

**НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ ІМЕНІ ІГОРЯ
СІКОРСЬКОГО»**

Навчально-науковий інститут атомної та теплової енергетики

Кафедра інженерії програмного забезпечення в енергетиці

ДО ЗАХИСТУ ДОПУЩЕНО

В.о. завідувача кафедри

Олександр КОВАЛЬ

«_____» _____ 202_р.

Дипломна робота

на здобуття ступеня бакалавра

**за освітньо-професійною програмою «Інженерія програмного
забезпечення інтелектуальних кібер-фізичних систем в енергетиці»
спеціальності 121 Інженерія програмного забезпечення**

на тему: «програмне забезпечення моделювання віддаленого контролю»

Виконав :

студент IV курсу, групи ТВ-22

Сівоздрав Артем Миколайович

(Підпис)

Керівник:

Ст. викладач Сарибоба Г.В.

(Підпис)

Рецензент:

(посада, науковий ступінь, вчене звання, прізвище та ініціали)

(Підпис)

Засвідчую, що у цій дипломній роботі немає
запозичень з праць інших авторів без
відповідних посилань.

Студент _____
(Підпис)

Київ – 2026

**Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»**

Навчально-науковий інститут атомної та теплової енергетики
Кафедра інженерії програмного забезпечення в енергетиці
Рівень вищої освіти перший (бакалаврський)
Спеціальність 121 Інженерія програмного забезпечення
Освітньо-професійна програма «Інженерія програмного забезпечення
інтелектуальних кібер - фізичних систем в енергетиці»

ЗАТВЕРДЖУЮ В.о завідувача кафедри
Олександр КОВАЛЬ
(Підпис)

« ____ » _____ 202__ р.

**ЗАВДАННЯ
на дипломну роботу студенту**

Сівоздраву Артему Миколайовичу

1. Тема роботи

Програмне забезпечення моделювання віддаленого контролю керівник роботи ст.
викладач Сарибоба Ганна Володимирівна.

2. Строк подання студентом роботи: _____

3. Вихідні дані до роботи: мови програмування C++, GO, генератор сценаріїв
складання Cmake, операційна система Debian, програмне забезпечення ExpressLRS
Configurator, бібліотека OpenCV, середовище розробки VS Code.

4. Зміст (дипломної роботи) пояснювальної записки (перелік завдань, які потрібно
розробити): реалізування схеми фізичного передавання даних між БПЛА та
пристроєм комунікації, розробка системи керування дроном, створення
клієнтського ПЗ системи керування, опрацювання етапів обробки відеокادру,
контсруювання телеметричного кадру csrf протоколу.

5. Перелік ілюстративного матеріалу: uml даіграми, що демонструють будову та
принцип роботи розроблених файлів, знімки екрану готової системи та
використаного прикладного програмного забезпечення.

6. Дата видачі завдання «31» жовтня 2025р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів виконання дипломної роботи	Строки виконання етапів роботи	Примітка
1	Отримання завдання	31.10.2025	Виконано
2	Дослідження предметної області	31.10.2025 – 01.11.2025	Виконано
3	Постановка вимог до проєктування системи	01.11.2025 – 30.11.2025	Виконано
4	Дослідження існуючих рішень	30.11.2025 – 31.12.2025	Виконано
5	Розробка програмного продукту	01.01.2026 – 03.31.2025	Виконано
6	Тестування	01.04.2026 – 01.05.2026	Виконано
7	Захист програмного продукту	19.05.2026	Виконано
8	Оформлення дипломної роботи	01.08.2026 – 08.06.2026	Виконано
9	Передзахист	05.06.2026	Виконано
10	Захист		

Студент

(Підпис)

Артем Сівоздрав

Керівник

(Підпис)

Ганна Сарибога

РЕФЕРАТ

Структура та обсяг дипломної роботи. Робота містить 60 сторінок, 15 рисунків, 2 додатки та 10 посилань.

Метою роботи є розробка програмного забезпечення моделювання віддаленого контролю.

Для досягнення поставленої мети виконано такі завдання:

- Проведено аналіз предметної області;
- Визначено функціональні та нефункціональні вимоги;
- Створено структуру системи віддаленого керування БПЛА;
- Реалізовано схеми фізичного передавання даних між БПЛА та пристроєм комунікації;
- Розроблено ПЗ системи керування дроном, клієнтської та серверної частини;
- Опрацьовано етапи обробки відеокадру;
- Сконструйовано телеметричний кадр crsf протоколу для керування дроном;
- Протестовано ПЗ та виправлено недоопрацювання.

Практичне значення одержаних результатів полягає у створенні робочого програмно-апаратного рішення, яке знімає обмеження дальності класичних FPV-систем та дозволяє керувати безпілотним літальним апаратом через публічну мережу Інтернет із будь-якої географічної точки, що в умовах військового застосування БПЛА дає змогу винести оператора у безпечний тил при мінімальній вартості апаратної частини.

Ключові слова : БПЛА, FPV, відео, ELRS, CRSF, приймач, передавач, відеопередавач, телеметрія, UDP, TCP, JPEG.

ABSTRACT

Structure and scope of the thesis. The work contains 60 pages, 15 figures, 2 appendices and 10 references.

The aim of the work is to develop software for modeling remote control.

To achieve this goal, the following tasks were performed:

- An analysis of the subject area was carried out;
- Functional and non-functional requirements were defined;
- The structure of the UAV remote control system was created;
- Schemes of physical data transmission between the UAV and the communication device were implemented;
- Software for the drone control system, client and server parts, was developed;
- The stages of video frame processing were elaborated;
- The telemetry frame of the CRSF protocol for drone control was constructed;
- The software was tested and shortcomings were corrected.

The practical significance of the obtained results lies in the creation of a working software and hardware solution that removes the range limitations of classical FPV systems and allows controlling an unmanned aerial vehicle via the public Internet from any geographical location, which in the context of the military use of UAVs makes it possible to relocate the operator to a safe rear area at minimal cost of the hardware part.

Keywords: UAV, FPV, video, ELRS, CRSF, receiver, transmitter, video transmitter, telemetry, UDP, TCP, JPEG

ЗМІСТ

ВСТУП	7
МЕТА ПРОГРАМНОГО ПРОДУКТУ	8
Висновки до розділу 1	9
АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ	10
2.1 Особливості дистанційного керування БПЛА	10
2.2 Огляд протоколів зв'язку та керування	11
2.3 Обґрунтування архітектури "Клієнт – Релейний сервер – Клієнт"	14
2.4 Розбір ELRS радіо протоколу	15
2.5 Фізичне передавання даних між БПЛА та комунікаційним пристроєм.....	17
Висновки до розділу 2	20
ЗАСОБИ РОЗРОБКИ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ.....	21
3.1 Мова програмування C++	21
3.2 Мова програмування GO.....	23
3.3 Операційна система Debian.....	24
3.4 Система збірки Stake	25
3.5 Програмне забезпечення ExpressLRS Configurator.....	26
Висновки до розділу 3	27
ОПИС ПРОГРАМНОЇ РЕАЛІЗАЦІЇ	28
4.1 Реалізація клієнтського застосунку.....	28
4.1.1 Файл main.cpp	28
4.1.2 Файл network.cpp	34
4.1.3 Файл camera.cpp.....	40
4.1.4 Файл serial_rx.cpp	43
4.1.5 Файл joystick.cpp.....	46
4.2 Реалізація серверної частини	48
Висновки до розділу 4	50
5.1 Інтерфейс користувача та система команд.....	51
5.2 Алгоритми розгортання клієнтських терміналів	54
5.3 Тестування програмного продукту	56
Висновки до розділу 5	58
ВИСНОВКИ.....	59
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ	60
ДОДАТОК А.....	61
ДОДАТОК Б	89

ВСТУП

Сьогодні технології безпілотних літальних апаратів розвиваються надзвичайно швидко. Згідно з оцінками, наведеними у фундаментальній праці з проєктування безпілотних систем [3], галузь демонструє стійке зростання, а за прогнозами експертів світовий ринок безпілотників найближчим часом може перевищити сто мільярдів доларів.

Якщо розглядати ситуацію в Україні, ця тенденція набуває ще більшого і життєво важливого значення. Безпілотні апарати надали українській армії значну перевагу, що дозволяє ефективно стримувати набагато більшого супротивника. Розвиток цих технологій безпосередньо впливає на підвищення бойової потужності країни. Якщо раніше дрони були дефіцитом, то сьогодні їх вітчизняне виробництво досягло вражаючих масштабів.

Однак на тлі зростання кількості техніки виникла інша гостра проблема — нестача кваліфікованих пілотів. Зрозуміло, що збільшення кількості операторів прямо пропорційно збільшує кількість успішно виконаних завдань. Але головною перешкодою для масового залучення нових людей є надзвичайно висока небезпека для їхнього життя. Звичайні системи радіокерування вимагають, щоб пілот знаходився відносно близько до зони польоту.

Щоб безпечно збільшити кількість екіпажів, необхідно знайти спосіб максимально збільшити відстань між людиною та самим літальним апаратом, перенісши робоче місце оператора в безпечний тил.

Саме створення такого програмного рішення і є головною метою цієї дипломної роботи. У рамках проєкту було розроблено систему, яка дозволяє дистанційно керувати безпілотником з будь-якої точки світу. Зв'язок забезпечується через глобальну мережу інтернет, що повністю знімає обмеження щодо відстані, які притаманні звичайному радіозв'язку. При використанні такого підходу єдиним технічним обмеженням залишається лише фізична затримка передачі даних інтернет-з'єднання.

1 МЕТА ПРОГРАМНОГО ПРОДУКТУ

Головною метою дипломного проєкту є розробка та програмно-апаратна реалізація надійного двостороннього каналу зв'язку, що забезпечує дистанційне керування безпілотним літальним апаратом через глобальну мережу інтернет. Розроблювана система повинна гарантувати ефективну взаємодію в режимі реального часу, надаючи оператору безперервний відеопотік з бортової камери та забезпечуючи миттєву трансляцію команд керування на борт літального апарата.

Сучасні безпілотні платформи переважно використовують спеціалізовані низькочастотні протоколи радіокерування для передачі телеметрії, а також аналогові стандарти відеозв'язку. Використання аналогового відеосигналу зумовлене його мінімальною затримкою та високою стійкістю до втрат на великих дистанціях за умови відсутності фізичних перешкод. Проте стандартні комп'ютерні системи не оснащені відповідними апаратними модулями для безпосередньої демодуляції та обробки таких специфічних радіочастотних сигналів.

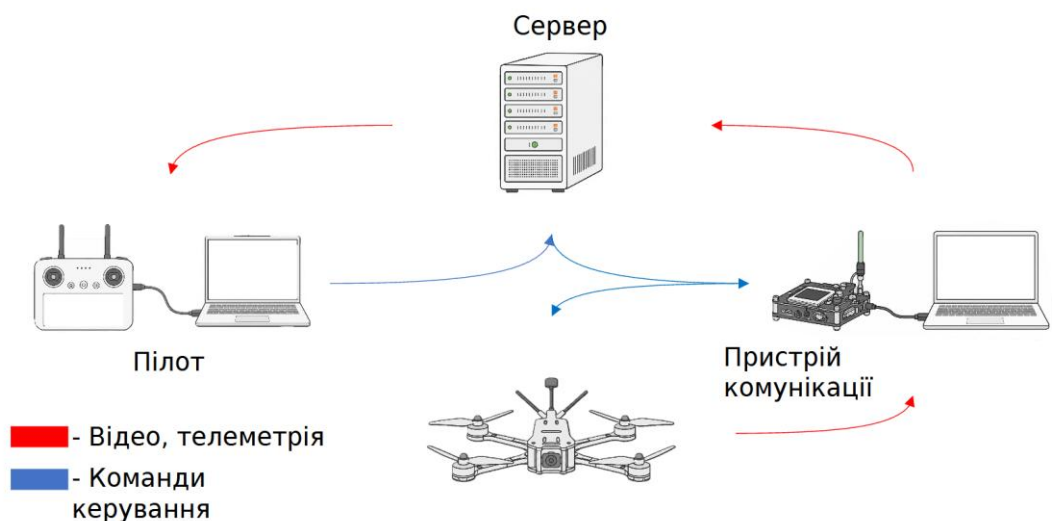


Рисунок 1.1 – Загальна схема роботи системи

Для подолання цієї апаратної несумісності архітектура системи вимагає розробки та інтеграції проміжного комунікаційного пристрою (шлюзу). Цей апаратний вузол повинен виконувати роль зв'язуючої ланки, що фізично приймає аналоговий відеопотік від літального апарата та оцифровує його, а також одночасно конвертує цифрові команди, отримані від комп'ютера, у відповідний радіоформат для подальшої передачі на польотний контролер.

Оскільки оператор та безпілотною можуть знаходитися на значній географічній відстані і не мають прямого мережевого з'єднання, критичною вимогою є використання централізованого маршрутизуючого сервера, доступного для обох вузлів.

Відповідно, безпосередньою метою програмного продукту є організація високошвидкісної та надійної маршрутизації цих інформаційних потоків. Програмне забезпечення повинно безперервно зчитувати маніпуляції пілота з пристроїв керування, формувати з них мережеві пакети, перенаправляти їх через сервер до комунікаційного пристрою дрона, а у зворотному напрямку — приймати оцифрований відеопотік і транслювати його на монітор оператора з мінімально можливою затримкою. Схематичний малюнок системи, продемонстрований на рисунку 1.1

Висновки до розділу 1

У першому розділі сформульовано мету дипломного проєкту — розробка двостороннього каналу зв'язку для дистанційного керування БПЛА через глобальну мережу. Обґрунтовано необхідність використання проміжного комунікаційного пристрою для подолання апаратної несумісності між стандартним FPV-обладнанням та інтернет-інфраструктурою, а також доцільність централізованого ретранслюючого сервера для обходу обмежень NAT та забезпечення стабільного маршрутизованого з'єднання між оператором і дроном незалежно від їхнього географічного розташування.

2 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

2.1 Особливості дистанційного керування БПЛА

Традиційно керування маневреними безпілотними літальними апаратами (зокрема FPV-дронами) здійснюється за допомогою прямого радіозв'язку між наземним пультом оператора та бортовим приймачем. Цей процес вимагає безперервного двостороннього обміну інформацією в режимі реального часу: передачі керуючих команд від пілота до літального апарата (Uplink) та отримання зворотного потоку відео і телеметрії (Downlink).

З прикладної та апаратної точок зору, такий зв'язок реалізується через два фізично розділені радіоконтури, кожен з яких працює у своєму діапазоні та має власні технічні особливості. Контур керування зазвичай використовує більш низькі частоти (наприклад, діапазони 868/915 МГц або 2.4 ГГц), оскільки довші радіохвилі забезпечують кращу проникаючу здатність сигналу та стійкість до перешкод. Натомість контур відеозв'язку вимагає значно ширшої смуги пропускання для передачі зображення. Тому бортові відеопередавачі (VTX) найчастіше використовують частоту 5.8 ГГц в аналоговому форматі. Саме аналоговий сигнал гарантує нульову затримку кодування (latency), що є критично важливим для просторового орієнтування оператора під час польоту на великих швидкостях.

Головна вразливість такої класичної архітектури полягає у фізиці поширення радіохвиль. Якщо сигнал керування ще здатен частково проникати крізь перешкоди, то високочастотний відеосигнал 5.8 ГГц є вкрай чутливим до середовища поширення. Він вимагає практично ідеальної прямої видимості (VLOS — Visual Line of Sight) між передавальною та приймальною антенами. Будь-які фізичні перешкоди, такі як складний рельєф місцевості, щільна міська забудова, пагорби або лісові масиви, викликають швидке загасання, розсіювання та дифракцію сигналу. Для оператора це означає миттєву втрату зображення (так звану "сліпоту") ще до того, як пропаде зв'язок керування.

Ці фізичні особливості радіочастот жорстко лімітують максимальну відстань ефективного застосування БПЛА. Щоб підтримувати стабільний відеозв'язок і не втратити дрон, екіпаж змушений фізично перебувати відносно близько до зони польоту, часто розгортаючи антени на відкритій місцевості. Крім того, безперервне випромінювання потужного радіопередавача на пульті керування стає помітним маркером для систем радіоелектронної розвідки. Таким чином, ключовим інженерним викликом у сучасній галузі безпілотних систем є пошук методів переходу до стандарту польотів поза зоною прямої видимості (BVLOS), щоб абстрагуватися від обмежень прямого радіоканалу та забезпечити віддалену роботу оператора.

2.2 Огляд протоколів зв'язку та керування

Надійна та швидка взаємодія між наземною станцією керування та безпілотним літальним апаратом забезпечується спеціалізованими протоколами зв'язку. У контексті архітектури БПЛА ці стандарти поділяються на радіопротоколи (визначають тип модуляції в ефірі) та послідовні цифрові протоколи (визначають формат обміну даними між радіоприймачем та польотним контролером по кабелю).

Еволюція систем телекерування супроводжувалася переходом від повільних аналогових стандартів до високошвидкісних цифрових пакетних рішень.

Аналогові та ранні цифрові стандарти (PWM, PPM, SBUS)

Історично першими застосовувалися широтно-імпульсна (PWM) та позиційно-імпульсна (PPM) модуляції, а згодом — послідовний цифровий протокол SBUS. Сьогодні ці технології вважаються застарілими для маневрених безпілотників.

Таблиця 2.1 – Переваги та недоліки старих протоколів

Переваги	Недоліки
Висока сумісність зі стрим обладнанням та простота апаратної реалізації	Односпрямованість (відсутність повноцінної двосторонньої телеметрії без додаткових проводів)
	Відсутність надійної перевірки цілісності пакетів даних у старих модуляціях

Сучасні далекобійні пропріетарні системи (TBS Crossfire / формат CRSF)

Протокол CRSF здійснив революцію у сфері пілотування на великих дистанціях, об'єднавши радіопередачу (на частотах 868/915 МГц) та швидкий серійний протокол зв'язку з контролером.

Таблиця 2.2 – Переваги та недоліки протоколу TBS Crossfire

Переваги	Недоліки
Висока швидкість оновлення пакетів (до 150 Гц) та низька затримка	Закрита архітектура, що унеможливорює модифікацію коду сторонніми розробниками
Двостороння комунікація по одному кабелю (команди та телеметрія)	Висока вартість апаратних модулів через монополію виробника
Надійна перевірка цілісності даних алгоритмом CRC8	
Динамічне керування потужністю передавача залежно від якості зв'язку	

Відкриті системи з наднизькою затримкою (ExpressLRS)

ExpressLRS (ELRS) — це сучасний протокол з відкритим вихідним кодом (Open Source), побудований на базі модуляції LoRa (Long Range). Повна технічна документація протоколу та його реалізацій публікується на офіційному вебресурсі проекту [6]. На відміну від класичного застосування LoRa для повільних пристроїв, ELRS оптимізовано спеціально для досягнення мінімальної затримки. Важливо зазначити, що приймачі ELRS використовують протокол CRSF для зв'язку з польотним контролером, забезпечуючи сумісність екосистем.

Таблиця 2.3 – переваги та недоліки протоколу ELRS

Переваги	Недоліки
Безпрецедентна швидкість (частота оновлення до 1000 Гц на 2.4 ГГц)	Складність початкового налаштування
Надвисока чутливість приймачів, що дозволяє підтримувати зв'язок на десятках кілометрів при мінімальній потужності передавача	Необхідність суворого контролю збігу мажорних версій прошивок на приймачі та передавачі
Відкрита архітектура, що стимулює появу великої кількості доступного обладнання від різних виробників	

Тенденції використання протоколів

На сучасному етапі розвитку галузі (зокрема у сегменті FPV та кастомних БПЛА) спостерігається абсолютне домінування технологій наступного покоління. Завдяки відкритості та високим радіотехнічним характеристикам, стандарт ExpressLRS займає понад дві третини ринку сучасного обладнання. TBS Crossfire утримує позиції у сегменті професійних далекобійних систем, тоді як застарілі протоколи типу SBUS залишаються переважно в ніші бюджетних або консервативних комерційних платформ.

2.3 Обґрунтування архітектури "Клієнт – Релейний сервер – Клієнт"

Як зазначено у класичній праці з комп'ютерних мереж [2], організація зв'язку між вузлами через глобальну мережу майже завжди виключає можливість прямого з'єднання за архітектурою "точка-точка". Це зумовлено тим, що наземна станція керування та комунікаційний пристрій дрона зазвичай використовують мобільний інтернет або знаходяться за побутовими маршрутизаторами. Відповідно, вони приховані за механізмами трансляції мережевих адрес (NAT) і не мають статичних публічних IP-адрес.

Для розв'язання цієї мережевої проблеми у розробленій системі застосовано архітектуру з проміжним релейним сервером. Сервер розгортається у хмарному середовищі, має виділену публічну IP-адресу та виступає центральним вузлом комунікації. Його використання обумовлене двома критичними задачами:

- **Обхід мережевих екранів** - Сервер дозволяє реалізувати механізм проходження NAT, формальний опис якого закріплений у відповідному стандарті IETF [8]. Обидва клієнти самостійно ініціюють вихідне підключення до сервера, відкриваючи порти на своїх маршрутизаторах.
- **Транзитна маршрутизація** - Після встановлення з'єднання сервер виконує роль прозорого високошвидкісного моста. Він приймає інформаційні пакети (оцифроване відео від дрона або команди керування від пілота) і миттєво ретранслює їх відповідному адресату.

Такий архітектурний підхід є єдиним надійним способом гарантувати встановлення стабільного двостороннього з'єднання незалежно від типу інтернет-провайдера на кінцевих точках, додаючи при цьому лише мінімальний апаратний час на пересилку пакетів.

2.4 Розбір ELRS радіо протоколу

ExpressLRS — це відкритий протокол радіозв'язку для систем керування безпілотними апаратами, що побудований на основі LoRa-модуляції. LoRa забезпечує значно більшу дальність при малій потужності сигналу порівняно з традиційними рішеннями, а відкритість протоколу дозволяє гнучко налаштовувати його параметри — частоту оновлення, чутливість приймача та потужність передавача — відповідно до конкретного сценарію застосування.

Важливо розуміти, що ELRS та CRSF є двома різними рівнями одного стека зв'язку і не є синонімами. ELRS відповідає виключно за бездротову радіолінію між передавачем і приймачем — він визначає, яким чином дані модулюються, кодуються та передаються в ефір. CRSF натомість є дротовим послідовним протоколом, що використовується на обох кінцях цієї радіолінії. З боку оператора застосунок надсилає CRSF-пакети на передавач через USB-порт, який виглядає для операційної системи як віртуальний серійний порт. Передавач отримує ці пакети, перекодовує їх у власний радіоформат і передає в ефір. На стороні дрона приймач отримує радіопакети, декодує їх та надсилає польотному контролеру вже у форматі CRSF через апаратний UART-інтерфейс. Таким чином, CRSF є єдиним уніфікованим форматом обміну даними на дротовому рівні, а ELRS є транспортом для цих даних у бездротовому середовищі.



Рисунок 2.1 – Структура CRSF пакету

Структура CRSF-фрейму, формальна специфікація якої опублікована у відкритому репозиторії компанії-розробника [10], зображена на рисунку 2.1 і складається з п'яти послідовних полів. Перше поле — байт синхронізації — є фіксованим ідентифікатором початку фрейму. Саме цей байт шукає кінцевий автомат парсера у нульовому стані, відкидаючи всі байти, що йому не відповідають. Друге поле — байт довжини — вказує кількість байтів що залишилися у фреймі починаючи з поля типу і включаючи контрольну суму. Це поле також перевіряється на допустимість: значення поза межами протоколу означає пошкоджений або фіктивний пакет. Третє поле — байт типу — визначає призначення та формат корисного навантаження. Кожному типу відповідає власна структура даних: наприклад, тип 0x16 позначає пакет з упакованими значеннями RC-каналів, а тип 0x08 — пакет телеметрії датчика акумулятора. Четверте поле — корисне навантаження — містить власне дані фрейму і має змінну довжину від нуля до шістдесяти байтів залежно від типу. П'яте поле — контрольна сума, що обчислюється за алгоритмом CRC-8 DVB-S2 і охоплює поле типу та все корисне навантаження. Варто зазначити, що у деяких специфікаціях CRSF контрольна сума описується як двобайтна величина за стандартом CRC-16 CCITT — це відповідає розширеному варіанту протоколу, тоді як у реалізації на основі ExpressLRS застосовується однобайтна CRC-8 DVB-S2, яка є стандартом для обладнання цього виробника.

БПЛА для керування приймає 16 каналів керування, це цифрові зміни в діапазоні [172; 1811]. Перші 4 канали, строго відповідають за вісі акселерації по тангажу, крену, ристання та значення газу. Всі інші канали відповідають за активацію певних функцій, таких як: запуск моторів, перемикання режиму польоту, активація звукового сигналу тощо.

Для пакета керування з шістнадцятьма каналами корисне навантаження становить рівно двадцять два байти, і разом із заголовком та контрольною сумою повна довжина такого фрейму складає двадцять шість байтів. Значення кожного каналу займає одинадцять біт без вирівнювання по байтових межах, тому байти

навантаження формуються через побітове пакування суміжних каналів з накладенням їх бітових полів одне на одне. Саме тому функція пакування виглядає як набір складних побітових операцій — по суті вона реалізує ручне серіалізування одинадцятибітних значень у безперервний бітовий потік.

Окремої уваги заслуговує механізм тактування, що принципово відрізняє ELRS від інших протоколів. Передавач у системі ELRS є єдиним джерелом часових міток для всього радіоканалу — він задає ритм передачі та визначає моменти, коли приймач може відповісти телеметрією. Приймач не має власного незалежного розкладу відправки: він лише очікує на черговий пакет від передавача і у відповідному вікні, що відкривається після прийому, надсилає назад дані телеметрії. Якщо передавач перестає надсилати пакети, приймач втрачає часову прив'язку і не має можливості визначити момент для відправки відповіді. Фактично телеметрія існує лише як реакція на активний потік команд з боку оператора.

Саме цей механізм обумовлює необхідність безперервного надсилання CRSF-пакетів у даній реалізації навіть тоді, коли оператор фактично не рухає стіки. Модуль серійного зв'язку надсилає пакет керування кожні двадцять мілісекунд незалежно від того, чи змінились значення каналів з попередньої ітерації. Цей постійний потік пакетів виконує роль серцебиття системи: він підтримує синхронізацію між передавачем та приймачем, утримує радіоканал у активному стані та забезпечує безперервне відкриття вікна телеметрії. Без нього приймач на борту дрона перейде у режим очікування і зв'язок з польотним контролером для отримання телеметрії буде неможливий, навіть якщо фізично радіобладнання залишається у зоні впевненого прийому.

2.5 Фізичне передавання даних між БПЛА та комунікаційним пристроєм

Система зв'язку між безпілотним літальним апаратом та оператором реалізована на основі двох незалежних радіоканалів, що функціонують на різних

частотах і виконують різні функції, схема на рис. 2.2 . Така архітектура є усталеною практикою у сфері FPV-систем, оскільки дозволяє розділити потоки даних з різними вимогами до пропускної здатності та затримки. Кожен канал оптимізований під свою задачу, що в сукупності забезпечує надійну та функціональну систему зв'язку навіть в умовах радіочастотних завад.

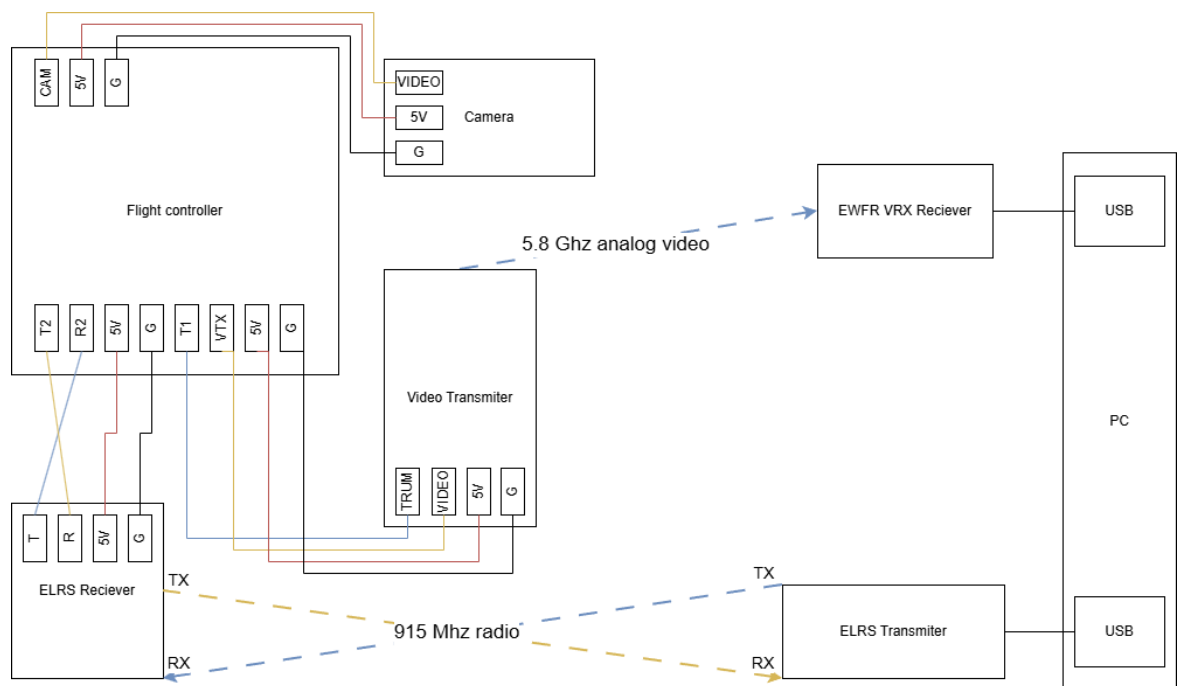


Рисунок 2.2 – Схема фізичного передавання даних між БПЛА та комунікаційним пристроєм

Контур дрона є центральним елементом бортової частини системи. Пільотний контролер виступає головним вузлом, до якого підключені всі периферійні пристрої. Він відповідає за стабілізацію апарату, обробку команд керування та формування телеметричних даних. До пільотного контролера підключені два ключові пристрої: ELRS-приймач та відеопередавач.

ELRS-приймач з'єднаний з пільотним контролером через апаратний UART-інтерфейс. UART — це асинхронний послідовний інтерфейс передачі даних,

широко застосовуваний у вбудованих системах завдяки простоті реалізації та достатній швидкості для обміну телеметрією і командами керування. Лінії TX та RX забезпечують повнодуплексний обмін: приймач передає на контролер декодовані команди керування у форматі CRSF із частотою 50 разів на секунду, а контролер у відповідь надсилає пакети телеметрії — дані про напругу акумулятора, висоту, швидкість та інші польотні параметри — назад через той самий фізичний інтерфейс у зворотному напрямку.

Відеотракт організований повністю незалежно від каналу керування. Камера підключена до відеопередавача по аналоговому композитному відеосигналу. Відеопередавач отримує живлення через виділений VTX-порт польотного контролера, що також дозволяє контролеру програмно керувати потужністю передавача та його частотним каналом через протокол Tramp або SmartAudio — відповідний сигнал передається по окремій лінії керування. Відеопередавач безперервно транслює аналоговий відеосигнал в ефір на частоті 5.8 ГГц.

Канал керування реалізований на базі протоколу ExpressLRS і працює на частоті 915 МГц. Ця частота належить до діапазону UHF і характеризується значно кращою проникністю крізь фізичні перешкоди порівняно з більш високочастотними діапазонами, а також меншим рівнем загасання сигналу на великих відстанях. Завдяки цьому канал керування забезпечує більшу дальність при відносно невеликій потужності випромінювання, що є критично важливим для збереження контролю над апаратом на межах зони покриття.

З боку оператора ELRS-передавач підключений до комп'ютера через інтерфейс USB і визначається системою як пристрій з віртуальним серійним портом. Клієнтська програма формує пакети керування у форматі CRSF із поточними значеннями каналів джойстика і передає їх на передавач через цей порт із частотою 50 Гц. Передавач транслює отримані пакети по радіоканалу, а бортовий ELRS-приймач їх отримує, декодує та передає на польотний контролер. Канал є повністю двостороннім — по тій самій радіолінії у зворотному напрямку передається телеметрія з борту, яка проходить шлях від польотного контролера

через приймач, по радіоканалу на передавач і далі повертається на комп'ютер оператора через USB.

Канал відео працює на частоті 5.8 ГГц і є однонаправленим — дані передаються виключно з борту на землю. Діапазон 5.8 ГГц має ширшу смугу пропускання порівняно з 915 МГц, що є необхідною умовою для передачі аналогового відеосигналу прийнятної якості в реальному часі. Водночас цей діапазон більш чутливий до фізичних перешкод і має меншу ефективну дальність, що є відомим компромісом при виборі частоти для відеоканалу.

На стороні оператора встановлено відеоприймач EWFR VRX, підключений до комп'ютера через USB. Пристрій приймає аналоговий радіосигнал на частоті 5.8 ГГц, демодулює його та оцифровує вхідний відеосигнал, після чого передає його на комп'ютер у вигляді стандартного відеопотоку. З точки зору операційної системи цей пристрій визначається як звичайна USB-відеокамера і доступний через стандартний інтерфейс DirectShow, що дозволяє клієнтській програмі захоплювати відео з нього так само, як і з будь-якої іншої підключеної камери.

Висновки до розділу 2

У другому розділі проаналізовано предметну область та обґрунтовано технічні рішення, покладені в основу системи. Розглянуто еволюцію протоколів радіокерування від аналогових стандартів до сучасних цифрових рішень з відкритим вихідним кодом, що дозволило обґрунтовано обрати ExpressLRS як базовий радіопротокол та CRSF як дротовий формат обміну даними. Розкрита структура CRSF-фрейму, особливості тактування ELRS-каналу та фізична організація двох незалежних радіоконтурів — керування на 915 МГц та відео на 5.8 ГГц, — що є архітектурною основою всієї подальшої програмної реалізації.

3 ЗАСОБИ РОЗРОБКИ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

3.1 Мова програмування C++

C++ — це компільована мова програмування загального призначення, що є розширенням мови C і підтримує об'єктно-орієнтовану, процедурну та узагальнену парадигми програмування. Мову було розроблено Б'ярне Страуструпом у Bell Labs на початку 1980-х років, а її фундаментальний опис, що є базовим джерелом для більшості сучасних посібників, наведений у його авторській монографії [1]. Відтоді мова пройшла через кілька суттєвих стандартизацій: C++98, C++11, C++14, C++17, C++20. У даному проєкті використовувався стандарт C++17, який надає зручні засоби роботи з багатопоточністю, атомарними операціями та сучасними контейнерами стандартної бібліотеки.

Головна причина вибору C++ для реалізації клієнтської частини системи полягає в тому, що ця мова забезпечує прямий доступ до апаратних ресурсів і системних API операційної системи без додаткових рівнів абстракції. Керування безпілотним літальним апаратом — це задача, де затримка між дією оператора та реакцією дрона має критичне значення. Мови з керованим середовищем виконання, такі як Python або Java, вносять непередбачувані паузи через роботу збирача сміття та інтерпретатора, що є неприйнятним в системах реального часу. C++ компілюється безпосередньо у машинний код, що гарантує детерміновану поведінку програми та мінімальні накладні витрати під час виконання.

Окрім продуктивності, важливим фактором стала зріла екосистема бібліотек для роботи з відео та апаратним забезпеченням. Бібліотека OpenCV, яка є галузевим стандартом у задачах комп'ютерного зору і докладно описана у профільному посібнику з її застосування на C++ [4], написана переважно на цій мові і надає найбільш повний та ефективний інтерфейс саме для неї. Використання OpenCV з

Python, наприклад, потребує додаткового шару біндінгів, що збільшує затримку та ускладнює тонке налаштування параметрів захоплення. Аналогічно, взаємодія з системними бібліотеками Windows для роботи з серійними портами (Win32 API), пристроями введення (WinMM) та відеозахопленням (DirectShow), повна довідкова документація яких опублікована на офіційному ресурсі Microsoft [11], є нативною саме для C і C++.

Розробка клієнтської частини велась у середовищі Visual Studio з компілятором MSVC, що забезпечило зручну інтеграцію з Windows SDK та спрощений процес підключення системних бібліотек через директиви `#pragma comment`. Код організований за модульним принципом: кожна підсистема — відеозахоплення, обробка вводу, серійний зв'язок, мережева передача — винесена в окремий модуль із власним заголовним файлом. Це дозволяє незалежно розробляти та тестувати кожен компонент, а також спрощує подальше масштабування проєкту.

Багатопоточність, яка є ключовою вимогою для паралельної роботи відеопотоку, опитування джойстика та мережевої передачі даних, реалізована засобами стандартної бібліотеки C++17 — зокрема `std::thread`, `std::mutex` та `std::atomic`. Це дозволило уникнути залежності від сторонніх фреймворків і зберегти повний контроль над поведінкою потоків. Атомарні змінні типу `std::atomic<uint16_t>` використовуються для обміну значеннями каналів керування між потоками без блокувань, що суттєво знижує ймовірність затримок у критичних ділянках коду.

C++ залишається актуальним вибором для подібних систем і надалі з кількох причин. По-перше, жодна інша мова не забезпечує одночасно такий рівень продуктивності, доступ до низькорівневих API і розвинуту екосистему для роботи з мультимедіа та вбудованими системами. По-друге, більшість програмного забезпечення для безпілотних систем, автопілотів і робототехніки — зокрема ArduPilot, PX4, ROS — написана на C++, що робить знання цієї мови стандартом у галузі. По-третє, стандарт мови активно розвивається: C++20 та C++23 привносять

нові можливості, зокрема концепти, корутини та покращену підтримку модульної системи, що робить мову ще більш зручною для великих проєктів. Таким чином, C++ є не застарілим інструментом, а навпаки — мовою, яка зберігає домінуючі позиції саме в тих галузях, де важливі надійність, швидкодія та контроль над ресурсами системи.

3.2 Мова програмування GO

Go — це компільована мова програмування з відкритим вихідним кодом, розроблена компанією Google у 2009 році. Її повний опис, що є основним джерелом для розробників, наведений у авторській книзі співавтора мови [5]. Мова створювалась із фокусом на простоті, надійності та ефективній підтримці конкурентного виконання, що робить її особливо зручною для розробки мережесервісів і серверних застосунків. На відміну від C++, Go не претендує на універсальність і низькорівневий контроль, натомість пропонує лаконічний синтаксис, вбудовану підтримку конкурентності та надзвичайно швидку компіляцію.

Для даного проєкту Go було обрано для реалізації серверної частини системи — релейного ретранслятора, який забезпечує маршрутизацію відеопотоку та команд керування між операторами та дронами. Ця задача за своєю природою є суто мережевою: сервер повинен одночасно обслуговувати велику кількість TCP і UDP з'єднань, не блокуючи обробку одних клієнтів через затримки інших. Саме тут Go демонструє свої сильні сторони — вбудований планувальник горутин дозволяє запускати тисячі легковагових конкурентних задач з мінімальними витратами пам'яті, тоді як аналогічне рішення на основі системних потоків у C++ вимагало б значно складнішого коду і більшого споживання ресурсів.

Стандартна бібліотека Go містить повноцінний пакет «net», який надає зручні інструменти для роботи з TCP і UDP без необхідності підключення сторонніх залежностей. Це дозволило реалізувати весь мережевий стек сервера виключно

засобами мови, що спрощує розгортання: скомпільований бінарний файл Go не потребує встановлення додаткових бібліотек на цільовій машині. Для синхронізації доступу до спільних даних між горутинами використовувався пакет «sync» із примітивами «sync.RWMutex», що забезпечує безпечний одночасний доступ до мапи підключених клієнтів.

Go також виграє у питанні розгортання та підтримки серверного коду. Мова має строгу статичну типізацію, що дозволяє виявляти більшість помилок ще на етапі компіляції, а вбудований інструментарій — форматувач «gofmt», менеджер залежностей «go mod» — стандартизує структуру проєкту і знижує організаційні витрати. У поєднанні з тим, що Go є однією з найпопулярніших мов для розробки хмарних і мережових сервісів, вибір цієї мови для серверної частини є обґрунтованим як з технічної, так і з практичної точки зору.

3.3 Операційна система Debian

Для розгортання серверної частини системи використовувалась операційна система Debian GNU/Linux. Debian є одним із найстаріших і найстабільніших дистрибутивів Linux, що активно підтримується спільнотою розробників з 1993 року. Ключовою перевагою Debian у контексті серверного застосування є його орієнтованість на стабільність: пакети, що потрапляють до стабільної гілки дистрибутива, проходять тривале тестування, що мінімізує ризик непередбачуваної поведінки системи в умовах тривалої роботи без перезавантаження.

Релейний сервер, написаний на Go, розгортався на Debian у вигляді скомпільованого бінарного файлу, що не потребував встановлення додаткових залежностей. Адміністрування системи здійснювалось через термінал з використанням стандартних інструментів — `systemd` для керування процесом сервера та мережових утиліт для діагностики з'єднань. Debian також є базою для Ubuntu Server, що робить набуті навички адміністрування універсальними і застосовними на більшості поширених серверних платформах.

3.4 Система збірки Cmake

CMake — це кросплатформенна система автоматизації збірки програмного забезпечення з відкритим вихідним кодом, розроблена компанією Kitware. На відміну від традиційних Makefile, CMake не збирає проєкт безпосередньо, а генерує файли збірки для цільової системи — будь то Makefile для Unix-середовищ, проєкти Visual Studio для Windows або файли для інших підтримуваних систем збірки. Такий підхід забезпечує незалежність опису проєкту від конкретного середовища розробки і спрощує перенесення коду між платформами.

У даному проєкті CMake використовувався як інструмент інтеграції бібліотеки OpenCV у клієнтський застосунок на C++. Підключення OpenCV вручну є нетривіальною задачею: бібліотека складається з великої кількості модулів, кожен з яких компілюється в окремий бінарний файл, і при ручному налаштуванні розробник змушений вказувати шляхи до заголовних файлів та лінкувати кожен залежність окремо. CMake вирішує цю проблему через механізм пошуку пакетів — директива `find_package(OpenCV REQUIRED)` автоматично знаходить встановлену бібліотеку, визначає всі необхідні шляхи та змінні, після чого достатньо лише вказати `target_link_libraries` для підключення потрібних модулів до цільового виконуваного файлу.

Конфігураційний файл `CMakeLists.txt` також дозволяє централізовано керувати параметрами компіляції: вказувати стандарт мови, задавати прапорці оптимізації та описувати структуру проєкту в декларативному стилі. Це робить процес збірки відтворюваним — будь-який розробник, що клонує репозиторій і має встановлені необхідні залежності, може зібрати проєкт однією командою без додаткового налаштування середовища.

CMake сьогодні є фактичним стандартом збірки у C++ екосистемі. Його підтримують усі провідні середовища розробки, включаючи Visual Studio, CLion та VS Code, а більшість великих C++ бібліотек, у тому числі OpenCV, поставляються з готовими CMake-конфігураціями. Це робить вибір CMake не просто зручним

рішенням для конкретного проєкту, а й інвестицією у навички, що застосовуються у переважній більшості реальних C++ проєктів.

3.5 Програмне забезпечення ExpressLRS Configurator

ExpressLRS Configurator — це графічна утиліта з відкритим вихідним кодом, призначена для компіляції та завантаження прошивки ExpressLRS на радіообладнання для безпілотних систем. Інтерфейс програми зображено на рис.3.1 ExpressLRS є відкритим протоколом радіозв'язку, що активно розвивається спільнотою і позиціонується як альтернатива комерційним рішенням завдяки низькій затримці, відносно великій дальності та підтримці широкого спектру апаратних платформ на базі мікроконтролерів ESP32 та ESP8285.

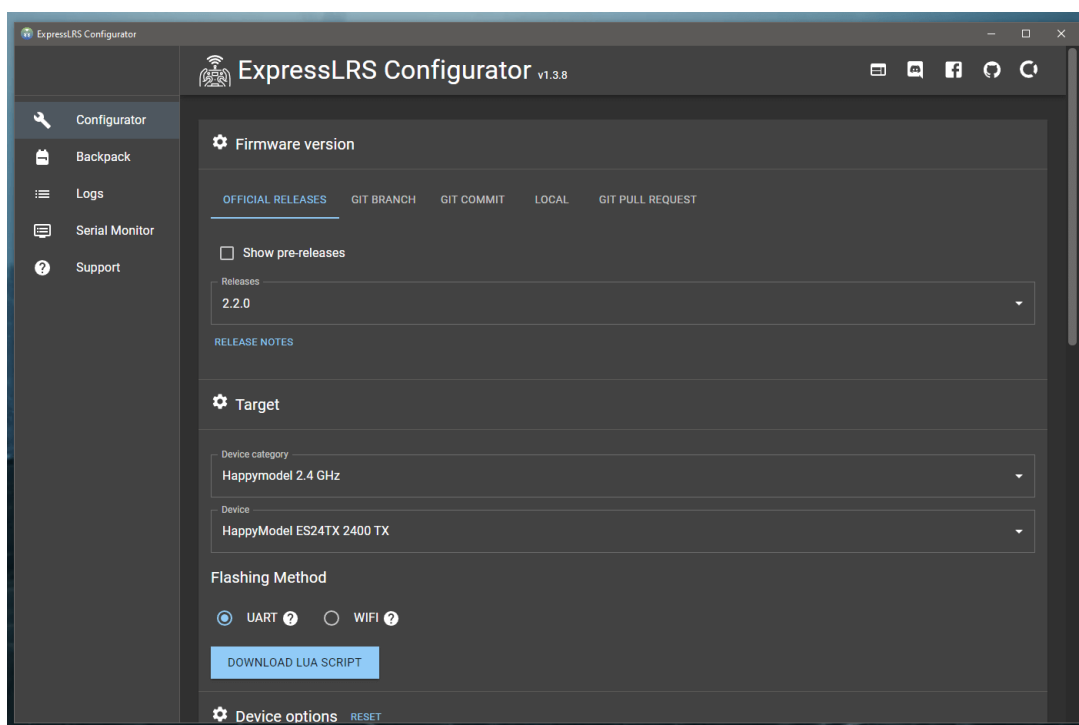


Рисунок 3.1 – Інтерфейс ExpressLRS Configurator

У даному проєкті ExpressLRS Configurator використовувався для вирішення конкретної інженерної задачі — здешевлення комунікаційного вузла системи керування. Стандартна схема передбачає використання окремого радіомодуля-

передавача на стороні оператора, проте такі модулі є значно дорожчими за звичайні приймачі. Оскільки прошивка ExpressLRS підтримує режим роботи у якості передавача не лише на спеціалізованих ТХ-модулях, а й на деяких моделях приймачів, було прийнято рішення перепрошити приймач у режим передавача. Це дозволило суттєво знизити вартість апаратної частини без втрати функціональності, оскільки з точки зору протоколу різниця між спеціалізованим передавачем і перепрошитим приймачем відсутня.

Процес прошивки здійснювався через інтерфейс ExpressLRS Configurator: для кожного пристрою вибирався відповідний цільовий профіль, вказувалась bind-фраза та метод завантаження прошивки. Bind-фраза в ExpressLRS є текстовим ідентифікатором, що вбудовується безпосередньо у прошивку і замінює традиційний процес ручного зв'язування пристроїв. Перепрошитий передавач та приймач — були прошиті з однаковою bind-фразою, що забезпечило їх автоматичне з'єднання без необхідності виконання процедури бінду вручну. Такий підхід також виявився практичнішим з огляду на те, що програмне введення передавача в режим бінду через послідовний інтерфейс потребує додаткового дослідження протоколу і виходило за межі пріоритетних задач проєкту.

Висновки до розділу 3

У третьому розділі обґрунтовано вибір засобів розробки відповідно до вимог двох принципово різних частин системи. Для клієнтського застосунку обрано C++ через прямий доступ до системних API Windows та зрілу екосистему бібліотек для роботи з відео і апаратним забезпеченням; для серверної частини — мову Go з вбудованою підтримкою конкурентності та простотою розгортання на Linux-середовищі Debian. Допоміжний інструментарій — CMake для інтеграції OpenCV у C++ збірку та ExpressLRS Configurator для прошивки радіообладнання — дозволив суттєво знизити складність налаштування проєкту та вартість апаратної частини.

4 ОПИС ПРОГРАМНОЇ РЕАЛІЗАЦІЇ

4.1 Реалізація клієнтського застосунку

4.1.1 Файл main.cpp

Файл main.cpp є точкою входу клієнтського застосунку і реалізує три принципово різні задачі в рамках одного модуля: визначення глобального стану програми, побудову інтерфейсу користувача засобами OpenCV та обробку командного вводу. Архітектурно застосунок організований за чотирьохшаровою моделлю, що відображена на рисунку 4.1, — від системних API на четвертому рівні до елементів інтерфейсу на першому. Саме main.cpp реалізує перший і другий рівні цієї моделі: безпосередні елементи відображення та контролер, що ними керує.

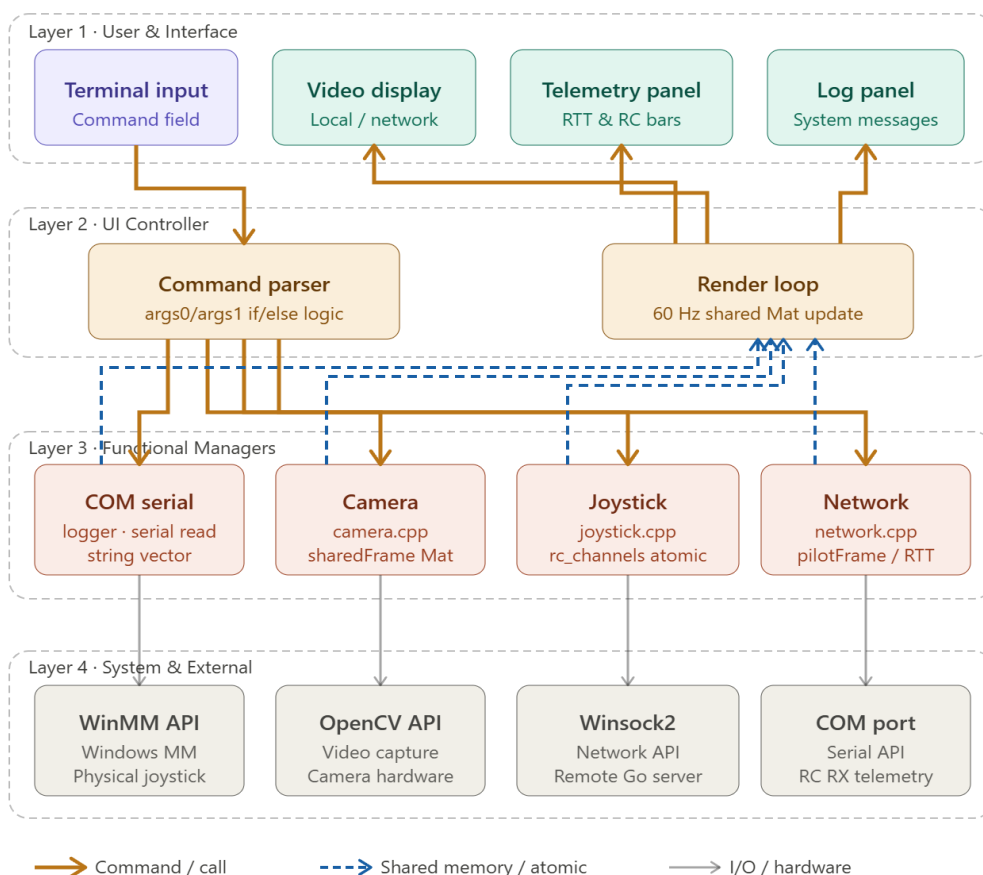


Рисунок 4.1 – Загальна архітектура ПЗ клієнта

Глобальний стан та ініціалізація

На початку файлу, поза будь-якою функцією, оголошується глобальний масив значень RC-каналів із шістнадцяти елементів. Це єдина глобальна змінна у всьому проєкті, і її розміщення саме у головному модулі є свідомим рішенням — вона оголошена тут як власник даних, а всі інші модулі отримують до неї доступ через відповідну директиву. Тип кожного елемента масиву є атомарним цілим числом без знаку, що обрано через те, що до масиву одночасно звертаються щонайменше три потоки виконання: потік джойстика записує нові значення з частотою 50 Гц, потік серійного зв'язку читає їх для формування пакетів керування, а головний потік рендерингу зчитує їх для відображення індикаторів. Атомарні операції гарантують коректність такого доступу без явних блокувань, що критично на шляху передачі команд керування. Центральна роль цього масиву у взаємодії між компонентами добре видна на діаграмі потоків даних на рисунку 4.2 — він є точкою перетину між процесами джойстика, мережевого менеджера та серійного зв'язку.

На початку головної функції масив ініціалізується у безпечний стан: усім шістнадцяти каналам присвоюється нейтральне значення 992, після чого окремими операціями атомарного запису перевизначаються три критичні канали — канал газу та канал армування встановлюються у мінімум 172, канал польотного режиму — також у 172, що відповідає найбезпечнішому режиму стабілізації. Такий порядок ініціалізації гарантує, що навіть якщо серійний порт буде відкритий до запуску джойстика, польотний контролер отримає пакет із газом у нулі та моторами у стані роззброєння.

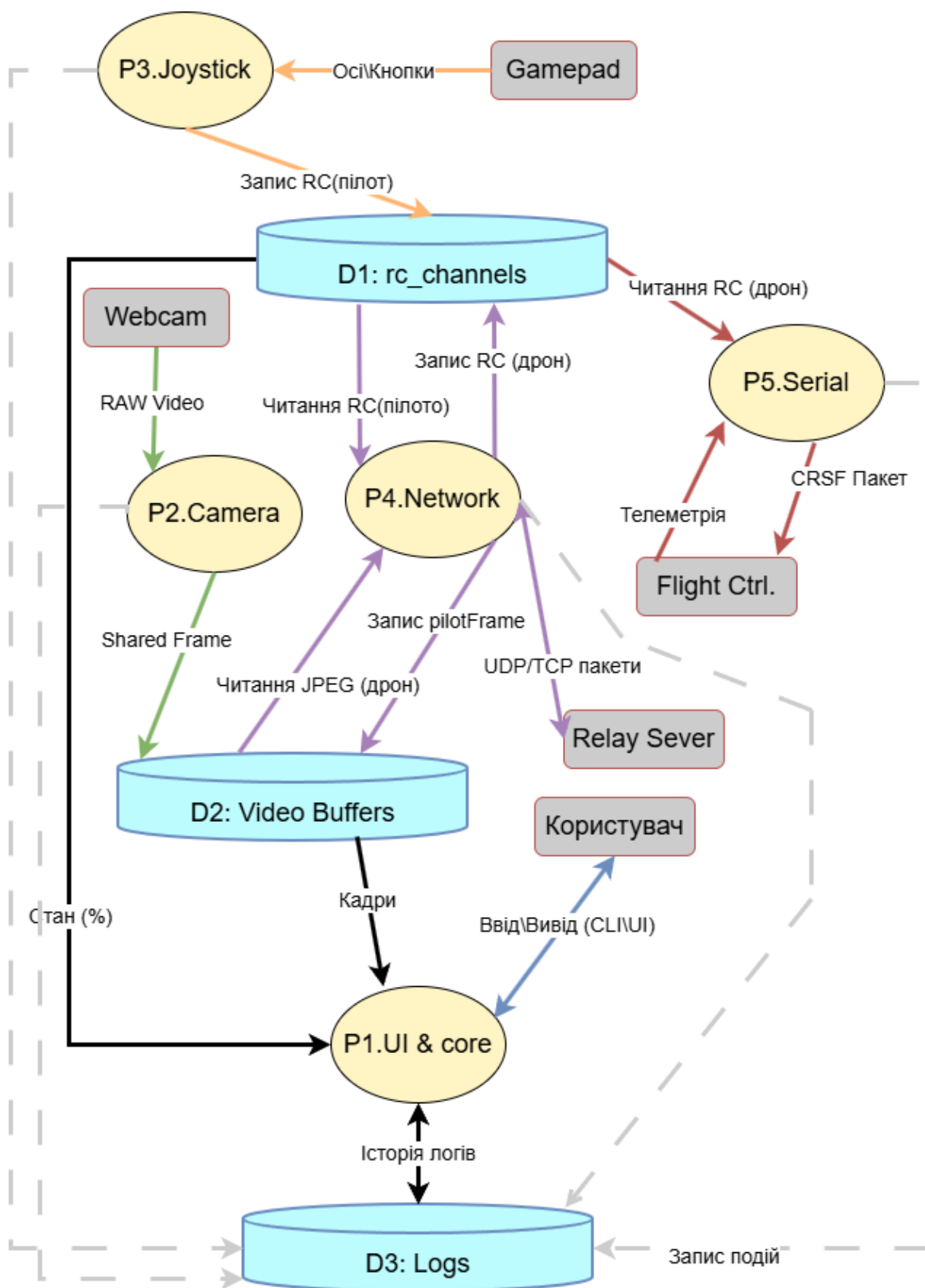


Рисунок 4.2 – DFD діаграма ПЗ клієнта

Організація головного циклу рендерингу

Головний цикл побудований навколо виклику функції очікування клавіші з інтервалом в одну мілісекунду, яка одночасно виконує дві функції: повертає код натиснутої клавіші та забезпечує мінімальну паузу між ітераціями. Завдяки тому, що кожна ітерація також виконує повноцінний рендеринг кадру, фактична частота оновлення інтерфейсу визначається сумарним часом виконання всього циклу і становить близько 60 Гц. Перевірка закриття вікна виконується на самому початку кожної ітерації — якщо вікно більше не видиме, прапор активності застосунку встановлюється у хибне значення і цикл завершується. Весь код циклу обгорнутий у блок обробки виключень для перехоплення помилок бібліотеки відеообробки.

На кожній ітерації спочатку полотно рендерингу розміром 1400×720 пікселів повністю очищується заповненням чорним кольором. Потім у локальні змінні копіюються поточний відеокадр та поточний стан журналу подій. Обидві операції виконуються навмисно на початку ітерації — це знімає один узгоджений знімок стану з асинхронних підсистем і гарантує, що впродовж рендерингу одного кадру відображаються дані з одного моменту часу, навіть якщо фонові потоки продовжують записувати нові значення. Джерело відеокадру визначається поточною роллю застосунку: якщо активний режим пілота, кадр отримується з мережевого відеобуфера, в усіх інших випадках — безпосередньо з локальної камери. Отриманий кадр масштабується до розміру 800×600 пікселів і вставляється у центральну зону полотна зі зміщенням рівно на ширину лівої панелі.

Реалізація відображення каналів

Шістнадцять RC-каналів відображаються у нижній частині центральної зони у вигляді сітки з чотирьох стовпців та чотирьох рядків. Координати кожного індикатора обчислюються через операції цілочисельного ділення та остачі від ділення порядкового номера каналу, що автоматично розкладає лінійний масив у матрицю без явного задання позицій. Для кожного каналу атомарно зчитується поточне значення, після чого обчислюється відсоток заповнення через

нормалізацію відносно мінімального та максимального значень протоколу з обрізанням результату до допустимого діапазону. На полотні спочатку малюється прямокутник-фон сірого кольору фіксованою шириною, поверх якого накладається білий заповнений прямокутник, ширина якого пропорційна обчисленому відсотку. Поряд виводиться текстова назва каналу та числове значення у відсотках.

Реалізація панелі журналу подій

Журнал подій відображається у правій панелі. Для рендерингу використовується посилання на відповідну підобласть полотна — операція, яка не копіює пікселі, а лише визначає прямокутну область, тому всі подальші операції малювання безпосередньо змінюють відповідну частину загального полотна. Рядки виводяться у зворотному порядку: ітерація йде знизу вгору від нижнього краю панелі з фіксованим кроком 15 пікселів, що забезпечує природну поведінку термінального виводу — нові повідомлення з'являються знизу. Кожен рядок перед виводом передається у допоміжну функцію перенесення тексту, яка розбиває його на підрядки фіксованою довжиною, обчисленою через апроксимовану ширину одного символу в пікселях. Для перенесених підрядків також застосовується зворотна ітерація, що в сукупності забезпечує правильний порядок читання при виводі знизу вгору.

Реалізація поля вводу команд

Поле вводу реалізоване як рядок тексту із окремою числовою змінною, що зберігає поточну позицію курсора у символах. Для підтримки введення команд, довших за видиму ширину поля, реалізований горизонтальний скролінг через змінну зміщення: якщо позиція курсора виходить за межі видимої зони, зміщення коригується так, щоб курсор завжди залишався у видимому вікні. Миготіння курсора реалізоване без окремого таймерного потоку — символ курсора чергується між вертикальною рисою та пробілом залежно від парності поточної мітки часу, поділеної на константний інтервал. Перед відображенням символ курсора

вставляється у рядок команди на відповідну позицію, після чого береться лише видима підстрічка відповідно до поточного зміщення скролінгу.

Для навігації по раніше введених командах підтримуються два окремі буфери: буфер відображення, де кожен запис містить префікс > і виводиться на лівій панелі, та буфер навігації з чистими рядками без префіксів, по якому здійснюється переміщення стрілками вгору та вниз. Показчик поточної позиції у буфері навігації скидається на позицію за останнім елементом після кожного введення команди, забезпечуючи стандартну поведінку термінальної навігації по історії.

Реалізація парсера команд

Парсинг відбувається при натисканні клавіші Enter: введений рядок передається у рядковий потік, з якого у циклі зчитуються токени у вектор аргументів. Перший елемент вектора є іменем команди, решта — її аргументами. Логіка розгалужується через ланцюжок умовних виразів по значенню імені команди. Перед виконанням команди Enter перехоплюється окремою гілкою: якщо активний режим калібрування джойстика, натискання Enter передається не парсеру, а менеджеру калібрування як підтвердження поточного кроку, а спеціальні слова skip та cancel обробляються як керуючі команди цього режиму.

Команди підключення перед викликом відповідних менеджерів виконують перевірки готовності підсистем: підключення в режимі дрона вимагає одночасно активної камери та наявності телеметрії від польотного контролера, підключення в режимі пілота — наявності відкаліброваного джойстика. У разі невиконання будь-якої з умов у журнал записується пояснювальне повідомлення і підключення не відбувається. Адреса сервера передається у форматі IP:PORT і розбивається на складові через пошук символу двокрапки у рядку аргументу.

Завершення роботи

Після виходу з головного циклу підсистеми зупиняються у строго визначеній послідовності: спочатку менеджер джойстика, потім мережеве з'єднання, серійний порт, камера і наприкінці — всі вікна бібліотеки відеообробки. Такий порядок обумовлений залежностями між модулями: мережевий менеджер читає відеобуфер камери у своєму фоновому потоці і повинен бути відключений до її зупинки, щоб уникнути звернення до вже звільнених ресурсів. Весь блок завершення виконується поза блоком обробки виключень, що гарантує звільнення ресурсів навіть у разі аварійного завершення головного циклу.

4.1.2 Файл `network.cpp`

Файл `network.cpp` реалізує весь мережевий стек клієнтського застосунку в рамках одного фонового потоку виконання. Модуль забезпечує двосторонній обмін даними між клієнтом та сервером-ретранслятором: у режимі дрона — передачу відеопотоку та прийом команд керування, у режимі пілота — прийом відеопотоку та передачу команд. Повна послідовність виконання модуля, включаючи всі умовні розгалуження та таймери, відображена на рисунку 4.3 у вигляді алгоритмічної схеми, якій і відповідає подальший опис реалізації.

Стан модуля організований через набір статичних полів класу. Прапор активності фонового потоку та прапор встановленого з'єднання реалізовані як атомарні булеві змінні, що дозволяє головному потоку безпечно перевіряти стан без явних блокувань. Поточна роль клієнта зберігається як значення перерахування з трьох станів: дрон, пілот або відключено. Поточне значення затримки каналу зв'язку та поточний рівень якості стиснення відеокادру також реалізовані як атомарні цілочисельні змінні, оскільки вони одночасно читаються і записуються з різних частин циклу. Крім того, в анонімному просторі імен оголошені буфер останнього зібраного відеокадру у вигляді вектора байтів та прапор його новизни — обидві змінні захищені окремим м'ютексом, оскільки до них звертається як

фоновий мережевий потік для запису, так і головний потік для зчитування та декодування.

Публічний інтерфейс модуля реалізований через два методи підключення та один метод відключення. Метод підключення в режимі дрона та метод підключення в режимі пілота влаштовані однаково: обидва спочатку викликають відключення для гарантованого завершення попереднього потоку якщо він існував, після чого встановлюють відповідну роль, зберігають ідентифікатор цільового дрона у випадку ролі пілота та запускають фоновий потік. Метод відключення встановлює прапор активності у хибне значення і очікує завершення потоку через блокуючий виклик. Такий підхід гарантує, що в будь-який момент часу існує не більше одного активного мережевого потоку.

Метод отримання відеокадру для пілота реалізований з відкладеним декодуванням. При кожному виклику спочатку під захистом м'ютекса перевіряється прапор новизни буфера — якщо новий кадр надійшов, вектор байтів копіюється у локальну змінну і прапор скидається. Після звільнення м'ютекса декодування JPEG виконується вже без блокування, що мінімізує час утримання спільного ресурсу. Результат декодування зберігається в окремому полі останнього успішного кадру, захищеному власним м'ютексом, і повертається викликачу через клонування. Якщо новий кадр не надходив, метод повертає попередній збережений кадр — завдяки цьому інтерфейс не повертає порожній результат у разі тимчасової відсутності нових даних з мережі.

Фоновий потік, що реалізує весь мережевий алгоритм відповідно до схеми на рисунку 4.3, починається з ініціалізації бібліотеки мережевих сокетів Winsock через стандартний виклик ініціалізації з версією протоколу 2.2. У разі невдачі потік завершується з відповідним записом у журнал. Після успішної ініціалізації створюється TCP-сокет, і виконується синхронне підключення до сервера за вказаними адресою та портом. TCP-з'єднання використовується виключно для авторизації: одразу після встановлення з'єднання сервер надсилає рядок з унікальним ідентифікатором клієнта у форматі з префіксом та двокрапкою як

роздільником. Ідентифікатор витягується через пошук позиції роздільника та операцію підрядка, після чого очищується від символів переведення рядка. Якщо роль клієнта є роллю пілота, одразу після отримання ідентифікатора через те саме ТСП-з'єднання надсилається команда підписки на відеопотік з ідентифікатором цільового дрона — це єдине повідомлення, яке пілот надсилає через ТСП.

Після завершення ТСП-авторизації створюється UDP-сокет і переводиться у неблокуючий режим через відповідний системний виклик керування введенням-виведенням. Додатково через розширений системний виклик вимикається генерація помилок при отриманні повідомлення про недосяжність порту — без цього налаштування на Windows будь-який відхилений UDP-пакет призводить до помилки при наступному виклику читання. Після налаштування сокета на сервер надсилається початковий ідентифікаційний пакет типу PING, що складається зі шістнадцятисимвольного ідентифікатора клієнта та рядка-маркера. Цей пакет є обов'язковим: сервер асоціює UDP-адресу клієнта з його ідентифікатором лише після отримання першого UDP-пакета з відомим ідентифікатором. Після відправки початкового пакета прапор встановленого з'єднання переводиться в істинне значення.

Перед входом у головний цикл ініціалізуються три часові мітки: момент наступної відправки керуючого пакета, момент наступної відправки відеокадру та мітка останнього keealive-пакета. Також ініціалізується лічильник ідентифікаторів відеокадрів, лічильник найвищого отриманого ідентифікатора та словник буферів збірки фреймів, де ключем є ідентифікатор кадру, а значенням — структура, що містить очікувану кількість чанків та словник вже отриманих чанків з їх порядковими номерами.

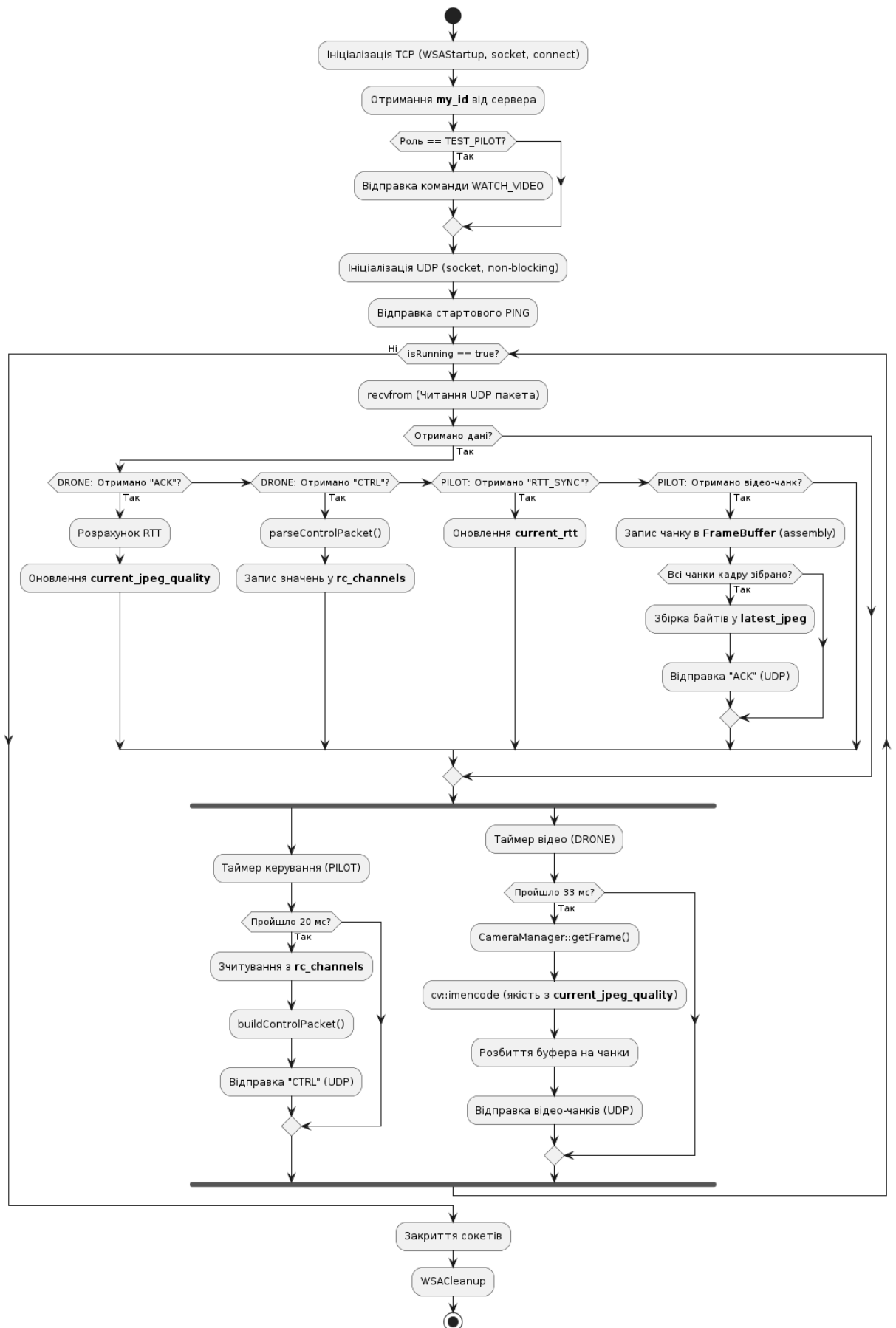


Рисунок 4.3 – Алгоритм роботи класу Network

Головний цикл потоку виконується допоки активний прапор виконання. На початку кожної ітерації у внутрішньому циклі виконується вичитування всіх доступних UDP-пакетів через неблокуючий виклик читання до тих пір, поки буфер прийому не спустіє — функція читання повертає код помилки WSAEWOULDBLOCK, що означає відсутність даних і є сигналом для виходу з внутрішнього циклу. Кожен отриманий пакет аналізується на предмет типу через пошук характерних рядків-маркерів. Логіка розгалужується на чотири незалежні гілки залежно від комбінації типу пакета та поточної ролі, що відображено у відповідних умовних блоках схеми на рисунку 4.3.

Перша гілка обробляє підтвердження доставки відеокадру, що надходять дрону від пілота. З тіла пакета витягується ідентифікатор підтвердженого кадру, після чого під захистом м'ютекса зі словника часових міток відправки вилучається запис про час відправки цього кадру. Затримка обчислюється як різниця між поточною міткою часу та збереженою міткою відправки у мілісекундах і зберігається атомарно. Після обчислення затримки дрон одразу надсилає пілоту синхронізуючий пакет із виміряним значенням, щоб пілот міг відображати актуальну затримку у своєму інтерфейсі. Далі виконується адаптація рівня якості стиснення: якщо затримка менша за 50 мілісекунд, якість поступово збільшується до максимального значення 50 одиниць; якщо затримка в межах 130 мілісекунд — якість утримується в середньому діапазоні; якщо затримка перевищує 130 мілісекунд, якість різко знижується на 5 одиниць до мінімуму 15, щоб зменшити обсяг даних і відновити стабільність каналу.

Друга гілка обробляє пакети керування, що надходять дрону від пілота. Метод розбору керуючого пакета шукає маркер типу в рядку пакета, після чого витягує підрядок значень, розбиває його за роздільником-комою через рядковий потік і послідовно записує отримані цілочисельні значення в атомарний масив каналів керування через блок обробки виключень для захисту від некоректних даних.

Третя гілка обробляє синхронізуючі пакети затримки, що надходять пілоту. З тіла пакета витягується числове значення затримки і зберігається атомарно — таким чином пілот відображає в інтерфейсі не власновимірну затримку, а значення, виміряне на стороні дрона, що є більш точним оскільки враховує реальний маршрут відеопакетів.

Четверта гілка обробляє відео-чанки, що надходять пілоту. Кожен чанк містить заголовок з шести байтів: чотири байти ідентифікатора кадру у мережевому порядку байтів, один байт загальної кількості чанків кадру та один байт порядкового номера поточного чанка. Після розбору заголовка чанк записується у відповідний запис словника збірки. Реалізовано захист від переповнення лічильника ідентифікаторів: якщо новий ідентифікатор виявляється значно меншим за найвищий відомий, словник збірки очищується і лічильник скидається. Якщо після запису кількість отриманих чанків для кадру дорівнює очікуваній загальній кількості, виконується збірка: обчислюється сумарний розмір, резервується вектор байтів і чанки послідовно конкатенуються у порядку зростання їх порядкових номерів. Зібраний вектор байтів переміщується у спільний буфер останнього кадру через операцію переміщення без копіювання і встановлюється прапор новизни. Після успішної збірки кадру пілот надсилає дрону підтвердження доставки з ідентифікатором зібраного кадру.

Після вичерпання вхідної черги UDP-пакетів у режимі пілота виконується очищення словника збірки від застарілих незавершених записів: якщо різниця між найвищим отриманим ідентифікатором та ідентифікатором запису перевищує порогове значення у тридцять кадрів, запис вважається безнадійно застарілим і видаляється. Це запобігає необмеженому зростанню словника у разі систематичної втрати пакетів.

Після обробки вхідних даних виконується фаза відправки, що також відповідає нижній частині схеми на рисунку 4.3. У режимі пілота кожні 20 мілісекунд зчитується поточний стан атомарного масиву каналів керування, формується рядок пакета з шістнадцяти значень розділених комами з префіксом

типу та ідентифікатором відправника і надсилається на сервер. Окремо раз на секунду надсилається keeralive-пакет типу PING для підтримки асоціації UDP-адреси на сервері. У режимі дрона кожні 33 мілісекунди, що відповідає частоті близько 30 кадрів на секунду, запитується поточний відеокادر з менеджера камери. Якщо кадр доступний, він стискається у JPEG з поточним адаптивним рівнем якості. Загальні принципи стиснення відеоданих, на яких базуються і сучасний стандарт H.264, і простіший формат JPEG, докладно розглянуті у [9]. Перед відправкою час поточного моменту записується у словник часових міток під ключем поточного ідентифікатора кадру для подальшого вимірювання затримки. Стиснений буфер розбивається на чанки фіксованого розміру 1300 байтів — значення обране як безпечне для більшості мереж з урахуванням MTU Ethernet та заголовків протоколів. Кожен чанк доповнюється заголовком з ідентифікатора відправника, ідентифікатора кадру у мережевому порядку байтів, загальної кількості чанків та порядкового номера і надсилається окремим UDP-датаграмою. Після відправки всіх чанків час наступної відправки зсувається на 33 мілісекунди вперед від поточного моменту, а не від фактичного часу відправки — це забезпечує стабільну частоту незалежно від часу виконання операцій стиснення та відправки. Кожна ітерація головного циклу завершується паузою в одну мілісекунду для уникнення надмірного навантаження на процесор при відсутності вхідних даних.

Після виходу з головного циклу обидва сокети закриваються у порядку, зворотному до їх створення — спочатку UDP, потім TCP — і викликається функція деініціалізації бібліотеки мережесокетів.

4.1.3 Файл camera.cpp

Файл реалізує управління USB-камерою та організацію безпечного доступу до відеокадрів з кількох потоків. Загальна архітектура модуля побудована за трирівневою схемою, де апаратний рівень камери, фоновий потік захоплення та публічний інтерфейс доступу розділені спільним буфером із захистом м'ютексом.

Перелік доступних відеопристроїв реалізований через низькорівневий інтерфейс DirectShow. Перед зверненням до системного API викликається ініціалізація COM-середовища в багатопотоковому режимі, результат якого зберігається у булевій змінній — деініціалізація викликається лише якщо ініціалізація була успішною, що запобігає некоректному завершенню у разі вкладених викликів. Далі через фабрику системних пристроїв створюється перерахувач відеопристроїв, який ітерується у циклі. Для кожного знайденого пристрою відкривається його сховище властивостей, з якого зчитується рядок дружньої назви у форматі широких символів. Перетворення у UTF-8 виконується двоетапно: спочатку викликом без буфера визначається необхідна довжина результуючого рядка, потім рядок заданої довжини заповнюється повторним викликом. Після ітерації всі COM-об'єкти звільняються у зворотному порядку отримання. Якщо жоден пристрій не знайдено, у результуючий вектор додається відповідне системне повідомлення.

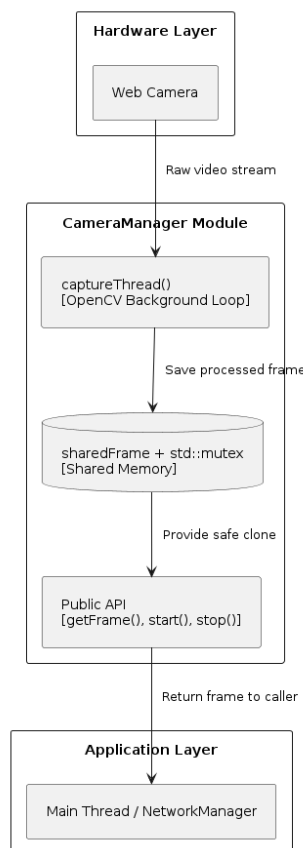


Рисунок 4.4 – Етапи роботи файлу camera

Запуск захоплення реалізований через перевірку прапора активності на початку методу — якщо потік вже запущений, повторний запуск ігнорується. За умови, що потік не активний, прапор встановлюється в істинне значення і запускається фоновий потік захоплення. Зупинка виконується встановленням прапора у хибне значення з подальшим блокуючим очікуванням завершення потоку.

Фоновий потік захоплення, що є центральним елементом модуля згідно зі схемою на рисунку 4.4, відкриває камеру через інтерфейс OpenCV з явним зазначенням DirectShow як бекенду. Одразу після відкриття встановлюються чотири параметри: роздільна здатність 640×480 , частота кадрів 30, та розмір внутрішнього буфера захоплення рівний одиниці. Останній параметр є принциповим — він вимикає внутрішню чергу кадрів OpenCV і гарантує, що кожен виклик отримання кадру повертає найсвіжіший доступний кадр, а не кадр що очікував у черзі. У разі невдалого відкриття у журнал записується повідомлення з ідентифікатором пристрою, прапор активності скидається і потік завершується. В основному циклі захоплення кадри отримуються двохетапно: спочатку викликається метод захоплення без декодування, після чого окремим викликом кадр декодується у матрицю пікселів. Такий розподіл є оптимізацією — захоплення і декодування виконуються послідовно, але без зайвого копіювання даних. Якщо декодований кадр не є порожнім, він масштабується до цільового розміру 640×480 та згладжується фільтром Гауса з ядром 3×3 . Оброблений кадр клонується у спільний буфер під захистом м'ютекса. Після виходу з циклу камера звільняється, у журнал записується повідомлення про зупинку і спільний буфер очищується присвоєнням порожньої матриці — також під захистом м'ютекса, щоб споживачі не отримали застарілий кадр після зупинки.

Метод отримання кадру споживачем реалізований як єдина захищена операція: під час утримання м'ютекса перевіряється непорожність спільного буфера і якщо кадр доступний — він клонується у вихідний параметр. Клонування

є обов'язковим, оскільки повернення посилання на внутрішній буфер без захисту м'ютексом призвело б до перегону даних із потоком захоплення.

4.1.4 Файл `serial_rx.cpp`

Файл `serial_rx.cpp` реалізує двосторонній обмін даними між клієнтським застосунком та польотним контролером через послідовний COM-порт. Модуль виконує дві принципово різні задачі одночасно в рамках одного фоновому потоку: читає телеметрію з польотного контролера та надсилає назад команди керування у форматі протоколу CRSF. Загальний алгоритм роботи модуля відображено на рисунку 4.5.

Стан модуля описується трьома статичними полями: атомарний прапор активності потоку, атомарний прапор наявності телеметрії та дескриптор фоновому потоку. Прапор телеметрії є ключовим для решти системи — саме через нього головний модуль визначає, чи підключений дрон до COM-порту перед тим як дозволити мережеве з'єднання.

Обчислення контрольної суми реалізоване за стандартом DVB-S2, що використовується у протоколі CRSF згідно з його офіційною специфікацією [10]. Алгоритм є побітовим: для кожного вхідного байта спочатку виконується операція виключного АБО з поточним накопиченим значенням, після чого запускається цикл з восьми ітерацій — по одній на кожен біт. На кожній ітерації перевіряється старший біт: якщо він встановлений, значення зсувається вліво і виконується виключне АБО з фіксованим поліномом протоколу, якщо ні — лише зсув вліво. Функція приймає поточне накопичене значення та один черговий байт, що дозволяє викликати її послідовно для довільної кількості байтів, передаючи результат попереднього виклику як аргумент наступного.

Пакування значень каналів керування виконується шляхом перетворення шістнадцяти одинадцятибітних значень у неперервний бітовий потік довжиною двадцять два байти. Кожен канал займає рівно 11 біт у вихідному масиві без

вирівнювання по байтових межах, через що кожен вихідний байт формується через комбінацію бітів двох сусідніх каналів за допомогою операцій побітового зсуву та об'єднання. Такий формат є частиною стандарту протоколу і безпосередньо інтерпретується польотним контролером без додаткового перетворення.

Запуск та зупинка модуля реалізовані симетрично. Метод запуску перевіряє прапор активності щоб уникнути подвійного запуску, скидає прапор телеметрії і запускає фоновий потік з переданими параметрами порту. Метод зупинки встановлює обидва прапори у хибне значення і блокуючи очікує завершення потоку — це гарантує, що після повернення з методу зупинки ресурси послідовного порту повністю звільнені і жоден потік більше не звертається до апаратного інтерфейсу.

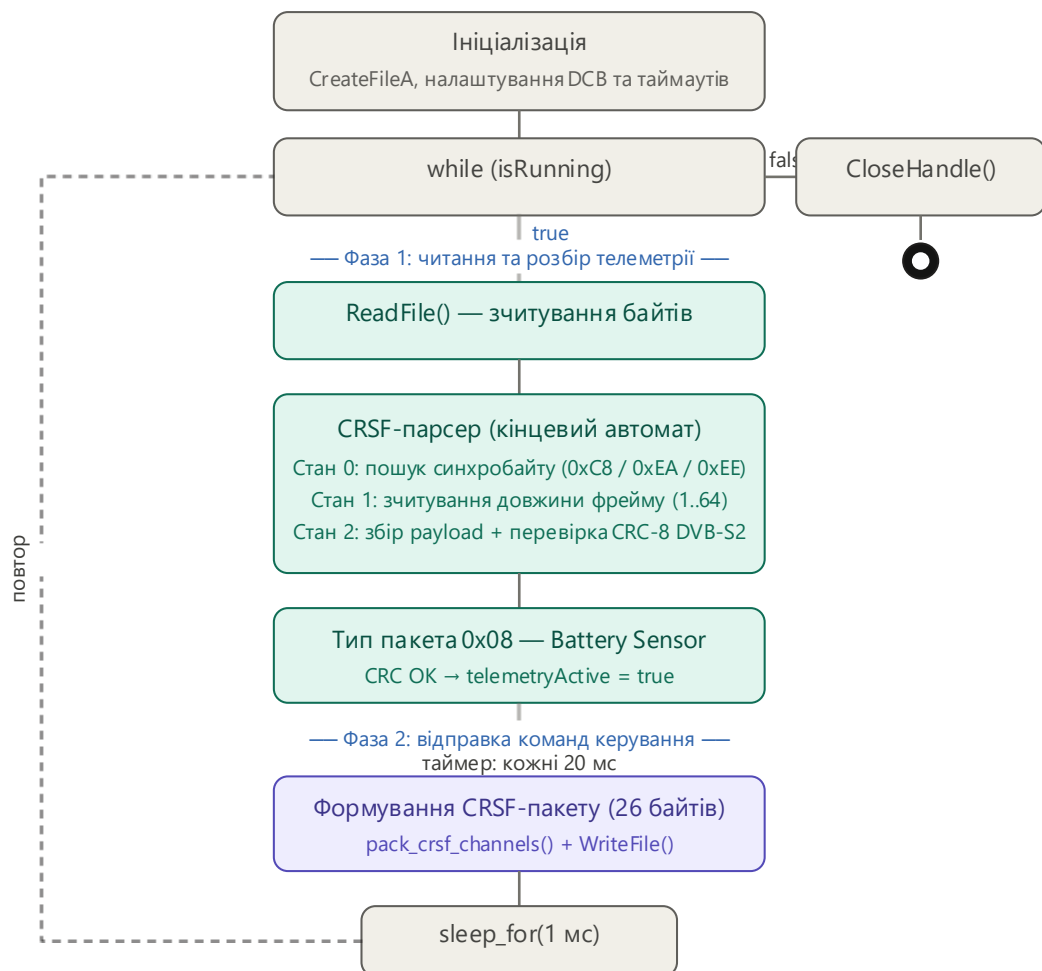


Рисунок 4.5 – Цикл функції керування ELRS передавачем

Фоновий потік відкриває послідовний порт через системний API з розширеним префіксом у назві пристрою, що є обов'язковою вимогою Windows для доступу до портів з номерами вище дев'яти. Після відкриття конфігурується структура параметрів порту: вісім біт даних, один стоп-біт, без перевірки парності. Тайм-аути читання налаштовуються особливим чином — встановлення максимального міжбайтового інтервалу при нульових загальних тайм-аутах переводить порт у режим негайного повернення. Це означає, що виклик читання завжди повертається одразу з тими байтами, що вже є у апаратному буфері, не чекаючи надходження нових. Завдяки цьому читання і запис можна чергувати в одному потоці без взаємних блокувань.

Кожна ітерація головного циклу складається з двох послідовних фаз. У першій фазі виконується читання наявних байтів і якщо вхідний буфер не порожній, вони обробляються поодиночі через кінцевий автомат з трьома станами. У нульовому стані автомат шукає байт синхронізації зі значень, що відповідають стандартним адресам пристроїв у протоколі CRSF. При знаходженні синхробайту він зберігається у внутрішньому буфері і автомат переходить у перший стан. У першому стані зчитується байт довжини фрейму і перевіряється його коректність: значення повинно бути у допустимих межах протоколу. Якщо довжина некоректна, автомат повертається у нульовий стан — це є захистом від сміттєвих даних, що можуть з'явитися у буфері при підключенні або перешкодах. При коректній довжині значення зберігається і автомат переходить у другий стан. У другому стані байти послідовно накопичуються у внутрішньому буфері до досягнення повної очікуваної довжини фрейму. Після цього обчислюється контрольна сума від байта типу до передостаннього байта і порівнюється з останнім байтом фрейму, який є еталонним значенням контрольної суми. Якщо значення збігаються, фрейм вважається валідним і перевіряється його тип. Тип, що відповідає датчику акумулятора, є ознакою активного польотного контролера — при першому отриманні такого валідного пакета прапор наявності телеметрії встановлюється в істинне значення і в журнал записується повідомлення про підключення дрона.

У другій фазі кожної ітерації перевіряється часовий інтервал з моменту попередньої відправки. Якщо з того моменту минуло щонайменше 20 мілісекунд, виконується формування та відправка пакета керування. Спочатку атомарно зчитуються поточні значення всіх шістнадцяти каналів керування, після чого формується повний CRSF-фрейм: байт синхронізації, байт довжини, байт типу, двадцять два байти запакованих значень каналів та байт контрольної суми. Контрольна сума обчислюється послідовним викликом функції CRC спочатку для байта типу, потім для кожного байта корисного навантаження. Сформований пакет записується у послідовний порт і часова мітка наступної відправки зсувається рівно на 20 мілісекунд вперед. Кожна ітерація завершується паузою в одну мілісекунду для зниження навантаження на процесор. Після виходу з циклу дескриптор порту закривається і в журнал записується повідомлення про завершення роботи.

4.1.5 Файл joystick.cpp

Даний файл реалізує зчитування даних з ігрового контролера та їх перетворення у значення каналів керування у форматі протоколу CRSF. Модуль організований навколо кінцевого автомата з трьома станами: стан очікування, стан калібрування та стан активної роботи. Перехід між станами визначає поведінку фонового потоку опитування, який в залежності від поточного стану виконує або збір даних для калібрування, або безпосереднє перетворення показань пристрою у команди керування.

Перелік доступних пристроїв отримується через мультимедійний API Windows шляхом послідовного запиту можливостей кожного зі слотів для ігрових контролерів. Для кожного доступного пристрою зчитується його текстова назва і формується рядок з порядковим номером та назвою для відображення оператору.

Перетворення сирого значення осі у значення каналу CRSF реалізоване як лінійне масштабування: вхідний діапазон значень від нуля до максимуму WinMM API відображається на допустимий діапазон протоколу від 172 до 1811 через

пропорційну формулу. Це забезпечує коректну передачу крайніх та центральних положень стіків без додаткових коригувань.

Процедура калібрування реалізована за принципом автоматичного виявлення прив'язки фізичних елементів керування до логічних каналів. На початку кожного кроку береться знімок поточних значень усіх шести осей та стану всіх кнопок як базові значення відліку. Фоновий потік у режимі калібрування безперервно вимірює відхилення кожної осі від базового значення і фіксує максимальне відхилення за весь час кроку, а також відстежує зміни у стані кнопок через побітове виключне АБО з базовим станом. Коли оператор підтверджує крок натисканням Enter, метод обробки кроку аналізує накопичені дані: якщо будь-яка вісь перевищила поріг у п'ятнадцять тисяч одиниць відхилення, вона вважається переміщеною і прив'язується до поточного каналу керування; якщо кнопок є, але осей немає — прив'язується перша змінена кнопка. Результат кожного кроку зберігається у словнику відповідностей, де ключем є назва каналу, а значенням — тип джерела (вісь або кнопка) та його індекс. Після десяти кроків автомат переходить у стан відкаліброваного пристрою. Оператор також може пропустити будь-який крок або скасувати калібрування повністю.

Фоновий потік після завершення калібрування переходить у режим активного перетворення даних. На кожній ітерації з частотою 50 Гц зчитується поточний стан пристрою, після чого для кожного з десяти налаштованих каналів за словником відповідностей визначається джерело: якщо це вісь — значення масштабується у діапазон CRSF через лінійне перетворення, якщо кнопка — канал отримує або максимальне, або мінімальне значення залежно від стану кнопки. Результируючий масив значень каналів формується спочатку у локальному буфері з нейтральними значеннями за замовчуванням, після чого всі шістнадцять значень атомарно записуються у глобальний масив каналів, доступний іншим модулям. Якщо під час роботи пристрій відключається, потік фіксує помилку опитування, скидає стан у режим очікування і завершується.

4.2 Реалізація серверної частини

Серверна частина системи реалізована як легковаговий ретранслятор на Go і відповідає виключно за маршрутизацію пакетів між клієнтами — вона не інтерпретує вміст даних і не зберігає відеопотік.

Центральною структурою даних є словник підключених клієнтів що видно на рис. 4.6, захищений читач-письменниковим м'ютексом. Кожен запис у словнику містить чотири поля: унікальний шістнадцятисимвольний ідентифікатор, з'єднання TCP, UDP-адресу та ідентифікатор цільового дрона для підписки на відео. Вибір читач-письменникового м'ютекса замість звичайного є свідомим рішенням — операції маршрутизації UDP-пакетів виконуються набагато частіше ніж операції реєстрації клієнтів, тому блокування лише на запис дозволяє кільком горутинам одночасно читати словник без взаємного очікування.

Сервер слухає одночасно обидва транспортні протоколи на порту 80. TCP-слухач запускається в головній горутині, UDP-слухач — у фоновій. Як показано на першому етапі діаграми, при кожному новому TCP-з'єднанні породжується окрема горутина обробки: вона генерує криптографічно випадковий ідентифікатор через восьмибайтний масив випадкових даних перетворений у шістнадцятковий рядок, реєструє клієнта у словнику і одразу надсилає йому рядок авторизації з присвоєним ідентифікатором. Це єдиний момент, коли клієнт дізнається свій ідентифікатор — надалі він зобов'язаний використовувати його як префікс кожного UDP-пакета. При розриві TCP-з'єднання горутина видаляє запис клієнта зі словника і завершується.

UDP-слухач читає датаграми у нескінченному циклі в єдиній горутині. Перші шістнадцять байтів кожного пакета трактуються як ідентифікатор відправника, решта — як корисне навантаження. Якщо ідентифікатор не знайдено у словнику клієнтів, пакет відкидається без обробки — це природний захист від пакетів з невідомих джерел. Як видно з третього етапу діаграми, UDP-адреса клієнта не передається при реєстрації через TCP, а визначається динамічно при першому отриманні UDP-пакета від нього: якщо збережена адреса відсутня або не збігається

з адресою відправника поточного пакета, вона оновлюється. Саме тому клієнти надсилають початковий ідентифікаційний пакет одразу після встановлення TCP-з'єднання.

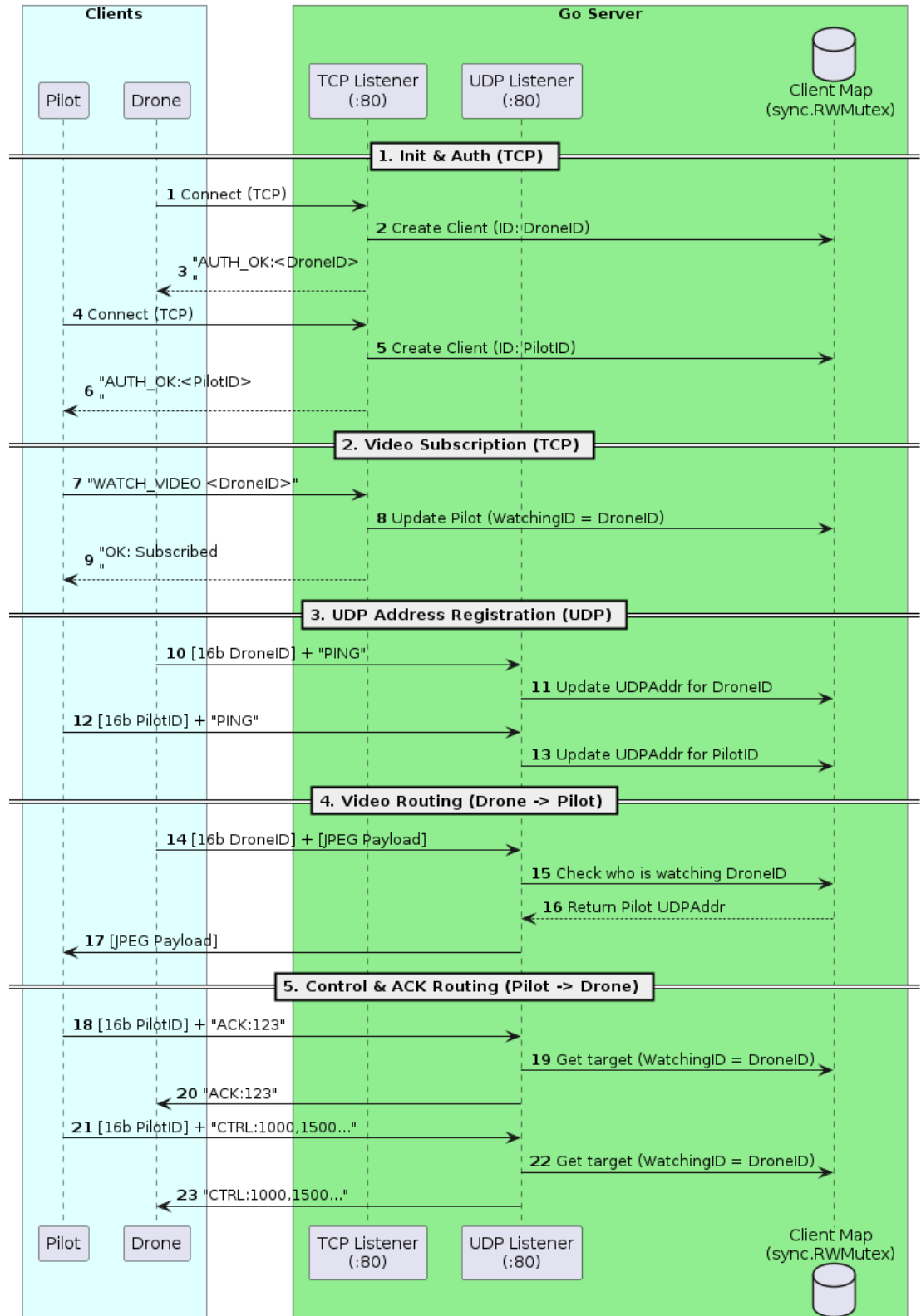


Рисунок 4.6 – Діаграма послідовності життєвого циклу пакетів

Маршрутизація пакетів реалізована через послідовну перевірку префіксів корисного навантаження. Якщо пакет містить префікс команди керування і відправник має заповнений ідентифікатор цільового дрона — пакет пересилається безпосередньо на UDP-адресу цього дрона. Якщо пакет є підтвердженням доставки відеокадру — він так само адресно пересилається дрону, після чого обробка завершується щоб підтвердження не потрапило в логіку розсилки відео. В усіх інших випадках пакет вважається відеочанком від дрона і розсилається усім клієнтам, у яких ідентифікатор підписки збігається з ідентифікатором відправника. Цей механізм відображено на четвертому та п'ятому етапах діаграми і дозволяє одному дрону одночасно обслуговувати кількох пілотів без будь-яких змін у коді.

TCP-частина обробки команд реалізована в окремому файлі і наразі підтримує єдину команду підписки на відеопотік. При отриманні цієї команди ідентифікатор цільового дрона записується у поле підписки клієнта під захистом запису м'ютекса. Саме наявність цього поля у клієнта-пілота є умовою для маршрутизації відеопакетів на його адресу, що видно з другого та четвертого етапів діаграми.

Висновки до розділу 4

У четвертому розділі детально описано програмну реалізацію всіх компонентів системи. Клієнтський застосунок побудований за чотирьохшаровою архітектурою з чіткою декомпозицією на модулі: головний модуль координує підсистеми та інтерфейс, мережевий модуль реалізує двосторонній UDP-обмін з адаптивною якістю стиснення, модуль камери захоплює відеопотік через DirectShow, модуль серійного зв'язку реалізує парсер CRSF-протоколу на основі кінцевого автомата, модуль джойстика — автоматичну прив'язку фізичних елементів керування до логічних каналів. Серверна частина реалізована як легковаговий маршрутизатор, що пересилає UDP-пакети між клієнтами без буферизації, що забезпечує мінімальну затримку транзитної передачі.

5 РОБОТА КОРИСТУВАЧА З РОЗРОБЛЕНОЮ СИСТЕМОЮ

5.1 Інтерфейс користувача та система команд

Графічний інтерфейс користувача клієнтського застосунку "Drone Control Client" реалізований у мінімалістичному термінальному стилі з використанням темної теми, зображено на рисунку 4.5.1 .Робочий простір логічно сегментований на п'ять функціональних блоків:



Рисунок 5.1 – інтерфейс програми користувача

Центральна область - Найбільша за площею зона, зарезервована для рендерингу відеопотоку з камери дрона або локального пристрою відеозахоплення.

Історія команд (Ліва верхня панель) - Блок для відображення хронології введених оператором текстових команд.

Ввід (ліва нижня панель) - Інтерактивне поле для введення команд керування системою.

Логування системи (права панель) - Журнал системних подій. Використовується для виведення довідкової інформації, статусів ініціалізації модулів, мережових повідомлень та діагностики помилок.

Поле каналів керування (нижня панель) - Блок моніторингу телеметрії керування. У реальному часі візуалізує стан основних осей керування (Roll, Pitch, Throttle, Yaw) та допоміжних каналів (AUX1–AUX12) у відсотках за допомогою графічних шкал. Також відображає загальний статус підключення.

Взаємодія з програмою здійснюється виключно через інтерфейс командного рядка (CLI). Нижче наведено таблицю з описом усіх підтримуваних команд, їх призначенням та синтаксисом використання, згідно з вбудованим системним посібником.

Таблиця 5.1 – Команди керування програмою

Команда	Призначення	Синтаксис
help	Виведення довідкового посібника з переліком доступних команд у панель логів.	help
cam	Керування модулем відеозахоплення. Дозволяє отримати список доступних камер, запустити трансляцію за ідентифікатором або зупинити її.	cam list cam start <ID> cam stop
serial	Керування модулем прийому телеметрії. Відкриває послідовний COM-порт із заданою швидкістю (baud rate) або закриває його	serial start <PORT> <BAUD> serial stop

Продовження таблиці 5.1

joy	Керування модулем прийому телеметрії. Відкриває послідовний COM-порт із заданою швидкістю (baud rate) або закриває його.	joy list joy calib <ID>
connect	Ініціалізація мережевого з'єднання з релейним сервером. Вимагає вказівки ролі (дрон, пілот, тестовий пілот), IP-адреси та порту сервера. Для пілота додатково вказується ідентифікатор дрона.	connect drone <IP:PORT> connect pilot <IP:PORT> <DRONE_ID> connect test_pilot <IP:PORT> <DRONE_ID>
cpid	Копіювання локального ідентифікатора мережі у буфер обміну операційної системи.	cpid
arm	Команда зняття блокування двигунів БПЛА (доступна виключно в режимі пілота).	arm
disarm	Команда примусового блокування двигунів БПЛА (доступна виключно в режимі пілота).	disarm

Продовження таблиці 5.1

mode	Зміна поточного режиму стабілізації польоту (Angle, Horizon, Acro).	mode angle mode horizon mode acro
disconnect	Примусове розірвання активного мережевого з'єднання з сервером.	disconnect
stop	Глобальна зупинка та деініціалізація всіх активних фонових модулів (камери, COM-порту, мережевого клієнта, опитування джойстика).	stop

5.2 Алгоритми розгортання клієнтських терміналів



Рисунок 5.2 – Блок схема типового сценарію роботи сторони дрона

Сценарій розгортання комунікаційного вузла на стороні безпілотної літального апарата починається з етапу апаратної ініціалізації. Процес вимагає паралельного налаштування периферійних пристроїв: запускається радіопередавач, подається живлення на дрон для встановлення зв'язку з бортовим приймачем, а також ініціалізується модуль захоплення відеосигналу. У разі відсутності або програмної недоступності будь-якого з цих компонентів система циклічно очікує їх підключення, блокуючи подальші кроки. Після успішної апаратної перевірки клієнтський застосунок ініціює TCP-з'єднання з релейним сервером. Сервер реєструє новий вузол, генерує унікальний ідентифікатор сесії (ID) та переводить програму в режим очікування пілота. Як тільки оператор підключається за цим ідентифікатором, термінал переходить в активну фазу маршрутизації: починається безперервна трансляція оцифрованого відеопотоку у мережу та зворотне передавання отриманих команд керування безпосередньо на польотний контролер. У разі розірвання зв'язку або примусового завершення сесії система автоматично повертається до етапу очікування підключення до сервера.



Рисунок 5.3 – Блок-схема типового сценарію роботи сторони пілота

Сценарій розгортання робочого місця оператора (пілота) зосереджений переважно на підготовці засобів вводу. Алгоритм стартує з вибору фізичного маніпулятора (пульта керування) та його калібрування для забезпечення точності передачі керівних сигналів. Якщо система не виявляє підключеного пульта, процес зупиняється на етапі очікування апаратного вводу. Після успішного калібрування осей оператор використовує унікальний ID сесії, згенерований терміналом дрона, для підключення до відповідної кімнати на релейному сервері. Встановлення з'єднання переводить програму у робочий режим телекерування, де застосунок декодує та виводить на екран відеопотік з віддаленого БПЛА, одночасно опитуючи маніпулятор і відправляючи пакети керування у мережу. Завершення сесії з боку пілота розриває мережевий транзит і повертає алгоритм до початкового стану вибору та перевірки пульта керування.

5.3 Тестування програмного продукту

Тестування проводилось із релейним сервером, розгорнутим на віддаленій Debian-машині. Клієнти у ролях дрона та пілота підключались до сервера через публічну мережу, увесь обмін відеопотоком і командами керування здійснювався виключно через ретранслятор. Перевірялись стабільність відеопотоку, затримка каналу керування та мережеве навантаження на сервер.

На рисунку 5.4 показано вікно клієнта у режимі пілота під час активної сесії. На зображенні видно накладений OSD польотного контролера — назву апарату, режим Angle, показники акумулятора та RSSI, що підтверджує проходження сигналу через увесь ланцюжок: бортова камера → відеопередавач → аналоговий приймач → USB → клієнт-дрон → сервер → клієнт-пілот. Виміряна затримка каналу склала 36 мс.

Таке значення затримки досягнуте за рахунок чотирьох рішень: транспорт UDP без механізму повторної передачі; мінімальний сервер, що пересилає пакети без буферизації; розбиття кадрів на чанки по 1300 байтів, що виключає IP-

фрагментацію; адаптивне зниження якості JPEG при зростанні затримки вище порогового значення.



Рисунок 5.4 – Демонстрація затримки програми

На рисунку 5.5 показано стан мережевого інтерфейсу сервера, знятий утилітою `bmop`. Вхідний потік — 795 KiB/c при 719 пакетах/с, вихідний — 770 KiB/c при 692 пакетах/с. Асиметрія між напрямками пояснюється тим, що службові пакети (підтвердження доставки та синхронізація затримки) йдуть у зворотному напрямку і значно менші за відеочанки.

Вимірний трафік відповідає розрахунковому. Кадр 640×480 у JPEG з якістю 20–25 займає 25–30 KiB, при 30 fps це дає 750–900 KiB/c — узгоджується з даними `bmop`. Кадр розміром 28 KiB розбивається на 22 чанки, що при 30 fps дає близько 660 відеопакетів/с; додаючи 50 пакетів керування та поодинокі службові пакети, отримуємо вимірні ~719 пакетів/с.

ВИСНОВКИ

У ході виконання дипломного проєкту було вирішено інженерне завдання з розробки програмного комплексу для дистанційного керування безпілотними літальними апаратами через глобальну мережу Інтернет. На основі аналізу недоліків традиційного радіозв'язку, зокрема жорсткої залежності високочастотного відео від прямої видимості, було спроектовано гібридну клієнт-серверну архітектуру. Її основою став релейний сервер маршрутизації, реалізований мовою Go, який забезпечує проходження механізмів трансляції адрес (NAT) кінцевих інтернет-провайдерів. Застосування транзитної передачі через протокол UDP без розпакування корисного навантаження на сервері дозволило звести програмну затримку до мінімуму.

Для наземної станції пілота та бортового комунікаційного вузла створено клієнтське програмне забезпечення мовою C++. Архітектура з багаторівневою ізоляцією потоків забезпечила стабільну роботу графічного інтерфейсу на базі OpenCV одночасно з асинхронним опитуванням периферії. У системі реалізовано низькорівневу взаємодію з апаратними модулями: пряме зчитування переривань маніпулятора через API WinMM, адаптивне стиснення відеокадрів у формат JPEG із фрагментацією під стандартні інтернет-пакети (MTU) та обмін даними через COM-порти за допомогою Windows Serial API.

Важливим етапом роботи стала інтеграція зі стандартом ExpressLRS шляхом програмної обробки протоколу CRSF. Реалізовано алгоритми пакування 11-бітних змінних керування у кадри стандарту 0x16 із розрахунком контрольної суми CRC8, а також зворотний парсинг телеметрії для інформування оператора про стан зв'язку. Розроблена програма повністю абстрагує процес пілотування від фізичних обмежень радіоканалу, дозволяючи керувати літальним апаратом на будь-якій відстані за умови наявності стабільного інтернет-з'єднання.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Страуструп Б. Мова програмування C++. Спеціальне видання. Київ : Діалектика, 2019. 1136 с.
2. Таненбаум Е., Везерілл Д. Комп'ютерні мережі. 6-те вид. Київ : Діалектика, 2022. 928 с.
3. Austin R. Unmanned Aircraft Systems: UAVs Design, Development and Deployment. Chichester : John Wiley & Sons, 2010. 332 p.
4. Bradski G., Kaehler A. Learning OpenCV 3: Computer Vision in C++ with the OpenCV Library. Sebastopol : O'Reilly Media, 2017. 1018 p.
5. Donovan A. A. A., Kernighan B. W. The Go Programming Language. New York : Addison-Wesley Professional, 2015. 400 p.
6. ExpressLRS Open Source Radio Link. URL: <https://www.expresslrs.org/> (дата звернення: 19.05.2026).
7. RFC 3022: Traditional IP Network Address Translator (Traditional NAT) / P. Srisuresh, K. Egevang. URL: <https://datatracker.ietf.org/doc/html/rfc3022> (дата звернення: 19.05.2026).
8. Richardson I. E. G. The H.264 Advanced Video Compression Standard. 2nd ed. Chichester : John Wiley & Sons, 2010. 348 p.
9. TBS Crossfire Protocol Specification / Team BlackSheep. URL: <https://github.com/tbs-fpv/tbs-crsf-spec/blob/main/crsf.md> (дата звернення: 19.05.2026).
10. Windows API Documentation / Microsoft Learn. URL: <https://learn.microsoft.com/en-us/windows/win32/> (дата звернення: 19.05.2026).

ДОДАТОК А

Програмне забезпечення моделювання віддаленого контролю

Лістинг головних модулів

УКР.НТУУ «КПІ ім. Ігоря Сікорського» НН ІАТЕ ТВ-22

Аркушів 26

Київ – 2026

Main.cpp

```
#define NOMINMAX
#include <opencv2/opencv.hpp>
#include <iostream>
#include <vector>
#include <string>
#include <chrono>
#include <atomic>
#include <sstream>
#include <algorithm>
#include <windows.h>
#include "logger.h"
#include "camera.h"
#include "serial_rx.h"
#include "network.h"
#include "joystick.h"

// Глобальний масив каналів
std::atomic<uint16_t> rc_channels[16];

// Допоміжна функція для переносу тексту в логах
std::vector<std::string> wrapText(const std::string& text, int maxWidth, double
fontScale, int thickness) {
    std::vector<std::string> wrappedLines;
    std::string currentLine = "";
    int baseline = 0;
    for (char c : text) {
        std::string testLine = currentLine + c;
        cv::Size textSize = cv::getTextSize(testLine, cv::FONT_HERSHEY_SIMPLEX,
fontScale, thickness, &baseline);
        if (textSize.width > maxWidth && !currentLine.empty()) {
            wrappedLines.push_back(currentLine);
            currentLine = std::string(1, c);
        } else {
            currentLine += c;
        }
    }
    if (!currentLine.empty()) wrappedLines.push_back(currentLine);
    return wrappedLines;
}

// Функція запису в буфер обміну Windows
void setClipboardText(const std::string& text) {
    if (OpenClipboard(nullptr)) {
        EmptyClipboard();
        HGLOBAL hMem = GlobalAlloc(GMEM_MOVEABLE, text.size() + 1);
        if (hMem != nullptr) {
```

```

        memcpy(GlobalLock(hMem), text.c_str(), text.size() + 1);
        GlobalUnlock(hMem);
        SetClipboardData(CF_TEXT, hMem);
    }
    CloseClipboard();
}

// Функція читання з буфера обміну Windows (Ctrl+V)
std::string getClipboardText() {
    std::string text = "";
    if (OpenClipboard(nullptr)) {
        HANDLE hData = GetClipboardData(CF_TEXT);
        if (hData != nullptr) {
            char* pszText = static_cast<char*>(GlobalLock(hData));
            if (pszText != nullptr) {
                text = pszText;
                GlobalUnlock(hData);
            }
        }
        CloseClipboard();
    }
    text.erase(std::remove(text.begin(), text.end(), '\n'), text.end());
    text.erase(std::remove(text.begin(), text.end(), '\r'), text.end());
    return text;
}

int main() {
    const int WINDOW_WIDTH = 1400;
    const int WINDOW_HEIGHT = 720;
    const int LEFT_PANEL_W = 300;
    const int RIGHT_PANEL_W = 300;
    const int INPUT_BOX_H = 60;
    const int VIDEO_MAX_HEIGHT = 600;

    const std::string chNames[16] = {
        "Roll", "Pitch", "Throttle", "Yaw",
        "ARM (AUX1)", "AUX2", "AUX3", "MODE (AUX4)",
        "AUX5", "AUX6", "AUX7", "AUX8",
        "AUX9", "AUX10", "AUX11", "AUX12"
    };

    // Ініціалізація значень каналів за замовчуванням
    for(int i = 0; i < 16; i++) rc_channels[i].store(992);
    rc_channels[2].store(172); // Газ в нуль
    rc_channels[4].store(172); // Disarm
    rc_channels[7].store(172); // Angle Mode

    Logger::addLog("[SYSTEM] Application started.");
}

```

```

std::vector<std::string> commandHistory;
std::vector<std::string> rawHistory;
int historyIndex = 0;
std::string currentCommand = "";
int cursorPosition = 0;
bool isAppRunning = true;

try {
    cv::namedWindow("Drone Control Client", cv::WINDOW_AUTOSIZE);

    while (isAppRunning) {
        if (cv::getWindowProperty("Drone Control Client", cv::WND_PROP_VISIBLE) <
1) {
            Logger::addLog("[SYSTEM] Window closed by user.");
            isAppRunning = false;
            break;
        }

        cv::Mat canvas(WINDOW_HEIGHT, WINDOW_WIDTH, CV_8UC3, cv::Scalar(0, 0,
0));
        cv::Mat frameCopy;

        // Отримання кадру залежно від ролі
        if (NetworkManager::getRole() == ClientRole::TEST_PILOT) {
            NetworkManager::getPilotFrame(frameCopy);
        } else {
            CameraManager::getFrame(frameCopy);
        }
        std::vector<std::string> logsCopy = Logger::getLogs();

        // --- 1. ВІДЕО ТА ПАНЕЛІ ---
        int currentVideoHeight = 0;
        if (!frameCopy.empty()) {
            currentVideoHeight = frameCopy.rows;
            cv::Rect roi(LEFT_PANEL_W, 0, frameCopy.cols, frameCopy.rows);
            frameCopy.copyTo(canvas(roi));
        } else {
            currentVideoHeight = VIDEO_MAX_HEIGHT;
        }

        cv::line(canvas, cv::Point(LEFT_PANEL_W, 0), cv::Point(LEFT_PANEL_W,
WINDOW_HEIGHT), cv::Scalar(255, 255, 255), 1);
        cv::line(canvas, cv::Point(WINDOW_WIDTH - RIGHT_PANEL_W, 0),
cv::Point(WINDOW_WIDTH - RIGHT_PANEL_W, WINDOW_HEIGHT), cv::Scalar(255, 255, 255),
1);
        cv::line(canvas, cv::Point(LEFT_PANEL_W, currentVideoHeight),
cv::Point(WINDOW_WIDTH - RIGHT_PANEL_W, currentVideoHeight), cv::Scalar(255, 255,
255), 1);

```

```

        std::string roleStr = (NetworkManager::getRole() == ClientRole::DRONE) ?
"[MODE: DRONE]" :
                                (NetworkManager::getRole() ==
ClientRole::TEST_PILOT) ? "[MODE: TEST_PILOT]" : "[MODE: OFFLINE]";

        // Зчитуємо RTT з мережевого модуля, якщо є з'єднання
        std::string rttStr = "";
        if (NetworkManager::isConnected()) {
            rttStr = " | RTT: " + std::to_string(NetworkManager::getRTT()) +
" ms";
        }

        cv::putText(canvas, "RC CHANNELS LIVE " + roleStr + rttStr,
cv::Point(LEFT_PANEL_W + 10, currentVideoHeight + 20), cv::FONT_HERSHEY_SIMPLEX,
0.45, cv::Scalar(255, 255, 255), 1);

        // --- 2. ВІДОБРАЖЕННЯ КАНАЛІВ (Смужки та назви) ---
        int chStartX = LEFT_PANEL_W + 10;
        int chStartY = currentVideoHeight + 40;
        int colWidth = 195;

        for (int i = 0; i < 16; i++) {
            int col = i % 4, row = i / 4;
            int x = chStartX + col * colWidth, y = chStartY + row * 20;
            uint16_t val = rc_channels[i].load();
            float pct = (val - 172.0f) / (1811.0f - 172.0f);
            pct = std::max(0.0f, std::min(1.0f, pct));
            int barWidth = 60, filledWidth = static_cast<int>(pct * barWidth);

            cv::putText(canvas, chNames[i] + ":", cv::Point(x, y),
cv::FONT_HERSHEY_SIMPLEX, 0.35, cv::Scalar(200, 200, 200), 1);
            cv::rectangle(canvas, cv::Rect(x + 85, y - 8, barWidth, 10),
cv::Scalar(100, 100, 100), 1);
            if (filledWidth > 0) cv::rectangle(canvas, cv::Rect(x + 85, y - 8,
filledWidth, 10), cv::Scalar(255, 255, 255), cv::FILLED);
            cv::putText(canvas, std::to_string(static_cast<int>(pct * 100)) +
"%", cv::Point(x + 150, y), cv::FONT_HERSHEY_SIMPLEX, 0.35, cv::Scalar(255, 255,
255), 1);
        }

        // --- 3. ЛОГИ (Знизу вгору) ---
        cv::Mat rightPanel = canvas(cv::Rect(WINDOW_WIDTH - RIGHT_PANEL_W, 0,
RIGHT_PANEL_W, WINDOW_HEIGHT));
        cv::putText(rightPanel, "SYSTEM LOGS", cv::Point(10, 25),
cv::FONT_HERSHEY_SIMPLEX, 0.5, cv::Scalar(255, 255, 255), 1);
        cv::line(rightPanel, cv::Point(0, 35), cv::Point(RIGHT_PANEL_W, 35),
cv::Scalar(255, 255, 255), 1);

```

```

        int logY = WINDOW_HEIGHT - 20;
        for (auto it = logsCopy.rbegin(); it != logsCopy.rend(); ++it) {
            if (logY < 50) break;
            std::vector<std::string> wrappedLogs = wrapText(*it, RIGHT_PANEL_W -
20, 0.4, 1);
            for (auto rit = wrappedLogs.rbegin(); rit != wrappedLogs.rend();
++rit) {
                if (logY < 50) break;
                cv::putText(rightPanel, *rit, cv::Point(10, logY),
cv::FONT_HERSHEY_SIMPLEX, 0.4, cv::Scalar(255, 255, 255), 1);
                logY -= 15;
            }
        }

        // --- 4. ТЕРМІНАЛ ТА ІСТОПІЯ ---
        cv::Mat leftPanel = canvas(cv::Rect(0, 0, LEFT_PANEL_W, WINDOW_HEIGHT));
        cv::rectangle(leftPanel, cv::Rect(0, WINDOW_HEIGHT - INPUT_BOX_H + 2, 58,
INPUT_BOX_H - 4), cv::Scalar(0, 0, 0), cv::FILLED);
        cv::rectangle(canvas, cv::Rect(0, WINDOW_HEIGHT - INPUT_BOX_H,
LEFT_PANEL_W, INPUT_BOX_H), cv::Scalar(255, 255, 255), 1);
        cv::putText(leftPanel, "Input:", cv::Point(10, WINDOW_HEIGHT - 35),
cv::FONT_HERSHEY_SIMPLEX, 0.5, cv::Scalar(255, 255, 255), 1);

        std::string displayCmd = currentCommand;
        std::string cursorStr =
((std::chrono::system_clock::now().time_since_epoch().count() / 5000000) % 2 == 0) ?
"|" : " ";
        if (cursorPosition >= 0 && cursorPosition <= (int)displayCmd.length())
displayCmd.insert(cursorPosition, cursorStr);

        std::string visibleCmd = displayCmd;
        int baseline = 0, maxTextWidth = LEFT_PANEL_W - 70;
        while (true) {
            cv::Size textSize = cv::getTextSize(visibleCmd,
cv::FONT_HERSHEY_SIMPLEX, 0.5, 1, &baseline);
            if (textSize.width > maxTextWidth && !visibleCmd.empty())
visibleCmd.erase(0, 1); else break;
        }
        cv::putText(leftPanel, visibleCmd, cv::Point(60, WINDOW_HEIGHT - 35),
cv::FONT_HERSHEY_SIMPLEX, 0.5, cv::Scalar(255, 255, 255), 1);

        cv::putText(leftPanel, "COMMAND HISTORY", cv::Point(10, 25),
cv::FONT_HERSHEY_SIMPLEX, 0.5, cv::Scalar(255, 255, 255), 1);
        cv::line(leftPanel, cv::Point(0, 35), cv::Point(LEFT_PANEL_W, 35),
cv::Scalar(255, 255, 255), 1);

        int historyY = WINDOW_HEIGHT - INPUT_BOX_H - 20;

```

```

        for (auto it = commandHistory.rbegin(); it != commandHistory.rend();
++it) {
            if (historyY < 50) break;
            std::vector<std::string> wrappedCmds = wrapText(*it, LEFT_PANEL_W -
20, 0.5, 1);
            for (auto rit = wrappedCmds.rbegin(); rit != wrappedCmds.rend();
++rit) {
                if (historyY < 50) break;
                cv::putText(leftPanel, *rit, cv::Point(10, historyY),
cv::FONT_HERSHEY_SIMPLEX, 0.5, cv::Scalar(255, 255, 255), 1);
                historyY -= 25;
            }
        }

        cv::imshow("Drone Control Client", canvas);

        // --- 5. ОБРОБКА КЛАВИАТУРЫ ТА ПАРСЕР ---
        int key = cv::waitKeyEx(15);
        if (key != -1) {
            if (key == 2490368 || key == 65362) { // ВГОРУ
                if (!rawHistory.empty() && historyIndex > 0) {
                    historyIndex--; currentCommand = rawHistory[historyIndex];
cursorPosition = currentCommand.length();
                }
            }
            else if (key == 2621440 || key == 65364) { // ВНИЗ
                if (!rawHistory.empty() && historyIndex < (int)rawHistory.size()
- 1) {
                    historyIndex++; currentCommand = rawHistory[historyIndex];
cursorPosition = currentCommand.length();
                } else if (historyIndex == (int)rawHistory.size() - 1) {
                    historyIndex++; currentCommand = ""; cursorPosition = 0;
                }
            }
            else if (key == 2424832 || key == 65361) { // ВЛИВО
                if (cursorPosition > 0) cursorPosition--;
            }
            else if (key == 2555904 || key == 65363) { // ВПРАВО
                if (cursorPosition < (int)currentCommand.length())
cursorPosition++;
            }
            else {
                int asciiCode = key & 0xFF;
                if (asciiCode == 27) isAppRunning = false;
                else if (asciiCode == 13 || asciiCode == 10) { // ENTER
                    // --- БЛОК ДЛЯ КАЛІБРУВАННЯ ---
                    if (JoystickManager::isCalibrating()) {
                        if (currentCommand.empty()) {
                            JoystickManager::nextCalibrationStep();

```

```

        continue;
    } else if (currentCommand == "skip") {
        JoystickManager::skipCalibrationStep();
        currentCommand = ""; cursorPosition = 0;
        continue;
    } else if (currentCommand == "cancel") {
        JoystickManager::cancelCalibration();
        currentCommand = ""; cursorPosition = 0;
        continue;
    }
}
if (!currentCommand.empty()) {
    commandHistory.push_back("> " + currentCommand);
    rawHistory.push_back(currentCommand);
    historyIndex = rawHistory.size();
    Logger::addLog("[CMD] " + currentCommand);

    std::vector<std::string> args;
    std::stringstream ss(currentCommand);
    std::string arg;
    while (ss >> arg) args.push_back(arg);

    if (!args.empty()) {
        std::string cmd = args[0];

        if (cmd == "help") {
            Logger::addLog("=== SYSTEM COMMANDS MANUAL ===");
            Logger::addLog("help      - Show this manual.");
            Logger::addLog("cam      - Video input (Usage:
cam list | cam start <ID> | cam stop)");
            Logger::addLog("serial    - Telemetry RX (Usage:
serial start <PORT> <BAUD> | serial stop)");
            Logger::addLog("joy      - RC Controller
(Usage: joy list | joy calib <ID>)");
            Logger::addLog("connect   - Establish network
link. Usage:");
            Logger::addLog("          > connect drone
<IP:PORT>");
            Logger::addLog("          > connect pilot
<IP:PORT> <DRONE_ID>");
            Logger::addLog("          > connect test_pilot
<IP:PORT> <DRONE_ID>");
            Logger::addLog("cpid     - Copy your local
network ID to clipboard.");
            Logger::addLog("arm      - Arm the drone motors
(Pilot only).");
            Logger::addLog("disarm   - Disarm the drone
motors (Pilot only).");

```

```

        Logger::addLog("mode          - Change flight mode
(Usage: mode angle|horizon|acro)");
        Logger::addLog("disconnect - Close current
network connection.");
        Logger::addLog("stop          - Kill all active
modules (Cam, Serial, Net, Joy).");
    }
    else if (cmd == "cam") {
        if (args.size() > 1 && args[1] == "list") {
            Logger::addLog("[SYS] Scanning video
devices...");

            std::vector<std::string> cams =
CameraManager::listCameras();

            for(const auto& cam : cams)
                Logger::addLog("  " + cam);
        }
        else if (args.size() > 2 && args[1] == "start")
        {
            int camId = std::stoi(args[2]);
            CameraManager::stop(); CameraManager::start(camId);
        }
        else if (args.size() > 1 && args[1] == "stop")
        {
            CameraManager::stop();
        }
        else {
            Logger::addLog("[ERR] Usage: cam list | cam
start <ID> | cam stop");
        }
    }
    else if (cmd == "serial") {
        if (args.size() > 3 && args[1] == "start") {
            SerialRX::stop(); SerialRX::start(args[2],
std::stoi(args[3]));
        }
        else if (args.size() > 1 && args[1] == "stop")
        {
            SerialRX::stop();
        }
        else {
            Logger::addLog("[ERR] Usage: serial start
<PORT> <BAUD> | serial stop");
        }
    }
    else if (cmd == "connect") {
        if (args.size() > 2 && args[1] == "drone") {
            if (!CameraManager::isCameraActive()) {
                Logger::addLog("[ERR] Connection blocked:
Camera is not running.");
            }
            else if (!SerialRX::isTelemetryActive()) {
                Logger::addLog("[ERR] Connection blocked:
No RC telemetry from serial.");
            }
            else {
                size_t cp = args[2].find(':');

```

```

        if (cp != std::string::npos) {
            NetworkManager::connectAsDrone(args[2]
.substr(0, cp), std::stoi(args[2].substr(cp + 1)));
        } else {
            Logger::addLog("[ERR] Invalid format.
Expected: connect drone <IP:PORT>");
        }
    }
} else if (args.size() > 3 && args[1] == "pilot")
{
    if (!JoystickManager::isCalibrated()) {
        Logger::addLog("[ERR] Connection blocked:
Joystick not calibrated.");
        Logger::addLog("[SYS] Connect remote and
run: 'joy list' -> 'joy calib <ID>'.");
    } else {
        size_t cp = args[2].find(':');
        if (cp != std::string::npos) {
            NetworkManager::connectAsPilot(args[2]
.substr(0, cp), std::stoi(args[2].substr(cp + 1)), args[3]);
        } else {
            Logger::addLog("[ERR] Invalid format.
Expected: connect pilot <IP:PORT> <DRONE_ID>");
        }
    }
} else if (args.size() > 3 && args[1] ==
"test_pilot") {
    size_t cp = args[2].find(':');
    if (cp != std::string::npos) {
        NetworkManager::connectAsPilot(args[2].su
bstr(0, cp), std::stoi(args[2].substr(cp + 1)), args[3]);
    } else {
        Logger::addLog("[ERR] Invalid format.
Expected: connect test_pilot <IP:PORT> <DRONE_ID>");
    }
} else {
    Logger::addLog("[ERR] Usage: connect drone
<IP:PORT> | connect [test_]pilot <IP:PORT> <ID>");
}
} else if (cmd == "joy") {
    if (args.size() > 1 && args[1] == "list") {
        Logger::addLog("[SYS] Scanning
joysticks...");
        std::vector<std::string> joys =
JoystickManager::listJoysticks();
        for(const auto& j : joys) Logger::addLog(" "
+ j);
    }
}

```

```

        } else if (args.size() > 2 && args[1] == "calib")
{
    JoystickManager::startCalibration(std::stoi(a
rgs[2]));

    } else {
        Logger::addLog("[ERR] Usage: joy list | joy
calib <ID>");
    }
}
else if (cmd == "cpid") {
    std::string id = NetworkManager::getMyID();
    if (!id.empty()) {
        setClipboardText(id);
        Logger::addLog("[SYS] My ID " + id + " copied
to clipboard.");
    } else {
        Logger::addLog("[ERR] Not connected to
server. No ID assigned yet.");
    }
}
else if (cmd == "arm") {
    if (NetworkManager::getRole() ==
ClientRole::TEST_PILOT || NetworkManager::getRole() == ClientRole::NONE) {
        rc_channels[4].store(1811);
        Logger::addLog("[PILOT] Sent: ARM");
    } else {
        Logger::addLog("[ERR] Command only available
for pilot.");
    }
}
else if (cmd == "disarm") {
    if (NetworkManager::getRole() ==
ClientRole::TEST_PILOT || NetworkManager::getRole() == ClientRole::NONE) {
        rc_channels[4].store(172);
        Logger::addLog("[PILOT] Sent: DISARM");
    } else {
        Logger::addLog("[ERR] Command only available
for pilot.");
    }
}
else if (cmd == "mode") {
    if ((NetworkManager::getRole() ==
ClientRole::TEST_PILOT || NetworkManager::getRole() == ClientRole::NONE) &&
args.size() > 1) {
        if (args[1] == "angle") {
            rc_channels[7].store(172); Logger::addLog("[PILOT] Set Mode: ANGLE"); }
        else if (args[1] == "horizon") {
            rc_channels[7].store(992); Logger::addLog("[PILOT] Set Mode: HORIZON"); }
    }
}

```

```

        else if (args[1] == "acro") {
rc_channels[7].store(1811); Logger::addLog("[PILOT] Set Mode: ACRO"); }
        else { Logger::addLog("[ERR] Unknown mode.
Usage: mode angle|horizon|acro"); }
    } else {
        Logger::addLog("[ERR] Usage: mode
<angle|horizon|acro>");
    }
    }
    else if (cmd == "disconnect") {
        NetworkManager::disconnect();
    }
    else if (cmd == "stop") {
        NetworkManager::disconnect();
CameraManager::stop(); SerialRX::stop(); JoystickManager::stop();
    }
    else {
        Logger::addLog("[ERR] Unknown command: " + cmd +
". Type 'help' for manual.");
    }
    }
    currentCommand = ""; cursorPosition = 0;
}
} else if (asciiCode == 8) { // BACKSPACE
    if (cursorPosition > 0 && !currentCommand.empty()) {
        currentCommand.erase(cursorPosition - 1, 1);
cursorPosition--;
    }
} else if (asciiCode == 22) { // CTRL+V
    std::string pt = getClipboardText();
    currentCommand.insert(cursorPosition, pt); cursorPosition +=
pt.length();
} else if (asciiCode >= 32 && asciiCode <= 126) {
    currentCommand.insert(cursorPosition, 1, (char)asciiCode);
cursorPosition++;
}
}
}
} catch (const std::exception& e) { std::cerr << "Error: " << e.what() <<
std::endl; }
    JoystickManager::stop(); NetworkManager::disconnect(); SerialRX::stop();
CameraManager::stop(); cv::destroyAllWindows();
    return 0;
}

```

Network.cpp

```
#define _WINSOCK_DEPRECATED_NO_WARNINGS
#include <winsock2.h>
#include <ws2tcpip.h>
#include "network.h"
#include "logger.h"
#include "camera.h"
#include <sstream>
#include <chrono>
#include <cmath>
#include <algorithm>

#pragma comment(lib, "ws2_32.lib")

extern std::atomic<uint16_t> rc_channels[16];

namespace {
    std::mutex jpegMutex;
    std::vector<uchar> latest_jpeg;
    bool is_new_jpeg = false;
}

std::atomic<bool> NetworkManager::isRunning(false);
std::atomic<bool> NetworkManager::connected(false);
std::thread NetworkManager::netThread;
ClientRole NetworkManager::currentRole = ClientRole::NONE;
std::string NetworkManager::my_id = "";
std::string NetworkManager::target_id = "";
std::atomic<long long> NetworkManager::current_rtt(0);

std::atomic<int> NetworkManager::current_jpeg_quality(20);
std::map<uint32_t, long long> NetworkManager::send_times;
cv::Mat NetworkManager::pilotFrame;
std::mutex NetworkManager::frameMutex;
std::mutex NetworkManager::timeMutex;

long long NetworkManager::getRTT() { return current_rtt.load(); }

std::string NetworkManager::getMyID() { return my_id; }

struct FrameBuffer {
    uint8_t total_chunks = 0;
    std::map<uint8_t, std::vector<char>> chunks;
};
```

```

long long now_ms() {
    return
std::chrono::duration_cast<std::chrono::milliseconds>(std::chrono::steady_clock::now(
).time_since_epoch()).count();
}

void NetworkManager::connectAsDrone(const std::string& ip, int port) {
    disconnect();
    isRunning = true; connected = false; currentRole = ClientRole::DRONE;
    netThread = std::thread(networkTask, ip, port, currentRole);
}

void NetworkManager::connectAsPilot(const std::string& ip, int port, const
std::string& droneId) {
    disconnect();
    isRunning = true; connected = false; currentRole = ClientRole::TEST_PILOT;
    target_id = droneId;
    netThread = std::thread(networkTask, ip, port, currentRole);
}

void NetworkManager::disconnect() {
    isRunning = false; connected = false; currentRole = ClientRole::NONE;
    if (netThread.joinable()) netThread.join();
    Logger::addLog("[NET] Disconnected.");
}

ClientRole NetworkManager::getRole() { return currentRole; }
bool NetworkManager::isConnected() { return connected.load(); }

bool NetworkManager::getPilotFrame(cv::Mat& outFrame) {
    bool should_decode = false;
    std::vector<uchar> local_jpeg;

    {
        std::lock_guard<std::mutex> lock(jpegMutex);
        if (is_new_jpeg) {
            local_jpeg = latest_jpeg;
            is_new_jpeg = false;
            should_decode = true;
        }
    }

    if (should_decode && !local_jpeg.empty()) {
        cv::Mat decoded = cv::imdecode(local_jpeg, cv::IMREAD_COLOR);
        if (!decoded.empty()) {
            std::lock_guard<std::mutex> lock(frameMutex);
            pilotFrame = decoded;
        } else {

```

```

        Logger::addLog("[SYS_WARN] Deferred frame decode failed.");
    }
}

std::lock_guard<std::mutex> lock(frameMutex);
if (!pilotFrame.empty()) {
    outFrame = pilotFrame.clone();
    return true;
}
return false;
}

void NetworkManager::parseControlPacket(const std::string& data) {
    if (data.find("CTRL:") == std::string::npos) return;
    size_t pos = data.find("CTRL:");
    std::string valuesStr = data.substr(pos + 5);
    std::stringstream ss(valuesStr);
    std::string item;
    int index = 0;
    while (std::getline(ss, item, ',') && index < 16) {
        try { rc_channels[index].store(std::stoi(item)); } catch (...) {}
        index++;
    }
}

std::string NetworkManager::buildControlPacket() {
    std::string packet = "CTRL:";
    for (int i = 0; i < 16; i++) {
        packet += std::to_string(rc_channels[i].load());
        if (i < 15) packet += ",";
    }
    return packet;
}

void NetworkManager::networkTask(std::string ip, int port, ClientRole role) {
    WSADATA wsaData;
    int wsaRes = WSASStartup(MAKEWORD(2, 2), &wsaData);
    if (wsaRes != 0) {
        Logger::addLog("[NET_ERR] WSASStartup failed. Code: " +
std::to_string(wsaRes));
        isRunning = false;
        return;
    }
    Logger::addLog("[SYS] WSASStartup successful.");

    // --- TCP AUTH ---
    SOCKET tcpSock = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
    if (tcpSock == INVALID_SOCKET) {

```

```

        Logger::addLog("[NET_ERR] TCP socket creation failed. Code: " +
std::to_string(WSAGetLastError()));
        WSACleanup();
        isRunning = false;
        return;
    }
    Logger::addLog("[SYS] TCP socket created.");

    sockaddr_in serverAddr;
    serverAddr.sin_family = AF_INET;
    serverAddr.sin_port = htons(port);

    if (inet_pton(AF_INET, ip.c_str(), &serverAddr.sin_addr) <= 0) {
        Logger::addLog("[NET_ERR] Invalid TCP IP address format: " + ip);
        closesocket(tcpSock); WSACleanup(); isRunning = false; return;
    }

    if (connect(tcpSock, (SOCKADDR*)&serverAddr, sizeof(serverAddr)) == SOCKET_ERROR)
    {
        Logger::addLog("[NET_ERR] TCP Server unreachable. Code: " +
std::to_string(WSAGetLastError()));
        closesocket(tcpSock); WSACleanup(); isRunning = false; return;
    }
    Logger::addLog("[SYS] TCP connection established.");

    char tcpBuf[1024];
    int r = recv(tcpSock, tcpBuf, 1023, 0);
    if (r <= 0) {
        Logger::addLog("[NET_ERR] Failed to receive ID packet or server closed
connection.");
        closesocket(tcpSock); WSACleanup(); isRunning = false; return;
    }
    tcpBuf[r] = '\0';
    Logger::addLog("[SYS] TCP Initial packet received.");

    std::string initStr(tcpBuf);
    size_t colonPos = initStr.find(':');
    if (colonPos == std::string::npos) {
        Logger::addLog("[NET_ERR] Malformed ID packet: " + initStr);
        closesocket(tcpSock); WSACleanup(); isRunning = false; return;
    }

    my_id = initStr.substr(colonPos + 1);
    my_id.erase(std::remove(my_id.begin(), my_id.end(), '\n'), my_id.end());
    my_id.erase(std::remove(my_id.begin(), my_id.end(), '\r'), my_id.end());
    Logger::addLog("[SYS] Client ID assigned: " + my_id);

    if (role == ClientRole::TEST_PILOT) {
        std::string watchCmd = "WATCH_VIDEO " + target_id + "\n";

```

```

        if (send(tcpSock, watchCmd.c_str(), watchCmd.length(), 0) == SOCKET_ERROR) {
            Logger::addLog("[NET_ERR] Failed to send WATCH_VIDEO command.");
        } else {
            Logger::addLog("[SYS] WATCH_VIDEO command sent for target: " +
target_id);
        }
    }

    // --- UDP SETUP ---
    SOCKET udpSock = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);
    if (udpSock == INVALID_SOCKET) {
        Logger::addLog("[NET_ERR] UDP socket creation failed. Code: " +
std::to_string(WSAGetLastError()));
        closesocket(tcpSock); WSACleanup(); isRunning = false; return;
    }
    Logger::addLog("[SYS] UDP socket created.");

    u_long mode = 1;
    if (ioctlsocket(udpSock, FIONBIO, &mode) == SOCKET_ERROR) {
        Logger::addLog("[NET_WARN] Failed to set non-blocking mode on UDP socket.");
    }

    int rBuf = 1024 * 1024;
    if (setsockopt(udpSock, SOL_SOCKET, SO_RCVBUF, (char*)&rBuf, sizeof(rBuf)) ==
SOCKET_ERROR) {
        Logger::addLog("[NET_WARN] Failed to set SO_RCVBUF size.");
    }

#define SIO_UDP_CONNRESET _WSAIOW(IOC_VENDOR, 12)
    BOOL bNewBehavior = FALSE;
    DWORD dwBytesReturned = 0;
    if (WSAIoctl(udpSock, SIO_UDP_CONNRESET, &bNewBehavior, sizeof(bNewBehavior),
NULL, 0, &dwBytesReturned, NULL, NULL) == SOCKET_ERROR) {
        Logger::addLog("[NET_WARN] WSAIoctl SIO_UDP_CONNRESET failed.");
    }

    sockaddr_in udpAddr;
    udpAddr.sin_family = AF_INET;
    udpAddr.sin_port = htons(port);
    if (inet_pton(AF_INET, ip.c_str(), &udpAddr.sin_addr) <= 0) {
        Logger::addLog("[NET_ERR] Invalid UDP IP address format: " + ip);
        closesocket(udpSock); closesocket(tcpSock); WSACleanup(); isRunning = false;
return;
    }

    std::string pingStr = my_id + "PING";
    if (sendto(udpSock, pingStr.c_str(), pingStr.length(), 0, (SOCKADDR*)&udpAddr,
sizeof(udpAddr)) == SOCKET_ERROR) {
        Logger::addLog("[NET_ERR] Initial UDP PING failed to send.");
    }

```

```

    } else {
        Logger::addLog("[SYS] Initial UDP PING sent.");
    }

    connected = true;
    Logger::addLog("[SYS] MTU-SAFE Link Active. ID: " + my_id);

    auto next_ctrl_time = std::chrono::steady_clock::now();
    auto next_video_time = std::chrono::steady_clock::now();
    long long last_keepalive = now_ms();

    uint32_t f_id = 0;
    uint32_t highest_f = 0;
    std::map<uint32_t, FrameBuffer> assembly;
    char buf[2048];
    const int CHUNK_SIZE = 1300;

    while (isRunning) {
        // --- ПРИЙОМ (UDP) ---
        while (true) {
            int n = recvfrom(udpSock, buf, sizeof(buf), 0, NULL, NULL);
            if (n <= 0) {
                int err = WSAGetLastError();
                if (err != WSAEWOULDBLOCK && err != 0) {
                    Logger::addLog("[NET_ERR] UDP recvfrom error: " +
std::to_string(err));
                }
                break;
            }

            std::string msg(buf, n);

            try {
                // 1. ДРОН: Отримує ACK, пахує RTT
                if (msg.find("ACK:") != std::string::npos && role ==
ClientRole::DRONE) {
                    uint32_t ack_f_id = std::stoul(msg.substr(msg.find("ACK:") + 4));
                    long long sent_time = 0;
                    {
                        std::lock_guard<std::mutex> lock(timeMutex);
                        if (send_times.count(ack_f_id)) { sent_time =
send_times[ack_f_id]; send_times.erase(ack_f_id); }
                    }
                    if (sent_time > 0) {
                        long long rtt = now_ms() - sent_time;
                        current_rtt.store(rtt);

                        std::string rtt_sync = my_id + "RTT_SYNC:" +
std::to_string(rtt);

```

```

        sendto(udpSock, rtt_sync.c_str(), rtt_sync.length(), 0,
(SOCKADDR*)&udpAddr, sizeof(udpAddr));

        int qual = current_jpeg_quality.load();
        if (rtt < 50) {
            qual += 1; if (qual > 50) qual = 50;
        } else if (rtt <= 130) {
            if (qual > 60) qual -= 1;
            else if (qual < 50) qual += 1;
        } else {
            qual -= 5; if (qual < 15) qual = 15;
        }
        current_jpeg_quality.store(qual);
    }
}
// 2. ДРОН: Отримує команди керування
else if (msg.find("CTRL:") != std::string::npos && role ==
ClientRole::DRONE) {
    parseControlPacket(msg);
}
// 3. ПІЛОТ: Отримує готовий RTT
else if (msg.find("RTT_SYNC:") != std::string::npos && role ==
ClientRole::TEST_PILOT) {
    long long synced_rtt =
std::stoll(msg.substr(msg.find("RTT_SYNC:") + 9));
    current_rtt.store(synced_rtt);
}
// 4. ПІЛОТ: Отримує відео чанки
else if (n > 6 && role == ClientRole::TEST_PILOT) {
    uint32_t nf; memcpy(&nf, buf, 4);
    uint32_t recv_f_id = ntohl(nf);
    uint8_t t_ch = buf[4];
    uint8_t ch_idx = buf[5];

    if (recv_f_id < highest_f && (highest_f - recv_f_id) > 1000) {
        highest_f = recv_f_id;
        assembly.clear();
    }

    if (recv_f_id >= highest_f) {
        highest_f = recv_f_id;
        assembly[recv_f_id].total_chunks = t_ch;
        assembly[recv_f_id].chunks[ch_idx] = std::vector<char>(buf +
6, buf + n);

        if (assembly[recv_f_id].chunks.size() == t_ch) {
            size_t total_size = 0;
            for (int i = 0; i < t_ch; i++) {
                total_size += assembly[recv_f_id].chunks[i].size();
            }
        }
    }
}

```

```

    }

    std::vector<uchar> jpg;
    jpg.reserve(total_size);
    for (int i = 0; i < t_ch; i++) {
        jpg.insert(jpg.end(),
assembly[recv_f_id].chunks[i].begin(), assembly[recv_f_id].chunks[i].end());
    }

    {
        std::lock_guard<std::mutex> lock(jpegMutex);
        latest_jpeg = std::move(jpg);
        is_new_jpeg = true;
    }
    assembly.erase(recv_f_id);

    std::string ack = my_id + "ACK:" +
std::to_string(recv_f_id);
    sendto(udpSock, ack.c_str(), ack.length(), 0,
(SOCKADDR*)&udpAddr, sizeof(udpAddr));
    }
    }
} catch (const std::exception& e) {}
}

if (role == ClientRole::TEST_PILOT) {
    for (auto it = assembly.begin(); it != assembly.end(); ) {
        if (highest_f > it->first && (highest_f - it->first) > 30) {
            it = assembly.erase(it);
        } else {
            ++it;
        }
    }
}

auto now = std::chrono::steady_clock::now();
if (role == ClientRole::TEST_PILOT) {
    if (now >= next_ctrl_time) {
        std::string ctrl = my_id + buildControlPacket();
        sendto(udpSock, ctrl.c_str(), ctrl.length(), 0, (SOCKADDR*)&udpAddr,
sizeof(udpAddr));
        next_ctrl_time = now + std::chrono::milliseconds(20);
    }

    if (now_ms() - last_keepalive > 1000) {
        std::string ka = my_id + "PING";
        sendto(udpSock, ka.c_str(), ka.length(), 0, (SOCKADDR*)&udpAddr,
sizeof(udpAddr));
    }
}

```

```

        last_keepalive = now_ms();
    }
} else if (role == ClientRole::DRONE) {
    if (now >= next_video_time) {
        cv::Mat frame;
        if (CameraManager::getFrame(frame)) {
            f_id++;
            std::vector<uchar> jpgBuf;
            std::vector<int> p = {cv::IMWRITE_JPEG_QUALITY,
current_jpeg_quality.load()};

            if (!cv::imencode(".jpg", frame, jpgBuf, p)) {
                Logger::addLog("[SYS_ERR] JPEG encoding failed.");
            } else {
                { std::lock_guard<std::mutex> lock(timeMutex);
send_times[f_id] = now_ms(); }

                uint8_t total_ch = std::ceil((float)jpgBuf.size() /
CHUNK_SIZE);

                for (uint8_t i = 0; i < total_ch; i++) {
                    std::vector<char> pkt;
                    pkt.reserve(CHUNK_SIZE + 32);
                    pkt.insert(pkt.end(), my_id.begin(), my_id.end());
                    uint32_t nf = htonl(f_id); pkt.insert(pkt.end(),
(char*)&nf, (char*)&nf + 4);
                    pkt.push_back(total_ch); pkt.push_back(i);

                    int off = i * CHUNK_SIZE;
                    int len = std::min((int)CHUNK_SIZE, (int)jpgBuf.size() -
off);

                    pkt.insert(pkt.end(), jpgBuf.begin() + off,
jpgBuf.begin() + off + len);

                    sendto(udpSock, pkt.data(), pkt.size(), 0,
(SOCKADDR*)&udpAddr, sizeof(udpAddr));
                }
            }
        }
        next_video_time = now + std::chrono::milliseconds(33);
    }
}
std::this_thread::sleep_for(std::chrono::milliseconds(1));
}

Logger::addLog("[SYS] Shutting down network connections.");
closesocket(udpSock);
closesocket(tcpSock);
WSACleanup();
}

```

Serial_rx.cpp

```
#include "serial_rx.h"
#include "logger.h"
#include <windows.h>
#include <sstream>
#include <iomanip>
#include <chrono>

// Доступ до глобального масиву каналів з main.cpp
extern std::atomic<uint16_t> rc_channels[16];

std::atomic<bool> SerialRX::isRunning(false);
std::atomic<bool> SerialRX::telemetryActive(false);
std::thread SerialRX::serialThread;

uint8_t SerialRX::crc8_dvb_s2(uint8_t crc, unsigned char a) {
    crc ^= a;
    for (int i = 0; i < 8; ++i) {
        if (crc & 0x80) crc = (crc << 1) ^ 0xD5;
        else crc = crc << 1;
    }
    return crc;
}

void SerialRX::pack_crsf_channels(uint16_t* channels, uint8_t* payload) {
    payload[0] = (channels[0] & 0x07FF);
    payload[1] = ((channels[0] & 0x07FF) >> 8) | ((channels[1] & 0x07FF) << 3);
    payload[2] = ((channels[1] & 0x07FF) >> 5) | ((channels[2] & 0x07FF) << 6);
    payload[3] = ((channels[2] & 0x07FF) >> 2);
    payload[4] = ((channels[2] & 0x07FF) >> 10) | ((channels[3] & 0x07FF) << 1);
    payload[5] = ((channels[3] & 0x07FF) >> 7) | ((channels[4] & 0x07FF) << 4);
    payload[6] = ((channels[4] & 0x07FF) >> 4) | ((channels[5] & 0x07FF) << 7);
    payload[7] = ((channels[5] & 0x07FF) >> 1);
    payload[8] = ((channels[5] & 0x07FF) >> 9) | ((channels[6] & 0x07FF) << 2);
    payload[9] = ((channels[6] & 0x07FF) >> 6) | ((channels[7] & 0x07FF) << 5);
    payload[10] = ((channels[7] & 0x07FF) >> 3);
    payload[11] = ((channels[7] & 0x07FF) >> 11) | ((channels[8] & 0x07FF) << 0);
    payload[12] = ((channels[8] & 0x07FF) >> 8) | ((channels[9] & 0x07FF) << 3);
    payload[13] = ((channels[9] & 0x07FF) >> 5) | ((channels[10] & 0x07FF) << 6);
    payload[14] = ((channels[10] & 0x07FF) >> 2);
    payload[15] = ((channels[10] & 0x07FF) >> 10) | ((channels[11] & 0x07FF) << 1);
    payload[16] = ((channels[11] & 0x07FF) >> 7) | ((channels[12] & 0x07FF) << 4);
```

```

    payload[17] = ((channels[12] & 0x07FF) >> 4) | ((channels[13] & 0x07FF) << 7);
    payload[18] = ((channels[13] & 0x07FF) >> 1);
    payload[19] = ((channels[13] & 0x07FF) >> 9) | ((channels[14] & 0x07FF) << 2);
    payload[20] = ((channels[14] & 0x07FF) >> 6) | ((channels[15] & 0x07FF) << 5);
    payload[21] = ((channels[15] & 0x07FF) >> 3);
}

void SerialRX::start(const std::string& portName, int baudRate) {
    if (isRunning) return;
    isRunning = true;
    telemetryActive = false;
    serialThread = std::thread(ioThread, portName, baudRate);
}

void SerialRX::stop() {
    isRunning = false;
    telemetryActive = false;
    if (serialThread.joinable()) {
        serialThread.join();
    }
}

bool SerialRX::isTelemetryActive() {
    return telemetryActive.load();
}

void SerialRX::ioThread(std::string portName, int baudRate) {
    std::string winPortName = "\\.\\" + portName;

    HANDLE hSerial = CreateFileA(winPortName.c_str(), GENERIC_READ | GENERIC_WRITE,
0, 0, OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, 0);

    if (hSerial == INVALID_HANDLE_VALUE) {
        Logger::addLog("[COM_ERR] Failed to open " + portName);
        isRunning = false;
        return;
    }

    DCB dcbSerialParams = {0};
    dcbSerialParams.DCBlength = sizeof(dcbSerialParams);

    if (GetCommState(hSerial, &dcbSerialParams)) {
        dcbSerialParams.BaudRate = baudRate;
        dcbSerialParams.ByteSize = 8;
        dcbSerialParams.StopBits = ONESTOPBIT;
        dcbSerialParams.Parity = NOPARITY;
        SetCommState(hSerial, &dcbSerialParams);
    }
}

```

```

COMMTIMEOUTS timeouts = {0};
timeouts.ReadIntervalTimeout = MAXDWORD;
timeouts.ReadTotalTimeoutConstant = 0;
timeouts.ReadTotalTimeoutMultiplier = 0;
SetCommTimeouts(hSerial, &timeouts);

Logger::addLog("[COM] Connected to " + portName + " at " +
std::to_string(baudRate));

char buffer[256];
DWORD bytesRead;
DWORD bytesWritten;

auto next_tx_time = std::chrono::steady_clock::now();
const auto TX_INTERVAL = std::chrono::milliseconds(20);

// Змінні для парсера протоколу CRSF
uint8_t rx_buf[256];
int rx_state = 0;
int rx_expected_len = 0;
int rx_ptr = 0;

while (isRunning) {
    // --- 1. ФАЗА ЧИТАННЯ ТА ВАЛІДАЦІЇ ТЕЛЕМЕТРІЇ ---
    if (ReadFile(hSerial, buffer, sizeof(buffer), &bytesRead, NULL) && bytesRead
> 0) {

        for (DWORD i = 0; i < bytesRead; ++i) {
            uint8_t b = (uint8_t)buffer[i];

            if (rx_state == 0) {
                // Шукаємо байт синхронізації
                if (b == 0xC8 || b == 0xEA || b == 0xEE) {
                    rx_buf[0] = b;
                    rx_state = 1;
                }
            } else if (rx_state == 1) {
                if (b > 1 && b <= 64) {
                    rx_buf[1] = b;
                    rx_expected_len = b;
                    rx_ptr = 2;
                    rx_state = 2;
                } else {
                    rx_state = 0;
                }
            } else if (rx_state == 2) {
                rx_buf[rx_ptr++] = b;

                if (rx_ptr == rx_expected_len + 2) {

```

```

        uint8_t crc = 0;
        for (int j = 2; j < rx_expected_len + 1; ++j) {
            crc = crc8_dvb_s2(crc, rx_buf[j]);
        }

        // Якщо CRC правильний
        if (crc == rx_buf[rx_expected_len + 1]) {
            uint8_t packet_type = rx_buf[2];

            // 0x08 - CRSF_FRAMETYPE_BATTERY_SENSOR
            if (packet_type == 0x08) {
                if (!telemetryActive) {
                    telemetryActive = true;
                    Logger::addLog("[SYS] Battery telemetry received.
Drone connected!");
                }
            }
        }

        rx_state = 0;
    }
}

// --- 2. ФАЗА ЗАПИСУ (КЕРУВАННЯ CRSF) ---
auto now = std::chrono::steady_clock::now();
if (now >= next_tx_time) {
    uint16_t current_channels[16];
    for (int i = 0; i < 16; i++) {
        current_channels[i] = rc_channels[i].load();
    }

    uint8_t crsf_packet[26];
    crsf_packet[0] = 0xC8;
    crsf_packet[1] = 24;
    crsf_packet[2] = 0x16; // Тип пакета: RC Channels Packed

    uint8_t payload[22];
    pack_crsf_channels(current_channels, payload);
    memcpy(&crsf_packet[3], payload, 22);

    uint8_t crc = 0;
    crc = crc8_dvb_s2(crc, crsf_packet[2]);
    for (int i = 0; i < 22; i++) {
        crc = crc8_dvb_s2(crc, payload[i]);
    }
    crsf_packet[25] = crc;

```

```

        WriteFile(hSerial, crsf_packet, sizeof(crsf_packet), &bytesWritten,
NULL);

        next_tx_time = now + TX_INTERVAL;
    }

    std::this_thread::sleep_for(std::chrono::milliseconds(1));
}

CloseHandle(hSerial);
Logger::addLog("[COM] Port closed.");
}

```

Camera.cpp

```

#include "camera.h"
#include "logger.h"
#include <windows.h>
#include <dshow.h>

#pragma comment(lib, "strmiids.lib")
#pragma comment(lib, "ole32.lib")
#pragma comment(lib, "oleaut32.lib")

std::atomic<bool> CameraManager::isRunning(false);
cv::Mat CameraManager::sharedFrame;
std::mutex CameraManager::frameMutex;
std::thread CameraManager::camThread;

std::vector<std::string> CameraManager::listCameras() {
    std::vector<std::string> cameras;
    HRESULT hr = CoInitializeEx(NULL, COINIT_MULTITHREADED);
    bool coInitCalled = SUCCEEDED(hr);

    ICreateDevEnum* pDevEnum = NULL;
    hr = CoCreateInstance(CLSID_SystemDeviceEnum, NULL, CLSCTX_INPROC_SERVER,
IID_PPV_ARGS(&pDevEnum));
    if (SUCCEEDED(hr)) {
        IEnumMoniker* pEnum = NULL;
        hr = pDevEnum->CreateClassEnumerator(CLSID_VideoInputDeviceCategory, &pEnum,
0);
        if (hr == S_OK) {
            IMoniker* pMoniker = NULL;
            int index = 0;
            while (pEnum->Next(1, &pMoniker, NULL) == S_OK) {
                IPropertyBag* pPropBag;

```

```

        hr = pMoniker->BindToStorage(0, 0, IID_PPV_ARGS(&pPropBag));
        if (SUCCEEDED(hr)) {
            VARIANT varName;
            VariantInit(&varName);
            hr = pPropBag->Read(L"FriendlyName", &varName, 0);
            if (SUCCEEDED(hr)) {
                int len = WideCharToMultiByte(CP_UTF8, 0, varName.bstrVal, -
1, NULL, 0, NULL, NULL);
                std::string name(len - 1, '\\0');
                WideCharToMultiByte(CP_UTF8, 0, varName.bstrVal, -1,
&name[0], len, NULL, NULL);
                cameras.push_back "[" + std::to_string(index) + "]" + name);
                VariantClear(&varName);
            }
            pPropBag->Release();
        }
        pMoniker->Release();
        index++;
    }
    pEnum->Release();
}
pDevEnum->Release();
}
if (coInitCalled) CoUninitialize();

if (cameras.empty()) {
    cameras.push_back("[SYS] No video devices found.");
}
return cameras;
}

void CameraManager::start(int cameraId) {
    if (isRunning) return;
    isRunning = true;
    camThread = std::thread(captureThread, cameraId);
}

void CameraManager::stop() {
    isRunning = false;
    if (camThread.joinable()) {
        camThread.join();
    }
}

bool CameraManager::getFrame(cv::Mat& outFrame) {
    std::lock_guard<std::mutex> lock(frameMutex);
    if (!sharedFrame.empty()) {
        outFrame = sharedFrame.clone();
        return true;
    }
}

```

```

    }
    return false;
}

void CameraManager::captureThread(int cameraId) {
    cv::VideoCapture cap(cameraId, cv::CAP_DSHOW);
    cap.set(cv::CAP_PROP_FRAME_WIDTH, 640);
    cap.set(cv::CAP_PROP_FRAME_HEIGHT, 480);
    cap.set(cv::CAP_PROP_FPS, 30);
    cap.set(cv::CAP_PROP_BUFFERSIZE, 1);

    if (!cap.isOpened()) {
        Logger::addLog("[ERROR] Cannot open camera ID: " + std::to_string(cameraId));
        isRunning = false;
        return;
    }

    Logger::addLog("[VIDEO] Camera ID " + std::to_string(cameraId) + "
initialized.");
    cv::Mat localFrame, resizedFrame;

    while (isRunning) {
        cap.grab();
        cap.retrieve(localFrame);

        if (!localFrame.empty()) {
            cv::resize(localFrame, resizedFrame, cv::Size(640, 480));
            cv::GaussianBlur(resizedFrame, resizedFrame, cv::Size(3, 3), 0);
            std::lock_guard<std::mutex> lock(frameMutex);
            sharedFrame = resizedFrame.clone();
        }
    }
    cap.release();
    Logger::addLog("[VIDEO] Camera thread stopped.");

    std::lock_guard<std::mutex> lock(frameMutex);
    sharedFrame = cv::Mat();
}

bool CameraManager::isCameraActive() {
    return isRunning.load();
}

```

ДОДАТОК Б

Програмне забезпечення моделювання віддаленого контролю

Презентація

УКР.НТУУ «КПІ ім. Ігоря Сікорського» НН ІАТЕ ТВ-22

Аркушів 15

Київ – 2026



Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»

Навчально-науковий інститут атомної та теплової енергетики

Кафедра інженерії програмного забезпечення в енергетиці

ПРОГРАМНЕ ЗАБЕЗПЕЧЕННЯ МОДЕЛЮВАННЯ ВІДДАЛЕНОГО КОНТРОЛЮ

виконав: студент групи ТВ-22 Сівоздрав Артем Миколайович

керівник: ст. викладач Сарибоба Ганна Володимирівна

2026 рік



Національний технічний університет України «Київський політехнічний інститут імені Ігоря Сікорського»

Актуальність



Зростання застосування БПЛА

Безпілотні літальні апарати активно використовуються у військовій сфері, моніторингу, логістиці, агросекторі та наукових дослідженнях.



Потреба у надійному зв'язку

Ефективне керування БПЛА на великих відстанях потребує стабільного каналу зв'язку для передачі керуючих команд і отримання телеметрії та відео.



Обмеження прямого зв'язку

Перешкоди рельєфу, забудови, лісових масивів та значна відстань призводять до втрати сигналу та зниження якості керування.



Актуальне рішення

Використання ретранслятора дозволяє суттєво збільшити дальність і надійність керування БПЛА завдяки проміжній передачі даних на основі стеку протоколів TCP/IP.



Без ретранслятора



З ретранслятором



Застосування стеку протоколів TCP/IP забезпечує стабільну передачу даних, масштабованість та сумісність із сучасними мережевими технологіями.

Мета і завдання

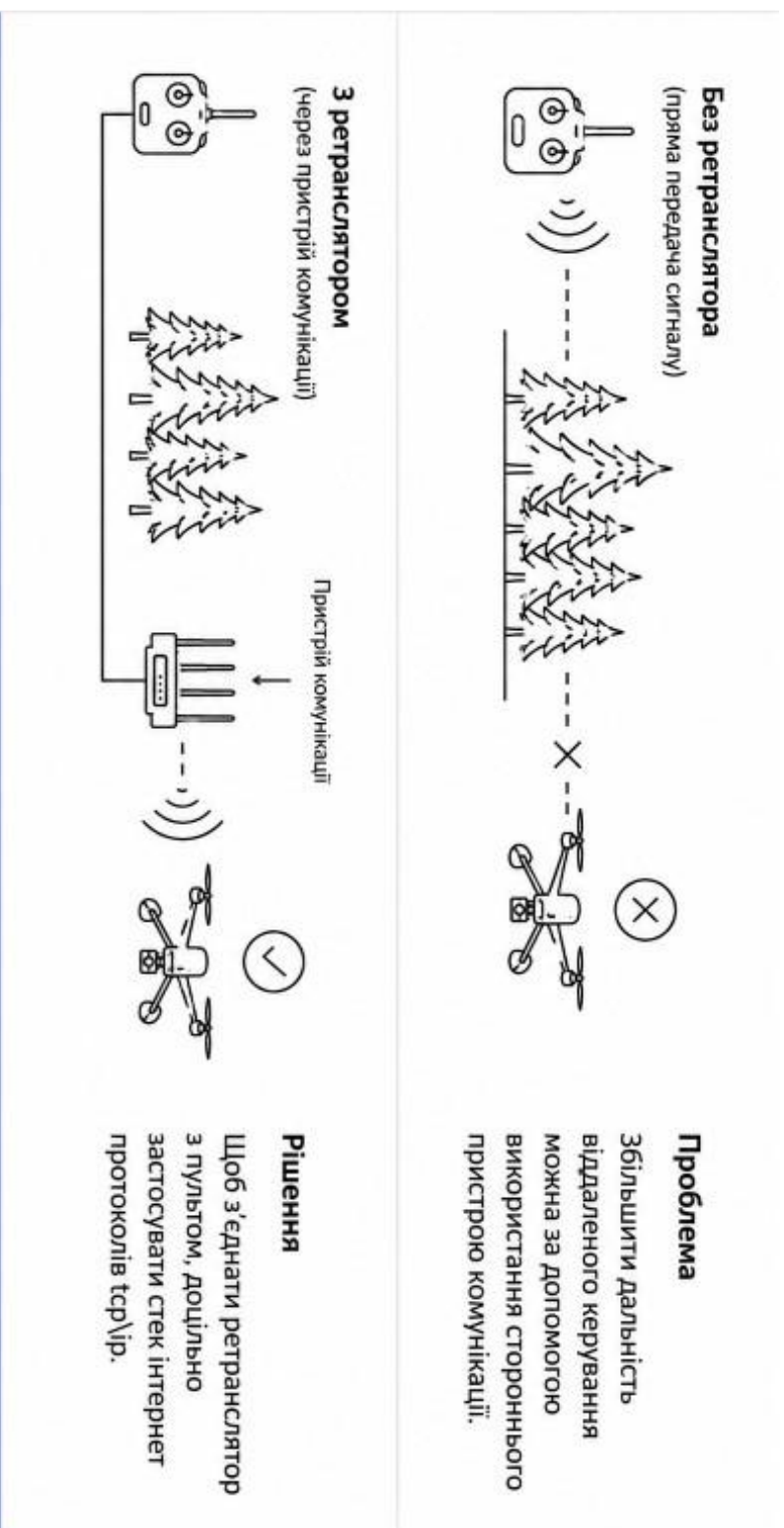
Мета роботи-розробка ПЗ моделювання віддаленого контролю БПЛА

Завдання:

- Аналіз предметної області;
- Визначення функціональних та нефункціональних вимог;
- Створення структури системи віддаленого керування БПЛА типу fpu;
- Реалізація схеми фізичного передавання даних між БПЛА та пристроєм комунікації;
- Розробка системи керування дроном. Серверна частина;
- Розробка ПЗ системи керування дроном. Клієнтська частина;
- Опрацювання етапів обробки відеокадру;
- Конструювання телеметричного кадру csi protocol для керування дроном;
- Тестування ПЗ та аналіз результатів тестування



Аналіз предметної області



Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»

Слайд 4

Функціональні і нефункціональні вимоги

Функціональні вимоги:

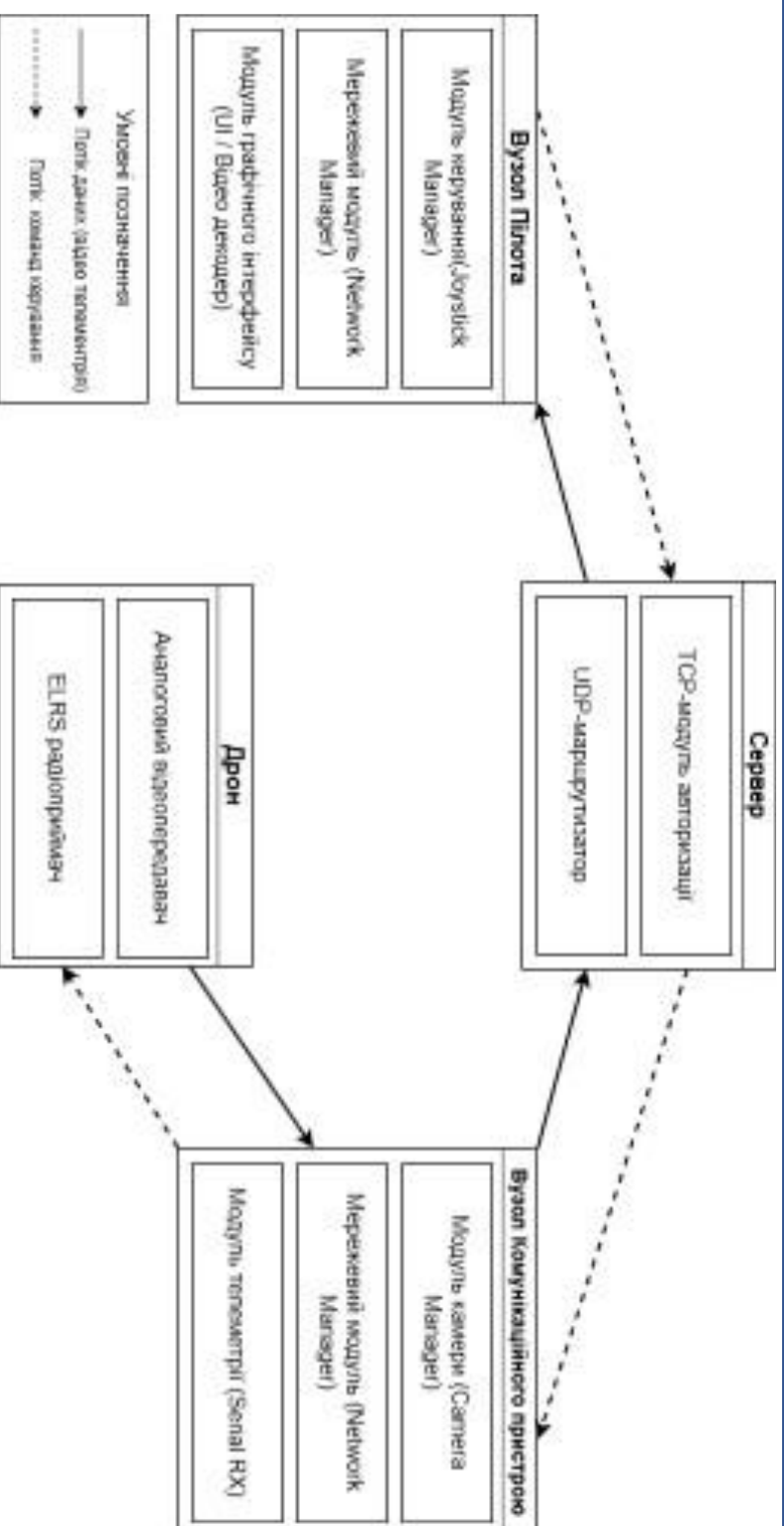
- Керування трьома режимами роботи «Дрон», «пілот» і «тест пілот»;
- Підтримка інтерфейсу командного рядка (CLI) для керування системою;
- Захоплення, стиснення (JPEG) та відправка відео по UDP у режимі «Дрон»;
- Прийом, декодування та відображення відео по UDP у режимі «Пілот»;
- Зчитування та конвертація даних з підключеного джойстика у формат CRSF;
- Формування пакетів керування та прийом телеметрії через UART (Serial порт);
- Ініціалізація TCP-з'єднання для авторизації та обміну ідентифікаторами;
- Маршрутизація даних (відео, команди, телеметрія) у реальному часі через UDP;
- Візуалізація UI: відео, статуси 16 каналів, консоль та системні логи;

Нефункціональні вимоги

- Надійність передачі;
- Низька затримка 40-60с;
- Живучість з'єднання;
- Безпека



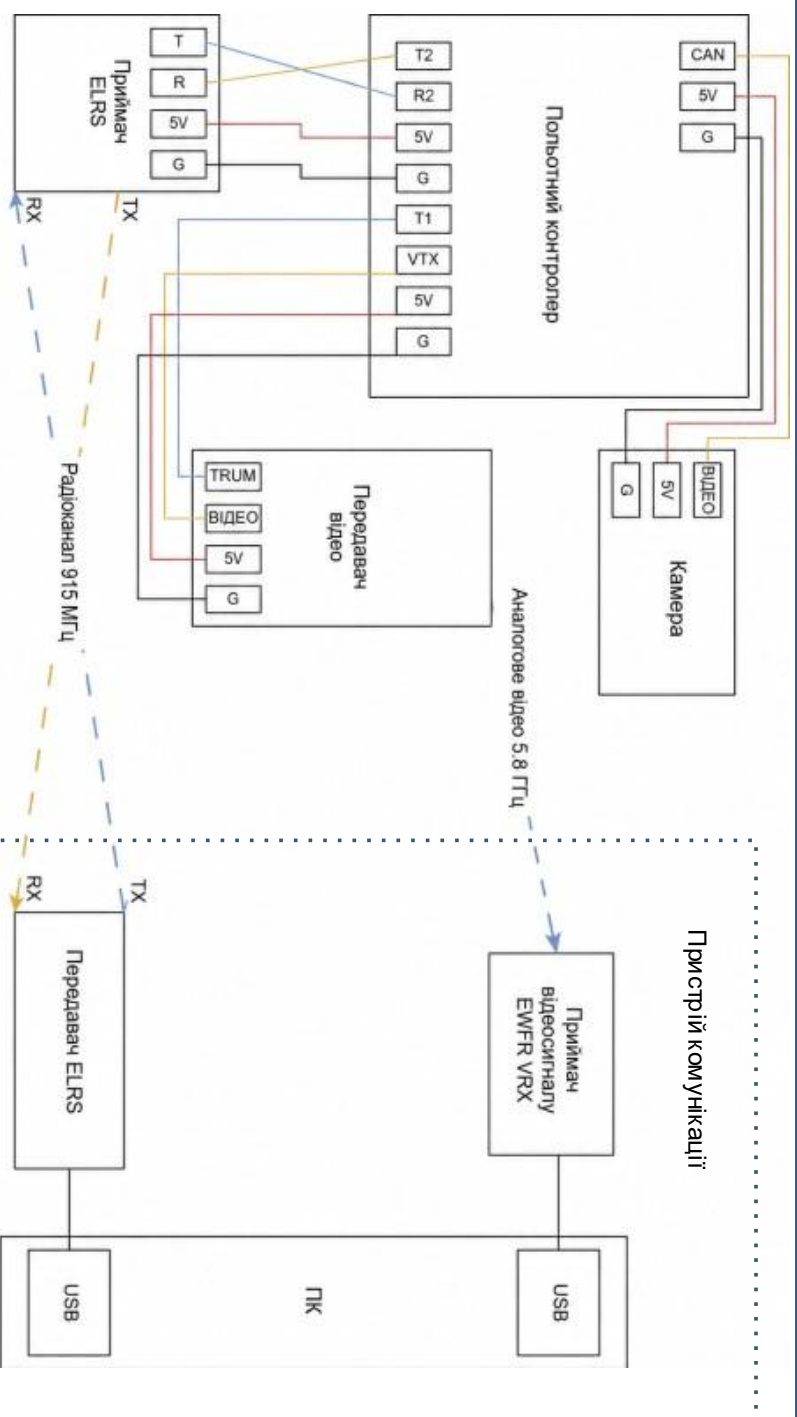
Структура системи віддаленого керування БПЛА типу frv



Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»

Слайд 6

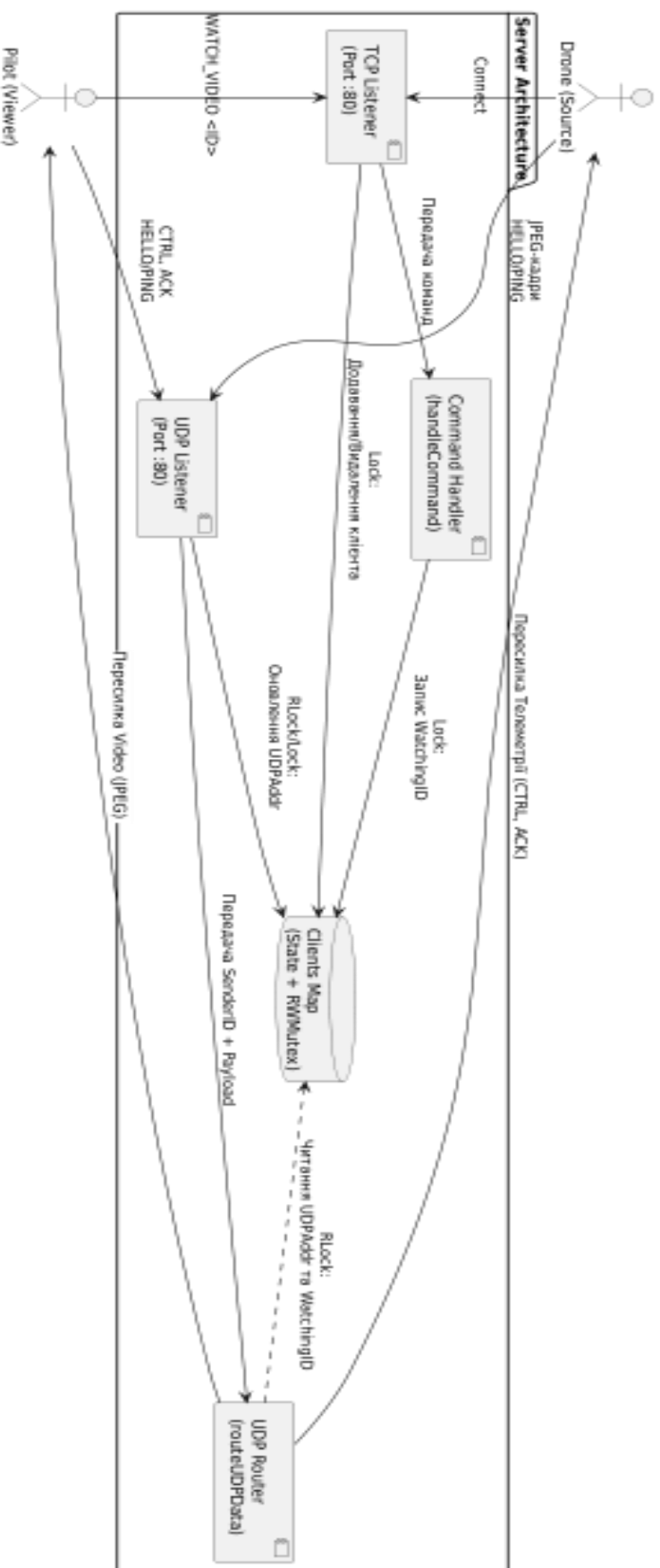
Фізична схема приймання/передачі телеметрії та відео



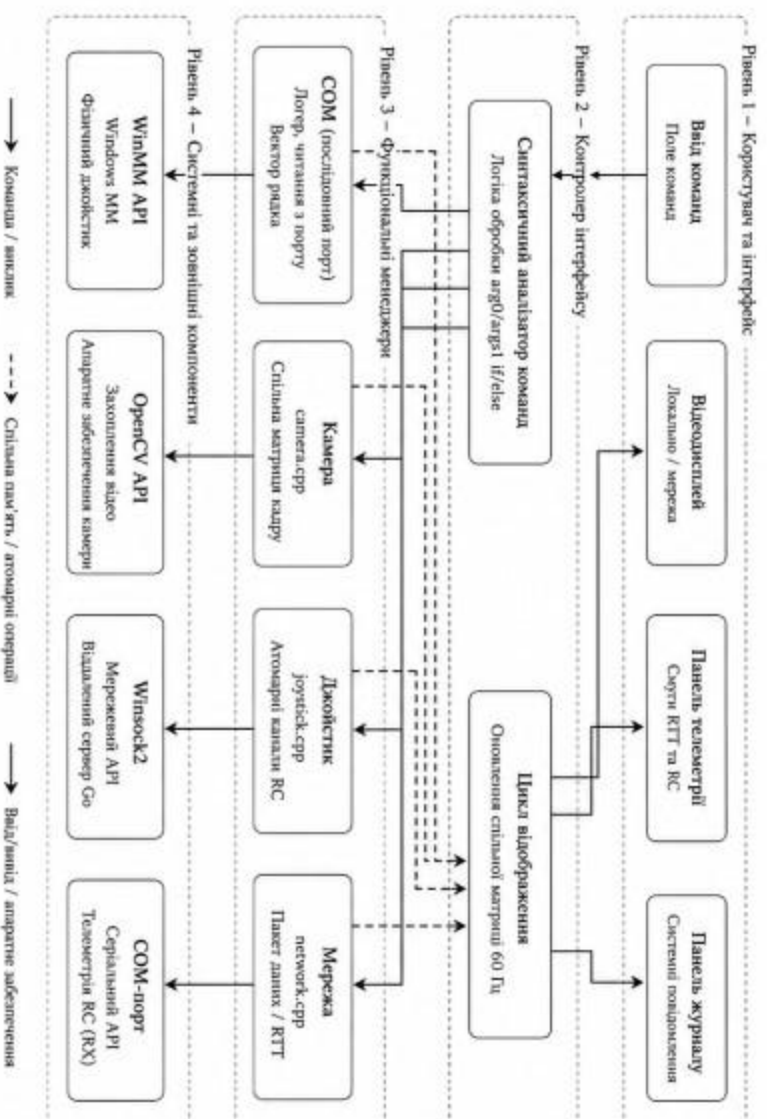
Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»

Слайд 7

UML діаграма компонентів Серверної частини

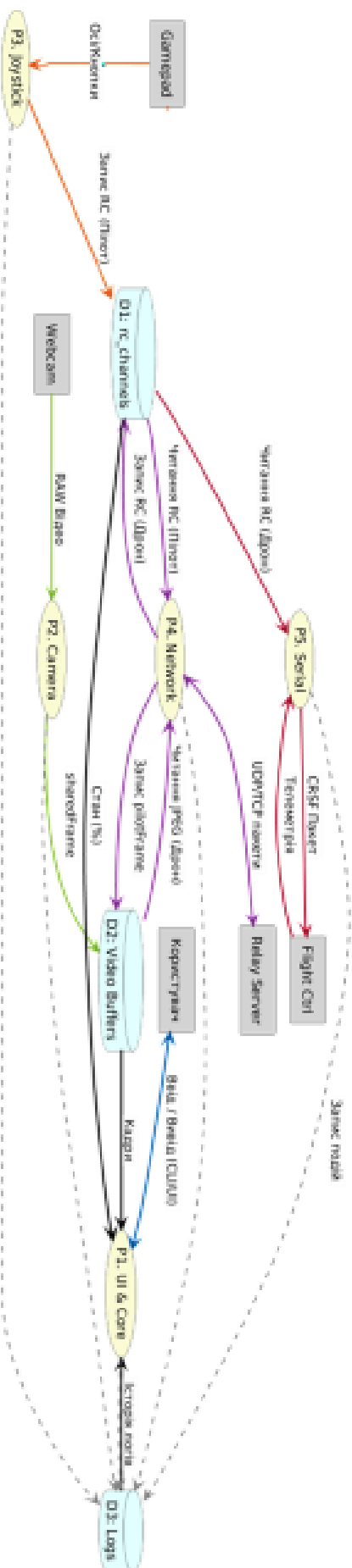


Алгоритм роботи клієнтської частини

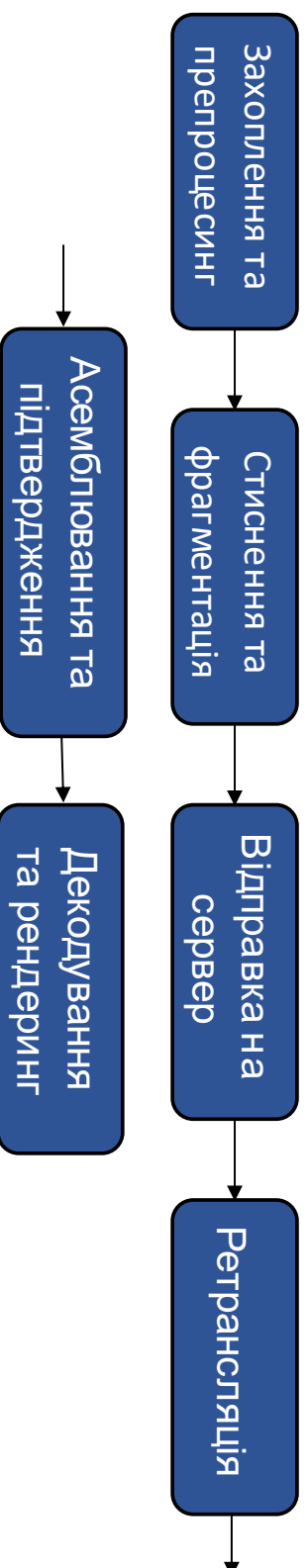
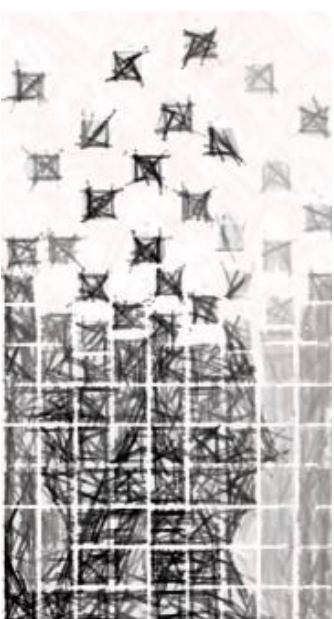


Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»

DFD діаграма потоків даних клієнтської частини



Життєвий цикл відео кадру



Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»

Слайд 11

Структура телеметричного кадру керування



Структура csf рами

Sync byte – байт синхронізації, вказує адресата пакета

Frame Length – вказує розмір пакету без перших двох байтів

Type – ідентифікатор типу даних, від цього байту залежить зміст корисного навантаження

Payload – корисне навантаження, містить основну інформацію пакета

CRC – контрольна сума, запобіжник для перевірки цілісності



Інтерфейс користувача



Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»

Слайд 13

Аналіз результатів тестування

$$T_{\text{заг}} = T_{\text{клієнт}} + T_{\text{мережа1}} + T_{\text{сервер}} + T_{\text{мережа2}} + T_{\text{БПЛА}}$$

де

$T_{\text{клієнт}}$ – час формування та обробки команди на клієнтській стороні;

$T_{\text{мережа1}}$ – затримка передавання даних від оператора до сервера;

$T_{\text{сервер}}$ – час обробки даних на сервері;

$T_{\text{мережа2}}$ – затримка передавання даних від сервера до пристрою комунікації або БПЛА;

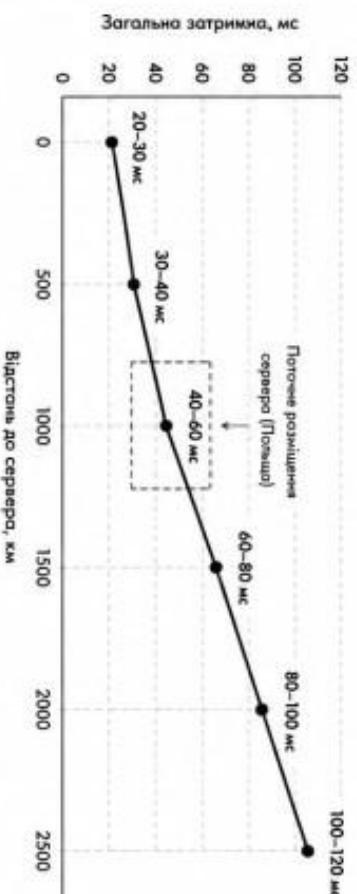
$T_{\text{БПЛА}}$ – час реакції бортової системи БПЛА.

ТИПОВІ ДІАПАЗОНИ ЗАТРИМКИ КОМПОНЕНТІВ (ОРІЄНТОВНО)

Компонент	Діапазон затримки, мс
Клієнтська обробка	5 – 15
Мережа (оператор → сервер)	10 – 30
Обробка на сервері	5 – 15
Мережа (сервер → пристрій/БПЛА)	10 – 30
Реакція бортової системи БПЛА	10 – 20
Загально затримка	40 – 60

У результаті тестування програмного продукту встановлено середню затримку на рівні **40–60 мс**, що є прийнятним показником для дистанційного пілотування БПЛА. Основними чинниками впливу є якість інтернет-з'єднання, віддаленість сервера, маршрутизація даних та швидкість обробки інформації на сервері.

ГРАФІК ЗАТРИМКИ ВІД ВІДСТАНІ СЕРВЕРА ДО ТОЧКИ КЕРУВАННЯ



Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»

Висновки

В ході розробки дипломної роботи було досягнуто мету та виконано поставлені завдання:

- Проведено аналіз предметної області;
- Визначено функціональні та нефункціональні вимоги;
- Створено структуру системи віддаленого керування БПЛА типу frv;
- Реалізовано схеми фізичного передавання даних між БПЛА та пристроєм комунікації;
- Розроблено системи керування дроном. Серверна частина;
- Розроблено ПЗ системи керування дроном. Клієнтська частина;
- Описано етапи обробки відеозапису;
- Сконструйовано телеметричний кадр crsf протоколу для керування дроном;
- Протестовано ПЗ та проведено аналіз результатів тестування

