МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ДЕРЖАВНИЙ УНІВЕРСИТЕТ «КИЇВСЬКИЙ АВІАЦІЙНИЙ ІНСТИТУТ»

ДОПУСТИТИ ДО ЗАХИСТУ
Завідувач кафедри

__________Олена НЕЧИПОРУК
«_____»____________2025 р.

КВАЛІФІКАЦІЙНА РОБОТА

(ПОЯСНЮВАЛЬНА ЗАПИСКА)

ЗДОБУВАЧА ВИЩОЇ ОСВІТИ
ОСВІТНЬОГО СТУПЕНЯ «МАГІСТР»

Тема: Програмний засіб віддаленого оновлення прошивки для ретрансляторів та модулів керування наземними станціями

Виконавець: ________________________________ Олександра ЗАДОРОЖНА

Керівник: ________________________________ Олександр ЛИТВИНЕНКО

Нормоконтролер: ________________________________ Євгеній ТУПОТА

Київ 2025

ДЕРЖАВНИЙ УНІВЕРСИТЕТ «КИЇВСЬКИЙ АВІАЦІЙНИЙ ІНСТИТУТ»

Факультет комп’ютерних наук та технологій
Кафедра інтелектуальних кібернетичних систем
Спеціальність 126 «Інформаційні системи та технології»
(шифр, найменування)

Освітньо-професійна програма «Інтелектуальні системи та технології»
Форма навчання денна

ЗАТВЕРДЖУЮ
Завідувач кафедри

____________ Олена НЕЧИПОРУК
«_____»___________________2025 р.

ЗАВДАННЯ
на виконання кваліфікаційного проєкту
Задорожної Олександри Віталіївни
(прізвище, ім'я, по батькові випускника в родовому відмінку)

1. Тема кваліфікаційної роботи (проєкту) «Програмний засіб віддаленого оновлення прошивки для ретрансляторів та модулів керування наземними станціями»
затверджена наказом ректора від « 28 » серпня 2025 р. № 1575/ст
2. Термін виконання роботи (проєкту): з 29. 09. 2025 по 31. 12.2025
3. Вихідні дані до роботи (проєкту): мова програмування *Python*, формат *JSON*, інтегроване середовище розробки *Visual Studio Code*, прикладний програмний інтерфейс.

4. Зміст пояснювальної записки:
1) аналіз та теоретичні основи віддаленого оновлення прошивки ретрансляторів та модулів керування наземних станціями;
2) засоби та технології розробки програмного засобу віддаленого оновлення прошивки;
3) реалізація та тестування програмного засобу віддаленого оновлення прошивки.

5. Перелік обов'язкового графічного (ілюстративного) матеріалу:
1) програмний засіб (схема алгоритму);
2) взаємодія компонентів програмного засобу;
3) *Use case* діаграма програмного засобу.

6. Календарний план-графік

№ пор.	Завдання	Термін виконання	Відмітка про виконання
1.	Ознайомитись з постановкою завдання на кваліфікаційну роботу	29.09.2025 – 01.10.2025	
2.	Дослідити спеціальну літературу	02.10.2025 – 04.10.2025	
3.	Проаналізувати область дослідження	05.10.2025 – 10.10.2025	
4.	Підготувати текст для першого розділу	12.10.2025 – 19.10.2025	
5.	Проаналізувати технологічні засоби розробки	20.10.2025 – 24.10.2025	
6.	Підготувати текст для другого розділу	26.11.2025 – 02.12.2025	
7.	Розробити схему алгоритм програмного засобу та графічний користувацький інтерфейс	03.12.2025 – 05.12.2025	
8.	Написати код програмного засобу	06.12.2025 – 16.12.2025	
9.	Підготувати текст для третього розділу	16.12.2025 – 20.12.2025	
10.	Оформити пояснювальну записку	21.12.2025 – 23.12.2025	
11.	Захистити кваліфікаційну роботу	26.12.2025	

7. Дата видачі завдання: “ 29 ” вересня 2025 р.

Керівник кваліфікаційного роботи (проєкту)________ Олександр ЛИТВИНЕНКО
(підпис керівника) (П.І.Б.)

Завдання прийняв до виконання _________________ Олександра ЗАДОРОЖНА

РЕФЕРАТ

Пояснювальна записка до кваліфікаційної роботи «Програмний засіб віддаленого оновлення прошивки ретрансляторів та модулів керування наземних станцій»: 80 с., 28 рис., 8 табл., 27 літературних джерел.

Об’єкт дослідження: процес віддаленого оновлення прошивки ретрансляторів та модулів керування наземних станцій.

Предмет дослідження: програмний засіб віддаленого оновлення прошивки ретрансляторів та модулів керування наземних станцій.

Мета кваліфікаційної роботи: розробка програмного засобу віддаленого оновлення прошивки ретрансляторів та модулів керування наземних станцій, який забезпечує надійне, безпечне та контрольоване розгортання оновлень у розподіленій системі без необхідності фізичного доступу до обладнання.

Методи дослідження: аналіз предметної області віддаленого оновлення прошивок у вбудованих системах, аналіз архітектури ретрансляторів і модулів керування наземних станцій, системний і порівняльний аналіз, моделювання архітектури програмних систем, розробка та експериментальне тестування програмного забезпечення.

Результати кваліфікаційної роботи можуть бути використані для обслуговування систем безпілотних літальних апаратів, наземних станцій управління та інших розподілених технічних комплексів із великою кількістю вбудованих пристроїв.

Наукова новизна полягає у поєднанні локального та віддаленого механізмів оновлення прошивки *ESP32* у межах єдиного клієнтського програмного засобу з графічним інтерфейсом користувача. Запропонований підхід дозволяє інтегрувати *OTA*-оновлення без використання сторонніх веб-інтерфейсів або командного рядка, що підвищує зручність і безпеку процесу оновлення.

ПРОГРАМНИЙ ЗАСІБ, *OTA*, *ESP32*, *JSON*, *PYTHON, GUI*, *UART*.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ ТА СКОРОЧЕНЬ 7
ВСТУП 8
РОЗДІЛ 1 АНАЛІЗ ТА ТЕОРЕТИЧНІ ОСНОВИ ВІДДАЛЕНОГО ОНОВЛЕННЯ ПРОШИВКИ РЕТРАНСЛЯТОРІВ ТА МОДУЛІВ КЕРУВАННЯ НАЗЕМНИХ СТАНЦІЯМИ 11
1.1. Поняття ретранслятора та його роль у системах зв’язку 11
1.2. Модулі керування наземними станціями 13
1.3. Мікроконтролер *ESP32* 15
1.4. Телеметрія у системах керування 18
1.5. Віддалене оновлення прошивки 24
1.6. Висновки до розділу 27
РОЗДІЛ 2 ЗАСОБИ ТА ТЕХНОЛОГІЇ РОЗРОБКИ ПРОГРАМНОГО ЗАСОБУ ВІДДАЛЕНОГО ОНОВЛЕННЯ ПРОШИВКИ 30
2.1. Загальна характеристика програмного середовища розробки *Visual Studio Code* 30
2.2. Мова програмування *Python* 33
2.3. Програмні бібліотеки та залежності програмного засобу 34
2.4. Засоби мережевої взаємодії та телеметрії 36
2.5. Технологія *Over-The-Air* 37
2.6. Формат обміну даними *JSON* 38
2.7. Робота з послідовними інтерфейсами 40
2.8. Вимоги до початкової прошивки *ESP32* для підтримки *OTA*-оновлення 42
2.9. Висновки до розділу 44

РОЗДІЛ 3 РЕАЛІЗАЦІЯ ТА ТЕСТУВАННЯ ПРОГРАМНОГО ЗАСОБУ ВІДДАЛЕНОГО ОНОВЛЕННЯ ПРОШИВКИ 47
3.1. Загальна структура програмного засобу 47
3.2. Розробка графічного інтерфейсу користувача 48
3.3. Виявлення та вибір підключених пристроїв 52
3.4. Реалізація локальної прошивки *ESP32* через інтерфейс *UART* 54
3.5. Робота з конфігураціями пристрою 57
3.6. Реалізація *OTA*-оновлення прошивки 59
3.7. Моніторинг та логування 60
3.8. Тестування та перевірка працездатності програмного засобу 60
3.9. Висновки до розділу 71
ВИСНОВКИ 74
СПИСОК БІБЛІОГРАФІЧНИХ ПОСИЛАНЬ ВИКОРИСТАНИХ ДЖЕРЕЛ. 78

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ ТА СКОРОЧЕНЬ

OTA	*Over-The-Air*
UART	*Universal Asynchronous Receiver-Transmitter*
COM-порт	Послідовний комунікаційний порт
SSID	*Service Set Identifier*
JSON	*JavaScript Object Notation*

ВСТУП

У сучасних умовах стрімкого розвитку інформаційних та телекомунікаційних технологій особливого значення набуває надійність і безперервність роботи розподілених систем керування та зв’язку. До таких систем належать ретранслятори, модулі керування наземними станціями, телеметричні вузли та інші вбудовані пристрої, які функціонують у складних умовах експлуатації та часто розміщуються у важкодоступних або віддалених локаціях.

Актуальність теми даної роботи суттєво зросла в умовах повномасштабної війни в Україні, коли значна кількість технічних засобів зв’язку, керування та моніторингу використовується в польових умовах. У таких ситуаціях фізичний доступ до обладнання є обмеженим або неможливим, а потреба в оперативному оновленні програмного забезпечення, виправленні помилок та адаптації функціоналу до нових умов виникає постійно. Відсутність ефективних механізмів віддаленого оновлення може призводити до зниження надійності систем, простоїв у роботі або повної втрати працездатності обладнання.

Особливої ваги набувають рішення, що дозволяють здійснювати оновлення прошивки мікроконтролерних пристроїв дистанційно, без необхідності їх демонтажу або транспортування до сервісних центрів. Використання механізмів віддаленого оновлення прошивки (*Over-The-Air, OTA*) дає змогу значно скоротити час реагування на виявлені проблеми, мінімізувати людський фактор та зменшити ризики, пов’язані з обслуговуванням техніки в небезпечних умовах.

Метою роботи є розробка програмного засобу для віддаленого оновлення прошивки мікроконтролерних пристроїв на базі *ESP32*, що використовуються у складі ретрансляторів і модулів керування наземними станціями, з метою підвищення надійності, зручності та оперативності їх експлуатації в умовах обмеженого фізичного доступу до обладнання.

Для досягнення поставленої мети в роботі необхідно було вирішити такі завдання:

– проаналізувати принципи роботи ретрансляторів, модулів керування наземними станціями та особливості їх експлуатації;
– дослідити можливості мікроконтролерів *ESP32* у контексті локального та віддаленого оновлення прошивки;
– обґрунтувати вибір програмних засобів, мов програмування та бібліотек для реалізації клієнтського застосунку;
– розробити архітектуру програмного засобу для локальної та *OTA*-прошивки *ESP32*;
– реалізувати механізми роботи з конфігураційними даними у форматі *JSON*;
– забезпечити зручний графічний інтерфейс користувача;
– провести тестування розробленого програмного засобу та проаналізувати його результати.

Об’єктом дослідження є процес оновлення програмного забезпечення мікроконтролерних пристроїв, що використовуються у складі ретрансляторів і модулів керування наземними станціями.

Предметом дослідження програмний засіб віддаленого оновлення прошивки пристроїв на базі *ESP32*, а також засоби керування цими процесами з боку клієнтського програмного забезпечення.

У процесі виконання роботи було використано такі методи дослідження:
– аналіз науково-технічної літератури та відкритих технічних джерел з питань *OTA*-оновлення та вбудованих систем;
– методи об’єктно-орієнтованого програмування;
– моделювання архітектури програмного забезпечення;
– експериментальні методи тестування програмного засобу;
– методи аналізу та обробки телеметричних і конфігураційних даних.

Наукова новизна роботи полягає у поєднанні локального та віддаленого механізмів оновлення прошивки *ESP32* у межах єдиного клієнтського програмного засобу з графічним інтерфейсом користувача. Запропонований підхід дозволяє

інтегрувати *OTA*-оновлення без використання сторонніх веб-інтерфейсів або командного рядка, що підвищує зручність і безпеку процесу оновлення.

Практичне значення отриманих результатів полягає у можливості використання розробленого програмного засобу для обслуговування мікроконтролерних пристроїв у реальних умовах експлуатації. Застосунок може бути використаний для первинної локальної прошивки *ESP32*, подальшого віддаленого *OTA*-оновлення прошивки, керування конфігураційними параметрами пристроїв, зменшення витрат часу та ресурсів на технічне обслуговування обладнання.

Результати роботи можуть бути використані у системах зв’язку, телеметрії та керування, зокрема в умовах обмеженого фізичного доступу до обладнання.

У першому розділі наведено аналіз предметної області та теоретичні основи віддаленого оновлення прошивки. Розглянуто принципи роботи ретрансляторів і модулів керування наземними станціями, особливості використання мікроконтролерів *ESP32*, роль телеметрії та значення механізмів *OTA*-оновлення в сучасних розподілених системах.

Другий розділ присвячено огляду програмних і апаратних засобів, що використовуються в процесі розробки програмного продукту. У розділі обґрунтовано вибір мови програмування, середовища розробки, бібліотек і форматів даних, а також розглянуто вимоги до базової прошивки *ESP32*, необхідної для реалізації механізмів локального та віддаленого оновлення прошивки.

У третьому розділі описано практичну реалізацію програмного засобу для локального та *OTA*-оновлення прошивки *ESP32*-пристроїв. Розглянуто архітектуру застосунку, реалізацію графічного інтерфейсу користувача, механізми прошивки, роботи з конфігураційними даними та результати функціонального тестування.

РОЗДІЛ 1
АНАЛІЗ ТА ТЕОРЕТИЧНІ ОСНОВИ ВІДДАЛЕНОГО ОНОВЛЕННЯ ПРОШИВКИ РЕТРАНСЛЯТОРІВ ТА МОДУЛІВ КЕРУВАННЯ НАЗЕМНИХ СТАНЦІЯМИ

1.1. Поняття ретранслятора та його роль у системах зв’язку

Ретранслятор – це електронний пристрій, призначений для прийому, підсилення, обробки та повторної передачі сигналу з метою збільшення дальності зв’язку або покращення якості переданих даних. У сучасних телекомунікаційних та керуючих системах ретранслятори широко застосовуються для забезпечення стабільного обміну інформацією між віддаленими компонентами системи.

У контексті наземних станцій ретранслятори виконують функцію проміжної ланки між кінцевими пристроями (датчиками, модулями керування, виконавчими механізмами) та центральними обчислювальними або керуючими вузлами. Вони забезпечують передавання команд керування, телеметричних даних та службової інформації на значні відстані, часто в умовах обмеженої пропускної здатності каналу або нестабільного радіозв’язку.

Надійність роботи ретранслятора безпосередньо впливає на функціонування всієї системи, тому актуальним є питання своєчасного оновлення його програмного забезпечення та конфігурації.

Основними функціями ретранслятора є:

– прийом радіосигналу на заданій частоті;
– фільтрація та підсилення сигналу;
– придушення шумів і завад;

– повторна передача сигналу на тій самій або іншій частоті;
– забезпечення стабільності параметрів зв'язку.

У цифрових системах ретранслятор може також виконувати додаткові функції:

– демодуляцію та повторну модуляцію сигналу;
– перевірку цілісності даних;
– буферизацію та маршрутизацію пакетів;
– адаптацію швидкості передавання.

Ретранслятори застосовуються у багатьох галузях таких як: професійні радіомережі, аварійно-рятувальні служби, диспетчерські системи, системи керування безпілотними апаратами, телеметричні комплекси, станції збору даних з віддалених об'єктів.

У всіх зазначених сферах актуальним є питання надійності, масштабованості та можливості віддаленого оновлення програмного забезпечення ретрансляторів.

Ретранслятори працюють у визначених частотних діапазонах залежно від призначення системи. Для наземних станцій та систем керування часто використовуються *ISM*-діапазони, оскільки вони не потребують ліцензування та підтримуються сучасними мікроконтролерами й радіомодулями.

У класичних ретрансляторах приймальна та передавальна частоти можуть бути однаковими або відрізнятися на фіксований зсув, що дозволяє одночасний прийом і передачу.

Під час роботи ретранслятора важливим аспектом є контроль гармонік та побічних випромінювань. Гармоніки – це сигнали, частота яких є кратною основній робочій частоті. Вони виникають унаслідок нелінійностей у підсилювальних каскадах [4].

Небажані гармоніки можуть створювати завади іншим радіосистемам, порушувати вимоги електромагнітної сумісності, знижувати ефективність роботи ретранслятора.

Для зменшення рівня гармонік застосовуються смугові та загороджувальні фільтри, лінійні підсилювачі, коректне налаштування вихідної потужності.

Програмне забезпечення ретранслятора також може впливати на спектральні характеристики шляхом керування режимами передавання та параметрами модуляції.

Ретранслятори можуть бути як аналоговими, так і цифровими. Характеристики цих видів ретрансляторів наведене в таблиці 1.1.

Таблиця 1.1

Характеристика аналогового та цифрового ретранслятора

Аналогові ретранслятори	Цифрові ретранслятори
Працюють з безперервними сигналами	Використовують цифрову модуляцію
Простіші за конструкцією	Забезпечують кращу завадостійкість
Чутливі до шумів і завад	Підтримують шифрування, корекцію помилок
Обмежені в можливостях керування та моніторингу	Дозволяють передавати телеметрію та керуючі дані

З розвитком вбудованих систем ретранслятори перетворилися з суто апаратних пристроїв на програмно-керовані системи. Використання мікроконтролерів, зокрема *ESP32*, дозволяє:

- динамічно змінювати параметри роботи;
- оновлювати прошивку та конфігурацію;
- реалізовувати телеметрію та віддалене керування;
- інтегрувати ретранслятори у клієнт-серверні системи.

Це робить задачу розробки програмного засобу для віддаленого оновлення прошивки ретрансляторів особливо актуальною.

1.2. Модулі керування наземними станціями

Модуль керування наземної станції – це апаратно-програмний компонент, що відповідає за прийом, обробку та виконання керуючих команд, а також за збір і передавання телеметричної інформації. Такі модулі можуть взаємодіяти з різними

периферійними пристроями: датчиками, радіомодулями, сервоприводами, індикаторами тощо.

Основними функціями модулів керування є:

- обробка сигналів від користувача або центральної системи;
- формування команд для керованих об'єктів;
- збір телеметричних даних;
- підтримка каналів зв'язку;
- забезпечення стабільної та безпечної роботи пристрою.

Програмне забезпечення модулів керування зазвичай зберігається у вбудованій *flash*-пам'яті мікроконтролера та може бути оновлене з метою виправлення помилок, розширення функціональності або адаптації до нових умов експлуатації.

Типовий модуль керування наземної станції складається з таких основних компонентів:

- мікроконтролер або процесорний блок, який виконує програмну логіку керування.
- модулі зв'язку, що забезпечують обмін даними з ретрансляторами та віддаленими пристроями.
- інтерфейси введення та виведення, призначені для взаємодії з датчиками, виконавчими механізмами та периферійними пристроями.
- підсистема живлення, що забезпечує стабільну роботу обладнання.
- пам'ять, у якій зберігається прошивка та конфігураційні дані.

У сучасних реалізаціях значна частина функціональності модуля керування визначається саме програмним забезпеченням.

Однією з ключових задач модуля керування є збір і передавання телеметричної інформації. Телеметрія дозволяє отримувати дані про поточний стан системи, параметри роботи пристроїв, наявність помилок або збоїв, версію програмного забезпечення, активні конфігурації.

Зібрані телеметричні дані можуть використовуватися для оперативного моніторингу, аналізу ефективності роботи та прийняття керуючих рішень.

Модулі керування тісно взаємодіють з ретрансляторами, які забезпечують передачу команд і телеметрії між наземною станцією та віддаленими об'єктами.

Модулі керування як і ретранслятори бувають аналоговими та цифровими. Характеристики цих типів модулів наведені в таблиці 1.2.

Таблиця 1.2

Характеристика аналогового та цифрового модуля керування наземною станцією

Аналогові модулі	Цифрові модулі
Використовують аналогові сигнали	Працюють з цифровими даними
Мають обмежену гнучкість	Підтримують складні алгоритми
Менш стійкі до завад	Забезпечують високу точність керування
Складні в масштабуванні	Дозволяють інтеграцію з мережевими системами

У сучасних системах наземні станції розглядаються як розподілені інформаційно-керуючі комплекси. Модулі керування в такій системі виконують роль автономних, інтелектуальних вузлів, здатних працювати як у складі мережі, так і в автономному режимі.

Їх програмна гнучкість, інтеграція з ретрансляторами та підтримка телеметрії створюють передумови для побудови ефективних і надійних систем керування.

1.3. Мікроконтролер *ESP32*

ESP32 – це високопродуктивний мікроконтролер із вбудованими засобами бездротового зв'язку, розроблений компанією *Espressif Systems* [14].

Він широко використовується у вбудованих системах, зокрема в ретрансляторах, модулях керування та *IoT*-пристроях.

Основні характеристики *ESP32*:

– двоядерний процесор з архітектурою *Xtensa*;

– вбудована *flash*-пам'ять та оперативна пам'ять;
– підтримка *Wi-Fi* та *Bluetooth*;
– наявність апаратних інтерфейсів (*UART*, *SPI*, *I²C*, *GPIO*);
– можливість роботи з файловими системами у *flash*-пам'яті.

Завдяки своїй універсальності та доступності *ESP32* є зручною платформою для реалізації складних вбудованих систем, що потребують гнучкого керування, мережевої взаємодії та можливості віддаленого оновлення програмного забезпечення.

Сімейство *ESP32* включає кілька серій, які відрізняються за архітектурою, продуктивністю та призначенням [8]:

ESP32 (класична серія) – двоядерні мікроконтролери з підтримкою *Wi-Fi* та *Bluetooth Classic/LE*.

ESP32-S2 – одноядерні мікроконтролери з акцентом на USB та зниженим енергоспоживанням.

ESP32-S3 – покращена серія з підтримкою векторних інструкцій, орієнтована на обробку даних і керування.

ESP32-C3/C6 – мікроконтролери на базі *RISC-V*, оптимізовані для *IoT*-застосувань.

Крім самих чипів, широко використовуються модулі *ESP32*, які містять мікроконтролер, *flash*-пам'ять, антену, елементи узгодження сигналу.

Це значно спрощує інтеграцію *ESP32* у готові пристрої.

ESP32 використовується у таких сферах: системи керування та автоматики, ретранслятори та радіомодулі, наземні станції керування, телеметричні системи, *IoT*-пристрої, промислові контролери, системи дистанційного моніторингу. Завдяки вбудованим засобам зв'язку *ESP32* дозволяє реалізовувати як локальне, так і віддалене керування пристроями.

Робота *ESP32* визначається прошивкою – програмним кодом, який зберігається у *flash*-пам'яті мікроконтролера [15]. Прошивка може включати:

– основний виконуваний код;
– драйвери апаратних інтерфейсів;

– мережеві стеки;

– файлову систему для збереження конфігурацій.

Для збереження налаштувань часто використовується файлова система *LittleFS*, яка дозволяє працювати з конфігураційними файлами у форматі *JSON*. Це забезпечує гнучкість та зручність оновлення параметрів роботи пристрою.

Керування *ESP32* може здійснюватися кількома способами:

– через послідовний інтерфейс (*UART*);

– через мережеві протоколи (*Wi-Fi*);

– через спеціалізовані протоколи обміну даними.

Під час розробки та обслуговування пристроїв широко використовується серійний монітор, який дозволяє отримувати службові повідомлення, передавати команди, контролювати процес завантаження.

Для прошивки *ESP32* застосовуються спеціалізовані інструменти, які забезпечують запис даних у *flash*-пам’ять, читання її вмісту та контроль цілісності запису.

Оновлення прошивки *ESP32* може виконуватися як локально через фізичне підключення, так і віддалено з використанням мережевих каналів.

Віддалене оновлення дозволяє швидко розповсюджувати нові версії ПЗ, усувати помилки без доступу до пристрою, централізовано керувати версіями прошивки.

Для підвищення надійності застосовуються механізми резервного копіювання та контролю коректності оновлення.

Переваги та недоліки використання мікроконтролерів *ESP32* у системах наземних станцій наведено у таблиці 1.3.

Таблиця 1.3

Переваги та недоліки використання мікроконтролерів *ESP32* у системах наземних станцій

Критерій	Переваги	Недоліки
Продуктивність	Достатня обчислювальна потужність для керування, телеметрії та мережевої взаємодії	Обмежені ресурси порівняно з повноцінними одноплатними комп’ютерами

Закінчення таблиці 1.3

Бездротовий зв'язок	Вбудована підтримка *Wi-Fi* та *Bluetooth* без додаткових модулів	Обмеження по дальності та стабільності без зовнішніх підсилювачів
Енергоспоживання	Підтримка режимів низького енергоспоживання	При активному *Wi-Fi* споживання може бути суттєвим
Гнучкість конфігурації	Програмне керування параметрами роботи та віддалене оновлення прошивки	Потребує ретельного налаштування структури *flash*-пам'яті
Вартість	Низька вартість мікроконтролерів і модулів	Обмежені можливості масштабування без зовнішніх компонентів
Інтеграція з ретрансляторами	Легко інтегрується з радіомодулями та датчиками	Потребує додаткових схем узгодження сигналів
Оновлення прошивки	Підтримка віддаленого оновлення та резервного копіювання	Існує ризик пошкодження прошивки при некоректному оновленні
Телеметрія та моніторинг	Можливість збору та передавання телеметричних даних у реальному часі	Обмежена пропускна здатність у порівнянні з провідними системами
Безпека	Підтримка *secure boot* і шифрування *flash*	Реалізація захисту потребує додаткових зусиль у програмуванні
Екосистема	Широка спільнота, велика кількість бібліотек та інструментів	Залежність від сторонніх бібліотек і *SDK*

Аналіз переваг і недоліків мікроконтролерів *ESP32* показує, що вони є доцільним вибором для побудови модулів керування та ретрансляторів у складі наземних станцій. Незважаючи на обмежені апаратні ресурси, *ESP32* забезпечує необхідний рівень продуктивності, гнучкості та інтеграції, що робить його ефективною платформою для реалізації систем віддаленого оновлення прошивки.

1.4. Телеметрія у системах керування

Телеметрія є невід'ємною складовою сучасних систем керування, зв'язку та моніторингу. Вона являє собою сукупність методів і засобів автоматизованого збору, передавання та аналізу інформації про стан об'єктів, що перебувають на відстані від центру керування [10]. У системах наземних станцій телеметрія

забезпечує інформаційний зв'язок між модулями керування, ретрансляторами та центральним програмним забезпеченням.

Основне призначення телеметрії полягає у забезпеченні оператора або автоматизованої системи керування актуальними даними про стан обладнання, параметри його роботи та результати виконання керуючих команд. Без телеметрії ефективне керування розподіленими системами стає неможливим, оскільки відсутній зворотний зв'язок між керуючим центром і віддаленими пристроями.

У системах з ретрансляторами та модулями керування телеметрія виконує подвійну функцію. З одного боку, вона дозволяє відстежувати технічний стан обладнання, з іншого – виступає інструментом підтвердження коректності передавання команд та оновлення програмного забезпечення.

Телеметричні дані можуть мати різну природу залежно від призначення системи. Найчастіше вони включають інформацію про апаратні характеристики пристрою, параметри зв'язку, стан програмного забезпечення та активну конфігурацію. До типових телеметричних параметрів належать:

- ідентифікаційні дані пристрою (*MAC*-адреса, тип мікроконтролера);
- версія прошивки та програмних модулів;
- параметри радіоканалу або мережевого з'єднання;
- дані про помилки та аварійні стани;
- поточні налаштування конфігурації.

Збирання телеметрії у сучасних системах зазвичай реалізується автоматично на рівні прошивки мікроконтролера. Пристрій періодично або за певною подією формує телеметричні пакети, які передаються до центральної системи через доступні канали зв'язку. Частота передавання телеметрії визначається вимогами до оперативності моніторингу та пропускною здатністю каналу.

Важливою особливістю телеметричних систем є необхідність балансування між обсягом переданих даних і навантаженням на канал зв'язку. Надмірна кількість телеметричних повідомлень може призводити до перевантаження мережі, тоді як недостатній обсяг даних ускладнює оперативне виявлення проблем. Тому на

практиці застосовуються адаптивні механізми, які змінюють інтенсивність телеметрії залежно від режиму роботи системи.

Телеметрія тісно пов’язана з процесами віддаленого оновлення прошивки. Під час оновлення програмного забезпечення телеметричні дані використовуються для:

- підтвердження початку та завершення процесу оновлення;
- контролю коректності запису даних у *flash*-пам’ять;
- виявлення збоїв або аварійних ситуацій;
- аналізу стану пристрою після перезавантаження.

Завдяки телеметрії центральний програмний засіб може не лише ініціювати оновлення, але й оцінювати його успішність без фізичного доступу до пристрою.

З погляду архітектури телеметричні системи найчастіше будуються на основі клієнт–серверної моделі. В цьому випадку віддалені пристрої виступають у ролі клієнтів, які надсилають телеметричні дані на сервер, а сервер забезпечує їх зберігання, обробку та візуалізацію. Такий підхід дозволяє централізовано керувати великою кількістю пристроїв і спрощує масштабування системи.

Особливу увагу при проєктуванні телеметрії приділяють питанням достовірності та цілісності даних. Телеметричні повідомлення повинні бути захищені від втрати, спотворення та несанкціонованого доступу. Для цього застосовуються механізми аутентифікації, шифрування та перевірки цілісності повідомлень.

У системах наземних станцій телеметрія також виконує аналітичну функцію. Зібрані дані можуть використовуватися для довготривалого аналізу роботи обладнання, прогнозування відмов та оптимізації режимів роботи. Таким чином, телеметрія стає не лише засобом моніторингу, а й інструментом підвищення надійності та ефективності всієї системи.

У таблиці 1.4 наведено основні типи телеметричних даних у системах наземних станцій.

Таблиця 1.4

Основні типи телеметричних даних у системах наземних станцій

Тип телеметрії	Зміст даних
Ідентифікаційна	*MAC*-адреса, тип пристрою, модель мікроконтролера
Програмна	Версія прошивки, дата оновлення, активні модулі
Конфігураційна	Параметри роботи, налаштування режимів
Станова	Статус з'єднання, помилки, аварійні події
Апаратна	Дані про чип, пам'ять, режими роботи
Службова	Логи, службові повідомлення

Узагальнюючи, можна зазначити, що телеметрія є ключовим елементом сучасних систем керування наземних станцій і ретрансляторів. Вона забезпечує зворотний зв'язок, дозволяє реалізувати віддалене оновлення прошивки, підтримує моніторинг стану пристроїв та створює передумови для автоматизації процесів керування. Саме тому розробка ефективних телеметричних механізмів є важливою складовою сучасних програмно-апаратних комплексів.

1.4.1. Протоколи передавання телеметрії в системах наземних станцій

Передавання телеметричних даних є критично важливим елементом функціонування систем керування наземних станцій і ретрансляторів. Ефективність телеметрії значною мірою визначається вибором протоколів обміну даними, які забезпечують доставку інформації з необхідною надійністю, швидкістю та рівнем безпеки.

Протокол телеметрії – це сукупність правил, які визначають формат повідомлень, порядок обміну даними, механізми підтвердження доставки та

обробки помилок. У системах наземних станцій протоколи телеметрії повинні враховувати обмеження вбудованих пристроїв, нестабільність каналів зв'язку та вимоги до оперативності передавання даних.

У практиці побудови таких систем застосовуються як низькорівневі, так і прикладні протоколи, кожен з яких має свою сферу доцільного використання.

На нижньому рівні телеметрія може передаватися через послідовні або радіоканали без складної мережевої інфраструктури. У таких випадках часто використовується *UART* або спеціалізовані радіопротоколи.

Передавання телеметрії на цьому рівні характеризується:

– мінімальними накладними витратами;
– високою швидкістю обробки;
– залежністю від фізичного середовища зв'язку.

Недоліком такого підходу є відсутність вбудованих механізмів маршрутизації, масштабування та захисту даних, що обмежує його застосування у великих системах.

У сучасних системах наземних станцій телеметрія часто передається через мережеві канали з використанням *TCP/IP* або *UDP/IP*. Це дозволяє інтегрувати вбудовані пристрої у клієнт-серверні архітектури та використовувати стандартну мережеву інфраструктуру.

Передавання телеметрії на мережевому рівні забезпечує:

– централізований збір даних;
– гнучке масштабування системи;
– можливість інтеграції з іншими інформаційними системами.

Вибір між *TCP* і *UDP* залежить від вимог до надійності та затримок. *TCP* забезпечує гарантовану доставку, але має більші накладні витрати, тоді як *UDP* дозволяє зменшити затримки ціною можливих втрат пакетів.

На прикладному рівні широко застосовуються протоколи, спеціально призначені для телеметрії та *IoT*-систем.

HTTP/HTTPS, які забезпечують простоту реалізації та сумісність з веб-сервісами;

MQTT, оптимізований для передачі невеликих телеметричних повідомлень;

WebSocket, що дозволяє організувати двонаправлений обмін даними в реальному часі.

У системах керування наземними станціями часто обирається *HTTP* або *HTTPS*, оскільки ці протоколи добре підходять для періодичного передавання телеметрії та легко інтегруються з серверними компонентами.

Незалежно від вибору протоколу, важливим аспектом є формат телеметричних повідомлень. Найчастіше використовується текстовий формат *JSON*, який забезпечує:

- читабельність даних;
- гнучкість структури;
- простоту обробки на сервері.

Телеметричні повідомлення зазвичай містять ідентифікаційні дані пристрою, параметри стану та додаткову службову інформацію. Такий підхід дозволяє уніфікувати обмін даними між різними типами пристроїв.

1.4.2. Надійність, безпека та контроль доставлення телеметрії

Одним з ключових вимог до телеметричних протоколів є надійність доставки даних. У системах наземних станцій застосовуються різні підходи до її забезпечення, зокрема повторна передача даних у разі помилки, підтвердження отримання повідомлень, буферизація телеметрії на пристрої.

Ці механізми дозволяють зменшити втрати інформації навіть у разі нестабільного зв’язку або тимчасових збоїв.

Телеметричні дані можуть містити чутливу інформацію про стан системи, тому важливо забезпечити їх захист. Основними заходами безпеки є:

- аутентифікація пристроїв;
- використання захищених каналів зв’язку;
- обмеження доступу до телеметричних сервісів.

У сучасних системах все частіше використовується шифрування на транспортному або прикладному рівні, що унеможливлює перехоплення або підміну телеметричних даних.

На рисунку 1.1. зображена структурну схему телеметрії в наземній станції.

Рис. 1.1. Структурна схема телеметрії в наземній станції

1.5. Віддалене оновлення прошивки

Віддалене оновлення прошивки – це процес заміни або модифікації програмного забезпечення пристрою без необхідності фізичного доступу до нього. Для систем з великою кількістю ретрансляторів і модулів керування цей підхід є критично важливим, оскільки дозволяє:

– оперативно усувати помилки;
– централізовано керувати версіями програмного забезпечення;
– зменшувати витрати на обслуговування;
– підвищувати загальну надійність системи.

Реалізація механізмів віддаленого оновлення потребує ретельного опрацювання питань безпеки, цілісності даних та контролю процесу прошивки, що робить дану задачу актуальною та науково обґрунтованою для магістерської роботи.

Віддалене оновлення прошивки є одним із найважливіших механізмів супроводу сучасних вбудованих систем, до яких належать ретранслятори та модулі керування наземних станцій. Прошивка визначає логіку роботи пристрою, алгоритми обробки сигналів, взаємодію з периферійними модулями та протоколами зв'язку. У процесі експлуатації системи виникає потреба в її модифікації, розширенні функціональності або усуненні виявлених помилок, що робить оновлення програмного забезпечення неминучим.

Традиційний підхід до оновлення прошивки передбачає фізичний доступ до пристрою та його перепрограмування через локальний інтерфейс. Такий спосіб є трудомістким, затратним і практично непридатним для систем із великою кількістю віддалених або важкодоступних вузлів. У випадку наземних станцій, які можуть включати десятки або сотні ретрансляторів і модулів керування, фізичне обслуговування кожного пристрою значно ускладнює експлуатацію системи.

Віддалене оновлення прошивки дозволяє здійснювати зміну програмного забезпечення без фізичного втручання, використовуючи наявні канали зв'язку. Такий підхід забезпечує централізоване керування версіями прошивки, скорочує час реагування на помилки та знижує експлуатаційні витрати. Крім того, віддалене оновлення створює умови для постійного розвитку системи без необхідності її апаратної модернізації.

Порівняння локального та віддаленого оновлення прошивки наведене у таблиці 1.5.

Таблиця 1.5

Порівняння локального та віддаленого оновлення прошивки

Критерій	Локальне оновлення	Віддалене оновлення
Необхідність фізичного доступу	Обов'язкова	Відсутня

Закінчення таблиці 1.5

Масштабованість	Низька	Висока
Час обслуговування	Значний	Мінімальний
Витрати на експлуатацію	Високі	Знижені
Оперативність виправлень	Обмежена	Висока
Ризик простою системи	Підвищений	Знижений
Централізований контроль	Відсутній	Присутній

Процес віддаленого оновлення прошивки зазвичай включає кілька етапів. Спочатку виконується перевірка стану пристрою та його готовності до оновлення. Далі відбувається передавання нового програмного коду або окремих компонентів прошивки через мережевий або радіоканал.

Після запису даних у *flash*-пам’ять здійснюється перевірка цілісності та перезапуск пристрою з новою версією програмного забезпечення. У разі виникнення помилки система повинна мати можливість відновлення до попереднього працездатного стану.

Однією з основних вимог до систем віддаленого оновлення є надійність. Некоректне оновлення прошивки може призвести до повної втрати працездатності пристрою, що особливо критично для систем керування та зв’язку.

Тому в сучасних рішеннях широко застосовуються механізми захисту, такі як резервне копіювання прошивки, перевірка контрольних сум, поетапне оновлення та підтвердження успішного завершення процесу.

Важливим аспектом є безпека віддаленого оновлення. Прошивка повинна надходити лише з довірених джерел, а сам процес оновлення має бути захищений від несанкціонованого доступу. Для цього використовуються механізми аутентифікації, шифрування та перевірки цифрових підписів. Безпека оновлення є критичною умовою для запобігання втручанню в роботу системи або компрометації її компонентів.

Віддалене оновлення прошивки тісно пов’язане з телеметрією. Телеметричні дані дозволяють відстежувати стан пристрою до, під час та після оновлення,

забезпечуючи зворотний зв’язок між віддаленим вузлом і центральною системою керування. Завдяки цьому оператор або автоматизована система можуть своєчасно виявляти помилки та приймати рішення щодо подальших дій.

Актуальність віддаленого оновлення прошивки зростає зі збільшенням масштабів систем наземних станцій та ускладненням їхньої структури. Сучасні системи характеризуються високим рівнем програмної складності, що потребує регулярних оновлень і модифікацій. Без механізмів віддаленого оновлення підтримка таких систем стає практично неможливою.

Крім того, віддалене оновлення прошивки дозволяє впроваджувати нові функціональні можливості, оптимізувати алгоритми роботи та адаптувати систему до змінних умов експлуатації. Це робить систему більш гнучкою та стійкою до зовнішніх впливів.

Таким чином, віддалене оновлення прошивки є невід’ємним елементом сучасних систем керування наземних станцій і ретрансляторів. Воно забезпечує безперервність розвитку програмного забезпечення, підвищує надійність і безпеку системи та суттєво знижує витрати на її обслуговування. Саме ці фактори зумовлюють високу актуальність розробки програмних засобів, орієнтованих на підтримку віддаленого оновлення прошивки у вбудованих системах.

1.6. Висновки до розділу

У першому розділі роботи було проведено аналіз теоретичних основ віддаленого оновлення прошивки ретрансляторів та модулів керування наземних станцій. Розглянуто загальні принципи побудови таких систем, їхню архітектуру та особливості функціонування в умовах розподіленої інфраструктури.

Детально проаналізовано призначення та роль ретрансляторів у системах зв’язку, визначено їх основні функції, сфери застосування та технічні особливості, зокрема використання аналогових і цифрових методів обробки сигналів. Показано, що ретранслятори є критично важливими елементами систем наземних станцій, від стабільності роботи яких залежить якість зв’язку та ефективність керування.

У розділі також розглянуто модулі керування наземних станцій як центральні вузли, що забезпечують координацію роботи обладнання, взаємодію з ретрансляторами та обробку телеметричних даних. Проаналізовано їхню функціональну роль у забезпеченні надійності, масштабованості та гнучкості системи керування.

Окрему увагу приділено мікроконтролерам *ESP32* як апаратній платформі для реалізації модулів керування та ретрансляторів. Розглянуто їх архітектуру, функціональні можливості, сфери застосування, а також переваги й обмеження використання в системах наземних станцій.

Проведений аналіз підтвердив доцільність застосування *ESP32* у подібних системах завдяки поєднанню достатньої обчислювальної потужності, підтримки бездротового зв’язку та можливості віддаленого оновлення прошивки.

Значну увагу в роботі було приділено телеметрії як основному механізму зворотного зв’язку в системах керування. Проаналізовано види телеметричних даних, режими їх формування та передавання, а також роль телеметрії у процесах моніторингу стану пристроїв і віддаленого оновлення прошивки. Показано, що телеметрія є ключовим елементом забезпечення надійності та безпеки системи.

Окремо розглянуто процес віддаленого оновлення прошивки, його етапи, вимоги та актуальність у сучасних умовах. Показано, що застосування механізмів віддаленого оновлення дозволяє:

– зменшити експлуатаційні витрати;
– підвищити оперативність обслуговування;
– забезпечити безперервний розвиток програмного забезпечення без фізичного доступу до пристроїв.

Важливим аспектом, розглянутим у межах теоретичного аналізу, є питання надійності та відмовостійкості систем віддаленого оновлення прошивки. Підкреслено необхідність використання механізмів контролю цілісності програмного коду, резервного зберігання попередніх версій прошивки та можливості автоматичного відновлення працездатного стану пристрою у разі виникнення помилок під час оновлення. Такі підходи є особливо актуальними для

систем наземних станцій, що функціонують у безперервному режимі та часто розміщені у важкодоступних або віддалених локаціях.

Ефективна реалізація *OTA*-механізмів потребує адаптивних алгоритмів обміну даними та тісної інтеграції з телеметричними системами. Це дозволяє забезпечити контроль перебігу оновлення, своєчасне виявлення помилок та прийняття рішень щодо повторної спроби або відкату прошивки.

Таким чином, проведений у першому розділі аналіз підтвердив актуальність розробки програмного засобу для віддаленого оновлення прошивки ретрансляторів і модулів керування наземних станцій.

Отримані теоретичні положення слугували основою для проєктування архітектури та реалізації програмного комплексу, опис яких наведено в наступному розділі роботи.

РОЗДІЛ 2
ЗАСОБИ ТА ТЕХНОЛОГІЇ РОЗРОБКИ ПРОГРАМНОГО ЗАСОБУ ВІДДАЛЕНОГО ОНОВЛЕННЯ ПРОШИВКИ

2.1. Загальна характеристика програмного середовища розробки *Visual Studio Code*

Розробка програмного засобу для віддаленого оновлення прошивки ретрансляторів і модулів керування наземних станцій потребує використання сучасних програмних інструментів, які забезпечують стабільність, масштабованість та зручність супроводу системи. З огляду на вимоги до функціональності, надійності та кросплатформності, у процесі розробки було обрано набір технологій, що дозволяє ефективно реалізувати як клієнтську частину програмного забезпечення, так і механізми взаємодії з вбудованими пристроями.

Програмний засіб реалізовано у вигляді *desktop*-застосунку, що працює під керуванням операційних систем сімейства *Windows*. Такий підхід дозволяє забезпечити прямий доступ до апаратних ресурсів комп'ютера, зокрема послідовних портів, а також реалізувати механізми керування прошивкою та моніторингу пристроїв у режимі реального часу.

Visual Studio Code – це легке кросплатформне середовище розробки від *Microsoft (Windows / Linux / macOS)*, яке поєднує редактор коду, систему розширень і зручні інструменти для налагодження, запуску та керування проєктом [17].

Основні можливості:

Підсвітка синтаксису (рис. 2.1.), автодоповнення, переходи до визначення, пошук посилань, рефакторинг.

```
1    import sys
2    import os
```

Рис. 2.1. Підсвітка синтаксису у *Visual Studio Code*

Швидкий пошук по файлах/символах, *«Go to Definition»*, *«Find All References»*.

Вбудований термінал, можливий запуск *python, pip, pytest*, скриптів, збірок і т.д. прямо в *IDE* (рис.2.2.).

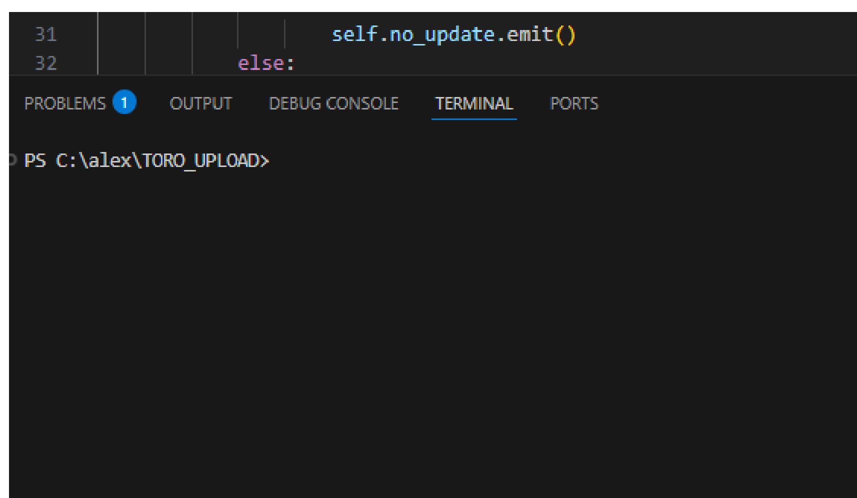


Рис. 2.2. Вбудований термінал у *Visual Studio Code*

Налагодження (*debug*), *breakpoints, step-by-step*, перегляд змінних, *call stack*. Для *Python* – через розширення *Python*.

Git-інтеграція, *commit*, *diff*, *branch*, *merge*, *history*, робота з *GitHub* (рис. 2.3.).

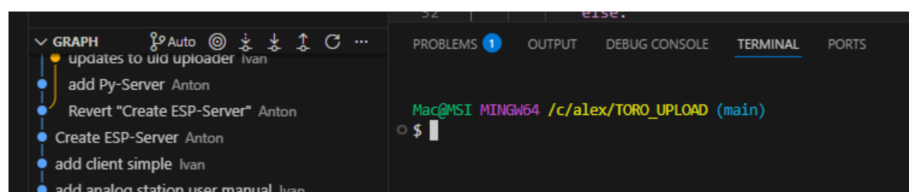


Рис. 2.3. *Git*-інтеграція у *Visual Studio Code*

Tasks/Launch configs: запуск типових команд через *tasks.json* і *launch.json*.

Workspace – зручна робота з проєктом як з простором (рис. 2.4.).

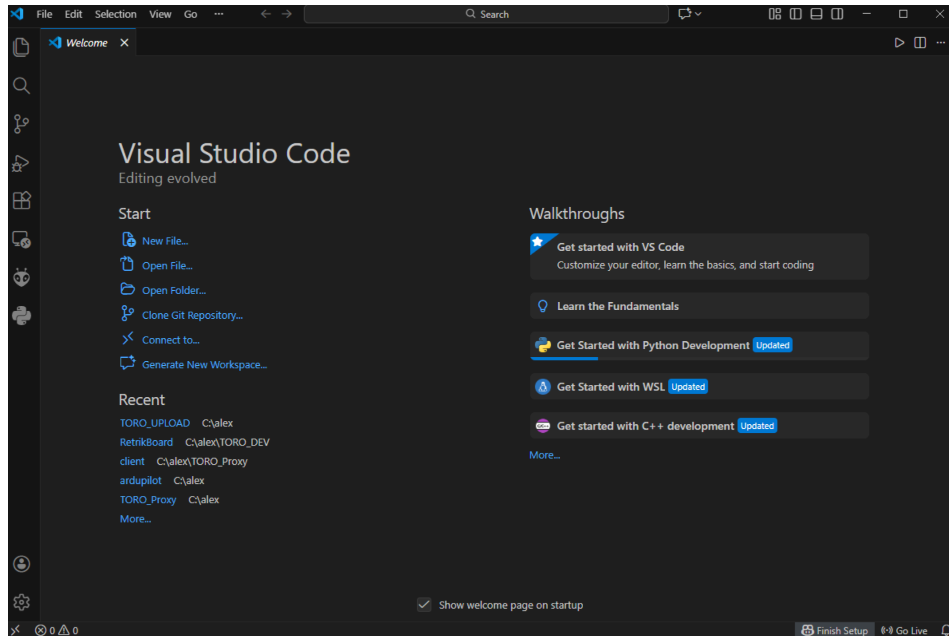


Рис. 2.4. Розгортання нового *Workspace* у *Visual Studio Code*

Розширення – практично будь-які мови/фреймворки/формати й інструменти (рис. 2.5.)

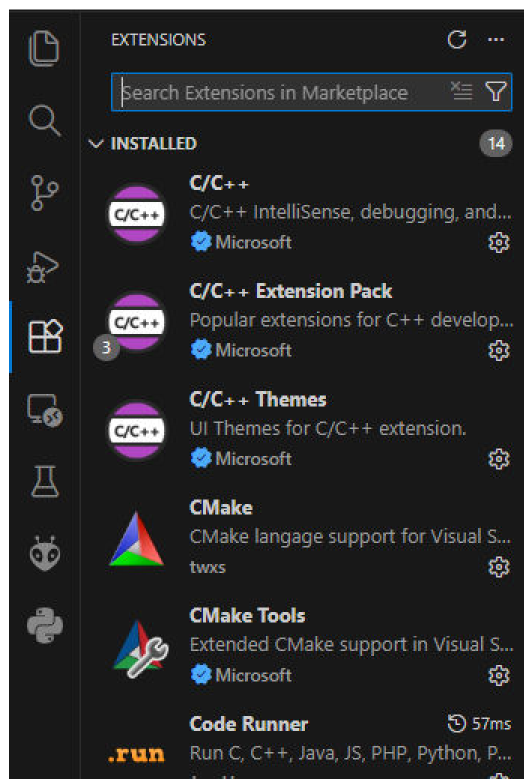


Рис. 2.5. Вікно з розширеннями у *Visual Studio Code*

2.2. Мова програмування *Python*

Основною мовою програмування, використаною в роботі, є *Python*. Вибір цієї мови зумовлений її високим рівнем абстракції, широкою екосистемою бібліотек та зручністю розробки програмних засобів різної складності. *Python* дозволяє швидко реалізовувати як прикладну логіку, так і засоби взаємодії з апаратним забезпеченням [18].

Важливою перевагою *Python* є можливість роботи з послідовними інтерфейсами, мережевими протоколами та файловими системами, що є критично важливим для задач прошивки та телеметрії. Крім того, *Python* добре підходить для створення графічних інтерфейсів користувача та багатопотокових застосунків.

До основних переваг використання *Python* у даному проєкті належать:

- простота синтаксису та читабельність коду;
- наявність великої кількості готових бібліотек;
- підтримка багатопотоковості та асинхронної обробки;
- зручність інтеграції з зовнішніми інструментами та утилітами.

Наведена таблиця 2.1 демонструє відмінності синтаксису мови *Python* у порівнянні з мовами *C++* та *Java*. *Python* характеризується мінімальною кількістю службових конструкцій, відсутністю обов'язкового оголошення типів змінних та більш компактним записом базових програмних конструкцій, що суттєво спрощує розробку та супровід програмного забезпечення.

Таблиця 2.1

Порівняння синтаксису мови *Python* з мовами *C++* та *Java*

Завдання	*Python*	*C++*	*Java*
Виведення тексту	*print("Hello")*	*std::cout << "Hello";*	*System.out.println("Hello");*
Умова *if*	*if x > 5: print(x)*	*if (x > 5) { std::cout << x;}*	*if (x > 5) { System.out.println(x);}*

Закінчення таблиці 2.1

Цикл *for*	*for i in range(5):\n print(i)*	*for (int i=0; i<5; i++) {\n std::cout << i;\n}*	*for (int i=0; i<5; i++) {\n System.out.println(i);\n}*

2.3. Програмні бібліотеки та залежності програмного засобу

Для коректної роботи програмного засобу для віддаленого оновлення прошивки ретрансляторів і модулів керування наземних станцій необхідно встановити низку програмних бібліотек, перелік яких наведено у файлі *requirements.txt*. Даний файл використовується для автоматизованого керування залежностями проєкту та забезпечує відтворюваність середовища розробки й виконання програми.

Кожна бібліотека виконує окрему функціональну роль у структурі програмного засобу.

esptool є спеціалізованою утилітою для роботи з мікроконтролерами сімейства *ESP*, зокрема *ESP32*. Вона забезпечує безпосередню взаємодію з *flash*-пам’яттю пристрою через послідовний інтерфейс.

У межах даного програмного засобу бібліотека *esptool* використовується для:

- запису прошивки у *flash*-пам’ять мікроконтролера;
- зчитування даних з *flash*-пам’яті;
- керування режимами перезавантаження пристрою;
- контролю процесу прошивки та отримання статусних повідомлень.

Застосування *esptool* дозволяє реалізувати повноцінний програматор *ESP32* без використання зовнішніх апаратних засобів.

Бібліотека *littlefs-python* призначена для роботи з файловою системою *LittleFS*, яка широко використовується у вбудованих системах для зберігання конфігураційних файлів [19].

У програмному засобі ця бібліотека використовується для:

– аналізу вмісту файлової системи, зчитаної з *flash*-пам’яті *ESP32*;
– модифікації конфігураційних файлів (*config.json* та інших);
– генерації нового образу файлової системи *LittleFS*;
– збереження конфігурацій без повного перезапису прошивки.

Використання *LittleFS* дозволяє розділити програмний код і конфігураційні дані, що значно підвищує гнучкість системи.

PyJWT – бібліотека для роботи з *JSON Web Token (JWT)*, які використовуються для автентифікації та авторизації користувачів.

У даному програмному засобі *PyJWT* застосовується для:

– обробки токенів доступу, отриманих після авторизації користувача;
– підтвердження прав доступу до серверних *API*;
– розмежування прав користувачів (адміністратор, звичайний користувач).

Це забезпечує базовий рівень безпеки взаємодії між *desktop*-клієнтом і серверною частиною системи.

PyQt6 є фреймворком для створення графічних інтерфейсів користувача на мові *Python*. Він базується на бібліотеці *Qt* та забезпечує широкі можливості для розробки desktop-застосунків.

У межах проєкту *PyQt6* використовується для:

– створення головного вікна програми;
– реалізації форм вибору прошивки та конфігурацій;
– відображення журналу подій і прогресу прошивки;
– організації багатовіконної взаємодії (монітор, налаштування, адмін-панель);
– реалізації багатопотоковості через *QThread*.

PyQt6 є ключовим компонентом клієнтської частини програмного засобу. Дана бібліотека містить безпосередньо бінарні компоненти *Qt6*, необхідні для роботи *PyQt6*. Вона включає модулі рендерингу, системи подій та базові компоненти *GUI*.

Без встановлення цього пакета коректна робота графічного інтерфейсу неможлива.

PyQt6_sip забезпечує механізм зв’язку між *Python*-кодом і нативними компонентами бібліотеки *Qt*. Він відповідає за коректну інтеграцію *PyQt6* з інтерпретатором *Python*.

Цей компонент є внутрішньою технічною залежністю, необхідною для стабільної роботи *PyQt6*-застосунку.

Бібліотека *pyserial* використовується для роботи з послідовними портами комп’ютера.

У програмному засобі вона застосовується для:

- визначення доступних *COM*-портів;
- встановлення з’єднання з *ESP32*;
- обміну даними через *UART*;
- реалізації серійного монітора для перегляду логів пристрою.

pyserial є ключовою бібліотекою для локальної прошивки та діагностики пристроїв.

Отже, файл *requirements.txt* визначає повний перелік програмних залежностей, необхідних для функціонування *desktop*-застосунку. Використання даного файлу дозволяє швидко розгорнути програмний засіб на новому робочому місці та гарантує узгодженість середовища виконання.

2.4. Засоби мережевої взаємодії та телеметрії

Для реалізації взаємодії з серверною частиною системи використовуються мережеві *HTTP*-запити. Бібліотека *requests* забезпечує простий і надійний механізм обміну даними між desktop-клієнтом та сервером, зокрема для отримання списків прошивок, конфігурацій та передавання телеметричних даних.

Для реалізації механізмів авторизації застосовується бібліотека *PyJWT*, яка дозволяє працювати з токенами доступу та забезпечувати розмежування прав користувачів.

2.5. Технологія *Over-The-Air*

У сучасних системах керування вбудованими пристроями дедалі більшого поширення набуває технологія *OTA* (*Over-The-Air*), яка забезпечує можливість віддаленого оновлення прошивки без фізичного підключення до пристрою. Для систем наземних станцій, що можуть включати значну кількість ретрансляторів і модулів керування, *OTA* є перспективним напрямом розвитку, який дозволяє зменшити експлуатаційні витрати та підвищити рівень автоматизації.

Мікроконтролери сімейства *ESP32* мають вбудовану апаратну та програмну підтримку *OTA*-оновлень. Архітектура *flash*-пам’яті *ESP32* передбачає використання декількох розділів (*partitions*), що дозволяє зберігати активну та резервну версії прошивки. У разі некоректного завершення оновлення пристрій може автоматично повернутися до попередньої стабільної версії програмного забезпечення, що суттєво підвищує надійність системи.

Процес *OTA*-оновлення прошивки включає кілька послідовних етапів. На початковому етапі пристрій встановлює мережеве з’єднання з сервером оновлень та перевіряє наявність нової версії прошивки. У разі виявлення оновлення відбувається завантаження відповідного файлу прошивки через захищений канал зв’язку. Далі нова версія записується у резервний розділ *flash*-пам’яті з обов’язковою перевіркою цілісності даних. Завершальним етапом є перезавантаження пристрою та запуск оновленої прошивки.

Важливою складовою *OTA* є забезпечення безпеки процесу оновлення. Для цього застосовуються механізми автентифікації сервера, перевірки цифрового підпису прошивки та шифрування каналу передавання даних. Недотримання вимог безпеки може призвести до несанкціонованого втручання у роботу пристрою або повної втрати контролю над системою.

2.6. Формат обміну даними *JSON*

JSON (*JavaScript Object Notation*) – це текстовий формат подання даних, призначений для зручного обміну структурованою інформацією між програмними системами. Попри походження з екосистеми *JavaScript*, *JSON* є мовонезалежним стандартом і підтримується більшістю сучасних мов програмування та платформ. Його ключові властивості – компактність, читабельність, простота парсингу та природна відповідність типам даних, що використовуються у програмуванні.

У межах розробленого програмного засобу *JSON* застосовується як універсальний формат для:

- зберігання конфігурацій пристроїв;
- передачі даних між *desktop*-клієнтом і сервером через *HTTP API*;
- формування телеметричних повідомлень;
- реалізації віддалених сценаріїв оновлення, де потрібні метадані версій, адреси завантаження, контроль цілісності тощо.

JSON-документ складається з двох базових структур. Об'єкт – набір пар «ключ-значення» у фігурних дужках *{ }*:

```
{
  "deviceType": "repeater",
  "enabled": true,
  "txPower": 20
}
```

Масив – упорядкований список значень у квадратних дужках *[ ]*:

```
{
  "channels": [1, 6, 11]
}
```

JSON підтримує обмежений, але достатній для більшості задач набір типів.

string – рядок у подвійних лапках *"..."*;

number – число без лапок;

boolean – *true* або *false*;

null – відсутність значення;

object – вкладена структура *{...}*;

array – список *[...]*.

Використання різних типів виглядатиме так:

```
{
  "ssid": "MyWiFi",
  "retryCount": 5,
  "timeoutSec": 2.5,
  "useDhcp": true,
  "staticIp": null,
  "allowedHosts": ["192.168.1.10", "192.168.1.11"]
}
```

JSON має строгий синтаксис. У *JSON* дозволені лише подвійні лапки *"..."*. Одинарні *'...'* – помилка. Після останнього елемента не може бути зайвої коми. Коментарі заборонені, *JSON* не підтримує *//* або */* */*. Ключами можуть бути лише рядки. *JSON* у сучасних системах зазвичай використовується з *UTF-8*, що важливо для українських символів.

JSON особливо зручний для конфігурацій, бо дозволяє описувати вкладені параметри (мережа, модулі, радіо, моніторинг), легко зберігати та передавати через *HTTP*, його можна серіалізувати/десеріалізувати стандартними засобами *Python*.

В програмному засобі для віддаленого оновлення прошивки конфігурація може зберігатися в *LittleFS* як *config.json* або розбитими файлами типу *wifi.json*, *radio.json* або бути вшитою/запатченою у визначену область *flash*.

Це дає гнучкість: прошивка може бути однакова, а поведінка пристрою змінюється конфігураційним файлом.

Переваги та недоліки *JSON* в контексті систем прошивки наведені в таблиці 2.2.

Таблиця 2.2

Переваги та недоліки *JSON* в контексті систем прошивки

Переваги	Недоліки
Простота читання і редагування	Відсутність коментарів
Мінімальна вага та швидка передача мережею	Суворий синтаксис
Широка підтримка в *Python*, *C/C++*, *JS*	Не містить багато типів напряму – все потрібно узгоджувати
Добре підходить для конфігураційні як дані, без жорсткої прив’язки до коду	При ручному редагуванні підвищується ризик помилок користувача

2.7. Робота з послідовними інтерфейсами

Послідовний інтерфейс є одним із найбільш поширених способів обміну даними між обчислювальними системами та вбудованими пристроями. Його принцип роботи полягає у передаванні інформації послідовно, біт за бітом, по одному або декількох фізичних провідниках. Такий підхід відрізняється простотою реалізації, надійністю та широкою апаратною підтримкою.

У системах на базі мікроконтролерів, зокрема *ESP32*, основним різновидом послідовного інтерфейсу є *UART* (*Universal Asynchronous Receiver-Transmitter*). На стороні персонального комп’ютера *UART*-пристрої зазвичай представлені у вигляді *COM*-портів, що дозволяє організувати прямий обмін даними між ПК та мікроконтролером.

UART є асинхронним інтерфейсом, що не потребує окремої лінії синхронізації. Обмін даними здійснюється за попередньо узгодженими параметрами, серед яких основними є швидкість передачі даних (*baudrate*), кількість біт даних, наявність або відсутність біта парності, кількість стоп-бітів.

У більшості випадків для *ESP32* використовується стандартна конфігурація *8N1* – 8 біт даних, без парності, 1 стоп-біт, що забезпечує сумісність з широким спектром програмних і апаратних засобів.

UART-інтерфейс застосовується для:

- первинного прошивання мікроконтролера;
- оновлення прошивки;
- виводу діагностичних повідомлень;
- обміну командами та статусною інформацією;
- моніторингу процесу завантаження пристрою.

На рівні операційної системи послідовні інтерфейси представлені у вигляді логічних портів.

Windows – COM1, *COM2*, *COM3* і т.д.;

Linux – /dev/ttyUSBx, /dev/ttyACMx;

macOS – /dev/cu., /dev/tty.*.*

Desktop-застосунок виконує автоматичний пошук доступних *COM*-портів, що дозволяє користувачеві швидко визначити підключений пристрій без необхідності ручної конфігурації. Це є важливою складовою зручності використання програмного засобу.

Послідовний інтерфейс відіграє ключову роль у процесі запису прошивки в пам'ять *ESP32*. Саме через *UART* здійснюється передача бінарних даних прошивки з ПК до мікроконтролера.

Процес прошивки включає такі етапи:

1) Відкриття *COM*-порту з заданою швидкістю передачі.
2) Переведення *ESP32* у режим завантажувача.
3) Послідовна передача блоків прошивки.
4) Контроль цілісності переданих даних.
5) Перезапуск пристрою після завершення операції.

Для реалізації цих функцій використовується спеціалізований інструмент *esptool*, який взаємодіє з *ESP32* саме через *UART*. *Desktop*-застосунок керує цим процесом, відображаючи прогрес та повідомлення про стан виконання.

Окрім прошивки, *UART* використовується для моніторингу роботи пристрою. *ESP32* може передавати через послідовний порт текстові повідомлення, що містять інформацію про процес завантаження, ініціалізацію модулів, мережеві події, помилки та попередження, службову телеметрію.

Застосування *UART/COM*-портів у системі прошивки та керування має низку переваг таких як простота апаратної реалізації, мінімальні вимоги до додаткового обладнання, висока сумісність з *ESP32*, стабільність та передбачуваність роботи, можливість використання у середовищах без мережевого доступу.

Завдяки цим властивостям послідовний інтерфейс є надійною основою для первинної прошивки, діагностики та аварійного відновлення пристроїв.

Попри численні переваги, послідовний інтерфейс має і певні обмеження, зокрема необхідність фізичного підключення пристрою до ПК. Саме тому у загальній архітектурі системи *UART*-інтерфейс розглядається як базовий та гарантований механізм доступу, який доповнюється мережевими технологіями, зокрема *OTA*-оновленням.

2.8. Вимоги до початкової прошивки *ESP32* для підтримки *OTA*-оновлення

Однією з ключових умов реалізації механізму віддаленого оновлення прошивки є наявність на мікроконтролері *ESP32* попередньо встановленої базової прошивки, яка забезпечує підтримку мережевого з’єднання та механізмів прийому нових програмних образів. *OTA*-оновлення не може бути виконане на порожньому або непідготовленому пристрої, оскільки для прийому прошивки необхідна активна програмна логіка, що керує мережевими інтерфейсами та процесом оновлення.

Базова прошивка виконує роль початкового програмного середовища, яке забезпечує ініціалізацію апаратних ресурсів *ESP32*, підключення до мережі *Wi-Fi* та запуск сервісів, необхідних для прийому *OTA*-запитів. Як правило, така прошивка реалізує один із двох сценаріїв: створення власної точки доступу для первинного налаштування або автоматичне підключення до заздалегідь заданої

мережі. У межах цього програмного середовища користувач має можливість задати параметри підключення, переглянути службову інформацію про пристрій та активувати режим оновлення.

Важливим компонентом базової прошивки є реалізація безпечного механізму оновлення, що передбачає використання резервних розділів флеш-пам'яті, перевірку цілісності отриманого програмного образу та контроль коректності завершення процесу прошивки.

У разі виникнення помилки або переривання процесу оновлення пристрій зберігає можливість повернення до попередньої працездатної версії програмного забезпечення, що є критично важливим для систем, які працюють у безперервному режимі.

Первинна прошивка *ESP32* є обов'язковою передумовою використання *OTA*-оновлення і фактично визначає подальші можливості віддаленого керування пристроєм. У розробленому програмному засобі цей аспект враховано шляхом розділення сценаріїв локального та віддаленого оновлення прошивки.

В таблиці 2.3 наведено перелік компонентів, які мають бути присутніми в первинній прошивці для можливості віддаленого оновлення.

Таблиця 2.3

Обов'язкові компоненти та їх призначення в первинній прошивці

Компонент базової прошивки	Призначення
Підключення до *Wi-Fi*	Забезпечення мережевого з'єднання для *OTA*
Режим точки доступу (*AP*)	Первинне налаштування параметрів мережі
OTA-сервіс	Прийом і запис нового образу прошивки
Механізм перевірки цілісності	Контроль коректності завантаженого файлу
Резервні розділи *Flash*	Можливість відкату до попередньої версії

Локальна прошивка використовується для первинної підготовки пристрою та встановлення базового програмного середовища, після чого всі наступні оновлення

можуть виконуватися дистанційно за допомогою *OTA*-механізмів, інтегрованих у клієнтський застосунок.

Запропонований підхід дозволяє поєднати переваги локального та віддаленого оновлення прошивки, забезпечуючи гнучкість, надійність і зручність експлуатації системи в реальних умовах використання.

2.9. Висновки до розділу

У другому розділі було здійснено огляд програмних та технічних засобів, що використовуються в процесі розробки програмного модуля для віддаленого оновлення прошивки ретрансляторів та модулів керування наземними станціями. Проаналізовано обране середовище розробки, мови програмування, бібліотеки та фреймворки, а також принципи організації програмної архітектури системи.

Як основне середовище розробки було обрано *Visual Studio Code*, що забезпечує кросплатформеність, підтримку розширень, зручну роботу з мовами програмування *Python* та *C/C++*, а також інтеграцію з системами контролю версій. Використання даного середовища дозволяє ефективно розробляти як *desktop*-застосунок для керування процесом прошивки, так і вбудоване програмне забезпечення для мікроконтролерів *ESP32*.

У межах проєкту застосовано мову програмування *Python* як основну для реалізації *desktop*-клієнта. Завдяки використанню бібліотек *PyQt6*, *esptool*, *pyserial* та *littlefs-python* було реалізовано графічний інтерфейс користувача, механізми обміну даними з пристроями через послідовний інтерфейс, а також роботу з файловими системами у *flash*-пам’яті мікроконтролера. Такий підхід забезпечує високу гнучкість, масштабованість і простоту підтримки програмного забезпечення.

У межах розділу детально розглянуто формат *JSON* як універсальний механізм представлення конфігурацій, телеметричних даних та параметрів оновлення прошивки. Застосування *JSON* дозволяє реалізувати уніфікований підхід до зберігання та передачі даних між компонентами системи, забезпечує

читабельність конфігурацій і спрощує їх обробку на стороні клієнта, сервера та пристрою. Врахування правил валідації, версіонування та структурної цілісності *JSON*-повідомлень підвищує надійність роботи системи загалом.

Окрему увагу приділено роботі з послідовними інтерфейсами *UART/COM*, які відіграють базову роль у процесі первинної прошивки, діагностики та аварійного відновлення пристроїв на базі *ESP32*. Послідовний інтерфейс розглядається як гарантований низькорівневий канал доступу, що забезпечує стабільну роботу навіть у випадках відсутності мережевого з'єднання або некоректної роботи основної прошивки.

Разом з тим, у рамках теоретичного огляду було розглянуто технологію *OTA* (*Over-The-Air*). Архітектура розробленого програмного засобу спроєктована з урахуванням можливості інтеграції *OTA*-оновлення, що дозволить виконувати прошивку пристроїв через мережеве з'єднання без фізичного доступу до них. Поєднання *desktop*-механізму прошивки з *OTA*-підходом створює гібридну модель оновлення, яка може бути адаптована до різних умов експлуатації та рівнів доступності мережі.

Окрему увагу в межах другого розділу було приділено вимогам до початкового програмного забезпечення мікроконтролера *ESP32*, необхідного для реалізації механізмів віддаленого оновлення прошивки. Було встановлено, що *OTA*-оновлення можливе лише за наявності базової прошивки, яка забезпечує ініціалізацію апаратних ресурсів, підтримку мережевого з'єднання та прийом нових програмних образів. Таким чином, базова прошивка розглядається не як допоміжний елемент, а як фундаментальна складова всієї системи віддаленого оновлення.

Аналіз принципів побудови такої базової прошивки показав, що вона має виконувати роль проміжного програмного шару між апаратною частиною пристрою та клієнтським програмним засобом. Саме на цьому етапі формується логіка безпечного оновлення, включаючи використання резервних розділів пам'яті, контроль цілісності даних і механізми відкату у разі помилки. Це дозволяє

забезпечити надійність роботи пристрою в умовах нестабільного мережевого з’єднання або часткової втрати даних під час *OTA*-оновлення.

Отже, розглянуті у другому розділі програмні засоби, формати даних, інтерфейси та архітектурні рішення створюють надійну технологічну основу для реалізації програмного модуля віддаленого оновлення прошивки. Вибір зазначених інструментів є обґрунтованим з точки зору надійності, масштабованості та перспектив подальшого розвитку системи, а отримані результати безпосередньо використовуються у практичній реалізації, представленій у третьому розділі.

РОЗДІЛ 3
РЕАЛІЗАЦІЯ ТА ТЕСТУВАННЯ ПРОГРАМНОГО ЗАСОБУ ВІДДАЛЕНОГО ОНОВЛЕННЯ ПРОШИВКИ

3.1. Загальна структура програмного засобу

Розробка програмного модуля віддаленого оновлення прошивки виконувалася з використанням модульного підходу. Програмний проєкт поділено на окремі логічні файли, кожен з яких відповідає за визначену функціональність. Така організація спрощує розуміння коду, його супровід та подальше розширення.

Основними компонентами проєкту є:

- головний графічний інтерфейс користувача;
- модулі прошивки через послідовний інтерфейс;
- модулі віддаленого *OTA*-оновлення;
- засоби роботи з конфігураціями;
- клієнт-серверна взаємодія;
- механізми багатопотокового виконання.

Структурна схема програмного модуля віддаленого оновлення прошивки наведена на рис. 3.1. Центральним елементом системи є *desktop*-застосунок з графічним інтерфейсом користувача, який забезпечує керування процесами локальної прошивки через послідовний інтерфейс та віддаленого *OTA*-оновлення через мережу.

Логіка оновлення реалізована у вигляді окремих модулів і виконується у фонових потоках, що гарантує стабільність роботи інтерфейсу. Конфігураційні дані обробляються у форматі *JSON* та зберігаються у файловій системі *LittleFS* на стороні *ESP32*.

Взаємодія з сервером забезпечує централізоване керування прошивками, телеметрією та оновленнями клієнтського програмного забезпечення.

Рис. 3.1. Структурна схема програмного модуля віддаленого оновлення прошивки

3.2. Розробка графічного інтерфейсу користувача

Графічний інтерфейс користувача (*GUI*) у розробленому програмному засобі відіграє ключову роль, оскільки забезпечує взаємодію оператора з функціями прошивки, оновлення, конфігурування та діагностики пристроїв на базі ш. Основною вимогою до інтерфейсу є *user-friendly* підхід, який мінімізує потребу у використанні сторонніх інструментів (термінал, браузер, ручні команди), а також забезпечує зрозумілий контроль стану виконання операцій.

Для реалізації *GUI* використано бібліотеку *PyQt6*, яка надає широкі можливості створення кросплатформених *desktop*-застосунків на *Python*. Інтерфейс будується у вигляді класу *ESP32FlasherUI* (файл *ui.py*), який наслідується від *QMainWindow*. Це дозволяє реалізувати стандартну структуру віконного

застосунку: головне вікно, центральний віджет, меню/панелі, діалогові вікна; та забезпечує гнучкість у розвитку проєкту.

Точкою входу у застосунок є файл *main.py*, в якому створюється екземпляр *QApplication*, а також ініціалізується головне вікно інтерфейсу:

```
def main():
    app = QApplication(sys.argv)
    window = ESP32FlasherUI()
    window.show()
    sys.exit(app.exec())
```

QApplication відповідає за запуск циклу обробки подій, взаємодію віджетів з операційною системою, обробку вводу користувача – миша, клавіатура, підтримку стилів, шрифтів і поведінки елементів *GUI*.

Після створення об’єкта *ESP32FlasherUI* викликається *show()*, що відображає вікно, а *app.exec()* запускає цикл подій, без якого графічний інтерфейс не працює.

Основний клас *GUI – ESP32FlasherUI*. У конструкторі *__init__()* виконуються ключові кроки такі як задання назви та розмірів вікна, ініціалізація внутрішніх змінних стану, побудова елементів інтерфейсу, застосування стилів, відновлення сесії/налаштувань, запуск таймерів оновлення портів і перевірки оновлень.

Фрагмент ініціалізації:

```
self.setWindowTitle(f"ESP32 Flasher Client v{CURRENT_VERSION}")
self.resize(950, 900)
self.firmware_path = None
self.port = None
self.token = None
self.user_role = None
self.username = "Гість"
```

Такі змінні формують контекст застосунку, де зберігається:

- вибраний *COM*-порт;
- вибрана прошивка;
- поточні *JSON*-конфігурації;

– токен авторизації та роль користувача;

– режими роботи.

Це дозволяє інтерфейсу бути стано-залежним, кнопки активуються або блокуються залежно від того, чи йде процес прошивки, чи відкритий монітор, чи є доступні порти тощо.

У *PyQt6* інтерфейс формується з віджетів (*QWidget*), які розміщуються у контейнері з використанням layout-менеджерів. У даному проєкті застосовано:

– *QVBoxLayout* – вертикальне розміщення блоків;

– *QHBoxLayout* – горизонтальні панелі керування;

– *QGroupBox* – логічне групування пов’язаних елементів.

Головне вікно містить *central widget*:

```
central = QWidget()
self.setCentralWidget(central)
main_layout = QVBoxLayout(central)
```

Верхня панель користувача та керування сесією (логін/вихід/налаштування).

```
self.user_label = QLabel(self.username)
self.settings_btn = QPushButton("⚙")
self.logout_btn = QPushButton("Вийти")
self.open_ip_btn = QPushButton("🌐 Відкрити пристрій у браузері")
self.open_ip_btn.setVisible(False)
```

Блок портів та доступу до *Serial Monitor*. Група «Порти підключення» включає список портів (*single* режим) або список з чекбоксами (*bulk* режим), кнопку оновлення портів, кнопку відкриття *Serial Monitor*.

```
self.port_combo = QComboBox()
self.ports_list = QListWidget()
self.monitor_btn = QPushButton("⌨ Монітор")
self.port_timer = QTimer()
self.port_timer.timeout.connect(self.check_ports_update)
self.port_timer.start(1000)
```

Блок вибору джерела прошивки *.bin*. Передбачено два варіанти: отримання *.bin* з репозиторію/сервера або ручний вибір файлу прошивки.

Для цього реалізовано групу *QButtonGroup* з двома радіокнопками:

```
self.src_server = QRadioButton("З репозиторію")
self.src_local = QRadioButton("Вибрати файл вручну")
if self.src_server.isChecked():
    self.server_widget.show()
    self.local_widget.hide()
else:
    self.server_widget.hide()
    self.local_widget.show()
```

Блок роботи з конфігураціями *JSON/LittleFS*. Конфігурації пристрою представлені у форматі *JSON* та можуть завантажуватися з сервера або з локальної папки. Окреме вікно для редагування реалізоване у класі *JsonEditorDialog*, що підвищує зручність роботи з великими конфігураційними файлами

```
self.json_editor = QTextEdit()
self.json_editor.setMinimumHeight(100)
```

Кнопка запуску прошивки:

```
self.flash_btn = QPushButton("🚀 ЗАПУСТИТИ ПРОШИВКУ")
self.flash_btn.clicked.connect(self.start_flash)
self.single_progress = QProgressBar()
self.bulk_scroll = QScrollArea()
```

Такий підхід забезпечує зрозумілу ієрархію: користувач зверху вниз проходить шлях «підключення → вибір файлу → конфіг → запуск».

Для забезпечення сучасного зовнішнього вигляду використано налаштування стилів через *setStyleSheet*. У стилях визначені:

– фон і кольори елементів;
– вигляд кнопок;
– оформлення групових блоків;
– стилі списків і прогрес-барів.

У проєкті застосовано кілька діалогових вікон.

LoginDialog – вхід користувача та отримання токена.

SettingsDialog – параметри інтерфейсу та прошивки (*baudrate, fast mode, pre-patch, bulk mode*).

AdminPanel – управління користувачами (для ролі *admin*);

JsonEditorDialog – розширене редагування конфігураційного файла.

Наявність діалогових вікон дозволяє винести другорядні функції з основного екрану і зробити головний інтерфейс менш перевантаженим.

Реалізований графічний інтерфейс користувача на базі *PyQt6* забезпечує повний цикл роботи з пристроями *ESP32*: вибір підключення, вибір прошивки, редагування конфігурацій, запуск локальної або віддаленої прошивки, відображення прогресу та логів, а також підтримку ролей користувачів і налаштувань застосунку. Використання модульної структури та багатопотокового виконання дозволило забезпечити стабільність роботи інтерфейсу і підвищити зручність експлуатації системи.

3.3. Виявлення та вибір підключених пристроїв

Одним із ключових етапів процесу прошивки та оновлення програмного забезпечення є коректне виявлення фізично підключених пристроїв. У випадку модулів на базі *ESP32* підключення до персонального комп’ютера зазвичай здійснюється через інтерфейс *USB–UART*, який в операційній системі відображається у вигляді *COM*-порту (у *Windows*) або відповідного послідовного порту в інших ОС.

Для роботи з послідовними портами використовується бібліотека *pyserial*, яка є стандартним інструментом *Python* для взаємодії з *UART/USB*-пристроями. Вона надає доступ до інформації про всі наявні в системі порти, незалежно від типу підключеного обладнання.

У програмі реалізована окрема функція:

def get_available_ports():

return sorted(p.device for p in serial.tools.list_ports.comports())

Дана функція виконує такі дії:

- викликає *serial.tools.list_ports.comports()*, яка повертає список об’єктів, що містять інформацію про всі активні послідовні порти.
- з кожного об’єкта береться поле *device*, яке містить системне ім’я порту.
- список портів сортується для зручності відображення в інтерфейсі.

Таким чином, на рівні логіки застосунку формується актуальний перелік доступних портів, який використовується для побудови елементів графічного інтерфейсу.

Отриманий список портів інтегрується у графічний інтерфейс користувача. Залежно від режиму роботи програми використовуються два різних підходи:

- одиночна прошивка – вибір одного порту зі списку;
- масова прошивка – вибір кількох портів одночасно.

Для одиночного режиму використовується випадаючий список *QComboBox*:

self.port_combo = QComboBox()

self.port_combo.currentTextChanged.connect(self.on_port_changed)

У масовому режимі активується список з чекбоксами *QListWidget*, який дозволяє вибрати кілька портів:

self.ports_list = QListWidget()

self.ports_list.setSelectionMode(QListWidget.SelectionMode.MultiSelection)

Однією з важливих особливостей реалізації є динамічне оновлення списку портів у реальному часі. Це дозволяє користувачу підключати або відключати пристрої *ESP32* без необхідності перезапуску програми.

Для цього використовується таймер *QTimer*:

self.port_timer = QTimer()

self.port_timer.timeout.connect(self.check_ports_update)

self.port_timer.start(1000)

Таймер виконує перевірку кожну секунду. У методі *check_ports_update()* порівнюється поточний список портів із попередньо збереженим:

def check_ports_update(self):

```
    current_ports = get_available_ports()
    if current_ports != self.last_known_ports:
        self.refresh_ports(current_ports)
        self.last_known_ports = current_ports
```

Метод *refresh_ports()* відповідає за безпосереднє оновлення елементів *GUI*:

```
def refresh_ports(self, ports=None):
    if ports is None:
        ports = get_available_ports()
```

Після вибору порту користувачем його значення зберігається у внутрішній змінній:

```
def on_port_changed(self, text):
    self.port = text if text and "Немає" not in text else None
```

Окрім вибору порту для прошивки, користувач має можливість відкрити *Serial Monitor* для перегляду діагностичних повідомлень від *ESP32*. Монітор працює з тим самим обраним портом, що гарантує узгодженість між прошивкою та відладкою.

Водночас у програмі реалізовано захист від конфліктів. Якщо *Serial Monitor* відкритий, прошивка блокується. Якщо триває прошивка, відкриття монітора забороняється.

Це критично важливо, оскільки одночасний доступ до одного *COM*-порту з кількох потоків може призвести до помилок або пошкодження процесу прошивки.

3.4. Реалізація локальної прошивки *ESP32* через інтерфейс *UART*

Локальна прошивка через послідовний інтерфейс *UART* є базовим та найбільш надійним способом оновлення програмного забезпечення мікроконтролерів *ESP32*. Даний метод використовується як для первинного запису прошивки, так і для аварійного відновлення пристроїв у випадках, коли мережеві інтерфейси недоступні або основна прошивка пошкоджена. У межах даної роботи

локальна прошивка реалізована як невід'ємна частина *desktop*-застосунку та інтегрована у єдиний графічний інтерфейс користувача.

Прошивка *ESP32* через *UART* базується на використанні вбудованого завантажувача (*bootloader*) мікроконтролера, який активується під час старту пристрою у спеціальному режимі. Через *USB-UART* перетворювач (*CP2102, CH340, FTDI* тощо) комп'ютер отримує доступ до *flash*-пам'яті мікроконтролера.

Для роботи з *ESP32* обрано бібліотеку *esptool*, що є стандартним інструментом компанії *Espressif*. Вона забезпечує низькорівневу взаємодію з *flash*-пам'яттю та підтримує більшість моделей *ESP32*.

У роботі *esptool* використовується не як зовнішня утиліта, а як *Python*-модуль.

Оскільки процес прошивки може тривати декілька секунд або хвилин, його виконання у головному потоці графічного інтерфейсу призвело б до блокування *UI*. Для усунення цієї проблеми прошивка реалізована у окремому потоці, який успадковується від класу *QThread*.

Основний клас, що відповідає за локальну прошивку, має вигляд:

```
class FlashThread(QThread):
    log = pyqtSignal(str)
    progress = pyqtSignal(int)
    finished = pyqtSignal(bool, str)
```

Перед запуском прошивки користувач обирає:

- *COM*-порт;
- файл прошивки;
- параметри швидкості передачі даних;
- режим оновлення.

Після цього створюється екземпляр потоку:

```
self.thread = FlashThread(
    port=self.port,
    firmware_path=fw_path,
    config_data=config_data,
    config_settings=config_settings,
```

```
    baudrate=self.baudrate
)
```

Усередині потоку формується набір параметрів для *esptool*. Формування команди виконується наступним чином:

```
cmd = [
    '--no-stub',
    '--chip', 'esp32',
    '--port', self.port,
    '--baud', baud,
    '--before', 'default-reset',
    '--after', 'hard-reset',
    'write-flash',
    '-z',
    '--flash-mode', 'dio',
    '--flash-freq', '40m',
    '--flash-size', 'detect',
    offset, file_path
]
```

Для коректного відображення процесу прошивки реалізовано спеціальний клас *EsptoolLogger*, який перехоплює стандартний вивід *esptool*:

```
sys.stdout = EsptoolLogger(self.log, self.progress)
```

Прогрес прошивки візуалізується за допомогою *QProgressBar*, що значно підвищує зручність роботи користувача.

Окрім одиночної прошивки, реалізовано режим масової прошивки, який дозволяє одночасно прошивати декілька *ESP32*, підключених до різних *COM*-портів.

Для кожного порту:

- створюється окремий потік прошивки;
- формується власний прогрес-бар;
- логи маркуються іменем порту.

Такий підхід дозволяє масштабувати процес оновлення без суттєвого ускладнення логіки застосунку.

3.5. Робота з конфігураціями пристрою

Однією з ключових вимог до систем керування ретрансляторами та наземними станціями є можливість гнучкого налаштування параметрів роботи пристрою без повної заміни прошивки. Для цього у розробленому програмному засобі реалізовано механізм роботи з конфігураціями, який базується на використанні формату *JSON* та файлової системи *LittleFS*, розміщеної у *flash*-пам'яті мікроконтролера *ESP32*.

У межах даної системи конфігурація пристрою являє собою набір параметрів, що описують:

- мережеві налаштування;
- режими роботи радіомодуля;
- параметри телеметрії;
- поведінку інтерфейсів та служб;
- специфічні особливості конкретного типу пристрою.

Усі ці параметри зберігаються у вигляді структурованих *JSON*-файлів, що мають читабельний формат і можуть бути легко змінені без перекомпіляції прошивки.

Основним файлом конфігурації є *config.json*, який зберігається у файловій системі *LittleFS* на *ESP32*.

Приклад *config.json:*

```
{
  "ssid": "ToSetUp",
  "password": "12345",
  "apSsid": "",
  "apPassword": "",
  "statusUpdateTime": 10,
```

```
  "reconnect": false,
  "vrxOpt": "OFF",
  "rctOpt": "OFF",
  "radio": false,
  "rssiDisabled": true
}
```

Для зберігання конфігурацій використовується *LittleFS* – легка журналююча файлова система, оптимізована для вбудованих пристроїв з обмеженим обсягом *flash*-пам'яті.

У *flash*-пам'яті *ESP32* виділяється окремий розділ під файлову систему, адреса та розмір якого визначаються у конфігурації прошивки:

```
CONFIGS = {
    "Standart": {
        "offset_fs": "0x3b0000",
        "fs_size": "0x40000"
    }
}
```

Для отримання поточної конфігурації з *ESP32* у застосунку реалізовано механізм зчитування файлової системи безпосередньо з *flash*-пам'яті.

Процес складається з таких етапів:

1) Зчитування дампу *flash*-пам'яті з області *LittleFS* за допомогою *esptool*.
2) Монтування файлової системи *LittleFS* у середовищі *Python*.
3) Пошук та читання конфігураційних файлів.
4) Перетворення даних у формат *JSON* для подальшого відображення в *UI*.

Фрагмент коду зчитування:

```
fs = LittleFS(block_size=4096, block_count=block_count)
fs.context.buffer = bytearray(fs_data)
fs.mount()
```

Після монтування здійснюється спроба зчитування файлу *config.json*. Якщо файл відсутній, програма намагається зчитати окремі конфігураційні файли (*wifi.json, radio.json* тощо), об'єднуючи їх у єдину структуру.

3.6. Реалізація *OTA*-оновлення прошивки

Реалізована система *OTA*-оновлення базується на клієнт-серверній взаємодії між:

- *desktop*-застосунком оператора;
- *ESP32*-пристроєм, що виконує роль *OTA*-клієнта або *OTA*-сервера;
- мережевою інфраструктурою, у межах якої пристрій доступний за *IP*-адресою.

У розробленій системі *OTA*-оновлення інтегровано безпосередньо у клієнтський застосунок і реалізовано за наступною логікою:

ESP32 після прошивки або налаштування підключається до локальної мережі *Wi-Fi*.

1) Пристрій повідомляє свою *IP*-адресу.
2) *Desktop*-застосунок зберігає *IP*-адресу пристрою.
3) Користувач обирає режим *OTA*-оновлення.
4) Прошивка передається на *ESP32* через *HTTP*-запит.
5) *ESP32* приймає бінарний файл і записує його у *flash*-пам'ять.
6) Після завершення оновлення пристрій автоматично перезавантажується.

Для реалізації *OTA*-оновлення використовується стандартний *HTTP*-протокол. На стороні *desktop*-застосунку передача прошивки здійснюється шляхом *HTTP POST*-запиту з використанням бібліотеки *requests:*

```
requests.post(
    f"http://{device_ip}/update",
    files={"firmware": open(firmware_path, "rb")},
    timeout=30
)
```

Як і у випадку локальної прошивки, *OTA*-оновлення виконується у фоновому потоці, що унеможливлює блокування графічного інтерфейсу.

Для цього створюється окремий клас потоку, наприклад:

```
class OTAUpdateThread(QThread):
    progress = pyqtSignal(int)
    finished = pyqtSignal(bool, str)
```

3.7. Моніторинг та логування

Для діагностики та аналізу роботи пристрою реалізовано серійний монітор, що відображає повідомлення, які передає *ESP32* через *UART*.

```
line = self.serial_conn.readline().decode("utf-8")
self.data_received.emit(line)
```

Логування також використовується під час прошивки та *OTA*-оновлення, що дозволяє швидко виявляти та аналізувати помилки.

3.8. Тестування та перевірка працездатності програмного засобу

Завершальним етапом розробки програмного модуля віддаленого оновлення прошивки є тестування та експериментальна перевірка його працездатності. Метою даного етапу є підтвердження коректності роботи реалізованих механізмів, оцінка стабільності системи, а також перевірка зручності використання програмного засобу в реальних умовах експлуатації.

Тестування проводилося для всіх ключових функціональних модулів застосунку, зокрема:

- виявлення та вибору підключених пристроїв;
- локальної прошивки *ESP32* через *UART*;
- роботи з конфігураціями (*JSON* + *LittleFS*);
- віддаленого *OTA*-оновлення прошивки;
- багатопотокової роботи та обробки помилок.

3.8.1. Тестування авторизації

При запуску застосунку відкривається вікно авторизації (рис. 3.1.)

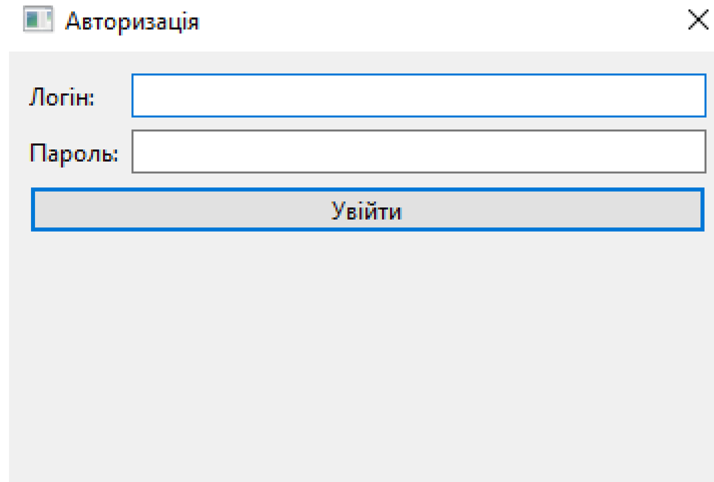


Рис. 3.1. Вікно авторизації

Передумови: сервер доступний, існує користувач.

Кроки:

1) Ввести валідний логін
2) Ввести валідний пароль
3) Натиснути «Увійти»

Очікувано:

– користувач авторизується, головне вікно відкривається;
– роль/ім’я відображається в *UI*;
– токен/сесія збережені.

В результаті авторизації з існуючими коректними логіном та паролем відкривається головне вікно застосунку, як зображено на рисунку 3.2.

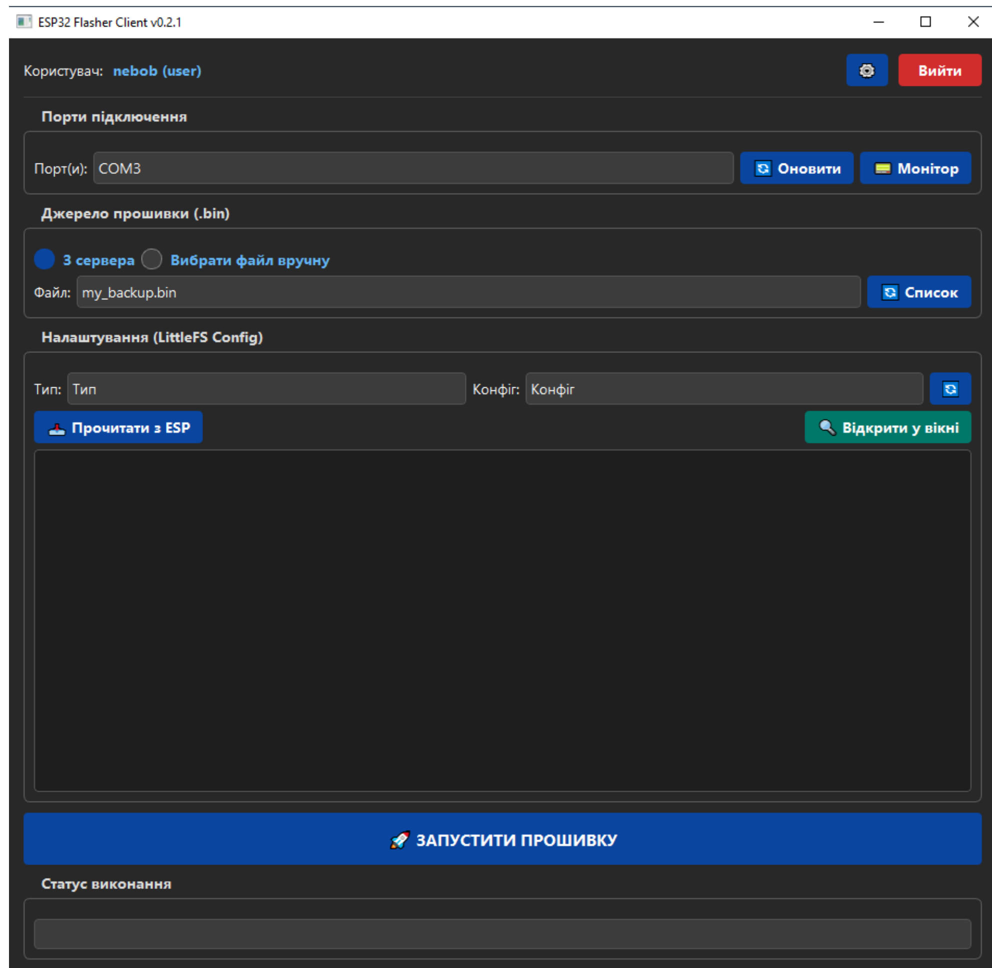


Рис. 3.2. Головне вікно застосунку

Тест 2. Порожній логін.

Кроки:

1) Логін порожній.
2) Пароль заповнений.
3) Натиснути «Увійти».

Очікувано**:**

– повідомлення «Заповніть логін»/«Заповніть поля»;
– запит на сервер не виконується.

Результатом виконання кроків стало виникнення помилки із запитом заповнити всі поля, як це зображено на рисунку 3.3.

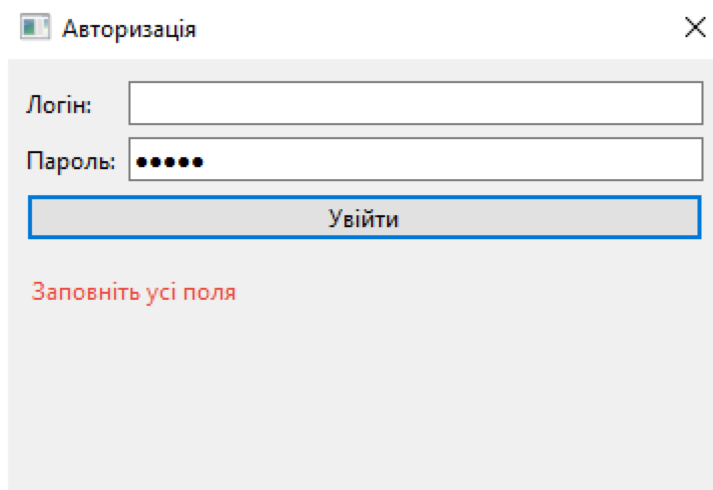


Рис. 3.3. Повідомлення про помилку при незаповненому логіні

Тест 3. Порожній пароль.

Кроки:

1) Пароль порожній.
2) Логін заповнений.
3) Натиснути «Увійти».

Очікувано**:**

– повідомлення «Заповніть пароль»/«Заповніть поля»;
– запит на сервер не виконується.

Результатом виконання кроків стало виникнення помилки із запитом заповнити всі поля, як це зображено на рис. 3.4.

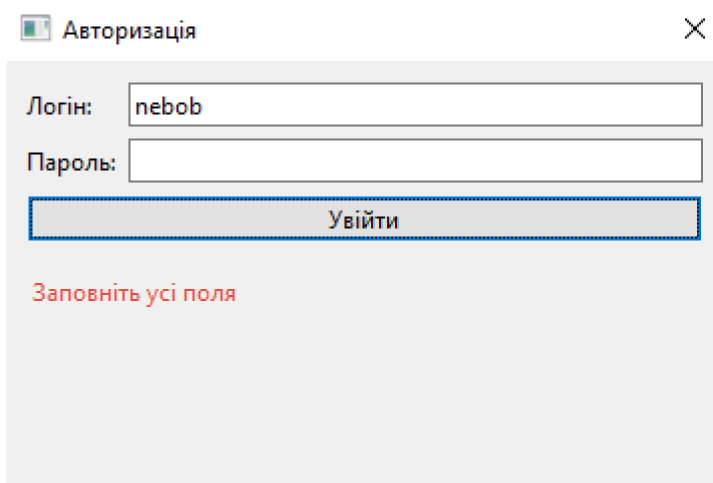


Рис. 3.4. Повідомлення про помилку при незаповненому паролі

Тест 4. Невірні облікові дані.

Передумови**:** сервер доступний.

Кроки:

1) Ввести існуючий логін.
2) Ввести неправильний пароль.
3) Натиснути «Увійти».

Очікувано:

– повідомлення «Невірний логін або пароль»;
– користувач залишається на екрані авторизації;
– токен не зберігається.

В результаті виконання кроків виводиться повідомлення про невірність облікових даних, як це зображено на рис. 3.5.

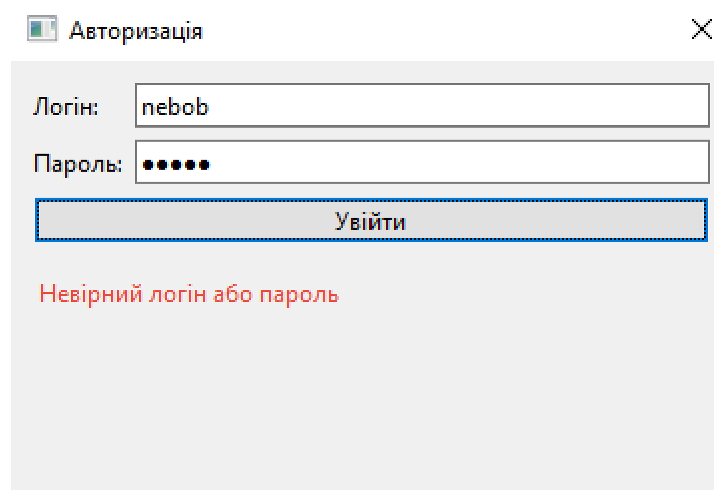


Рис. 3.5. Повідомлення про помилку при невірно вказаних облікових даних

3.8.2. Тест локальної прошивки *ESP32* через *UART*

Тест успішної локальної прошивки.

Мета тесту: перевірка коректності процесу прошивки через *COM*-порт.

Кроки виконання:

1) Підключити *ESP32* до ПК через *USB*.

2) Обрати відповідний *COM*-порт у застосунку (рис. 3.6.).

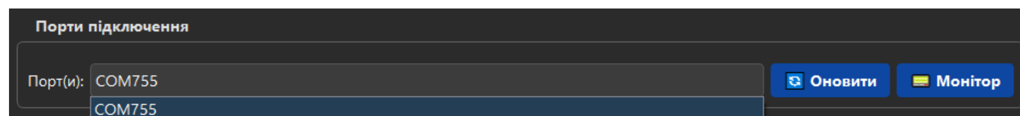


Рис. 3.6. Обраний відповідний *COM*-порт

3) Обрати файл прошивки *.bin* (рис. 3.7.).

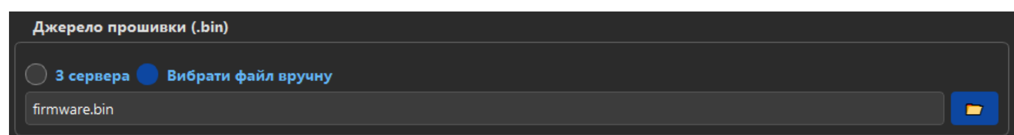


Рис. 3.7. Обраний локально файл з прошивкою

4) Натиснути кнопку «Запустити прошивку».

Очікуваний результат: прошивка виконується успішно, відображається прогрес виконання (рис. 3.8.), після завершення користувач отримує повідомлення про успішну прошивку (рис. 3.9.).

Результат: відповідність очікуванням.

Рис. 3.8. Відсоток виконання процесу прошивки

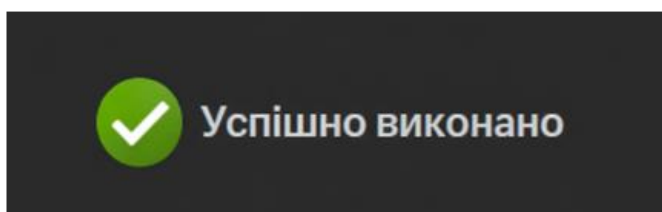


Рис. 3.9. Повідомлення про успішну прошивку

Тест прошивки без файлу прошивки.

Кроки виконання:

1) Обрати *COM*-порт.

2) Не обирати файл прошивки.

3) Натиснути кнопку «Запустити прошивку».

Очікуваний результат: прошивка не виконується, відображається повідомлення про помилку (рис. 3.10).

Результат: відповідність очікуванням.

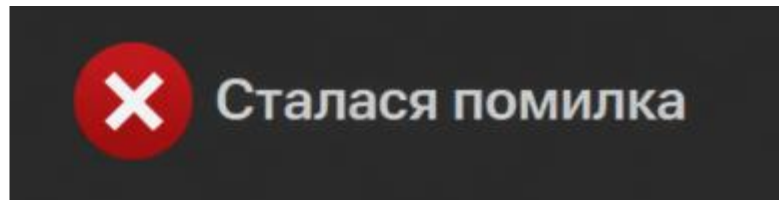


Рис. 3.10. Повідомлення про те, що сталася помилка

3.8.3. Тест роботи з конфігурацією *JSON* + *LittleFS*

Тест зчитування конфігурації з пристрою

Мета тесту: перевірка коректного зчитування конфігураційних даних.

Кроки виконання:

1) Підключити *ESP32* до ПК.

2) Обрати *COM*-порт.

3) Натиснути кнопку «Прочитати з *ESP*» (рис. 3.11.).

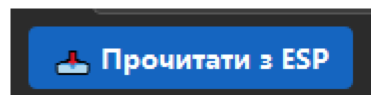


Рис. 3.11. Кнопка «Прочитати з *ESP*»

Очікуваний результат: конфігурація зчитується з пристрою та відображається у вигляді *JSON* у редакторі (рис. 3.12.).

```
"apSsid": null,
"apPassword": null,
"reconnect": null,
"statusUpdateTime": 2,
"vrxOpt": "3.3",
"rctOpt": "SLT1",
"radio": true,
```

Рис. 3.12. Відображення конфігурації у форматі *JSON* у редакторі

3.8.4. Тест *OTA*-оновлення прошивки

Тест *OTA*-оновлення

Мета тесту: перевірка працездатності *OTA*-оновлення.

Кроки виконання:

1) Переконатися, що *ESP32* підключена до *Wi-Fi* (рис. 3.13.).

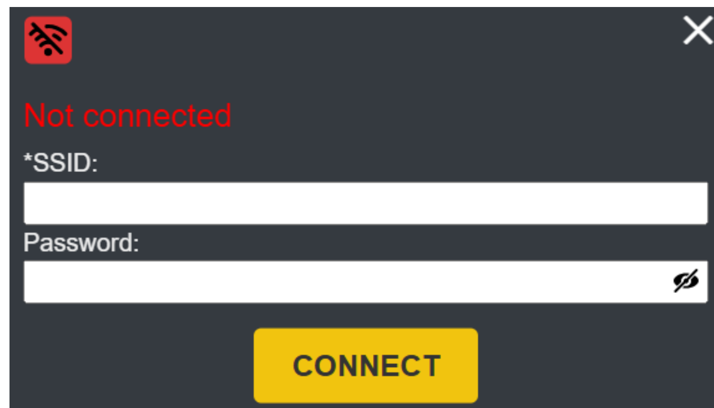


Рис. 3.13. Під'єднання *ESP32* локальної мережі

2) Отримати *IP*-адресу пристрою.
3) Обрати режим *OTA*-оновлення (рис. 3.14.).

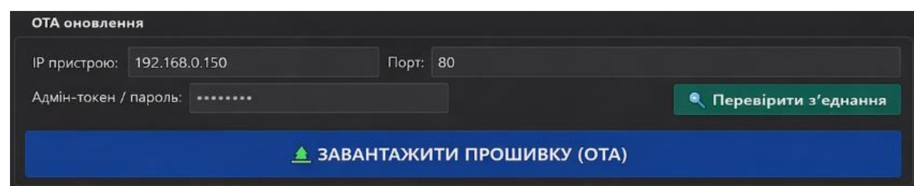


Рис. 3.14. Блок з режимом оновленням прошивки *OTA*

4) Обрати файл прошивки.

5) Запустити *OTA*-оновлення.

Очікуваний результат: прошивка передається на пристрій, відображається прогрес, після завершення *ESP32* автоматично перезавантажується.

Результат: процес прошивки та перезавантаження *ESP32* успішний, що можна побачити у вікні логів (рис. 3.15.)

```
Writing at 0x0011ce1a... (89 %)
Writing at 0x00124ef4... (91 %)
Writing at 0x0012a1c6... (93 %)
Writing at 0x0012fbee... (95 %)
Writing at 0x0013529a... (97 %)
Writing at 0x0013a71e... (100 %)
Wrote 1240448 bytes (749653 compressed) at 0x00010000 in 11.4 seconds (effective 874.2 kbit/s)...
Hash of data verified.

Leaving...
Hard resetting via RTS pin...
```

Рис. 3.15. Вікно з виведенням логів

3.8.5. Тестування сторінки налаштувань

Тест відкриття сторінки налаштувань

Мета тесту: перевірка доступності сторінки налаштувань.

Кроки виконання:

1) Запустити програмний засіб.
2) Натиснути кнопку «Налаштування» (рис. 3.16).

Рис. 3.16. Кнопка «Налаштування»

Очікуваний результат: відкривається вікно налаштувань із відображенням поточної версії програми та доступних параметрів конфігурації (рис. 3.17.).

Результат повністю відповідає очікуванням.

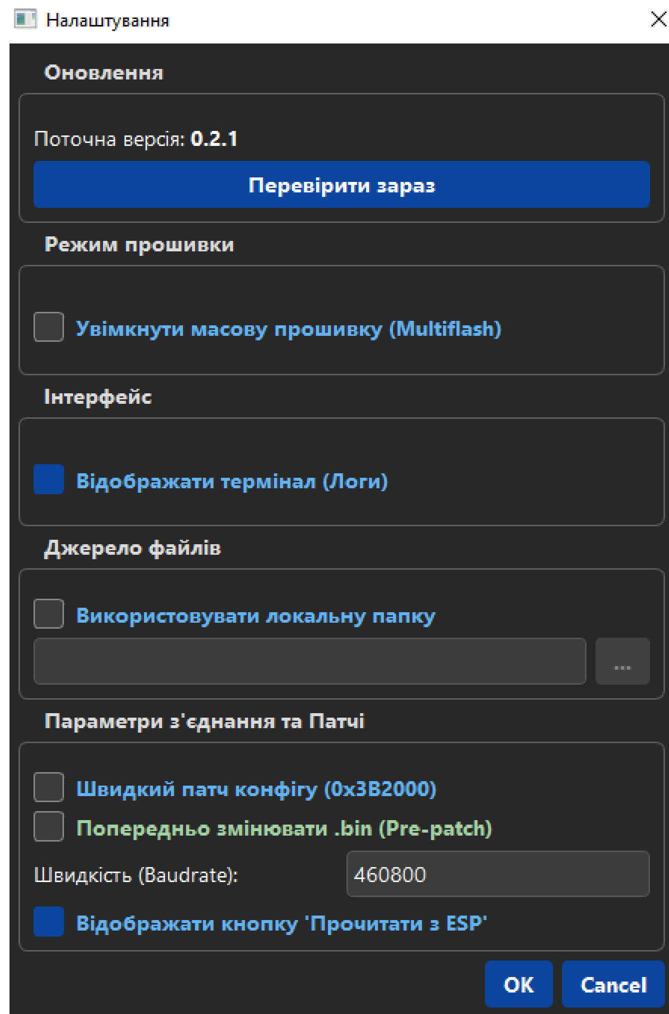


Рис. 3.17. Розгорнута вкладка налаштувань

Тест перевірки оновлень.

Кроки виконання:

1) Відкрити сторінку налаштувань.
2) Натиснути кнопку «Перевірити зараз».

Очікуваний результат: виконується перевірка наявності оновлень, користувач отримує повідомлення про наявність або відсутність нової версії (рис. 3.18.).

Результат відповідає очікуванням.

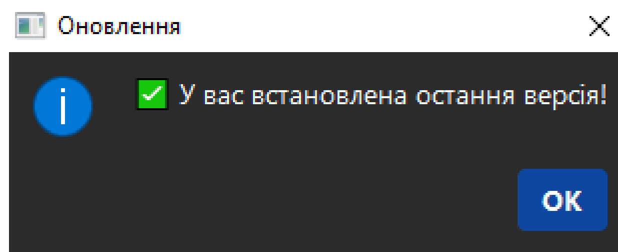


Рис. 3.18. Повідомлення про відсутність нової версії

Тест увімкнення режиму масової прошивки (*Multiflash*)

Кроки виконання:

1) Увімкнути опцію «Увімкнути масову прошивку».
2) Підтвердити налаштування кнопкою *OK*.

Очікуваний результат: застосунок переходить у режим масової прошивки, інтерфейс головного вікна адаптується до вибраного режиму (рис. 3.19.).

Результат відповідає очікуванням.

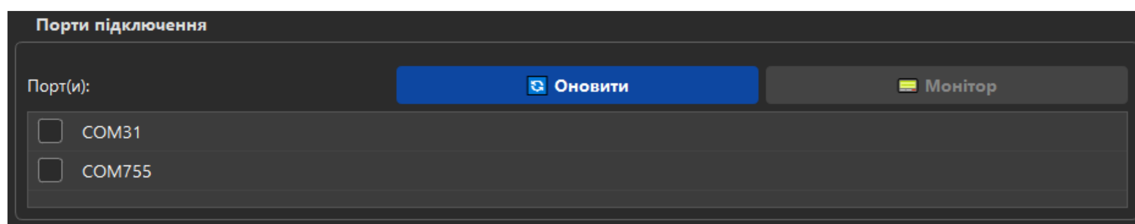


Рис. 3.19. Відображення блока з масовою прошивкою в головному вікні

Тест зміни швидкості з'єднання.

Кроки виконання:

1) Замінити значення параметра *Baudrate* (рис. 3.20.).

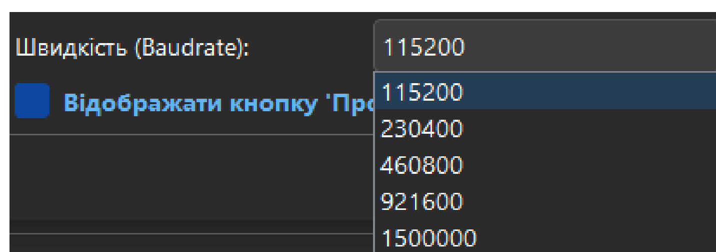


Рис. 3.20. Зміна параметра *Baudrate* з 460800 на 115200

2) Зберегти налаштування.

3) Запустити прошивку.

Очікуваний результат: прошивка виконується з використанням заданої швидкості передачі даних.

Результат:

З встановленим *Baudrate* 460800, прошивка зайняла 41,05 секунду, це можна побачити в логах (рис. 3.21.)

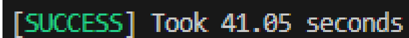


Рис. 3.21. Час, що був потрібен для прошивки з *Baudrate* 460800

З встановленим *Baudrate* 115200, прошивка зайняла 75,64 секунди, це можна побачити в логах (рис. 3.22.)

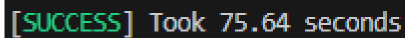


Рис. 3.22. Час, що був потрібен для прошивки з *Baudrate* 115200

3.9. Висновки до розділу

У третьому розділі було детально розглянуто процес практичної розробки програмного засобу для локального та віддаленого оновлення прошивки *ESP32*-пристроїв, що використовуються у складі ретрансляторів і модулів керування наземними станціями. Основна увага була зосереджена на поетапному проєктуванні архітектури застосунку, реалізації його функціональних модулів, інтеграції графічного інтерфейсу користувача та забезпеченні зручності й надійності процесу прошивки.

У ході розробки було реалізовано клієнтський десктопний застосунок на мові програмування *Python* із використанням бібліотеки *PyQt6*, що забезпечило створення сучасного, інтуїтивно зрозумілого та масштабованого графічного

інтерфейсу. Інтерфейс програмного засобу було структуровано за функціональними блоками, що дозволило логічно розділити етапи вибору джерела прошивки, налаштування параметрів з’єднання, роботи з конфігураційними даними та запуску процесу оновлення.

Значну увагу приділено реалізації механізму локальної прошивки через *COM*-порт із використанням інструменту *esptool*. Було продемонстровано процес вибору порту, керування швидкістю передачі даних (*baudrate*), відображення прогресу прошивки та обробку можливих помилок. Реалізований підхід дозволяє адаптувати процес прошивки до різних апаратних умов, забезпечуючи баланс між швидкістю виконання та стабільністю з’єднання.

Окремим важливим результатом розробки стала реалізація механізму роботи з конфігураційними даними пристрою на основі файлової системи *LittleFS*. Забезпечено можливість зчитування конфігурації безпосередньо з *ESP32*, її редагування у форматі *JSON* та повторного запису на пристрій. Також було впроваджено оптимізовані режими роботи з конфігурацією, зокрема швидкий патч визначеного сектору флеш-пам’яті та режим попередньої модифікації бінарного файлу (*Pre-patch*), що суттєво скорочує час оновлення у практичних сценаріях.

Важливим етапом розробки стало впровадження режиму масової прошивки (*Multiflash*), який дозволяє одночасно оновлювати декілька пристроїв. Реалізація цього режиму з використанням багатопотокової обробки забезпечила незалежне керування процесом прошивки для кожного пристрою, відображення індивідуального прогресу та коректну синхронізацію завершення операцій.

Особливу практичну цінність має інтеграція механізму *OTA*-оновлення прошивки. На відміну від традиційного підходу, що передбачає використання веб-інтерфейсу та сторонніх інструментів, у розробленому програмному засобі *OTA*-оновлення реалізовано безпосередньо в клієнтському застосунку.

Користувач отримує можливість виконувати віддалене оновлення прошивки через мережу, вводячи лише *IP*-адресу пристрою та необхідні облікові дані. Такий підхід значно підвищує зручність використання системи, зменшує кількість ручних дій та виключає необхідність роботи з командним рядком.

Реалізована сторінка налаштувань дозволяє гнучко керувати всіма основними параметрами роботи програмного засобу, включно з режимами прошивки, джерелами файлів, відображенням логів та технічними параметрами з'єднання. Забезпечено збереження налаштувань між сеансами роботи, що підвищує зручність експлуатації програмного продукту.

У межах третього розділу також було проведено функціональне тестування основних компонентів системи, зокрема авторизації користувачів, процесу локальної та *OTA*-прошивки, роботи з конфігурацією, режиму масової прошивки та сторінки налаштувань. Результати тестування підтвердили коректність роботи розробленого програмного засобу, його стійкість до типових помилок користувача та здатність коректно обробляти аварійні ситуації.

Таким чином, у третьому розділі було реалізовано та продемонстровано повний цикл створення програмного засобу для керування оновленням прошивки *ESP32*-пристроїв – від проєктування архітектури до практичної реалізації, інтеграції *OTA*-механізмів та тестування. Отриманий результат є функціонально завершеним, масштабованим і придатним для використання в реальних системах керування та телеметрії наземних станцій.

ВИСНОВКИ

У роботі було розглянуто та вирішено актуальну науково-прикладну задачу розробки програмного засобу для локального та віддаленого оновлення прошивки мікроконтролерних пристроїв на базі *ESP32*, що використовуються у складі ретрансляторів і модулів керування наземними станціями. Актуальність теми зумовлена зростаючими вимогами до надійності, оперативності обслуговування та безперервності роботи телеметричних і керувальних систем, особливо в умовах обмеженого фізичного доступу до обладнання.

У першому розділі було виконано аналіз предметної області та теоретичних основ віддаленого оновлення прошивки. Розглянуто принципи роботи ретрансляторів і модулів керування наземними станціями, їх функціональне призначення та особливості експлуатації. Значну увагу приділено характеристикам мікроконтролерів *ESP32*, форматам обміну даними, ролі телеметрії та значенню механізмів *OTA*-оновлення. Проведений аналіз підтвердив, що віддалене оновлення прошивки є критично важливим компонентом сучасних розподілених систем керування, оскільки дозволяє оперативно впроваджувати зміни, усувати помилки та підвищувати рівень кібер- та експлуатаційної безпеки.

У другому розділі виконано огляд програмних та апаратних засобів, що використовуються в процесі розробки. Обґрунтовано вибір мови програмування *Python* як основної мови реалізації клієнтського застосунку, а також середовища розробки *Visual Studio Code*. Розглянуто застосування бібліотек *PyQt6*, *esptool*, *pyserial*, *littlefs-python* та інших компонентів, необхідних для взаємодії з *ESP32*, роботи з файловими системами та побудови графічного інтерфейсу користувача.

Окремо проаналізовано формат *JSON* як універсальний засіб зберігання та передачі конфігураційних даних, що забезпечує гнучкість, читабельність і сумісність системи. Також у другому розділі розглянуто принципи *OTA*-оновлення та їх інтеграцію у клієнтські програмні рішення.

Третій розділ було присвячено практичній реалізації програмного засобу. Поетапно описано архітектуру застосунку, реалізацію локальної прошивки через

COM-порт, механізм роботи з конфігураційними даними на базі файлової системи *LittleFS*, режими швидкого оновлення та попереднього патчінгу прошивки. Значну увагу приділено реалізації режиму масової прошивки, що дозволяє одночасно оновлювати декілька пристроїв, а також інтеграції *OTA*-оновлення прошивки безпосередньо в клієнтський застосунок. Такий підхід усуває необхідність використання сторонніх веб-інтерфейсів і командного рядка, що суттєво підвищує зручність і безпеку використання системи.

У результаті виконання роботи було створено функціонально завершений програмний продукт, який забезпечує повний цикл оновлення прошивки *ESP32*-пристроїв як у локальному, так і у віддаленому режимах. Розроблений застосунок підтримує збереження сесій, керування користувачами, відображення прогресу виконання операцій, обробку помилок та логування подій. Проведене функціональне тестування підтвердило коректність роботи основних модулів системи, її стійкість до типових експлуатаційних помилок та придатність до використання в реальних умовах.

Практична цінність отриманих результатів полягає у можливості застосування розробленого програмного засобу в реальних системах керування, телеметрії та зв’язку, де фізичний доступ до обладнання є ускладненим або неможливим. Запропоновані підходи можуть бути використані для подальшого розвитку систем віддаленого адміністрування, автоматизованого моніторингу та оновлення мікроконтролерних пристроїв.

Окремо слід відзначити, що розроблений програмний засіб орієнтований не лише на вирішення поточних експлуатаційних задач, а й на подальше масштабування. Архітектура застосунку побудована з урахуванням можливості розширення функціоналу, зокрема додавання підтримки інших мікроконтролерних платформ, нових протоколів зв’язку або інтеграції з серверними системами моніторингу. Це дозволяє розглядати запропоноване рішення як основу для створення комплексної системи віддаленого керування пристроями.

Важливою перевагою реалізованого підходу є зниження людського фактору під час виконання операцій оновлення прошивки. Автоматизація процесів вибору

прошивки, перевірки з'єднання, передачі даних та контролю результатів зменшує ймовірність помилок, пов'язаних із некоректними діями користувача. Це особливо актуально для умов експлуатації, де оновлення виконують непрофільні спеціалісти або користувачі без глибоких технічних знань.

З точки зору експлуатаційної безпеки, реалізація механізмів резервного оновлення, перевірки цілісності прошивки та можливості відкату до попередньої версії значно підвищує надійність системи. Це дозволяє мінімізувати ризики повної втрати працездатності пристрою внаслідок збоїв під час оновлення, що є критично важливим для систем зв'язку та керування, які працюють у безперервному режимі.

Важливим аспектом розробленого програмного засобу є врахування питань інформаційної безпеки під час віддаленого оновлення прошивки. У процесі реалізації *OTA*-механізмів передбачено використання контрольованого доступу до пристроїв, автентифікації користувачів та передачі даних через захищені мережеві з'єднання. Використання токенів доступу, обмеження ролей користувачів та перевірка цілісності прошивок дозволяють зменшити ризик несанкціонованого втручання, підміни програмного забезпечення або компрометації пристрою. Це особливо важливо для систем наземних станцій і ретрансляторів, які можуть бути об'єктами підвищеної уваги з боку сторонніх осіб.

Окрім технічних переваг, запропоноване рішення має суттєву економічну ефективність. Використання віддаленого оновлення прошивки дозволяє значно зменшити витрати на фізичне обслуговування обладнання, логістику та простої систем у разі виявлення програмних помилок. Можливість оперативного внесення змін до програмного забезпечення без демонтажу пристроїв сприяє скороченню часу реагування на позаштатні ситуації та підвищенню загальної ефективності експлуатації. Таким чином, впровадження розробленого програмного засобу є доцільним не лише з технічної, а й з економічної точки зору.

Перспективами подальших досліджень і розвитку роботи є розширення функціоналу *OTA*-оновлення за рахунок впровадження криптографічного підпису прошивок, централізованого серверного керування версіями програмного забезпечення, а також автоматизованого збору телеметричних даних для аналізу

стану пристроїв у реальному часі. Реалізація таких можливостей дозволить створити повноцінну екосистему віддаленого адміністрування мікроконтролерних пристроїв, що відповідає сучасним вимогам до безпеки та надійності.

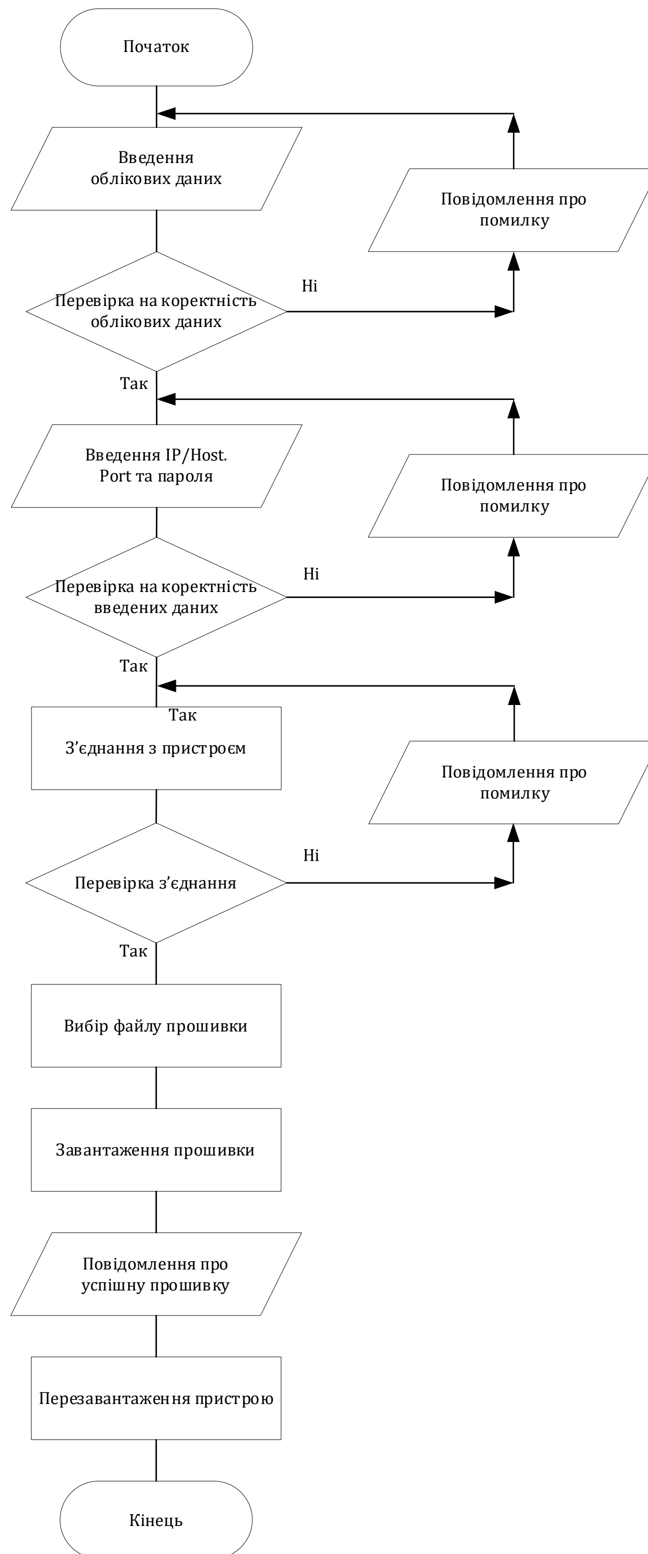
Поставлену мету роботи досягнуто, а всі сформульовані завдання виконано у повному обсязі. Отримані результати мають як практичну, так і наукову цінність та можуть бути використані як основа для подальших досліджень і вдосконалення систем віддаленого оновлення прошивки в розподілених керувальних комплексах.

СПИСОК БІБЛІОГРАФІЧНИХ ПОСИЛАНЬ ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Слободян О. Положення про кваліфікаційні роботи (проекти) здобувачів вищої освіти Національного авіаційного університету. – К.: Видавництво НАУ, 2024. – 62с.
2. ДСТУ ГОСТ 3008:2015 «Документація. Звіти у сфері науки і техніки. Структура і правила оформлення».
3. *IEEE Std* 802.11™-2020**.** *IEEE Standard for Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications.*
4. Грабко В. В., Кучерук В. Ю., Возняк О. М. Мікропроцесорні системи керування електроприводами: навч. посіб. – Вінниця: Вінницький національний технічний університет, 2008. – 149 с.
5. Нікітчук А. В. Програмування вбудованих систем Інтернету речей: конспект лекцій. – Київ: КПІ ім. Ігоря Сікорського, 2024. – 120 с.
6. *Barr M., Massa A. Programming Embedded Systems. – 2nd ed. – O'Reilly Media*, 2006. – 354 *p.*
7. *Stallings W. Network Security Essentials: Applications and Standards*. – 6*th ed*. – *Pearson*, 2017. – 432 *p*.
8. *Kolban N. Kolban's Book on ESP32***.** *– Leanpub*, 2019. – 560 *p.*
9. *Valvano J. Embedded Systems: Real-Time Operating Systems for ARM Cortex-M Microcontrollers. – Austin: University of Texas*, 2017. – 512 *p*.
10. Прокопов Р. О. Передавач телеметричної інформації: дис. за спец. 172 «Телекомунікації та радіотехніка» / Нац. техн. ун-т України «КПІ ім. Ігоря Сікорського». – Київ, 2024. – 79 с.
11. Мельник А. О. Архітектура комп'ютера: підруч. для студ. вищ. навч. закл. – Луцьк: Волинська обласна друкарня, 2008. – 470 с.
12. Гольц В. Д., Ірха М. С. Телекомунікаційні та інформаційні мережі: навч. посіб. – Київ: КПІ ім. Ігоря Сікорського, 2021. – 250 с.
13. Д. Д. Татарчук, Ю. В. Діденко. Мікропроцесори та мікроконтролери: курс лекцій: навч. посіб. / «КПІ ім. Ігоря Сікорського». – Київ, 2020. – 238 с.

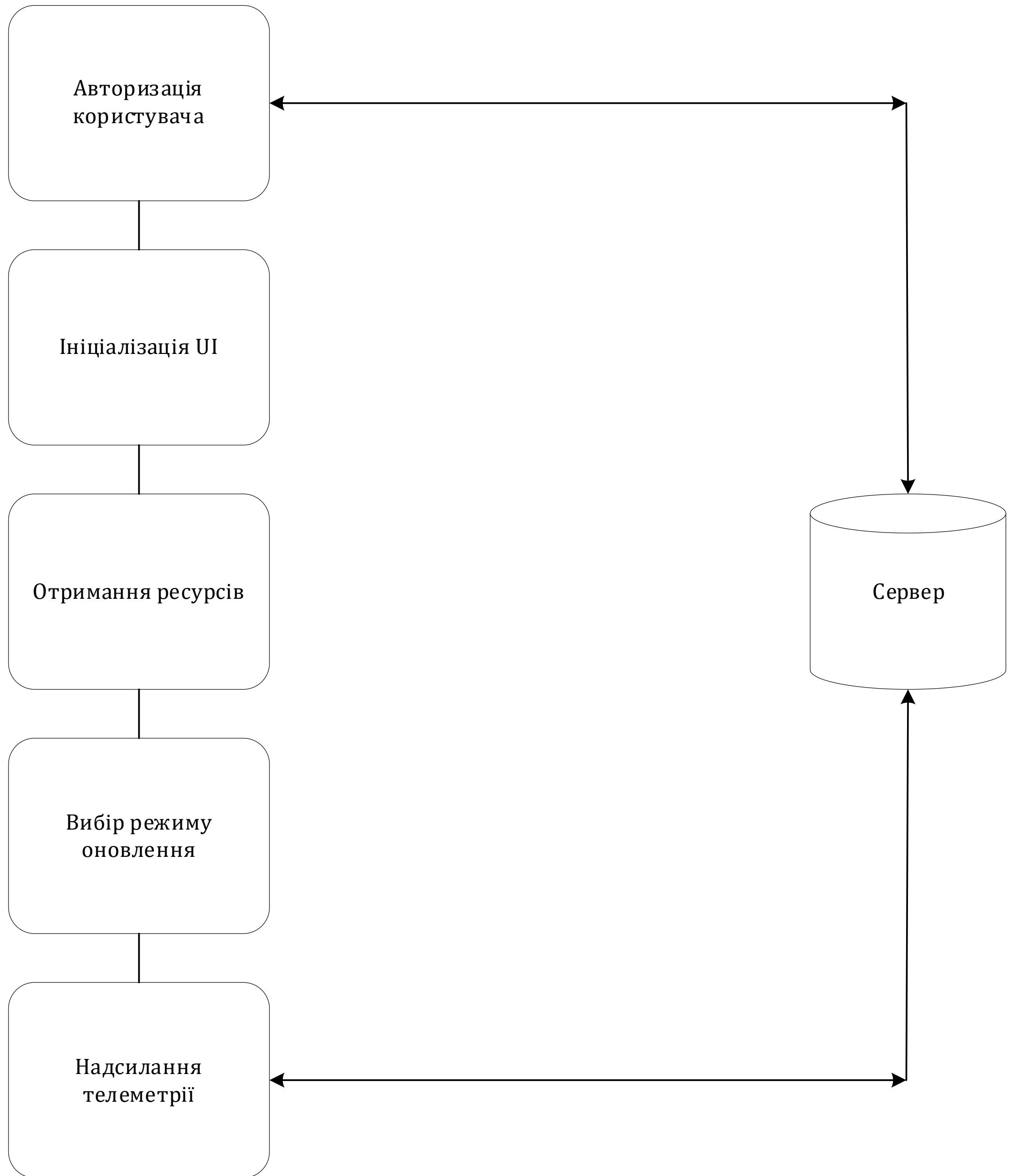
14. *Espressif Systems. ESP32 Technical Reference Manual* [Електронний ресурс]. – *URL*: *https://www.espressif.com/sites/default/files/documentation/esp32_technical_reference_manual_en.pdf* (дата звернення: 01.10.2025).
15. *Espressif Systems. ESP-IDF Programming Guide* [Електронний ресурс]. – *URL: https://docs.espressif.com/projects/esp-idf* (дата звернення: 13.10.2025).
16. *Espressif Systems. Over-the-Air Updates (OTA)* [Електронний ресурс]. – *URL: https://docs.espressif.com/projects/esp-idf* (дата звернення: 17.10.2025).
17. *Visual Studio Code Documentation*. [Електронний ресурс]. *URL*: *https://code.visualstudio.com/docs* (дата звернення: 18.10.2025).
18. *Python Software Foundation. Python Documentation* [Електронний ресурс]. – *URL*: *https://docs.python.org* (дата звернення: 18.10.2025).
19. *LittleFS Project. LittleFS Documentation* [Електронний ресурс]. – *URL*: *https://github.com/littlefs-project/littlefs* (дата звернення: 22.10.2025).
20. *Arduino*. *Arduino OTA Update Guide* [Електронний ресурс]. – *URL: https://www.arduino.cc* (дата звернення: 03.11.2025).
21. *Rescorla E. HTTP Over TLS (RFC 2818)* [Електронний ресурс]. – *URL*: *https://www.rfc-editor.org/rfc/rfc2818* (дата звернення: 03.11.2025).
22. *GitHub Docs*. *GitHub – Version Control and Collaboration* [Електронний ресурс]. – *URL*: *https://docs.github.com* (дата звернення: 04.11.2025).
23. *Wolf M. Computers as Components: Principles of Embedded Computing System Design*. – 4*th ed. – Morgan Kaufmann*, 2016. – 800 *p*.
24. *ARM Ltd. Firmware Update and Security Guidelines* [Електронний ресурс]. – *URL: https://developer.arm.com* (дата звернення: 15.11.2025).
25. *National Instruments. Remote Monitoring and Firmware Update in Embedded Systems* [Електронний ресурс]. – *URL: https://www.ni.com* (дата звернення: 20.11.2025).

26. *MathWorks. Telemetry Systems Overview* [Електронний ресурс]. – *URL: https://www.mathworks.com* (дата звернення: 20.11.2025).
27. *FreeRTOS. FreeRTOS Kernel Documentation* [Електронний ресурс]. – *URL: https://www.freertos.org* (дата звернення: 23.11.2025).



					КАІ 25 06 68 001 ПМ			
Зм.	Лист	№ документа	Підпис	Дата	Програмний засіб (схема алгоритму)	Літера	Маса	Масштаб
Виконала		Задорожна О. В.				Д		
Керівник		Литвиненко О Є.				Лист 1	Листів 1	
Консульт.						М-126-24-1-ІТ		
Н. контроль		Тупота Є.В.						
Затвердив		Нечипорук О.П..						

Взаємодія компонентів програмного засобу



Зм.	Лист	№ документа	Підпис	Дата
Виконала		Задорожна О. В.		
Керівник		Литвиненко О. Є.		
Консульт.				
Н. контроль		Тупота Є.В.		
Зав. каф.		Нечипорук О. П.		

КАІ 25 06 68 002 ПЛ

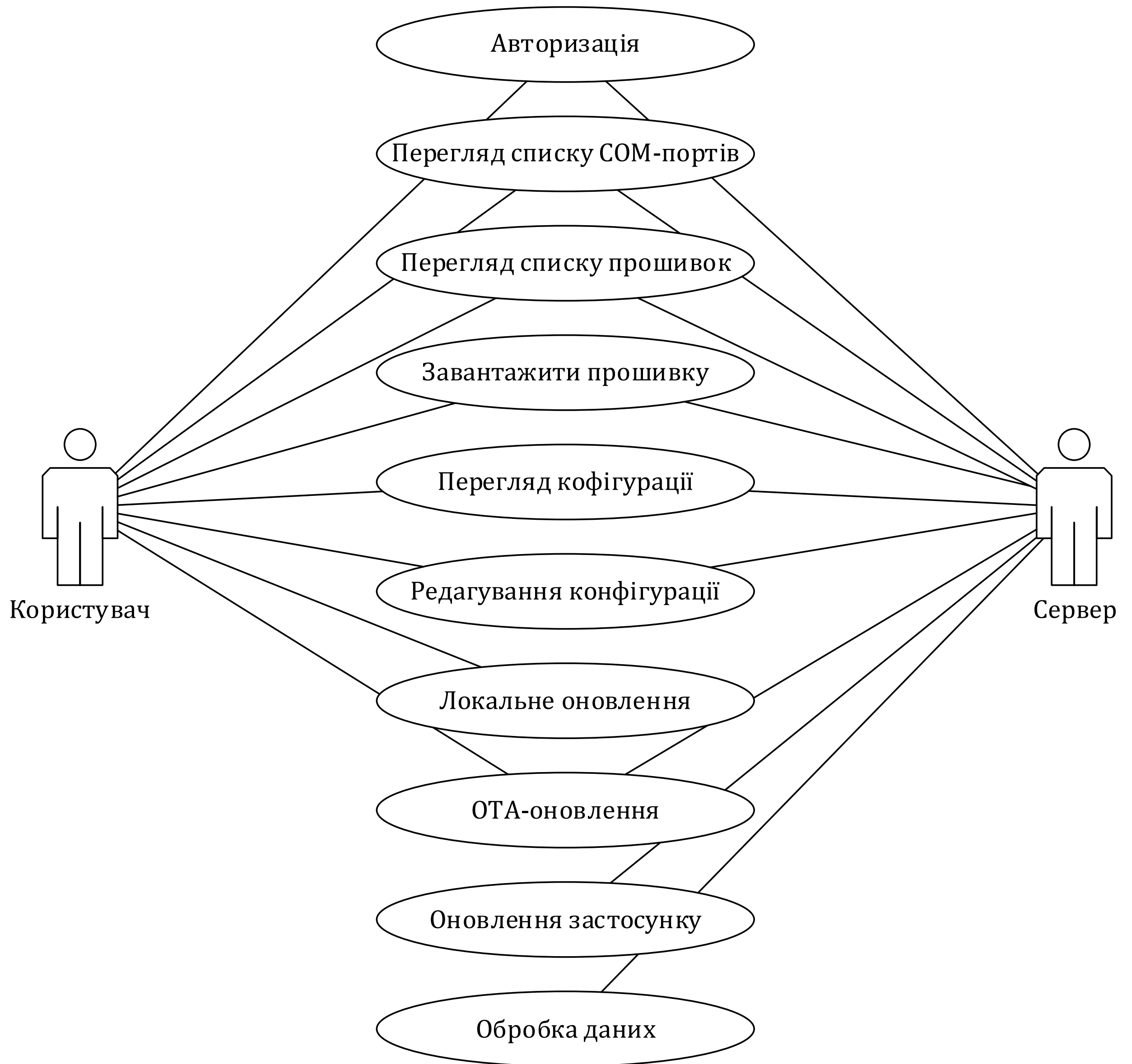
Взаємодія компонентів програмного засобу

Літера	Маса	Масштаб
Д		

Лист 1 | Листів 1

М-126-24-1-ІТ

***Use case* діаграма програмного застосунку**



					КАІ 25 06 68 003 ПЛ			
Зм.	Лист	№ документа	Підпис	Дата	Use case діаграма програмного застосунку	Літера	Маса	Масштаб
Виконала		Задорожна О. В.				Д		
Керівник		Литвиненко О. Є.				Лист 1	Листів 1	
Консульт.						М-126-24-1-ІТ		
Н. контроль		Тупота Є.В.						
Затвердив		Нечипорук О.П.						