", ahtOk ? "Ready" : "Not found");
   if (!loraInit()) {
      Serial.println("[LORA] Init failed");
      applyState(SystemState::RED);       return;    }
   Serial.println("[LORA] Ready");
   servo.attach(SERVO_PIN);
   servo.write(SERVO_REST_DEG);
   servoTimer = millis();
   txTimer    = millis();
   applyState(ahtOk ? SystemState::YELLOW : SystemState::RED);}
void masterLoop() {
   esp_task_wdt_reset();   // feed watchdog
   unsigned long now = millis();
   if (servoAtRest) {
      if (now – servoTimer >= SERVO_INTERVAL_MS) {
         servo.write(SERVO_MOVE_DEG);
         servoAtRest = false;
         servoTimer  = now;
         logServo(SERVO_MOVE_DEG);       }
   } else {
      if (now – servoTimer >= SERVO_HOLD_MS) {
         servo.write(SERVO_REST_DEG);
         servoAtRest = true;
         servoTimer  = now;
         logServo(SERVO_REST_DEG);
      }   }
   if (now – txTimer >= TX_INTERVAL_MS) {
      sendTelemetry();
      txTimer = now;
   }
}
#endif
```

ДОДАТОК Б

ПОВНИЙ КОД SLAVE–ВУЗЛА

Код яким зашивається модуль slave який має в собі відмовостійку логіку та працює з LoRa трансивером та з отриманих пакетів даних виводить інформацію на веб–інтерфейс через точку доступу

```
#ifdef ROLE_SLAVE
#include "slave.h"
#include "config.h"
#include "system_state.h"
#include "lora_packet.h"
#include <SPI.h>
#include <SX128XLT.h>
#include <WiFi.h>
#include <WebServer.h>
#include <math.h>
#include <esp_task_wdt.h>
static SX128XLT  LT;
static WebServer server(SLAVE_WIFI_PORT);
// ── Shared telemetry
// ──────────────────────────────────────────────────────────────────────
// Written from Core 0 (LoRa task), read from Core 1 (web + loop).
static volatile int16_t       lastDeg      = 0;
static volatile uint16_t      lastUs       = 0;
static volatile int16_t       lastRssi     = 0;
static volatile int8_t        lastSnr      = 0;
static volatile int16_t       lastTempC10    = 0;   // °C × 10
static volatile uint16_t      lastHumPct10   = 0;   // %  × 10
static volatile uint8_t       lastMasterState = MASTER_OK;
static volatile unsigned long lastRxTime      = 0;
static unsigned long          startTime    = 0;   // set at boot, baseline before first packet
// Free–space path loss at 2.4 GHz: d = 10^((TxPower – RSSI – 40.05) / 20) m
static float rssiToDistance(int16_t rssi) {
    return powf(10.0f, ((float)LORA_TX_POWER – (float)rssi – 40.05f) / 20.0f);
}
// Human–readable reason for the current state – set by getOverallState()
static const char* stateReason = "Initialising";
// Maps MASTER_* code → { SystemState, description }
struct MasterInfo { SystemState s; const char* reason; };
static const MasterInfo MASTER_MAP[] = {
```

```
    { SystemState::GREEN,  "All systems OK"       },  // 0 MASTER_OK
    { SystemState::YELLOW, "Sensor intermittent"  },  // 1 MASTER_SENSOR_WARN
    { SystemState::YELLOW, "TX intermittent"      },  // 2 MASTER_TX_WARN
    { SystemState::RED,    "Sensor not found"     },  // 3 MASTER_NO_SENSOR
    { SystemState::RED,    "Servo detached"       },  // 4 MASTER_NO_SERVO
    { SystemState::RED,    "Sensor failed"        },  // 5 MASTER_SENSOR_FAIL
    { SystemState::RED,    "TX failed"            },  // 6 MASTER_TX_FAIL
    { SystemState::YELLOW, "Temperature high"     },  // 7 MASTER_TEMP_WARN
    { SystemState::RED,    "Temperature critical" },  // 8 MASTER_TEMP_CRIT
};
static SystemState getOverallState() {
    // ── Link health ──────────────────────────────────────────────────────────
    unsigned long ref = (lastRxTime == 0) ? startTime : (unsigned long)lastRxTime;
    unsigned long age = millis() – ref;
    static unsigned long dbgTimer = 0;
    if (millis() – dbgTimer >= 2000) {
        Serial.printf("[STATE] link_age=%lums  master_code=%u\n", age, (unsigned)lastMasterState);
        dbgTimer = millis();
    }
    SystemState linkState;
    const char*  linkReason;
    if      (age < 3000) { linkState = SystemState::GREEN;  linkReason = nullptr;         }
    else if (age < 8000) { linkState = SystemState::YELLOW; linkReason = "Link degraded";  }
    else                 { linkState = SystemState::RED;    linkReason = "Link lost";      }
    // ── Master health ────────────────────────────────────────────────────────
    uint8_t code = (lastMasterState < 9) ? (uint8_t)lastMasterState : MASTER_OK;
    SystemState masterState  = MASTER_MAP[code].s;
    const char* masterReason = MASTER_MAP[code].reason;
    // ── Combine: worst wins; link problem takes display priority ─────────────
    if (linkState == SystemState::RED || masterState == SystemState::RED) {
        stateReason = (linkState == SystemState::RED) ? linkReason : masterReason;
        return SystemState::RED;
    }
    if (linkState == SystemState::YELLOW || masterState == SystemState::YELLOW) {
        stateReason = (linkState == SystemState::YELLOW) ? linkReason : masterReason;
        return SystemState::YELLOW;
    }
    stateReason = masterReason;   // "All systems OK"
    return SystemState::GREEN;
```

```
}
// ── LoRa receive task – Core 0 ──────────────────────────────────────────────
static void loraRxTask(void*) {
    // IDLE0 never runs since we busy–wait here – remove from WDT, add this task instead
    esp_task_wdt_delete(xTaskGetIdleTaskHandleForCPU(0));
    esp_task_wdt_add(NULL);   // WDT now monitors lora_rx; resets every receive() cycle
    for (;;) {
        uint8_t rxBuf[sizeof(ServoPacket)] = {};
        uint8_t rxLen = LT.receive(rxBuf, sizeof(rxBuf), 5000, WAIT_RX);
        esp_task_wdt_reset();   // receive() returned (packet or 5 s timeout) – feed WDT
        if (rxLen >= (uint8_t)sizeof(ServoPacket)) {
            auto* pkt = reinterpret_cast<ServoPacket*>(rxBuf);
            if (pkt–>type == PKT_SERVO && pkt–>magic == PKT_MAGIC) {
                lastDeg      = pkt–>deg;
                lastUs       = pkt–>us;
                lastTempC10    = pkt–>tempC10;
                lastHumPct10   = pkt–>humPct10;
                lastMasterState = pkt–>masterState;
                lastRssi       = LT.readPacketRSSI();
                lastSnr      = LT.readPacketSNR();
                lastRxTime   = millis();
                Serial.printf("[LORA] RX  deg=%3d  us=%4u  %.1fC  %.1f%%  "
                              "RSSI=%d dBm  SNR=%d dB  dist~%.1f m\n",
                              (int)lastDeg, (unsigned)lastUs,
                              lastTempC10 / 10.0f, lastHumPct10 / 10.0f,
                              (int)lastRssi, (int)lastSnr,
                              rssiToDistance(lastRssi));
            }
        }
    }
}
// ── Web page
─────────────────────────────────────────────────────────────────────────────

static const char HTML[] = R"html(
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF–8">
<title>RoboController Telemetry</title>
<style>
  *{box–sizing:border–box;margin:0;padding:0}
```

```
 body{font–family:monospace;background:#1a1a2e;color:#e0e0e0;padding:24px}
 h1{color:#00d4ff;margin–bottom:20px;font–size:1.4em}
 .grid{display:grid;grid–template–columns:1fr 1fr;gap:10px;max–width:480px}
 .card{background:#16213e;border–radius:8px;padding:16px}
 .wide{grid–column:span 2}
 .label{color:#888;font–size:.75em;text–transform:uppercase;letter–spacing:.05em;margin–bottom:6px}
 .value{font–size:1.9em;font–weight:bold}
 .status{display:flex;align–items:center;gap:10px;background:#16213e;
       border–radius:8px;padding:16px;max–width:480px;margin–top:10px}
 .dot{width:14px;height:14px;border–radius:50%;flex–shrink:0}
 .GREEN{background:#00ff88} .YELLOW{background:#ffd700} .RED{background:#ff4444}
</style>
</head>
<body>
<h1>RoboController Telemetry</h1>
<div class="grid">
 <div class="card">
  <div class="label">Servo Angle</div>
  <div class="value" id="deg">—</div>
 </div>
 <div class="card">
  <div class="label">Servo PWM</div>
  <div class="value" id="us">—</div>
 </div>
 <div class="card">
  <div class="label">Temperature</div>
  <div class="value" id="temp">—</div>
 </div>
 <div class="card">
  <div class="label">Humidity</div>
  <div class="value" id="hum">—</div>
 </div>
 <div class="card">
  <div class="label">RSSI</div>
  <div class="value" id="rssi">—</div>
 </div>
 <div class="card">
  <div class="label">SNR</div>
  <div class="value" id="snr">—</div>
 </div>
 <div class="card wide">
```

```
    <div class="label">Distance (RSSI estimate)</div>
    <div class="value" id="dist">—</div>
  </div>
</div>
<div class="status">
  <div class="dot" id="dot"></div>
  <span id="state">Waiting for signal...</span>
</div>
<script>
function poll() {
  fetch('/data')
    .then(r => r.json())
    .then(d => {
      document.getElementById('deg').textContent  = d.deg  + ' deg';
      document.getElementById('us').textContent   = d.us   + ' us';
      document.getElementById('temp').textContent = d.temp + ' C';
      document.getElementById('hum').textContent  = d.hum  + ' %';
      document.getElementById('rssi').textContent = d.rssi + ' dBm';
      document.getElementById('snr').textContent  = d.snr  + ' dB';
      document.getElementById('dist').textContent = d.dist + ' m';
      document.getElementById('state').textContent = d.reason;
      document.getElementById('dot').className = 'dot ' + d.state;
    })
    .catch(() => {});
}
setInterval(poll, 1000);
poll();
</script>
</body>
</html>
)html";
// ── HTTP handlers
──────────────────────────────────────────────────────────────────────────────

static void handleRoot() {
    server.send(200, "text/html", HTML);}
static void handleData() {
    SystemState s = getOverallState();
    applyState(s);
    const char* stateStr;
    switch (s) {
        case SystemState::GREEN:  stateStr = "GREEN";  break;
```

```
        case SystemState::YELLOW: stateStr = "YELLOW"; break;
        default:                  stateStr = "RED";    break;    }
    char distStr[12], tempStr[10], humStr[10];
    snprintf(distStr, sizeof(distStr), "%.1f", rssiToDistance(lastRssi));
    snprintf(tempStr, sizeof(tempStr), "%.1f", lastTempC10  / 10.0f);
    snprintf(humStr,  sizeof(humStr),  "%.1f", lastHumPct10 / 10.0f);
    char buf[260];
    snprintf(buf, sizeof(buf),
        "{\"deg\":%d,\"us\":%u,\"temp\":%s,\"hum\":%s,"
        "\"rssi\":%d,\"snr\":%d,\"dist\":%s,\"state\":\"%s\",\"reason\":\"%s\"}",
        (int)lastDeg, (unsigned)lastUs, tempStr, humStr,
        (int)lastRssi, (int)lastSnr, distStr, stateStr, stateReason);
    server.send(200, "application/json", buf);
}
// ── Setup / Loop
──────────────────────────────────────────────────────────────────────────────
void slaveSetup() {
    esp_task_wdt_init(15, true);  // 15 s timeout, panic–reset on trigger
    esp_task_wdt_add(NULL);       // subscribe Arduino loop task (Core 1)
    startTime = millis();    WiFi.softAP(SLAVE_WIFI_SSID, SLAVE_WIFI_PASS);
    Serial.printf("[WiFi] AP  SSID: %s  IP: %s\n",
                SLAVE_WIFI_SSID, WiFi.softAPIP().toString().c_str());
    server.on("/",     handleRoot);    server.on("/data", handleData);
    server.begin();    Serial.println("[HTTP] Ready");
    SPI.begin(LORA_SCK, LORA_MISO, LORA_MOSI, LORA_NSS);
    if (!LT.begin(LORA_NSS, LORA_NRESET, LORA_RFBUSY, LORA_DIO1, DEVICE_SX1280)) {
        Serial.println("[LORA] Init failed");
        applyState(SystemState::RED);
        return;    }
    LT.setupLoRa(LORA_FREQ_HZ, 0, LORA_SF7, LORA_BW_0800, LORA_CR_4_5);
    Serial.println("[LORA] Ready");
    // LoRa receive (busy–wait on DIO1) runs on Core 0.
    // Core 1 stays free for web server + WiFi stack.
    xTaskCreatePinnedToCore(loraRxTask, "lora_rx", 4096, nullptr, 1, nullptr, 0);
    applyState(SystemState::YELLOW);
}
void slaveLoop() {
    esp_task_wdt_reset();           // feed watchdog
    applyState(getOverallState());   // LEDs update independently of browser
    server.handleClient();}
#endif
```

ДОДАТОК В

ФАЙЛ PLATFORMIO.INI ІЗ КОНФІГУРАЦІЄЮ СЕРЕДОВИЩА ЗБІРКИ

Конфіг проекту в якому описано таргет, який використовується framework, плата та бібліотеки

```
; PlatformIO Project Configuration File
; https://docs.platformio.org/page/projectconf.html

[common]
platform = espressif32
board = esp32doit–devkit–v1
framework = arduino
monitor_speed = 115200
lib_deps =
   madhephaestus/ESP32Servo @ ^3.0.5
   bblanchon/ArduinoJson @ ^7.3.1

; pio run –e master  → build master firmware
[env:master]
extends = common
build_flags = –D ROLE_MASTER

; pio run –e slave   → build slave firmware
[env:slave]
extends = common
build_flags = –D ROLE_SLAVE
```

ДОДАТОК Г

ФАЙЛ CONFIG.H ІЗ ОГОЛОШЕННЯМИ СТАНІВ ТА GPIO

Файл в якому зберігаються оголошення призначення всіх задіяних пінів плати, станів та налаштування точки доступу.

```
#pragma once
// ——— LoRa SX1280 (custom SPI bus) ——————————————————————————————
#define LORA_MISO    19
#define LORA_MOSI    23
#define LORA_SCK     18
#define LORA_NSS     12
#define LORA_NRESET  13
#define LORA_RFBUSY  14
#define LORA_DIO1    27   // IRQ

#define LORA_FREQ_HZ   2400000000UL
#define LORA_TX_POWER  10     // dBm
#define LORA_TIMEOUT   5000   // ms – packet timeout → state degrades

// ——— Status LEDs – active–low (anode→3.3V, cathode→GPIO) ———————————
// LOW = ON, HIGH = OFF
#define LED_RED     2
#define LED_YELLOW  4
#define LED_GREEN   5
#define LED_BLUE    26   // power indicator, always on

// ——— Master–only pins
————————————————————————————————————————————————

#define SERVO_PIN   25
#define I2C_SDA     21   // AHT10 SDA
#define I2C_SCL     22   // AHT10 SCL

// ——— Temperature thresholds ————————————————————————————————————
#define TEMP_WARN_C   30.0f   // ≥ this → YELLOW "Temperature high"
#define TEMP_CRIT_C   32.0f   // ≥ this → RED    "Temperature critical"

// ——— Slave–only
————————————————————————————————————————————————————

#define SLAVE_WIFI_SSID  "RoboController"
#define SLAVE_WIFI_PASS  "12345678"
```

```
#define SLAVE_WIFI_PORT  80
```

ДОДАТОК Д

ДЕРЕВО ВІДМОВ

На рисунках показано розподілення по ядрам всіх завдань(tasks), і їх перехід станів при відмовах певних вузлів на master та slave

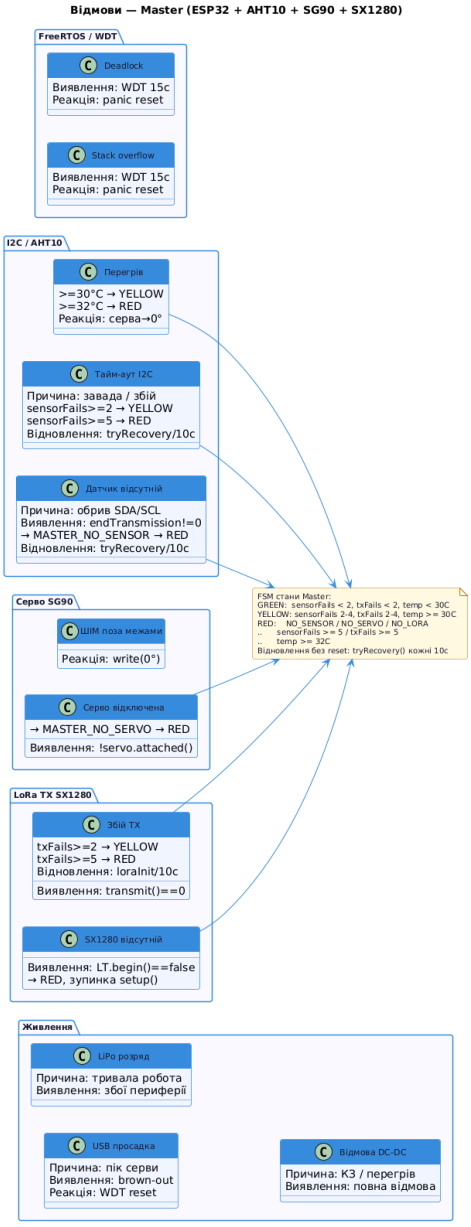

Рисунок Д.1 – Дерево відмов Master

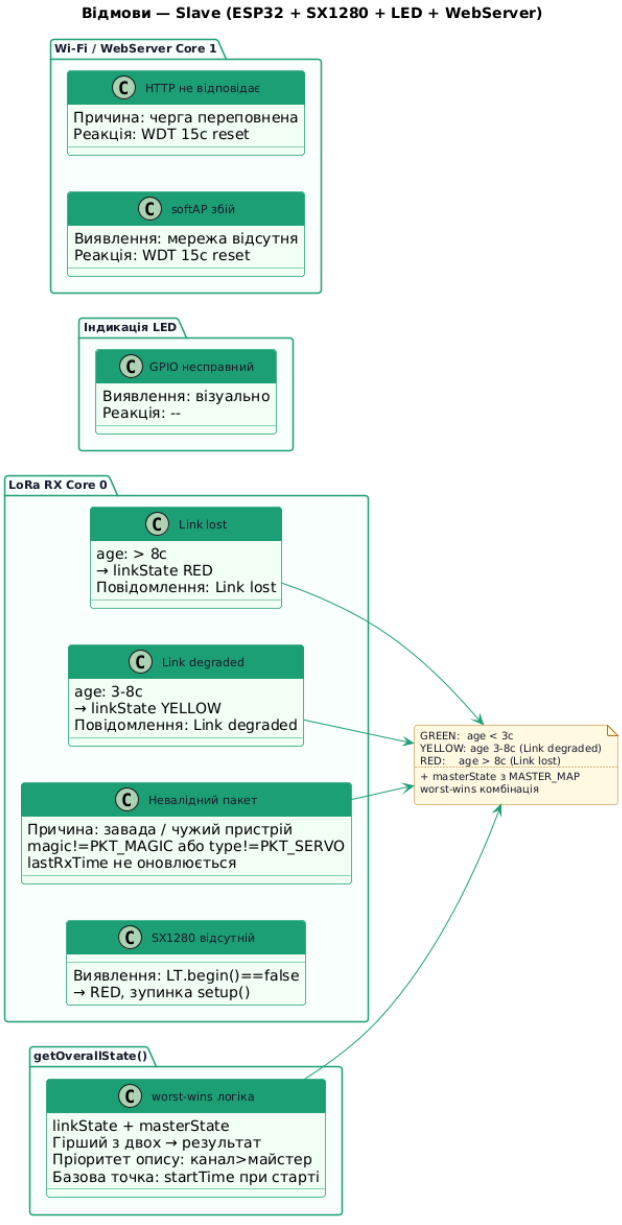

Рисунок Д.2 – Дерево відмов Slave

ДОДАТОК Е

СХЕМИ MASTER ТА SLAVE

На плакаті зображено схеми підключення модулів з описом вузлів. Master модуль має в собі більше периферії, тоді як слейв лише показники станів вигляді діодів та трансивер

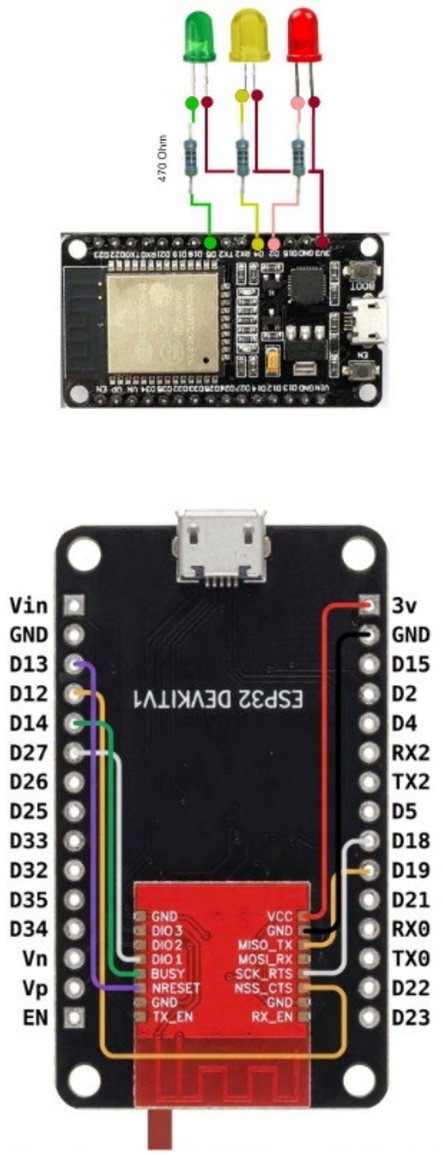

Рисунок Е.1 – Схема Slave та підключення LoRa трансивера

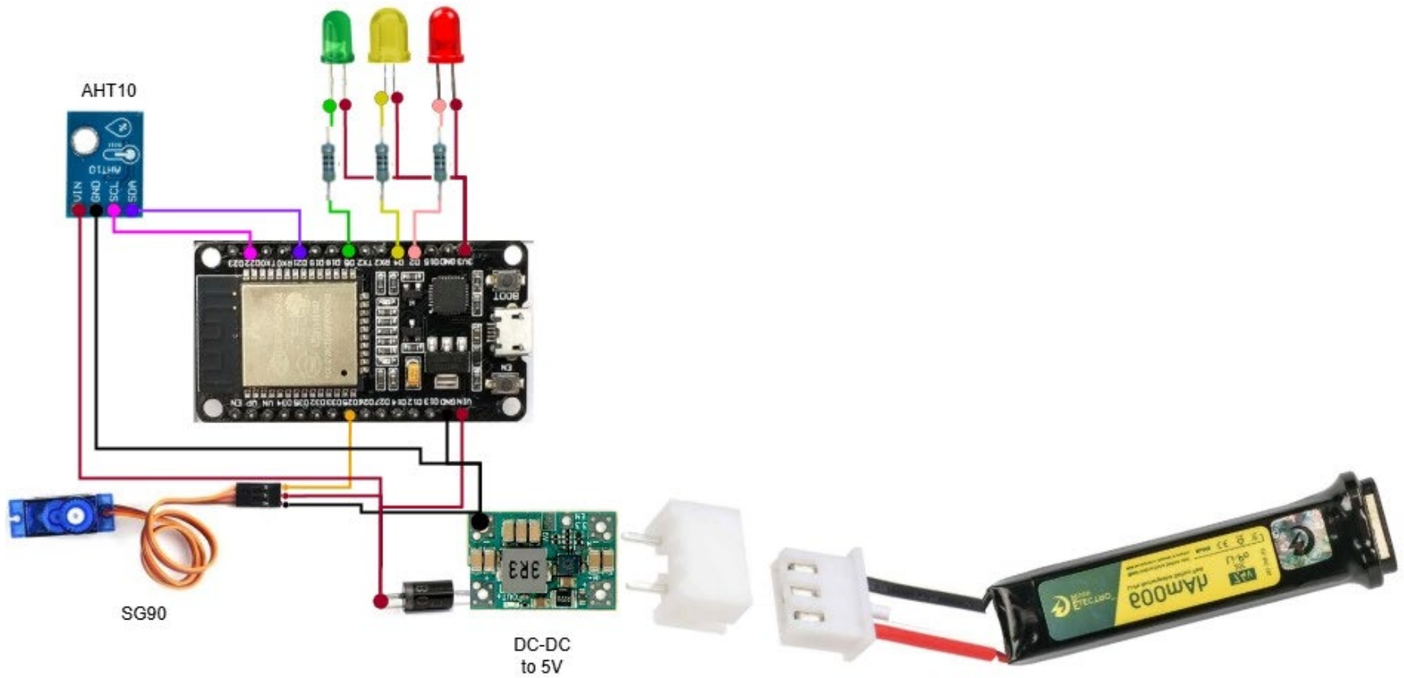

Рисунок Е.2 – Схема Master