**ДЕРЖАВНИЙ УНІВЕРСИТЕТ**
**ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ**
**НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ**
**ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ**
**КАФЕДРА ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ**

# КВАЛІФІКАЦІЙНА РОБОТА

на тему: «Розробка системи збору і аналізу даних з електролічильників з використанням технології ModBus (RS485)»

на здобуття освітнього ступеня бакалавра
зі спеціальності 121 Інженерія програмного забезпечення
освітньо-професійної програми «Інженерія програмного забезпечення»

*Кваліфікаційна робота містить результати власних досліджень. Використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело*

_____ Владислав ЄМЕЛЬЯНОВ
*(підпис)*

Виконав: здобувач вищої освіти групи ПД-41

Владислав ЄМЕЛЬЯНОВ

Керівник: Ірина ЗАМРІЙ
*д.т.н., професор*

Рецензент: _____

**Київ 2025**

**ДЕРЖАВНИЙ УНІВЕРСИТЕТ**
**ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ**

**Навчально-науковий інститут інформаційних технологій**

Кафедра Інженерії програмного забезпечення

Ступінь вищої освіти Бакалавр

Спеціальність 121 Інженерія програмного забезпечення

Освітньо-професійна програма «Інженерія програмного забезпечення»

**ЗАТВЕРДЖУЮ**
Завідувач кафедри
Інженерії програмного забезпечення
_____ Ірина ЗАМРІЙ
« ____ » _____ 2025 р.

# ЗАВДАННЯ
# НА КВАЛІФІКАЦІЙНУ РОБОТУ

Ємельянову Владиславу Богдановичу

1. Тема кваліфікаційної роботи: «Розробка системи збору і аналізу даних з електро-лічильників з використанням технології ModBus (RS485)»
керівник кваліфікаційної роботи д.т.н., професор Ірина ЗАМРІЙ,
затверджені наказом Державного університету інформаційно-комунікаційних технологій від «24» лютого 2025 р. № 56.

2. Строк подання кваліфікаційної роботи «02» червня 2025 р.

3. Вихідні дані до кваліфікаційної роботи: теоретичні відомості про методи комунікації через серійний порт та шину RS485, опис роботи десктопних додатків з пакетом Qt, документація бібліотек PySide6, TortoiseORM, xlsxwriter.

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)

1. Провести аналіз способів збору та аналізу даних в електролічильників за допомогою протоколу ModBus.
2. Дослідити та проаналізувати існуючі аналоги.
3. Сформувати функціональні та нефункціональні вимоги до додатку.
4. Спроєктувати архітектуру модулів екранного відображення (pyQt) та потокового збору даних за допомогою asyncio.

5. Реалізувати десктопний додаток за визначеними вимогами.
6. Провести тестування основного функціоналу додатку.
7. Провести розгортання системи на різних операціних системах (Windows, Linux) та архітектурах (x86, x64, ARM64).
8. Реалізувати механізм автоматизації розгортання в специфічних середовищах (RaspberryPi).
9. Запропонувати можливі покращення системи в перспективі подальшої розробки.

5. Перелік графічного матеріалу: *презентація*
1. Аналіз аналогів.
2. Вимоги до програмного забезпечення.
3. Програмні засоби реалізації.
4. Діаграма варіантів використання.
5. Діаграма алгоритму збору даних.
6. Діаграма алгоритму роботи потоку збору даних
7. Діаграма класів модулю потокового збору даних.
8. Діаграма моделей бази даних
9. Екранні форми.
10. Апробація результатів дослідження

6. Дата видачі завдання «25» лютого 2025 р.

## КАЛЕНДАРНИЙ ПЛАН

| № з/п | Назва етапів кваліфікаційної роботи | Строк виконання етапів роботи | Примітка |
|---|---|---|---|
| 1 | Підбір та аналіз науково-технічної літератури | 25.02-04.03.2025 | |
| 2 | Аналіз та дослідження існуючих аналогів | 05.03-13.03.2025 | |
| 3 | Провести аналіз способів збору та аналізу даних в електролічильників за допомогою протоколу ModBus. | 14.03-20.03.2025 | |
| 4 | Дослідити та проаналізувати існуючі аналоги. | 21.03-26.03.2025 | |
| 5 | Сформувати функціональні та нефункціональні вимоги до додатку. | 27.03-02.04.2025 | |
| 6 | Спроєктувати архітектуру модулів екранного відображення (pyQt) та потокового збору даних за допомогою asyncio. | 03.04-11.04.2025 | |
| 7 | Реалізувати десктопний додаток за визначеними вимогами. | 12.04-12.05.2025 | |
| 8 | Провести тестування основного функціоналу додатку. | 13.05-16.05.2025 | |
| 9 | Провести розгортання системи на різних операціних системах (Windows, Linux) та архітектурах (x86, x64, ARM64). | 17.05-21.05.2025 | |

| | | | |
|---|---|---|---|
| 10 | Реалізувати механізм автоматизації розгортання в специфічних серидовищах (RaspberryPi). | 22.05.2025 | |
| 11 | Запропонувати можливі покращення системи в перспективі подальшої розробки. | 23.05-25.05.2025 | |
| 12 | Оформлення роботи: вступ, висновки, реферат | 26.05-27.05.2025 | |
| 13 | Розробка демонстраційних матеріалів | 28.05.2025 | |
| 14 | Попередній захист роботи | 29.05-30.05.2025 | |

Здобувач вищої освіти _____ Владислав ЄМЕЛЬЯНОВ
*(підпис)*

Керівник
кваліфікаційної роботи _____ Ірина ЗАМРІЙ
*(підпис)*

# РЕФЕРАТ

Текстова частина кваліфікаційної роботи на здобуття освітнього ступеня бакалавра: _56_ стор., __1__ табл., _10_ рис., _21_ джерел.

*Мета роботи* – спрощення збору та аналізу даних з аналізаторів мережі, покращення масштабованості відносно існуючих рішень.

*Об'єкт дослідження* – процеси збору та аналізу даних з електролічильників.

*Предмет дослідження* – система збору та аналізу даних з електролічильнків (аналізаторів мережі).

*Короткий зміст роботи*: В роботі проаналізовано алгоритми та методи для збору та аналізу даних з аналізаторів мережі. Проаналізовано програмні засоби для збору та аналізу даних з електролічильників: Schneider Electric EcoStruxure Power Monitoring Expert, Siemens Power Manager та Ability™ Energy and Asset Management (ABB). Розроблено архітектуру та алгоритм роботи застосунку і окремих модулів та програмно реалізовані ключові функціональні можливості, зокрема: встановлення стабільного підключення до аналізаторів мережі через серійний порт, шину RS485 за протоколом ModBus. Перегляд та аналіз даних з підключених лічильнків та відображення поточних показників лічильників в реальному часі. Відслідковування пікових значень на лічильниках. В роботі використано бібліотеку pySide6 для віконної частини додатку, xlsxwriter для створення звітів у .xlsx форматі, tortoise-orm для асинхронного керування і доступу до бази даних, SQLite в якості бази даних.

Сферою використання застосунку є організація моніторингу і обліку енергоспоживання на побутових та промислових об'єктах.

КЛЮЧОВІ СЛОВА: СЕРІЙНИЙ ПОРТ, RS485, MODBUS, АНАЛІЗАТОР МЕРЕЖІ, ЕЛЕКТРОЕНЕРГІЯ, ЕНЕРГОСПОЖИВАННЯ, АСИНХРОННІСТЬ, АРХІТЕКТУРА, АЛГОРИТМ, PYTHON, QT, ДЕСКТОПНИЙ ДОДАТОК, БАЗА ДАНИХ.

**ЗМІСТ**

# ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

ПЗ – Програмне забезпечення

IDE – Integrated Development Environment

ORM – Object-Relational Mapping

SQL – Structured Query Language

SCADA - Supervisory Control And Data Acquisition

LDAP - Lightweight Directory Access Protocol

TCP - Transmission Control Protocol

RTU - Remote Terminal Unit

ООП – Об’єктно Орієнтовне Програмування

UI – User Interface

ARM – Advanced RISC Machine

USB – Universal Serial Bus

GPIO – General-Purpose Input/Output

DPI – Dots Per Inch

CRC – Cyclic Redundancy Check

# ВСТУП

*Актуальність:* сучасні енергетичні системи потребують ефективних інструментів моніторингу та аналізу споживання електроенергії. Зростання цін на енергоносії та необхідність оптимізації витрат роблять системи обліку електроенергії критично важливими для промислових підприємств та комерційних організацій. Протокол ModBus через інтерфейс RS485 став промисловим стандартом для комунікації з електролічильниками, що відкриває широкі можливості для автоматизації збору даних.

Розробка програмного забезпечення для роботи з електролічильниками є особливо актуальною через:

– необхідність точного обліку енергоспоживання;

– можливість виявлення аномалій у роботі обладнання;

– оптимізацію витрат на електроенергію;

– відсутність доступних універсальних рішень для малих та середніх підприємств.

*Об'єктом дослідження* є процеси збору та аналізу даних з електролічильників.

*Предметом дослідження* є система збору та аналізу даних з електролічильнків (аналізаторів мережі).

*Метою роботи* є спрощення збору та аналізу даних з аналізаторів мережі, покращення масштабованості відносно існуючих рішень.

*Методи дослідження:* першим кроком було проведено дослідження в сфері принципів та процесів збору та аналізу даних з аналізаторів мережі. Другим кроком було проаналізовано інсуючі системи для визначення особливостей роботи та використання цих систем та для формулювання початкових функціональних та нефункціональних вимог до свого програмного забезпечення. Після чого було проведено огляд стеку технологій, що будуть максимально відповідати вимогам до програмного забезпечення, з ухилом в

максимальне спрощення розробки і подальшої підтримки та розширення системи. Було обрано Python 3 як основну мову програмування, з метою використання наявних потужних, але легких бібліотек Tortoise ORM для асинхронного керування локальною базою даних SQLite, xlsxwriter для створення складних та функціональних знімків бази даних в форматі .xlsx, PySide6 для кросплатформеної розробки віконного додатку на базі універсального пакету Qt. Дослідження реалізації застосунку проводилося під час безпосередньої розробки.

*Наукова новизна* роботи полягає в таких функціональних складових: інструмент уніфікації збору даних з різних моделей лічильників з наявною шиною RS485; механізм паралельного опитування пристроїв з використанням потоків asyncio.

*Практична значущість результатів* полягає в можливості спростити розгортання системи для збору та аналізу даних з електролічильників без використання додаткового устаткування та спеціалізованого персоналу та використання однієї уніфікованої системи для роботи з багатьма лічильниками різних виробників та типів, за умови наявності в них шини RS485.

# 1 АНАЛІЗ ПРИНЦИПІВ ТА ПРОЦЕСІВ ЗБОРУ ТА АНАЛІЗУ ДАНИХ З АНАЛІЗАТОРІВ МЕРЕЖІ

## 1.1 Огляд літературних джерел предметної області

Протокол **Modbus RTU**, розроблений компанією Modicon у 1979 році, став одним із найпоширеніших стандартів для промислових систем збору та обміну даними. Його популярність зумовлена відкритістю, простотою реалізації та сумісністю з широким спектром обладнання, зокрема з аналізаторами електричних мереж.

У дослідженні Devanshi N. Patel та Prof. Sunil B. Somani [1] представлено реалізацію системи збору даних на базі контролера Cortex M3 з використанням протоколу Modbus RTU через інтерфейс RS-485. Автори підкреслюють переваги RTU-режиму над ASCII, зокрема вищу швидкість передачі та ефективність у багатоточкових мережах.

Інше дослідження [2], проведене на базі програмного забезпечення C#, демонструє розробку системи моніторингу електричних параметрів з використанням Modbus RTU та RS-485. У роботі описано створення графічного інтерфейсу користувача для відображення даних, зібраних з аналізаторів енергії, що підкреслює гнучкість та універсальність протоколу Modbus у різних програмних середовищах.

Також варто зазначити, що Modbus RTU активно використовується в системах автоматизації та контролю процесів [3]. Його здатність до роботи в реальному часі та підтримка широкого спектру функцій робить його придатним для застосування в різних галузях промисловості, включаючи енергетику, виробництво та будівництво.

Таким чином, аналіз літературних джерел свідчить про широке застосування протоколу Modbus RTU у системах збору та аналізу даних з

аналізаторів мережі. Його відкритість, гнучкість та сумісність з різними пристроями роблять його оптимальним вибором для реалізації ефективних та масштабованих систем моніторингу енергоспоживання.

### 1.2 Вибір інтерфесів підключення: RS485 vs RJ45

Інтерфейс **RS485** [4] — це надійне рішення, що використовується в промисловості вже десятки років. Його головна перевага — можливість передавати дані на великі відстані **(до 1200 метрів)** з високим рівнем захисту від перешкод. Це особливо важливо в умовах електрощитових приміщень, де присутні сильні електромагнітні поля. RS485 підтримує мультидропну топологію: до однієї шини можна підключити до 32 пристроїв без необхідності встановлення ретрансляторів. При цьому комунікація відбувається послідовно, що спрощує кабельне з'єднання та здешевлює монтаж.

**Ethernet (RJ45)** [4] — це сучасний інтерфейс, який забезпечує високі швидкості обміну даними та повноцінну мережеву інтеграцію. Через використання TCP/IP [5] протоколу, Ethernet дає змогу зручно підключати пристрої до серверів, локальних мереж, хмарних платформ або SCADA-систем. Але в реальній практиці підключення до Ethernet для лічильників зазвичай означає використання проміжного обладнання — наприклад, конвертерів RS485-to-Ethernet або окремих шлюзів, що може ускладнити систему і підвищити її вартість. Крім того, Ethernet менш стійкий до індустріальних завад, а кабельну інфраструктуру потрібно прокладати обережніше, аби уникнути впливу шумів.

Беручи до уваги всі ці чинники, у моєму проєкті було прийнято рішення на користь **RS485**. Це рішення зумовлено кількома практичними перевагами:

- простота реалізації — з'єднання кількох лічильників по одній шині;
- дешевизна — відсутність потреби в дорогому мережевому обладнанні;
- висока сумісність — більшість сучасних промислових лічильників підтримують RS485 та протокол ModBus RTU [5];
- надійність у складних умовах — стійкість до шумів та перешкод.

Таким чином, **RS485 став оптимальним вибором** для реалізації енергоефективної та масштабованої системи збору даних.

### 1.3 Порівняння протоколів ModBus та M-Bus

Окрім вибору фізичного інтерфейсу, важливо було визначитися з протоколом обміну даними між програмним забезпеченням і самими приладами. Для пристроїв, що підключаються через шину RS485, існує кілька варіантів протоколів, серед яких найбільш поширеними є **ModBus RTU** та **M-Bus**.

**ModBus** [6] — це відкритий і добре документований протокол, який підтримується більшістю промислових пристроїв, зокрема лічильниками від компанії Eastron (моделі SDM120, SDM630, SDM72-M тощо). Він дозволяє опитувати пристрої через реєстри, зчитувати значення у форматі 16- або 32-бітних слів та навіть надсилати команди для налаштувань. ModBus може працювати як у форматі RTU (послідовна передача даних, зазвичай через RS485), так і TCP (передача через Ethernet), що робить його надзвичайно гнучким.

Окремо варто згадати, що ModBus підтримує адресацію пристроїв — тобто дозволяє опитувати кілька лічильників по одній шині без конфліктів. Крім того, програмна реалізація цього протоколу є досить простою, що полегшує розробку систем збору даних навіть для невеликих проєктів.

**M-Bus** — інший поширений протокол, спеціально розроблений для лічильників ресурсів (тепла, води, газу, електроенергії). Він дозволяє будувати великі системи з низьким енергоспоживанням та може працювати як у дротовому, так і бездротовому варіанті. Проте, у порівнянні з ModBus, він є більш спеціалізованим та обмеженим у плані сумісності — не всі моделі лічильників підтримують його «з коробки». Крім того, реалізація M-Bus зазвичай потребує додаткових модулів або конвертерів.

У моєму випадку вибір на користь ModBus RTU був очевидним:

- підтримується всіма пристроями, які мають RS-485 шину;

- дозволяє просто зчитувати ключові параметри — напругу, струм, потужність, споживання енергії;
- має добре описані відкриті карти регістрів (реєстр-мапи), що значно спрощує процес програмування;
- легко масштабується — один **ModBus-мастер** може опитувати десятки пристроїв без значного ускладнення логіки.

З огляду на це, ModBus RTU забезпечив ідеальний баланс між простотою реалізації, гнучкістю та надійністю, що робить його найкращим вибором для мого завдання.

### 1.4 Аналіз потреб підприємтсва

Під час дослідження ринку та бесід із замовниками було виявлено, що існуючі комерційні рішення (наприклад, ПЗ від виробників лічильників) часто є дорогими, закритими та прив’язаними до конкретних моделей пристроїв. Натомість клієнти (особливо малі та середні підприємства) потребують:

- Дешевої системи – без необхідності купівлі додаткового обладнання (достатньо RS485-USB конвертера за ~2$) з максимальним використанням наявного.
- Універсальності – можливості підключення будь-яких лічильників з ModBus, а не лише фірмових.
- Простоти інтеграції – експорт даних у Excel (.xlsx) для подальшого аналізу в SCADA або бухгалтерських системах.

Підприємтсва, незалежно від напрямку їхньої роботи, усі використовують електроенерігю, вони закупяють її у міста чи області де знаходяться, у інших приватних підприємств чи генеруючть самі. Усі вони намагаються здешевити енергію, яку використовують – тим самим здешевлюючи свою послугу чи продукт.

Приклад застосування: Промислове підприємство з динамічним енергоспоживанням

Розглянемо виробничий завод, який:

- купує електроенергію з різних джерел (місто, інші приватні підприєства, власна генерація, альтернативна енергія) та типів генерації (гідро-, тепло- та сонячні електростанції);
- має змінні тарифи для виробництв залежно від часу доби;
- потребує точного та розгалудженого аналізу споживання для оптимізації витрат.

За допомогою системи збору та аналізу даних з аналізаторів мережі підприємство може:

1. Збирати дані з усіх лічильників у реальному часі.
2. Порівнювати періоди (день/ніч, робочі/вихідні дні) для виявлення пікових навантажень.
3. Генерувати звіти у Excel, щоб оцінити, коли вигідніше використовувати власну генерацію.
4. Інтегрувати дані в існуючі системи обліку через API або експорт.

Це дозволить знизити витрати на енергію на 10–20% лише за рахунок аналітики, без дорогих оновлень обладнання.

## 1.5 Результати аналізу предметної області

Проведений аналіз фізичних інтерфейсів підключення та протоколів обміну даними дозволив обґрунтовано визначити оптимальне технічне рішення для реалізації системи збору даних з аналізаторів електромережі. У результаті порівняння інтерфейсів RS485 та Ethernet (RJ45) було встановлено, що RS485 є найбільш придатним для промислових умов завдяки своїй надійності, стійкості до електромагнітних завад, підтримці мультидропної топології та простоті реалізації. Його використання дозволяє мінімізувати витрати на обладнання та монтаж, що особливо актуально для підприємств із обмеженим бюджетом.

У свою чергу, серед доступних протоколів обміну, таких як ModBus та M-Bus, перевагу було віддано ModBus RTU через його відкриту специфікацію,

широку підтримку промисловими пристроями, зокрема лічильниками Eastron, простоту програмної реалізації та гнучку систему адресації. Завдяки цьому ModBus RTU забезпечує легку масштабованість, дозволяючи зчитувати дані з десятків пристроїв по одній шині без ускладнення логіки опитування.

Особливу увагу було приділено реальним потребам цільової аудиторії — малих і середніх підприємств. Було встановлено, що наявні комерційні рішення є часто надто дорогими або закритими, тоді як користувачі потребують гнучких, універсальних і доступних систем, які можуть працювати з різними моделями лічильників, не потребують спеціалізованого мережевого обладнання та дозволяють просто експортувати дані у зручні формати (наприклад, .xlsx).

Таким чином, обрана архітектура, що базується на RS485 як фізичному інтерфейсі та ModBus RTU як протоколі зв'язку, дозволяє створити ефективну, надійну й масштабовану систему збору та аналізу даних. Вона відповідає як технічним вимогам, так і практичним очікуванням кінцевих користувачів, забезпечуючи високу сумісність, простоту розгортання та потенціал для подальшої інтеграції з бізнес-процесами підприємства.

# 2 АНАЛІЗ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ ДЛЯ ЗБОРУ ТА АНАЛІЗУ ДАНИХ З АНАЛІЗАТОРІВ МЕРЕЖІ З ВИКОРИСТАННЯМ ТЕХНОЛОГІЇ MODBUS

## 2.1 Дослідження аналогів

Сучасний ринок програмного забезпечення для моніторингу енергоспоживання пропонує різні підходи до збору та аналізу даних. У цьому розділі проаналізовано три провідні промислові рішення: Schneider Electric EcoStruxure Power Monitoring Expert [7], Siemens Power Manager [8] та Ability™ Energy and Asset Management (ABB) [9]. Особливу увагу приділено зручності управління, налаштуванню систем та можливостям аналітики.

### 2.1.1 Schneider Electric EcoStruxure Power Monitoring Expert

Особливості системи:

- розроблено для комплексного моніторингу енергоспоживання на великих промислових об'єктах;
- використовує власну екосистему пристроїв Schneider Electric;
- підтримує роботу як з ModBus RTU, так і з ModBus TCP.

Зручність управління:

- інтуїтивний web-інтерфейс з можливістю віддаленого доступу;
- готові шаблони дашбордів для різних типів об'єктів (Рис. 2.1);
- детальні налаштування прав доступу для різних груп користувачів.

Налаштування та впровадження:

- вимагає кваліфікованого персоналу для початкового налаштування;
- час впровадження може сягати кількох тижнів для складних систем;
- обмежена гнучкість при роботі з нестандартними конфігураціями.

Аналітичні можливості:

- вбудовані інструменти для порівняльного аналізу періодів;
- можливість створення користувацьких звітів;
- підтримка прогнозування енергоспоживання.

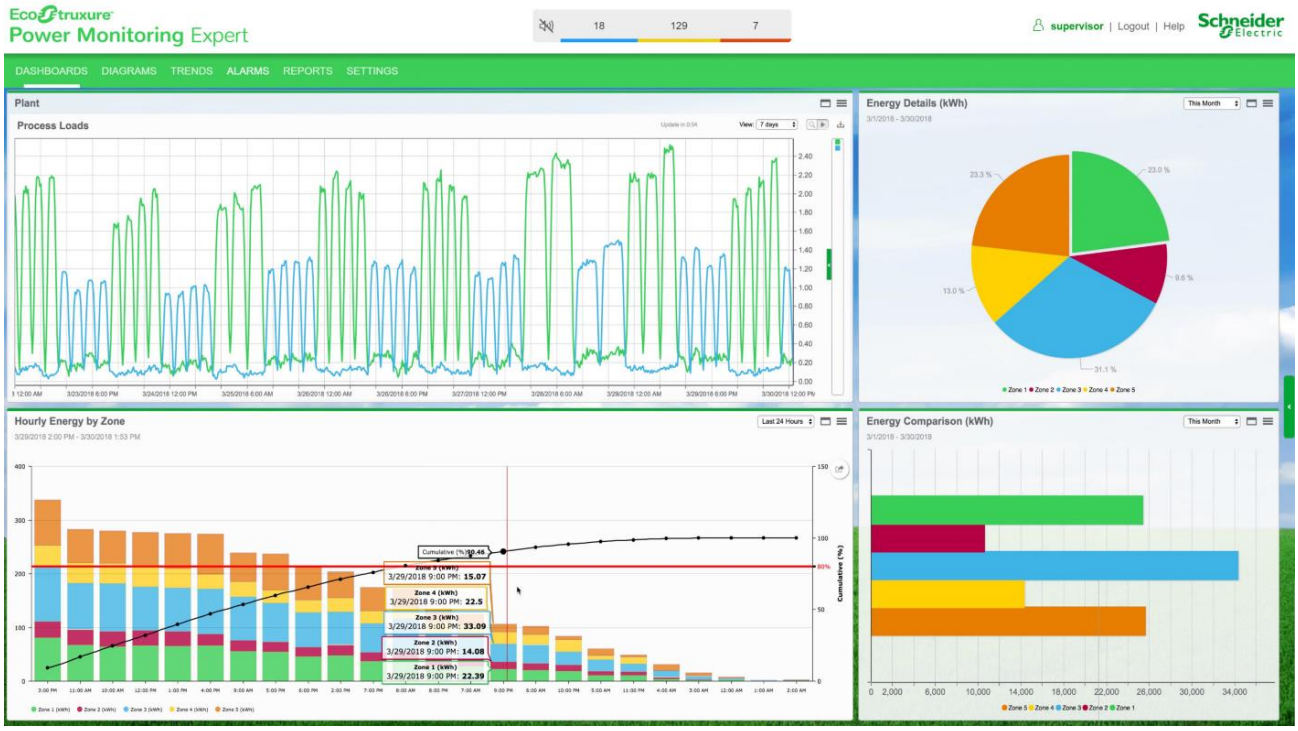



Рис. 2.1 Приклад екранної форми Schneider Electric EcoStruxure Power Monitoring Expert

Аналізуючи сімейство систем збору та аналізу даних від компанії Schneider Electric, я виявив певні переваги та недоліки їхнього ПЗ.

Переваги:

- широкий спектр аналітичної інформації та інструментів аналізу;
- набори шаблонів для кастомізації власних дашбордів з аналітичними даними;
- інструменти прямого порівняння періодів та прогнози споживання.

Недоліки:

- прив’язка виключно до продуктів компанії;

- складість налаштування і використання для малих і середніх клієнтів;
- необхідність додаткового кваліфікованого персоналу та спеціалізованого обладнання для впровадження системи.

### 2.1.2 Ability™ Energy and Asset Management (ABB)

Особливості системи:

- розроблено для комплексного управління енергоресурсами та активами;
- використовує власну екосистему пристроїв ABB;
- підтримує роботу з ModBus RTU та іншими промисловими протоколами.

Зручність управління:

- сучасний інтерфейс з drag-and-drop функціоналом (Рис. 2.2);
- гнучкі робочі області для різних фахівців;
- деталізована візуалізація потоків енергії.

Налаштування та впровадження:

- вимагає сертифікованих фахівців для впровадження;
- тривалий процес налаштування (від 2 тижнів);
- обмежена підтримка нестандартних конфігурацій.

Аналітичні можливості:

- потужні інструменти кореляційного аналізу;
- можливість побудови складних моделей енергоспоживання;
- вбудовані механізми виявлення аномалій.

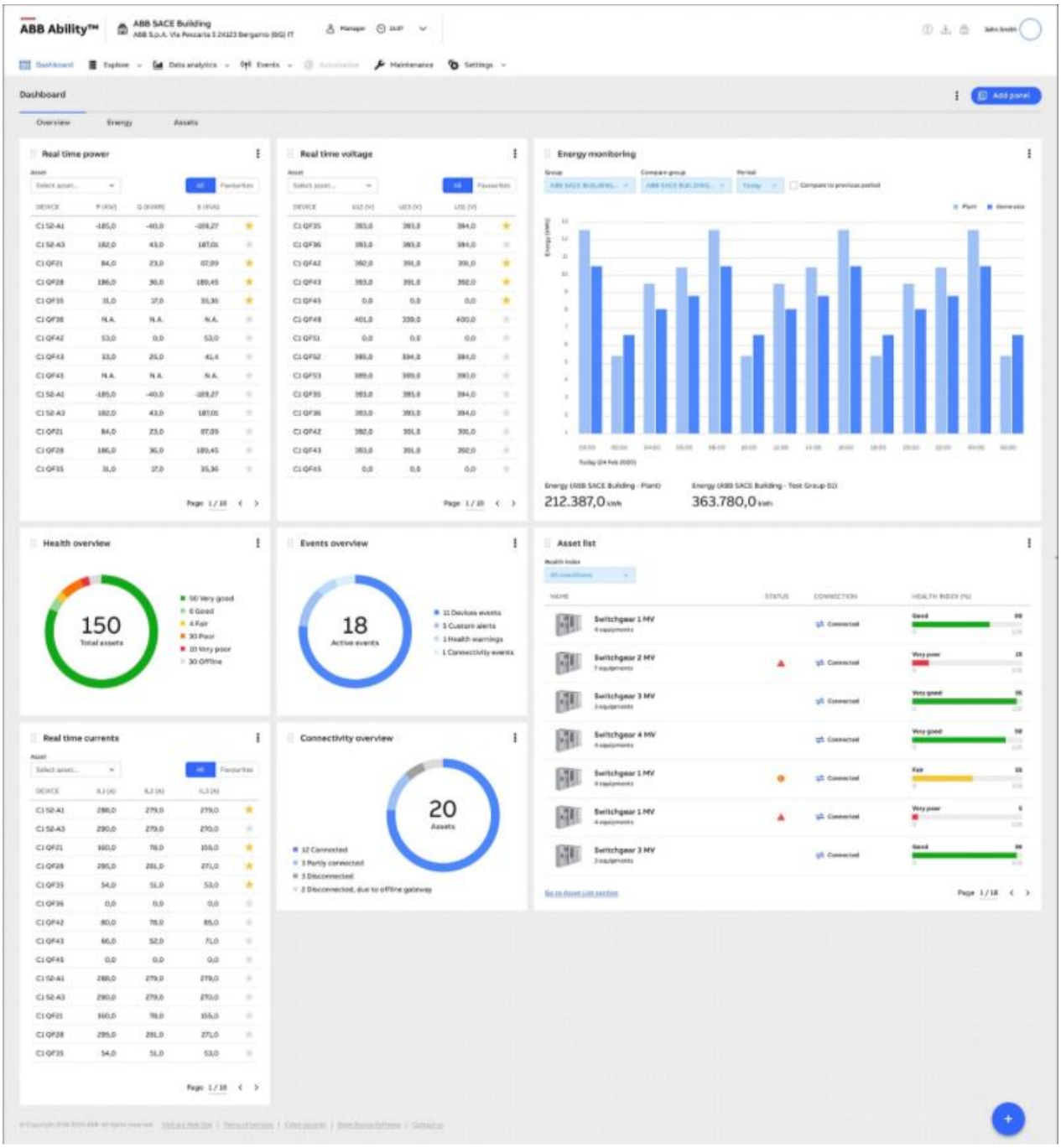



Рис. 2.2 Приклад екранної форми Ability™ Energy and Asset Management

Аналіз переваг та недоліків

Переваги:

- поглиблений аналіз енергоефективності;
- інструменти для управління життєвим циклом обладнання;
- можливість інтеграції з іншими системами ABB.

Недоліки:

- висока фінансова та часова вартість встановлення та підтримки;
- складність адаптації для малих підприємств;
- обмежена сумісність з обладнанням інших виробників.

### 2.1.3 Siemens Power Manager

Особливості системи:

- розроблено для промислового енергетичного моніторингу;
- оптимізовано для роботи з обладнанням Siemens;
- підтримує ModBus TCP та власні протоколи Siemens.

Зручність управління:

- чітко структурований промисловий інтерфейс, що можна побачити на Рис. 2.3;
- мобільний додаток для оперативного контролю;
- система тривожних сповіщень.

Налаштування та впровадження:

- жорстка прив'язка до апаратної платформи Siemens;
- висока складність кастомізації;
- обмежена документація для сторонніх інтеграцій.

Аналітичні можливості:

- стандартні набори аналітичних інструментів;
- можливість інтеграції з SIEMENS MindSphere;
- базові функції порівняльного аналізу.

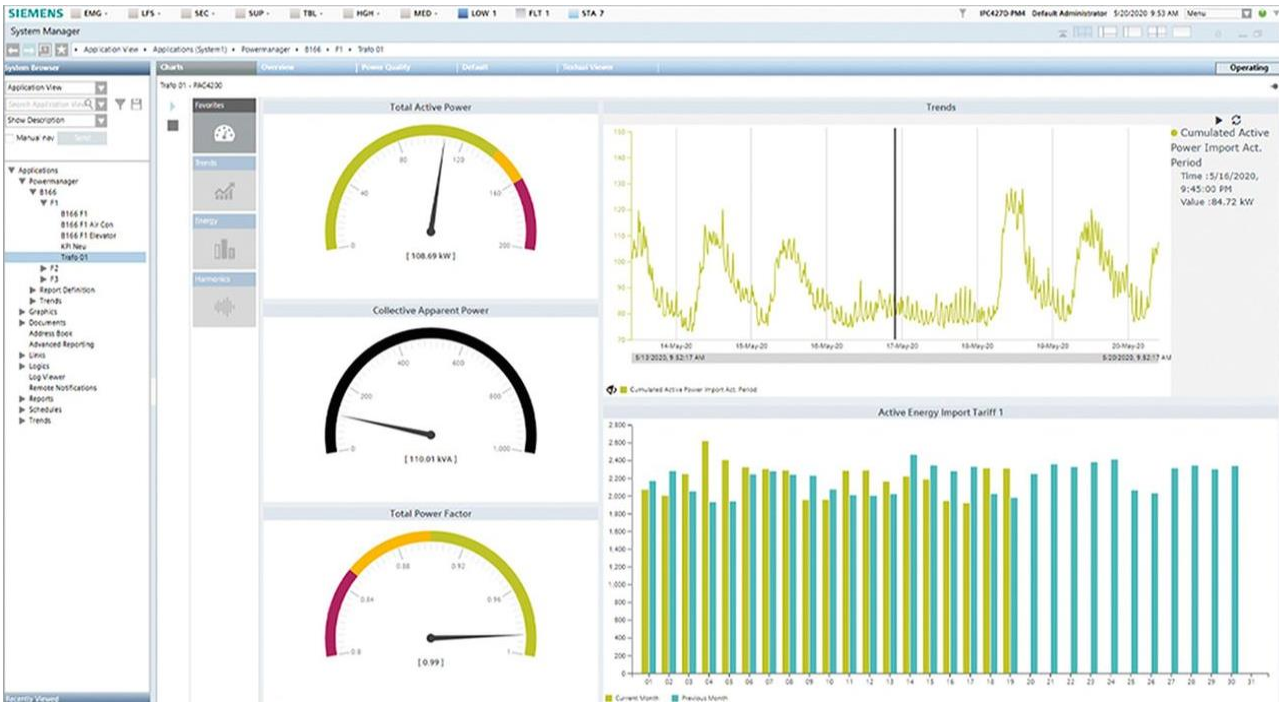



Рис. 2.3 Приклад екранної форми Siemens Power Manager

Аналіз переваг та недоліків

Переваги:

- висока надійність і стабільність роботи;
- готові рішення для промислових підприємств;
- інтеграція з іншими продуктами Siemens.

Недоліки:

- обмежена гнучкість у налаштуванні;
- складність адаптації під нестандартні вимоги;
- жорстка прив'язка до апаратної платформи Siemens.

Система Siemens Power Manager є «найбільш промисловою» системою з усіх розглянутих, вона є складною як в розгортанні і налаштуванні так і в повсякденному користуванні, зазвичай її використовують на дуже великих підприємствах, з метою уніфікувати усі датчики, лічильники та аналізатори в одній системі. Для керування нею часто наймається інженер з роботи з цією системою, що не є кращим вибором для малого та середнього бізнесу.

## 2.2 Зведені результати аналізу аналогів

Таблиця 2.1

Зведені результати аналізу характеристик ПЗ для збору та аналізу даних з електролічильників

| Показник | Schneider Electric EcoStruxure Power Monitoring Expert | Ability™ Energy and Asset Management | Siemens Power Manager |
|---|---|---|---|
| Платформа | Windows Server | Linux/Windows | Windows Server |
| Інтерфейс | Web-інтерфейс | Drag-and-drop | Промисловий |
| Необхідність додаткового устаткування | + | + | + |
| Прив’язка до устаткування компанії | + | + | + |
| Ролі користувачів | Адміністратор, Оператор, Гість | Адміністратор, Енергетик, Аналітик | Адміністратор, Оператор, Технік |
| Механізми безпеки | LDAP-аутентифікація, Розмежування прав доступу | Двофакторна аутентифікація, Шифрування даних | Рольова модель доступу, Аудит дій |
| Підтримка SCADA | + | - | + |
| Цільова аудиторія | Великі підприємства | Великі промислові об'єкти | Великі промислові підприємства |
| Інші особливості | Вбудований калькулятор вуглецевого сліду, інтеграція з системами BMS, підтримка AR-інтерфейсів для технічного обслуговування | Розширена система керування життєвим циклом обладнання, AI-аналіз стану обладнання, інтеграція з ERP-системами | Вбудовані шаблони відповідності стандартам ISO 50001 [10], підтримка цифрових двійників обладнання, спеціальні модулі для галузевих рішень |

## 2.3 Визначення вимог до застосунку

Для розробки ефективної та зручної системи збору й аналізу даних з електролічильників за технологією ModBus необхідно чітко визначити функціональні та нефункціональні вимоги. З огляду на цільове призначення програмного забезпечення — простота використання для звичайного користувача (наприклад, власника магазину або адміністратора підприємства) — система повинна бути інтуїтивно зрозумілою, стабільною та придатною для самостійного налаштування без залучення спеціалістів.

До основних **функціональних вимог** належить забезпечення можливості зчитування даних з аналізаторів енергоспоживання, які працюють за протоколом ModBus через інтерфейс RS485. Система повинна підтримувати додавання нових електролічильників, їх налаштування та видалення з конфігурації без перезапуску основного застосунку. Кожен пристрій зберігається в локальній базі даних, що також містить історію збору параметрів, дозволяючи формувати звіти у форматі .xlsx за обрані часові періоди. Інтерфейс повинен містити візуальні засоби для аналізу — графіки, таблиці, цифрові індикатори — з можливістю вибору діапазону часу та порівняння періодів.

Система має реалізовувати механізм авторизації, який дозволяє відрізняти адміністратора та гостя. Адміністратор після реєстрації має повний доступ до функціоналу — зміна налаштувань, додавання нових пристроїв, управління проєктами тощо. Гість може лише переглядати дані, без можливості внесення змін. Це дозволяє забезпечити базовий рівень розмежування прав доступу без складних серверних механізмів.

Серед **нефункціональних вимог** ключовими є вимоги до стабільності, продуктивності та універсальності. Система має забезпечувати безперебійний асинхронний збір даних у фоновому режимі — навіть при згортанні графічної оболонки додаток має продовжувати свою роботу в системному треї. Частота збору даних залежить від швидкодії фізичного з'єднання та обраних параметрів, але частота запису звітів може бути гнучко налаштована користувачем (від 2 до 59 хвилин). Завдяки використанню локальної бази даних (SQLite) система не

потребує серверної інфраструктури, а її архітектура дозволяє запуск на широкому спектрі пристроїв — від стандартних ПК з Windows до вбудованих систем на базі ARM64 [11] (наприклад, Raspberry Pi під керуванням Linux). Також передбачено можливість розширення підтримки нових моделей лічильників через додавання відповідних конфігурацій без істотної модифікації основного коду.

Вимоги до інтерфейсу передбачають інтуїтивну побудову навігації, чітке візуальне відображення стану системи, швидкий доступ до поточних та історичних даних, а також можливість зміни мови інтерфейсу. Крім того, система повинна бути розширювана і гнучка — користувач повинен мати змогу оновлювати налаштування або конфігурацію пристроїв без необхідності перезапуску програми або втрати поточних даних.

### 2.3.1 Функціональні вимоги

- можливість зчитування даних з електролічильників за протоколом ModBus (через інтерфейс RS485);
- підтримка додавання, редагування та видалення пристроїв без перезапуску програми;
- збереження даних у локальній базі даних з історією параметрів;
- формування звітів у форматі .xlsx за обрані часові періоди;
- наявність графіків, таблиць та індикаторів для візуального аналізу даних;
- механізм реєстрації та авторизації користувача (адміністратор або гість);
- адміністратор має повний доступ, гість — лише перегляд даних;
- налаштування періодичності запису даних до звітів (від 2 до 59 хвилин).

### 2.3.2 Нефункціональні вимоги

- система має бути простою у використанні — встановлення, налаштування та експлуатація без технічних знань;

- графічний інтерфейс працює окремо від модуля збору даних (архітектура з розділенням логіки GUI та фонових процесів);
- у разі закриття графічного вікна програма згортається в системний трей, не припиняючи збір даних;
- частота збору даних залежить від якості лінії зв'язку та конфігурації;
- застосунок є десктопним, автономним, не потребує доступу до інтернету;
- усі дані зберігаються локально (SQLite);
- підтримка запуску як на Windows, так і на Linux (включно з ARM64 пристроями, наприклад Raspberry Pi);
- можливість розширення підтримки нових моделей пристроїв без суттєвої модифікації коду.

## 2.4 Результати аналізу аналогів

У ході аналізу програмного забезпечення, призначеного для збору та обробки даних з аналізаторів електричних мереж за допомогою протоколу Modbus, було виявлено, що більшість існуючих рішень мають суттєві обмеження щодо гнучкості, масштабованості та адаптивності під конкретні сценарії використання. Частина з них є комерційними та вимагає придбання ліцензії, інші ж мають обмежений функціонал або складну процедуру розгортання та налаштування, що ускладнює їхнє застосування в невеликих чи кастомних проєктах.

Також було встановлено, що більшість таких систем орієнтовані на використання у промислових умовах, часто не мають зручного або адаптивного інтерфейсу, що ускладнює доступ до інформації для широкого кола користувачів.

Аналіз дозволив виявити ключові функціональні та нефункціональні вимоги до власної розробки: підтримка протоколу Modbus RTU, зручний мінімалістичний інтерфейс, сумісність з різними ОС і архітектурами, можливість адаптації до нестандартних пристроїв, а також прозорість і контроль над усіма етапами збору та обробки даних. Отже, створення індивідуального застосунку дозволяє повністю покрити специфічні потреби проєкту та уникнути недоліків існуючих аналогів.

# 3 ЗАСОБИ РЕАЛІЗАЦІЇ

## 3.1 Середовище розробки

Для розробки програмного забезпечення у даному проєкті було обрано інтегроване середовище розробки (IDE) PyCharm Professional 2025 від компанії JetBrains. Це середовище є одним із найпопулярніших інструментів серед Python-розробників завдяки своїй гнучкості, широкому набору функціональних можливостей та зручному інтерфейсу.

PyCharm надає повноцінну підтримку для розробки десктопних додатків із графічним інтерфейсом, працюючи в поєднанні з такими бібліотеками як PySide6 (Qt), matplotlib, pandas, asyncio та іншими. Крім цього, IDE має вбудовану підтримку систем контролю версій (Git), інструменти для інтерактивної відладки, профілювання продуктивності коду, написання юніт-тестів, перевірки якості коду за допомогою lint-аналізу, автоматичного форматування тощо.

Під час роботи над проєктом активно використовувались такі можливості PyCharm:

- інтелектуальний автозавершувач коду;
- інтеграція з віртуальним середовищем (venv [12]);
- підтримка плагінів, зокрема для перегляду вмісту таблиць бази даних;
- інтерфейс з Docker для можливого майбутнього розгортання;
- візуальний перегляд структури класів, баз даних, сигналів та слотів (у випадку PySide).

Крім того, PyCharm дозволяє ефективно працювати з файлами ресурсів, зокрема з .ui файлами, що згенеровані в Qt Designer, а також із графічними ресурсами, які використовуються в інтерфейсі програми (іконки, шрифти тощо).

Операційна система, на якій велася розробка — Windows 11 x64, проте код протестовано й у середовищах Linux (Debian-based дистрибутиви) та ARM64 (Raspberry Pi OS).

### 3.2 Мова програмування

Основною мовою реалізації програмного забезпечення було обрано мову Python версії 3.12. Python є інтерпретованою, об’єктно-орієнтованою мовою програмування високого рівня, яка завдяки своїй простоті, читабельності коду та великій екосистемі бібліотек стала ідеальним вибором для створення десктопних застосунків, що працюють із обладнанням через послідовні порти (наприклад, RS485), мають графічний інтерфейс та вимагають багатопоточну архітектуру.

Серед переваг Python, які стали вирішальними при виборі:

- висока швидкість розробки та простота підтримки коду;
- потужна стандартна бібліотека (модулі для роботи з потоками, файлами, датою, базами даних тощо);
- великий вибір сторонніх бібліотек для роботи з протоколами зв’язку, графікою, базами даних;
- кросплатформеність — Python-код може бути запущено на Windows, Linux, ARM-пристроях без змін.

У межах даного проєкту використовувались такі ключові бібліотеки:

- PySide6 [13-14] — для створення графічного інтерфейсу користувача (Qt for Python). Вона включає в себе багато різних пакетів, в тому числі для роботи з потоковістю;
- PySerial — для роботи з послідовним портом RS485;
- pymodbus — для реалізації взаємодії за протоколом Modbus RTU;
- Toritoise-orm — вбудована підтримка баз даних SQLite з асинхронним доступом;

- matplotlib — побудова графіків на основі зібраних даних;
- pandas — для обробки та підготовки даних до звітності;
- xlsxwriter — для формування звітів у форматі Excel (.xlsx);
- asyncio та threading — для реалізації асинхронного та багатопоточного збору даних у фоновому режимі.

Python, як мова з відкритим кодом, дозволила не лише скоротити витрати на розробку, а й забезпечила високу гнучкість та масштабованість застосунку. Обрані інструменти дозволяють при потребі легко адаптувати програму до нових моделей пристроїв, змін у протоколах обміну або майбутнього розширення функціоналу.

### 3.3 Основні бібліотеки

У процесі розробки програмного забезпечення було використано низку сторонніх бібліотек і фреймворків, які значно пришвидшили та спростили реалізацію ключових функцій застосунку. Всі бібліотеки є відкритими, активно підтримуються спільнотою, мають якісну документацію, що стало вирішальним фактором при їх виборі.

#### 3.3.1 PySide6

PySide6 — це офіційна обгортка мови Python для бібліотеки Qt 6 (Qt for Python), яка надає повноцінний доступ до всіх можливостей багатоплатформенного фреймворку Qt. Вона дозволяє створювати сучасні, адаптивні, інтуїтивно зрозумілі графічні інтерфейси користувача. У рамках проєкту PySide6 виконує основну функцію побудови GUI, включаючи структуру основного вікна, діалогових форм, кнопок, меню, віджетів відображення таблиць та графіків.

Серед переваг використання PySide6:

- кросплатформеність: додаток виглядає однаково на Windows, Linux та macOS;

- інтеграція асинхронних процесів: бібліотека нативно дозволяє виконувати певні дії в окремих потоках;
- можливість використання сучасних стилів, анімацій, систем сигналів і слотів;
- повна інтеграція з PyQt-подібною архітектурою обробки подій.

#### 3.3.1.1 Qt

Qt — це C++-бібліотека з відкритим вихідним кодом, яка вже багато років є стандартом у створенні кросплатформених графічних додатків. Її головною перевагою є надзвичайна гнучкість та модульність. Вона підтримує компоненти для візуалізації графіки, роботи з файлами, мережею, базами даних, мультимедіа тощо.

**Завдяки Qt забезпечується:**

- стабільна і швидка робота графічного інтерфейсу;
- побудова віконних інтерфейсів з підтримкою тем оформлення;
- адаптація до різних роздільностей екрану та DPI;
- відображення кастомних віджетів і графіків.

### 3.3.2 XlsxWriter

XlsxWriter — це потужна Python-бібліотека, призначена для створення файлів Excel (.xlsx) без необхідності встановлення Microsoft Office. У межах проєкту ця бібліотека використовується для генерації звітів на основі зібраних із пристроїв даних. Вона дозволяє форматувати таблиці, додавати стилі, об'єднувати клітинки, використовувати формули, створювати діаграми.

Основні переваги:

- висока продуктивність — навіть великі файли створюються швидко;
- підтримка Unicode [15] — важливо для української мови;
- можливість повністю контролювати структуру звіту;
- незалежність від зовнішніх офісних пакетів.

Таким чином, за допомогою **XlsxWriter** реалізовано функціонал автоматичного створення та збереження детальних звітів по кожному пристрою з точним зазначенням часу, типу параметрів та значень.

### 3.3.3 Rich

**Rich** — це бібліотека Python, яка дозволяє створювати красиво оформлені текстові інтерфейси в терміналі. У рамках даного застосунку вона інтегрована у вигляді консольного віджета всередині графічного інтерфейсу (на сторінці деталей пристрою) для відображення повідомлень, логів, системних подій, стану з'єднання, результатів запитів тощо.

Rich дозволяє:

- застосовувати кольорове форматування до тексту;
- використовувати структури таблиць, панелей, списків;
- виділяти помилки, попередження, успішні операції різними кольорами;
- зручно форматувати текст у реальному часі, що підвищує читаємість логів.

Це дозволило створити прозору, візуально зручну систему логування, яка доступна навіть для недосвідчених користувачів — повідомлення зрозумілі, добре помітні, легко інтерпретуються без необхідності відкривати зовнішні логи або консолі.

## 3.4 Система управління базами даних (СУБД) та ORM

Для зберігання, організації та обробки даних у межах застосунку використовується вбудована база даних **SQLite** у поєднанні з ORM-фреймворком **Tortoise ORM**. Такий підхід дозволяє досягти високої гнучкості, простоти розгортання та зручності у подальшому супроводі. Усе працює локально, без потреби в окремому сервері баз даних або додатковому налаштуванні з боку користувача.

### 3.4.1 SQLite

**SQLite** — це легка **реляційна база даних**, яка працює без окремого серверного процесу. Вона ідеально підходить для настільних додатків, таких як мій, де немає потреби в багатокористувацькому доступі через мережу. База даних є звичайним файлом, який автоматично створюється та оновлюється програмою.

Переваги використання SQLite у моїй програмі:

- простота розгортання: не потребує встановлення СУБД [16] окремо — усе вбудовано в застосунок;
- висока продуктивність для локальних додатків і невеликих обсягів даних;
- збереження даних у єдиному файлі, що спрощує резервне копіювання;
- підтримка стандартного SQL для запитів;
- достатньо потужна для задач збору та аналізу даних з до 128 пристроїв одночасно, як у моєму випадку.

У SQLite зберігаються всі ключові сутності програми: користувачі, пристрої, проєкти, звіти, розклади, параметри моделі пристрою, тощо.

### 3.4.2 Tortoise ORM

Для зручного та гнучкого доступу до бази даних я використовую **Tortoise ORM** — сучасну асинхронну бібліотеку, яка дозволяє працювати з базами даних на основі об’єктно-реляційного відображення (ORM). Це означає, що вся взаємодія з даними відбувається через Python-класи, а не через текстові SQL-запити. Такий підхід робить код чистішим, безпечнішим, зрозумілішим для розробника, зменшує кількість помилок при роботі з базою даних.

Основні переваги використання Tortoise ORM у моєму проєкті:

- мінімальний поріг входу — структура моделей інтуїтивно зрозуміла;
- підтримка типів полів, зовнішніх ключів, відношень OneToMany, ManyToMany [17];
- можливість міграцій (через бібліотеку Aerich);

- гнучке налаштування моделей: кожна таблиця — окремий Python-клас;
- автоматичне створення структури бази даних під час запуску застосунку.

#### 3.4.2.1 Інтеграція з asyncio

Tortoise ORM повністю орієнтований на асинхронне програмування та працює на основі Python-механізму **asyncio**. Це дуже важливо для мого застосунку, оскільки збір даних із пристроїв відбувається паралельно у фонових потоках, без блокування головного графічного інтерфейсу. Завдяки такій інтеграції можливо:

- одночасно опитувати декілька пристроїв без втрати продуктивності;
- продовжувати роботу інтерфейсу без зависань навіть при великому обсязі даних;
- виконувати запис даних у базу в реальному часі паралельно з виведенням повідомлень у логах;
- легко масштабувати функціонал збору без зміни архітектури.

Уся взаємодія з базою даних (запис звітів, зчитування історії, оновлення параметрів) виконується через async/await, що забезпечує високу стабільність та швидкість роботи програми.

## 3.5 Система контролю версій

Для керування кодовою базою, синхронізації змін між різними середовищами розробки, ведення історії змін і командної роботи використовується система контролю версій Git у поєднанні з хостинговою платформою GitHub**.** Це дозволяє підтримувати надійний процес розробки, забезпечувати резервне копіювання, а також ефективно розмежовувати стабільний і експериментальний функціонал програми.

### 3.5.1 Git та GitHub

Git — це розподілена система контролю версій, яка дозволяє зберігати історію змін, працювати з гілками, зливати їх між собою (merge), відстежувати відмінності між версіями та ефективно контролювати кожен етап розробки. Я використовую Git на всіх етапах розробки — від раннього прототипування до підготовки релізу.

GitHub — це хмарний сервіс, який дозволяє зручно розміщувати репозиторій, керувати віддаленими гілками, створювати pull request'и, перевіряти історію змін, працювати з issues та автоматизувати CI/CD (у майбутньому це може бути корисно для автозбірок). Хостинг у GitHub також забезпечує можливість розгортання тестових чи релізних версій прямо на ARM-пристрої.

### 3.5.2 Правила поділу на гілки

Для організації розробки я використовую перевірений підхід **git flow**. Це стандартна схема, яка дозволяє розділити стабільну частину коду (main/master) від розробки (develop), а також забезпечити контроль над тимчасовими гілками для додавання нових функцій чи виправлення помилок.

У моєму випадку структура гілок виглядає наступним чином:

- main — стабільна гілка для релізів, яка містить лише перевірений код;
- develop — основна гілка активної розробки;
- feature/* — тимчасові гілки для розробки окремих функцій або модулів;
- bugfix/* — гілки для виправлення конкретних помилок;
- release/* — підготовка до випуску (інтеграція кількох функцій);
- hotfix/* — термінове виправлення критичних помилок безпосередньо в main.

Додатково, з урахуванням того, що мій **застосунок повинен запускатися на ARM-пристроях** (наприклад, Raspberry Pi), я створив спеціалізовані гілки:

- linux-arm-develop — гілка для активної розробки безпосередньо на ARM-системах;

– linux-arm-release — стабільна ARM-збірка, яка не залежить від x86-розробки.

Ці гілки існують окремо через складність або навіть неможливість ефективної крос-компіляції графічного застосунку на PySide6 для ARM у середовищі x86. Розробка в цих гілках ведеться безпосередньо на пристрої, з урахуванням його обмежень.

### 3.5.3 Правила написання комітів

У моїй практиці я використовую структурований підхід до оформлення комітів, заснований на стилі **Conventional Commits** [19]. Це — набір правил, що описує, як формулювати повідомлення комітів, щоб вони були однозначно зрозумілими, машинно-оброблюваними та зручними для побудови історії змін, генерації changelog'ів та автоматичного семантичного версіонування.

Основний формат:

**тип: короткий опис зміни**

Основні типи комітів, які я використовую:

– feat: Додано новий функціонал (наприклад, feat: додано вікно налаштувань);
– fix або bug: Виправлення помилки (наприклад, fix: не оновлювались значення в таблиці);
– refactor: Переробка коду без зміни поведінки (наприклад, refactor: спрощено логіку refresh-обробника);
– chore: Службові зміни (оновлення залежностей, налаштування CI тощо)
– docs: Зміни у документації;
– test: Додавання або зміна тестів;

Такий підхід має багато переваг:

– забезпечує послідовність у повідомленнях комітів;
– полегшує пошук по історії (наприклад, всі fix: або feat: легко знаходяться через git log);

- дозволяє автоматично генерувати зміст змін для релізів (changelog);
- формує культуру структурованого внесення змін;
- значно спрощує командну роботу в майбутньому.

### 3.6 Результати підбору засобів реалізації

У цьому розділі було розглянуто та обґрунтовано вибір інструментів, бібліотек і технологій, які стали основою для реалізації системи збору та візуалізації даних з електролічильників за допомогою протоколу Modbus RTU. Основною мовою програмування було обрано Python через його гнучкість, велику кількість бібліотек для роботи з асинхронними задачами, а також підтримку сучасних фреймворків для створення графічного інтерфейсу.

Для побудови інтерфейсу користувача використано бібліотеку PySide6, яка забезпечує кросплатформену підтримку та дозволяє створювати гнучкі й адаптивні графічні компоненти без потреби у використанні файлів .ui. Обрано темну тему оформлення з Fusion-стилем, що забезпечує однаковий зовнішній вигляд застосунку на різних операційних системах.

Для забезпечення асинхронного зчитування даних через послідовний порт застосовано asyncio в поєднанні з кастомним модулем SerialReaderRS485. Архітектура побудована з урахуванням принципів модульності та розмежування відповідальностей. Збереження та обробку даних реалізовано за допомогою Tortoise ORM з використанням SQLite як зручного рішення для вбудованої бази даних.

Обрані інструменти дозволили досягти високої гнучкості, розширюваності та зручності в розробці та тестуванні системи. Таким чином, реалізація на базі цих засобів забезпечила відповідність як функціональним, так і нефункціональним вимогам, що були визначені на попередніх етапах.

# 4 ПРОЄКТУВАННЯ

## 4.1 Проєктування архітектури системи

Архітектура системи була розроблена з урахуванням її розподіленості та необхідності синхронної роботи двох основних компонентів: графічного інтерфейсу користувача та модуля збору даних. Для досягнення максимальної ефективності та стабільності в роботі системи була реалізована паралельна обробка даних, що дозволяє системі функціонувати без затримок навіть при інтенсивному зборі інформації.

Основні компоненти системи:

1. **Графічний інтерфейс користувача (GUI)**. Використовує єдину головну форму (MainWindow), в якій відображаються всі екрани системи. Кожен екран у вигляді віджета вставляється у вертикальний віджет головного вікна. Всі екрани пов’язані між собою через передачу об’єктів даних, таких як проєкти, пристрої чи деталі пристроїв, для подальшого їх використання на наступних етапах взаємодії. Це дозволяє зберігати контекст переходів між екранами та забезпечує логічну цілісність даних при їх оновленні.
2. **Модуль збору даних**. Цей модуль працює асинхронно, окремо від графічного інтерфейсу, щоб уникнути блокування віконного інтерфейсу. Він працює в фоновому режимі, здійснюючи періодичний збір і оновлення даних з пристроїв за допомогою таймерів та асинхронних задач. Взаємодія з модулем збору даних не порушує роботу основного інтерфейсу, і користувач може продовжувати роботу з програмою без затримок.
3. **Система оновлень даних**. Для оновлення даних у системі передбачено кілька механізмів. Користувач може вручну оновлювати дані через відповідні кнопки на екрані, що ініціюють запити до модуля збору даних. Об’єкти даних, такі як проєкти, пристрої та деталі пристроїв, взаємодіють

один з одним через метафункціональність, що дає можливість зберігати їхній контекст під час переходів між екранами.

4. **Асинхронність і незалежність компонентів**. Важливим аспектом архітектури є використання асинхронного програмування для обробки даних, що дозволяє уникнути блокування інтерфейсу користувача при виконанні складних обчислень або операцій збору даних. Використовуються потоки та асинхронні задачі для збору, зберігання і оновлення даних без втрати продуктивності інтерфейсу.

### 4.2 Проєктування графічного модуля

Графічний інтерфейс програми був розроблений з основним акцентом на простоту використання та мінімалізм. Цей підхід дозволяє користувачам з невеликим технічним досвідом швидко освоїти програму без потреби в глибокому розумінні технічних аспектів її роботи.

Основні принципи проектування:

1. **Єдине головне вікно (MainWindow)**. Весь інтерфейс побудований навколо одного головного вікна, яке містить всі необхідні екрани системи. Замість численних вкладок або розділених вікон, вся функціональність інтегрована в єдиному інтерфейсі, де кожен екран подається у вигляді вкладеного віджета в головному вікні. Такий підхід забезпечує зручність навігації та логічну організацію інтерфейсу, при цьому дозволяє повернутися на екран назад, без втрати його вмісту і без необхідності повторного завантаження даних.
2. **Передача об’єктів між екранами**. На кожному етапі користувач може вибирати елементи, що передаються на наступний екран для збереження контексту. Наприклад, після вибору проєкту в списку, об'єкт цього проєкту передається на екран пристроїв для подальшої роботи. Той самий принцип працює і для пристроїв, що передаються на екран деталей пристрою.

3. **Мінімальна кількість елементів управління**. Графічний інтерфейс побудовано таким чином, щоб на екрані одночасно з’являлося мінімум елементів управління, що не відволікають користувача від основної мети. Кожна дія користувача (наприклад, вибір проєкту або пристрою) відображена через прості й очевидні кнопки, що безпосередньо зв’язані з елементами, до яких застосовуються ці дії (наприклад, кнопка редагування пристрою на його тайлі).
4. **Контекстні дії через кнопки на тайлах**. Для кожного елементу (проєкт, пристрій) реалізовані контекстні кнопки безпосередньо на їхніх тайлах. Це дозволяє користувачеві виконувати основні операції, такі як додавання нового проєкту чи пристрою, редагування або видалення, без необхідності переходити в інші екрани чи меню.
5. **Відсутність блокування інтерфейсу під час збору даних**. Віконна частина не заблокована при зборі даних завдяки використанню **асинхронних задач**. Таким чином, навіть коли програма виконує важкі операції з обробки чи збору даних, інтерфейс залишається доступним для користувача. Завдяки цьому можна продовжувати роботу з програмою без затримок, наприклад, оновлюючи або переглядаючи дані, не чекаючи на завершення операції збору.
6. **Простота навігації та інтуїтивно зрозуміле управління**. Вся система управління побудована таким чином, щоб користувач не витрачав час на пошук необхідних функцій. Навігація між екранами мінімальна, а всі основні операції відбуваються через прості й чітко позначені кнопки.

Загалом, графічний інтерфейс розроблений так, щоб він був зручним і зрозумілим навіть для користувачів з мінімальним досвідом в роботі з подібними програмами, водночас забезпечуючи всю необхідну функціональність для професійних користувачів.

Ці принципи дозволяють не тільки зберігати простоту, а й підвищувати продуктивність роботи з програмою, надаючи користувачеві можливість швидко та ефективно виконувати всі необхідні операції.

### 4.3 Проєктування модуля збору даних

**Модуль збору даних** на Рис. 4.1, є програмно відокремленою частиною загальної архітектури системи. Його основне завдання — забезпечити неперервний та стабільний процес опитування підключених пристроїв (лічильників) і збереження отриманих показників у базу даних. Цей модуль працює незалежно від графічного інтерфейсу користувача, що дозволяє уникнути блокувань або уповільнень у роботі інтерфейсу під час інтенсивного збору даних.

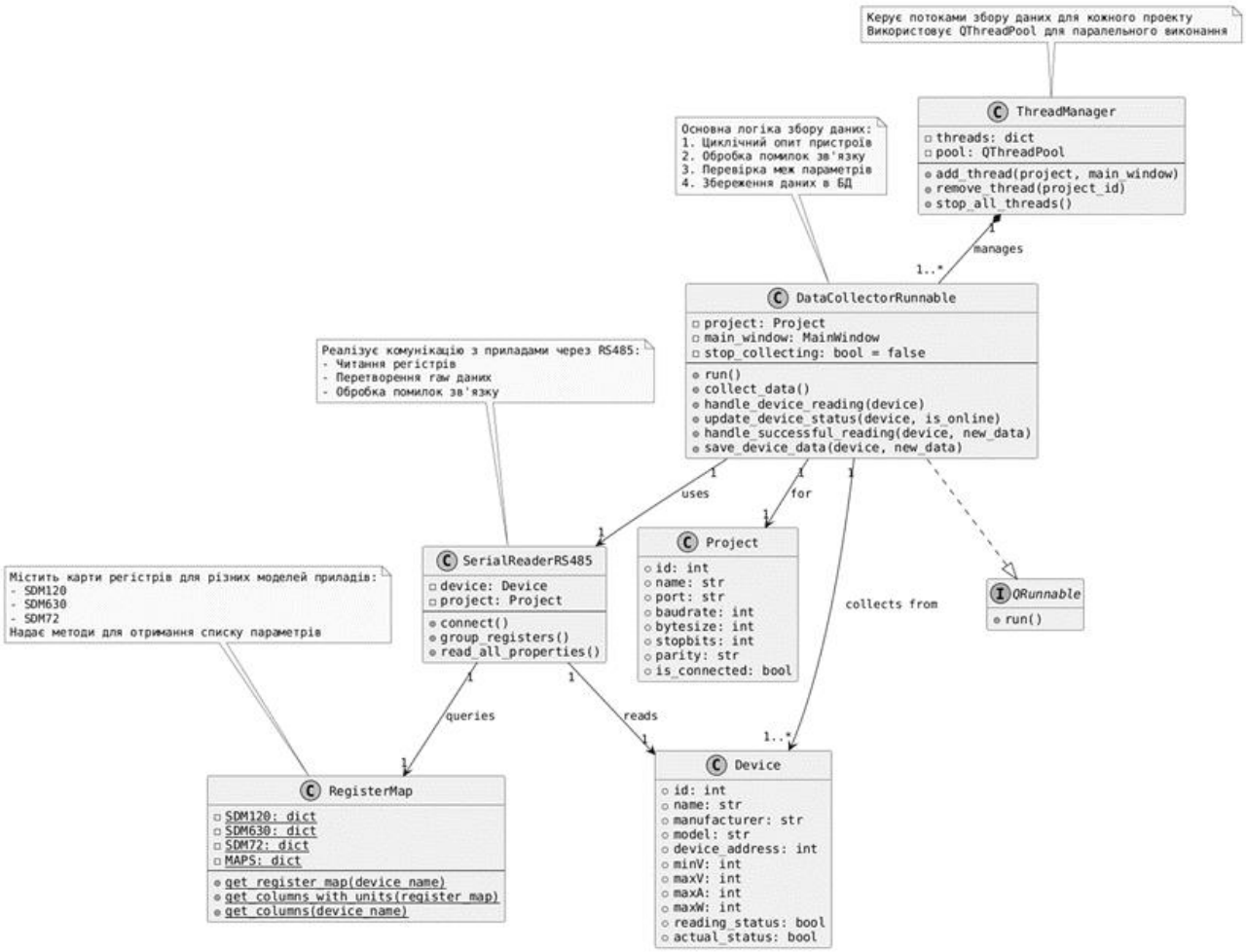



Рис. 4.1 Модуль збору даних

Усі дані, що надходять від лічильників, зберігаються **у двох таблицях**:

- **(Model)Report** — основна таблиця, призначена для збереження історичних даних, які надалі використовуються для побудови графіків, таблиць і статистичних аналізів;
- **(Model)ReportTmp** — таблиця тимчасових значень, яка використовується для оновлення в реальному часі. У цій таблиці для кожного пристрою зберігається лише один актуальний запис, що постійно оновлюється. Ці дані застосовуються, наприклад, на екрані з деталями пристрою для показу поточних значень у вигляді індикаторів

**Модуль збору** також включає механізм контролю за межами допустимих значень основних параметрів: напруги, струму та потужності. Якщо значення якогось із параметрів перевищує встановлені порогові межі, то відповідний звіт примусово зберігається в таблицю (Model)Report незалежно від поточного розкладу опитування. Це дозволяє зафіксувати критичні або пікові навантаження для подальшого аналізу.

Користувач має можливість задавати розклад збереження звітів у таких режимах:

- фіксований інтервал (від 2 до 59 хвилин);
- один раз на добу у визначений час (актуально для випадків, коли система використовується виключно з метою щоденного обліку споживання).

### 4.3.1 Система управління потоками

Ключовим елементом реалізації модуля збору даних є система потоків. Для кожного окремого проєкту (який фактично відповідає одній фізичній лінії RS-485) створюється окремий потік, який діє автономно. Кожен потік циклічно опитує всі пристрої, що входять до відповідного проєкту, та зберігає отримані дані в базу даних. Таким чином досягається масштабованість і незалежність опитування між різними проєктами.

Модуль реалізовано через три основних класи:

- **DataCollector.py** — клас, що відповідає за логіку одного потоку, включаючи опитування, зчитування параметрів та запис у відповідні таблиці бази даних;
- **ThreadManager.py** — клас, який керує створенням, видаленням і адмініструванням усіх потоків збору даних. Він зберігає список активних потоків та забезпечує їхню синхронізацію з діючими проєктами в системі. Наприклад, у випадку видалення проєкту, відповідний потік буде автоматично завершено та прибрано зі списку;
- **SerialReaderRS485.py** – клас, який реалізує алгоритм одного опитування в один звіт.

Такий підхід до організації збору даних дозволяє забезпечити надійну, масштабовану та продуктивну роботу системи, з акцентом на безперервність процесу та гнучкість конфігурації.

### 4.3.2 Алгоритм опитування

Алгоритм опитування пристрою реалізується в окремому класі SerialReaderRS485, який відповідає за низькорівневу комунікацію з пристроями через послідовний інтерфейс RS-485. Основне завдання цього класу — прочитати параметри з пристрою, об’єднати їх у відповідну структуру та повернути результат для подальшого збереження в базу даних.

На Рис. 4.2 представлений алгоритм опитування, що складається з етапів:

1. **Ініціалізація з’єднання:**

   Відбувається створення екземпляра SerialReaderRS485.

   Ініціалізується клієнт (Modbus) з усіма необхідними параметрами: порт, адреса пристрою, швидкість, бітність, парність, кількість байтів стопу, кількість спроб підключення тощо.

2. **Спроба з’єднання:**

   Проводиться спроба встановити з’єднання з пристроєм.

Якщо з’єднання не вдалось, помилка логуються, і функція повертає порожній словник. Це важливо для забезпечення безперервності роботи потоку збору даних: замість викидання виключення функція обробляє помилку «м’яко», не перериваючи роботу всієї системи.

3. **Оптимізація запитів:**

   Після успішного з’єднання всі параметри (регістри), які потрібно зчитати, групуються за адресами.

   Виконується об’єднання послідовних регістрів в одну групу. Наприклад, якщо потрібно зчитати регістри 1–2, 3–4 і 7–8, то алгоритм згенерує два запити: до регістрів 1–4 та 7–8. Це дозволяє зменшити кількість звернень до пристрою та пришвидшити опитування.

4. **Зчитування даних:**

   Кожна сформована група регістрів опитуються окремо.

   Якщо відповідь відсутня або містить помилку (наприклад, CRC-помилка, тайм-аут, некоректний розмір), результат логуються, і функція повертає порожній словник.

5. **Декодування даних:**

   Якщо відповідь дійсна, з неї виділяються значення регістрів.

   Дані декодуються відповідно до специфікації: підтримуються як однорегістрові, так і дворегістрові значення (WORD, DWORD, SDWORD, FLOAT).

6. **Повернення результату:**

   Усі коректно прочитані та декодовані значення формуються у словник, де ключами є імена параметрів, а значеннями — їх поточні показники.

   Цей словник повертається як результат функції.

**Обробка помилок:**

У всіх етапах передбачена обробка виняткових ситуацій, серед яких:

– Неможливість встановити з’єднання з пристроєм.
– Відсутність відповіді на запит.
– Помилкова відповідь (некоректна структура, CRC-помилка).

– Проблеми при закритті з’єднання.

У всіх випадках при виникненні помилки функція не викидає винятки назовні, а повертає порожній словник. Це дозволяє підтримувати стабільність роботи потоку збору даних і уникнути його аварійного завершення через одиничний збій у пристрої.

У фінальному блоці з’єднання обов’язково закривається, навіть якщо сталася помилка, що гарантує правильне звільнення ресурсів.

Цей алгоритм є основою для забезпечення ефективного та безпечного опитування пристроїв, що працюють через Modbus RTU, і є ключовим елементом у загальній системі збору та збереження енергетичних даних.

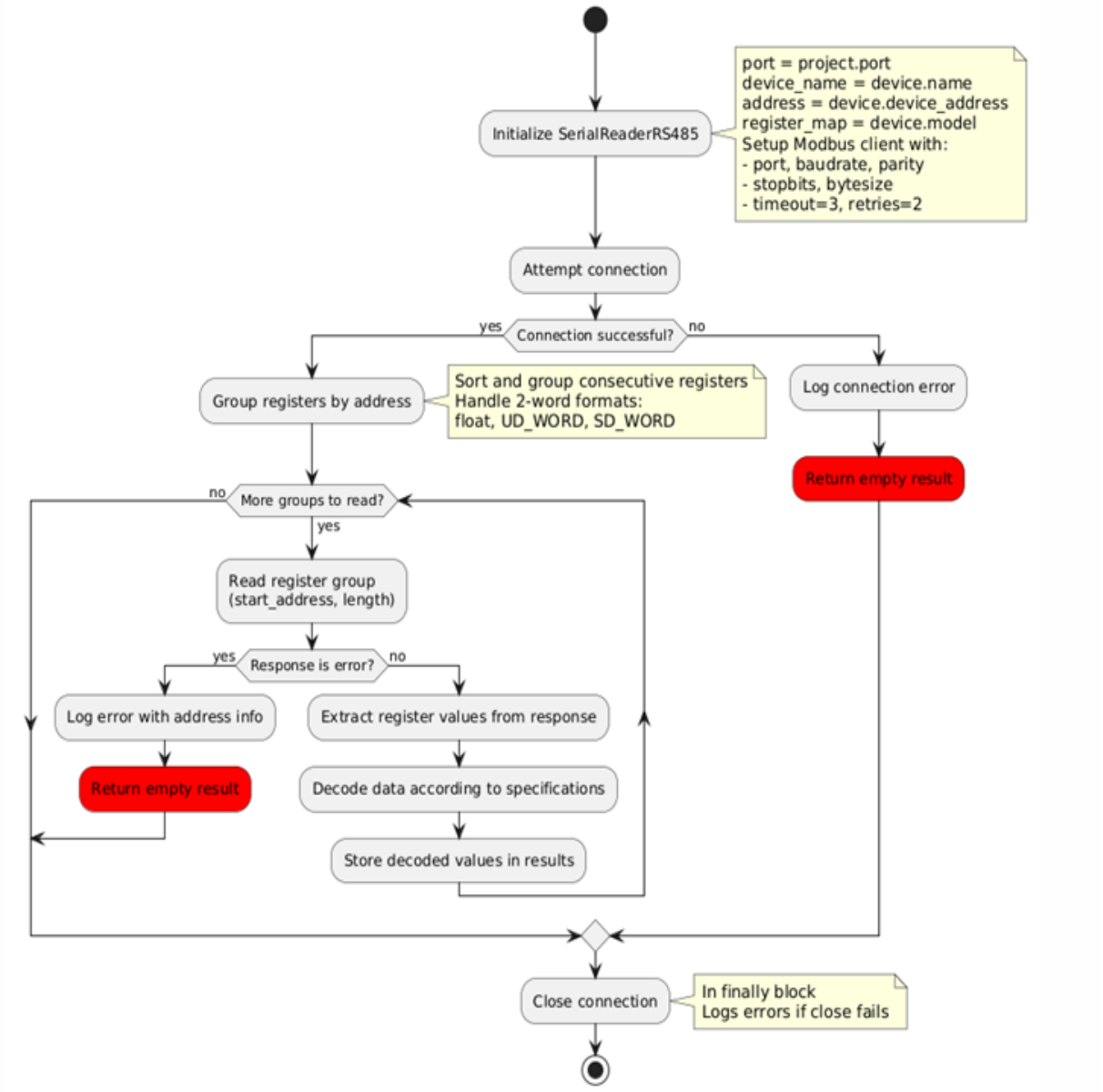



Рис. 4.2 Алгоритм опитування

### 4.3.3 Алгоритм роботи потоку збору даних

На Рис. 4.3 зображено повний цикл роботи одного потоку збору даних, реалізованого через клас DataCollectorRunnable.

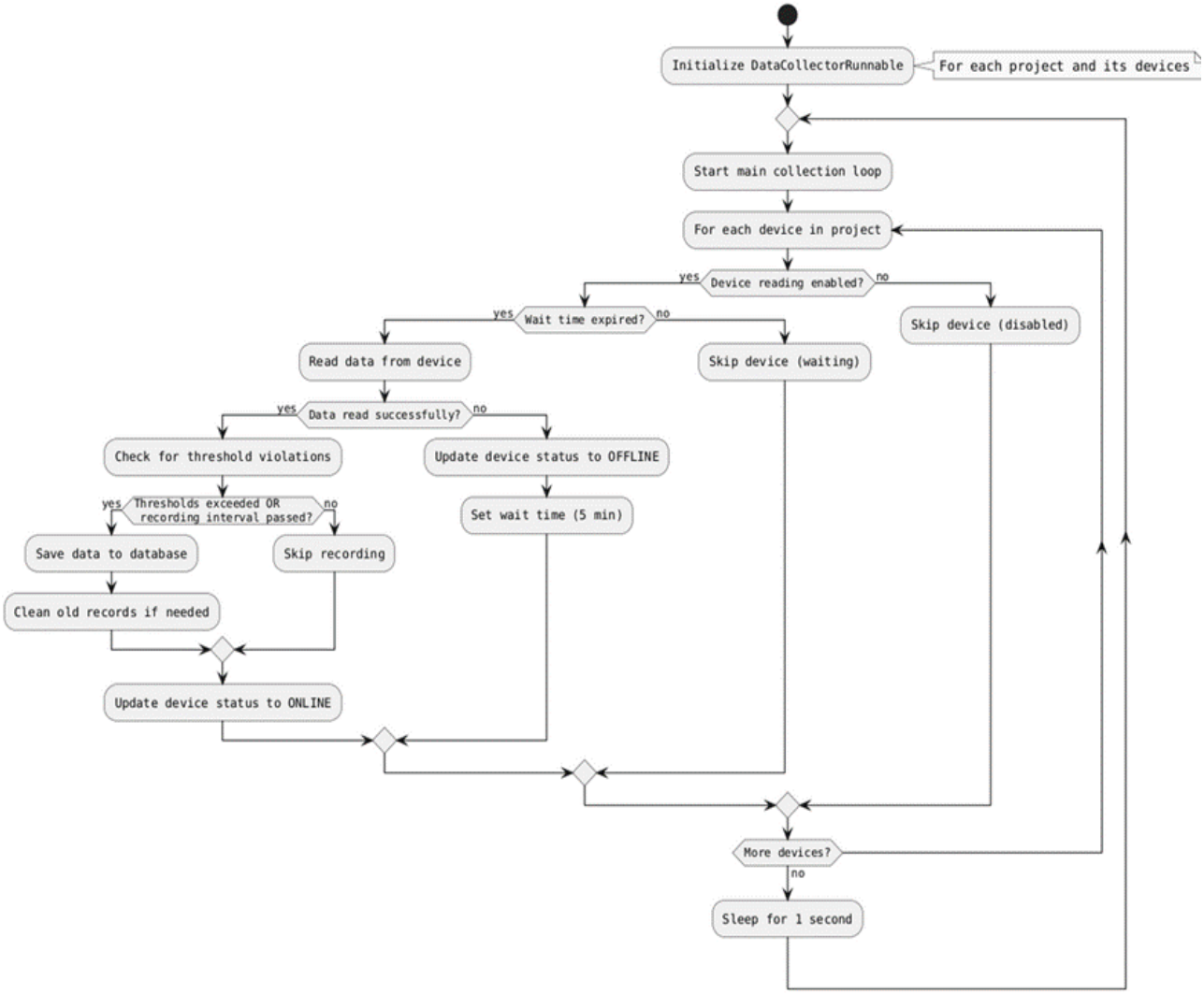



Рис. 4.3 Алгоритм роботи потоку збору даних

Для кожного проєкту ThreadManager створює окремий потік, який обробляє лише пристрої цього проєкту.

1. **Потік ініціалізується** для відповідного проєкту разом з усіма його пристроями.
2. **На початку кожного циклу** потоку з бази даних витягується актуальна інформація про проєкт та список пристроїв (один раз на цикл, а не для кожного пристрою окремо).
3. Далі запускається **головний цикл** опрацювання пристроїв проєкту:

- Для кожного пристрою перевіряється, чи увімкнено опитування (тобто чи дозволено зчитування даних).
  - Якщо зчитування вимкнено — пристрій пропускається.
- Якщо пристрій має статус **ОЧІКУВАННЯ** (waiting), перевіряється, чи завершився період очікування:
  - Якщо ні — пристрій пропускається до наступного циклу.
  - Якщо так — переходимо до зчитування.
- Виконується зчитування даних із пристрою:
  - Якщо зчитування не вдалося (функція повернула порожній словник), пристрою присвоюється статус **ОЧІКУВАННЯ**, і встановлюється час наступної спроби через 5 хвилин.
  - Якщо зчитування успішне — виконується перевірка параметрів на перевищення граничних значень (наприклад, напруга, струм, потужність тощо).
    - Якщо для пристрою активовано запис у базу даних — значення записуються, при потребі видаляються старі записи.
    - Якщо запис вимкнено — дані все одно зчитуються, але лише для перевірки перевищень, без збереження.
- Якщо пристрій раніше був у статусі **ОЧІКУВАННЯ**, і зараз зчитування вдалося — статус оновлюється до **ОНЛАЙН**.

4. Після опрацювання кожного пристрою відбувається пауза тривалістю 1 секунда перед переходом до наступного пристрою.
5. По завершенню масиву всіх пристроїв проєкту — цикл повторюється.

Цей алгоритм дозволяє системі надійно опрацьовувати велику кількість пристроїв, незалежно від наявності помилок в окремих пристроях. Усі винятки або невдачі обробляються без переривання циклу, що гарантує стабільну роботу потоку збору.

## 4.4 Проєктування бази даних

Для реалізації системи збору та збереження даних з енергетичних пристроїв було спроєктовано реляційну модель бази даних, яка дозволяє гнучко зберігати конфігурацію проєктів, параметри пристроїв, а також самі дані вимірювань у вигляді звітів. Структура побудована відповідно до принципів нормалізації, з акцентом на масштабованість та логічну узгодженість між таблицями, зображена на Рис. 4.4.

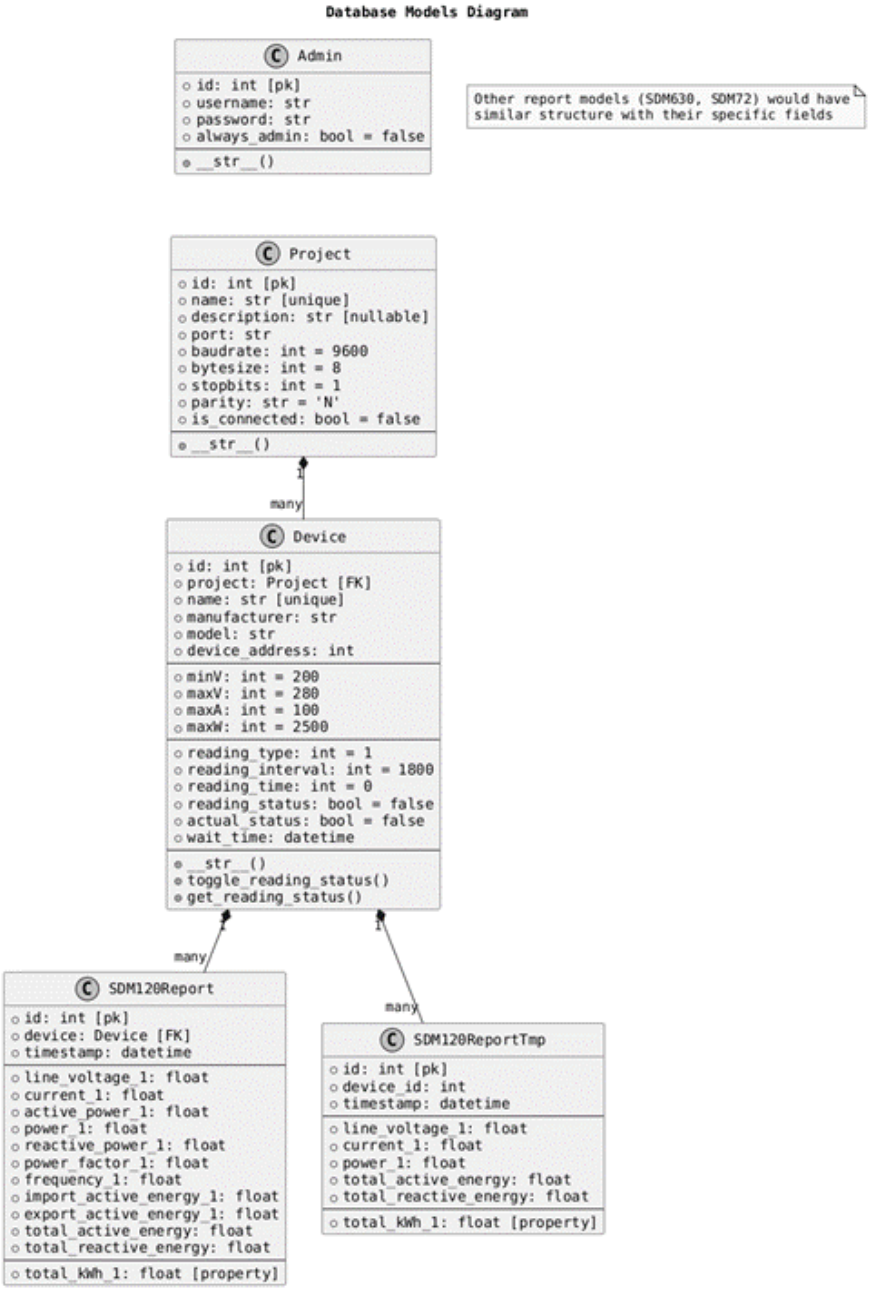



Рис. 4.4 Моделі бази даних

Нижче наведено опис основних моделей, які використовуються в системі:

**Admin**

Ця модель відповідає за зберігання адміністративних даних — логіна і пароля адміністратора системи. Усі критичні налаштування можуть змінюватися лише після автентифікації користувача з правами адміністратора.

- username (str): Ім’я користувача.
- password (str): Пароль користувача.
- Особливість: у системі зберігається лише один запис у таблиці Admin.

**Project**

Кожен проєкт у системі відповідає окремій фізичній лінії RS485, до якої підключені пристрої. Ця модель зберігає всі параметри зв'язку, необхідні для встановлення з'єднання з пристроями.

- name (str): Назва проєкту.
- port(str): Серійний порт в системі для доступу до проєкту
- baudrate (int): Швидкість обміну даними.
- parity (str): Тип парності (none, even, odd).
- bytesize (int): Розмір байта (часто 8).
- stopbits (float): Кількість стоп-бітів (1 або 2).

**Device**

Це основна модель для зберігання конфігурації пристроїв, що підключені до певного проєкту. Вона підтримує гнучке налаштування параметрів вимірювань і режимів зчитування.

- project (FK → Project): Посилання на проєкт.
- name (str): Назва пристрою (довільна).
- manufacturer (str): Виробник.
- model (str): Модель пристрою (наприклад, SDM120, SDM630).
- device_address (int): Адреса пристрою у мережі ModBus.
- minV (int): Мінімально допустима напруга.
- maxV (int): Максимально допустима напруга.
- maxA (int): Максимально допустимий струм.

- maxW (int): Максимальна потужність.
- reading_type (int): Тип зчитування (інтервальний 1 або щоденний 2).
- reading_interval (int, nullable): Інтервал (у секундах) для інтервального зчитування.
- reading_time (time, nullable): Час доби для щоденного зчитування.
- reading_status (bool): Статус, що вказує чи дозволене зчитування користувачем.
- actual_status (bool): Поточний статус пристрою (ONLINE, WAITING).
- wait_time (datetime, nullable): Час наступної спроби зчитування, якщо попередня була невдалою.
- Метод toggle_reading_status(): Змінює reading_status.
- Метод get_reading_status(): Повертає поточне значення reading_status.

**(*DeviceModel)Report**

**Маєтсья на увазі що це загальна структура для усіх таблиць цього типу, їх багато через те що для кожної моделі потрібна своя таблиця*

Ця модель зберігає історичні звіти з пристроїв, які зчитуються або за розкладом, або вручну. Дані зберігаються для подальшого аналізу та побудови графіків.

- device (FK → Device): Посилання на пристрій.
- timestamp (datetime): Час зчитування.
- voltage (float): Поточна напруга.
- current (float): Поточний струм.
- power (float): Потужність.
- energy (float): Сумарне споживання енергії.
- ... (інші параметри, залежно від моделі пристрою).

**(*DeviceModel)ReportTmp**

Модель ReportTmp використовується як буфер тимчасових даних для оновлення індикаторів в UI. На відміну від Report, ця таблиця зберігає лише один (останній) запис на кожен пристрій, що постійно оновлюється.

- device_id (FK → Device, unique): Посилання на пристрій.
- timestamp (datetime): Час останнього оновлення.
- voltage (float): Остання напруга.
- current (float): Останній струм.
- power (float): Остання потужність.
- ... (інші параметри, залежно від моделі пристрою).

Ця модель використовується в основному для візуального відображення останніх даних у таблицях, без зайвого навантаження на основну базу звітів.

**Загальні реляції між моделями:**

- Один Project → багато Device.
- Один Device → багато Report.
- Один Device → один ReportTmp.

**Примітка:**

- Для побудови моделей використовується Tortoise ORM.
- При запуску програми усі таблиці створюються автоматично, якщо якоїсь бракує – вона просто додається.
- Дані в Report зберігаються постійно, а ReportTmp — це «live-таблиця», яка оновлюється після кожного циклу зчитування.
- Статус actual_status допомагає відстежувати, які пристрої потребують уваги (наприклад, не відповідають).

# 5 РЕАЛІЗАЦІЯ ТА ТЕСТУВАННЯ

## 5.1 Дизайн інтерфейсу

Процес розробки графічного інтерфейсу користувача (GUI) здійснювався без попереднього проєктування або створення прототипів у сторонніх дизайнерських інструментах. Основною причиною такого підходу стало те, що на початкових етапах розробки не було чіткого бачення остаточного функціонального наповнення програми та зовнішнього вигляду інтерфейсу. У зв’язку з цим дизайн формувався поступово, "на ходу", паралельно з реалізацією бізнес-логіки програми.

Для побудови всіх елементів інтерфейсу не використовувалися файли формату .ui, створювані за допомогою Qt Designer. Уся візуальна логіка реалізована безпосередньо в Python-коді з використанням бібліотеки PySide6. Такий підхід на початкових етапах дозволив гнучко змінювати структуру форм, вікон і взаємодій, не прив’язуючись до фіксованої структури .ui-файлів, що особливо важливо при частих змінах логіки взаємодії з пристроями та базою даних.

Проте у перспективі передбачається перехід на використання .ui-файлів для пришвидшення розробки, уніфікації стилю вікон та зменшення обсягу коду. Це також дозволить легко передавати проєкт іншим розробникам або дизайнерам, що спростить супровід і підтримку проєкту. Крім того, редагування інтерфейсу через Qt Designer є більш наочним, ніж ручне створення вікон засобами коду.

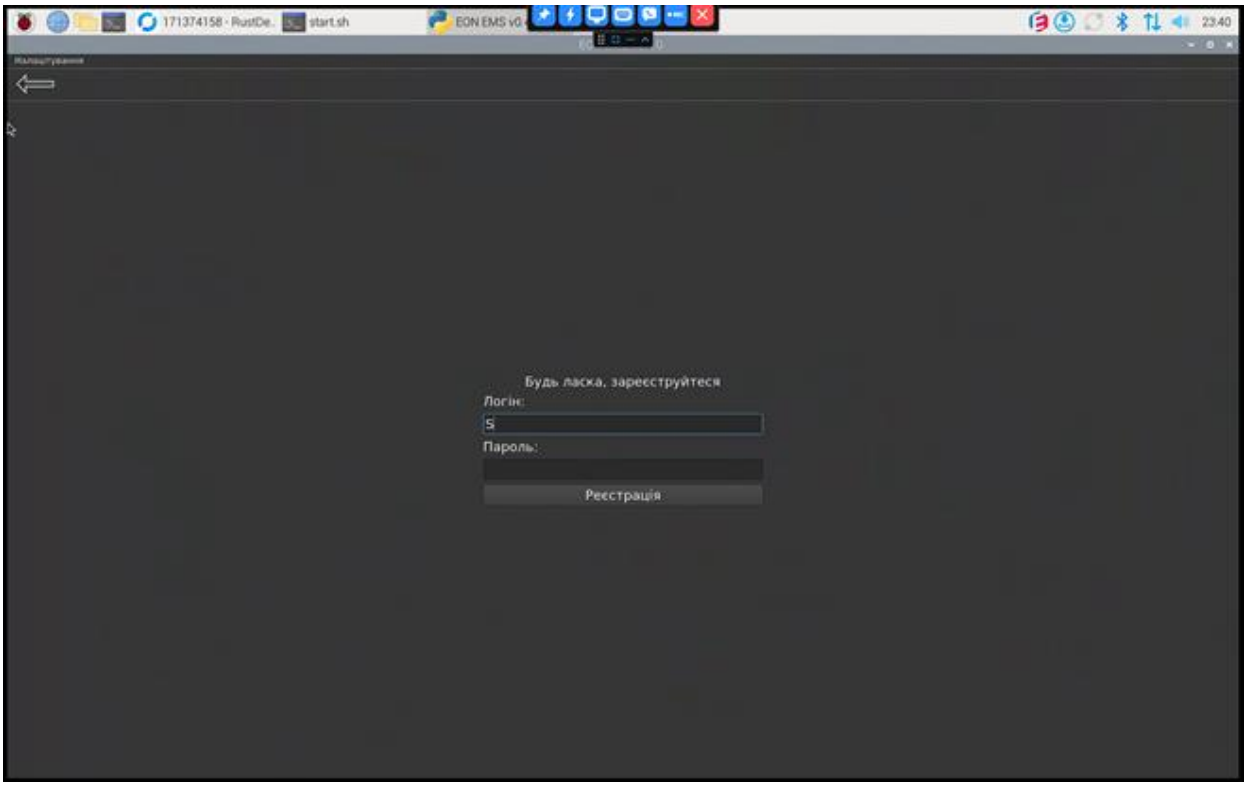



Рис. 5.1 Екран реєстрації адміністратора

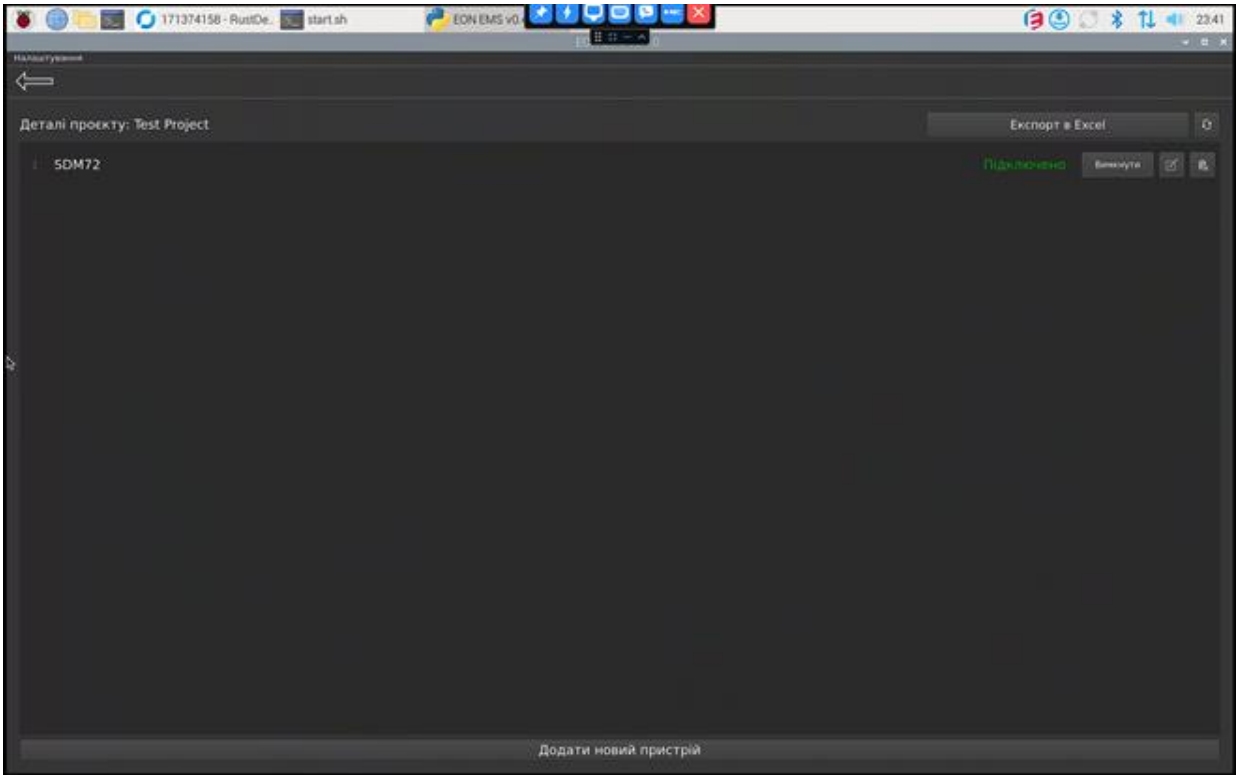



Рис. 5.2 Екран зі списком пристроїв одного проєкту

У якості стилю інтерфейсу було обрано **Fusion** – вбудований стиль Qt, який забезпечує однаковий вигляд елементів керування на різних операційних системах (Windows, Linux). Це важливо, оскільки додаток повинен коректно функціонувати та виглядати ідентично незалежно від платформи, на якій його буде запущено.

Було також прийнято рішення про використання темної теми з приглушено-сірими кольорами. Такий вибір дозволив зменшити навантаження на очі під час тривалої роботи з програмою та надати інтерфейсу сучасного вигляду. Основний колір акценту — синій, який виділяє активні елементи інтерфейсу (наприклад, натиснуті кнопки), створюючи контраст на фоні темної палітри.

На Рис. 5.1 та 5.2 можна побачити, що всі вікна інтерфейсу виконано у мінімалістичному стилі: великі кнопки з чіткими написами, зручна навігація між розділами, логічна структура меню. Основна мета — забезпечити доступність та зручність для найширшого кола користувачів: як молодих інженерів, які використовують додаток для аналізу споживання в режимі реального часу, так і людей без технічного досвіду (наприклад, консьєрж у будинку), який може просто переглянути інформацію про рівень споживання електроенергії в окремих під’їздах чи квартирах.

Інтерфейс повністю адаптивний і коректно масштабується під будь-який розмір екрана. Незалежно від того, чи додаток відкрито на маленькому екрані ноутбука, чи на великому дисплеї панелі оператора, усі елементи залишаються пропорційними, функціональними та читабельними.

### 5.2 Реалізація графічного модуля

Графічний модуль додатку розроблений за допомогою бібліотеки PySide6. Вся логіка взаємодії з користувачем реалізована через вікна, форми та діалогові вікна, що створюються динамічно.

Фактично усі переходи між вікнами це просто накладання поверх існуючих вікон нових або їхнє видалення, це допомогло зменшити кількість звернень до бази даних і пришвидшити роботу програми.

**Уривок коду класу MainWindow.py**

```
class MainWindow(QMainWindow):
    def __init__(self, thread_manager):
        super().__init__()
        self.initTrayIcon()
        self.is_exit = False

        self.thread_manager = thread_manager

        self.projects_widget = None
        self.project_view_widget = None
        self.device_details_widget = None

        self.isAdmin = False

        self.setWindowTitle("EON EMS v0.4.0")
        self.setGeometry(100, 100, 1200, 800)
        self.setMinimumWidth(800)
        self.setMinimumHeight(600)

        screen_geometry =
QtCore.QRect(QtGui.QGuiApplication.primaryScreen().geometry())
        x = (screen_geometry.width() - self.width()) // 2
        y = (screen_geometry.height() - self.height()) // 2
        self.move(x, y)

        self.menu_bar = self.menuBar()
```

```
back_icon = QIcon(resource_path("pyqt/icons/back.png"))
exitAct = QAction(back_icon, "Exit", self)
exitAct.setShortcut("Esc")
exitAct.triggered.connect(self.go_back)
self.toolbar = self.addToolBar("Exit")
self.toolbar.addAction(exitAct)
self.toolbar.setMovable(False)
self.toolbar.setIconSize(QtCore.QSize(60, 30))
…..
settings_action_timezone = QAction("Часовий пояс", self)
settings_action_timezone.triggered.connect(self.open_timezone_dialog)
settings_menu.addAction(settings_action_timezone)

self.central_widget = QWidget(self)
self.setCentralWidget(self.central_widget)

self.stacked_widget = QStackedWidget(self.central_widget)
self.central_layout = QVBoxLayout(self.central_widget)
self.central_layout.addWidget(self.stacked_widget)

self.registration_widget = RegistrationLoginForm(self)
self.stacked_widget.addWidget(self.registration_widget)

self.stacked_widget.setCurrentIndex(0)
self.showMaximized()
```

В продемонстрованому уривку коду можна наочно побачити як саме працює основна частина графічного модулю додатку:

1. Спершу відбуваються технічні процеси, ініціалізується іконка додатку в трей, присвоюється об’єкт менеджера потоків (для передачі і отримання сигналів від модулю збору даних).
2. Наступним кроком відбувається називання вікна та налаштування геометрії.
3. Після цього створюються кнопки в верхньому контекстному меню, різні налаштування і кнопка для повернення на попередній екран
4. Слідом же створюється StackedWidget – це тип віджету який і дозволяє мені додавати нові екрани поверх інших не видаляючи попередні і не перестворюючи їх заново, просто малюючи їх над попередніми. Також тут створюється і додається екран реєстрації авторизації.

**Методи для відкриття нових екранів**

```
def open_projects_list(self):
    self.projects_widget = ProjectsWidget(self)
    self.stacked_widget.addWidget(self.projects_widget)
    self.stacked_widget.setCurrentIndex(1)

def open_project_details(self, project):
    self.project_view_widget = ProjectViewWidget(self, project)
    self.stacked_widget.addWidget(self.project_view_widget)
    self.stacked_widget.setCurrentIndex(2)

def open_device_details(self, device):
    widget_class = {
        "SDM120": SDM120DeviceDetailsWidget,
        "SDM630": SDM630DeviceDetailsWidget,
        "SDM72": SDM72DeviceDetailsWidget,
    }.get(device.model, BaseDeviceDetailsWidget)

    self.device_details_widget = widget_class(self, device)
```

```
        self.stacked_widget.addWidget(self.device_details_widget)
        self.stacked_widget.setCurrentIndex(3)
```

Ці методи наочно демонструють як саме створюються і додаються нові вікна, без видалення попередніх або перезаписування поточного.

**Метод переходу на попереднє вікно**

```
    def go_back(self):
current_index = self.stacked_widget.currentIndex()
self.projects_widget.load_projects()

if current_index > 0:
    current_widget = self.stacked_widget.currentWidget()
    self.stacked_widget.removeWidget(current_widget)

    self.stacked_widget.setCurrentIndex(current_index - 1)
if current_index == 1:
    self.isAdmin = False
    self.stacked_widget.removeWidget(self.registration_widget)

    self.registration_widget = RegistrationLoginForm(self)
    self.stacked_widget.addWidget(self.registration_widget)
    self.stacked_widget.setCurrentIndex(0)
```

В цьому методі продемонстровано як реалізовано перехід на попереднє вікно, він перевіряє яке вікно буде показано якщо перейти на одне назад, якщо це буде останнє вікно, то метод видалить це вікно, поверне статус адміністратора до початкового (False) і перестворить це вікно заново, щоб видалити усі введені попередньо в ньому дані.

Також важливою частиною реалізації графічного модулю програми є використання базових принципів ООП [20], а саме принципу наслідування в реалізації найважливіших і найгроміздкіших екранів програми – екранів з детальною інформацією про пристрій (DeviceDetailsWidget).

Більшість лічильників мають відносно однаковий набір показників і параметрів які вони зберігають (Напруга, струм, потужність, спожита енергія), незалежно від виробника чи моделі. Проте не всі, деякі, як наприклад SDM72 не має показника спожитої енергії по кожній з трьох наявних у нього фаз, тому один уніфікований клас би призвів до величезної кількості if перевірок, щоб визначити що саме, в якому порядку і з якими назвами параметрів потрібно показати на екрані, які графіки чи який набір індикаторів.

Щоб вирішити цю проблему я використав принцип наслідування і реалізував клас **BaseDeviceDetailsWidget** в якому описав процеси створення UI в обтічній формі, уникаючи прив’язки до якоїсь моделі, проте беручи за основу найпопулярніший і найпростіший для реалізації пристрій (Eastron SDM120).

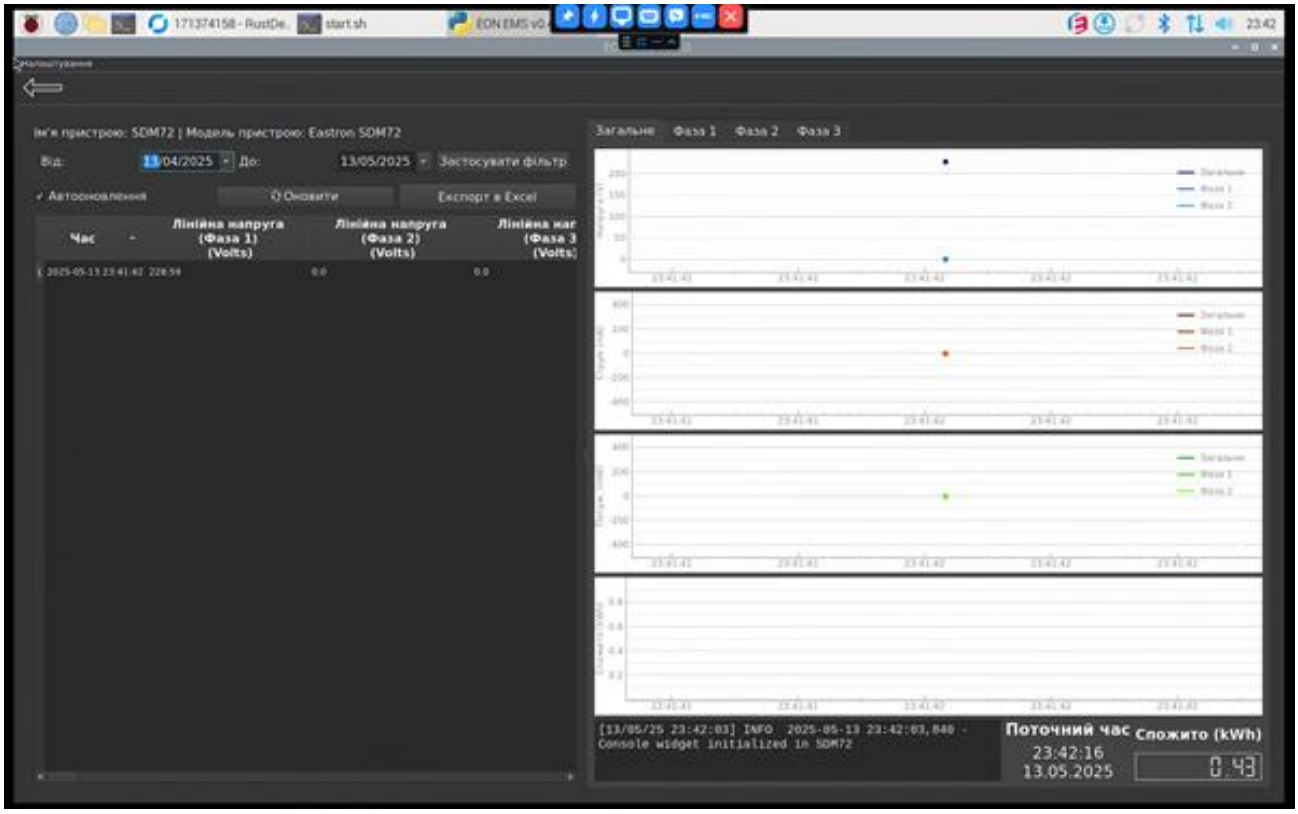



Рис. 5.3 Екран з детальною інформацією про пристрій

Більшість елементів, як фільтр по даті, керування оновленням даних на екрані, експорт в .xlsx файл, таблиця відображення бази даних, її сортування чи консоль під графіками, як на Рис. 5.3 – не залежать від моделі пристрою для якого створюється екран. Тому цей функціонал реалізований в базовому класі, його можна перевизначити якщо це необхідно. Також в класі є змінні з моделями бази даних через які можна отримати дані про пристрій і його звіти, ці змінні перевизначаються кожним дочірнім класом, що дозволило уніфікувати також процеси завантаження даних з бази даних.

А от графіки і індикатори, які напряму залежать від наявності чи відсутності певних параметрів у пристрою – є унікально перевизначеними для кожного пристрою.

Таким чином я досягнув того що знизилися складність та витрати по часу на додавання графічної підтримки нових пристроїв, для цього тепер достатньо тільки видозмінити під пристрій механізми створення графіків і індикаторів, або ж залишити їх такими як в базовому класі, якщо цей пристрій не має своїх особливостей відносно найпопулярнішого рішення. Також спростилося додавання підтримки пристроїв з унікальними можливостями, впроваджуючи їх я можу точно знати, що всі інші пристрої не «зламаються».

В усьому іншому – графічний модуль реалізовано як будь-який інший для будь-якого іншого ПЗ з використанням того чи іншого варіанту реалізації функціоналу Qt.

### 5.3 Реалізація модулю збору даних

Модуль збору даних реалізовано згідно запланованої архітектури модулю, яка була сформована на етапі проєктування.

Він реалізований за допомогою ThreadManager який створює та авторизує потоки для збору даних для кожного окремого проєкту. Самі потоки реалізовано в класі DataCollector, де виконується розроблений алгоритм роботи потоку, а

саме опитування лічильників відбувається в класі SerailReaderRS485, який створює RTU клієнт, групує регістри для зменшення кількості запитів і виконує ці запити, передаючи результат в DataCollector, який виконує операції збереження даних, безпосередньо, в базу даних.

На початкових етапах розробки система збору даних, зокрема класи SerialReaderRS485 та DataCollector, була реалізована без урахування основних принципів розробки програмного забезпечення, таких як SOLID, KISS чи DRY. Усе створювалося "на ходу", що призвело до занадто тісного зв’язку між модулями. Наприклад, у класі зчитування з RS485 безпосередньо створювалося вікно з помилкою, яке блокувало основний графічний інтерфейс - навіть якщо виклик відбувався в окремому потоці. Подібна реалізація постійно призводила до неочікуваних збоїв, складності масштабування та майже неможливості тестування окремих частин системи.

Після численних помилок і труднощів із супроводом коду було ухвалено рішення провести повну реорганізацію модуля збору даних. Було виділено час на рефакторинг, під час якого реалізацію переписано з урахуванням принципів розділення обов’язків (SRP) та інверсії залежностей. Уся логіка була поділена на малі, незалежні компоненти, кожна з яких відповідає лише за свою конкретну функцію: зчитування, обробку помилок, логування, передачу даних тощо. Всі залежності, які раніше "тягнули" графічний інтерфейс у глибини комунікації з пристроями, були усунені.

Нова архітектура дозволяє працювати модулю RS485 повністю незалежно - він не впливає на графічну частину програми, навіть у разі помилок. Замість прямих викликів інтерфейсу тепер застосовуються сигнали або абстрактні callback-інтерфейси, які передають інформацію про помилки на верхній рівень без жорсткого зв’язування. Завдяки такому підходу програма стала значно стабільнішою, а код — не лише структурованішим, але й коротшим і легшим для розуміння.

Очевидно, що застосування базових принципів розробки позитивно вплинуло не лише на якість коду, але й на зручність його розвитку в майбутньому.

### 5.4 Тестування застосунку з розгортанням

Процес тестування застосунку виявився доволі складним на початкових етапах, головним чином через те, що повноцінна перевірка роботи системи вимагала фізичного розгортання обладнання — тобто наявності лічильників, зібраної схеми підключення та відповідного середовища для передачі даних через RS485. У зв’язку з цим тестування "в польових умовах" могло проводитись лише на завершальних етапах, після налаштування всієї апаратної частини.

Для спрощення цього процесу було розроблено окремий модуль-імітатор, який замінював собою компонент SerialReaderRS485. Цей модуль генерував випадкові, але реалістичні дані, що дозволяло здійснювати повноцінне тестування графічного інтерфейсу, не маючи підключеного лічильника. Такий підхід дав змогу перевірити, як відображаються значення параметрів пристроїв, зокрема у вікні з деталями пристрою, а також оцінити поведінку інтерфейсу при зміні даних у реальному часі.

Окрім цього, всі функції були вручну протестовані на відповідність до вимог функціональної специфікації. Зокрема було перевірено:

- коректність зчитування та відображення даних;
- оновлення індикаторів та таблиць без перезавантаження інтерфейсу;
- стабільність циклічного збору даних у фоновому потоці;
- реакцію системи на відсутність зв’язку з пристроєм;
- зміну параметрів проєктів і пристроїв у реальному часі;
- адекватне логування та обробку помилок;
- автоматичне оновлення інтерфейсу після змін в базі даних.

Також були проведені стрес-тестування — зокрема симуляція великої кількості пристроїв, активного оновлення даних з частотою в кілька секунд, і паралельної роботи кількох вікон з таблицями.

Усі випробування підтвердили, що система здатна працювати стабільно та ефективно навіть у навантажених умовах, що дозволяє з упевненістю говорити про її готовність до реального розгортання та експлуатації.

### 5.5 Тестування застосунку в умовах різних архітектур

Окрім повноцінного тестування на платформі Windows, де відпрацьовувався основний функціонал, окрему увагу було приділено перевірці кросплатформеності застосунку. З самого початку однією з ключових цілей розробки була підтримка запуску програми не лише на звичних ПК, але й на більш компактних і бюджетних рішеннях, таких як одноплатні комп’ютери. Саме тому додаток був додатково протестований на Linux-системах, зокрема на Ubuntu Desktop та Raspberry Pi OS (на базі Debian), розгорнутих на платформі Raspberry Pi 4 B [21] — який і був обраний як базова апаратна частина для готового рішення, що буде постачатися кінцевим користувачам.

Під час тестування було виявлено низку особливостей, притаманних кожному з середовищ. Наприклад, на Windows виникали труднощі з підключенням через серійний інтерфейс RS485 через драйвери чипів CH340 — їх доводилося встановлювати вручну, а у деяких випадках — ще й конфігурувати. Натомість на Linux-системах, включаючи Raspberry OS, підтримка таких драйверів реалізована "з коробки", що суттєво полегшило процес розгортання. Однак, у Linux існує своя система доступу до портів — наприклад, потрібні права доступу до пристроїв /dev/ttyUSB*, що вимагало окремої адаптації модуля доступу до послідовного порту, враховуючи групи доступу користувача та можливість запуску програми з обмеженими правами.

Графічний модуль також зазнав необхідних змін — на Linux-системах Fusion-стиль, хоч і залишився сумісним, вимагав тонкого налаштування відображення шрифтів, розміру елементів та масштабування. Також довелося

внести зміни до системи обробки шляху до іконок і ресурсів, адже структура файлової системи в Linux значно відрізняється від Windows.

Попри всі ці нюанси, усі версії були успішно зібрані, розгорнуті та протестовані без критичних помилок. Це дало підстави стверджувати, що застосунок успішно виконав одну з основних вимог — повноцінна підтримка різних операційних систем і апаратних архітектур. Завдяки модульній структурі і відокремленню логіки від графіки, вдалося досягти гнучкості, яка дозволяє запускати систему як на повноцінному сервері, так і на мінімальному пристрої з ARM-архітектурою, без втрати стабільності чи функціоналу.

Таким чином, проєкт підтвердив свою готовність до практичного застосування у гетерогенному середовищі — від великих енергетичних об'єктів до невеликих житлових комплексів, де компактність і надійність мають вирішальне значення.

### 5.5.1 Raspberry Pi та його роль у реалізації проєкту

**Raspberry Pi** — це одноплатний комп’ютер, який став надзвичайно популярним завдяки своїй доступності, компактності та енергоефективності. В основі його популярності лежить баланс між низькою вартістю та достатньою продуктивністю для виконання багатьох прикладних задач. У рамках цього проєкту було використано модель Raspberry Pi 4 B, яка має чотириядерний ARM-процесор, до 8 ГБ оперативної пам’яті, а також підтримку USB та GPIO-інтерфейсів, що дозволяє легко підключати різноманітне периферійне обладнання, включаючи адаптери RS485.

Операційною системою було обрано **RaspberryOS** — офіційну ОС, яка базується на Debian Linux і оптимізована спеціально для Raspberry. Вона має всі необхідні компоненти для запуску Python-додатків, доступ до бібліотек для роботи з серійними портами, а також підтримку графічного інтерфейсу.

Raspberry Pi був обраний як платформа для розгортання клієнтської частини системи, оскільки він ідеально підходить для задач цілодобового моніторингу в реальному часі з мінімальними витратами електроенергії. Це дає змогу створити готове рішення для встановлення в житлових або промислових об'єктах без необхідності використовувати повноцінний ПК, що значно знижує вартість впровадження системи.

# ВИСНОВКИ

Результатом виконання дипломної роботи став кросплатформовий десктопний застосунок для збору, обробки та візуалізації даних з електролічильників за протоколом Modbus RTU через інтерфейс RS-485. Система дозволяє здійснювати централізований моніторинг енергоспоживання, зберігати історію показників, контролювати роботу пристроїв та формувати звіти. Завдяки гнучкій архітектурі та мінімалістичному інтерфейсу вона є універсальним рішенням як для промислового, так і для побутового застосування. У ході розробки були вирішені всі поставлені задачі:

• Проведено аналіз принципів роботи протоколу Modbus, досліджено методи зчитування та інтерпретації даних з електролічильників, розглянуто специфіку обміну даними у мережі RS-485.

• Оцінено існуючі рішення (в тому числі комерційні), визначено їх недоліки, серед яких – відсутність кросплатформеності, складність у розгортанні та низька адаптивність під нестандартні сценарії. На основі цього обґрунтовано потребу у створенні власного застосунку.

• Сформовано вимоги до функціоналу: підтримка різних типів пристроїв, адаптивний графічний інтерфейс, журналювання подій, формування звітів, підтримка декількох режимів зчитування даних, а також нефункціональні – стабільність, кросплатформеність, легкість в адаптації до нових середовищ.

• Спроєктовано архітектуру системи, що складається з незалежних модулів: графічного (на PyQt6), потокового збору даних (на основі asyncio), обробки подій, збереження даних та керування інтерфейсом. Проведено декомпозицію функціоналу відповідно до принципів KISS, DRY та SOLID.

• Реалізовано застосунок, в якому вся логіка оформлена у вигляді окремих класів і компонентів, що легко масштабуються. Графічний інтерфейс побудовано вручну, без використання .ui-файлів, із можливістю адаптації під будь-яке розширення екрана.

• Проведено комплексне тестування: як ручне функціональне, так і стрес-тести на довготривалу роботу. Для тестування в умовах відсутності фізичного

пристрою розроблено емулятор, який генерує правдоподібні випадкові дані. Усі ключові вимоги були виконані: правильне зчитування, стабільність GUI, відсутність блокувань, правильна індикація статусів.

• Проведено успішне розгортання системи на різних операційних системах: Windows, Ubuntu та Raspberry Pi OS. Враховано специфіку доступу до серійних портів на кожній платформі, а архітектура адаптована до ARM-процесорів, що підтверджує відповідність вимозі кросархітектурності.

• Розроблено інструкції для автоматизованого розгортання застосунку на пристроях Raspberry Pi, які можуть бути використані як основа для кінцевого вбудованого рішення в умовах реального середовища (наприклад, у щитових житлових будинків чи промислових об’єктах).

• Запропоновано напрямки подальшого розвитку системи: реалізація веб-інтерфейсу, збереження даних у хмарних базах, аналітичні модулі, розширення підтримуваних пристроїв, авторизація користувачів та міграція до .ui-файлів для зручності супроводу інтерфейсу.

Цілі поставлені на початку кваліфікаційної роботи досягнуто повністю. Побудована система задовольняє сучасні вимоги до надійності, гнучкості та масштабованості, має простий для користувача інтерфейс та готова до подальшого розгортання і розвитку.

Робота пройшла апробацію. За її результатами було опубліковано наступні тези доповідей:

1. Ємельянов В.Б., Замрій І.В. Проблема вибору кросплатформної та кросархітектурної бібліотеки для розробки віконного додатку на Python. // Матеріали VI Всеукраїнської науково-технічної конференції «Застосування програмного забезпечення в інформаційно-комунікаційних технологіях», 24.04.2025, ДУІКТ, Київ, Україна, с. 553-554.

2. Ємельянов В.Б., Замрій І.В. Порівняння RS485 та RJ45 в контексті протоколу передеачі даних modbus // Матеріали XII Всеукраїнської науково-практичної конференції молодих учених «Інформаційні технології – 2025». 15.05.25, КСУ імені Бориса Грінченка, Київ, Україна, с. 126-128.

# ПЕРЕЛІК ПОСИЛАНЬ

1. Patel D. N., Somani S. B. Implementation of Modbus RTU protocol on Cortex M3 Controller. International Journal of Electrical and Electronics Research. 2017. Vol. 9, no. 4. P. 678–684. URL: https://www.ripublication.com/ijeer17/ijeerv9n4_16.pdf (дата звернення: 11.03.2025).

2. Sulaiman S. A., Saad M. A. Development of a Data Acquisition and Monitoring System Based on MODBUS RTU Communication Protocol / ResearchGate. 2020. URL: https://www.researchgate.net/publication/342513211 (дата звернення: 11.03.2025).

3. Modbus Organization. Modbus over Serial Line – Specification and Implementation Guide Vol 1.02. / The Modbus Organization, 2006. – URL: https://modbus.org/docs/Modbus_over_serial_line_V1_02.pdf (дата звернення: 11.03.2025).

4. Ethernet vs RS485: Understanding the Key Differences. URL: https://www.pusr.com/blog/Ethernet-vs-RS485-Understanding-the-Key-Differences (дата звернення: 12.03.2025).

5. Modbus RTU vs. Modbus TCP/IP: A Comprehensive Guide TC. URL: https://www.alotceriot.com/modbus-rtu-vs-modbus-tcpip-comprehensive-guide/ (дата звернення: 12.03.2025).

6. Modbus Protocol Overview. URL: https://www.waveshare.com/wiki/Modbus_Protocol_Specification (дата звернення: 15.05.2025).

7. Офіційний сайт Schneider Electric EcoStruxure Power Monitoring Expert. URL: https://www.se.com/ua/uk/product-range/65404-ecostruxure-power-monitoring-expert/#overview (дата звернення: 11.04.2025).

8. Офіційний сайт SIEMENS. URL: https://www.siemens.com/global/en/products/energy/low-voltage/software/sentron-

powermanager.html (дата звернення: 11.04.2025).

9. Офіційний сайт ABB Ability™ Energy and Asset Manager. URL: https://global.abb/group/en/technology/did-you-know/abb-ability--energy-and-asset-manager (дата звернення: 11.04.2025).

10. Table of Contents. ISO 50001. 2019. С. 7. URL: https://doi.org/10.2307/j.ctvs32qbc.2 (дата звернення: 01.05.2025).

11. X86 vs. ARM64 / X. Chen та ін. Middleware '23: 24th International Middleware Conference, м. Bologna Italy. New York, NY, USA, 2023. URL: https://doi.org/10.1145/3631295.3631394 (дата звернення: 02.05.2025).

12. New Virtual Environment Based on Python Programming. Iraqi Journal for Computer Science and Mathematics. 2022. С. 111–118. URL: https://doi.org/10.52866/ijcsm.2022.02.01.012 (дата звернення: 02.05.2025).

13. Tkinter 8.5 reference: a GUI for Python. URL: https://docs.python.org/3/library/tkinter.html (дата звернення: 02.05.2025).

14. Differences Between PySide and PyQt. URL: https://wiki.qt.io/Differences_Between_PySide_and_PyQt (дата звернення: 08.05.2025).

15. Munson E. V. Unicode. Encyclopedia of Database Systems. New York, NY, 2017. С. 1. URL: https://doi.org/10.1007/978-1-4899-7993-3_5045-2 (дата звернення: 09.05.2025).

16. АНАЛІЗ СУЧАСНИХ СИСТЕМ УПРАВЛІННЯ БАЗАМИ ДАНИХ / В. Третяк та ін. InterConf. 2021. С. 453–465. URL: https://doi.org/10.51582/interconf.7-8.10.2021.050 (дата звернення: 09.05.2025).

17. Дьомін М. К., Полупан Ю. В. Аналіз методів представлення ієрархічної інформації в документо-орієнтованих базах даних. Вісник Східноукраїнського національного університету імені Володимира Даля. 2025. № 1 (287). С. 5–11. URL: https://doi.org/10.33216/1998-7927-2025-287-1-5-11 (дата звернення: 12.05.2025).

18. Ліпчевський А. Git Branching Strategies: Шлях до Ефективної Розробки та Успішного Бізнес-Плану, 2023. URL: https://surl.li/lavovc (дата звернення: 12.05.2025).

19. Tsitoara M. Commits. Beginning Git and GitHub. Berkeley, CA, 2024. С. 67–84. URL: https://doi.org/10.1007/979-8-8688-0215-7_5 (дата звернення: 16.05.2025).

20. Коваленко Ю. В. Візуальне об'єктно-орієнтоване середовище програмування. Шар візуалізації : thesis. 2021. URL: http://ir.stu.cn.ua/123456789/22782 (дата звернення: 16.05.2025).

21. Raspberry Pi Foundation. (2021). The Official Raspberry Pi Beginner's Guide (5th ed.). Raspberry Pi Press.

# ДОДАТОК А. ДЕМОНСТРАЦІЙНІ МАТЕРІАЛИ

ДЕРЖАВНИЙ УНІВЕРСИТЕТ ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ

НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

КАФЕДРА ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ


Розробка системи збору і аналізу даних з електролічильників з використанням технології ModBus (RS485)

Виконав студент 4 курсу
групи ПД-41
Ємельянов Владислав Богданович
Керівник роботи
д.т.н, проф., завідувач кафедри ІПЗ Замрій Ірина Вікторівна

Київ – 2025

## МЕТА, ОБ'ЄКТ ТА ПРЕДМЕТ ДОСЛІДЖЕННЯ

**Мета роботи –** спрощення збору та аналізу даних з аналізаторів мережі, покращення масштабованості відносно існуючих рішень.

**Об'єкт дослідження –** процеси збору та аналізу даних з електролічильників (аналізаторів мережі).

**Предмет дослідження –** система збору та аналізу даних з аналізаторів мережі за допомогою Modbus (RS485).

# ЗАДАЧІ ДИПЛОМНОЇ РОБОТИ

1. Провести аналіз способів збору та аналізу даних в електролічильників за допомогою протоколу ModBus.
2. Дослідити та проаналізувати існуючі аналоги.
3. Сформувати функціональні та нефункціональні вимоги до додатку.
4. Спроєктувати архітектуру модулів екранного відображення (pyQt) та потокового збору даних за допомогою asyncio.
5. Реалізувати десктопний додаток за визначеними вимогами.
6. Провести тестування основного функціоналу додатку.
7. Провести розгортання системи на різних операціних системах (Windows, Linux) та архітектурах (x86, x64, ARM64).
8. Реалізувати механізм автоматизації розгортання в специфічних серидовищах (RaspberryPi).
9. Запропонувати можливі покращення системи в перспективі подальшої розробки.

3

# АНАЛІЗ АНАЛОГІВ

| Характеристика | Schneider Electric EcoStruxure Power Monitoring Expert | Siemens Power Manager | Ability™ Energy and Asset Management (ABB) | EON EMS |
|---|---|---|---|---|
| Аналіз даних | Розширений аналіз якості електроенергії, тренди, звіти | Розширений аналіз енергоспоживання, порівняння, прогнозування | Аналіз ефективності, діагностика обладнання, оптимізація енергії | Аналіз енергоспоживання |
| Візуалізація даних | Інтуїтивно зрозумілі дашборди, графіки, однолінійні схеми | Налаштовувані дашборди, візуалізація в реальному часі | Кастомізовані панелі, інтерактивні діаграми, географічні карти | Лінійні, точкові та стовбчикові графіки (погодинного енергоспоживання), максимальна простота і мінімалізм |
| Масштабованість | Зазвичай висока, підтримує велику кількість пристроїв та їхніх типів, компанії Schneider | Зазвичай висока, модульна архітектура, проте тільки окремими модулями від компанії | Зазвичай висока, хмарні рішення, потребує складне специфіковане обладнання | Висока, залежить від наявного устаткування, не потребує специфікованого обладнання. |
| Складність впровадження | Може бути складною, потребує спеціальних знань з обладнання Schneider Electric, інтеграція тільки з продуктами компанії | Може бути складною, інтеграція тільки з екосистемою Siemens | Може бути складною, особливо хмарних рішень | Відносно легка, потрібна тільки комутація і локальний комп'ютер |
| Особливості | Глибока інтеграція з обладнанням Schneider Electric, розширені функції якості електроенергії | Сильна інтеграція з промисловими системами Siemens, фокус на енергоефективності | Можливість хмарного розгортання, велика кількість підтримуваних типів пристроїв, тільки пристрої компанії ABB | Підтримка абсолютно усіх аналізаторів мережі у яких наявний RS485 шлюз, максимальна інтуїтивність та одноразове і просте налаштування |

4

# ВИМОГИ ДО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

## Функціональні вимоги:

– Можливість зчитування даних з електролічильників за протоколом ModBus (через інтерфейс RS485).
– Підтримка додавання, редагування та видалення пристроїв без перезапуску програми.
– Збереження даних у локальній базі даних з історією параметрів.
– Формування звітів у форматі .xlsx за обрані часові періоди.
– Наявність графіків, таблиць та індикаторів для візуального аналізу даних.
– Механізм реєстрації та авторизації користувача (адміністратор або гість).
– Адміністратор має повний доступ, гість — лише перегляд даних.
– Налаштування періодичності запису даних до звітів (від 2 до 59 хвилин).

## Нефункціональні вимоги:

– Система має бути простою у використанні — встановлення, налаштування та експлуатація без технічних знань.
– Графічний інтерфейс працює окремо від модуля збору даних (архітектура з розділенням логіки GUI та фонових процесів).
– У разі закриття графічного вікна програма згортається в системний трей, не припиняючи збір даних.
– Частота збору даних залежить від якості лінії зв'язку та конфігурації.
– Застосунок є десктопним, автономним, не потребує доступу до інтернету.
– Усі дані зберігаються локально (SQLite).
– Підтримка запуску як на Windows, так і на Linux (включно з ARM64 пристроями, наприклад Raspberry Pi).
– Можливість розширення підтримки нових моделей пристроїв без суттєвої модифікації коду.

5

# ПРОГРАМНІ ЗАСОБИ РЕАЛІЗАЦІЇ


Python 3 & pyQT

SQLite


XLSXwriter

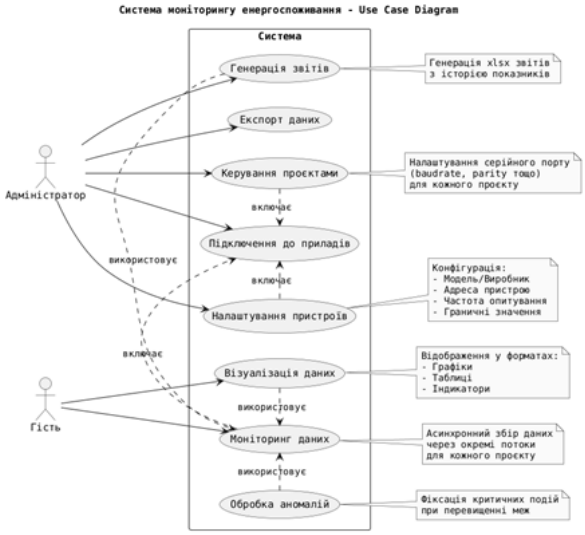

PyCharm Professional

RustDesk



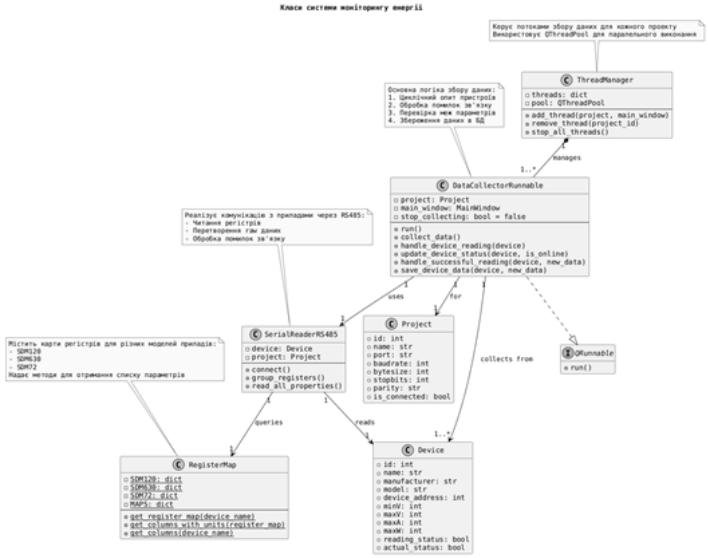
6

# ДІАГРАМА ВАРІАНТІВ ВИКОРИСТАННЯ



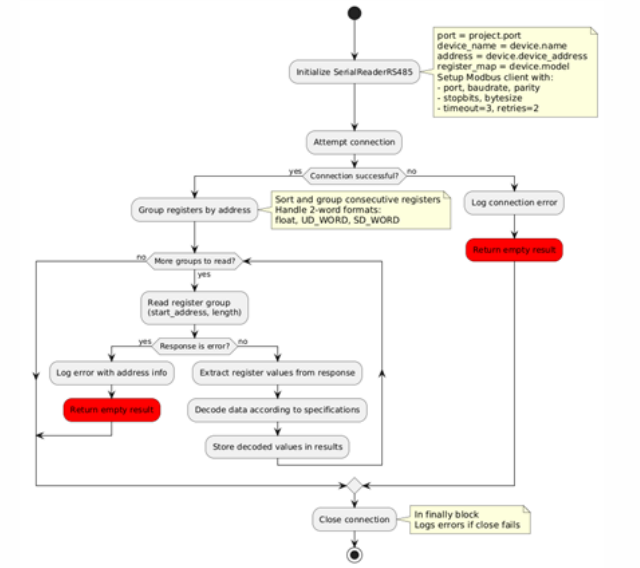

7

# ДІАГРАМА КЛАСІВ МОДУЛЮ ПОТОКОВОГО ЗБОРУ ДАНИХ



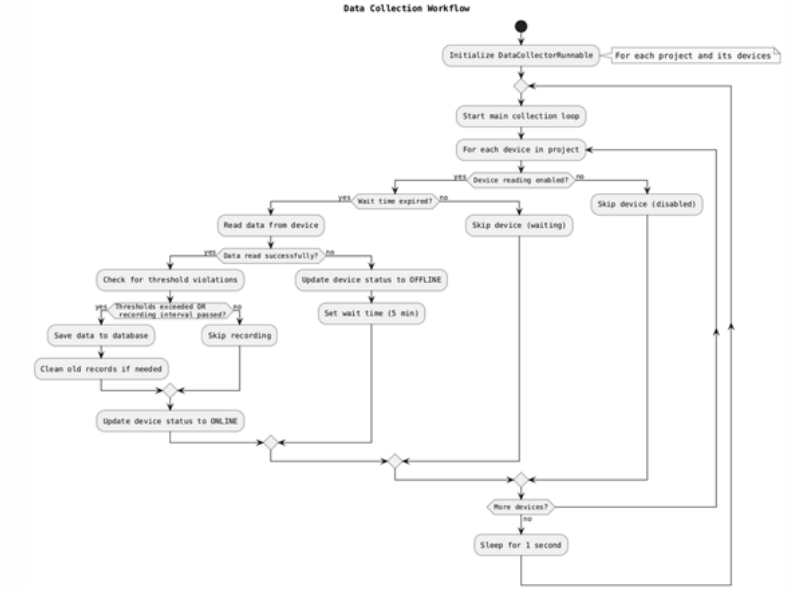

8

# ДІАГРАМА АЛГОРИТМУ ЗБОРУ ДАНИХ



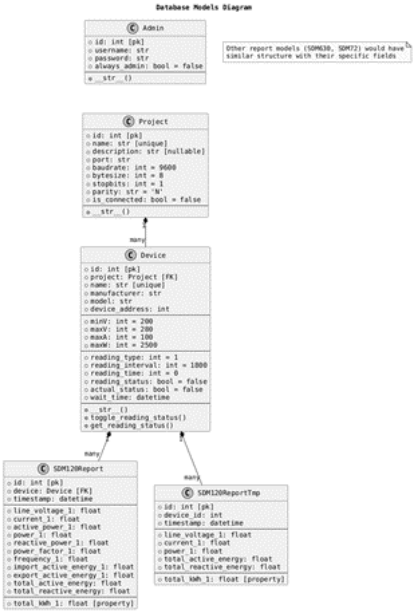

9

# ДІАГРАМА АЛГОРИТМУ РОБОТИ ПОТОКУ ЗБОРУ ДАНИХ



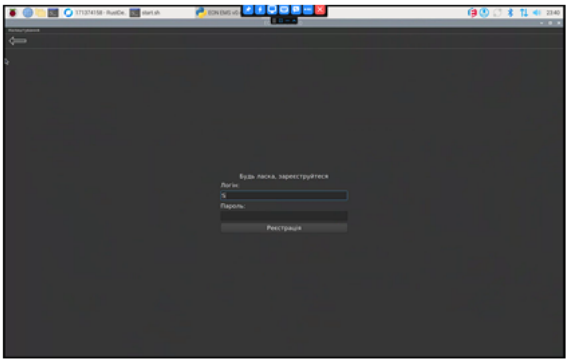

10

# ДІАГРАМА МОДЕЛЕЙ БАЗИ ДАНИХ

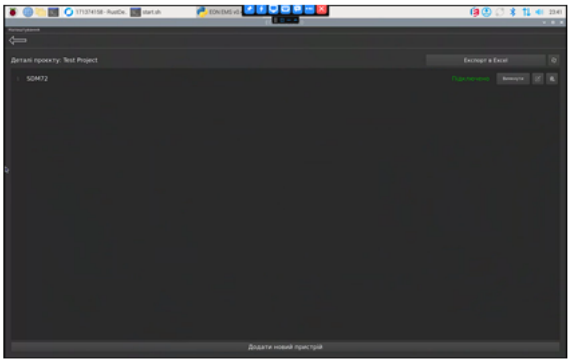



11

# ЕКРАННІ ФОРМИ



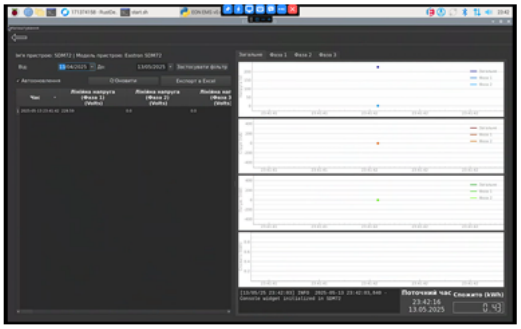

Екран реєстрації адміністратора



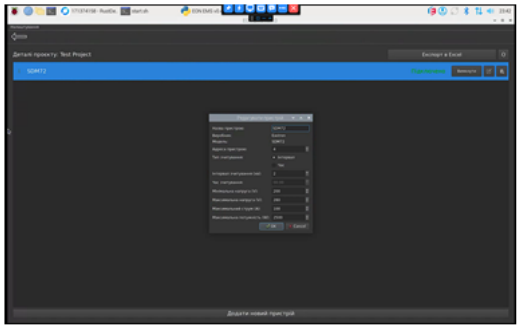

Список пристроїв проєкту

12

# ЕКРАННІ ФОРМИ



Екран з деталями на аналізом по девайсу



Вікно налаштувань пристрою



# АПРОБАЦІЯ РЕЗУЛЬТАТІВ ДОСЛІДЖЕННЯ

1. Ємельянов В.Б., Замрій І.В. Проблема вибору кросплатформної та кросархітектурної бібліотеки для розробки віконного додатку на Python. // Матеріали VI Всеукраїнської науково-технічної конференції «Застосування програмного забезпечення в інформаційно-комунікаційних технологіях», 24.04.2025, ДУІКТ, Київ, Україна, с. 553-554.
2. Ємельянов В.Б., Замрій І.В. Порівняння RS485 та RJ45 в контексті протоколу передеачі даних modbus // Матеріали XII Всеукраїнської науково-практичної конференції молодих учених «Інформаційні технології – 2025». 15.05.25, КСУ імені Бориса Грінченка, Київ, Україна, с. 126-128.

# ВИСНОВКИ

1. Проведено аналіз способів зчитування та обробки даних з електролічильників, що працюють за протоколом Modbus RTU через інтерфейс RS-485.
2. Розглянуто сучасні програмні рішення для збору енергетичних даних, виявлено їх ключові недоліки та враховано їх при побудові власної системи.
3. Визначено повний перелік функціональних та нефункціональних вимог, які стали основою для розробки гнучкого і масштабованого застосунку.
4. Спроєктовано модульну архітектуру системи, що дозволяє розділити інтерфейсну частину та логіку збору даних, з використанням бібліотек PyQt та asyncio.
5. Розроблено кросплатформовий десктопний застосунок, який забезпечує стабільну роботу згідно з визначеними вимогами.
6. Здійснено всебічне тестування функціональності, включаючи поведінку в умовах навантаження та збоїв у комунікації.
7. Проведено успішне розгортання системи на основних ОС (Windows, Linux) та апаратних архітектурах, включаючи ARM64.
8. Реалізовано сценарії автоматизованого розгортання та конфігурації для вбудованих систем на базі Raspberry Pi.
9. На основі отриманих результатів запропоновано подальші шляхи вдосконалення системи: інтеграцію хмарних сервісів, веб-інтерфейс та модулі аналітики.

Цілі поставлені на початку кваліфікаційної роботи досягнуто повністю. Побудована система задовольняє сучасні вимоги до надійності, гнучкості та масштабованості, має простий для користувача інтерфейс та готова до подальшого розширення і розвитку.

# ДОДАТОК Б. ЛІСТИНГИ ПРОГРАМНИХ МОДУЛІВ

```
import logging
logger = logging.getLogger(__name__)

import asyncio
import os
import sys

from AsyncioPySide6 import AsyncioPySide6
from PySide6.QtGui import QIcon
from PySide6.QtWidgets import (
    QApplication,
)
from tortoise import Tortoise

from tools import config
from pyqt.MainWindow import MainWindow
from tools.ThreadManager import
stop_threads_synchronously, ThreadManager,
initialize_threads
from tools.start_tools import is_already_running,
show_warning_message, get_database_path, init_database, \
    get_darkModePalette



def on_about_to_quit(loop, thread_manager):
    stop_threads_synchronously(thread_manager)
    async def shutdown():
        try:
            logger.info("Closing database connections...")
            await Tortoise.close_connections()

            logger.info("Cancelling all asyncio tasks...")
            tasks = [task for task in asyncio.all_tasks() if task is
not asyncio.current_task()]
            await asyncio.gather(*tasks, return_exceptions=True)
            logger.info("All cleanup completed.")
        except Exception as e:
            logger.error(f"Error during shutdown: {e}")

    if loop and loop.is_running():
        loop.call_soon_threadsafe(lambda:
asyncio.run(shutdown()))
    else:
        asyncio.run(shutdown())


if __name__ == "__main__":

    if is_already_running():
        show_warning_message()
        sys.exit(1)

    config.init_config()
    config.init_logger()

    logger.info('Starting EON EMS')
    db_path = os.path.join(get_database_path())
    asyncio.run(init_database(db_path))

    app = QApplication(sys.argv)

app.setWindowIcon(QIcon(config.resource_path("pyqt/icons
/app-icon.ico")))
    app.setStyle("Fusion")
    app.setPalette(get_darkModePalette(app))
    thread_manager = ThreadManager()

    with AsyncioPySide6.use_asyncio() as loop:
        main_window = MainWindow(thread_manager)
        main_window.show()
```

```
AsyncioPySide6.runTask(initialize_threads(main_window, thread_manager))

    app.aboutToQuit.connect(lambda: on_about_to_quit(loop, thread_manager))

    try:
        sys.exit(app.exec())
    except Exception as e:
        logger.error(f"Error during shutdown: {e}")
```

```
import logging
logger = logging.getLogger(__name__)

from PySide6.QtCore import QThreadPool
from rtu.DataCollector import DataCollectorRunnable


class ThreadManager:
    def __init__(self):
        self.threads = {}
        self.pool = QThreadPool()

    def add_thread(self, project, main_window):
        if project.id in self.threads:
            logger.warning(f"Thread for project {project.id} already exists.")
            return

        task = DataCollectorRunnable(project, main_window)
        self.pool.start(task)
        self.threads[project.id] = task

    def remove_thread(self, project_id):
        task = self.threads.get(project_id)
        if task:
            task.stop_collecting = True
            del self.threads[project_id]
            logger.info(f"Task for project {project_id} stopped and removed.")
        else:
            logger.warning(f"No task found for project {project_id}.")

    def stop_all_threads(self):
        for project_id in list(self.threads.keys()):
            self.remove_thread(project_id)
        logger.info("All tasks stopped.")
        self.pool.waitForDone()

async def initialize_threads(main_window, thread_manager):
    from models.Project import Project

    projects = await Project.all()
    for project in projects:
        thread_manager.add_thread(project, main_window)

def stop_threads_synchronously(thread_manager):
    logger.info("Stopping all threads...")
    thread_manager.stop_all_threads()
    logger.info("All threads stopped.")
```

```
import asyncio
import logging
from datetime import datetime, timedelta

import pytz

from models.Project import Project

logger = logging.getLogger(__name__)

from AsyncioPySide6 import AsyncioPySide6
from PySide6.QtCore import QRunnable
from PySide6.QtWidgets import QMessageBox

from tools.config import get_deleting_time, get_timezone
from models.Device import Device
from models.Report import SDM120Report, SDM120ReportTmp, SDM630Report, SDM72Report, SDM630ReportTmp, SDM72ReportTmp
from rtu.SerialReaderRS485 import SerialReaderRS485
```

```
async def get_data_from_device(device, project,
main_window):
    try:
        client = SerialReaderRS485(device, project)

        if
main_window.thread_manager.threads.get(project.id).stop_c
ollecting:
            return {}

        return await client.read_all_properties()

    except asyncio.CancelledError:
        logger.warning(f"{device.name} - Task cancelled")
        return {}
    except Exception as e:
        logger.warning(f"{device.name} - Task failed: {e}")
        return {}

def is_voltage_out_of_range(new_data, device, phase):
    voltage_key = f"line_voltage_{phase}"
    voltage_value = new_data.get(voltage_key)

    if voltage_key in new_data:
        if voltage_value > device.maxV:
            logger.warning(f"Voltage of {phase} phase in
{device.name} ({device.model}) is above
({device.maxV}V). \n" +
                f"Current value: {voltage_value}V. Extra:
{voltage_value - device.maxV}V.")
            return True
        elif voltage_value < device.minV:
            if voltage_value == 0:
                return False
            print(f"Voltage of {phase} phase in {device.name}
({device.model}) is less than ({device.minV}V). \n" +
                f"Current value: {voltage_value}V. Lack:
{device.minV - voltage_value}V.")
            return True
    return False

def is_current_over_limit(new_data, device, phase):
    current_key = f"current_{phase}"
    current_value = new_data.get(current_key)

    if current_key in new_data:
        if current_value > device.maxA:
            logger.warning(f"Current of {phase} phase in
{device.name} ({device.model}) is above
({device.maxA}A). \n" +
                f"Current value: {current_value}A. Extra:
{current_value - device.maxA}A.")
            return True
    return False

def is_power_over_limit(new_data, device, phase):
    power_key = f"power_{phase}"
    power_value = new_data.get(power_key)

    if power_key in new_data:
        if power_value > device.maxW:
            logger.warning(f"Power of {phase} phase in
{device.name} ({device.model}) is above
({device.maxW}W). \n" +
                f"Current value: {power_value}W. Extra:
{power_value - device.maxW}W.")
            return True
    return False


class DataCollectorRunnable(QRunnable):
    def __init__(self, project, main_window):
        super().__init__()
        self.project = project
        self.main_window = main_window
        self.stop_collecting = False

    def run(self):
        AsyncioPySide6.runTask(self.collect_data())

    async def collect_data(self):
        while not self.stop_collecting:
            try:
```

```
            self.project = await
Project.filter(id=self.project.id).first()
            devices = await
Device.filter(project=self.project).all()
            for device in devices:
                await self.handle_device_reading(device)
        except Exception as e:
            logger.error(f"Error in main collection loop: {e}")

        if not self.stop_collecting:
            await asyncio.sleep(1)

    async def handle_device_reading(self, device):
        try:
            local_tz = get_timezone()
            now_local = datetime.now(local_tz)
            if device.reading_status is False:
                return
            if device.wait_time and device.wait_time >
now_local:
                return

            new_data = await get_data_from_device(device,
self.project, self.main_window)

            if not new_data or new_data == {}:
                await self.handle_read_error(device)
                return

            await self.handle_successful_reading(device,
new_data)

        except Exception as e:
            logger.error(f"Unexpected error handling device
{device.name}: {e}")
            await self.update_device_status(device, False)

    async def update_device_status(self, device, is_online):
        try:
            if device.actual_status != is_online:
                device.actual_status = is_online
                if not is_online:
                    tz = get_timezone()
                    now_utc =
datetime.utcnow().replace(tzinfo=pytz.utc)
                    wait_time_local = now_utc +
timedelta(seconds=300)
                    device.wait_time =
wait_time_local.astimezone(tz)

                await device.save(update_fields=['actual_status',
'wait_time'])

                if self.main_window.project_view_widget is not
None:

self.main_window.project_view_widget.load_devices()

                status_msg = "online" if is_online else "offline"
                logger.info(f"Device {device.name} -
{device.model} is now {status_msg}")
        except Exception as e:
            logger.error(f"Error updating device {device.name}
status: {e}")

    async def handle_read_error(self, device):
        await self.update_device_status(device, False)

    async def handle_successful_reading(self, device,
new_data):
        if not device.actual_status:
            await self.update_device_status(device, True)

        phases = self.get_phases(device)
        immediate_record = any(
            is_voltage_out_of_range(new_data, device, phase) or
            is_current_over_limit(new_data, device, phase) or
            is_power_over_limit(new_data, device, phase)
            for phase in phases
        )

        if not immediate_record:
            last_report = await self.get_last_report(device)
            if last_report and not self.should_record_now(device,
last_report):
```

```
            return

        await self.save_device_data(device, new_data)

    async def get_last_report(self, device):
        main_db_model, _ = self.get_db_model(device)
        return await main_db_model.filter(device=device).last()

    def should_record_now(self, device, last_report):
        device_reading_interval = device.reading_interval
        last_report_time = last_report.timestamp.replace(tzinfo=None)
        calculated_time = last_report_time + timedelta(seconds=device_reading_interval)
        current_time = datetime.now().replace(tzinfo=None)

        if device.reading_type == 2:
            reading_time = device.reading_time
            start_of_day = datetime.now().replace(hour=0, minute=0, second=0, microsecond=0)
            return current_time >= (start_of_day + timedelta(minutes=reading_time)) and last_report_time < start_of_day

        return current_time >= calculated_time

    async def save_device_data(self, device, new_data):
        try:
            main_db_model, tmp_db_model = self.get_db_model(device)

            tmp_report_data = self.get_tmp_data(device, new_data)
            existing_tmp_report = await tmp_db_model.filter(device_id=device.id).first()

            if existing_tmp_report:
                for key, value in tmp_report_data.items():
                    setattr(existing_tmp_report, key, value)
                await existing_tmp_report.save()
            else:
                tmp_report = tmp_db_model(**tmp_report_data)
                await tmp_report.save()

            report_data = {"device_id": device.id}
            report_data.update({key: value for key, value in new_data.items() if value is not None})

            new_report = main_db_model(**report_data)
            await new_report.save()

            logger.info(f"Report saved - {device.name}, {device.model}")

            await self.clean_old_records(device, main_db_model)

        except Exception as e:
            logger.error(f"Error saving data for device {device.name}: {e}")

    async def clean_old_records(self, device, db_model):
        deleting_time = get_deleting_time()
        if deleting_time > 0:
            delete_before_date = datetime.now().replace(tzinfo=None) - timedelta(days=deleting_time)
            first_report = await db_model.filter(device=device).first()
            if first_report and first_report.timestamp.replace(tzinfo=None) < delete_before_date:
                await first_report.delete()

    def get_tmp_data(self, device, new_data):
        if device.model == "SDM120":
            tmp_report_data = {
                "device_id": device.id,
                "line_voltage_1": new_data.get("line_voltage_1"),
                "current_1": new_data.get("current_1"),
                "power_1": new_data.get("power_1"),
                "total_active_energy": new_data.get("total_active_energy"),
                "total_reactive_energy": new_data.get("total_reactive_energy")
            }
            return tmp_report_data
        elif device.model == "SDM630":
```

```
            tmp_report_data = {
                "device_id": device.id,
                "line_voltage_1": new_data.get("line_voltage_1"),
                "line_voltage_2": new_data.get("line_voltage_2"),
                "line_voltage_3": new_data.get("line_voltage_3"),
                "current_1": new_data.get("current_1"),
                "current_2": new_data.get("current_2"),
                "current_3": new_data.get("current_3"),
                "power_1": new_data.get("power_1"),
                "power_2": new_data.get("power_2"),
                "power_3": new_data.get("power_3"),
                "total_kWh_1": new_data.get("total_kWh_1"),
                "total_kWh_2": new_data.get("total_kWh_2"),
                "total_kWh_3": new_data.get("total_kWh_3"),
                "total_kWh": new_data.get("total_kWh"),
            }
            return tmp_report_data
        elif device.model == "SDM72":
            tmp_report_data = {
                "device_id": device.id,
                "line_voltage_1": new_data.get("line_voltage_1"),
                "line_voltage_2": new_data.get("line_voltage_2"),
                "line_voltage_3": new_data.get("line_voltage_3"),
                "current_1": new_data.get("current_1"),
                "current_2": new_data.get("current_2"),
                "current_3": new_data.get("current_3"),
                "power_1": new_data.get("power_1"),
                "power_2": new_data.get("power_2"),
                "power_3": new_data.get("power_3"),
                "total_kWh": new_data.get("total_kWh"),
            }
            return tmp_report_data
        else:
            QMessageBox.warning(
                self.main_window, f"{device.name}",
f"{device.model} - Невідома модель",
QMessageBox.StandardButton.Ok,
                QMessageBox.StandardButton.Cancel)


    def get_db_model(self, device):
        if device.model == "SDM120":
            self.phases = ['1']
            return SDM120Report, SDM120ReportTmp
        elif device.model == "SDM630":
            self.phases = ['1', '2', '3']
            return SDM630Report, SDM630ReportTmp
        elif device.model == "SDM72":
            self.phases = ['1', '2', '3']
            return SDM72Report, SDM72ReportTmp
        else:
            logger.error("Unknown device model")


    def get_phases(self, device):
        if device.model == "SDM120":
            return ['1']
        elif device.model == "SDM630":
            return ['1', '2', '3']
        elif device.model == "SDM72":
            return ['1', '2', '3']
        else:
            logger.error("Unknown device model")


import logging

logger = logging.getLogger(__name__)

from pymodbus.client import ModbusSerialClient
from pymodbus.constants import Endian
from pymodbus.payload import BinaryPayloadDecoder

from register_maps.RegisterMaps import RegisterMap


def decode_data(data, property_specifications):
    decoded_data = 0

    if property_specifications["format"] == "float":
        decoded_data = decode_32bit_float(data)
    elif property_specifications["format"] == "U_WORD":
```

```
        decoded_data = data[0]
    elif property_specifications["format"] == "UD_WORD":
        decoded_data = (data[0] << 16) + data[1]
    elif property_specifications["format"] == "S_WORD":
        decoded_data = decode_16bit_signed(data)
    elif property_specifications["format"] == "SD_WORD":
        decoded_data = decode_32bit_signed(data)

    if "divider" in property_specifications:
        decoded_data /= property_specifications["divider"]

    return round(decoded_data, 2)



def decode_16bit_signed(data):
    return BinaryPayloadDecoder.fromRegisters(data,
byteorder=Endian.BIG,
wordorder=Endian.BIG).decode_16bit_int()



def decode_32bit_signed(data):
    return BinaryPayloadDecoder.fromRegisters(data,
byteorder=Endian.BIG,
wordorder=Endian.BIG).decode_32bit_int()



def decode_32bit_float(data):
    decoded_data =
BinaryPayloadDecoder.fromRegisters(data,
byteorder=Endian.BIG,
wordorder=Endian.BIG).decode_32bit_float()
    return decoded_data



class SerialReaderRS485:
    def __init__(self, device, project):
        self.port = project.port
        self.device_custom_name = device.name
        self.device_address = device.device_address
        self.register_map =
RegisterMap.get_register_map(device.model)

        self.client = ModbusSerialClient(
            port=f"COM{project.port}",
baudrate=project.baudrate, parity=project.parity,
            stopbits=project.stopbits, bytesize=project.bytesize,
timeout=3, retries=2
        )

    def connect(self):
        try:
            return self.client.connect()
        except Exception as e:
            logger.error(f"{self.device_custom_name} -
Connection error on port {self.port}: {str(e)}")
            return False

    def group_registers(self):
        grouped = []
        sorted_registers = sorted(self.register_map.items(),
key=lambda x: x[1]['register'])
        current_group = {'start': None, 'length': 0, 'items': []}

        for name, spec in sorted_registers:
            start = spec['register']
            length = 2 if spec['format'] in ["float", "UD_WORD",
"SD_WORD"] else 1

            if current_group['start'] is None:
                current_group['start'] = start
                current_group['length'] = length
                current_group['items'].append((name, length))
            elif start == current_group['start'] +
current_group['length']:
                current_group['length'] += length
                current_group['items'].append((name, length))
            else:
                grouped.append(current_group)
                current_group = {'start': start, 'length': length,
'items': [(name, length)]}

        if current_group['items']:
```

```
            grouped.append(current_group)

        return grouped

    async def read_all_properties(self):
        result = {}
        try:
            if not self.connect():
                logger.error(f"{self.device_custom_name} - No connection on port {self.port}")
                return {}

            grouped_registers = self.group_registers()

            for group in grouped_registers:
                start_address = group['start']
                total_length = group['length']
                try:
                    response = self.client.read_input_registers(
                        start_address, count=total_length, slave=self.device_address
                    )

                    if response.isError():
                        logger.error(f"{self.device_custom_name} - No response from {start_address} address")
                        return {}

                    registers = response.registers
                    idx = 0
                    for name, length in group['items']:
                        spec = self.register_map[name]
                        data = registers[idx: idx + length]
                        result[name] = decode_data(data, spec)
                        idx += length

                except Exception as e:
                    logger.error(
                        f"{self.device_custom_name} - Error reading registers {start_address}-{start_address + total_length}: {str(e)}")
                    return {}

            return result

        except Exception as e:
            logger.error(f"{self.device_custom_name} - Unexpected error: {str(e)}")
            return {}
        finally:
            try:
                self.client.close()
            except Exception as e:
                logger.error(f"{self.device_custom_name} - Error closing connection: {str(e)}")
```