МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ЗАПОРІЗЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

МАТЕМАТИЧНИЙ ФАКУЛЬТЕТ
Кафедра програмної інженерії

КВАЛІФІКАЦІЙНА РОБОТА МАГІСТРА

на тему: **«РОЗРОБКА АПАРАТНОГО МЕНЕДЖЕРА ПАРОЛІВ НА БАЗІ МІКРОКОНТРОЛЕРА З ПІДКЛЮЧЕННЯМ ЧЕРЕЗ USB»**

Виконав: студент 2 курсу, групи 8.1214-іпз-1
спеціальності 121 інженерія програмного забезпечення
(шифр і назва спеціальності)
освітньої програми інженерія програмного забезпечення
(назва освітньої програми)
В.В. Одайський
(ініціали та прізвище)
Керівник доцент кафедри програмної інженерії, доцент, к.т.н. Лимаренко Ю.О.
(посада, вчене звання, науковий ступінь, прізвище та ініціали)
Рецензент Директор ТОВ «Голден-Тім» Калін М.В.
(посада, вчене звання, науковий ступінь, прізвище та ініціали)

Запоріжжя – 2025

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

ЗАПОРІЗЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

Факультет математичний

Кафедра програмної інженерії

Рівень вищої освіти магістр

Спеціальність 121 інженерія програмного забезпечення
(шифр і назва)

Освітня програма інженерія програмного забезпечення

ЗАТВЕРДЖУЮ
Завідувач кафедри програмної інженерії, к.ф.-м.н., доцент

_____ Лісняк А.О.
(підпис)

“ ____ ” _____ 2025 р.

З А В Д А Н Н Я

НА КВАЛІФІКАЦІЙНУ РОБОТУ СТУДЕНТОВІ

Одайському Володимиру Вячеславовичу
(прізвище, ім’я та по-батькові)

1. Тема роботи Розробка апаратного менеджера паролів на базі мікроконтролера з підключенням через USB

керівник роботи Лимаренко Юлія Олексіївна, к.т.н., доцент
(прізвище, ім’я та по-батькові, науковий ступінь, вчене звання)

затверджені наказом ЗНУ від « 28 » лютого 2025 року № 326-с

2. Строк подання студентом роботи 05.12.2025 р.

3. Вихідні дані до роботи 1. Постановка задачі.
2. Перелік літератури.

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)
1. Аналіз предметної області.
2. Проєктування архітектури системи.
3. Розробка пристрою.

5. Перелік графічного матеріалу (з точним зазначенням обов’язкових креслень)
презентація за темою доповіді

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		Завдання видав	Завдання прийняв

7. Дата видачі завдання 07.03.2025 р.

К А Л Е Н Д А Р Н И Й П Л А Н

№	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1.	Розробка плану роботи.	21.03.2025	
2.	Збір вихідних даних.	18.04.2025	
3.	Обробка методичних та теоретичних	16.05.2025	
	джерел.		
4.	Розробка першого та другого розділу.	29.08.2025	
5.	Розробка третього розділу.	24.11.2025	
6.	Оформлення та нормоконтроль	01.12.2025	
	кваліфікаційної роботи магістра.		
7.	Захист кваліфікаційної роботи.	19.12.2025	

Студент _____ В.В. Одайський
(підпис) (ініціали та прізвище)

Керівник роботи _____ Ю.О. Лимаренко
(підпис) (ініціали та прізвище)

Нормоконтроль пройдено

Нормоконтролер _____ А.В. Столярова
(підпис) (ініціали та прізвище)

РЕФЕРАТ

Кваліфікаційна робота магістра «Розробка апаратного менеджера паролів на базі мікроконтролера з підключенням через USB»: 67 с., 42 рис., 4 табл., 23 джерела.

АВТОНОМНИЙ ПРИСТРІЙ, БЕЗПЕКА ДАНИХ, МЕНЕДЖЕР ПАРОЛІВ, МІКРОКОНТРОЛЕР, ШИФРУВАННЯ, USB-ІНТЕРФЕЙС.

Об’єкт дослідження – апаратний менеджер паролів та процеси захищеного зберігання і обробки чутливої інформації.

Мета роботи – розробити пристрій для безпечного зберігання та обробки паролів із використанням фізичної ізоляції, шифрування та апаратних механізмів підтвердження операцій.

Методи дослідження – аналіз архітектур апаратних криптографічних систем, проєктування апаратно-програмних комплексів, моделювання протоколів взаємодії через USB.

У роботі представлено розробку автономного менеджера паролів на базі мікроконтролера з підключенням через USB. Пристрій забезпечує зберігання конфіденційних даних, зокрема паролів, завдяки фізичній ізоляції від основного комп’ютера та обов’язковому підтвердженню операцій користувачем. Застосовано шифрування з використанням ключа, що зберігається на RFID-картці. Запропоноване рішення відрізняється від комерційних аналогів розширеними можливостями та підвищеним рівнем конфіденційності.

Актуальність проєкту зумовлена зростанням загроз інформаційній безпеці, зокрема ризиками перехоплення майстер-паролів шкідливим програмним забезпеченням. Створений пристрій дозволяє мінімізувати ці ризики шляхом фізичного відокремлення чутливих даних від операційної системи та мережевого середовища.

SUMMARY

Master’s qualifying paper “Development of a Microcontroller-based Hardware Password Manager with USB Connectivity”: 67 pages, 42 figures, 4 tables, 23 references.

AUTONOMOUS DEVICE, DATA SECURITY, ENCRYPTION, MICROCONTROLLER, PASSWORD MANAGER, USB INTERFACE.

The object of the study is a hardware password manager and the processes of secure storage and handling of sensitive information.

The aim of the study is to develop an autonomous hardware device for secure storage and processing of passwords using physical isolation, encryption, and hardware-based operation confirmation mechanisms.

The methods of research are analysis of hardware cryptographic system architectures, design of hardware-software complexes, modeling of USB interaction protocols.

The work presents the development of an autonomous hardware password manager based on a microcontroller with USB connectivity. The device ensures secure storage of confidential data, particularly passwords, thanks to physical isolation from the host computer and mandatory user confirmation of operations via a hardware button. Support for control USB transfers with a data stage has been implemented, an SD card has been integrated as a secure storage medium, and encryption is applied using a key stored on an RFID card. The proposed solution differs from commercial counterparts by its extensive abilities and an increased level of confidentiality.

The relevance of the project is driven by the growing threats to information security, particularly the risks of master-password interception by malicious software. The developed device helps minimize these risks by physically separating sensitive data from the operating system and network environment.

ЗМІСТ

Завдання на кваліфікаційну роботу 2
Реферат 4
Summary 5
Вступ 8
1 Аналіз предметної області та сучасних підходів 10
1.1 Проблематика зберігання паролів в умовах сучасної кіберзагрози 10
1.2 Визначення підходів до керування паролями 11
1.2.1 Управління паролями без допоміжних засобів 11
1.2.2 Програмні менеджери паролів 12
1.2.3 Апаратні менеджери паролів 14
1.3 Порівняльний аналіз підходів 16
1.4 Інструменти та стек технологій 16
1.4.1 Апаратна платформа 17
1.4.2 Збереження даних 17
1.4.3 Майстер-пароль 17
1.4.4 Протокол взаємодії з ПК 18
1.4.5 Програмне забезпечення пристрою 19
1.4.6 Клієнтське програмне забезпечення 19
2 Архітектура системи 21
2.1 Загальна архітектура системи 21
2.2 Формат даних та структура збереження 24
2.3 Алгоритми, що використовуються 28
2.3.1 Шифрування та дешифрування (AES-256-CBC) 28
2.3.2 Алгоритм FNV-1a (64-бітний варіант) 28
3 Реалізація системи 30
3.1 Комплектація пристрою та взаємодія з компонентами 30
3.2 Програмне забезпечення пристрою 41

3.3 Розробка фонової служби 55
3.4 Розширення браузера 60
Висновки 63
Перелік посилань 65

ВСТУП

Актуальність теми. У сучасному інформаційному суспільстві безпека особистих даних, зокрема паролів, є однією з ключових проблем, що потребують ефективних та надійних технічних рішень. Широке використання інтернет-сервісів, електронної комерції та цифрових платформ обумовлює необхідність зберігання великої кількості паролів, що підвищує ризики їх компрометації [1]. Відомо, що традиційні програмні менеджери паролів, попри свою популярність, часто залишаються вразливими до атак зловмисників через можливість перехоплення майстер-пароля за допомогою кейлогерів чи шкідливого програмного забезпечення [2].

Апаратні менеджери паролів на базі мікроконтролерів, що фізично ізолюють секретні дані від основного комп'ютера, є перспективним напрямом підвищення рівня захисту. Вони дозволяють мінімізувати вплив потенційних атак на операційну систему, забезпечуючи безпечне зберігання та обробку секретів [3]. Однак існуючі комерційні рішення, такі як, наприклад, YubiKey, мають певні обмеження та недоліки.

Актуальність роботи полягає в розробці апаратного менеджера паролів на базі мікроконтролера архітектури AVR, який через інтерфейс USB забезпечуватиме надійне, захищене та зручне зберігання і передачу чутливої інформації. Важливою особливістю розробки є шифрування даних за допомогою ключа, що зберігається на RFID-картці, а також забезпечення інтерактивної взаємодії з користувачем через апаратну кнопку підтвердження транзакцій. Таке рішення дає змогу не лише підвищити рівень безпеки, а й розширити функціонал пристрою.

Новизна розробки полягає у комплексному підході до безпечного зберігання паролів із фізичною ізоляцією від ПК, при якому пристрій є повністю автономним.

Практична значимість роботи полягає у можливості застосування розробленого апаратного менеджера паролів широким колом користувачів,

які прагнуть підвищити безпеку своїх облікових даних. Запропоноване рішення дозволяє знизити ризик компрометації майстер-пароля шляхом його фізичного відокремлення від основної операційної системи та передачі тільки необхідних для роботи даних.

Об’єкт дослідження. Об’єктом дослідження є система апаратно-програмного забезпечення для керування паролями. Предметом дослідження виступають апаратні інтерфейси, програмне забезпечення мікроконтролера, а також протокол обміну даними через USB, що забезпечує взаємодію пристрою з користувачем і зовнішніми системами.

Мета і завдання дослідження. Метою роботи є розробка безпечного, надійного та зручного у використанні апаратного менеджера паролів на основі мікроконтролера AVR із підтримкою USB-з’єднання та захищеним зберіганням інформації. Для досягнення цієї мети було поставлено наступні завдання:

- провести аналіз сучасних рішень у сфері апаратних і програмних менеджерів паролів;
- розробити архітектуру пристрою з урахуванням апаратних і програмних компонентів;
- реалізувати прототип із використанням стеку технологій LUFA для USB-інтерфейсу та SD-картки для зашифрованого зберігання даних;
- забезпечити інтерактивний інтерфейс для користувача, що включає дисплей та апаратну кнопку підтвердження операцій, а також вебінтерфейс на підключеному ПК;
- провести тестування функціональності та безпеки розробленого пристрою.

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА СУЧАСНИХ ПІДХОДІВ

1.1 Проблематика зберігання паролів в умовах сучасної кіберзагрози

Паролі є основним механізмом автентифікації користувачів у більшості сучасних інформаційних систем. Вони використовуються для доступу до електронної пошти, соціальних мереж, банківських сервісів, внутрішніх корпоративних ресурсів тощо. Водночас саме паролі залишаються найвразливішим елементом у системі захисту інформації [4]. За даними звітів з інформаційної безпеки, більшість інцидентів із витоком даних так чи інакше пов’язані з компрометацією облікових даних користувачів [5].

Одна з основних загроз – це викрадення пароля внаслідок зараження комп’ютера шкідливим програмним забезпеченням. Зокрема, кейлогери (keyloggers) можуть фіксувати всі натискання клавіш користувача, включно з введенням майстер-пароля до менеджера паролів [6]. Ще однією поширеною технікою є фішинг – обман користувача з метою змусити його ввести пароль на підробленому вебсайті. Крім того, навіть надійні програмні менеджери паролів можуть стати жертвою експлойтів до операційної системи або самої програми, що дозволяє зловмиснику отримати доступ до зашифрованої бази даних [7].

Серйозну загрозу становить повторне використання паролів на різних ресурсах. У разі компрометації одного акаунту зловмисник може легко отримати доступ до інших. Ще одна типова проблема – це слабкі паролі, які можна зламати за допомогою брутфорсу або словникової атаки [8].

На практиці багато користувачів зберігають паролі у браузерах, текстових файлах або навіть на папері. Такі підходи знижують рівень безпеки. З іншого боку, застосування складних паролів і регулярна їх зміна – вимога, яку складно реалізувати без автоматизованого інструменту керування паролями.

Таким чином, ефективне зберігання та захист паролів є критично важливим завданням, що вимагає збалансованого поєднання зручності для користувача та високого рівня безпеки. Особливої актуальності набувають апаратні рішення, що фізично ізолюють секретні дані від основної операційної системи, знижуючи ризик їх перехоплення.

1.2 Визначення підходів до керування паролями

1.2.1 Управління паролями без допоміжних засобів

Найпростішим і водночас найбільш вразливим підходом до зберігання паролів є їхнє запам’ятовування користувачем без використання спеціальних програм або апаратних засобів. Хоча такий спосіб не потребує технічних рішень і здається зручним у повсякденному використанні, він має низку критичних недоліків.

Насамперед, людська пам’ять має обмежену здатність зберігати велику кількість складних і унікальних паролів. Унаслідок цього користувачі часто вдаються до повторного використання одного й того ж пароля на різних ресурсах або створюють варіації на його основі [9]. Така поведінка суттєво знижує загальний рівень захисту: достатньо зламати лише один обліковий запис, щоб отримати доступ до інших.

Інша характерна проблема – схильність до створення простих, легко запам’ятовуваних паролів. На практиці це означає використання слів зі словника, імен, дат народження чи елементарних числових комбінацій. Подібні паролі вразливі до словникових атак і брутфорсу, що дає змогу зловмисникам зламати обліковий запис за короткий час.

Ще один спосіб уникнення забування – запис паролів на папері або в щоденнику. Хоча це виключає цифрові ризики, такий підхід створює нову вразливість: будь-хто, хто отримає фізичний доступ до записів, може легко

скомпрометувати облікові дані. Особливо це актуально в робочих або публічних просторах.

З огляду на вразливості, притаманні ручному управлінню паролями, постає потреба у комплексному рішенні, яке б забезпечувало як зручність для користувача, так і високий рівень безпеки. Сучасний менеджер паролів має поєднувати кілька рівнів захисту – як фізичних, так і програмних. Ідеальний інструмент повинен ізолювати секретні дані від основної операційної системи, застосовувати надійне шифрування, мати механізми захисту від несанкціонованого доступу.

1.2.2 Програмні менеджери паролів

Популярність програмних менеджерів паролів стрімко зкростає серед користувачів, що мають велику кількість облікових записів [10]. Вони забезпечують централізоване зберігання, генерацію, автозаповнення та синхронізацію паролів між пристроями. Ці застосунки можуть працювати локально або зберігати зашифровану базу даних у хмарі для зручності доступу з різних платформ. Розглянемо найвідоміші з них.

LastPass. LastPass є одним із найбільш відомих хмарних менеджерів паролів, що використовує модель централізованого зберігання зашифрованої бази на серверах компанії [11]. Розшифрування відбувається на стороні клієнта за майстер-паролем, який не передається у хмару. Сервіс підтримує інтеграцію з популярними браузерами, забезпечує синхронізацію між пристроями та пропонує вбудовані механізми додаткової автентифікації. Утім, робота в хмарній моделі передбачає прив’язку до серверної інфраструктури провайдера та залежність від її надійності й захищеності.

Bitwarden. Bitwarden позиціонується як відкрите та прозоре рішення з можливістю локального або самостійно розгорнутого серверного зберігання. Усі криптографічні операції виконуються на клієнті, а база даних передається

у вже зашифрованому вигляді [12]. Завдяки відкритому вихідному коду користувачі можуть перевіряти механізми шифрування та модифікувати рішення під власні потреби. Водночас робота Bitwarden залежить від платформи, на якій він використовується, та від захищеності операційної системи, у межах якої виконується застосунок.

1Password. 1Password є комерційним менеджером паролів, орієнтованим на приватних користувачів і корпоративні команди. Система пропонує зручні механізми автозаповнення, підтримує широкий спектр операційних систем і дозволяє гнучко керувати доступом усередині організацій [14]. Високий рівень реалізації поєднується з використанням закритої архітектури та централізованих рішень для зберігання та синхронізації даних, що робить сервіс залежним від моделі підписки.

KeePass / KeePassXC. KeePass і його кросплатформова модифікація KeePassXC є локальними менеджерами паролів із відкритим кодом, які зберігають інформацію у файлі бази *.kdbx*. Для доступу може використовуватися майстер-пароль, ключовий файл або їх комбінація [15]. Відсутність хмарної залежності забезпечує повну автономність і дає змогу контролювати весь життєвий цикл зберігання даних. Однак синхронізація між пристроями потребує окремих механізмів, які користувач повинен організувати вручну.

Dashlane / NordPass. Dashlane та NordPass представляють собою сучасні комерційні рішення, що орієнтуються на широке коло кінцевих користувачів. Вони поєднують браузерні розширення, мобільні клієнти та сервіси моніторингу безпеки, надаючи комплексний підхід до керування обліковими даними [13]. Оскільки обидві системи функціонують на основі закритих протоколів і централізованих серверів, їхня робота передбачає залежність від хмарної інфраструктури та моделі підписки.

Попри широкий вибір, програмні рішення мають спільну вразливість: усі вони працюють у межах операційної системи користувача, яка може бути скомпрометована. Наприклад, шкідливе ПЗ може перехопити натиснення

клавіш або отримати доступ до розшифрованої бази після автентифікації користувача. Крім того, введення майстер-пароля в будь-яку програму не гарантує його конфіденційність у разі зараження системи.

Таким чином, хоча програмні менеджери значно полегшують керування паролями, вони залишаються вразливими до низки атак, що робить доцільним пошук альтернативних, менш залежних від ОС рішень.

1.2.3 Апаратні менеджери паролів

Апаратні менеджери паролів – це окремі фізичні пристрої, призначені для захищеного зберігання облікових даних із мінімальною залежністю від основної обчислювальної системи користувача. Основною перевагою таких рішень є фізична ізоляція чутливої інформації, що знижує ризики, пов’язані з зараженням комп’ютера шкідливим ПЗ, зокрема кейлогерами або троянами.

YubiKey. YubiKey – це апаратний ключ автентифікації, що реалізує такі стандарти, як OTP, U2F та FIDO2 [16]. Він є ефективним у сценаріях двофакторної та безпарольної автентифікації, але його функціональність обмежена цими сценаріями. На відміну від розробленого пристрою, YubiKey не підтримує зберігання довільних даних, не може виступати повноцінним менеджером паролів і не має можливості автозаповнення через браузерне розширення. Виконання будь-яких дій з паролями залежить від сторонніх сервісів, а введення облікових даних не автоматизовано. Таким чином, YubiKey забезпечує лише функцію токена автентифікації та не може конкурувати з комплексним функціоналом автономного сховища.

OnlyKey. OnlyKey поєднує автентифікаційні функції з обмеженим сховищем облікових записів. Хоча він здатен «надрукувати» логін і пароль у текстові поля, це не є повноцінною інтеграцією з браузером і потребує ручної взаємодії користувача [17]. Кількість записів суворо обмежена, а зберігання довільних даних не передбачено: OnlyKey працює лише зі структурованими слотами для облікових даних. Крім того, для розблокування потрібно

вводити PIN на самому пристрої, що ускладнює використання. У підсумку OnlyKey забезпечує базові апаратні функції, але поступається запропонованому пристрою у гнучкості та зручності.

Nitrokey. Nitrokey – сімейство пристроїв з відкритим кодом, які можуть виконувати функції автентифікації, керування ключами й, у деяких моделях, зберігання зашифрованих файлів [18]. Однак навіть у моделях Nitrokey Storage обсяг доступного сховища суттєво менший за той, що може бути реалізований у спеціалізованому пристрої з великим масивом пам’яті. Nitrokey не має власних засобів роботи з браузером: інтеграції для автоматичного заповнення паролів відсутні. Усі операції з паролями покладаються на інші застосунки або механізми. Крім того, керування PIN-кодами і ключами потребує додаткових налаштувань та технічної компетентності, що робить Nitrokey менш зручним у щоденному використанні. За сукупністю показників пристрій залишається ефективним криптографічним токеном, але не забезпечує високий рівень інтеграції, функціональності та автономності.

Попри потужний функціонал, більшість існуючих апаратних менеджерів обмежені жорстко заданими протоколами або мають високий поріг входу для зміни логіки роботи пристрою. Крім того, такі рішення часто мають закриту або частково відкриту архітектуру, що не дозволяє легко адаптувати їх під специфічні потреби користувача [19]. До того ж, більшість перелічених пристроїв потребує введення майстер пароля на підключеному ПК для використання деяких функцій.

Розроблюваний у цій роботі пристрій спрямований на заповнення цієї ніші: він пропонує відкриту архітектуру, що легко модифікується, дозволяє зберігати довільні зашифровані дані на картці пам’яті, використовує механізми фізичного підтвердження транзакцій, і має низьку собівартість. Такий підхід забезпечує високу гнучкість і розширюваність під будь-які сценарії використання – від зберігання паролів до обміну ключами або передачі конфіденційної інформації, ізолюючи її від потенційно вразливої ОС користувача.

1.3 Порівняльний аналіз підходів

Кожен з підходів має свої переваги та обмеження. Програмні рішення мають зручний інтерфейс, розвинену функціональність і часто синхронізуються між пристроями, однак є вразливими до атак із боку шкідливого ПЗ, особливо кейлогерів або шпигунських модулів, здатних перехопити майстер-пароль.

Апаратні ключі типу YubiKey знижують ці ризики, але є обмеженими у сценаріях використання: вони не завжди дозволяють зберігати довільну інформацію, можуть бути складними у використанні, та погано інтегруються з іншими програмами. Nitrokey у свою чергу забезпечує вищу гнучкість і підтримує зберігання файлів, проте має високу вартість і обмежену доступність, що ускладнює його застосування на масовому рівні або в освітніх / хобі-проєктах.

Розроблювана система вирізняється поєднанням низької вартості, фізичної безпеки, відкритої архітектури та гнучкості, що робить її привабливою як для ентузіастів, так і для користувачів, які прагнуть підвищити безпеку зберігання особистих даних без залежності від сторонніх платформ.

1.4 Інструменти та стек технологій

Проєкт апаратного менеджера паролів потребує застосування як апаратних, так і програмних засобів для посягнення найвищого рівня безпеки і зручності. У цьому підрозділі розглянуто технологічний стек, інструменти та обґрунтування їх вибору для реалізації системи, що забезпечує надійне зберігання й передачу конфіденційних даних.

1.4.1 Апаратна платформа

Основою для розробки пристрою обрано мікроконтролер AVR (AT90USB1286), який використовується, зокрема, в Teensy++ 2.0. Головними критеріями вибору стали:

– наявність вбудованого USB-контролера, що дозволяє реалізувати протокол взаємодії з комп’ютером без потреби в аппаратній емуляції;
– підтримка інтерфейсів SPI та UART, що необхідні для підключення зовнішніх модулів – таких як SD-карта та RFID-зчитувач;
– широка підтримка в середовищах розробки та активна спільнота, що полегшує налагодження.

1.4.2 Збереження даних

Для зберігання зашифрованих паролів використовується стандартна MicroSD-карта, підключена через SPI-інтерфейс. Такий підхід забезпечує:

– незалежність зберігання від мікроконтролера;
– модульність;
– можливість масштабування обсягу пам’яті та створення резервних копій.

1.4.3 Майстер-пароль

Для зберігання криптографічного ключа передбачено використання RFID-картки. Її фізичне вилучення з пристрою унеможливлює доступ до даних навіть у випадку компрометації пристрою. Це реалізує принцип розділення доступу: ключ зберігається окремо від даних.

1.4.4 Протокол взаємодії з ПК

Взаємодія між пристроєм та персональним комп’ютером здійснюється за допомогою інтерфейсу USB (Universal Serial Bus) – універсальної послідовної шини, яка є стандартом для підключення периферійних пристроїв до комп’ютерів. USB забезпечує підтримку обміну даними між хостом (ПК) та пристроєм.

Архітектура USB ґрунтується на суворій моделі з розподілом ролей між хостом і пристроєм: хост завжди ініціює обмін даними, а пристрій лише відповідає на запити [20]. Передача даних у USB організована через ендоінти.

Ендпоінт – це логічна точка входу/виходу даних у пристрої. Кожен ендпоінт має визначений напрямок (IN або OUT), тип передачі (керування, переривання, ізохронна або пакетна) та номер. Ендпоінти групуються у інтерфейси, які представляють функціональні складові пристрою (наприклад, клавіатура, миша, аудіо тощо). Пристрій може мати кілька інтерфейсів, а також кілька альтернативних конфігурацій інтерфейсів для різних режимів роботи [21].

Для комунікації пристрою з комп’ютером використовується контрольний тип передачі – базовий і універсальний механізм USB, який передбачає три фази [22]:

– Setup – ініціалізація запиту від хоста;
– Data (необов’язкова) – передача або отримання даних;
– Status – підтвердження виконання запиту.

Контрольні трансфери, незважаючи на низьку пропускну здатність, є придатними для обміну короткими та критичними даними, такими як команди управління, конфігурація пристрою чи передача конфіденційної інформації. Вони завжди доступні на ендоінті 0, зарезервованому для керування пристроєм [21].

Завдяки такому підходу, можливо реалізувати простий протокол взаємодії, в якому комп’ютер надсилає чітко визначені запити, а пристрій

відповідає лише у межах дозволених дій. Це дозволяє ефективно обмежити поверхню атаки, контролювати логіку обробки команд та забезпечити ізоляцію чутливих операцій – зокрема, тих, що пов’язані з обробкою паролів або ключів.

У разі потреби розширення функціональності або підвищення швидкості обміну, пристрій може додатково реалізовувати інші типи передач – наприклад, інтервальні чи масові (bulk) [23], однак саме контрольні трансфери залишаються основою безпечної та стабільної комунікації.

1.4.5 Програмне забезпечення пристрою

Прошивка мікроконтролера розробляється з використанням LUFA (Lightweight USB Framework for AVRs) – потужного фреймворку для реалізації USB-девайсів на мікроконтролерах AVR. LUFA забезпечує:

– підтримку різних класів USB пристроїв;
– низькорівневий контроль передачі даних;
– абстракцію від конкретних чіпів;
– зручні інструменти компіляції;
– приклади реалізацій і документацію, що спрощує розробку.

Фреймворк написаний на мові програмування C.

1.4.6 Клієнтське програмне забезпечення

На стороні комп’ютера планується створити мінімалістичний клієнт для надсилання запитів до пристрою, отримання та обробки відповідей. Він буде реалізований у вигляді HTTP сервіса, що працює у режимі служби. Цей сервіс надсилатиме контрольні трансфери до пристрою, і оброблятиме відповіді. Будь-яка програма на комп’ютері може використовувати дану

службу для взаємодії з пристроєм. Одним з сценаріїв використання служби стане вебінтерфейс.

2 АРХІТЕКТУРА СИСТЕМИ

2.1 Загальна архітектура системи

Апаратний менеджер паролів – це автономний пристрій на основі мікроконтролера AVR, який поєднує фізичні та програмні механізми безпеки і підключається до персонального комп’ютера через USB. Основна мета системи – забезпечити зберігання та передачу чутливих даних таким чином, щоб операційна система комп’ютера ніколи не мала прямого доступу до вмісту сховища.

Фундаментальний принцип побудови системи полягає в наявності чіткої фізичної межі між середовищем комп’ютера та середовищем, де фактично розміщуються та обробляються секретні дані. Комп’ютер працює лише в рамках суворо визначеного протоколу обміну, не маючи можливості керувати внутрішньою структурою пам’яті чи криптографічними операціями. Це означає, що навіть повністю скомпрометоване програмне середовище на ПК не здатне ініціювати довільний несанкціонований доступ до секретів, а лише надсилати обмежений набір запитів, які додатково фільтруються та перевіряються самим пристроєм.

Критичним елементом системи є картка пам’яті, на якій зберігаються зашифровані дані. Вона є центральним вузлом зберігання, однак її фізична втрата або вилучення не дорівнює автоматичній компрометації: інформація на картці захищена криптографічно, ключовий матеріал розподілений між кількома компонентами, а успішна атака потребує одночасного збігу ряду умов (наявність апаратного пристрою, необхідних ключів, знання протоколу, коректного сценарію взаємодії тощо). Таким чином, навіть при фізичному доступі до носія даних відтворення повної логіки системи та отримання секретів залишається суттєво ускладненим.

Окремо використовується RFID-картка, яка виконує роль фізичного ключа для розблокування пристрою. Вона є важливим, але не єдиним

фактором безпеки: її наявності недостатньо без відповідної апаратної платформи, коректно реалізованого протоколу та інших параметрів, що перевіряються під час взаємодії.

З огляду на те, що середовищу комп'ютера (операційна система, драйвери, прикладне ПЗ) не можна повністю довіряти, протокол обміну між ПК та пристроєм навмисно зроблено мінімалістичним та жорстко обмеженим. Комп'ютер не має змоги ініціювати довільні операції з ключами, модифікувати сховище в обхід встановлених правил або отримати необмежений доступ до вмісту пам'яті – він може лише формувати запити у фіксованому форматі, а рішення щодо їхнього виконання приймається виключно на стороні апаратного менеджера.

Апаратна частина системи реалізована на базі мікроконтролера AVR, який відповідає за зберігання, шифрування, дешифрування та контрольовану передачу даних. Пристрій має доступ до картки пам'яті, де паролі зберігаються у зашифрованому вигляді, та використовує RFID-картку як фактор аутентифікації для розблокування. Комунікація з комп'ютером здійснюється через USB з використанням контрольних трансферів відповідно до специфікацій протоколу. На корпусі розміщено дисплей (наприклад, OLED 128×64) для індикації стану системи та параметрів запитів, а також кнопку ручного підтвердження операцій, що унеможливлює автоматичне виконання потенційно небезпечних дій без участі користувача.

Пристрій надає доступ до обмеженого ряду операцій. Фактично він є базою даних типу ключ-значення, де всі значення зберігаються у зашифрованому вигляді. Перелік доступних операцій наведено в таблицях 1 та 2.

Таблиця 1 – Операції з передачею даних до хоста

Операція	**Структура даних**	**Опис**
SEND_SECRET	128 байт ключ + дані	Створює новий запис для даного ключа.
RESET_DEVICE	0 байт	Скидає дані на пристрої, форматує заголовки
SET_KEY	128 байт ключ	Встановлює значення регістра KEY
SET_ADDRESS	8 байт адреса	Встановлює значення регістра ADDRESS

Таблиця 2 – Операції з передачею даних до пристроя

Операція	**Структура даних**	**Опис**
READ_SECRET	розшифроване значення	Зчитує розшифроване значення секрету за даним ключем
SECRET_META	512 байт заголовку запису	Зчитує заголовок запису за ключем у регістрі KEY
GET_OP_STATUS	1 байт	Зчитує статус останньої операції
GET_DEV_INFO	64 байти	Статична інформація про налаштування пристроя
GET_ADDR_INDEX	послідовні блоки індексу	Зчитує весь індекс заголовків
GET_BLOCK	512 байт	Зчитує значення блоку за адресою з регістра ADDRESS

Через специфіку USB-протоколу, а саме однонаправленість передачі даних, деякі операції потребують підготовки у декілька етапів. Так, аби отримати розшифроване значення за ключем, спочатку потрібно викликати операцію SET_KEY, підготувавши регістр на пристрої, після чого можна викликати READ_SECRET, що зчитає і розшифрує значення, використовуючи цей регістр. Через цю особливість також виникла необхідність у команді GET_OP_STATUS, що надає можливість отримати статус останньої операції. Через такий підхід для унеможливлення маніпуляцій регістрами, що могли б ввести користувача у оману, як-от зміна значення регістра вже після ініціації запиту пароля, було додане обмеження, яке забороняє запити на зміну регістрів у той час, коли вже існує транзакція, що очікує на підтвердження користувачем.

Клієнтська частина працює на комп’ютері та реалізована у вигляді фонової служби, яка запускається автоматично під час старту операційної системи. Ця служба відкриває локальний HTTP API, до якого можуть звертатися браузери або інші локальні застосунки. Саме через цей API здійснюється взаємодія між пристроєм і користувацькими програмами.

На рівні інтеграції з браузером передбачена можливість встановлення розширення, яке автоматично взаємодіє з локальним API. Коли користувач відкриває сайт, розширення може ініціювати запит на авторизацію. Пристрій

отримує домен сайту, звіряє його з наявними записами в пам’яті та виводить відповідну інформацію на екран. Користувач самостійно приймає рішення про підтвердження або відхилення запиту, натискаючи кнопку на пристрої.

Архітектура системи ґрунтується на кількох базових принципах. Ізоляція полягає в тому, що доступ до секретних даних можливий лише через фізичний пристрій, що істотно знижує ризик витоку інформації через зловмисне програмне забезпечення на комп’ютері. Фізична аутентифікація передбачає необхідність одночасної присутності RFID-картки та активної дії з боку користувача (натискання кнопки) для виконання транзакції. Захист від фішингу реалізовано шляхом прив’язки кожного пароля до конкретного доменного імені: навіть якщо зловмисний сайт імітує вигляд справжнього, запит на авторизацію буде відхилено через невідповідність переданого домену до заголовку Origin у надходящому запиті. Цей заголовок встановлюється браузером, і його неможливо підробити з контексту вебдодатку. Звісно, перевірка заголовків не захистить від подібних спроб фішингу, що виконуються поза межими браузера, таких, як, наприклад, десктопний застосунок, про те на цей випадок користувач все ще матиме можливість розпізнати зловмисну поведінку, оскільки дані про запит відображаються на LED-дисплеї, та потребують підтверження кнопкою.

2.2 Формат даних та структура збереження

Для забезпечення зберігання та структурованого доступу до чутливої інформації, апаратний менеджер паролів використовує власний формат збереження даних на SD-картці. Пристрій розроблено для використання саме High-Capacity SD карток (SDHC). Ці картки мають від двох гігабайт пам’яті, та використовують блоки розміром 512 байт. Проєкт розрахований на використання з карткою об’ємом 8 гігабайт.

Оскільки для реалізації поставлених задач зовсім не обов’язково мати повноцінну файлову систему, було розроблено спеціальний формат даних,

що працює напряму з блоками пам’яті. Вимогами до розробленого формату є можливість швидкого створення нових записів, перелічення існуючих, отримання і видалення записів за ключем, а також швидкого форматування. Формат розбиває карту пам’яті на 4 сектори (див. рис. 1):

– заголовок;
– індекс секретів;
– заголовки секретів;
– блоки даних.

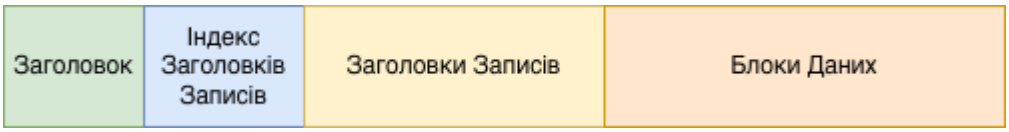


Рисунок 1 – Розташування секторів у пам’яті

Перший блок пам’яті містить заголовок пристрою. Заголовок складається з 4 статичних байтів 0x424F4C42, версії пристрою, кількості записів на пристрої, вказівника на перший вільній блок в секції даних (див. табл. 3).

Таблиця 3 – Формат запису в заголовку

Поле	**Розмір**	**Тип**	**Опис**
blob_magic	4 байта	uint32 (LE)	BLOB
version	4 байта	uint32 (LE)	Версія пристрою
num_records	4 байта	uint32 (LE)	Кількість записів
storage_head	8 байт	uint64 (LE)	вказівник на перший вільний блок даних

Основною ідеєю для забезпечення швидкого доступу до даних за ключем є принцип хеш-таблиці. Ключ перетворюється на числове значення хеш-функцією, числове значення приводиться до значення в межах простору адрес заголовків записів. У разі колізії, коли отримана адреса вже зайнята, буде взято наступний вільний блок, тож в результаті буде обрано вільний

блок з найменшою адресою, що більша або дорівнює отриманій адресі. У цей блок буде записано заголовок запису. Заголовок, в свою чергу, має формат, вказаний у таблиці 4.

Таблиця 4 – Формат заголовку запису

Поле	**Тип**	**Опис**
status	1 байт	Маркер зайнятості
key	128 байт	Ключ
data_length	8 байт	Довжина запису (зашифрований вигляд)
next_block	8 байт	адреса початку даних
reserved	152 байт	не використовується

Заголовок запису починається з байта-індикатора. Якщо цей байт не дорівнює 87, цей блок гарантовано вільний. Якщо цей байт має значення 87, блок може бути вільним. Для виконання умови швидкого форматування пристрій не потребує щоб вільні блоки даних встановлювали коректне значення цього байта. Якщо байт вказує на можливу зайнятість блоку, для гарантованої перевірки його статусу використовується індекс заголовків записів. Далі заголовок містить сам ключ. При пошуку потрібного блоку за хеш-сумою виконується послідовне порівняння кожного блоку, аж допоки не буде досягнуто блок, що має шуканий ключ, або вільний блок, що означатиме, що шуканого запису не існує. Далі заголовок містить довжину запису, що необхідна при його зчитуванні. Далі міститься адреса самого запису.

Посередині між заголовком пристрою і заголовками записів знаходиться відносно невеликий сектор індексу заголовків записів. Саме завдяки цьому сектору можливо гарантовано визначити, чи є блок зайнятим. Цей сектор відображає статус кожного блока в секторі заголовків записів, по одному біту на блок. Статус блока під певним номером в секторі заголовків записів відображено бітом з відповідним порядковим номером в секторі

індекса. Таким чином, одного блока в індексі вистачить, аби відобразити статус 4096 блоків. Також цей сектор відіграє важливу роль у функції переліку створених записів на пристрої. Достатньо зчитати лише невелику кількість блоків щоб дізнатись адреси всіх заголовків записів, після чого їх можна зчитувати індивідуально.

За сектором заголовків записів знаходиться сектор даних. Це великий сектор, який містить безпосередньо зашифровані блоки даних, на які і вказують заголовки. Ці блоки виділяються послідовно, від меншої адреси до більшої. Адреса поточної голови секції зберігається в заголовку пристрою. У разі, якщо запис завеликий для одного блоку, можливий запис у декілька послідовних блоків.

Визначення розмірів логічних секторів сховища виконується на основі фіксованих параметрів: загальної місткості носія, розміру блока, цільової максимальної кількості записів та заданого коефіцієнта заповнення. На першому етапі оцінюється необхідна кількість блоків для розміщення заголовків записів. Для цього цільову кількість підтримуваних записів поділено на коефіцієнт заповнення, менший за одиницю, що забезпечує наявність запасу в секторі заголовків і дає змогу уникнути роботи хеш-таблиці в режимі граничного заповнення. Одержане значення визначає розмір області, призначеної для зберігання заголовків записів.

Далі розраховується розмір індекса заголовків записів, який реалізовано у формі бітової карти. Кожному блоку заголовка відповідає один біт, що відображає його стан (зайнятий/вільний). Один блок індекса містить фіксовану кількість бітів, здатних описати стани великої множини блоків заголовків, тому загальну кількість блоків заголовків поділено на цю місткість та отриманий результат округлено у бік збільшення. Це гарантує достатню кількість бітів для позначення стану кожного блока заголовків.

Після цього послідовно визначаються межі всіх логічних областей пам’яті. Перший блок носія відводиться під службовий заголовок пристрою, що містить параметри конфігурації та поточний стан. Безпосередньо за ним

розташовується компактний сектор індекса заголовків записів, розмір якого визначено на етапі розрахунку бітової карти. Наступною областю є сектор заголовків записів, розмір якого відповідає заздалегідь обчисленій кількості блоків з урахуванням коефіцієнта заповнення. Завершальною є область даних, яка починається одразу після сектору заголовків і займає весь залишковий простір носія; у цій області послідовно розміщуються зашифровані фрагменти корисних даних.

2.3 Алгоритми, що використовуються

2.3.1 Шифрування та дешифрування (AES-256-CBC)

Всі конфіденційні дані шифруються з використанням симетричного алгоритму AES у режимі CBC (Cipher Block Chaining) з довжиною ключа 256 біт. Ключ шифрування зберігається на RFID-картці користувача і ніколи не передається на хост-пристрій. Для блоків даних IV генерується з використанням адреси блоку. Доповнення до кратної довжини не є необхідним, оскільки шифруються цілі блоки, що мають довжину 512 байт.

Цей підхід дозволяє досягти високого рівня безпеки навіть у разі компрометації носія (SD-картки), оскільки зловмисник без RFID-картки не має доступу до ключа.

2.3.2 Алгоритм FNV-1a (64-бітний варіант)

Для формування унікальних ідентифікаторів ключів у системі використовується хеш-алгоритм FNV-1a у 64-бітній модифікації. Даний алгоритм було обрано через його високу швидкість роботи, простоту реалізації та достатню рівномірність розподілу хеш-значень, що робить його ефективним для адресації даних у зовнішніх накопичувачах.

Алгоритм FNV-1a працює за принципом ітеративної хеш-функції, де кожен байт вхідних даних послідовно впливає на кінцевий результат. У процесі обчислення початкове значення встановлюється у фіксовану константу – FNV offset basis. Далі алгоритм для кожного байта виконує операцію XOR з поточним значенням хеша, після чого результат множиться на фіксоване просте число. Наявність простих арифметичних операцій робить FNV-1a придатним для використання на мікроконтролерах із обмеженими ресурсами, зокрема на пристроях, що працюють у режимі реального часу.

У розробленому пристрої FNV-1a 64 використовується для обчислення хешу ключа, який надалі застосовується як адреса при визначенні місця розташування відповідного блоку даних на SD-картці. Завдяки рівномірному розподілу хеш-значень вхідні ключі розміщуються по файловому простору без концентрації в окремих ділянках, що знижує кількість колізій і забезпечує ефективний доступ до даних.

3 РЕАЛІЗАЦІЯ СИСТЕМИ

3.1 Комплектація пристрою та взаємодія з компонентами

Мікроконтроллер та USB-стек. Для реалізації пристрою було обрано мікроконтроллер на базі чіпа AT90USB1286. Цей контролер має вбудовану підтримку протоколів USB, SPI, I2C; 128K Flash пам’яті, 8K оперативної пам’яті, 4K EEPROM пам’яті; 8 аналогових виходів, 46 цифрових I/O-пінів, 9 PWM-виходів (див. рис. 2).

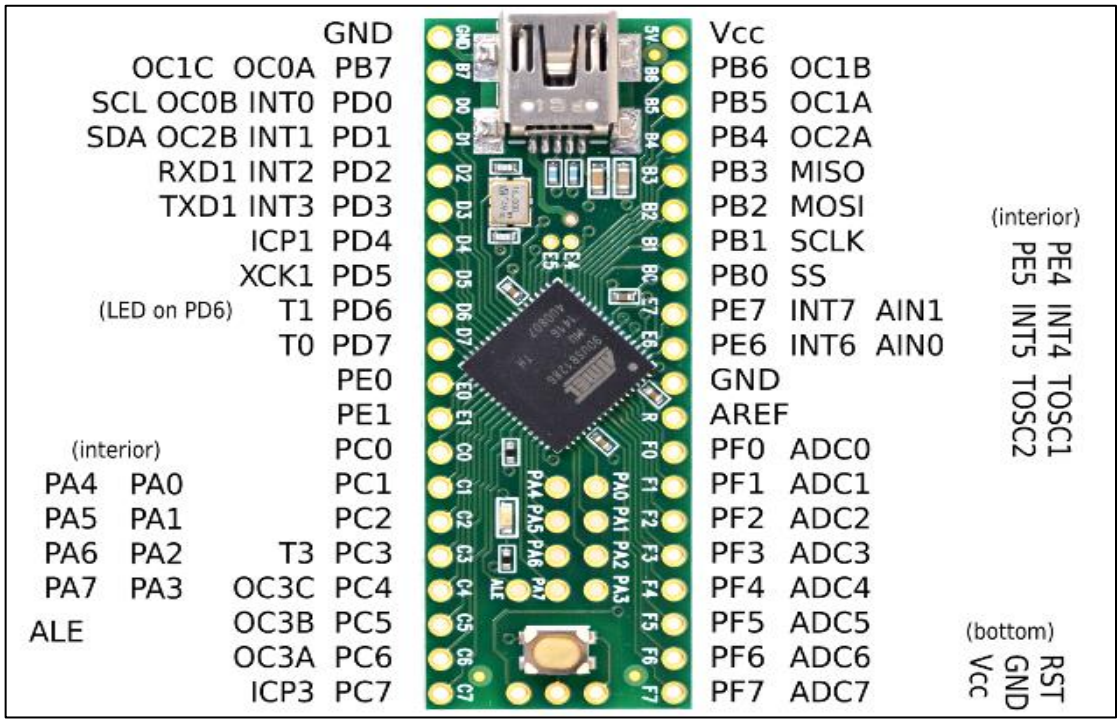


Рисунок 2 – Структура мікроконтроллера Teensy++ 2.0

Для спрощення роботи з протоколом USB було використано бібліотеку LUFA (Lightweight USB for AVRs). Ця бібліотека надає необхідні засоби для коректної енумерації пристрою, визначення дескрипторів, обробки всіх видів трансферів. Одним з ключових видів передачі даних у USB є контрольні трансфери, які призначені для конфігурації пристрою, обміну командами та невеликими блоками даних. Вони завжди проходять через нульовий ендпоінт

(Endpoint 0) і мають строго визначену структуру, що складається з трьох основних стадій: Setup, Data та Status.

Стадія Setup завжди виконується першою і містить спеціальний пакет розміром 8 байт, який описує запит (див. рис. 3). Цей пакет включає поля: bmRequestType, що визначає напрямок передачі (IN або OUT), тип запиту (стандартний, класовий або власний), та одержувач (пристрій, інтерфейс або ендпоінт); bRequest, що вказує конкретну команду; wValue, який може містити додаткову інформацію для запиту; wIndex, що зазвичай використовується для індексації інтерфейсів або ендпоінтів; і wLength, який визначає кількість байт у наступній стадії даних, якщо вона присутня.

Offset	Field	Size	Value	Description
0	*bmRequestType*	1	Bitmap	Characteristics of request: D7: Data transfer direction 0 = Host-to-device 1 = Device-to-host D6...5: Type 0 = Standard 1 = Class 2 = Vendor 3 = Reserved D4...0: Recipient 0 = Device 1 = Interface 2 = Endpoint 3 = Other 4...31 = Reserved
1	*bRequest*	1	Value	Specific request (refer to Table 9-3)
2	*wValue*	2	Value	Word-sized field that varies according to request
4	*wIndex*	2	Index or Offset	Word-sized field that varies according to request; typically used to pass an index or offset
6	*wLength*	2	Count	Number of bytes to transfer if there is a Data stage

Рисунок 3 – Структура пакета setup

Стадія Data є опціональною і присутня лише для запитів, які передбачають передачу або прийом даних. Вона може бути спрямована від

хоста до пристрою (OUT) або від пристрою до хоста (IN). Дані передаються блоками, розмір яких не перевищує максимальний розмір пакета ендпоінта. Кожен пакет містить заголовок протоколу USB та самі дані. Ця стадія дозволяє передавати як прості команди, так і невеликі масиви інформації.

Стадія Status завершує трансфер і служить для підтвердження його успішності. Для цього використовується пакет зворотного зв'язку: якщо була стадія IN, пристрій відправляє "пустий" OUT-пакет, сигналізуючи хосту, що дані прийняті; якщо OUT – хост надсилає "пустий" IN-пакет, підтверджуючи отримання даних пристроєм. Таким чином, кожен контрольний трансфер забезпечує гарантовану доставку і правильну синхронізацію між хостом і пристроєм.

Пристрій визначено як CDC-ACM (Communications Device Class) пристрій. Це означає, що на комп'ютері він відображається як серійний порт. Дескриптор пристрою наведено на рисунку 4.

```
const USB_Descriptor_Device_t PROGMEM DeviceDescriptor =
{
        .Header                 = {.Size = sizeof(USB_Descriptor_Device_t), .Type = DTYPE_Device},

        .USBSpecification       = VERSION_BCD(1,1,0),
        .Class                  = USB_CSCP_IADDeviceClass,
        .SubClass               = USB_CSCP_IADDeviceSubclass,
        .Protocol               = USB_CSCP_IADDeviceProtocol,

        .Endpoint0Size          = FIXED_CONTROL_ENDPOINT_SIZE,

        .VendorID               = 0x03EB,
        .ProductID              = 0x204B,
        .ReleaseNumber          = VERSION_BCD(0,0,1),

        .ManufacturerStrIndex   = STRING_ID_Manufacturer,
        .ProductStrIndex        = STRING_ID_Product,
        .SerialNumStrIndex      = USE_INTERNAL_SERIAL,

        .NumberOfConfigurations = FIXED_NUM_CONFIGURATIONS
};
```

Рисунок 4 – Визначення дескриптора пристрою

Використання пристрою як серійного порта спрощує процес відлагодження, дозволяючи зручно обмінюватись повідомленнями. Для

роботи основного функціоналу пристрою, в свою чергу, використовується кінцева точка за нульовою адресою, яка здатна обробляти контрольні трансфери.

У конфігураційному дескрипторі пристрій декларує одну конфігурацію з двома інтерфейсами, об’єднаними через IAD (Interface Association Descriptor) у єдиний логічний CDC-ACM блок. Фрагмент дескриптора, що визначає перший інтерфейс, наведено на рисунку 5.

```
.CDC1_CCI_Interface =
{
        .Header                 = {.Size = sizeof(USB_Descriptor_Interface_t), .Type = DTYPE_Interface},

        .InterfaceNumber        = INTERFACE_ID_CDC_CCI,
        .AlternateSetting       = 0,

        .TotalEndpoints         = 1,

        .Class                  = CDC_CSCP_CDCClass,
        .SubClass               = CDC_CSCP_ACMSubclass,
        .Protocol               = CDC_CSCP_ATCommandProtocol,

        .InterfaceStrIndex      = NO_DESCRIPTOR
},
```

Рисунок 5 – Конфігурація першого інтерфейсу

Перший інтерфейс є керуючим (Communication Class Interface) і використовується для сигналізації стану та налаштування параметрів віртуального послідовного порта.

Другий інтерфейс є інтерфейсом даних (Data Class Interface) і містить два bulk-ендпоінти: один у напрямку OUT для приймання даних від хоста і один у напрямку IN для передавання даних до хоста (див. рис. 6).

Саме ці два ендпоінти утворюють двонапрямний канал обміну, який операційна система відображає у вигляді байтового потоку стандартного послідовного порта (див. рис. 7).

Конфігураційний дескриптор також містить набір функціональних дескрипторів CDC (Header, ACM, Union), якими пристрій явно вказує, що підтримує модель ACM та пов’язує керуючий інтерфейс із інтерфейсом

даних. Завдяки цьому хост може автоматично застосувати стандартний драйвер USB-CDC і коректно інтерпретувати як службовий трафік, так і користувацькі дані, що передаються через bulk-ендпоінти.

```
.CDC1_DCI_Interface =
{
        .Header                 = {.Size = sizeof(USB_Descriptor_Interface_t), .Type = DTYPE_Interface},

        .InterfaceNumber        = INTERFACE_ID_CDC_DCI,
        .AlternateSetting       = 0,

        .TotalEndpoints         = 2,

        .Class                  = CDC_CSCP_CDCDataClass,
        .SubClass               = CDC_CSCP_NoDataSubclass,
        .Protocol               = CDC_CSCP_NoDataProtocol,

        .InterfaceStrIndex      = NO_DESCRIPTOR
},
```

Рисунок 6 – Конфігурація другого інтерфейсу

```
.CDC1_DataOutEndpoint =
{
        .Header                 = {.Size = sizeof(USB_Descriptor_Endpoint_t), .Type = DTYPE_Endpoint},

        .EndpointAddress        = CDC1_RX_EPADDR,
        .Attributes             = (EP_TYPE_BULK | ENDPOINT_ATTR_NO_SYNC | ENDPOINT_USAGE_DATA),
        .EndpointSize           = CDC_TXRX_EPSIZE,
        .PollingIntervalMS      = 0x05
},

.CDC1_DataInEndpoint =
{
        .Header                 = {.Size = sizeof(USB_Descriptor_Endpoint_t), .Type = DTYPE_Endpoint},

        .EndpointAddress        = CDC1_TX_EPADDR,
        .Attributes             = (EP_TYPE_BULK | ENDPOINT_ATTR_NO_SYNC | ENDPOINT_USAGE_DATA),
        .EndpointSize           = CDC_TXRX_EPSIZE,
        .PollingIntervalMS      = 0x05
},
```

Рисунок 7 – Конфугурація двох ендпоінтів даних

LUFA бере на себе відповідальність за обробку основних контрольних трансферів, таких як робота з дескрипторами. Користувачу залишається визначити EVENT_USB_Device_ControlRequest обробник подій, з метою обробки нестандартних (vendor-specific) трансферів,

EVENT_USB_Device_ConfigurationChanged обробник для оновлення налаштувань ендпоінтів, та вчасно викликати USB_USBTask, щоб фреймворк мав можливість обробити вхідні дані.

Взаємодія з SD-Карткою. Для збереження даних у проєкті використовується SD-картка з високою місткістю (SDHC). Ці картки можуть працювати у двох режимах – SD Bus та SPI Bus. Оскільки контроллер підтримує SPI на апаратному рівні, було реалізовано саме цей протокол.

Протокол SPI для SD-картки передбачає обмін даними між контролером та карткою через послідовний інтерфейс із чотирма лініями: MOSI (Master Out, Slave In), MISO (Master In, Slave Out), SCK (Serial Clock) та CS (Chip Select). Схему підключень для обох протоколів наведено на рисунку 8.

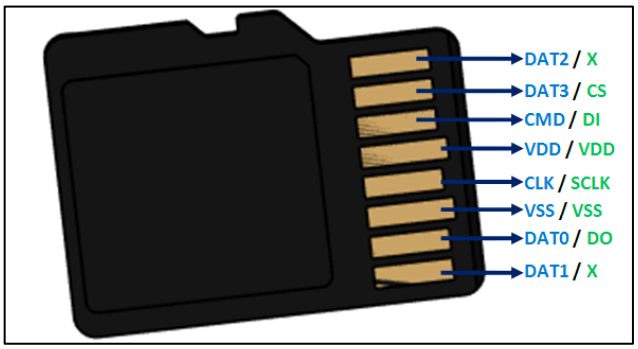


Рисунок 8 – Схема підключень SD катрки в SD-Bus та SPI режимах

Лінія MOSI (Master Out, Slave In**)** використовується для передачі даних від контролера до SD-картки: усі команди, аргументи, блоки даних для запису надсилаються саме по цій лінії. Лінія MISO (Master In, Slave Out) працює у зворотному напрямку – по ній картка повертає статусні байти, відповіді на команди та самі дані під час операції читання. Лінія SCK (Serial Clock) – це тактовий сигнал, який формує лише майстер (контролер). Картка «знімає» біти з MOSI та виставляє біти на MISO синхронно з фронтами цього тактового сигналу: без SCK обмін даними не відбувається взагалі.

Сигнал CS (Chip Select) використовується для вибору конкретного підлеглого пристрою на шині SPI. Коли CS для SD-картки знаходиться в

активному низькому стані, картка слухає лінію MOSI та відповідає на команди через MISO. Коли CS переведено у високий рівень, картка ігнорує трафік на шині, а лінія MISO переходить у високий імпеданс, дозволяючи під’єднати до тієї ж шини інші пристрої. У типовому сценарії роботи з SD-карткою контролер спочатку опускає CS, генерує тактові імпульси на SCK і послідовно передає байти по MOSI, а картка відповідає по MISO, поки CS залишається активним.

У режимі SPI контролер виступає майстром, а картка – підлеглим. Кожна операція відбувається через надсилання командних пакетів (Command Frame), які складаються з одного байта коду команди, чотирьох байт аргументу та одного байта контрольної суми (CRC). Після отримання команди картка відповідає певним статусним кодом (R1, R2 або R3/R7), який сигналізує про стан готовності, помилки або додаткову інформацію.

Процес ініціалізації картки починається з встановлення лінії CS у високий стан та надсилання серії тактових імпульсів із MOSI для приведення картки у SPI-режим. Далі контролер переводить лінію CS у низький стан і надсилає команду CMD0 (GO_IDLE_STATE) для переведення картки у стан очікування, після чого слідують команди CMD8 та ACMD41 для визначення типу картки та завершення ініціалізації. Картка відповідає, коли готова до передачі даних.

Для читання блоків даних контролер надсилає команду CMD17 (READ_SINGLE_BLOCK) або CMD18 (READ_MULTIPLE_BLOCK), вказуючи адресу блоку. Картка відповідає статусом готовності та після цього надсилає блоки даних із маркером початку блоку, самі дані та контрольну суму CRC.

Для запису контролер використовує команди CMD24 (WRITE_SINGLE_BLOCK) або CMD25 (WRITE_MULTIPLE_BLOCK). Після отримання команди картка сигналізує готовність до прийому, після чого контролер надсилає блок даних разом із маркером початку та CRC. Картка підтверджує отримання спеціальним кодом відповіді Data Response, після чого процес можна повторювати для наступних блоків.

Через те, що використаний модуль для підключення SD карток не відпускає лінію MISO при високому рівні CS, було прийнято рішення використовувати окрему програмну шину SPI, залишивши апаратну для потреб інших модулів. Основні функції програмного SPI протоколу реалізовані, як показано на рисунку 9.

```
void SD_SendByte(uint8_t b) {
    for (uint8_t i = 0; i < 8; i++) {
        if (b & 0x80) {
            SD_MOSI_HIGH();
        } else {
            SD_MOSI_LOW();
        }
        b <<= 1;
        sd_spi_delay();
        SD_SCK_HIGH();
        sd_spi_delay();
        SD_SCK_LOW();
    }
}
uint8_t SD_ReadByte() {
    uint8_t b = 0;
    SD_MOSI_HIGH();
    for (uint8_t i = 0; i < 8; i++) {
        b <<= 1;
        sd_spi_delay();
        SD_SCK_HIGH();
        if (SD_MISO_READ()) {
            b |= 1;
        }
        sd_spi_delay();
        SD_SCK_LOW();
    }
    return b;
}
```

Рисунок 9 – Програмна реалізація читання і запису по SPI

Взаємодія з OLED-Дисплеєм. Пристрій має невеликий OLED дисплей (див. рис. 10), що відображає поточний стан системи. Комунікація з дисплеєм відбувається за протоколом I2C. Цей інтерфейс використовує всього дві лінії – SCL (Serial Clock) та SDA (Serial Data). У ролі мастера виступає мікроконтролер, який формує тактові імпульси на SCL і керує передачею даних по SDA.

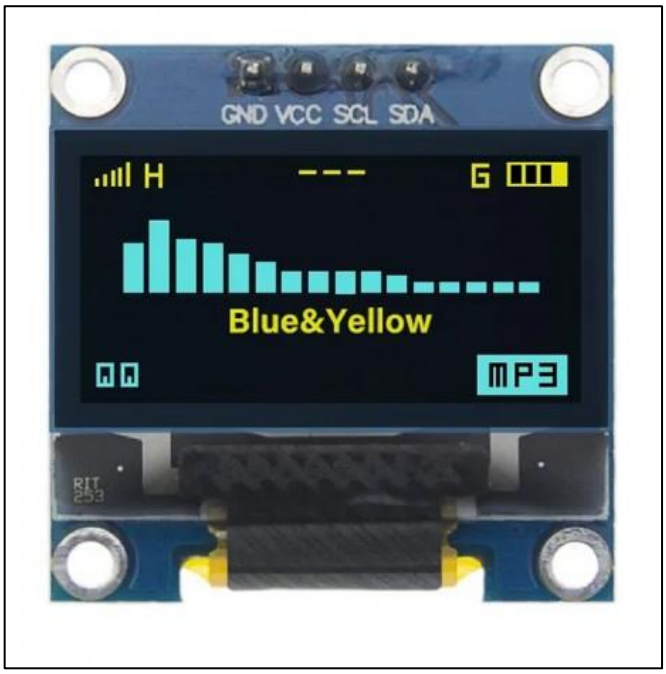


Рисунок 10 – Модуль OLED дисплея

Обмін із дисплеєм відбувається шляхом надсилання послідовностей байтів, що містять як команди керування, так і графічні дані, які відповідають вмісту буфера екрану. Спочатку контролер ініціалізує дисплей, надсилаючи набір стандартних команд (вмикання живлення матриці, вибір режиму адресації, контрасту тощо), після чого періодично оновлює його вміст – наприклад, виводить поточний статус пристрою, інформаційні повідомлення або значення основних параметрів системи.

Взаємодія з бузером. У пристрої застосовується п’єзоелектричний звуковий випромінювач (бузер), який виконує роль акустичного індикатора стану системи та подій, що потребують уваги або взаємодії користувача (підтвердження дії, сигнал помилки, запит введення тощо). Бузер має два виводи: лінію живлення та загальну шину (GND). Один із виводів підключається до загальної землі пристрою, інший – до лінії керування, реалізованої на GPIO-виводі мікроконтролера.

Керування бузером здійснюється шляхом подачі на GPIO-вивід відповідного логічного рівня або формування ШІМ імпульсів.

Кнопка керування. Для підтвердження операцій, що потребують явної згоди або дії користувача, у пристрої передбачено окрему тактову кнопку.

Вона використовується як основний елемент керування для запуску критичних або потенційно необоротних операцій (наприклад, підтвердження запису даних, виконання службових команд тощо), зменшуючи ризик випадкового або несанкціонованого спрацювання.

Електрично кнопка підключена до загальної шини (GND) та одного з GPIO-виводів мікроконтролера. На GPIO-виводі використовується внутрішній підтягувальний резистор до живлення, завдяки чому у відпущеному стані на вході фіксується логічна одиниця, а при натисканні кнопки вхід з’єднується із землею, формуючи стабільний логічний нуль. Обробка натискань здійснюється у програмному забезпеченні.

Взаємодія з NFC модулем. Для зчитування ключа шифрування з RFID картки було використано модуль PN532 (див. рис. 11). Як і модуль SD картки, цей модуль під’єднано по протоколу SPI. Модуль призначений для роботи з безконтактними картками стандарту ISO/IEC 14443A, включно з картками сімейства MIFARE Classic. Обмін по SPI базується на власному фреймовому протоколі PN532, який визначає структуру службових полів, довжини та контрольних сум кожного пакета.

Передача відбувається в синхронному режимі, а кожна команда має бути обгорнута у фрейм, який містить преамбулу, стартовий код, довжину, тип передавача, корисні дані, контрольну суму та постамбулу.

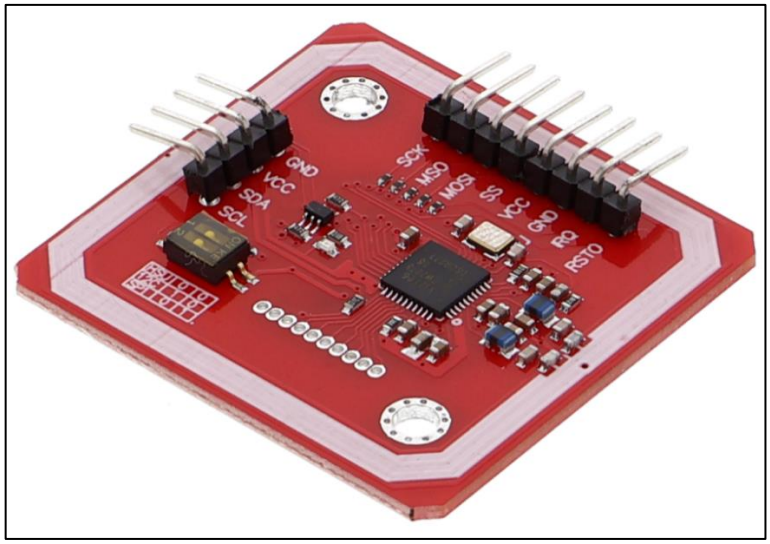


Рисунок 11 – Модуль PN532

Перед прийомом або передачею даних мікроконтролер зчитує внутрішній статус PN532, який сигналізує про готовність чіпа. Поширені стани – “busy”, коли контролер виконує попередню команду, “ready” при готовності до передачі відповіді, та “data available”, що означає наявність сформованого пакета, який можна зчитати. Така система станів забезпечує синхронізований обмін і запобігає втраті даних.

Робота PN532 в режимі зчитувача починається з ініціалізації, під час якої виконується команда конфігурації SAM, що переводить чіп у режим Reader/Writer. Після цього модуль здійснює опитування поля та пошук картки командою InListPassiveTarget, у результаті чого повертається ідентифікатор картки, параметри протоколу та інша службова інформація. Коли картку виявлено, система переходить до етапу аутентифікації.

Для MIFARE Classic доступ до кожного блоку захищено ключем A або ключем B, і перед читанням сектору необхідно надіслати команду аутентифікації через InDataExchange з відповідним типом ключа і UID картки. Лише після успішної аутентифікації контролер дозволяє виконувати операції читання.

Процес читання блока MIFARE Classic здійснюється командою MIFARE Read, також переданою через InDataExchange. У команду передається номер блоку, після чого PN532 повертає відповідь, що містить 16 байтів даних. Для кожного нового сектору аутентифікацію необхідно повторювати, оскільки картка ізольовано контролює доступ до кожної області пам’яті.

Структура пам’яті MIFARE Classic складається із секторів, кожен з яких містить три звичайні блоки та один службовий блок, де зберігаються ключі та бітові структури доступу. Це зумовлює необхідність точного дотримання послідовності команд і підтверджень під час роботи з контролером.

У прєкті модуль виконує всього дві операції – читання та запис 32 байтів ключа шифрування.

3.2 Програмне забезпечення пристрою

В момент вмикання пристрою виконується його ініціалізація. Ініціалізується USB стек та глобальний стан пристрою, структура якого наведена на рисунку 12.

```
struct PM_AppState {
    bool requested;
    bool pressed;
    bool signaled;
    bool unlocked;
    uint64_t ttl;
    uint8_t encryption_key[ENC_KEY_LENGTH];
    uint8_t registry_key[KEY_LENGTH];
    uint64_t registry_address;
};
```

Рисунок 12 – Структура глобального стану пристрою

Стан пристрою відслідковує такі параметри системи як наявність ключа шифрування, стан підтвердження операції, стан нотифікації користувача, наявність запиту в обробці, та значення регістрів.

Також при ініціалізації налаштовуються шини SPI, I2C та підключені до них пристрої, SD-картка педеводиться в режим SPI, OLED дисплей переходить в активний режим і отримує інструкції з конфігурації, встановлюються режими роботи GPIO-виводів, виконується зчитування заголовочного блоку даних (нульового блоку) SD-картки. Після завершення ініціалізації, програма входить в безкінечний цикл (див. рис. 13).

```
for (;;) {
    TickAppState(&AppState);
    CDC_Device_USBTask(&VirtualSerial1_CDC_Interface);
    USB_USBTask();
}
```

Рисунок 13 – Головний цикл програми

У цьому циклі постійно оновлюється глобальний стан (див. рис. 14), стан віртуального послідовного інтерфейсу та всієї USB-шини.

Під час оновлення глобального стану відбувається оновлення перефірії. Перевіряється стан кнопки, і в разі, коли є непідтверджений запит, він переходить у підтверджений стан, а пристрій генерує аудіо сигнал. При наявності запиту, про який ще не було повідомлено користувачу, пристрій генерує подвійний аудіосигнал, і виставляє відповідний флаг. Також відбувається зворотній відлік часу до інвалідації запиту, коли лічильник досягає нуля, стан запиту буде скинуто.

```
void TickAppState(struct PM_AppState* state) {
    if (state->requested && !state->signaled) {
        state->signaled = 1;
        BeepTwice();
    }
    if (IsButtonPressed() && state->requested) {
        state->pressed = 1;
        state->requested = 0;
        BeepOnce();
    }
    if (state->requested) {
        if (--state->ttl <= 0) {
            ResetStatus(state);
        }
    }
    if (!state->unlocked) {
        pn532_begin();
        if (pn532_read_512_bytes(state->encryption_key)) {
            state->unlocked = true;
        }
    }
}
```

Рисунок 14 – Логіка оновлення стану пристроя

Основна логіка пристрою реалізована в обробнику контрольних трансферів EVENT_USB_Device_ControlRequest, що має структуру, описану на рисунку 15. Цей обробник викликається фреймворком LUFA в момент ініціації хостом контрольного трансферу, одразу після того, як було зчитано стадію Setup (зазвичай в контексті переривання). Обробник має реалізувати роботу з даними, зокрема читання або запис, в залежності від напрямку транзакції. Всередині обробника доцільно використовувати глобальну змінну USB_ControlRequest аби отримати доступ безпосередньо до вмісту Setup пакету.

```
void EVENT_USB_Device_ControlRequest(void) {
    bool handled = false;
    if ((USB_ControlRequest.bmRequestType & CONTROL_REQTYPE_TYPE) == REQTYPE_VENDOR &&
        (USB_ControlRequest.bmRequestType & CONTROL_REQTYPE_DIRECTION) == REQDIR_HOSTTODEVICE
    ) {
        switch (USB_ControlRequest.bRequest) {
            // ...
        }
    }
    if ((USB_ControlRequest.bmRequestType & CONTROL_REQTYPE_TYPE) == REQTYPE_VENDOR &&
        (USB_ControlRequest.bmRequestType & CONTROL_REQTYPE_DIRECTION) == REQDIR_DEVICETOHOST
    ) {
        switch (USB_ControlRequest.bRequest) {
            // ...
        }
    }
    if (!handled) {
        CDC_Device_ProcessControlRequest(&VirtualSerial1_CDC_Interface);
    }
}
```

Рисунок 15 – Структура обробника контрольних трансферів

Реалізація READ_BLOCK команди. Для забезпечення прямого доступу до блоків даних на носії реалізовано команду READ_BLOCK. Здебільшого ця команда необхідна для відлагодження пристрою. Вона дозволяє реалізувати функцію перегляду вихідних бінарних даних, проте лише для блоків в незашифрованих секторах пам’яті. Такий функціонал є особливо корисним при пошуку проблем файлової системи пристрою. При отриманні команди виконується зчитування вмісту одного блоку за адресою з регістра ADDRESS, що має бути заздалегіть встановлений командою SET_ADDRESS. Код обробника запиту наведено на рисунку 16.

```
Endpoint_ClearSETUP();
#ifndef DEBUG
if (REGISTRY_ADDRESS >= PACKED_BLOCKS_START) {
    OperationStatus.code = ERROR_OUT_OF_PUBLIC_RANGE;
    DISCARD_OUT_BUFFER(USB_ControlRequest.wLength);
}
else
#endif
{
    uint8_t secret[BLOCK_SIZE];
    SD_ReadBlock(REGISTRY_ADDRESS, secret);
    while (!Endpoint_IsINReady()){}
    Endpoint_Write_Control_Stream_LE(secret, BLOCK_SIZE);
}
Endpoint_ClearStatusStage();
```

Рисунок 16 – Обробник запиту READ_ BLOCK

У разі спроби зчитування даних з адресного простору блоків даних, програма встановлює код помилки, скидає дані з вхідного буфера, та завершує обробку запиту. При умові коректності запиту, програма зчитує 512 байт блоку з картки пам’яті в буфер, після чого записує їх у вихідний буфер пристрою, звідки вони будуть зчитані хостом.

Реалізація GET_STATE команди. Операція GET_STATE забезпечує передачу до хоста внутрішнього стану модуля. Пристрій формує структуру DevState, до якої заносяться поточні значення полів AppState: стан блокування, наявність транзакції, зворотній відлік до інвалідації транзакції, а також значення регістрів. Сформована структура передається хосту. Код обробника запиту наведено на рисунку 17. Ці дані можна використати для розробки користувацького інтерфейсу на комп’ютері, до якого підключений пристрій. Команда не потребує авторизації чи підтвердження, тому інформація про пристрій може бути візуальзована одразу при підключенні.

```
Endpoint_ClearSETUP();
while (!Endpoint_IsINReady()){}
struct DevState state;
state.unlocked = AppState.unlocked ? 1 : 0;
state.requested = AppState.requested;
state.pressed = AppState.pressed;
state.signaled = AppState.signaled;
state.ttl = AppState.ttl;
memcpy(state.registry_key, AppState.registry_key, KEY_LENGTH);
state.registry_address = AppState.registry_address;
Endpoint_Write_Control_Stream_LE(&state, sizeof(state));
Endpoint_ClearStatusStage();
```

Рисунок 17 – Обробник запиту GET_STATE

Реалізація READ_SECRET команди. Для читання значення запису за ключем, було реалізовано команду READ_SECRET. Ця команда потребує заздалегідь підготовленого значення ключа секруту у регістрі KEY. Перед початком читання вона перевіряє, чи вже було ініційовано запит на підтвердження дії користувачем. Якщо ні, запит буде створено, якщо запит не було підтверджено, читання не відбудеться, та статус операції буде встановлено на ERROR_WAITING_INTERACTION, а якщо запит відбувся, і

був підтверджений, – запис буде зчитано, розшифровано і надіслано на хост. Передбачено, що ця команда буде виконуватись повторювано в очикувані того, як або спливе час на підтвердження, або запит буде підтверджено, і дані повернуто. У разі, коли спливає час на підтвердження транзакції, пристрій скине стан запиту, і в тому числі значення резістрів. Це означатиме, що подальші спроби хоста отримати доступ до запиту повернуть помилку, що сигналізуватиме про реуспішність операції. Код обробника запиту наведено на рисунку 18.

```
Endpoint_ClearSETUP();
while (!Endpoint_IsINReady()){}
if (!VerifyOrRequireInteraction(&ButtonStatus, OP_TTL)) {
    OperationStatus.code = ERROR_WAITING_INTERACTION;
    DISCARD_OUT_BUFFER(USB_ControlRequest.wLength);
} else {
    ReadSecret(
        REGISTRY_KEY,
        USB_ControlRequest.wLength,
        Endpoint_Write_Control_Stream_Continuous_LE
    );
    Endpoint_Write_Control_Stream_Complete();
}
Endpoint_ClearStatusStage();
```

Рисунок 18 – Обробник запиту READ_SECRET

Функція ReadSecret (див. рис. 19), реалізує послідовність дій для читання зашифрованих даних зі сховища, забезпечуючи пошук необхідного заголовку запису, подальше визначення адреси першого блоку зашифрованих даних, послідовне розшифрування всіх наявних блоків даних з використанням ключа, що було отримано при розблокуванні пристрою, і в решті решт передачу розшифрованого запису на хост.

Після виклику здійснюється перевірка стану застосунку: якщо ключ шифрування не доступний, тобто система перебуває у заблокованому стані, підсистема статусів операцій зберігає код помилки про відсутній ключ, і виконання негайно завершується, без передачі будь-яких даних на хост пристрій.

```
void ReadSecret(uint8_t* key, uint64_t data_len, WRITER_SIG(writer)) {
    if (!AppState.unlocked) {
        OperationStatus.code = ERROR_NO_KEY;
        return;
    }
    struct SecretHeader header;
    OperationStatus.code = FindSecretHeader(key, &header);
    if (OperationStatus.code != ERROR_NONE) {
        return;
    }
    ReadStorageData(header.next_block, data_len, writer);
}
```

Рисунок 19 – Реалізація функції ReadSecret

В іншому випадку виконується пошук заголовку запису за допомогою функції FindSecretHeader (див. рис. 20), що шукає у сховищі відповідний заголовок за переданим ключем.

```
enum ErrorCode FindSecretHeader(uint8_t* key, struct SecretHeader* outHeader) {
    uint64_t start_address = KeyToAddress(key);
    do {
        SD_ReadBlock(start_address, outHeader, BLOCK_SIZE);
    } while (
        outHeader->status == HEADER_TAKEN &&
        !KEYS_EQUAL(outHeader->key, key) &&
        start_address++
    );

    if (outHeader->status != HEADER_TAKEN) {
        return ERROR_NOT_FOUND;
    }
    return ERROR_NONE;
}
```

Рисунок 20 – Реалізація функції FindSecretHeader

Ця функція перетворює ключ на хеш-суму використовуючи алгоритм fnv1a64, що після цього перетворюється на адресу в адресному просторі блоків заголовків. Алгоритм перетворення наведено на рисунку 21.

```
uint64_t KeyToAddress(uint8_t* key) {
    uint64_t hash = fnv1a64(key, KEY_LENGTH);
    return ADDRESSABLE_BLOCKS_START + hash % ADDRESSABLE_BLOCKS;
}
```

Рисунок 21 – Реалізація функції KeyToAddress

Отримавши адресу з ключа, функція починає послідовно перевіряти блоки, починаючи з даної адреси. Оскільки колізії можливі, важливо співставити значення шуканого ключа з ключем, що збережено у блоці. Якщо ключі співпадають, то шуканий блок знайдено. Якщо ж ключі не співпадають, пошук продовжується до тих пір, поки не буде знайдено блок з співпадаючим ключем, або не буде знайдено порожній блок, що означатиме, що запису з заданим ключем не існує.

Результат пошуку зберігається у глобальному статусі операції. Якщо під час пошуку виникла помилка, функція завершує роботу, не переходячи до наступних етапів. У випадку успішного знаходження заголовка виконується операція читання фактичних даних зі сховища. Для цього використовується функція ReadStorageData (див. рис. 22).

```
void ReadStorageData(uint64_t address, uint64_t data_len, WRITER_SIG(writer)) {
    uint8_t buffer[BLOCK_SIZE];
    while (data_len > 0) {
        uint64_t to_read = min(data_len, BLOCK_SIZE);
        SD_ReadBlock(address, buffer, BLOCK_SIZE);
        DecryptBlock(buffer, address++);
        writer(buffer, to_read);
        data_len -= to_read;
    }
}
```

Рисунок 22 – Реалізація функції ReadStorageData

Ця функція послідовно читає блоки даних починаючи з адреси, що визначена в заголовку запису, допоки не буде досягнуто необхідної довжини, що так само визначена заголовком запису. Дані зчитуються з SD-карти у буфер, після чого дешифруються, і надсилаються напряму до хоста через абстрактний writer.

Важливо помітити, що не дивлячись на вкрай обмежений ресурс оперативної пам’яті, пристрій може працювати з даними, розмір який значно перевищує розмір пам’яті пристрою. Це досягається шляхом прямого транслювання даних, уникаючи їх буферизації. LUFA не надає інструментів для подібних сценаріїв, тому було розроблено функції uint8_t

Endpoint_Write_Control_Stream_Continuous_LE (const void* const, uint16_t); та uint8_t Endpoint_Write_Control_Stream_Complete(), що є варіацією функції Endpoint_Write_Control_Stream_LE(const void* const, uint16_t), що може працювати з сегментами даних. На відміну від останньої, Endpoint_Write_Control_Stream_Continuous_LE у разі, коли неможливо заповнити пакет до повної довжини, не відправляє його, а лишає байти в банку. Наступний її виклик продовжить заповнення пакета з того моменту, де зупинився останній. Після того, як останній байт був записаний, має бути викликана функція Endpoint_Write_Control_Stream_Complete, що завершає трансфер даних надсилаючи незаповнений пакет, і зчитуючи відповідь хоста. Завдяки цим функціям запис може відбуватись поблочно.

Реалізація READ_SECRET_META команди. Однією з команд, необхідних для перелічення збережених записів є READ_SECRET_META (див. рис. 23). Ця команда зчитує заголовок запису за ключем з регістру KEY.

```
Endpoint_ClearSETUP();
while (!Endpoint_IsINReady()){}
struct SecretHeader header;
OperationStatus.code = FindSecretHeader(AppState.registry_key, &header);
Endpoint_Write_Control_Stream_LE(&header, sizeof(header));
Endpoint_ClearStatusStage();
```

Рисунок 23 – Обробник запиту READ_SECRET_META

Реалізація GET_OP_STATUS команди. Для отримання інформації про результат останньої операції, було реалізовано команду GET_OP_STATUS (див. рис. 24).

```
Endpoint_ClearSETUP();
while (!Endpoint_IsINReady()){}
Endpoint_Write_Control_Stream_LE(
    &OperationStatus,
    sizeof(OperationStatus)
);
OperationStatus.code = UNSET;
Endpoint_ClearStatusStage();
```

Рисунок 24 – Обробник команди GET_OP_STATUS

Реалізація GET_DEV_INFO команди. Для отримання базової інформції про пристрій, необхідної для початку роботи, реалізовано команду GET_DEV_INFO. Ця команда повертає статичні відомості про пристрій, такі як розмір блоку даних, розмір ключа, адреси секторів (див. рис. 25).

```
Endpoint_ClearSETUP();
while (!Endpoint_IsINReady()){}
Endpoint_Write_Control_Stream_LE(&dev_info, sizeof(dev_info));
Endpoint_ClearStatusStage();
```

Рисунок 25 – Обробник команди GET_DEV_INFO

Реалізація GET_ADDR_INDEX команди. Ця команда повертає весь сектор індексу заголовків записів. Ця інформація використовується хостом здебільшого щоб сформувати список адрес заголовків, що збережені в системі (див. рис. 26).

```
Endpoint_ClearSETUP();
while (!Endpoint_IsINReady()){}
ReadAddressableIndex(
    Endpoint_Write_Control_Stream_Continuous_LE
);
Endpoint_Write_Control_Stream_Complete();
Endpoint_ClearStatusStage();
```

Рисунок 26 – Обробник команди GET_ADDR_INDEX

Реалізація SEND_SECRET команди. Для створення нового запису на пристрої, реалізовано команду SEND_SECRET. У фазі даних ця команда приймає від хоста 128 байт ключа, та залишок (wLength – 128) байт самого запису (див. рис. 27).

```
Endpoint_ClearSETUP();
struct WriteSecretRequest req;
Endpoint_Read_Control_Stream_LE(&req, sizeof(req));
WriteSecret(
    req.key,
    USB_ControlRequest.wLength - sizeof(req),
    Endpoint_Read_Control_Stream_LE
);
Endpoint_ClearStatusStage();
```

Рисунок 27 – Обробник команди SEND_SECRET

Після того, як було отримано ключ, викликається функція WriteSecret, яка приймає як аргумент функцію зчитування даних, і залишок байтів, що представляє собою сам запис, будуть зчитані вже у цій функції. Реалізація функції WriteSecret наведена на рисунку 28.

```
void WriteSecret(uint8_t* key, uint64_t data_len, READER_SIG(reader)) {
    if (!AppState.unlocked) {
        OperationStatus.code = ERROR_NO_KEY;
        return;
    }

    uint64_t address = KeyToAddress(key);
    uint64_t out_address;

    OperationStatus.code = FindFirstFreeAddressOrDuplicate(
            address, key, &out_address
    );
    if (OperationStatus.code != ERROR_NONE) {
        return;
    }
    WriteSecretAtAddress(address, key, data_len, reader);
}
```

Рисунок 28 – Реалізація функції WriteSecret

В першу чергу функція валідує стан програми, переконуючись, що пристрій розблоковано і ключ шифрування доступний. Після цього функція перетворює ключ на адресу за алгоритмом, що вже було наведено вище, та виконує пошук першого доступного блоку шляхом виклику FindFirstFreeAddressOrDuplicate (див. рис. 29).

```
enum ErrorCode FindFirstFreeAddressOrDuplicate(
    uint64_t start_address,
    uint8_t* key,
    uint64_t* out_address
) {
    struct SecretHeader header;
    *out_address = start_address;
    do {
        ReadSecretHeader(*out_address, &header);
        if (IsHeaderTaken(&header, *out_address) && KEYS_EQUAL(header.key, key)) {
            return ERROR_ALREADY_EXISTS;
        }
    } while (header.status == HEADER_TAKEN && (*out_address)++);

    return ERROR_NONE;
}
```

Рисунок 29 – Реалізація функції FindFirstFreeAddressOrDuplicate

Функція FindFirstFreeAddressOrDuplicate виконує пошук першої вільної адреси, яка є більшою або дорівнює заданій. Для цього вона ітеративно перевіряє блоки, починаючи з заданого, і зупиняєтья лише якщо блок є зайнятим, та має ключ, ідентичний до того, що має бути збережений, тобто знайдено дублікат, або коли все ж таки знайдено вільний блок. У разі, коли FindFirstFreeAddressOrDuplicate завершила своє виконання успішно, виконується власне запис даних до сховища через виклик WriteSecretAtAddress (див. рис. 30).

```
void WriteSecretAtAddress(
    uint64_t address,
    uint8_t* key,
    uint64_t data_len,
    READER_SIG(reader)
) {
    struct SecretHeader header;
    header.status = HEADER_TAKEN;
    memcpy(header.key, key, KEY_LENGTH);
    header.data_length = data_len;
    header.next_block = WriteToStorage(data_len, reader);
    WriteHeaderBlock(address, &header);
    SaveHeaderAddress(address);
}
```

Рисунок 30 – Реалізація функції WriteSecretAtAddress

Ця функція формує блок заголовка, встановлюючи байт маркер зайнятості, записуючи 128 байт ключа, довжину корисних даних, і адресу блоку початку самих даних, після чого блок записується на картку, а кореневий заголовок разом з індексом заголовків записів оновлюється через функцію SaveHeaderAddress (див. рис. 31).

```
void SaveHeaderAddress(uint64_t address) {
    SetHeaderTaken(address, true);
    rootHeader.num_records++;
    WriteRootHeader();
}
```

Рисунок 31 – Реалізація функції SaveHeaderAddress

Для оновлення бітової карти індекса заголовків записів використовується функція SetHeaderTaken (див. рис. 32).

```
void SetHeaderTaken(uint64_t address, uint8_t taken) {
    struct HeaderIndexFlag info = getIndexFlagInfo(address);

    uint8_t buffer[BLOCK_SIZE];
    SD_ReadBlock(info.block, buffer, BLOCK_SIZE);
    if (taken) {
        buffer[info.byte_index] |= info.bit_mask;
    } else {
        buffer[info.byte_index] &= ~info.bit_mask;
    }
    SD_WriteBlock(info.block, buffer, BLOCK_SIZE);
}
```

Рисунок 32 – Реалізація функції SetHeaderTaken

Ця функція перезаписує один біт одного байту у блоці, що відповідає блоку заголовку, що був щойно створений.

Для запису безпосередньо корисних даних до сектора даних, їх шифрування, і отримання назад адреси їх початку, використовується функція WriteToStorage (див. рис. 33). Ця функція послідовно зчитує дані по мірі їх надходження, записує до 512 байт у буфер, і по досягненню повного буфера або закінчення даних шифрує і записує його до сховища. Адреса запису визначається через кореневий заголовок, що містить монотонно зростаючий вказівник на перший вільний блок. По завершенню операції вказівник інкрементується.

```
uint64_t WriteToStorage(uint64_t data_len, READER_SIG(reader)) {
    uint64_t start_address = rootHeader.storage_head;
    while (data_len > 0) {
        uint8_t buffer[BLOCK_SIZE] = {};
        uint64_t to_read = min(BLOCK_SIZE, data_len);
        reader(buffer, to_read);
        EncryptBlock(buffer, rootHeader.storage_head);
        SD_WriteBlock(rootHeader.storage_head++, &buffer, BLOCK_SIZE);
        data_len -= to_read;
    }
    WriteRootHeader();
    return start_address;
}
```

Рисунок 33 – Реалізація функції WriteToStorage

Після того, як всі блоки було записано, кореневий заголовок зберігається, і функція повертає адресу першого блока даних.

Реалізація RESET_DEVICE команди. Для того, щоб мати можливість скинути дані пристрою, було реалізовано команду RESET_DEVICE. Ця команда орієнтована на швидкість виконання, оскільки повний перезапис даних в кожному блоці може займати значний час. Тільки заголовок пристрою та індекс заголовків записів переписуються при виконанні команди, що відбувається майже миттєво.

Для більшої ефективності замість індивідуальних команд запису використовується CMD25 (WRITE_MULTIPLE_BLOCK). Використання цієї команди можливо, оскільки всі блоки, що мають бути перезаписані, розташовані послідовно. Після форматування пристрій переходить у стан перезапису ключ-картки, і знаходиться в ньому, поки користувач не піднесе картку до пристрою, і не відбудеться успішний запис нового, псевдо-випадкового ключа. Код обробника наведено на рисунку 34.

```
Endpoint_ClearSETUP();
FormatState();
uint8_t key[ENC_KEY_LENGTH];
MakePseudoRandomKey(key, sizeof(key));
pn532_begin();
while(!pn532_write_16_bytes(key)) {}
Endpoint_ClearStatusStage();
```

Рисунок 34 – Обробник команди RESET_DEVICE

Реалізація SET_KEY команди. Ця команда дозволяє встановити значення регістру KEY, що необхідно в межах підготовки до подальших запитів, що його використовують. Цей запит реалізує додаткову перевірку, що забороняє міняти стан регістру тоді, коли вже існує транзакція в прогресі, що дозволяє запобігти ситуацій, коли шкідливе ПЗ на комп’ютері підміняє ключ, значення за яким має бути отримане, в останній момент перед тим, як користувач підтвердить транзакцію. Код обробника наведено на рисунку 35.

```
Endpoint_ClearSETUP();
if (HasPendingRequest(&AppState)) {
    OperationStatus.code = ERROR_TRANSACTION_IN_PROGRESS;
} else {
    Endpoint_Read_Control_Stream_LE(AppState.registry_key, KEY_LENGTH);
}
Endpoint_ClearStatusStage();
```

Рисунок 35 – Обробник команди SET_KEY

Реалізація SET_ADDRESS команди. Подібно до SET_KEY, ця команда встановлює значення регістру ADDRESS, що необхідний для виконання деяких команд. Код обробника наведено на рисунку 36.

```
Endpoint_ClearSETUP();
while (!Endpoint_IsINReady()){}
Endpoint_Read_Control_Stream_LE(
    &AppState.registry_address,
    sizeof(AppState.registry_address)
);
Endpoint_ClearStatusStage();
```

Рисунок 36 – Обробник команди SET_ADDRESS

Логування повідомлень. Для зручної роботи з пристроєм, на ньому визначено CDC-ACM інтерфейс, що сприймається комп’ютером як послідовний порт. Це означає, що при підключенні такого пристрою операційна система сама завантажує необхідні драйвери для роботи з пристроєм. При необхідності програма може записувати системні повідомлення напряму в відповідний Bulk ендпоінт (див. рис. 37).

```
void UART_Log(const char* data, ...) {
    va_list args;
    va_start(args, data);
    uint8_t buffer[64];
    int n = vsnprintf(buffer, sizeof(buffer), data, args);
    va_end(args);

    if (n > 0) {
        CDC_Device_SendData(&VirtualSerial1_CDC_Interface, buffer, n);
        CDC_Device_SendData(&VirtualSerial1_CDC_Interface, "\r\n", 2);
        CDC_Device_Flush(&VirtualSerial1_CDC_Interface);
    }
}
```

Рисунок 37 – Реалізація функції логування через серійний порт

3.3 Розробка фонової служби

Розроблений програмний модуль являє собою фоновий сервіс, що реалізує вебінтерфейс для доступу до апаратного пристрою менеджера паролів. Сервіс побудований на основі фреймворку FastAPI та забезпечує комунікацію з пристроєм через абстракцію PasswordDevice. Основним призначенням сервісу є надання стандартизованого HTTP API для читання, запису й управління секретною інформацією, яка фізично зберігається на апаратному носії. Перелік кінцевих точок вебсервіса наведено на рисунку 38.



Рисунок 38 – Кінцеві точки вебсервіса

Для забезпечення крос-доменного доступу застосовано middleware CORS, яке дозволяє обробляти запити з будь-яких джерел, що є важливим у

контексті інтеграції з різними клієнтськими застосунками, включно з браузерними розширеннями.

Ключовим компонентом сервісу є функція отримання екземпляра пристрою, яка інкапсулює логіку доступу до апаратного модуля. У випадку, якщо підключений пристрій не виявлено, сервіс повертає відповідну помилку з кодом 500. Це гарантує коректне поводження клієнтської частини у ситуаціях, коли апаратний менеджер паролів недоступний.

Сервіс надає низку HTTP-методів для роботи з даними. Метод отримання загальної інформації про пристрій повертає основні характеристики й метадані, отримані безпосередньо з апаратної частини. Окрема кінцева точка реалізує можливість завантаження секрету у вигляді двійкового файлу, що дозволяє користувачу експортувати дані у необробленій формі. Кінцева точка /get-secret передбачає безпечне отримання значення секрету у вигляді тексту; при цьому передбачено механізм додаткової перевірки домену, що унеможливлює спроби підміни походження запиту стороннім вебсайтом.

Методи доступу до заголовків записів дозволяють отримувати метадані секретів та перелік усіх наявних записів. Для цього викликаються внутрішні функції пристрою, що повертають список структур даних, які у сервісі перетворюються на формат JSON. Окремі методи надають можливість записувати секрети як у форматі текстового значення, що передається в параметрах запиту, так і у вигляді бінарного вмісту, що завантажується як файл. Наявність двох способів передачі даних забезпечує сумісність з різними сценаріями використання, включаючи інтеграцію з вебклієнтом та програмними інструментами автоматизації.

Додатково реалізовано функцію скидання пристрою, що запускає процедуру очищення або переформатування внутрішньої пам’яті апаратного модуля відповідно до його специфікації. У режимі налагодження сервіс забезпечує можливість читання окремих блоків пам’яті за їхньою фізичною адресою, що корисно при тестуванні та діагностиці низькорівневих операцій.

Кожна кінцева точка сервісу містить обробку виняткових ситуацій та формує уніфіковані повідомлення про помилки. Це забезпечує стабільність роботи системи й полегшує відлагодження. Загалом сервіс формує повноцінний програмний інтерфейс для взаємодії з апаратним менеджером паролів, забезпечуючи безпечний, уніфікований та розширюваний доступ до операцій читання, запису й управління конфіденційною інформацією.

Основна логіка взаєсодії з пристроєм на низькому рівні реалізована в класі PasswordDevice, публічний інтерфейс якого наведено на рисунку 39.

```
class PasswordDevice:
    def __init__(self, dev: usb.core.Device): ...
    @cached_property
    def device_info(self): ...
    def get_op_status(self) -> Status: ...
    def usb_send(self, request: ControlRequest, data: bytes): ...
    def usb_receive(self, request: ControlRequest, length: int) -> bytes: ...
    def get_device_info(self): ...
    def get_state(self): ...
    def set_secret(self, key: str | bytes, value: bytes): ...
    def list_header_blocks(self): ...
    def list_headers(self) -> Iterable[SecretHeader]: ...
    def reset(self): ...
    def lock(self): ...
```

Рисунок 39 – Публічний інтерфейс класу PasswordDevice

Для комунікації з пристроєм через протокол USB використано бібліотеку pyusb. Перерахування ControlRequest визначає набір доступних операцій, серед яких отримання та встановлення секретів, читання блоків пам’яті, скидання пристрою, отримання інформації про стан та структуру пам’яті. Таке централізоване визначення кодів команд забезпечує узгодженість між прошивкою апаратного модуля та програмним забезпеченням хост-системи. Аналогічно, клас Status задає коди повернення, що вказують на результат виконання операції. Це дозволяє реалізувати уніфікований механізм обробки помилок, включаючи ситуації, які потребують взаємодії з користувачем у процесі виконання транзакції. Код реалізації взаємодії з пристроєм на рівні USB наведено на рисунку 40.

```
def usb_send(self, request: ControlRequest, data: bytes):
    with self._lock:
        self.dev.ctrl_transfer(
            bmRequestType_SEND,
            request,
            0,
            0,
            data,
            timeout=60000
        )

        if (status := self._get_op_status_unlocked()) not in (Status.UNSET, Status.ERROR_NONE):
            raise StatusError(status)

def usb_receive(self, request: ControlRequest, length: int) -> bytes:
    with self._lock:
        response = self.dev.ctrl_transfer(
            bmRequestType_RECV,
            request,
            0,
            0,
            length,
            timeout=60000
        )

        if (status := self._get_op_status_unlocked()) not in (Status.UNSET, Status.ERROR_NONE):
            raise StatusError(status)

        return bytes(response)
```

Рисунок 40 – Взаємодія з пристроєм шляхом використання бібліотеки pyusb

Константи bmRequestType_SEND та bmRequestType_RECV визначають значення 1-байтних bitmap значень, що задають напрям передавання даних у межах USB-контрольних запитів як аргумент bmRequestType. Функція пошуку пристрою через usb.core.find забезпечує прив’язку до конкретного апаратного виробу за відомими ідентифікаторами постачальника та продукту. Описові структури SecretHeader та DeviceInfo, оголошені як dataclass, інкапсулюють метадані, які зчитуються з пристрою. Зокрема, інформація про секрет включає ключ, довжину та індекс наступного блоку, а структура даних про пристрій містить конфігураційні параметри пам’яті й розміщення адресованих блоків. Це дозволяє додатку працювати з метаданими у типовій для Python формі.

Додаткові сервісні функції забезпечують трансформацію байтових даних у текстовий формат, коректне представлення двійкових масивів у

вигляді шістнадцяткового дампу та кодування ключів до фіксованої довжини згідно з вимогами протоколу. Спеціалізований виняток StatusError інкапсулює інформацію про невдалий статус виконання операції та використовується для генерації контрольованих помилок у роботі вищих рівнів сервісу.

Основна функціональність зосереджена у класі PasswordDevice, який інкапсулює всі механізми взаємодії з апаратною частиною. Клас отримує дескриптор USB-пристрою і надалі використовує його для надсилання контрольних пакетів. Метод get_op_status виконує додатковий запит до пристрою для отримання стану останньої виконаної операції, що дозволяє відокремити фактичну відповідь пристрою від службового статусу. Методи usb_send та usb_receive реалізують універсальні механізми надсилання та отримання контрольних пакетів з автоматичною перевіркою статусу, що мінімізує дублювання коду та уніфікує логіку обробки помилок.

Окремі службові методи оперують з високорівневими сутностями, такими як заголовки секретів та блоки пам’яті. get_device_info зчитує інформацію про конфігурацію пам’яті пристрою, у тому числі розміри блоків, загальну місткість та розподіл адресованих секцій. Методи get_header і get_secret забезпечують отримання метаданих та вмісту секрету відповідно. Реалізація читання секрету передбачає обробку стану, коли пристрій потребує додаткової взаємодії з користувачем, наприклад підтвердження на апаратній кнопці, і повторює запит доти, доки не буде отримано результат.

Функція запису секрету формує коректний двійковий пакет, що складається з закодованого ключа та значення, і передає його за допомогою відповідної USB-команди. Окремі методи дозволяють перелічувати блоки пам’яті, які містять заголовки секретів, та зчитувати ці заголовки для формування повного списку збережених записів. Це забезпечує можливість побудови індексації на рівні програмного забезпечення. Нарешті, операція скидання пристрою викликає відповідну команду, що ініціює повне очищення пам’яті згідно з алгоритмами прошивки.

Таким чином, клас реалізує повний цикл управління апаратним менеджером паролів через інтерфейс USB, забезпечуючи абстракцію високого рівня для роботи з конфіденційними даними, а також стабільний і контрольований механізм обміну командами та перевірки коректності їх виконання.

3.4 Розширення браузера

Для зручного використання фонового сервіса менеджера паролей було розроблено спеціальне розширення браузера. Воно реалізує інтерактивну взаємодію з полями введення пароля на вебсторінках та забезпечує автоматизоване отримання, збереження й підстановку паролів через локальний API сервісу керування секретами. Основною задачею розширення є перехоплення появи та фокусування полів типу password, динамічне формування контекстного вікна з доступними діями, а також здійснення запитів до бекенд-сервера з метою отримання або збереження відповідних даних. Логіка роботи вибудувана таким чином, щоб мінімізувати вплив на сторінку, та забезпечити стабільну роботу при асинхронних операціях.

Контент-скрипт використовує домен поточного сайту як ключ доступу до секрету. На основі цього ключа він викликає відповідні API-ендпоїнти для перевірки існування пароля, його завантаження або створення нового. Виклик API здійснюється через HTTP-запити до локального сервера, причому передбачена обробка всіх можливих помилкових відповідей. У разі виникнення проблем, таких як недоступність сервера, неправильний формат даних або відсутність секрету, користувач отримує візуальне повідомлення у вигляді toast-повідомлення, що дає змогу не переривати взаємодію з полем вводу.

Однією з ключових функціональних складових є механізм побудови та позиціонування спливаючого вікна (див. рис. 41).

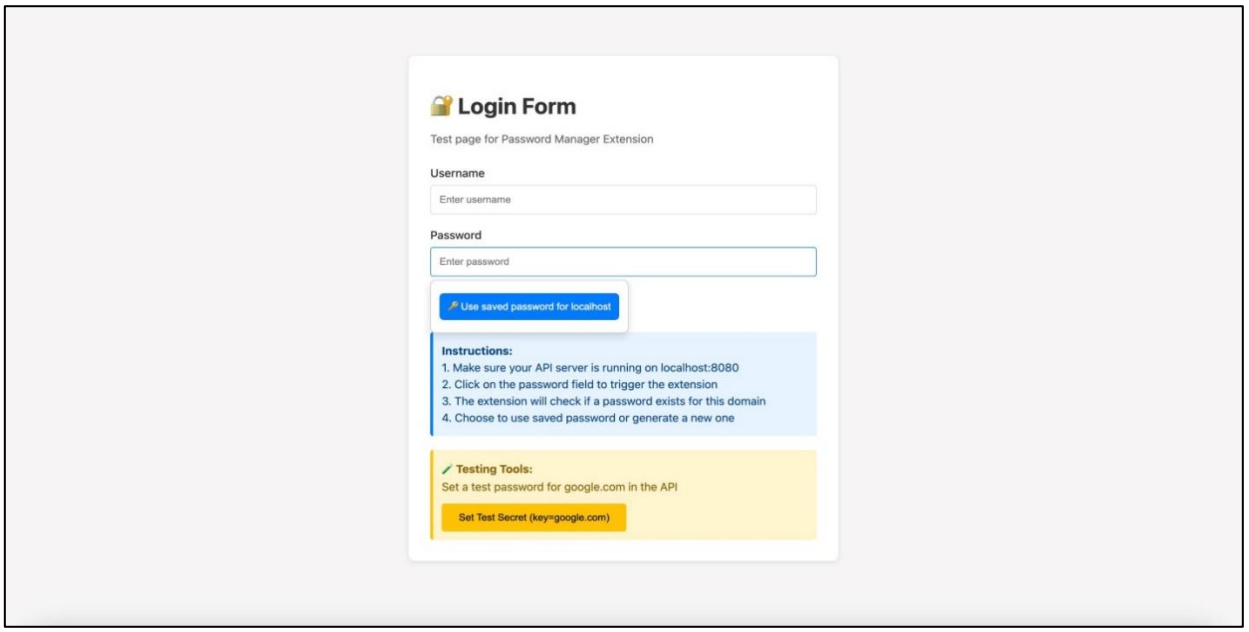


Рисунок 41 – Розширення пропонує автоматичне заповнення паролю

При фокусуванні на полі з паролем розширення створює інтерфейсну панель, яка розміщується безпосередньо під відповідним елементом форми. Вікно може перебувати у стані очікування, поки триває перевірка наявності пароля, або ж пропонувати користувачеві вибрати одну з доступних дій. Якщо пароль для поточного домену вже існує, користувачу пропонується заповнити поле збереженим значенням

Якщо ж запис відсутній, розширення дозволяє згенерувати новий надійний пароль. Згенерований пароль автоматично надсилається на сервер для збереження, після чого вводиться у відповідне поле форми (див. рис. 42).

Передбачено обробку асинхронних операцій таким чином, щоб запобігти виникненню помилок у разі, коли елемент форми зникає зі сторінки або змінюється її структура. Для цього реалізовано збереження посилань на активне поле вводу, перевірку його наявності в DOM після виконання асинхронних запитів, а також акуратне відновлення первинного стану інтерфейсу. Щоб обмежити кількість одночасно відкритих спливаючих елементів, скрипт закриває попередні вікна при появі нового та видаляє їх при кліку поза межами елемента або під час прокручування сторінки.

Додатково реалізовано механізм спостереження за змінами DOM-дерева для коректної роботи на динамічних сайтах, де поля введення можуть

з’являтися без перезавантаження сторінки. За допомогою MutationObserver скрипт відстежує появу нових елементів і автоматично підключає обробники подій до нових інпутів. Це забезпечує повну сумісність із сучасними односторінковими застосунками.

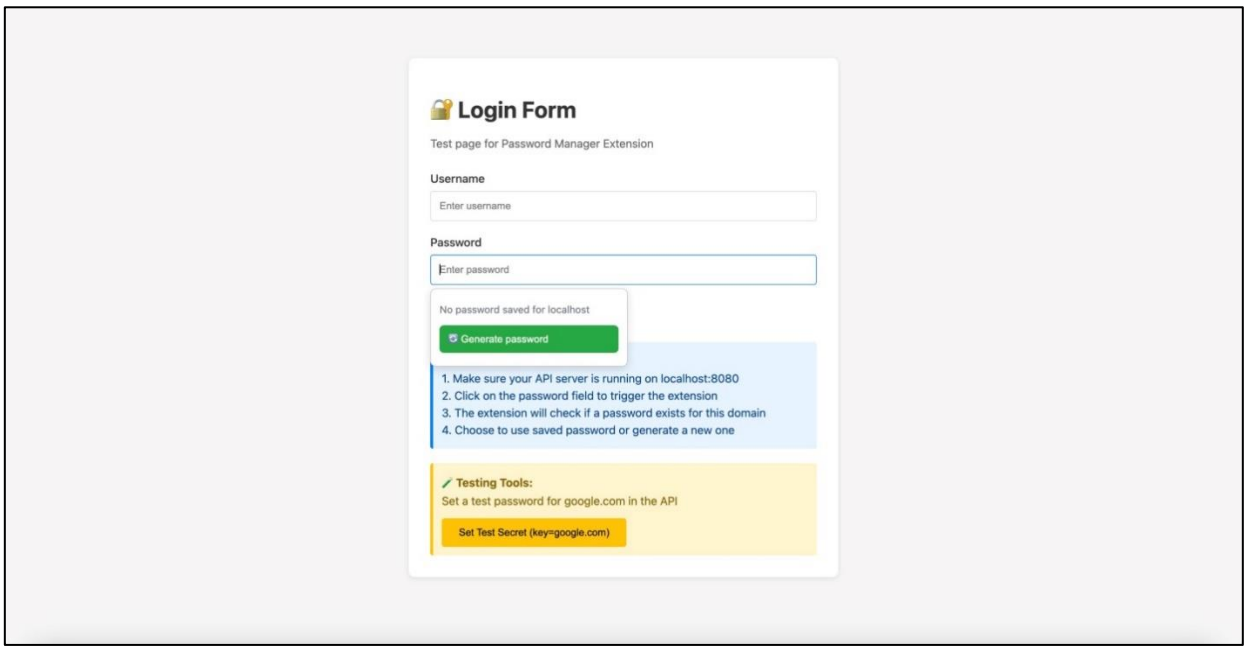


Рисунок 42 – Розширення пропонує генерацію нового паролю

Для візуального сповіщення користувача передбачено систему toast-повідомлень, що реалізують інформування про успішні або помилкові операції.

Загалом розширення працює як прозорий проміжний рівень між вебсторінкою та сервером керування секретами. Воно значно зменшує ризики, пов’язані з ручним введенням паролів, забезпечує централізоване збереження облікових даних та підвищує зручність користувача за рахунок автоматизованого генерування і підставлення паролів. Така архітектура робить розширення універсальним рішенням, яке може легко інтегруватися в робочий процес без потреби змінювати код цільових сайтів.

ВИСНОВКИ

У рамках даної роботи було розроблено прототип апаратного менеджера паролів, що забезпечує підвищений рівень безпеки завдяки поєднанню криптографічних методів, фізичної ізоляції та апаратних механізмів контролю доступу. Проведений аналіз сучасних рішень у сфері управління паролями показав, що традиційні програмні менеджери залишаються вразливими до атак шкідливого програмного забезпечення та перехоплення майстер-пароля. Апаратні рішення, хоч і пропонують вищий рівень захисту, мають обмеження у гнучкості, відкритості та функціональності. Це обґрунтовує доцільність створення пристрою з відкритою архітектурою, можливістю модифікації та підтримкою зберігання довільних зашифрованих даних.

У ході реалізації було спроєктовано архітектуру системи, що включає мікроконтролер AVR з USB-інтерфейсом, SD-карту для зберігання зашифрованих даних, RFID-картку як фізичний носій ключа шифрування, OLED-дисплей і кнопку підтвердження операцій. Розроблено спеціальний формат зберігання даних без використання файлової системи, що забезпечує швидкий доступ до записів та можливість роботи з великими носіями. Для шифрування застосовано алгоритм AES-256-CBC, де ключ ніколи не покидає меж апаратної частини системи. Протокол взаємодії з ПК реалізовано через контрольні USB-трансфери, що мінімізує поверхню атаки та обмежує хост у наборі дозволених дій.

Було створено прошивку мікроконтролера на основі бібліотеки LUFA, яка забезпечує коректну роботу USB-стека та обробку команд. Реалізовано клієнтську частину системи у формі служби, що дозволяє браузерному розширенню взаємодіяти з апаратним модулем через локальний HTTP-інтерфейс, забезпечуючи зручність використання та інтеграцію у повсякденні робочі процеси.

Проведене тестування підтвердило працездатність основних функцій: ініціалізація та робота SD-картки, шифрування та дешифрування даних, взаємодія з RFID-карткою, коректна обробка USB-команд та робота запобіжних механізмів, зокрема фізичного підтвердження транзакцій. Отримані результати засвідчують, що запропонована система успішно вирішує поставлені задачі та може бути основою для подальшого розвитку.

Перспективи подальших досліджень передбачають розширення протоколу взаємодії, реалізацію додаткових криптографічних механізмів, підтримку складніших сценаріїв керування ключами, а також розробку повноцінного кросплатформенного клієнта. Створений пристрій демонструє можливість побудови доступного та гнучкого апаратного менеджера паролів, який поєднує високу безпеку, відкритість і практичність застосування.

ПЕРЕЛІК ПОСИЛАНЬ

1. Juggling security: How many passwords does the average person have in 2024? URL: https://nordpass.com/blog/how-many-passwords-does-average-person-have/ (дата звернення: 22.04.2025).

2. Keep your memory dump shut: Unveiling data leaks in password managers. URL: https://arxiv.org/abs/2404.00423 (дата звернення: 22.04.25).

3. A Security and Usability Analysis of Local Attacks Against FIDO2. URL: https://www.ndss-symposium.org/wp-content/uploads/2024-327-paper.pdf (дата звернення: 23.04.2025).

4. 19 billion leaked passwords reveal deepening crisis: lazy, reused, and stolen. URL: https://cybernews.com/security/password-leak-study-unveils-2025-trends-reused-and-lazy (дата звернення: 25.04.2025).

5. Data Breach Statistics 2025: Key Trends, Costs & Risks Revealed. URL: https://sqmagazine.co.uk/data-breach-statistics (дата звернення: 25.04.2025).

6. What is a keylogger? A total protection guide. URL: https://us.norton.com/blog/malware/what-is-a-keylogger (дата звернення: 26.04.2025).

7. Security Evaluation of Password Managers: A Comparative Analysis and Penetration Testing of Existing Solutions. URL: https://papers.academic-conferences.org/index.php/iccws/article/view/3330 (дата звернення: 31.04.2025).

8. Аналіз безпеки паролів користувачів за допомогою Python-скриптів. URL: https://vottp.khmnu.edu.ua/index.php/vottp/article/view/594 (дата звернення: 02.05.2025).

9. Passwords Usage and Human Memory Limitations: A Survey across Age and Educational Background. URL: https://journals.plos.org/plosone/article?id=10.1371/journal.pone.0051067 (дата звернення: 02.05.2025).

10. Password Manager Statistics 2025: How Passwords Are Changing

Security. URL: https://sqmagazine.co.uk/password-manager-statistics/ (дата звернення: 04.05.2025).

11. An Encryption Model that Prioritizes your Security. URL: https://www.lastpass.com/security/zero-knowledge-security (дата звернення: 10.05.2025).

12. Bitwarden reviewer’s guide. URL: https://bitwarden.com/resources/bitwarden-reviewers-guide/ (дата звернення: 10.05.2025).

13. Dashlane Review: Features, Pricing & More. URL: https://www.forbes.com/advisor/business/software/dashlane-review/ (дата звернення: 10.05.2025).

14. 1Password Features. URL: https://1password.com/features (дата звернення: 14.05.2025).

15. KeePassXC: User Guide. URL: https://keepassxc.org/docs/KeePassXC_UserGuide (дата звернення: 14.05.2025).

16. What is a YubiKey? URL: https://docs.yubico.com/yesdk/users-manual/getting-started/what-is-a-yubikey.html (дата звернення: 10.10.2025).

17. OnlyKey Features. URL: https://docs.onlykey.io/features.html (дата звернення: 15.05.2025).

18. Nitrokey Website. URL: https://www.nitrokey.com/products/nitrokeys (дата звернення: 16.05.2025).

19. Alexandre Dumas – Vingt ans après. URL: https://beq.ebooksgratuits.com/vents/Dumas-20ans-1.pdf (дата звернення: 16.05.2025)

20. Microchip Developer Help. URL: https://developerhelp.microchip.com/xwiki/bin/view/applications/usb/how-it-works/hosts/ (дата звернення: 02.06.2025).

21. USB endpoints and their pipes. URL: https://learn.microsoft.com/en-us/windows-hardware/drivers/usbcon/usb-endpoints-and-their-pipes (дата звернення: 02.06.2025).

22. How to send a USB control transfer. URL: https://learn.microsoft.com/en-us/windows-hardware/drivers/usbcon/usb-control-transfer (дата звернення: 03.06.2025).

23. How to send USB bulk transfer requests. URL: https://learn.microsoft.com/en-us/windows-hardware/drivers/usbcon/usb-bulk-and-interrupt-transfer (дата звернення: 03.06.2025).