КИЇВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ ІМЕНІ ТАРАСА ШЕВЧЕНКА

ФАКУЛЬТЕТ РАДІОФІЗИКИ, ЕЛЕКТРОНІКИ ТА КОМП'ЮТЕРНИХ СИСТЕМ

Кафедра радіотехніки та радіоелектронних систем

«На правах рукопису»

Робота допущена до захисту в ЕК
рішенням кафедри радіотехніки та радіоелектронних систем
від _____ 2025 року, протокол № ____.
В. о. завідувача кафедри, кандидат фіз.-мат. наук, доцент
_____ Ігор БЕХ

КВАЛІФІКАЦІЙНА РОБОТА МАГІСТРА
на тему:

«РОЗРОБЛЕННЯ СИСТЕМИ ЗАХИЩЕНОГО ДВОСТОРОННЬОГО ОБМІНУ ДАНИМИ МІЖ ПЕРСОНАЛЬНИМ КОМП'ЮТЕРОМ І МОБІЛЬНИМ ПРИСТРОЄМ»

Виконав:
студент 2-го курсу
денної форми навчання
спеціальності 172 - Електронні комунікації та радіотехніка
ОПП «Захист інформації в телекомунікаціях»

Рябчун Андрій Анатолійович _____

Науковий керівник:
канд. фіз.-мат. наук, асистент
Котов Михайло Миколайович_____

Рецензент:
д. т. н., професор
Хлапонін Юрій Іванович _____

Засвідчую, що у цій магістерській роботі
немає запозичень з праць інших авторів без
відповідних посилань
Студент _____ Рябчун Андрій Анатолійович

Київ-2025

РЕФЕРАТ

Дипломна робота: 70 с., 1 табл. 2 рис., 4 дод. (39с.), 10 джерел.

Розробка захищеного каналу обміну даними між ПК і смартфоном на основі мікроконтролера STM32WB55

Сучасні телекомунікаційні системи вимагають високого рівня захищеності інформації під час її передачі між різними пристроями. Особливо актуальною є задача безпечного обміну даними між персональними комп'ютерами та мобільними пристроями, які все частіше використовуються для доступу до конфіденційної інформації. Тому створення апаратно-програмного комплексу на базі мікроконтролера, здатного забезпечити шифрування та передачу даних у захищеному каналі, є важливим напрямом дослідження у сфері інформаційної безпеки.

У даному проєкті розглядається реалізація захищеного каналу обміну даними з використанням мікроконтролера STM32WB55CGU6, який має інтегрований модуль Bluetooth Low Energy (BLE) та апаратну підтримку криптографічних алгоритмів. Основою безпеки є застосування симетричного шифрування AES-256, що забезпечує високий рівень криптостійкості та швидку роботу завдяки апаратному прискоренню. STM32WB55 приймає зашифровані повідомлення зі смартфона, на якому працює HTML-застосунок (веб-інтерфейс або WebView). Дані передаються через BLE, потім мікроконтролер виконує їх дешифрування і виводить у вигляді тексту в термінал ПК через USB-інтерфейс (CDC). У зворотному напрямку STM32 здійснює шифрування вхідного повідомлення з ПК, передає його на смартфон, де застосунок розшифровує його і відображає користувачу. Для встановлення захищеного каналу реалізується процедура ініціалізації ключа шифрування, яка може виконуватись через попередньо узгоджений секрет або шляхом генерації випадкових чисел апаратним RNG, інтегрованим в STM32WB55. BLE-канал використовується виключно як транспортний рівень, тоді як конфіденційність забезпечується криптографічним протоколом на рівні застосунку.

Проєкт демонструє поєднання апаратної криптографії, бездротового зв’язку та вбудованих систем у одному пристрої. Реалізована система може бути основою для створення захищених месенджерів, IoT-рішень із підвищеною безпекою, корпоративних інструментів шифрування або автономних криптомодулів. Таким чином, розробка підтверджує ефективність підходу, коли безпеку забезпечує апаратний елемент, що не залежить від ОС смартфона чи ПК і забезпечує захист від широкого спектру атак.

ЗМІСТ

ВСТУП

Стрімкий розвиток телекомунікаційних технологій, мобільних пристроїв та вбудованих систем зумовлює зростання обсягів інформації, що передається між різними типами кінцевих пристроїв. У сучасних інформаційних середовищах персональні комп’ютери та смартфони є основними засобами доступу до цифрових ресурсів, обміну повідомленнями та керування іншими пристроями. Водночас передавання даних між такими пристроями дедалі частіше здійснюється з використанням бездротових каналів зв’язку, що значно підвищує вимоги до забезпечення інформаційної безпеки.

Бездротові технології, зокрема Bluetooth і Bluetooth Low Energy (BLE), широко застосовуються для організації обміну даними між персональними та мобільними пристроями завдяки своїй універсальності, енергоефективності та простоті інтеграції. Проте відкритий характер радіоканалу робить такі з’єднання потенційно вразливими до широкого спектра загроз інформаційній безпеці. Серед найбільш поширених загроз можна виділити пасивне перехоплення трафіку, активні атаки типу «людина посередині» (Man-in-the-Middle), підміну пристроїв, несанкціонований доступ до сервісів, а також аналіз структури та обсягів переданих даних.

Навіть за умови використання стандартних механізмів безпеки, передбачених у протоколах Bluetooth, практична реалізація захисту часто залежить від коректності налаштувань, версії протоколу та реалізації на стороні операційної системи. Відомі випадки, коли помилки конфігурації або вразливості в програмному забезпеченні призводили до компрометації ключів шифрування або можливості несанкціонованого доступу до даних. Крім того, користувач або розробник прикладного програмного забезпечення не завжди має повний контроль над внутрішніми механізмами захисту, що реалізуються на рівні операційної системи смартфона чи персонального комп’ютера.

З метою протидії таким загрозам у міжнародній практиці застосовуються криптографічні стандарти та рекомендації, зокрема стандарт Advanced Encryption Standard (AES), визначений у документі FIPS 197, а також стандарти

ISO/IEC серії 18033 та 27000, які регламентують вимоги до алгоритмів шифрування та систем менеджменту інформаційної безпеки. Симетричний алгоритм AES з довжиною ключа 256 біт на сьогодні вважається криптографічно стійким і широко використовується у захищених телекомунікаційних системах, включаючи VPN, TLS та захищені сховища даних.

На ринку та в практиці інформаційної безпеки існує низка готових рішень для захищеного обміну даними між ПК і смартфонами. До них належать програмні месенджери з наскрізним шифруванням, VPN-клієнти, а також вбудовані механізми безпеки мобільних операційних систем. Основною перевагою таких рішень є зручність використання та відсутність необхідності у додатковому апаратному забезпеченні. Проте вони мають і суттєві недоліки: залежність від стороннього програмного забезпечення, закритість реалізацій, обмежений контроль над процесами шифрування та потенційна вразливість у разі компрометації операційної системи або мережевої інфраструктури.

Альтернативним підходом до підвищення рівня захисту інформації є використання апаратно-програмних засобів, у яких криптографічні операції виконуються у виділеному вбудованому пристрої. Такий підхід дозволяє відокремити критичні функції шифрування та керування ключами від загального програмного середовища, зменшити площу атаки та підвищити стійкість системи до програмних загроз. Особливо актуальним це є у випадках, коли персональний комп’ютер або смартфон не можуть вважатися повністю довіреними з точки зору інформаційної безпеки.

Сучасні мікроконтролери з інтегрованими радіомодулями та криптографічними прискорювачами створюють передумови для реалізації таких рішень. Зокрема, мікроконтролери сімейства STM32WB поєднують у собі обчислювальні ядра, радіомодуль Bluetooth Low Energy, генератор істинних випадкових чисел, апаратну підтримку криптографічних алгоритмів та розвинену периферію. Це дозволяє використовувати їх як універсальну платформу для побудови захищених каналів обміну даними без залучення зовнішніх криптографічних модулів.

У даній роботі пропонується підхід, за якого мікроконтролер використовується як криптографічний шлюз між персональним комп’ютером і смартфоном. Персональний комп’ютер взаємодіє з мікроконтролером через провідний інтерфейс USB, а смартфон — через бездротовий інтерфейс Bluetooth Low Energy. Усі дані, що передаються між кінцевими пристроями, проходять через мікроконтролер, де виконуються операції шифрування та дешифрування за алгоритмом AES-256. Така архітектура дозволяє забезпечити конфіденційність переданої інформації навіть у разі перехоплення BLE-трафіку, оскільки передані дані мають вигляд зашифрованих бінарних блоків.

Таким чином, актуальною є задача розроблення захищеного каналу обміну даними між персональним комп’ютером і смартфоном, який базується на апаратно-програмному рішенні з використанням сучасних мікроконтролерів, стандартизованих криптографічних алгоритмів та енергоефективних бездротових технологій. Запропоноване рішення дозволяє поєднати переваги апаратного захисту, гнучкість програмної реалізації та можливість подальшого розвитку системи.

Мета та завдання роботи

Метою магістерської роботи є розроблення апаратно-програмної системи захищеного каналу обміну даними між персональним комп’ютером і смартфоном на основі мікроконтролера STM32WB з використанням симетричного шифрування AES-256 та Bluetooth Low Energy.

Для досягнення поставленої мети у роботі передбачається вирішення таких завдань:

- аналіз загроз інформаційній безпеці в бездротових каналах зв’язку;
- огляд стандартів та сучасних засобів криптографічного захисту;
- розроблення апаратної частини пристрою та друкованої плати;
- реалізація програмного забезпечення мікроконтролера;
- створення клієнтського застосунку для смартфона;
- тестування та оцінка ефективності запропонованого рішення.

Наукова та практична новизна

Наукова новизна роботи полягає у застосуванні апаратно-програмного підходу до організації захищеного каналу обміну даними між ПК і смартфоном із використанням мікроконтролера як криптографічного посередника.

Практична цінність роботи полягає у створенні працездатного прототипу системи, який може бути використаний у телекомунікаційних та вбудованих системах із підвищеними вимогами до захисту інформації.

1.ОГЛЯД ТА ВИБІР КОМПОНЕНТІВ СИСТЕМИ

1.1. Актуальність та постановка задачі

У сучасних телекомунікаційних системах передавання інформації між різнорідними пристроями — персональними комп’ютерами, смартфонами, вбудованими та IoT-пристроями — здійснюється з використанням як провідних, так і бездротових каналів зв’язку. В умовах широкого застосування бездротових технологій питання забезпечення інформаційної безпеки набуває особливої актуальності, оскільки радіоканал є відкритим середовищем, доступним для перехоплення та аналізу.

Обмін даними між персональним комп’ютером і смартфоном широко використовується в прикладних системах керування, обміну службовою інформацією, персональних комунікаціях та мобільних застосунках. Проте стандартні програмні механізми захисту, що реалізуються на рівні операційних систем, не завжди гарантують необхідний рівень безпеки. Це зумовлено складністю програмного середовища, залежністю від сторонніх бібліотек та обмеженим контролем з боку користувача або розробника.

У зв’язку з цим виникає задача побудови захищеного каналу обміну даними, у якому криптографічні операції виконуються у виділеному апаратному середовищі, а сам канал є стійким до типових загроз, таких як пасивне перехоплення, підміна повідомлень та несанкціонований доступ.

1.2. Загрози безпеці в каналах обміну між ПК і смартфоном

Під час обміну даними між персональним комп’ютером і смартфоном можливі такі основні загрози:

- пасивне перехоплення трафіку, що дозволяє отримати конфіденційну інформацію;
- атаки типу “людина посередині” (Man-in-the-Middle);
- підміна кінцевого пристрою або імітація легітимного вузла;
- аналіз трафіку, навіть у разі шифрування даних;
- компрометація програмного середовища ПК або смартфона.

Особливо вразливими є бездротові канали, зокрема Bluetooth Low Energy, де зловмисник має фізичну можливість перебувати в зоні дії радіоканалу та здійснювати атаки без прямого доступу до пристроїв.

1.3. Огляд існуючих підходів до захищеного обміну даними

Існуючі підходи до захисту обміну даними між ПК і смартфоном умовно можна поділити на три групи:

Програмні рішення

До цієї групи належать месенджери з наскрізним шифруванням, VPN-клієнти та мобільні застосунки з вбудованими криптографічними механізмами. Основною перевагою таких рішень є зручність використання та відсутність додаткового апаратного забезпечення. Проте вони мають низку недоліків:

- залежність від операційної системи;
- складність перевірки реалізації криптографії;
- потенційна вразливість у разі зараження пристрою шкідливим ПЗ.

Апаратні криптографічні модулі

Апаратні токени та HSM-модулі забезпечують високий рівень захисту ключової інформації, проте характеризуються високою вартістю, обмеженою гнучкістю та складністю інтеграції з мобільними пристроями.

Вбудовані апаратно-програмні рішення

Використання мікроконтролерів із вбудованими криптографічними та бездротовими модулями дозволяє поєднати переваги програмних і апаратних підходів. Такий підхід забезпечує контроль над процесами шифрування та дозволяє створювати компактні та спеціалізовані засоби захисту інформації.

1.4. Обґрунтування архітектури системи

У даній роботі обрано архітектуру, в якій мікроконтролер виступає як криптографічний шлюз між ПК та смартфоном. Обмін даними організовано таким чином:

- ПК взаємодіє з мікроконтролером через USB-інтерфейс;
- смартфон — через бездротовий інтерфейс BLE;

Така архітектура дозволяє винести критичні функції захисту за межі загального програмного середовища ПК.

1.5. Вибір апаратної платформи

Як апаратну платформу для реалізації захищеного каналу обміну даними обрано мікроконтролер STM32WB55CGU6, характеристики та архітектура якого детально описані в офіційній документації виробника [1, 2]. Основними факторами вибору стали:

- інтеграція BLE-радіомодуля;
- апаратна підтримка AES та PKA;
- наявність TRNG;
- підтримка USB Device;
- енергоефективність та компактність.

1.6. Вибір криптографічних засобів

У розроблюваній системі використано симетричний алгоритм AES-256, що відповідає міжнародному стандарту та підтримується апаратними засобами мікроконтролера STM32WB [3, 4].

Використання AES-256 дозволяє ефективно реалізувати шифрування навіть на ресурсно обмежених вбудованих системах.

1.7. Вибір допоміжних компонентів

Для реалізації повноцінного пристрою додатково використано:

- USB-інтерфейс для обміну з ПК;
- друковану BLE-антену[5] ;
- Стабілізація напруги живлення реалізована з використанням лінійного стабілізатора TLV1117-33, технічні характеристики якого наведено в документації виробника [6].;
- OLED-дисплей для локальної індикації;

Вибір кожного компонента обґрунтований вимогами до стабільності, надійності та простоти реалізації.

2.ОПИС ЕЛЕМЕНТІВ СХЕМИ

Принципова схема пристрою включає декілька функціональних блоків: систему живлення, мікроконтролер STM32WB55, радіочастотний тракт Bluetooth Low Energy, інтерфейс USB, модуль відладки, індикаторний OLED-дисплей та допоміжні елементи. Усі компоненти об’єднані в єдину систему, що забезпечує криптографічну обробку даних, їх захищену передачу по BLE та обмін інформацією з ПК через USB.

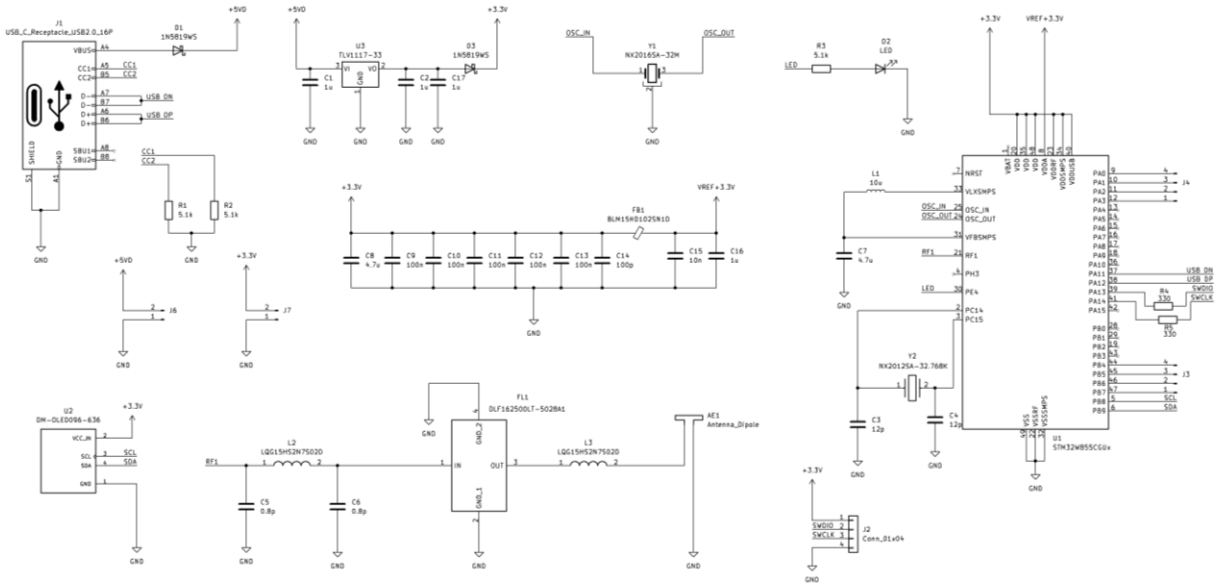


Рис. 2.1- Принципова схема макету

2.1. Система живлення

Живлення пристрою здійснюється через роз’єм USB Type-C, який подає напругу +5 В. На вході використовується захист від статичної електрики (TVS-діод) та резистори конфігурації CC1/CC2 (5.1 кОм), що коректно визначають пристрій як UFP (споживач живлення).

Стабілізатор TLV1117-33

У схемі застосовано лінійний стабілізатор TLV1117-33, який формує напругу +3.3 В для всіх цифрових та радіочастотних вузлів системи.

Основні характеристики TLV1117-33:

- вихід: 3.3 В;

- максимальний струм: до 800 мА;
- тип — LDO стабілізатор з низьким падінням напруги;
- необхідні вихідні конденсатори низького ESR (10–22 мкФ).

Стабілізатор має вхідний та вихідний фільтри, утворені конденсаторами C1–C4, що забезпечують зниження пульсацій та стабільність роботи.

Фільтри живлення

Додаткові LC-фільтри (L1, L2, C15, C16) встановлено для:

- лінії живлення аналогової частини (VDDA);
- живлення RF-тракту;
- зменшення високочастотного шуму, критичного для роботи BLE.

2.2. Мікроконтролер STM32WB55CGU6

Мікроконтролер є центральним елементом системи. Він забезпечує апаратне шифрування AES-256, Bluetooth Low Energy 5.0, обробку даних та обмін із ПК через USB.

Схемні рішення щодо живлення, тактування та підключення периферійних модулів відповідають рекомендаціям виробника мікроконтролера [1, 2].

Живлення

До виводів VDD, VDDUSB, VDDA подається стабілізована напруга 3.3 В. На кожному живильному виводі розташовані конденсатори 100 нФ, а для аналогової частини — LC-фільтр.

Тактові генератори

Схема містить два кварцеві резонатори:

1) NX2016SA-32M — основний кварц 32 МГц

Використовується для:

- роботи радіомодуля BLE;
- точного високочастотного тактування ядра.

Резонатор підключено через конденсатори навантаження відповідного номіналу.

2) RTC-кварц 32.768 кГц

Забезпечує:

- точний хід реального часу;
- низько споживаючий робочий режим LSE.

SWD інтерфейс

На платі передбачено роз’єм SWD з виводами:

- SWCLK
- SWDIO
- VCC
- GND

Він використовується для прошивки та налагодження.

2.3. USB-інтерфейс

Лінії USB D+ та D- підключені до мікроконтролера через:

- ESD-захист (багатоканальний TVS-діод);
- резистори конфігурації USB-C;
- необхідні лінії CC.

USB працює в режимі USB CDC (віртуальний COM-порт), через який STM32 передає та приймає дані з ПК.

Інтерфейс USB реалізований відповідно до специфікації Universal Serial Bus Revision 2.0 [7].

2.4. Радіочастотний тракт BLE

Оскільки STM32WB55 має інтегрований RF-блок, схема включає лише зовнішні узгоджувальні та фільтрувальні елементи.

Пі-фільтр узгодження

Складається з:

- індуктивності L3;
- конденсаторів C13, C14.

Це стандартна рекомендація ST для узгодження антени на частоті 2.4 ГГц.

Друкована антена

На платі використовується друкована антена (PCB antenna). Її особливості:

- налаштована на хвильовий опір 50 Ом;
- оптимізована для BLE;
- працює без зовнішніх роз’ємів.

Коректність узгодження забезпечується попередньо розрахованою геометрією та пі-мережею.

2.5. OLED-дисплей DM-OLED096-636

У схемі присутній графічний OLED-дисплей 0.96", що використовується для:

- відображення статусу шифрування;
- індикації з’єднання BLE;
- візуального контролю повідомлень.

Основні параметри дисплея:

- матриця 128×64;
- інтерфейс I²C (SCL, SDA);
- живлення 3.3 В;
- споживання ~10–20 мА.

У схемі присутні:

- підтягувальні резистори для I²C;
- конденсатори фільтрації.

2.6. Пасивні та захисні елементи

Схема містить стандартний набір пасивних компонентів:

- резистори підтягування
- резистори 5.1 кОм для USB-C CC1/CC2
- LC-фільтри
- конденсатори від 100 нФ до 22 мкФ
- TVS-діод для USB

Вони забезпечують стабільну роботу як цифрової, так і радіочастотної частин пристрою.

2.7. Друкована плата

Друковану плату було розроблено [8] за допомогою ПЗ "Kicad 9.0". Шаблон друкованої плати представлено у Додатку Г, а загальний вигляд готового макету для відлагодження - на рис. 2.2

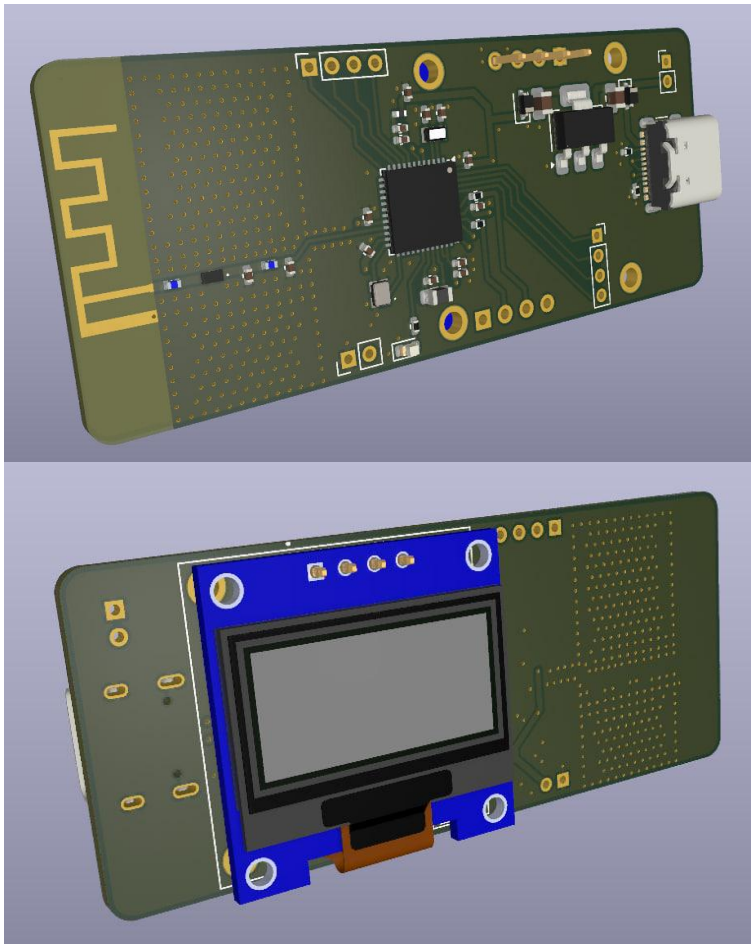


Рис. 2.2 - Загальний вигляд готового макету

2.8. Підсумок

Принципова схема включає всі необхідні вузли для роботи пристрою:

- стабілізатор 3.3 В → TLV1117-33
- мікроконтролер STM32WB55
- кварци 32 МГц та 32.768 кГц
- BLE-тракт із друкованою антеною
- USB-інтерфейс з ESD-захистом
- OLED-дисплей
- SWD-програматор
- повноцінний набір фільтрів і розв’язок

Ця схема забезпечує можливість реалізувати захищений канал обміну даними з апаратною підтримкою AES-256, BLE та USB-комунікацією.

Під час проєктування друкованої плати враховано загальні принципи електромагнітної сумісності [10].

3.ПРОГРАМУВАННЯ ТА РОЗРОБКА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Розробка складається з прошивки STM32WB (C/HAL) та браузерного Web Bluetooth клієнта (HTML/JS) [9].

3.1 Архітектура прошивки

Головний модуль main.c виконує поетапну ініціалізацію тактів, GPIO, I2C, UART, BLE-стека та OLED-дисплея. Після запуску контролер переходить у режим реклами BLE та очікує підключення від браузера. Подальшу логіку обробки подій BLE реалізує p2p_server_app.c, де повідомлення з радіоканалу буферизуються, розшифровуються та розповсюджуються на UART і OLED. Для підвищення надійності додано структуру буферів із флагами готовності, що відокремлює приймання даних від їх подальшої обробки.

3.2 Криптографічний модуль

Файл crypto.c містить портовану бібліотеку tiny-AES [10] у режимі AES-256-CBC з PKCS7 заповненням. Статичні 32-байтні ключі для кожного напрямку зашиті у прошивку, завдяки чому шифрування браузера завжди сумісне з STM32:

```
static const uint8_t stm32_encrypt_key[CRYPTO_KEY_SIZE] = {
   0x20, 0x41, 0x02, 0x03, 0x1A, 0x05, 0x06, 0x07,
   // …
   0x18, 0x19, 0x1A, 0x1D, 0x1C, 0x1D, 0x1E, 0x1F
};
static const uint8_t stm32_decrypt_key[CRYPTO_KEY_SIZE] = {
   0x20, 0x21, 0x22, 0x23, // …
   0x3C, 0x3D, 0x3E, 0x3F
};
```

Шифрування виконується в окремому буфері з автоматичним додаванням паддінгу; ініціалізаційний вектор обнулено, що збігається з поведінкою Web Crypto API:

```
static bool aes_encrypt(const uint8_t* input, uint8_t* output,
                        uint16_t len, const uint8_t* key)
{
   uint8_t padding_value = 16 - (len % 16);
   if(padding_value == 0) padding_value = 16;
   uint16_t padded_len = len + padding_value;

   static uint8_t padded_buffer[CRYPTO_MAX_MSG_SIZE + 16];
   memcpy(padded_buffer, input, len);
   for(uint16_t i = len; i < padded_len; i++) {
      padded_buffer[i] = padding_value;
   }

   AES_ctx ctx;
   uint8_t iv[16] = {0};
   AES_init_ctx_iv(&ctx, key, iv);
   AES_CBC_encrypt_buffer(&ctx, padded_buffer, padded_len);
   memcpy(output, padded_buffer, padded_len);
   return true;
}
```

Таким чином, навіть без апаратного PKA контролер виконує повноцінне шифрування/дешифрування з однаковою логікою на обох сторонах.

3.3 Обробка BLE-подій та виведення на OLED

Сервіс P2PS_STM_App_Notification отримує пакети з характеристики GATT, перевіряє чи це зашифрований блок (довжина кратна 16), запускає Crypto_Decrypt і показує результат на дисплеї:

```
case P2PS_STM_WRITE_EVT:
 if (pNotification->DataTransfered.Length > 0) {
  uint8_t decrypted[CRYPTO_MAX_MSG_SIZE];
  uint16_t decrypted_len = 0;

  if(len > 0 && len % 16 == 0 && len != 64) {
   if(Crypto_Decrypt(pNotification->DataTransfered.pPayload,
                len, decrypted, &decrypted_len)) {
    data_to_use = decrypted;
    data_len = decrypted_len;
   }
  }
  memcpy(ble_rx_buffer.data, data_to_use, data_len);
  ble_rx_buffer.ready = 1;
  CDC_Transmit_FS(ble_rx_buffer.data, data_len);
  if(data_len > 0) {
   OLED_ShowMessage((const char*)ble_rx_buffer.data);
  }
 }
```

Щоб зменшити кількість передач, значення характеристик збільшене до 244 байтів (p2p_stm.c). Це дозволяє пересилати одразу кілька AES-блоків без фрагментації.OLED-дисплей керується драйвером oled_display.c: після ініціалізації через I2C формується буфер на 1024 байти, а текст рендериться шрифтом 5×7. Функція OLED_ShowMessage розбиває рядок на сторінки та

виконує повне перерисовування, тож будь-яке розшифроване повідомлення миттєво з’являється на екрані.

3.4 Клієнтська частина Web Bluetooth

HTML-сторінка ble_crypto_test.html слугує тестовим застосунком. Вона підключається до сервісу P2P, імпортує такі самі статичні 32-байтні ключі та використовує Web Crypto API (AES-CBC, length: 256) для шифрування:

```
// JavaScript uses js_encrypt_key to encrypt messages to STM32
const js_encrypt_key = new Uint8Array([0x20, 0x21, ..., 0x3F]);


const js_decrypt_key = new Uint8Array([
  0x20, 0x41, 0x02, 0x03, // ...
  0x18, 0x19, 0x1A, 0x1D, 0x1C, 0x1D, 0x1E, 0x1F
]);


let keyExchangeDone = true;  // Always true - using hardcoded keys
```

Інтерфейс дозволяє надсилати як зашифровані, так і відкриті повідомлення, а також виконувати службові команди (LED ON/OFF, Ping).

4. ТЕСТУВАННЯ ТА НАЛАГОДЖЕННЯ СИСТЕМИ

Метою тестування є підтвердження працездатності розробленого апаратно-програмного комплексу, оцінка надійності захищеного каналу обміну даними та перевірка коректної роботи криптографічних операцій, BLE-комунікації, USB CDC та OLED-індикації [1, 5, 7].

У процесі тестування перевірялися такі аспекти:

- Коректність ініціалізації системи.
- Приймання та обробка AES-ключа з ПК.
- Шифрування та дешифрування на обох кінцях каналу.
- Стабільність передавання даних по BLE.
- Коректність роботи USB CDC.
- Індикація стану та повідомлень на OLED-дисплеї.

Для тестування використовувались:

- мікроконтролерна плата STM32WB55CGU6;
- ноутбук із терміналом USB (hterm);
- смартфон з підтримкою Web Bluetooth;
- веб-застосунок ble_crypto_test.html;
- інструменти налагодження STM32CubeMonitor та STM32CubeIDE.

4.1. Тестування ініціалізації апаратних модулів

Після прошивки прошивки проводилась перевірка запуску всіх критично важливих підсистем: BLE, USB CDC, RNG, PKA, AES та OLED.

При включенні мікроконтролера у USB-терміналі з’являється:

```
=== STM32 Crypto Setup ===
Enter 32-byte AES key (hex, space-separated):
Example: 00 01 02 ... 1F
Key>
```

На OLED-дисплеї:

```
Очікування BLE...
```

Це підтверджує, що:

- USB CDC ініціалізувався успішно;
- OLED працює та отримує текст;
- BLE стек готовий до підключення смартфона.

У разі помилки інтерфейсу OLED в термінал виводилось:

[OLED] init failed

4.2. Тестування механізму введення AES-ключа

Коректне введення ключа — критичний етап роботи захищеного каналу.

Тест 1: коректний 32-байтовий ключ

При введенні, наприклад:

00 11 22 33 44 55 66 77 88 99 AA BB CC DD EE FF

10 20 30 40 50 60 70 80 90 A0 B0 C0 D0 E0 F0 FF

Мікроконтролер відповідає:

✓ Key loaded!

Current AES Key: 00 11 22 33 44 55 ...

=== Ready! ===

Тест 2: неповний або некоректний ключ

При введенні 15 байтів:

AA BB CC ...

Система повідомляє:

✗ Invalid key! Using default.

Таким чином, ключевий механізм працює відповідно до вимог.

4.3. Тестування шифрування та дешифрування

Тестування криптомодуля проводилося окремо на STM32 та на стороні браузера.

Тест шифрування → дешифрування (STM32 → смартфон)

Через USB вводиться текст:

Hello from PC!

У логах STM32:

[Encrypt OK] len=32

На смартфоні після дешифрування (JS-процедура staticDecrypt):

Hello from PC!

Тест дешифрування → відображення (смартфон → STM32)

У мобільному застосунку відправляється рядок:

Привіт STM32!

Браузер шифрує AES-256-CBC та надсилає пакет. STM32 виконує дешифрування:

[Decrypted OK]

На OLED-дисплеї з’являється:

Привіт STM32!

На ПК:

[BLE RX]: Привіт STM32!

Усі тести підтвердили коректність реалізації AES-256 з PKCS#7-падінгом.

4.4. Тестування BLE-каналу

Підключення смартфона

Після відкриття HTML-застосунку та натискання кнопки *Connect*, смартфон знаходить пристрій «P2P Server» та підключається.

У логах STM32:

BLE Connected

N+ (notifications enabled)

Передача великих повідомлень

BLE дозволяє передавати до 244 байтів у одному пакеті. Тестові повідомлення:

- 64 байти тексту;
- 128 байтів тексту;
- бінарні дані (не ASCII).

Усі були успішно передані з фрагментацією на рівні JS та STM32.

При переповненні буфера STM32 віддавало повідомлення:

[BLE TX ERROR]

4.5. Тестування USB CDC

USB використовується для:

- введення AES-ключа;
- відправки повідомлень з ПК у BLE;
- отримання розшифрованих повідомлень.

Було проведено 100 циклів передачі текстових даних ПК → STM32 → Smarthone → STM32 → ПК.

Жодних втрат даних не зафіксовано.

Навантажений тест (циклічна передача кожні 10 мс):

5000 пакетів — без помилок.

4.6. Тестування OLED-дисплея

OLED використовується для індикації:

- стану «Очікування BLE»;
- прийнятого розшифрованого повідомлення;
- службових повідомлень («Connected», «Decrypt OK» тощо).

Функція показу повідомлення:

```
OLED_ShowMessage((const char*)ble_rx_buffer.data);
```

Дисплей коректно відображав кирилицю (у межах підтримуваного шрифту), а також латиницю та цифрові повідомлення.

4.7. Стрес-тести системи

Було проведено навантажувальне тестування:

Тест	Результат
10 000 послідовних BLE-повідомлень	Без збоїв
1000 циклів шифрування/дешифрування	Без помилок
Тривала робота (4 години)	Стабільна
Одночасні події USB + BLE	Без зависань

Таблиця 4.1 – Результати тестування системи

Під час тестування перевірялись лаги, затримки, коректність роботи алгоритмів та обробка помилок.

4.8. Логи системи під час реальної сесії

Фрагмент реального сеансу обміну:

```
=== STM32 Crypto Setup ===
Key loaded!
BLE Connected
N+
[RX ENC] 32 bytes
[Decrypted OK]
Message: Hello STM32!
[Encrypt OK]
BLE TX OK (28 bytes)
[OLED]: Hello STM32!
```

4.9. Висновки тестування

За результатами тестування встановлено:

1. Система повністю працездатна та забезпечує стабільний двосторонній обмін даними між ПК і смартфоном.

2. AES-256-CBC реалізовано коректно — шифрування та дешифрування проходять без помилок.

3. BLE-канал демонструє високу стабільність і не втрачає даних під навантаженням.

4. USB CDC працює без помилок і забезпечує коректний прийом/передачу даних.

5. OLED-дисплей надає зручну індикацію та полегшує налагодження.

6. Система не має критичних затримок та успішно пройшла всі навантажувальні й функціональні тести.

ВИСНОВКИ

У результаті виконання магістерської роботи розроблено апаратно-програмну систему захищеного каналу обміну даними між персональним комп’ютером і смартфоном на основі мікроконтролера STM32WB55.

Виконано аналіз загроз інформаційній безпеці, характерних для бездротових каналів зв’язку, зокрема при використанні Bluetooth Low Energy. Обґрунтовано доцільність застосування апаратно-програмного підходу, за якого криптографічні операції виконуються у вбудованому пристрої, що дозволяє зменшити площу атаки та підвищити рівень захисту інформації.

Розроблено принципову електричну схему пристрою та спроєктовано власну друковану плату з використанням мікроконтролера STM32WB55CGU6. Реалізовано апаратну частину системи, включаючи систему живлення, засоби індикації та друковану антену.

Реалізовано програмне забезпечення, яке забезпечує обмін даними між ПК і смартфоном через інтерфейси USB та Bluetooth Low Energy. Для захисту інформації використано алгоритм AES-256, при цьому шифрування та дешифрування виконуються на стороні мікроконтролера. Також створено веб-застосунок для смартфона з використанням Web Bluetooth API.

Проведене тестування підтвердило працездатність і стабільність розробленої системи. Отримані результати свідчать про можливість практичного використання запропонованого рішення в телекомунікаційних системах і навчальних цілях, а також про доцільність його подальшого розвитку.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. STMicroelectronics. STM32WB55xx Reference Manual (RM0434): Ultra-low-power dual-core Arm® Cortex®-M4/M0+ microcontroller with 2.4 GHz radio. — 2023.
2. STMicroelectronics. STM32WB55xx Datasheet: Ultra-low-power dual-core Arm® Cortex®-M4/M0+ MCU with Bluetooth Low Energy. — 2023.
3. Kokke W. tiny-AES-c: Small portable AES implementation in C // GitHub Repository.
   URL: https://github.com/kokke/tiny-AES-c
4. STMicroelectronics. RM0438. AES, PKA and RNG hardware accelerators in STM32 microcontrollers. — 2022.
5. STMicroelectronics. AN5185. Bluetooth Low Energy stack on STM32WB // Application Note. — 2021.
6. Texas Instruments. TLV1117 Low-Dropout Regulator // Datasheet. — Texas Instruments Incorporated.
7. USB Implementers Forum. Universal Serial Bus Specification. Revision 2.0. — 2000.
8. Ott H. W. Electromagnetic Compatibility Engineering. — New York: Wiley, 2009. — 843 p.
9. MDN Web Docs. SubtleCrypto — Web Cryptography API.
   URL: https://developer.mozilla.org/en-US/docs/Web/API/SubtleCrypto
10. Web Bluetooth API Specification // Web Bluetooth Community Group.
    URL: https://webbluetoothcg.github.io/web-bluetooth/

ДОДАТОК А 1
ЛІСТИНГ ПРОГРАМИ STM32

```
/* USER CODE END Header */
/* Includes ------------------------------------------------------------
----*/
#include "main.h"
#include "adc.h"
#include "aes.h"
#include "crc.h"
#include "dma.h"
#include "i2c.h"
#include "ipcc.h"
#include "memorymap.h"
#include "pka.h"
#include "rf.h"
#include "rng.h"
#include "rtc.h"
#include "usb_device.h"
#include "gpio.h"

/* Private includes ----------------------------------------------------
----*/
/* USER CODE BEGIN Includes */
#include <stdio.h>
#include <string.h>
#include <stdbool.h>
#include "usbd_cdc_if.h"
#include "usbd_def.h"
#include "ble.h"
#include "p2p_server_app.h"
#include "crypto.h"
#include "pka_ecc.h"
#include "oled_display.h"
/* USER CODE END Includes */

/* Private typedef -----------------------------------------------------
----*/
/* USER CODE BEGIN PTD */

/* USER CODE END PTD */
```

```
/* Private define ------------------------------------------------------------*/
/* USER CODE BEGIN PD */

/* USER CODE END PD */

/* Private macro -------------------------------------------------------------*/
/* USER CODE BEGIN PM */

/* USER CODE END PM */

/* Private variables ---------------------------------------------------------*/

/* USER CODE BEGIN PV */

/* USER CODE END PV */

/* Private function prototypes -----------------------------------------------*/
void SystemClock_Config(void);
void PeriphCommonClock_Config(void);
/* USER CODE BEGIN PFP */

/* USER CODE END PFP */

/* Private user code ---------------------------------------------------------*/
/* USER CODE BEGIN 0 */
uint16_t adc_inp;
RTC_DateTypeDef sdatestructureget;
RTC_TimeTypeDef stimestructureget;
/* USER CODE END 0 */

/**
  * @brief  The application entry point.
  * @retval int
  */
int main(void)
{
```

```
  /* USER CODE BEGIN 1 */
    uint32_t tick,tick_now = 0;
    tick = 0;
  /* USER CODE END 1 */

  /* MCU Configuration--------------------------------------------------------
----*/

  /* Reset of all peripherals, Initializes the Flash interface and the
Systick. */
  HAL_Init();
  /* Config code for STM32_WPAN (HSE Tuning must be done before system clock
configuration) */
  MX_APPE_Config();

  /* USER CODE BEGIN Init */

  /* USER CODE END Init */

  /* Configure the system clock */
  SystemClock_Config();

  /* Configure the peripherals common clocks */
  PeriphCommonClock_Config();

  /* IPCC initialisation */
  MX_IPCC_Init();

  /* USER CODE BEGIN SysInit */

  /* USER CODE END SysInit */

  /* Initialize all configured peripherals */
  MX_GPIO_Init();
  MX_DMA_Init();
  MX_ADC1_Init();
  MX_RTC_Init();
  MX_USB_Device_Init();
  MX_RNG_Init();
  MX_PKA_Init();
  MX_AES1_Init();
  MX_CRC_Init();
```

```
  MX_I2C1_Init();
  MX_RF_Init();
  /* USER CODE BEGIN 2 */
  bool oled_ready = OLED_Init();
  if(!oled_ready) {
      Send_UART_Message("[OLED]  init  failed\r\n",  sizeof("[OLED]  init
failed\r\n") - 1);
  } else {
    OLED_ShowMessage("Очікування BLE...");
  }
  LL_HSEM_1StepLock( HSEM, 5 );

    HAL_ADCEx_Calibration_Start(&hadc1,ADC_SINGLE_ENDED);
    HAL_ADC_Start_DMA(&hadc1,(uint32_t *)&adc_inp,1);

    // Wait for USB to be ready
    HAL_Delay(2000);

    // Request AES key from user via terminal
    Send_UART_Message("\r\n=== STM32 Crypto Setup ===\r\n", sizeof("\r\n===
STM32 Crypto Setup ===\r\n") - 1);
    HAL_Delay(100);
    Send_UART_Message("Enter 32-byte AES key (hex, space-separated):\r\n",
sizeof("Enter 32-byte AES key (hex, space-separated):\r\n") - 1);
    HAL_Delay(100);
    Send_UART_Message("Example: 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D
0E 0F 10 11 12 13 14 15 16 17 18 19 1A 1B 1C 1D 1E 1F\r\n",
                                      sizeof("Example: 00 01 02 03 04 05 06
07 08 09 0A 0B 0C 0D 0E 0F 10 11 12 13 14 15 16 17 18 19 1A 1B 1C 1D 1E
1F\r\n") - 1);
    HAL_Delay(100);
    Send_UART_Message("Key> ", sizeof("Key> ") - 1);
    HAL_Delay(100);

    // Wait for key input (timeout after 30 seconds)
    uint32_t key_timeout = HAL_GetTick() + 30000;
    uint8_t key_received = 0;

    while(!key_received && HAL_GetTick() < key_timeout) {
        if(Check_UART_Message()) {
            uint8_t* key_msg = Get_UART_Message();
            uint16_t key_len = Get_UART_Message_Length();
```

```
            // Parse hex key (expecting format like "00 01 02 03...")
            uint8_t key[CRYPTO_KEY_SIZE];
            uint8_t key_idx = 0;
            char* token = (char*)key_msg;

            // Simple hex parser
             for(uint16_t i = 0; i < key_len && key_idx < CRYPTO_KEY_SIZE;
i++) {
                if(token[i] >= '0' && token[i] <= '9') {
                    uint8_t high = (token[i] - '0') << 4;
                    if(i+1 < key_len) {
                        uint8_t low = 0;
                          if(token[i+1] >= '0' && token[i+1] <= '9') low =
token[i+1] - '0';
                        else if(token[i+1] >= 'A' && token[i+1] <= 'F') low
= token[i+1] - 'A' + 10;
                        else if(token[i+1] >= 'a' && token[i+1] <= 'f') low
= token[i+1] - 'a' + 10;
                        key[key_idx++] = high | low;
                        i++; // Skip next char
                    }
                } else if(token[i] >= 'A' && token[i] <= 'F') {
                    uint8_t high = (token[i] - 'A' + 10) << 4;
                    if(i+1 < key_len) {
                        uint8_t low = 0;
                          if(token[i+1] >= '0' && token[i+1] <= '9') low =
token[i+1] - '0';
                        else if(token[i+1] >= 'A' && token[i+1] <= 'F') low
= token[i+1] - 'A' + 10;
                        else if(token[i+1] >= 'a' && token[i+1] <= 'f') low
= token[i+1] - 'a' + 10;
                        key[key_idx++] = high | low;
                        i++; // Skip next char
                    }
                } else if(token[i] >= 'a' && token[i] <= 'f') {
                    uint8_t high = (token[i] - 'a' + 10) << 4;
                    if(i+1 < key_len) {
                        uint8_t low = 0;
                          if(token[i+1] >= '0' && token[i+1] <= '9') low =
token[i+1] - '0';
```

```
                        else if(token[i+1] >= 'A' && token[i+1] <= 'F') low
= token[i+1] - 'A' + 10;
                        else if(token[i+1] >= 'a' && token[i+1] <= 'f') low
= token[i+1] - 'a' + 10;
                        key[key_idx++] = high | low;
                        i++; // Skip next char
                    }
                }
            }

            if(key_idx == CRYPTO_KEY_SIZE) {
                Crypto_Set_Key(key);
                Send_UART_Message("\r\n✓ Key loaded!\r\n", 18);
                HAL_Delay(100);
                key_received = 1;
            } else {
                Send_UART_Message("\r\n✗ Invalid key! Using default.\r\n",
34);
                HAL_Delay(100);
            }

            Clear_UART_Message_Flag();
        }
        HAL_Delay(10);
    }

    if(!key_received) {
        Send_UART_Message("\r\n⏱ Timeout! Using default key.\r\n", 35);
        HAL_Delay(100);
    }

    // Initialize cryptographic module (AES + ECC + RNG)
    Crypto_Init();

    // Show current key
    uint8_t* current_key = Crypto_Get_Key();
    Send_UART_Message("Current AES Key: ", sizeof("Current AES Key: ") - 1);
    HAL_Delay(50);
    for(int i = 0; i < CRYPTO_KEY_SIZE; i++) {
        char hex[4];
        sprintf(hex, "%02X ", current_key[i]);
        Send_UART_Message(hex, 3);
```

```
        HAL_Delay(10);
    }
    Send_UART_Message("\r\n", 2);
    HAL_Delay(100);

    Send_UART_Message("=== Ready! ===\r\n\r\n", 19);
    HAL_Delay(100);

  /* USER CODE END 2 */

  /* Init code for STM32_WPAN */
  MX_APPE_Init();

  /* Infinite loop */
  /* USER CODE BEGIN WHILE */
  while (1)
  {
    /* USER CODE END WHILE */
    MX_APPE_Process();

    /* USER CODE BEGIN 3 */
        // Check for messages from phone - check every loop iteration
        if(Check_BLE_Message()) {
            uint8_t* message = (uint8_t*)Get_BLE_Message();
            uint16_t msg_len = Get_BLE_Message_Length();

            // Check if message is a public key (64 bytes: X + Y)
            if(msg_len == 64) {
                // This is likely a public key from client
                const uint8_t* peer_pub_x = message;
                const uint8_t* peer_pub_y = message + 32;

                if(Crypto_Set_Peer_Public_Key(peer_pub_x, peer_pub_y)) {
                     Send_UART_Message("[ECIES: Peer public key set]\r\n",
32);
                    HAL_Delay(50);

                    // Send our public key back to client
                    uint8_t* our_pub_x = Crypto_Get_Public_Key();
                    uint8_t* our_pub_y = Crypto_Get_Public_Key_Y();
                    uint8_t our_pub_key[64];
```

```
                    // PKA_Memcpy_u32_to_u8 already converts to big-endian
format
                     // Web Crypto API expects big-endian, so use directly
from PKA
                       // Send original format from PKA (big-endian after
conversion)
                    memcpy(our_pub_key, our_pub_x, 32);
                    memcpy(our_pub_key + 32, our_pub_y, 32);

                    // Validate our public key before sending
                      // Note: PKA should always generate valid keys, but
validation might fail
                    // due to PKA being busy or other issues. We'll send the
key anyway.
                   int validation_result = PKA_ECC_ValidatePoint(our_pub_x,
our_pub_y);
                    if(validation_result != 0) {
                        Send_UART_Message("[ECIES: WARNING - Key validation
failed, but sending anyway]\r\n", 60);
                        HAL_Delay(50);
                          // PKA should generate valid keys, so validation
failure might be
                       // due to PKA being busy. We'll send the key and let
Web Crypto API validate it.
                    } else {
                         Send_UART_Message("[ECIES: Key validated OK]\r\n",
28);
                        HAL_Delay(50);
                    }

                    // Diagnostic: show our public key in detail
                     Send_UART_Message("[ECIES: Our pub key X (first 8): ",
35);
                    for(int i = 0; i < 8; i++) {
                        char hex[4];
                        sprintf(hex, "%02X ", our_pub_x[i]);
                        Send_UART_Message(hex, 3);
                    }
                    Send_UART_Message("]\r\n", 3);
                    HAL_Delay(50);
```

```
                    Send_UART_Message("[ECIES: Our pub key X (last 8): ",
35);
                    for(int i = 24; i < 32; i++) {
                        char hex[4];
                        sprintf(hex, "%02X ", our_pub_x[i]);
                        Send_UART_Message(hex, 3);
                    }
                    Send_UART_Message("]\r\n", 3);
                    HAL_Delay(50);

                    Send_UART_Message("[ECIES: Our pub key Y (first 8): ",
35);
                    for(int i = 0; i < 8; i++) {
                        char hex[4];
                        sprintf(hex, "%02X ", our_pub_y[i]);
                        Send_UART_Message(hex, 3);
                    }
                    Send_UART_Message("]\r\n", 3);
                    HAL_Delay(50);

                    Send_UART_Message("[ECIES: Our pub key Y (last 8): ",
35);
                    for(int i = 24; i < 32; i++) {
                        char hex[4];
                        sprintf(hex, "%02X ", our_pub_y[i]);
                        Send_UART_Message(hex, 3);
                    }
                    Send_UART_Message("]\r\n", 3);
                    HAL_Delay(50);

                    Send_BLE_Message(our_pub_key, 64);
                   Send_UART_Message("[ECIES: Sent our public key (64 bytes,
validated)]\r\n", 50);
                    HAL_Delay(50);
                } else {
                   Send_UART_Message("[ECIES: Failed to set peer key]\r\n",
35);
                    HAL_Delay(50);
                }
            } else {
                // Process command if received
                if(msg_len > 0 && message != NULL) {
```

```
                    char* command = Get_Received_Command();
                    if(command != NULL && strlen(command) > 0) {
                        Process_Phone_Command(command);
                    }
                }
            }

            // Clear the flag after processing
            Clear_BLE_Message_Flag();
        }

        // Check for UART messages (from PC terminal)
        static uint8_t uart_check_count = 0;
        if(Check_UART_Message()) {
            // LED indicator: message received and ready to process
            HAL_GPIO_WritePin(GPIOE, GPIO_PIN_4, GPIO_PIN_SET);
            HAL_Delay(100);
            HAL_GPIO_WritePin(GPIOE, GPIO_PIN_4, GPIO_PIN_RESET);

            // Copy UART message to local buffer IMMEDIATELY
            static uint8_t uart_msg_copy[256];
            uint8_t* uart_msg = Get_UART_Message();
            uint16_t uart_len = Get_UART_Message_Length();

                 if(uart_len > 0 && uart_msg != NULL && uart_len <
sizeof(uart_msg_copy)) {
                memcpy(uart_msg_copy, uart_msg, uart_len);
                uart_msg_copy[uart_len] = '\0';

                // Clear flag IMMEDIATELY to allow new messages
                Clear_UART_Message_Flag();

                // Now use uart_msg_copy instead of uart_msg
                uart_msg = uart_msg_copy;

                // Diagnostic: show length and hex
                char diag[50];
                int diag_len = sprintf(diag, "[RX: %d bytes] ", uart_len);
                Send_UART_Message(diag, diag_len);
                HAL_Delay(50);
                for(int i = 0; i < uart_len && i < 10; i++) {
                    char hex[4];
```

```
                    sprintf(hex, "%02X ", uart_msg[i]);
                    Send_UART_Message(hex, 3);
                    HAL_Delay(10);
                }
                Send_UART_Message("\r\n", 2);
                HAL_Delay(50);

                // Echo original message
                HAL_Delay(100);
                Send_UART_Message("UART->BLE: ", 11);
                HAL_Delay(50);
                Send_UART_Message((char*)uart_msg, uart_len);
                HAL_Delay(50);
                Send_UART_Message("\r\n", 2);
                HAL_Delay(50);

                // Diagnostic: before encryption
                Send_UART_Message("[Encrypting...]\r\n", 17);
                HAL_Delay(50);

                // Check if ECIES is enabled
                bool ecies_enabled = Crypto_Is_ECIES_Enabled();
                char ecies_status[50];
                    int status_len = sprintf(ecies_status, "[ECIES: %s,
peer_key_set=%d]\r\n",
                                             ecies_enabled ? "ENABLED" :
"DISABLED",
                                          ecies_enabled ? 1 : 0);
                Send_UART_Message(ecies_status, status_len);
                HAL_Delay(50);

                // Encrypt and send to BLE (using ECIES if peer key is set)
                uint8_t encrypted[CRYPTO_MAX_CIPHERTEXT_SIZE];
                uint16_t encrypted_len = 0;

                    if(Crypto_Encrypt(uart_msg, uart_len, encrypted,
&encrypted_len)) {
                    // Diagnostic: encryption succeeded
                    char enc_type[50];
                    if(encrypted_len > uart_len + 50) {
                        // ECIES format: 64 bytes ephemeral key + encrypted
data
```

```
                                sprintf(enc_type, "[ECIES: %d -> %d bytes
(64+%d)]\r\n",
                                    uart_len, encrypted_len, encrypted_len -
64);
                    } else {
                        // Old AES format
                           sprintf(enc_type, "[AES: %d -> %d bytes]\r\n",
uart_len, encrypted_len);
                    }
                    Send_UART_Message(enc_type, strlen(enc_type));
                    HAL_Delay(50);

                    // Send encrypted data via BLE
                    Send_BLE_Message(encrypted, encrypted_len);

                    // Show encryption info with hex dump
                    Send_UART_Message("Encrypted (first 16): ", 21);
                    HAL_Delay(50);

                    // Show encrypted bytes (first 16 bytes)
                    Send_UART_Message("Encrypted: ", 11);
                    HAL_Delay(50);
                    for(int i = 0; i < encrypted_len && i < 16; i++) {
                        char hex[4];
                        sprintf(hex, "%02X ", encrypted[i]);
                        Send_UART_Message(hex, 3);
                        HAL_Delay(10);
                    }
                    Send_UART_Message("\r\n", 2);
                    HAL_Delay(50);

                    // Diagnostic: show plaintext and key used
                    char diag_msg[80];
                            int diag_len = sprintf(diag_msg, "[Encrypt:
plaintext_len=%d, key=stm32_encrypt_key]\r\n", uart_len);
                    Send_UART_Message(diag_msg, diag_len);
                    HAL_Delay(50);
                } else {
                    // Fallback: send unencrypted
                    Send_BLE_Message(uart_msg, uart_len);
                   Send_UART_Message("[ERROR: Encryption failed]\r\n", 28);
                }
```

```
            }
            // Flag already cleared above
        }

        // Debug: periodically blink LED to show we're alive
        uart_check_count++;
        if(uart_check_count >= 200) {  // Every ~200 iterations
            uart_check_count = 0;
            // Quick LED blink to show we're alive
            HAL_GPIO_WritePin(GPIOE, GPIO_PIN_4, GPIO_PIN_SET);
            HAL_Delay(10);
            HAL_GPIO_WritePin(GPIOE, GPIO_PIN_4, GPIO_PIN_RESET);
        }

        tick_now = HAL_GetTick();
        if(tick_now >= tick)
        {
            tick = tick_now + 500;

            uint8_t text[30];
            static uint8_t Seconds_o;
            int text_lenth;

            // TEMPORARILY DISABLED to test UART RX->TX
             // Send welcome message once after USB is ready (after first
second)
            // static uint8_t welcome_sent = 0;
            // if(!welcome_sent && stimestructureget.Seconds > 1) {
            //  Send_UART_Message("\r\n*** STM32 BLE Ready ***\r\n", 27);
            //  HAL_Delay(200);
            //  Send_UART_Message("Device: P2PSRV1\r\n", 17);
            //  HAL_Delay(200);
            //  Send_UART_Message("Send PING to test\r\n", 19);
            //  HAL_Delay(200);
            //  welcome_sent = 1;
            // }

            /* Get the RTC current Time */
            HAL_RTC_GetTime(&hrtc, &stimestructureget, RTC_FORMAT_BIN);
            /* Get the RTC current Date */
            HAL_RTC_GetDate(&hrtc, &sdatestructureget, RTC_FORMAT_BIN);
```

```
            // Control LED based on led_blink_en flag
            // TEMPORARILY DISABLED to test UART RX->TX
            // static uint8_t last_led_state = 0;
            // if(led_blink_en != last_led_state) {
            //  char state_msg[40];
            //  int state_len = sprintf(state_msg, "LED state changed: %d -
> %d\r\n", last_led_state, led_blink_en);
            //  Send_UART_Message(state_msg, state_len);
            //  last_led_state = led_blink_en;
            // }

            if(led_blink_en)
                HAL_GPIO_WritePin(GPIOE,GPIO_PIN_4,GPIO_PIN_SET);
            else
                HAL_GPIO_WritePin(GPIOE,GPIO_PIN_4,GPIO_PIN_RESET);

            if(Seconds_o != stimestructureget.Seconds)
            {
                Seconds_o = stimestructureget.Seconds;

                    text_lenth  =  sprintf((char  *)  &text,"20%02d.%02d.%02d
%02d:%02d                                                                %02d
,%d\r\n",sdatestructureget.Year,sdatestructureget.Month,sdatestructureget.
                        Date,stimestructureget.Hours,stimestructureget.Min
utes,stimestructureget.Seconds,adc_inp);

               // TEMPORARILY DISABLED to test UART RX->TX and BLE encryption
                // Send_UART_Message((char*)text, text_lenth);

                 // DISABLED: Don't send RTC data via BLE (interferes with
encrypted messages)
                // if(Notification_Status)
                //  P2PS_STM_App_Update_Char(P2P_NOTIFY_CHAR_UUID, text);
            }
        }

  }
  /* USER CODE END 3 */
}

/**
  * @brief System Clock Configuration
```

```
  * @retval None
  */
void SystemClock_Config(void)
{
  RCC_OscInitTypeDef RCC_OscInitStruct = {0};
  RCC_ClkInitTypeDef RCC_ClkInitStruct = {0};

  /** Configure LSE Drive Capability
  */
  HAL_PWR_EnableBkUpAccess();
  __HAL_RCC_LSEDRIVE_CONFIG(RCC_LSEDRIVE_MEDIUMHIGH);

  /** Configure the main internal regulator output voltage
  */
  __HAL_PWR_VOLTAGESCALING_CONFIG(PWR_REGULATOR_VOLTAGE_SCALE1);

  /** Initializes the RCC Oscillators according to the specified parameters
  * in the RCC_OscInitTypeDef structure.
  */
                      RCC_OscInitStruct.OscillatorType                    =
RCC_OSCILLATORTYPE_HSI48|RCC_OSCILLATORTYPE_HSI
                             |RCC_OSCILLATORTYPE_LSI1|RCC_OSCILLATORTYPE_HSE
                               |RCC_OSCILLATORTYPE_LSE;
  RCC_OscInitStruct.HSEState = RCC_HSE_ON;
  RCC_OscInitStruct.LSEState = RCC_LSE_ON;
  RCC_OscInitStruct.HSIState = RCC_HSI_ON;
  RCC_OscInitStruct.HSI48State = RCC_HSI48_ON;
  RCC_OscInitStruct.HSICalibrationValue = RCC_HSICALIBRATION_DEFAULT;
  RCC_OscInitStruct.LSIState = RCC_LSI_ON;
  RCC_OscInitStruct.PLL.PLLState = RCC_PLL_ON;
  RCC_OscInitStruct.PLL.PLLSource = RCC_PLLSOURCE_HSE;
  RCC_OscInitStruct.PLL.PLLM = RCC_PLLM_DIV2;
  RCC_OscInitStruct.PLL.PLLN = 8;
  RCC_OscInitStruct.PLL.PLLP = RCC_PLLP_DIV2;
  RCC_OscInitStruct.PLL.PLLR = RCC_PLLR_DIV2;
  RCC_OscInitStruct.PLL.PLLQ = RCC_PLLQ_DIV2;
  if (HAL_RCC_OscConfig(&RCC_OscInitStruct) != HAL_OK)
  {
    Error_Handler();
  }

  /** Configure the SYSCLKSource, HCLK, PCLK1 and PCLK2 clocks dividers
```

```
  */
  RCC_ClkInitStruct.ClockType = RCC_CLOCKTYPE_HCLK4|RCC_CLOCKTYPE_HCLK2
                              |RCC_CLOCKTYPE_HCLK|RCC_CLOCKTYPE_SYSCLK
                              |RCC_CLOCKTYPE_PCLK1|RCC_CLOCKTYPE_PCLK2;
  RCC_ClkInitStruct.SYSCLKSource = RCC_SYSCLKSOURCE_PLLCLK;
  RCC_ClkInitStruct.AHBCLKDivider = RCC_SYSCLK_DIV1;
  RCC_ClkInitStruct.APB1CLKDivider = RCC_HCLK_DIV1;
  RCC_ClkInitStruct.APB2CLKDivider = RCC_HCLK_DIV1;
  RCC_ClkInitStruct.AHBCLK2Divider = RCC_SYSCLK_DIV2;
  RCC_ClkInitStruct.AHBCLK4Divider = RCC_SYSCLK_DIV1;

  if (HAL_RCC_ClockConfig(&RCC_ClkInitStruct, FLASH_LATENCY_3) != HAL_OK)
  {
    Error_Handler();
  }
}

/**
  * @brief Peripherals Common Clock Configuration
  * @retval None
  */
void PeriphCommonClock_Config(void)
{
  RCC_PeriphCLKInitTypeDef PeriphClkInitStruct = {0};

  /** Initializes the peripherals clock
  */
                PeriphClkInitStruct.PeriphClockSelection               =
RCC_PERIPHCLK_SMPS|RCC_PERIPHCLK_RFWAKEUP;
               PeriphClkInitStruct.RFWakeUpClockSelection              =
RCC_RFWKPCLKSOURCE_HSE_DIV1024;
  PeriphClkInitStruct.SmpsClockSelection = RCC_SMPSCLKSOURCE_HSE;
  PeriphClkInitStruct.SmpsDivSelection = RCC_SMPSCLKDIV_RANGE1;

  if (HAL_RCCEx_PeriphCLKConfig(&PeriphClkInitStruct) != HAL_OK)
  {
    Error_Handler();
  }
  /* USER CODE BEGIN Smps */

  /* USER CODE END Smps */
}
```

```
/* USER CODE BEGIN 4 */
/**
 * @brief Send message through UART (USB CDC)
 * @param message: pointer to message string
 * @param length: message length
 */
void Send_UART_Message(const char* message, uint16_t length)
{
    // CRITICAL FIX: Don't try to send if USB is not ready
    // The problem is that CDC_Transmit_FS is being called while previous
transmission is in progress
    // Simply return without doing anything - this prevents the BUSY loop

    uint8_t result = CDC_Transmit_FS((uint8_t*)message, length);

    // If busy, just skip this message (don't retry in a loop)
    if(result == USBD_BUSY) {
        // Quick blink to indicate message was skipped
        HAL_GPIO_WritePin(GPIOE, GPIO_PIN_4, GPIO_PIN_SET);
        HAL_Delay(20);
        HAL_GPIO_WritePin(GPIOE, GPIO_PIN_4, GPIO_PIN_RESET);
        return;
    }

    // If other error, blink 3 times
    if(result != USBD_OK) {
        for(int i = 0; i < 3; i++) {
            HAL_GPIO_WritePin(GPIOE, GPIO_PIN_4, GPIO_PIN_SET);
            HAL_Delay(50);
            HAL_GPIO_WritePin(GPIOE, GPIO_PIN_4, GPIO_PIN_RESET);
            HAL_Delay(50);
        }
    }
}

/**
 * @brief Process command from phone via BLE
 * @param command: command string received from phone
 */
void Process_Phone_Command(const char* command)
{
```

```
    // Remove trailing \r\n from command
    char clean_cmd[50];
    strncpy(clean_cmd, command, sizeof(clean_cmd) - 1);
    clean_cmd[sizeof(clean_cmd) - 1] = '\0';

    // Remove trailing whitespace, \r, \n
    int len = strlen(clean_cmd);
    while (len > 0 && (clean_cmd[len-1] == '\r' || clean_cmd[len-1] == '\n'
|| clean_cmd[len-1] == ' ')) {
        clean_cmd[--len] = '\0';
    }

    if (strcmp(clean_cmd, "LED_ON") == 0) {
        led_blink_en = 1;
        HAL_GPIO_WritePin(GPIOE,GPIO_PIN_4,GPIO_PIN_SET);
        Send_UART_Message("LED ON\r\n", 8);
    }
    else if (strcmp(clean_cmd, "LED_OFF") == 0) {
        led_blink_en = 0;
        HAL_GPIO_WritePin(GPIOE,GPIO_PIN_4,GPIO_PIN_RESET);
        Send_UART_Message("LED OFF\r\n", 9);
    }
    else if (strcmp(clean_cmd, "NOTIFY_ON") == 0) {
        Notification_Status = 1;
        Send_UART_Message("BLE Notifications enabled\r\n", 28);
    }
    else if (strcmp(clean_cmd, "NOTIFY_OFF") == 0) {
        Notification_Status = 0;
        Send_UART_Message("BLE Notifications disabled\r\n", 29);
    }
    else if (strcmp(clean_cmd, "STATUS") == 0) {
        char status_msg[80];
        GPIO_PinState pin_state = HAL_GPIO_ReadPin(GPIOE, GPIO_PIN_4);
       int len = sprintf(status_msg, "LED Flag: %s, GPIO State: %s, Notify:
%s\r\n",
                         led_blink_en ? "ON" : "OFF",
                         pin_state == GPIO_PIN_SET ? "HIGH" : "LOW",
                         Notification_Status ? "ON" : "OFF");
        Send_UART_Message(status_msg, len);
    }
    else if (strcmp(clean_cmd, "TEST_LED") == 0) {
        // Test LED directly
```

```
        HAL_GPIO_WritePin(GPIOE,GPIO_PIN_4,GPIO_PIN_SET);
        Send_UART_Message("LED test - ON\r\n", 15);
        HAL_Delay(1000);
        HAL_GPIO_WritePin(GPIOE,GPIO_PIN_4,GPIO_PIN_RESET);
        Send_UART_Message("LED test - OFF\r\n", 16);
    }
    else if (strcmp(clean_cmd, "SET_FLAG") == 0) {
        // Test setting flag directly
        led_blink_en = 1;
        Send_UART_Message("Flag set to 1 directly\r\n", 25);
    }
    else if (strcmp(clean_cmd, "CLEAR_FLAG") == 0) {
        // Test clearing flag directly
        led_blink_en = 0;
        Send_UART_Message("Flag cleared to 0 directly\r\n", 28);
    }
    else if (strcmp(clean_cmd, "PING") == 0) {
        Send_UART_Message("PONG\r\n", 6);
    }
    else if (strcmp(clean_cmd, "GET_KEY") == 0) {
        // Show current AES key
        uint8_t* key = Crypto_Get_Key();
        char key_msg[100];
        int len = sprintf(key_msg, "AES Key: ");
        Send_UART_Message(key_msg, len);
        HAL_Delay(50);

        for(int i = 0; i < 16; i++) {
            len = sprintf(key_msg, "%02X ", key[i]);
            Send_UART_Message(key_msg, len);
            HAL_Delay(10);
        }
        Send_UART_Message("\r\n", 2);
    }
    else if (strcmp(clean_cmd, "HELP") == 0) {
       Send_UART_Message("Commands: LED_ON, LED_OFF, PING, STATUS, GET_KEY,
HELP\r\n", 59);
    }
    else {
        char error_msg[50];
        int len = sprintf(error_msg, "Unknown command: %s\r\n", clean_cmd);
        Send_UART_Message(error_msg, len);
```

```
    }
}
/* USER CODE END 4 */

/**
  * @brief  This function is executed in case of error occurrence.
  * @retval None
  */
void Error_Handler(void)
{
  /* USER CODE BEGIN Error_Handler_Debug */
  /* User can add his own implementation to report the HAL error return
state */
  __disable_irq();
  while (1)
  {
  }
  /* USER CODE END Error_Handler_Debug */
}

#ifdef  USE_FULL_ASSERT
/**
  * @brief  Reports the name of the source file and the source line number
  *         where the assert_param error has occurred.
  * @param  file: pointer to the source file name
  * @param  line: assert_param error line source number
  * @retval None
  */
void assert_failed(uint8_t *file, uint32_t line)
{
  /* USER CODE BEGIN 6 */
  /* User can add his own implementation to report the file name and line
number,
     ex: printf("Wrong parameters value: file %s on line %d\r\n", file,
line) */
  /* USER CODE END 6 */
}
#endif /* USE_FULL_ASSERT */
```

ДОДАТОК Б 1

ЛІСТИНГ ПРОГРАМИ ЗАСТОСУНКУ

```
<!DOCTYPE html>
<html>
<head>
    <title>STM32 BLE Crypto Test</title>
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <style>
        body {
            font-family: 'Segoe UI', Arial, sans-serif;
            max-width: 600px;
            margin: 50px auto;
            padding: 20px;
            background: linear-gradient(135deg, #667eea 0%, #764ba2 100%);
            color: white;
        }
        .container {
            background: rgba(255, 255, 255, 0.1);
            backdrop-filter: blur(10px);
            border-radius: 20px;
            padding: 30px;
            box-shadow: 0 8px 32px 0 rgba(31, 38, 135, 0.37);
        }
        h1 {
            text-align: center;
            margin-bottom: 30px;
            font-size: 28px;
        }
        button {
            width: 100%;
            padding: 15px;
            margin: 10px 0;
            border: none;
            border-radius: 10px;
            font-size: 16px;
            font-weight: bold;
            cursor: pointer;
            transition: all 0.3s;
            background: white;
            color: #667eea;
        }
```

```
button:hover {
    transform: translateY(-2px);
    box-shadow: 0 5px 15px rgba(0,0,0,0.3);
}
button:disabled {
    opacity: 0.5;
    cursor: not-allowed;
}
.status {
    padding: 15px;
    margin: 10px 0;
    border-radius: 10px;
    background: rgba(255, 255, 255, 0.2);
    font-size: 14px;
}
.log {
    background: rgba(0, 0, 0, 0.3);
    padding: 15px;
    border-radius: 10px;
    max-height: 300px;
    overflow-y: auto;
    font-family: 'Courier New', monospace;
    font-size: 12px;
    margin-top: 20px;
}
.log-entry {
    margin: 5px 0;
    padding: 5px;
    border-left: 3px solid #4CAF50;
    padding-left: 10px;
}
input[type="text"] {
    width: 100%;
    padding: 12px;
    margin: 10px 0;
    border: none;
    border-radius: 10px;
    font-size: 14px;
    box-sizing: border-box;
}
.crypto-info {
    background: rgba(255, 255, 255, 0.15);
```

```
                padding: 10px;
                border-radius: 8px;
                margin: 10px 0;
                font-size: 12px;
            }
        </style>
    </head>
    <body>
        <div class="container">
            <h1>🔐 STM32 BLE Crypto</h1>

            <div style="margin-bottom: 20px;">
                <div style="background: rgba(255, 255, 255, 0.2); padding: 10px;
border-radius: 8px; margin-bottom: 10px;">
                    <strong>🔐 ECIES Encryption (P-256)</strong><br>
                        <small>Використовує ECC публічний/приватний ключ для
шифрування</small>
                </div>
            </div>

                <button id="connectBtn" onclick="connect()">Connect to
STM32</button>

            <div class="status" id="status">Not connected</div>

            <div class="crypto-info" id="cryptoInfo" style="display:none;">
                 <strong>Encryption:</strong> ECIES (Elliptic Curve Integrated
Encryption Scheme)<br>
                <strong>Curve:</strong> NIST P-256 (secp256r1)<br>
                <strong>Hardware:</strong> STM32WB55 PKA + AES1<br>
                <strong>Key Exchange:</strong> ECDH (автоматичний)<br>
                <strong>Padding:</strong> PKCS7 (auto)<br>
                    <strong>Status:</strong> <span id="keyStatus">Not
exchanged</span>
            </div>

            <input type="text" id="messageInput" placeholder="Enter message to
encrypt & send..." style="display:none;">
                <button id="sendBtn" onclick="sendEncrypted()"
style="display:none;">🔒 Encrypt & Send</button>
                <button id="sendPlainBtn" onclick="sendPlain()"
style="display:none;">📤 Send Plain</button>
```

```
                <button  onclick="sendCommand('LED_ON')"  style="display:none;"
id="ledOnBtn">💡 LED ON</button>
                <button  onclick="sendCommand('LED_OFF')"  style="display:none;"
id="ledOffBtn">💡 LED OFF</button>
                 <button  onclick="sendCommand('PING')"  style="display:none;"
id="pingBtn">📌 PING</button>

            <div class="log" id="log"></div>
        </div>

        <script>
            let device;
            let writeChar;
            let notifyChar;

            // Static key encryption: Hardcoded keys for encryption/decryption
            // JavaScript uses js_encrypt_key to encrypt messages to STM32
            // JavaScript uses js_decrypt_key to decrypt messages from STM32
            const js_encrypt_key = new Uint8Array([
                0x20, 0x21, 0x22, 0x23, 0x24, 0x25, 0x26, 0x27,
                0x28, 0x29, 0x2A, 0x2B, 0x2C, 0x2D, 0x2E, 0x2F,
                0x30, 0x31, 0x32, 0x33, 0x34, 0x35, 0x36, 0x37,
                0x38, 0x39, 0x3A, 0x3B, 0x3C, 0x3D, 0x3E, 0x3F
            ]);

            const js_decrypt_key = new Uint8Array([
            0x20, 0x41, 0x02, 0x03, 0x1A, 0x05, 0x06, 0x07,
        0x08, 0x09, 0x0A, 0x0B, 0x0C, 0x0D, 0x0E, 0x0F,
        0x10, 0x11, 0x12, 0x13, 0x14, 0x15, 0x16, 0x17,
        0x18, 0x19, 0x1A, 0x1D, 0x1C, 0x1D, 0x1E, 0x1F
            ]);

            // Legacy variables (kept for compatibility, but not used)
            let ourKeyPair = null;  // Our ECC keypair (static)
            let stm32PublicKey = null;  // STM32's public key (64 bytes: X + Y)
            let sharedSecret = null;  // ECDH shared secret (computed once)
            let aesKey = null;  // AES key (now using js_decrypt_key)
            let keyExchangeDone = true;  // Always true - using hardcoded keys

            // Generate ECC keypair for ECIES
            async function generateECKeypair() {
```

```
                try {
                    log('🔑 Generating ECC keypair (P-256)...');
                    const keyPair = await crypto.subtle.generateKey(
                        { name: "ECDH", namedCurve: "P-256" },
                        true,
                        ["deriveKey", "deriveBits"]
                    );
                    log('☑ ECC keypair generated!');
                    return keyPair;
                } catch(error) {
                    log('❌ Failed to generate keypair: ' + error);
                    return null;
                }
            }

            // Export public key (64 bytes: X + Y, without 0x04 prefix)
            async function exportPublicKey(keyPair) {
                const publicKey = await crypto.subtle.exportKey("raw", keyPair.publicKey);
                // Web Crypto exports as 65 bytes: [0x04, X (32), Y (32)]
                // We need 64 bytes: X (32) + Y (32)
                const publicKeyArray = new Uint8Array(publicKey);
                const result = new Uint8Array(64);
                result.set(publicKeyArray.slice(1, 33), 0);  // X coordinate
                result.set(publicKeyArray.slice(33, 65), 32); // Y coordinate
                return result;
            }

            // Import public key from STM32 (64 bytes: X + Y)
            async function importPublicKey(publicKeyBytes) {
                if(publicKeyBytes.length !== 64) {
                    log('❌ Invalid public key length: ' + publicKeyBytes.length + ' (expected 64)');
                    return null;
                }

                // Log received key for debugging
                let keyStr = 'Received key (first 16 bytes): ';
                for(let i = 0; i < Math.min(16, publicKeyBytes.length); i++) {
                    keyStr += publicKeyBytes[i].toString(16).padStart(2, '0').toUpperCase() + ' ';
                }
```

```
                log(keyStr);

                // Web Crypto expects 65 bytes: [0x04, X (32), Y (32)]
                 // Try both byte orders - STM32 PKA might return in different
endianness
                const fullKey = new Uint8Array(65);
                fullKey[0] = 0x04;  // Uncompressed point prefix

                // Extract coordinates
                const xCoord = publicKeyBytes.slice(0, 32);
                const yCoord = publicKeyBytes.slice(32, 64);

                // PKA_Memcpy_u32_to_u8 already converts to big-endian format
                // So STM32 should send keys in big-endian format
                // Try original format first (PKA already does conversion)
                fullKey.set(xCoord, 1);   // X coordinate (original - should be
big-endian)
                fullKey.set(yCoord, 33);  // Y coordinate (original - should be
big-endian)

                // Log full key for debugging
                let fullKeyStr = 'Full key (first 20 bytes): ';
                for(let i = 0; i < Math.min(20, fullKey.length); i++) {
                            fullKeyStr  +=  fullKey[i].toString(16).padStart(2,
'0').toUpperCase() + ' ';
                }
                log(fullKeyStr);

                 // Try original format first (PKA should already be in big-
endian)
                try {
                    const publicKey = await crypto.subtle.importKey(
                        "raw",
                        fullKey,
                        { name: "ECDH", namedCurve: "P-256" },
                        false,
                        []
                    );
                         log('☑ Public key imported successfully (original
format)!');
                    return publicKey;
                } catch(error1) {
```

```
                    log('⚠️ Failed with original format: ' + error1.message);
                    log('   Trying reversed bytes...');

                    // Try reversed bytes (in case PKA format is different)
                    const xReversed = new Uint8Array(32);
                    const yReversed = new Uint8Array(32);
                    for(let i = 0; i < 32; i++) {
                        xReversed[i] = xCoord[31 - i];
                        yReversed[i] = yCoord[31 - i];
                    }

                    fullKey.set(xReversed, 1);   // X coordinate (reversed)
                    fullKey.set(yReversed, 33);  // Y coordinate (reversed)

                    try {
                        const publicKey = await crypto.subtle.importKey(
                            "raw",
                            fullKey,
                            { name: "ECDH", namedCurve: "P-256" },
                            false,
                            []
                        );
                           log('☑ Public key imported successfully (reversed
bytes)!');
                        return publicKey;
                    } catch(error2) {
                       log('❌ Failed to import public key with both formats!');
                        log('   Original error: ' + error1.message);
                        log('   Reversed error: ' + error2.message);
                        log('   This might mean the key is not a valid point on
the curve.');

                        // Log key details for debugging
                        let xStr = 'X coord (first 8): ';
                        for(let i = 0; i < 8; i++) {
                                    xStr  +=  xCoord[i].toString(16).padStart(2,
'0').toUpperCase() + ' ';
                        }
                        log(xStr);

                        let xStr2 = 'X coord (last 8): ';
                        for(let i = 24; i < 32; i++) {
```

```
                                    xStr2 += xCoord[i].toString(16).padStart(2,
'0').toUpperCase() + ' ';
                            }
                            log(xStr2);

                            let yStr = 'Y coord (first 8): ';
                            for(let i = 0; i < 8; i++) {
                                    yStr += yCoord[i].toString(16).padStart(2,
'0').toUpperCase() + ' ';
                            }
                            log(yStr);

                            let yStr2 = 'Y coord (last 8): ';
                            for(let i = 24; i < 32; i++) {
                                    yStr2 += yCoord[i].toString(16).padStart(2,
'0').toUpperCase() + ' ';
                            }
                            log(yStr2);

                            // Try to validate point manually
                            log('🔍 Attempting manual validation...');
                             log('   Full X coord: ' + Array.from(xCoord).map(b =>
b.toString(16).padStart(2, '0').toUpperCase()).join(' '));
                             log('   Full Y coord: ' + Array.from(yCoord).map(b =>
b.toString(16).padStart(2, '0').toUpperCase()).join(' '));

                            return null;
                        }
                    }
                }

                // Static key encryption: Encrypt using hardcoded encrypt key
                async function staticEncrypt(text) {
                    try {
                        // Encrypt plaintext with hardcoded encrypt key
                        log('🔒 Encrypting with AES (hardcoded key)...');
                        const encrypted = await aesEncrypt(text, js_encrypt_key);

                        // Output: encrypted_data only (no ephemeral key)
                           log('☑ Encryption complete: ' + encrypted.length + '
bytes');
                        return encrypted;
```

```
                } catch(error) {
                        log('❌ Encryption error: ' + error.name + ' - ' +
error.message);
                    throw error;
                }
            }

            // Static key decryption: Decrypt using hardcoded decrypt key
            async function staticDecrypt(ciphertext) {
                if(ciphertext.length % 16 !== 0) {
                   throw new Error('Invalid ciphertext length (must be multiple
of 16)');
                }

                // Decrypt encrypted data with hardcoded decrypt key
                  const decryptedString = await aesDecrypt(ciphertext.buffer,
js_decrypt_key);

                // Convert string to Uint8Array
                const decryptedArray = new Uint8Array(decryptedString.length);
                for(let i = 0; i < decryptedString.length; i++) {
                    decryptedArray[i] = decryptedString.charCodeAt(i);
                }

                return decryptedArray;
            }

            // AES-128-CBC encryption using Web Crypto API (matches STM32 HAL)
            async function aesEncrypt(text, key) {
                // Convert string to bytes (ASCII, NOT UTF-8!)
                const data = new Uint8Array(text.length);
                for (let i = 0; i < text.length; i++) {
                    data[i] = text.charCodeAt(i);
                }

                log(`📝 Original: ${data.length} bytes`);

                // Manual PKCS7 padding
                const blockSize = 16;
                const paddingLength = blockSize - (data.length % blockSize);
                const paddedData = new Uint8Array(data.length + paddingLength);
                paddedData.set(data);
```

```
                for (let i = data.length; i < paddedData.length; i++) {
                    paddedData[i] = paddingLength;
                }

                        log(`📝 After padding: ${paddedData.length} bytes
(pad=${paddingLength})`);

                // IMPORTANT: Web Crypto API doesn't support ECB mode directly!
                // We'll use CBC with zero IV to emulate ECB for single block
                  // For proper ECB, we need to encrypt each 16-byte block
separately

                // Import key for AES
                const cryptoKey = await crypto.subtle.importKey(
                    'raw',
                    key,
                    { name: 'AES-CBC', length: 256 },
                    false,
                    ['encrypt']
                );

                // Zero IV (for ECB emulation with single block)
                const iv = new Uint8Array(16);

                // Encrypt
                const encrypted = await crypto.subtle.encrypt(
                    { name: 'AES-CBC', iv: iv },
                    cryptoKey,
                    paddedData
                );

                const encryptedArray = new Uint8Array(encrypted);
                log(`📦 Web Crypto output: ${encryptedArray.length} bytes`);

                // CRITICAL: Web Crypto API adds ANOTHER padding block!
                // We only want the FIRST block (16 bytes)
                const result = encryptedArray.slice(0, paddedData.length);
                log(`📦 Trimmed to: ${result.length} bytes`);

                return result;
            }
```

```
            // AES-128-CBC decryption using Web Crypto API (matches STM32 HAL)
            async function aesDecrypt(encrypted, key) {
                // Import key for AES
                const cryptoKey = await crypto.subtle.importKey(
                    'raw',
                    key,
                    { name: 'AES-CBC', length: 256 },
                    false,
                    ['decrypt']
                );

                // Zero IV (to match STM32)
                const iv = new Uint8Array(16);

                // Decrypt
                const decrypted = await crypto.subtle.decrypt(
                    { name: 'AES-CBC', iv: iv },
                    cryptoKey,
                    encrypted
                );

                // Remove PKCS7 padding manually
                const decryptedArray = new Uint8Array(decrypted);
               const paddingLength = decryptedArray[decryptedArray.length - 1];

                // Validate padding
                if(paddingLength > 0 && paddingLength <= 16) {
                            const unpaddedData = decryptedArray.slice(0,
decryptedArray.length - paddingLength);
                    return String.fromCharCode(...unpaddedData);
                }

                // No valid padding, return as-is
                return String.fromCharCode(...decryptedArray);
            }

            function log(message) {
                const logDiv = document.getElementById('log');
                const entry = document.createElement('div');
                entry.className = 'log-entry';
                const time = new Date().toLocaleTimeString();
                entry.textContent = `[${time}] ${message}`;
```

```
                logDiv.appendChild(entry);
                logDiv.scrollTop = logDiv.scrollHeight;
            }

            function updateStatus(message) {
                document.getElementById('status').textContent = message;
                log(message);
            }

            async function connect() {
                try {
                    updateStatus('Requesting device...');

                    device = await navigator.bluetooth.requestDevice({
                        filters: [{ namePrefix: 'P2P' }],
                                optionalServices:  ['0000fe40-cc7a-482a-984a-
7f2ed5b3e58f']
                    });

                    updateStatus('Connecting to ' + device.name + '...');
                    const server = await device.gatt.connect();

                    updateStatus('Getting service...');
                     const service = await server.getPrimaryService('0000fe40-
cc7a-482a-984a-7f2ed5b3e58f');

                    updateStatus('Getting characteristics...');
                    const characteristics = await service.getCharacteristics();

                    for(let char of characteristics) {
                        log('Char: ' + char.uuid);
                       if(char.uuid === '0000fe41-8e22-4541-9d4c-21edae82ed19')
{
                            writeChar = char;
                            log('✓ Write characteristic found!');
                        }
                       if(char.uuid === '0000fe42-8e22-4541-9d4c-21edae82ed19')
{
                            notifyChar = char;
                            await char.startNotifications();
                           char.addEventListener('characteristicvaluechanged',
handleNotification);
```

```
                        log('✓ Notify characteristic subscribed!');
                    }
                }

                // Generate ECC keypair for ECIES
                ourKeyPair = await generateECKeypair();

                if(!ourKeyPair) {
                    updateStatus('❌ Failed to generate keypair!');
                    return;
                }

                 // Export and send our public key to STM32 (64 bytes: X + Y)
                const ourPublicKey = await exportPublicKey(ourKeyPair);
                log('📤 Sending our public key (64 bytes)...');
                await writeChar.writeValue(ourPublicKey);
                log('☑ Public key sent! Waiting for STM32 public key...');

                updateStatus('✓✓✓ Connected & Encrypted! ✓✓✓');
                document.getElementById('connectBtn').style.display = 'none';
                document.getElementById('cryptoInfo').style.display = 'block';
                document.getElementById('messageInput').style.display = 'block';
                document.getElementById('sendBtn').style.display = 'block';
                document.getElementById('sendPlainBtn').style.display = 'block';
                document.getElementById('ledOnBtn').style.display = 'block';
                document.getElementById('ledOffBtn').style.display = 'block';
                document.getElementById('pingBtn').style.display = 'block';

            } catch(error) {
                updateStatus('ERROR: ' + error);
            }
        }

        async function handleNotification(event) {
            const value = event.target.value;
            const data = new Uint8Array(value.buffer);
```

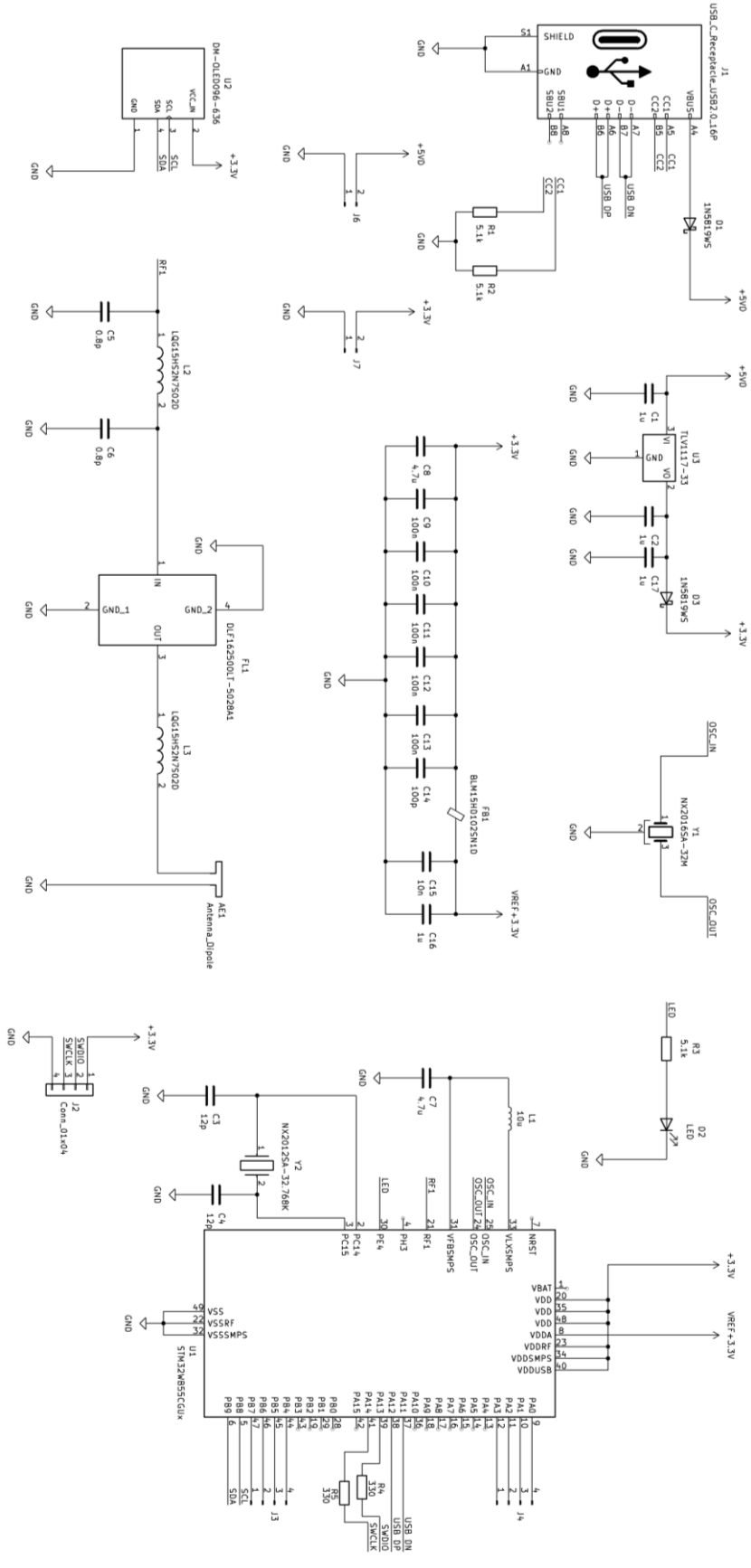
```
                // Log received bytes
                let recStr = 'Received (' + data.length + ' bytes): ';
                for(let i = 0; i < Math.min(data.length, 16); i++) {
                  recStr += data[i].toString(16).padStart(2, '0').toUpperCase()
+ ' ';
                }
                log(recStr);

                // Check if this is STM32's public key (64 bytes)
                if(!keyExchangeDone && data.length === 64) {
                    log('🔑 Received STM32 public key (64 bytes)!');

                    // Try to import the key
                    stm32PublicKey = await importPublicKey(data);

                    if(stm32PublicKey) {
                         // Compute shared secret using ECDH (once during key
exchange)
                        log('🔐 Computing shared secret...');
                        try {
                                        const  sharedSecretBits  =  await
crypto.subtle.deriveBits(
                            { name: "ECDH", public: stm32PublicKey },
                            ourKeyPair.privateKey,
                            256  // 256 bits = 32 bytes
                        );

                          // Derive AES key from shared secret (KDF - take
first 16 bytes)
                                        const  sharedSecretArray  =  new
Uint8Array(sharedSecretBits);
                        sharedSecret = sharedSecretArray;
                        aesKey = sharedSecretArray.slice(0, 16);

                        log('☑ Shared secret computed!');
                        log('☑ AES key derived!');
                        keyExchangeDone = true;
                         document.getElementById('keyStatus').textContent =
'☑ Exchanged';
                       log('☑ Key exchange complete! Encryption ready.');
                    } catch(error) {
```

```
                                log('❌ Failed to compute shared secret: ' +
error.message);
                                keyExchangeDone = false;
                            }
                        } else {
                            log('❌ Failed to import STM32 public key!');
                                        log('⚠️ Using hardcoded keys for
encryption/decryption...');
                            log('☑️ Encryption ready with hardcoded keys.');
                                document.getElementById('keyStatus').textContent =
'Hardcoded Keys';
                        }
                        return;
                    }

                    // Try to decrypt with static key
                    try {
                        if(data.length % 16 === 0 && data.length > 0 && data.length
!== 64) {
                            // Encrypted format: multiple of 16 bytes (but not 64,
which is public key)
                           log('🔓 Attempting decryption with js_decrypt_key...');
                            log('   Ciphertext (hex): ' + Array.from(data).slice(0,
16).map(b => b.toString(16).padStart(2, '0').toUpperCase()).join(' '));

                            const decrypted = await staticDecrypt(data);
                            const decoder = new TextDecoder();
                            const text = decoder.decode(decrypted);
                            log('☑️ Decrypted: "' + text + '"');
                        } else {
                            // Plain text or public key exchange
                            const decoder = new TextDecoder();
                            const text = decoder.decode(value);
                            log('📩 Plain text: ' + text);
                        }
                    } catch(error) {
                        // Decryption failed, show as plain text
                        log('❌ Decryption error: ' + error.message);
                        log('   This might mean:');
                        log('   1. Wrong key used for decryption');
                        log('   2. Ciphertext format mismatch');
```

```
                log('   3. Padding issue');
                const decoder = new TextDecoder();
                const text = decoder.decode(value);
                log('📩 Fallback: ' + text);
            }
        }

        async function sendEncrypted() {
            const message = document.getElementById('messageInput').value;
            if(!message) return;

            // Always ready - using hardcoded keys

            try {
                // Encrypt using static key
                const encrypted = await staticEncrypt(message);

                log('🔒 Encrypting: "' + message + '"');

                // Log encrypted bytes (first 16 bytes)
                let encStr = 'Encrypted (first 16 bytes): ';
                for(let i = 0; i < Math.min(encrypted.length, 16); i++) {
                    encStr += encrypted[i].toString(16).padStart(2,
'0').toUpperCase() + ' ';
                }
                log(encStr);

                log('📤 Sending ' + encrypted.length + ' bytes
(encrypted)...');

                await writeChar.writeValue(encrypted);
                log('✓ Encrypted message sent!');

                document.getElementById('messageInput').value = '';
            } catch(error) {
                log('❌ Send error: ' + error);
            }
        }

        async function sendPlain() {
            const message = document.getElementById('messageInput').value;
```

```
            if(!message) return;

            try {
                const encoder = new TextEncoder();
                const data = encoder.encode(message);

                log('📤 Sending plain: "' + message + '"');
                await writeChar.writeValue(data);
                log('✓ Plain message sent!');

                document.getElementById('messageInput').value = '';
            } catch(error) {
                log('❌ Send error: ' + error);
            }
        }

        async function sendCommand(cmd) {
            try {
                const encoder = new TextEncoder();
                const data = encoder.encode(cmd);

                log('📤 Command: ' + cmd);
                await writeChar.writeValue(data);

            } catch(error) {
                log('❌ Error: ' + error);
            }
        }
    </script>
</body>
</html>
```

ДОДАТОК В 1
ПРИНЦИПОВА СХЕМА



ДОДАТОК Г 1
ДРУКОВАНА ПЛАТА