МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

ПрАТ «ПРИВАТНИЙ ВИЩИЙ НАВЧАЛЬНИЙ ЗАКЛАД «ЗАПОРІЗЬКИЙ ІНСТИТУТ ЕКОНОМІКИ ТА ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ»

Кафедра Інформаційних технологій

ДО ЗАХИСТУ ДОПУЩЕНА

Зав.кафедрою _____
д.е.н.,доцент Левицький С.І.

КВАЛІФІКАЦІЙНА БАКАЛАВРСЬКА РОБОТА

СИСТЕМА МОНІТОРИНГУ ЕЛЕКТРОМОБІЛЯ З WEB-ІНТЕРФЕЙСОМ

Виконав
ст. гр. КІ-228 _____ О.В. Сердечний
(підпис)

Керівник
к.т.н. _____ О.А. Хараджян
(підпис)

Запоріжжя
2023

ПРАТ «ПВНЗ «ЗАПОРІЗЬКИЙ ІНСТИТУТ ЕКОНОМІКИ

ТА ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ»

Кафедра Інформаційних технологій

ЗАТВЕРДЖУЮ
Зав. кафедрою

_____
д.е.н.,доцент Левицький С.І.

З А В Д А Н Н Я
НА КВАЛІФІКАЦІЙНУ БАКАЛАВРСЬКУ РОБОТУ

Студенту гр. *КІ – 228*, спеціальності *«Комп'ютерна інженерія»*

Сердечний Олег Вікторович

1.Тема: *Система моніторингу електромобіля з Web-інтерфейсом*.

затверджена наказом по інституту № 02-10 27.01.2023 р.

2. Термін здачі студентом закінченої роботи: 12.06. 2023 р.

3. Перелік питань, що підлягають розробці:

1. Аналіз систем моніторингу параметрів.
2. IoT як засіб оптимізації роботи електромобіля.
3. Засоби підключення до систем електромобіля.
4. Розробка алгоритму моніторингу параметрів.
5. Розробка апаратнлї платформи для системи моніторингу.
6. Аналіз бібліотеки для створення WEB-серверу.
7. Аналіз бібліотеки для взаємодії з інтерфейсом CAN.

8. Інтерфейс бібліотеки для взаємодії за протоколом OBD2.
9. Розробка програма моніторингу.
10. Тестування програми моніторингу.

Дата видачі завдання: 16.01.2023 р.

Студент _____ О.В. Сердечний
(підпис) (прізвище та ініціали)

Керівник роботи _____ О.А. Хараджян
(підпис) (прізвище та ініціали)

РЕФЕРАТ

Кваліфікаційна бакалаврська робота містить 50 сторінок, 3 рисунків, 6 таблиць, 1 додаток, 10 використаних джерел.

Об’єкт роботи: системи моніторингу систем електромобіля.

Предмет роботи: системи моніторингу систем електромобіля з дистанційним доступом.

Мета роботи: розробка системи моніторингу систем електромобіля з Web-інтерфейсом.

Задачі роботи: аналіз систем моніторингу параметрів; IoT як засіб оптимізації роботи електромобіля; засоби підключення до систем електромобіля; розробка алгоритму моніторингу параметрів; розробка апаратнлї платформи для системи моніторингу; аналіз бібліотеки для створення WEB-серверу; аналіз бібліотеки для взаємодії з інтерфейсом CAN; інтерфейс бібліотеки для взаємодії за протоколом OBD2; розробка програма моніторингу; тестування програми моніторингу.

У міру зростання рівня забруднення повітря та деградації навколишнього середовища міські простори шукають шляхи вирішення цієї проблеми. Електромобілі є інноваційним кроком у цьому напрямку. Максимальне використання ресурсів і ефективність електричної інфраструктури зі зниженими витратами є основними прагненнями при прийнятті до використання електромобілів.

Розроблена система дозволяє виконати функції моніторингу. Розроблена система побудована на основі мікроконтролеру. Розроблена програма реалізує точку доступа WiFi до якої можливо під’єднатися за допомогою будь-якого пристрою і у Web-браузері відкрити сторінку з результатами роботи програми.

ЕЛЕКТРОМОБІЛЬ, ЕЛЕКТРОНИЙ БЛОК УПРАВЛІННЯ, МОНІТОРИНГ, CAN, OBD2

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

Скорочення	Повна назва	Пояснення/переклад
ШІ	Штучний інтелект	
BMS	Battery management system	Система управління акумулятором
DTC	Diagnostic trouble codes	Діагностичні коди помилок
EV	Electric vehicle	Електромобіль
ECU	Electronic control unit	Електроний блок керування
IoT	Internet of things	Internet речей
MIL	Malfunction indicator light	Лампа аварійної сигналізації
OBD	On-board diagnostics	Бортова діагностика

ВСТУП

У міру зростання рівня забруднення повітря та деградації навколишнього середовища міські простори шукають шляхи вирішення цієї проблеми. Одним із ключових способів контролювати забруднення повітря в містах є зменшення забруднення транспортними засобами. Електромобілі є інноваційним кроком у цьому напрямку. Електромобілі — це високотехнологічні машини, які використовують багато даних для забезпечення оптимальної продуктивності.

Широкий моніторинг і сповіщення в реальному часі запобігають поломці електромобілів. За допомогою телематичної технології на основі IoT дані збираються при підключенні до датчиків автомобіля, і їх можна швидко відображати через віджети, надсилати миттєві сповіщення та створювати автоматичні звіти.

Роль IoT в управлінні електромобілем: моніторинг системи керування акумулятором, аналіз безпеки та розумне водіння, можна вимірювати абсолютні та відносні параметри якості керування, такі як швидкість, прискорення та інші, для надання підказок у реальному часі для забезпечення кращої продуктивності; крадіжці можна запобігти за допомогою відстеження в режимі реального часу геоданих. Система оповіщення про несправності та профілактичного обслуговування.

Максимальне використання ресурсів і ефективність електричної інфраструктури зі зниженими витратами є основними прагненнями в режимі прийняття до використання електромобілів.

Об’єкт роботи: системи моніторингу систем електромобіля.

Предмет роботи: системи моніторингу систем електромобіля з дистанційним доступом.

Мета роботи: розробка системи моніторингу систем електромобіля з Web-інтерфейсом.

Задачі роботи:

- аналіз систем моніторингу параметрів;
- IoT як засіб оптимізації роботи електромобіля;
- засоби підключення до систем електромобіля;
- розробка алгоритму моніторингу параметрів;
- розробка апаратнлї платформи для системи моніторингу;
- аналіз бібліотеки для створення WEB-серверу;
- аналіз бібліотеки для взаємодії з інтерфейсом CAN;
- інтерфейс бібліотеки для взаємодії за протоколом OBD2;
- розробка програма моніторингу;
- тестування програми моніторингу.

РОЗДІЛ 1
СИСТЕМИ МОНІТОРИНГУ ПАРАМЕТРІВ

1.1. IoT як засіб оптимізації роботи електромобіля

У міру зростання рівня забруднення повітря та деградації навколишнього середовища міські простори шукають шляхи вирішення цієї проблеми. Одним із ключових способів контролювати забруднення повітря в містах є зменшення забруднення транспортними засобами. Електромобілі є інноваційним кроком у цьому напрямку. Електромобілі — це високотехнологічні машини, які використовують багато даних для забезпечення оптимальної продуктивності. Параметри продуктивності включають моніторинг швидкості, прискорення, пробігу, управління батареєю, заряджання, попередження про несправності та системи прогнозованого обслуговування. IoT відіграє велику роль у моніторингу цих електромобілів.

Роль IoT в управлінні електромобілем. Розглянемо кожен аспект системи керування електромобілем IoT і способи, якими вони додають ефективності електромобілів.

Система керування акумулятором. Основними функціями системи керування батареями BMS є моніторинг і контроль процесів батареї. Це включає в себе цикл заряджання та розряджання, що забезпечує справність батареї та мінімізує ризик пошкодження батареї, забезпечуючи оптимізовану подачу енергії для живлення автомобіля. Схема моніторингу в BMS контролює ключові параметри батареї. Вони включають напругу, силу струму та температуру під час заряджання та розряджання. Модуль BMS оцінює такі параметри, як потужність, стан заряду SoC, загальний стан батареї SoH і забезпечує працездатність на основі вимірювань. IoT відіграє важливу роль у

моніторингу та контролі, оскільки він дає змогу віддалено реєструвати дані про параметри акумулятора, умови тощо.

Виробники електромобілів все частіше вибирають високоякісні літій-іонні акумулятори через їх довший термін служби та кращий діапазон високої щільності енергії. Окрім цих переваг, є й деякі недоліки.

У сценаріях, коли батарея виходить з ладу, дані бортового датчика, отримані через IoT, можуть допомогти впоратися з проблемами. Потім їх можна перевірити через моделі на основі ШІ для оцінки продуктивності. Тести можна проводити на деяких літій-іонних батареях, щоб оцінити закономірності часткового та повного заряджання та розряджання. Моделі характеризуються за допомогою даних, зібраних на кожному кроці. Вони інтегруються з ШІ перед розгортанням на сервері. EV надсилає ключові дані датчиків на сервер, надаючи інформацію про подальші дії та продуктивність. Можна сказати, що стан електромобіля контролюється сервером.

Безпека та розумне водіння. Використання IoT також дозволяє контролювати транспортні засоби та їх ключові компоненти в реальному часі. Це допомагає в профілактичному обслуговуванні, яке пропонує технологія, завдяки чому користувачі вважають її більш надійною.

Пристрої IoT, вбудовані в електромобілі, можуть надати користувачам наступні функції.

Можна вимірювати абсолютні та відносні параметри якості керування, такі як швидкість, прискорення та інші, для надання підказок у реальному часі для забезпечення кращої продуктивності;

Крадіжці можна запобігти за допомогою відстеження в режимі реального часу геоданих. Таким чином, буде підвищена безпека, щоб зменшити залежність від страхування.

Пристрій IoT відстежуватиме дані про ефективність транспортного засобу, на основі яких виробники електромобілів і акумуляторів можуть покращувати продукти. Параметри включають діапазон для кожного заряду, використання транспортного засобу, відмінності в продуктивності залежно від

географії, погодних умов, віку та зміни діапазону для кожного заряду за певний період.

Система оповіщення про несправності та профілактичного обслуговування. Будучи високотехнологічною машиною, електромобіль обов'язково має технічні збої. Системи оповіщення про несправності на основі IoT допомагають попередити водіїв транспортних засобів про несправності електромобілів, даючи їм час діяти та усунути їх.

Незважаючи на те, що електромобілі ретельно розроблені, можуть виникати ситуації, коли компоненти можуть ламатися або виходити з ладу. Щоб передбачити це, ключову роль можуть зіграти алгоритми штучного інтелекту та віддалені дані IoT. Це допомагає завчасно попередити користувачів електромобілів, щоб усунути можливі проблеми або запобігти повній поломці. Це призведе до кращого досвіду клієнтів, оскільки вони вважатимуть це надійним.

Надзвичайно важливо стежити за загальними умовами температури та вологості в різних регіонах і віддалено відстежувати продуктивність. Це не тільки допоможе швидко вирішити проблеми, але й забезпечить користувачам відчуття комфорту та безпеки.

Телематичні дані. За допомогою телематичної технології на основі IoT дані збираються при підключенні до датчиків автомобіля, і їх можна швидко відображати через віджети, надсилати миттєві сповіщення та створювати автоматичні звіти.

Конкретні дані, пов'язані з транспортними засобами, є основною рушійною силою прийняття таких рішень, як інвестиції в інфраструктуру зарядних станцій, диверсифікацію автопарку та підвищення кваліфікації водіїв. Саме тому парки електромобілів вважаються близькими до «переломної точки» масового впровадження. Моніторинг електромобілів забезпечує надійність шляхом виявлення, діагностики та прогнозування.

Урядова політика щодо скорочення викидів змінить сучасний вигляд транспорту. Містобудівники та менеджери використовують потужність

транспортних даних для чистого довкілля без викидів. Електромобіль(и) відповідають на ці складні питання, ґрунтуючись на своїй здатності надавати надійні дані в реальному часі. Інтелектуальна система моніторингу електромобілів збирає, звітує, аналізує та перекладає ці дані для ефективного управління автопарком для реалізації більш широких завдань.

Дані моніторингу електромобіля. Рішення для управління парком електромобілів зазвичай базуються на телематиці зі зручними портативними пристроями. Під час моніторингу електричного парку в реальному часі збираються та обробляються конкретні дані для індивідуального маршруту, який найкраще підходить для кожного транспортного засобу. Сигнал GPS і аналітика даних, які є основою моніторингу продуктивності парку, дозволяють робити надійні прогнози, спрямовані на підвищення ефективності. Ефективний моніторинг дає додаткову інформацію.

Дані транспортного засобу - Профілювання транспортних засобів працює в хмарі та постійно вивчається та вдосконалюється. Дані про електромобілі інформують менеджерів про фактичний запас ходу транспортних засобів, споживання енергії та загальну продуктивність автопарку.

Дані батареї EV - Дані про використання акумулятора та стан заряджання можуть бути корисними для визначення справності акумулятора. Моніторинг електромобілів підтримує оцінку стану заряду парків і окремих транспортних засобів. Дані корисні для того, щоб уникнути тривоги щодо заряду.

Дані про маршрут і розклад - Оптимальне планування маршруту електропарку є ключовою динамікою загальної вартості володіння електромобілем. Інтегрований підхід із вбудованими картами Google збирає дані в режимі реального часу для найкращих можливих точок зарядки на маршруті автомобіля. Цей процес гарантує, що автомобіль продовжує працювати з безпечним лімітом заряду, і не викликає занепокоєння водіїв або менеджерів автопарків. Крім того, весь електропарк працює синхронно з показниками заряду електроенергії та оптимальним використанням активів. Такий моніторинг виявився корисним, зокрема, для електробусів.

Дані неконтрольованого фактора - Моніторинг неконтрольованих умов, як-от географія, тип дороги чи клімат, дав інформацію про стан заряду, що корисно для точного прогнозування діапазону протягом певної пори року чи дня.

Дані про поведінку водія - І останнє, але не менш важливе: на основі даних на ходу можна досягти зміни поведінки водія в бік більш безпечного водіння. Механізм моніторингу є досить ефективним щодо рівня аварійності та оперативного введення коригувальних заходів.

Максимальне використання ресурсів і ефективність електричної інфраструктури зі зниженими витратами є основними прагненнями в режимі прийняття EV. Кілька можливостей, які існують в основі моніторингу електромобілів, це:

- уникнення дорогих модернізацій електричної інфраструктури;
- технологічний прогрес і масштабованість;
- використання IoT та розумних технологій для пом’якшення витратних бар’єрів;
- досягнення ефективності використання електропарків.

Роль телематики в електромобілях. Глобальне потепління, швидке виснаження природних ресурсів і зростання глобальних викидів вуглецю є значними факторами, що сприяють збільшенню кількості електромобілів. Електромобілі виробляють нульові викиди, що може значно покращити якість повітря. Однак з електромобілями пов’язані певні проблеми. Одним з таких є віддалений моніторинг транспортних засобів для ефективної роботи.

Існує кілька проблем, пов’язаних із широким впровадженням електромобілів. Виробники акумуляторів для електромобілів стикаються з кількома технічними та вартісними проблемами. Літій-іонна батарея в автомобілях EV досягла своєї межі, і виробники досліджують розробку передової технології акумуляторів. Ще однією серйозною проблемою є відсутність зарядної інфраструктури для електромобілів.

Телематика може забезпечити вирішення цих проблем шляхом легкого захоплення, обміну та зберігання даних автомобільної телеметрії. Інтеграція акумуляторів електромобілів із телематичними системами дозволить виробникам контролювати продуктивність акумулятора та визначати потенційні причини негативного впливу на термін служби та продуктивність акумулятора. Чітке визначення особливостей заряджання та розряджання акумуляторів електромобілів дозволить спростити роботу з конструкцією акумулятора для покращення відстані та терміну служби акумулятора.

Широкий моніторинг і сповіщення в реальному часі запобігають поломці електромобілів. Журнали даних завжди доступні для проведення аналізу вирішення проблем, щоб уникнути подібних випадків у майбутньому.

Отримуйте сповіщення, коли ваші батареї наближаються до критичного рівня і настав час їх зарядити.

Активні сервісні сповіщення, такі як сповіщення одометра, забезпечують простий спосіб створити профілактичний догляд і заощадити додаткові витрати.

Звітування про ефективність і визначення впливу та повернення інвестицій від електрифікації

Параметри управління флотом, такі як відстеження та оптимізація маршрутів, виробництво суден тощо, пропонують можливість досягти кращого рівня обслуговування клієнтів.

Пріоритезація підзарядки залежно від стану батареї та графіків роботи

Програмне забезпечення можна інтегрувати із зарядними станціями, щоб представити найближчу станцію як автостоянку. Це може бути ефективною допомогою для користувачів електромобілів.

Покращений моніторинг часу роботи акумулятора покращує цикли зарядки.

Зменшення витрат завдяки плануванню циклу зарядки на основі пікових тарифів на електроенергію

1.2. Засоби підключення до систем електромобіля

Бортова діагностика OBD — це система, що забезпечує самодіагностику автомобіля та можливість моніторингу. Системи OBD надають власнику транспортного засобу або майстру з ремонту доступ до стану різних підсистем автомобіля. Обсяг діагностичної інформації, доступної через OBD, сильно змінювався з моменту його появи у різних версіях бортових комп'ютерів автомобіля. Ранні версії OBD просто висвітлювали індикатор несправності MIL, якщо було виявлено проблему, але не надавали жодної інформації щодо характеру проблеми. Сучасні системи OBD використовують стандартизований цифровий комунікаційний порт для надання даних у реальному часі на додаток до стандартизованої серії діагностичних кодів несправностей DTC, які дозволяють швидко визначати несправності в автомобілі.

Стандарт OBD-II є вдосконаленням стандарту OBD-I як щодо можливостей, так і щодо стандартизації. Стандарт OBD-II визначає тип діагностичного роз'єму та його контакти, доступні протоколи електричних сигналів та формат повідомлень. Він також надає список рекомендованих параметрів транспортного засобу для моніторингу разом із способами кодування даних. У роз'ємі є контакт, який забезпечує живлення скануючого приладу від акумулятора автомобіля, що позбавляє від необхідності підключати скануючий прилад до джерела живлення окремо. Стандарт OBD-II надає розширений список кодів DTC.

Стандартизація OBD-II була викликана вимогами щодо шкідливих викидів, і хоча через нього передбачалось передавати лише коди та дані, пов'язані з шкідливими викидами, більшість виробників зробили роз'єм каналу передачі даних OBD-II єдиним у транспортному засобі, за допомогою якого діагностуються всі системи.

Діагностичні коди несправностей OBD-II складаються з 4 цифр, перед якими стоїть літера: P - для трансмісії (двигун і трансмісія), B - для кузова, C - для шасі та U - для мережі.

OBD-II надає доступ до даних з блоку керування двигуном (engine control unit ECU) і є цінним джерелом інформації під час усунення несправностей усередині автомобіля. Стандарт SAE J1979 визначає метод запиту різноманітних діагностичних даних і список стандартних параметрів, які можуть бути доступні з ECU. Різні параметри, які доступні, адресуються ідентифікаційними номерами параметрів (parameter identification numbers, PID), які визначені в J1979. Від виробників не вимагається впровадження всіх PID, перелічених у J1979, і їм дозволяється включати власні PID, яких немає в списку. Система запиту PID і отримання даних надає доступ до даних в реальному часі, а також до DTC. Індивідуальні виробники часто доповнюють набір кодів OBD-II додатковими власними кодами DTC.

OBD-II PID – це коди, які використовуються для запиту даних від автомобіля, які використовуються як діагностичний інструмент. Усі легкі транспортні засоби, продані в Північній Америці з 1996 року, а також транспортні засоби середньої вантажопідйомності, починаючи з 2005 року, і транспортні засоби великої вантажопідйомності, починаючи з 2010 року, потрібні для підтримки діагностики OBD-II за допомогою стандартизованого каналу передачі даних роз'єм і підмножина визначених SAE J1979 PID.

В останньому стандарті OBD-II SAE J1979 описано десять режимів роботи (табл. 1.1).

Таблиця 1.1 - Режими роботи OBD-II SAE J1979

Номер режиму	Опис
0x01	Показати поточні дані
0x02	Показати дані стоп-кадру
0x03	Показати збережені діагностичні коди несправностей

0x04	Очистити діагностичні коди несправностей і збережені значення
0x05	Результати тестування, моніторинг датчика кисню (тільки не CAN)
0x06	Результати випробувань, моніторинг інших компонентів/систем (результати випробувань, моніторинг датчика кисню лише для CAN)
0x07	Показати незавершені діагностичні коди несправностей (виявлені під час поточного або останнього циклу водіння)
0x08	Контроль роботи бортового компонента/системи
0x09	Запит інформації про автомобіль
0x0A	Постійні DTC (Очищені DTC)

Виробники транспортних засобів не зобов’язані підтримувати всі режими.

Кожен виробник може визначити додаткові режими вище 9 (наприклад: режим 22, як визначено SAE J2190 для Ford/GM, режим 21 для Toyota) для іншої інформації (наприклад: напруга тягової батареї у HEV).

Стандартні PID. У таблиці 2.2 показано деякі стандартні OBD-II PID, які визначені у SAE J1979. Очікувана відповідь для кожного PID є разом з інформацією про те, як перевести відповідь у значущі дані. Знову ж таки, не всі транспортні засоби будуть підтримують усі PID, і можуть бути визначені виробником спеціальні PID, які не визначені в стандарті OBD-II.

Зверніть увагу, що режими 1 і 2 в основному ідентичні, за винятком того, що режим 1 надає поточну інформацію, тоді як режим 2 забезпечує моментальний знімок тих самих даних, отриманих на момент встановлення останнього діагностичного коду несправності. Винятки є PID 01, який доступний лише в режимі 1, і PID 02, який доступний лише в режимі 2. Якщо режим 2 PID 02 повертає нуль, тоді немає знімка, і всі інші дані режиму 2 не мають сенсу.

Таблиця 1.2 - Приклади PID

Режим(hex)	PID(hex)	Возвращённые данные	Описание	Min значение	Max значение	Units	Формула
1	5	1	Температура охлаждающей жидкости	-40	215	°C	A-40
1	0A	1	Давление топлива	0	765	kPa (gauge)	A*3
1	0B	1	Давление во впускном коллекторе (абсолютное)	0	255	kPa (absolute)	A
1	0C	2	Обороты двигателя	0	16,383.75	rpm	((A*256)+B)/4
1	0D	1	Скорость автомобиля	0	255	km/h	A
1	0F	1	Температура всасываемого воздуха	-40	215	°C	A-40
1	10	2	Массовый расход воздуха	0	655.35	grams/sec	((A*256)+B) / 100
1	11	1	Положение дроссельной заслонки	0	100	%	A*100/255
1	5B	1	Заряд силовой батареи гибрида	0	100	%	A*100/255

Формат шини CAN (11 біт). Запит PID і відповідь надходять на шину CAN автомобіля. Стандартні запити та відповіді OBD використовують функціональні адреси. Діагностичний зчитувач ініціює запит, використовуючи

CAN ID 7DFh, який діє як широкомовна адреса, і приймає відповіді від будь-якого ідентифікатора в діапазоні від 7E8h до 7EFh. ЕБУ, які можуть відповідати на запити OBD, слухають як функціональний ідентифікатор трансляції 7DFh, так і один призначений ідентифікатор у діапазоні від 7E0h до 7E7h. Їхня відповідь містить ідентифікатор із присвоєного їм ідентифікатора плюс 8, наприклад. 7E8h до 7EFh.

Цей підхід дозволяє використовувати до восьми ECU, кожен з яких незалежно відповідає на запити OBD. Діагностичний зчитувач може використовувати ідентифікатор у кадрі відповіді ECU для продовження зв’язку з певним ECU. Зокрема, мультикадровий зв’язок вимагає відповіді на конкретний ідентифікатор ECU, а не на ідентифікатор 7DFh.

Шина CAN також може використовуватися для зв’язку поза стандартними повідомленнями OBD. Фізична адресація використовує певні ідентифікатори CAN для конкретних модулів (наприклад, 720h для панелі приладів у Ford) із власним корисними навантаженнями кадрів.

Функціональний запит PID надсилається до автомобіля по шині CAN за ідентифікатором 7DFh, використовуючи 8 байтів даних (табл. 1.3).

Таблиця 1.3 - Структура пакету запиту

	Байт							
PID Тип	0	1	2	3	4	5	6	7
SAE Standard	Число додаткових байтів даних: 2	Сервіс 01 = поточні дані; 02 = стоп-кадр; тощо	код PID	не використовується = CCh				
Vehicle specific	Число додаткових байти даних: 3	Спеціальний сервіс	код PID (два байта)		не використовується = CCh			

Відповідь. Автомобіль відповідає на запит PID на шині CAN ідентифікаторами повідомлень, які залежать від того, який модуль відповів. Зазвичай двигун або основний ЕБУ реагують на ID 7E8h. Інші модулі, наприклад гібридний контролер або контролер батареї в Prius, відповідають на 07E9h, 07EAh, 07EBh тощо. Це на 8h вище, ніж фізична адреса, на яку відповідає модуль. Незважаючи на те, що кількість байтів у повернутому значенні є змінною, повідомлення CAN використовує 8 байтів даних.

Таблиця 1.4 - Структура пакету відповіді

<table>
<tr><td rowspan="2">Байт</td><td colspan="3">CAN Address</td></tr>
<tr><td>SAE Standard
7E8h, 7E9h, 7EAh та інші</td><td>Vehicle specific
7E8h або 8h + фізичний ID модуля</td><td>Vehicle specific
7E8h або 8h + фізичний ID модуля</td></tr>
<tr><td>0</td><td>Число додаткових байти даних: 3 до 6</td><td>Число додатковихй байт даних: 4 до 7</td><td>Число додатковихй байт даних: 3</td></tr>
<tr><td>1</td><td>Номер служби плюс 40h</td><td>Номер служби плюс 40h</td><td>7Fh – модуль не розпізнав запит</td></tr>
<tr><td>2</td><td>код PID</td><td rowspan="2">код PID (два байта)</td><td>Номер служби</td></tr>
<tr><td>3</td><td>значення параметра, байт 0</td><td>31h</td></tr>
<tr><td>4</td><td>значення, параметра байт 1 (необов'язково)</td><td>значення параметра, байт 0</td><td rowspan="4">не використовується (00h)</td></tr>
<tr><td>5</td><td>значення, параметра байт 2 (необов'язково)</td><td>значення параметра, байт 1 (необов'язково)</td></tr>
<tr><td>6</td><td>значення, параметра байт 3 (необов'язково)</td><td>значення параметра, байт 2 (необов'язково)</td></tr>
<tr><td>7</td><td>не використовується (00h або 55h)</td><td>значення параметра, байт 3 (необов'язково)</td></tr>
</table>

РОЗДІЛ 2
РОЗРОБКА АЛГОРИТМУ МОНІТОРИНГУ ПАРАМЕТРІВ

2.1. Апаратна платформа для системи моніторингу

ESP32 — це мікроконтролер з інтегрованими модулями Wi-Fi і Bluetooth на частоті 2.4 ГГц. Ядром мікроконтролера є малопотужний процесор TSMC 40 нм технології.

ESP32 — це високоінтегроване рішення для додатків Wi-Fi і Bluetooth IoT із приблизно 20 зовнішніми компонентами. ESP32 інтегрує антенний перемикач, радіочастотний балун, підсилювач потужності, малошумний приймальний підсилювач, фільтри, і модулі керування живленням. Таким чином, вся схема займає мінімум площі друкованої плати.

ESP32 розроблений для мобільних пристроїв, носимої електроніки та додатків Інтернету речей (IoT). Він може забезпечити дрібнозернисту синхронізацію, кілька режимів живлення, і динамічне масштабування потужності. Наприклад, у прикладному сценарії концентратора сенсора IoT з низьким енергоспоживанням ESP32 виводиться з режиму сну періодично лише тоді, коли виявляється певний стан. Регулювання потужності вихідного підсилювача також оптимізується з урахуванням компромісу між дальністю зв'язку, швидкістю передачі даних і енергоспоживанням.

ESP32 побудовано з використанням технології CMOS для однокристального повністю інтегрованого рішення радіосистеми, а також інтегрує розширені схеми калібрування, які дозволяють усунути дефекти зовнішніх ланцюгів або пристосуватися до змін зовнішні умови. Таким чином, масове виробництво рішень ESP32 не потребує спеціалізованого обладнання для тестування Wi-Fi.

Таблиця 2.1 - Характеристики мікроконтролера

Модуль	Характеристика
CPU	Xtensa® single-/dual-core 32-bit LX6 microprocessor 240 MHz
ROM	448 KB
SRAM	520 KB
SRAM in RTC	16 KB
QSPI	Підтримка додаткової пам’яті flash/SRAM chips
Clocks	Внутрішній генератор 8 MHz Внутрішній RC генератор Зовнішний 2-60 MHz кварцевий генератор Зовнішний 32 kHz кварцевий генератор для RTC
Timers	Дві групи таймерів 2 × 64-bit та watchdog у кожній групі Один RTC таймер та watchdog
Peripheral Interfaces	34 програмованих GPIO 12-bit SAR АЦП до 18 каналів 2 × 8-bit ЦАП 4 × SPI 2 × I2S 2 × I2C 3 × UART 1 host (SD/eMMC/SDIO)
Основні функції WiFi	802.11n (2,4 ГГц), до 150 Мбіт/с TX/RX A-MPDU, RX A-MSDU Автоматичний моніторинг Beacon (апаратний TSF) 4 × віртуальні інтерфейси Wi-Fi

2.2. Бібліотека для створення WEB-серверу

ESP32 HTTPS Server

Бібліотек реалізує HTTPS сервер, який можна використовувати з ESP32 Arduino Core. Ця бібліотека також підтримує протокол HTTP.

Особливості бібліотеки:

- забезпечення підтримки протоколів HTTP, HTTPS або обох одночасно;
- обробка запитів у функціях зворотного виклику, які можуть бути прив’язані до URL-адрес;
- абстракція обробки запитів HTTP та забезпечення простого API для цього, наприклад, для доступу до параметрів, заголовків, HTTP Basic Auth тощо;
- використання функцій проміжного програмного забезпечення як проксі для кожного запиту для виконання централізованих завдань, таких як аутентифікація або журналювання;
- використовується вбудоване шифрування модуля ESP32 для HTTPS;
- паралельна робота з кількома клієнтами (макс. 3-4 клієнти TLS через обмеження пам’яті);
- використання підключення: підтримка активності та повторне використання сеансу SSL для зменшення накладних витрат на підключення SSL і прискорення передачі даних.

Бібліотека є самодостатньою, їй потрібні лише системні бібліотеки Arduino та ESP32. Також при використанні модуля WiFi потрібна бібліотека WiFi.

Приклади роботи бібліотеки демонструють основні концепції та підходи до побудови програми.

- Static-Page: короткий приклад, що показує, як обслуговувати деякі статичні ресурси на сервері;
- Parameters: показує, як можна отримати доступ до параметрів запиту (частина після знака питання в URL-адресі) або параметрів у динамічних URL-адресах (наприклад, /led/1, /led/2, ...);
- Put-Post-Echo: реалізує просту службу відлуння для запитів PUT і POST, яка повертає тіло запиту як тіло відповіді. Також показано, як відрізнити кілька методів HTTP для однієї URL-адреси;
- HTTPS-i-HTTP: показує, як обслуговувати ресурси через HTTP і HTTPS паралельно та як перевірити, чи використовує користувач безпечне з'єднання під час обробки запиту;
- Middleware: показує, як використовувати API проміжного програмного забезпечення для журналювання. Функції проміжного програмного забезпечення дуже подібні до веб-серверів;
- Authentication: реалізує ланцюжок із двох функцій проміжного програмного забезпечення для обробки аутентифікації та авторизації за допомогою базової аутентифікації HTTP;
- Sync-Server: як у прикладі Static-Page, але сервер виконується в окремому завданні на ESP32, тому не потрібно викликати функцію loop() у основному циклі;
- Websocket-Chat: надає чат у веб-переглядачі, побудований на веб-сокетах;
- Parameter-Validation: показує, як можна інтегрувати функції валідатора для виконання формальних перевірок параметрів у URL-адресі;
- Самопідписаний сертифікат: показує, як генерувати самопідписаний сертифікат на льоту в ESP під час запуску програми;
- REST-API: використовує завантаження файлів ArduinoJSON і SPIFFS для обслуговування невеликого веб-інтерфейсу, який надає REST API.

Якщо під час виконання прикладу ви зіткнулися з повідомленнями про помилку про те, що cert.h або private_key.h відсутні, спочатку запустіть create_cert.sh (див. Інструкції зі встановлення).

Для нормальної роботи бібліотеки необхідно потрібно спочатку виконати сценарій extras/create_cert.sh. Цей сценарій створить простий ключ для підпису сертифікатів (файли cert.h або private_key.h).

Щоб мати можливість налаштувати сервер, необхідні наступні компоненти.

```
// Включення файлу для налаштування сервера
#include <HTTPSServer.hpp>
// Включення файлу даних сертифіката для сервера (сертифікат і закритий ключ)
#include <SSLCert.hpp>
// Включення файлу визначення зворотних викликів обробника запитів
#include <HTTPRequest.hpp>
#include <HTTPResponse.hpp>
// Необхідно визначити ResourceNodes
#include <ResourceNode.hpp>
// Полегшення доступу до класів сервера
using namespace httpsserver
```

Створення екземпляру сервера HTTP:

```
HTTPServer myServer = HTTPServer();
```

Створення екземпляру сервера HTTPS:

```
// Створення даних сертифіката (див. extras/README.md, як це створити)
SSLCert cert = SSLCert(
crt_DER, // Дані сертифіката у форматі DER
crt_DER_len, // довжина даних сертифіката
key_DER, // закритий ключ для цього сертифіката
key_DER_len // Довжина закритого ключа
```

```
);
// Налаштувати сервер із конфігурацією за замовчуванням і сертифікат, який був вказаний раніше
HTTPSServer myServer = HTTPSServer(cert);
```

За замовчуванням сервер слухатиме порт 443. Якщо необхідно змінити його (або деякі інші параметри), то порт вказується у додаткових параметрах конструкторів HTTPServer або HTTPSServer.

Єдина відмінність між версіями HTTP і HTTPS полягає в сертифікаті, який потрібно налаштувати. Кожне створене з'єднання HTTPSServer з підтримкою TLS потребує додатково від 40 до 50 Кб пам'яті купи (heap) для самого з'єднання TLS. Це слід враховувати при збільшенні maxConnections.

Кожен маршрут (або шлях), який має бути доступним на сервері, має бути налаштований як так званий ResourceNode. Такий вузол пов'язує функцію обробки з певним маршрутом (наприклад, /test) і методом HTTP (наприклад, GET). Функція обробки може виглядати так:

```
void handleRoot(HTTPRequest * req, HTTPResponse * res) {
// Надаємо HTML-сторінку, тому встановлюємо тип вмісту
res->setHeader("Content-Type", "text/html");
// Відповідь реалізує інтерфейс Print, тому використовується так само, як запис в Serial.
res->println("<!DOCTYPE html>");
res->println("<html>");
res->println("<head><title>Hello World!</title></head>");
res->println("<body>");
res->println("<h1> Hello World!</h1>");
res->print("<p>... from your ESP32!</p>");
res->println("</body>");
res->println("</html>");
}
```

Функція отримує посилання на HTTPRequest і HTTPResponse. Можна використовувати запит для читання заголовків, параметрів, інформації аутентифікації тощо. Відповідь можна використовувати для надсилання даних клієнту, встановлення заголовків або кодів стану HTTP.

Тепер необхідно повідомити серверу, яку URL-адресу має обслуговувати ця функція. Це можна зробити, створивши `ResourceNode` (зазвичай у функції `setup()`).

```
ResourceNode * nodeRoot = new ResourceNode("/", "GET", &handleRoot);
```

Перший параметр визначає маршрут. Він завжди має починатися зі слеша, що означає, що функція буде викликана для запитів до кореня сервера (наприклад, https://10.0.x.x/).

Другим параметром є метод HTTP, у цьому випадку "GET".

І наступним параметром передається посилання на функцію обробки запитів, щоб зв’язати її з маршрутом і методом.

Тепер потрібно зареєструвати створений ResourceNode на сервері:

```
myServer.registerNode(nodeRoot);
```

Це все, що потрібно зробити для однієї веб-сторінки на сервері.

Можна визначити один ResourceNode через HTTPServer::setDefaultNode(), який буде викликаний, якщо жоден інший вузол на сервері не відповідає. У цьому випадку метод і маршрут ігноруються. У більшості випадків це використовується для визначення обробника помилки відсутності ресурсу 404. Якщо вузол за замовчуванням не вказано, сервер повернеться з невеликою сторінкою помилки, якщо відповідного маршруту не знайдено.

Виклик HTTPServer::start() запустить сервер, щоб він прослуховував попередньо вказаний порт:

```
myServer.start();
```

Цей код зазвичай входить у функцію setup(). Можна використовувати `HTTPServer::isRunning()`, щоб перевірити, чи успішно запущено сервер.

За замовчуванням потрібно явно передати керування серверу. Це робиться шляхом виклику функції `HTTPServer::loop()`, яка зазвичай додається до функції `loop()` програми. Після виклику сервер спочатку перевірить наявність вхідного з'єднання (до максимальної кількості з'єднань, визначеної в конструкторі), а потім оброблятиме кожне відкрите з'єднання, якщо воно має нові дані в сокеті. Таким чином, ваші функції обробки запитів будуть викликані під час виклику `loop()`. Зауважте, що якщо одна з ваших функцій обробника блокує, вона блокуватиме й усі інші з'єднання.

Якщо необхідно, щоб сервер працював у фоновому режимі (а не викликав `loop()` самостійно кожні кілька мілісекунд), можна скористатися функцією завдань ESP32 і поставити весь сервер в окреме завдання.

Щоб заощадити простір програми на мікроконтролері, є деякі частини бібліотеки, які можна вимкнути під час компіляції, і тоді вони не будуть частиною вашої програми. Наразі доступні такі позначки:

`HTTPS_DISABLE_SELFSIGNING` - видаляє код для створення самопідписаного сертифіката під час виконання. Вам потрібно буде надати дані сертифіката та приватного ключа з іншого джерела даних, щоб використовувати сервер HTTPSServer.

Сервер забезпечує деяке внутрішнє журналювання, яке за замовчуванням активовано на рівні INFO. Це виглядатиме так на послідовній консолі:

```
[HTTPS:I] New connection. SocketFID=55
[HTTPS:I] Request: GET / (FID=55)
[HTTPS:I] Connection closed. Socket FID=55
```

Виведення журналу також можна контролювати за допомогою прапорів компілятора.

Є два параметри, які можна налаштувати:

`HTTPS_LOGLEVEL` визначає рівень журналу для використання

`HTTPS_LOGTIMESTAMP` додає мітку часу (на основі часу роботи) до кожного запису журналу

Таблиця 2.2 - Рівені журналу

Значення HTTPS_LOGLEVEL	Error	Warning	Info	Debug
0				
1	✓			
2	✓	✓		
3	✓	✓	✓	
4	✓	✓	✓	✓

HTTP сервер забезпечує відображення сторінок, що додано як node. Для кожного URL необхідно створити окремий node.

2.3. Бібліотека для взаємодії з інтерфейсом CAN

Шина CAN була розроблена компанією BOSCH як багатоголовна система широкомовної передачі повідомлень, яка визначає максимальну швидкість передачі сигналу 1 Мбіт/с. На відміну від традиційних мереж, таких як USB або Ethernet, CAN не надсилає великі блоки даних точка-точка від вузла A до вузла B під наглядом головного центрального шини. У мережі CAN багато коротких повідомлень, як-от температура або кількість обертів на хвилину,

транслюються на всю мережу, що забезпечує узгодженість даних у кожному вузлі системи. Після пояснення основ CAN, таких як формат повідомлення, ідентифікатори повідомлення та побітовий арбітраж, головна перевага схеми сигналізації CAN, розглядається реалізація шини CAN, представлені типові сигнали та досліджуються функції трансивера.

Стандарт CAN CAN — це шина послідовного зв’язку, визначена Міжнародною організацією стандартизації (ISO), спочатку розроблена для автомобільної промисловості для заміни складного джгута проводів на двопровідну шину. Специфікація вимагає високої стійкості до електричних перешкод і здатності самостійно діагностувати та виправляти помилки даних. Ці функції привели до популярності CAN у різних галузях, включаючи автоматизацію будівель, медицину та виробництво. Протокол зв’язку CAN, ISO-11898: 2003, описує, як інформація передається між пристроями в мережі, і відповідає моделі взаємозв’язку відкритих систем (OSI), яка визначається в термінах рівнів. Фактичний зв'язок між пристроями, підключеними фізичним середовищем, визначається фізичним рівнем моделі. Архітектура ISO 11898 визначає два найнижчі рівні семирівневої моделі OSI/ISO як рівень каналу даних і фізичний рівень на малюнку 1

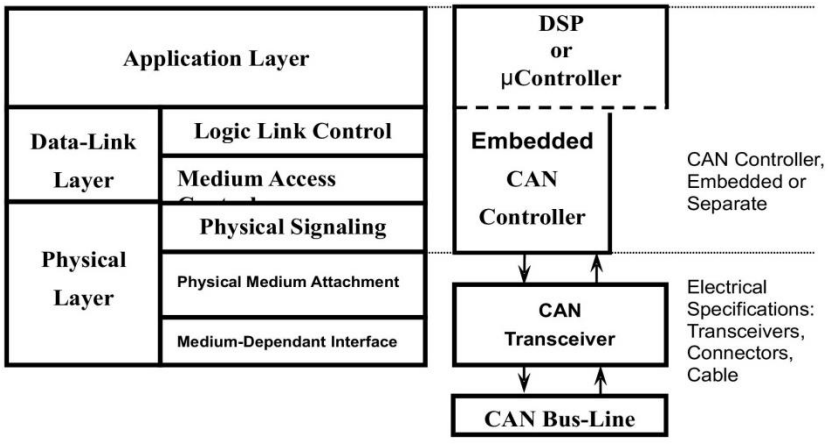

Рис.2.1 – Рівні мережевої архітектури в стандарті ISO 11898

На рис. 2.1 прикладний рівень встановлює зв’язок із протоколом програми верхнього рівня, таким як незалежний від постачальника протокол

CANopen™. Цей протокол підтримується міжнародною групою користувачів і виробників CAN in Automation (CiA). Багато протоколів призначені для конкретних застосувань, таких як промислова автоматизація, дизельні двигуни або авіація. Іншими прикладами стандартних протоколів на основі CAN є CAN Kingdom від KVASER і DeviceNet™ від Rockwell Automation

Стандартний CAN або розширений CAN. Протокол зв'язку CAN — це протокол множинного доступу з визначенням несучої, із виявленням колізій і арбітражем щодо пріоритету повідомлення (CSMA/CD+AMP). CSMA означає, що кожен вузол на шині повинен чекати визначений період бездіяльності, перш ніж спробувати надіслати повідомлення. CD+AMP означає, що колізії вирішуються за допомогою побітового арбітражу на основі попередньо запрограмованого пріоритету кожного повідомлення в полі ідентифікатора повідомлення. Ідентифікатор з вищим пріоритетом завжди отримує доступ до шини. Тобто остання логіка з високим рівнем ідентифікатора продовжує передачу, оскільки вона має найвищий пріоритет. Оскільки кожен вузол на шині бере участь у записі кожного біта «як він записується», арбітражний вузол знає, чи розмістив він логічний старший біт на шині. Стандарт ISO-11898:2003 зі стандартним 11-бітним ідентифікатором забезпечує швидкість передачі сигналу від 125 Кбіт/с до 1 Мбіт/с. Пізніше стандарт був доповнений «розширеним» 29-бітним ідентифікатором. Стандартне 11-бітне поле ідентифікатора на малюнку 2 забезпечує 211 або 2048 різних ідентифікаторів повідомлень, тоді як розширений 29-бітовий ідентифікатор на малюнку 3 забезпечує 229 або 537 мільйонів ідентифікаторів.

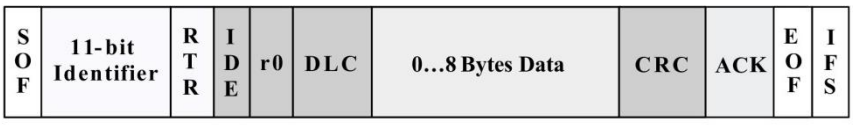

Рис. 2.2 – Пакет CAN з ідентифікатором 11-bit.

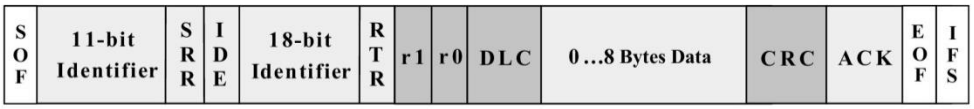

Рис. 2.3 – Пакет CAN з ідентифікатором 29-bit.

Програмний інтерфейс бібліотеки CAN. Перед використанням функцій бібліотеки необхідно підключити її файл декларацій

```
#include <CAN.h>
```

Ініціалізізація інтерфейсу CAN виконується з заданою швидкістю передачі даних bitrate у бітах за секунду (1M, 500k, 250k, 200k, 125k, 100k, 80k, 50k, 40k , 20k, 10k, 5k)

```
CAN.begin(bitrate);
```

Функція повертає «1» у разі успіху, «0» у разі невдачі.

Перед ініціалізацією інтерфейса необхідно виконати налаштування виводів процесора. У випадку використання зовнішнього модуля MCP2515 необхідно визначити виводи сигналів CS і INT, які використовуються бібліотекою.

```
CAN.setPins(cs, irq);
// cs - новий пін для сигналу вибору;
// irq - новий пін для сигналу переривання. Повинен мати можливість налаштування переривання через attachInterrupt.
```

У випадку використання внутрішнього модуля ESP32 необхідно визначити контакти CTX і CRX, які використовуються бібліотекою.

```
CAN.setPins(rx, tx);
```

```
// rx - новий пін CRX для використання;
// tx - новий пін CTX для використання.
```

Також у випадку використання зовнішнього модуля MCP2515 необхідно встановити частоту SPI шини.

```
CAN.setSPIFrequency(частота);
// frequency - нова частота SPI шини для використання.
```

Деякі перетворювачі логічного рівня не можуть підтримувати високі швидкості, такі як 10 МГц.

Далі необхідно встановити тактову частоту MCP2515, яка визначається або зовнішним генератором, або кварцевим резонатором

```
CAN.setClockFrequency(clockFrequency);
// clockFrequency - нова тактова частота.
```

Зупинити роботу інтерфейсу CAN можна за допомогою функції

```
CAN.end()
```

Надсилання даних починається з формування пакету для відправки. Бібліотека підтримує два режима стандартний та розширений.

```
CAN.beginPacket(id);
CAN.beginPacket(id, dlc);
CAN.beginPacket(id, dlc, rtr);
CAN.beginExtendedPacket(id);
CAN.beginExtendedPacket(id, dlc);
CAN.beginExtendedPacket(id, dlc, rtr);
// id - 11-бітний ідентифікатор (стандартний пакет) або 29-бітний ідентифікатор пакета (розширений пакет);
```

```
// dlc - значення поля Data Length Code (DLC) пакета, за замовчуванням це розмір даних, записаних у пакеті;
// rtr - значення поля Remote Transmission Request (RTR) пакета (`false` або `true`), за замовчуванням `false`. Пакети RTR не містять даних, поле DLC пакета представляє запитувану довжину.
```

Після формування пакету у нього необхідно записати дані для відправки. Кожен пакет може містити до 8 байт.

```
CAN.write(byte);
CAN.write(buffer, length);
// byte - один байт для запису в пакет;
// buffer - дані для запису в пакет;
// length - розмір даних для запису.
```

Функція повертає кількість записаних байтів. Для бібліотеки CAN можна використовувати потокові методи, які аналогічні Print.

Завершити створення пакету та відправити його можна функцією

```
CAN.endPacket()
```

Отримання даних є більш складним процесом. Необхідно перевірити, чи отримано пакет.

```
int packetSize = CAN.parsePacket();
```

Функція повертає розмір пакета в байтах або 0, якщо пакет не отримано. Для пакетів RTR розмір відображає поле DLC пакета.

Для отримання та аналізу пакету у функції переривання необхідно зареєструвати функцію зворотного виклику для отримання пакета.

```
CAN.onReceive(onReceive);
```

```
void onReceive(int packetSize) {
  // ...
}
// onReceive - функція для виклику при отриманні пакету.
```

Для полученого пакету можна визначити ідентифікатор пакета ID

```
long id = CAN.packetId();
```

Функція повертає ідентифікатор ID (11 або 29 біт) отриманого пакету. Стандартні пакети мають 11-бітний ідентифікатор, розширені пакети мають 29-бітний ідентифікатор.

Визначити прийом розширеного пакету можна за допомогою фукції

```
bool extended = CAN.packetExtended();
```

Функція повертає `true`, якщо отриманий пакет розширений, `false` інакше.

Аналогічну перевірку можна виконати для пакету типу RTR

```
bool rtr = CAN.packetRtr();
```

Функція повертає значення поля Remote Transmission Request (RTR) пакета `true`/`false`. Пакети RTR не містять даних, поле DLC містить запитувану довжину даних.

Довжина поля даних DLC визначається функцією

```
int DLC = CAN.packetDlc();
```

Наявність пакету даних в прийомному буфері можна за допомогою функції, яка повертає кількість байтів, доступних для читання.

```
int availableBytes = CAN.available()
```

Переглянути наступний байт та прочитати наступний байт у пакеті

```
int b = CAN.peek();
int b = CAN.read();
```

Також для зчитування байт даних можна використовувати потоковий інтерфейс.

Одним з необхідних механізмів роботи шини CAN, є фільтрація пакетів за ідентифікаторами. Налаштування фільтрування пакетів, що відповідають потрібним критеріям, виконується методами

```
CAN.filter(id);
CAN.filter(id, mask);
CAN.filterExtended(id);
CAN.filterExtended(id, mask);
```

id - 11-бітний ідентифікатор (стандартний пакет) або 29-бітний ідентифікатор пакета (розширений пакет)

mask - (необов'язково) 11-бітна маска (стандартний пакет) або 29-бітна маска (розширений пакет), за замовчуванням `0x7ff` або `0x1fffffff` (розширений)

Лише пакети, які відповідають наступним критеріям, підтверджуються та приймаються, інші пакети ігноруються:

```
if ((packetId & mask) == id) {
   // підтверджено та отримано
} else {
```

```
    // ігнорується
}
```

Бібліотека CAN підтримує додаткові режими роботи модуля.

Режим Loopback переводить контролер CAN у режим зворотного зв’язку, усі вихідні пакети також будуть отримані.

```
CAN.loopback();
```

Режим сну переведіть контролер CAN у сплячий режим.

```
CAN.sleep();
```

Вивід контролер CAN з режиму сну.

```
CAN.wakeup();
```

Для взаємодії за протоколом OBD2 бібліотека протоколу CAN використовується як шар нижнього рівня.

РОЗДІЛ 3
ОПИС ПРОГРАМИ МОНІТОРИНГУ

3.1. Інтерфейс бібліотеки для взаємодії за протоколом OBD2

Для обміну даними з авто використовується бібліотека OBD2 API з деякими змінами для підтримки специфічних запитів. Підключення файлу декларацій бібліотеки

```
#include <OBD2.h>
```

Робота з бібліотекою розпочинається з ініціалізаціїї бібліотеки, під час якої виконується спроба прочитати всі підтримувані PID автомобіля.

```
OBD2.begin();
```

Фукнція повертає «1» у разі успіху, «0» у разі невдачі. Припинення використання бібліотеки виконується наступним викликом

```
OBD2.end()
```

Під час ініціалізації об’єкту бібліотеки необхідно встановити час очікування відповіді

```
OBD2.setTimeout(timeout);
// timeout - нова відповідь у мілісекундах.
```

Явна перевірка того чи підтримує ECU PID, чи ні, виконується функцією

```
bool pidSupported = OBD2.pidSupported(pid);
// pid - PID для перевірки підтримки
```

Повертає true, якщо транспортний засіб підтримує pid, інакше false.

Якщо значення PID зчитується у необробленому вигляді

```
bool pidValueRaw = OBD2.pidValueRaw(pid);
// pid - PID для перевірки підтримки
```

Повертає true, якщо бібліотека НЕ підтримує перетворення значення pid, інакше false. Якщо pid використовує необроблений формат, то для його зчитування необхідно використовувати функцію `OBD2.pidReadRaw(pid)`, щоб прочитати необроблене значення. В іншому випадку неоьхідно використовувати функцію `OBD2.pidRead(pid)` для читання перетвореного значення як float.

Функція визначення назви PID повертає назву PID у вигляді рядка або "Unknown", якщо назва невідома.

```
String name = OBD2.pidName(pid);
// pid - PID для отримання імені
```

Для визначення одиниць вимірювання для PID використовується функція pidUnits(), яка повертає рядок.

```
String units = OBD2.pidUnits(pid);
// pid - PID для отримання одиниць
```

Функція зчитування перетвореного значення PID. Необроблене значення PID перетворюється на значення з плаваючою крапкою на основі формату даних PID. Функція повертає значення PID як float, а у випадку помилки - NAN.

```
float value = OBD2.pidRead(pid);
// pid - PID для отримання значення
```

Для PID у яких не задано спосіб перетворення можна зчитати необроблене значення PID. Функція повертає значення PID як unsigned long.

```
unsigned long value = OBD2.pidReadRaw (pid);
// pid - PID для отримання вихідного значення
```

Для зчитування VIN автомобіля використовується функція, яка повертає VIN автомобіля або порожній рядок у разі помилки.

```
String vin = OBD2.vinRead();
```

Для зчитування назви ECU використовується функція, яка повертає назву ECU або порожній рядок у разі помилки.

```
String ecu = OBD2.ecuNameRead();
```

Вище наведено загальний алгоритм зчитування даних з блоків ECU.

3.2. Програма моніторингу

Розглянемо структуру програми моніторингу стану електромобіля з модулем відображення результатів у вигляді Web-сторінки.

Програма реалізує точку доступа WiFi до якої можливо під’єднатися за допомогою будь-якого пристрою і у Web-браузері відкрити сторінку з результатами роботи програми.

Для взаємодії з пристроєм необхідно вказати SSID та пароль для точки доступу WiFi.

```
const char* ssid = ".......";
const char* password = ".......";
```

Створення об’єкту Web-сервераа

```
WebServer server(80);
```

Для зберігання даних, що зчитані з електромобіля використовується масив структур наступного виду

```
typedef struct
{
  int pid;
  float value;
  unsigned long value_raw;
} data_array_t;
```

Масив ініціалізується початковими значеннями номерів PID параметрів

```
data_array_t PIDS[] = {
  {CALCULATED_ENGINE_LOAD, 0, 0},
  {ENGINE_COOLANT_TEMPERATURE, 0, 0},
  {ENGINE_RPM, 0, 0},
  {VEHICLE_SPEED, 0, 0},
  {AIR_INTAKE_TEMPERATURE, 0, 0},
  {MAF_AIR_FLOW_RATE, 0, 0},
  {THROTTLE_POSITION, 0, 0},
  {RUN_TIME_SINCE_ENGINE_START, 0, 0},
  {FUEL_TANK_LEVEL_INPUT, 0, 0},
  {ABSOLULTE_BAROMETRIC_PRESSURE, 0, 0},
```

```
  {ABSOLUTE_LOAD_VALUE, 0, 0},
  {RELATIVE_THROTTLE_POSITION 0, 0}
};
const int NUM_PIDS = sizeof(PIDS) / sizeof(PIDS[0]);
```

Запит даних з електромобіля виконується функцією processPid, яка приймає індекс в масиві номерів PID

```
void processPid(int idx)
{
```

Виконується перевірка PID, що підримує електромобіль

```
  if (!OBD2.pidSupported(PIDS[idx].pid))
  {
    return;
  }
```

Для допустимих PID виконується запит даних або у вигляді шестнадцятирічних даних, або у вигляді дробного числа

```
  if (OBD2.pidValueRaw(PIDS[idx].pid))
  {
    PIDS[idx].value_raw = OBD2.pidReadRaw(PIDS[idx].pid);

    res = "0x"+String(pidRawValue, HEX);
  }
  else
  {
    PIDS[idx].value = OBD2.pidRead(PIDS[idx].pid);

  }
}
```

Для відображення даних використовується функція перетворення даних PID у рядок з урахуванням можливих помилок

```
String getPid(int idx)
{
  String res;
  if (!OBD2.pidSupported(PIDS[idx].pid))
  {
    return res;
  }

  if (OBD2.pidValueRaw(PIDS[idx].pid)) {
    res = "0x"+String(PIDS[idx].value_raw, HEX);
  } else {
    if (isnan(PIDS[idx].value)) {
      res = "error";
    } else {
      res = String(PIDS[idx].value);
    }  }  }
```

Формування даних для HTML-сторінки виконується з шаблонів. Шаблони зберігаються у програмній пам’яті для економії місця.

Шаблон початку HTML-сторінки

```
const char HTTP_HEAD[] PROGMEM = "<!DOCTYPE html><html><head><meta http-equiv=\"Content-Type\" content=\"text/html; charset=UTF-8\" name=\"viewport\" content=\"width=device-width, initial-scale=1, user-scalable=yes\"/>"
"</head><body>";
```

Шаблон заголовку таблиці з даними

```
const char HTTP_DATA_TABLE[] PROGMEM = "<table><tr><th>Param Name</th><th>Value</th><th>Unit</th></tr>";
```

Шаблон завершення таблиці з даними

```
const char HTTP_DATA_TABLE_END[] PROGMEM = "</table>";
```

Шаблон завершення HTML-сторінки

```
const char HTTP_END[] PROGMEM              = "</body></html>";
```

Функція-обробник для нормального запиту виконує створення сторінки і відправляє її до клієнта

```
void handleGET()
{
  String page;
  page += FPSTR(HTTP_HEAD);
  page += FPSTR(HTTP_DATA_TABLE);
  int i;
  for(i=0; i<NUM_PIDS; i++ )
  {
    page += "<tr><td>" +OBD2.pidName(PIDS[i].pid)+"</td>";
    page += "<td>"+ getPid(i) +"</td>";
    page += "<td>" +OBD2.pidUnits(PIDS[i].pid) +"</td></tr>";
  }

  page += FPSTR(HTTP_DATA_TABLE_END);
  page += FPSTR(HTTP_END);
  server.send(200, "text/html; charset=utf-8", page);
}
```

У разі помилкового запиту виконується відправка повідомлення про помилку

```
void handleNotFound()
{
  digitalWrite(led, 1);
  String message = "File Not Found\n\n";
  message += "URI: ";
  message += server.uri();
  message += "\nMethod: ";
  message += (server.method() == HTTP_GET) ? "GET" : "POST";
  message += "\nArguments: ";
  message += server.args();
  message += "\n";
  for (uint8_t i = 0; i < server.args(); i++) {
    message += " " + server.argName(i) + ": " + server.arg(i)
+ "\n";
  }
  server.send(404, "text/plain", message);
  digitalWrite(led, 0);
}
```

Для виконання запитів до ЕБУ електромобіля через інтервали 100мс та почерговий запит PID-параметрів використовується два лічильника

```
long count_time;
long count_pid_idx;
```

Функція початкової ініціалізації setup виконує налаштування обладнання та запускає на виконання Web-сервер.

```
void setup()
{
  count_time=0;
```

```
count_pid_idx=0;

Serial.begin(9600);
while (!Serial);
Serial.println(F("OBD2 Supported PIDs"));
Serial.println(F("OBD2 Key Stats"));
```

Створення з’єднання з ЕБУ електромобіля і очікування відповіді

```
while (true) {
  Serial.print(F("Attempting to connect to OBD2 CAN bus ...
"));
  if (!OBD2.begin()) {
    Serial.println(F("failed!"));
    delay(1000);
  } else {
    Serial.println(F("success"));
    break;
  }  }
```

Зчитування основної інформації та вивід її у послідовний порт для моніторингу

```
Serial.println();
Serial.print("VIN = ");
Serial.println(OBD2.vinRead());
Serial.print("ECU Name = ");
Serial.println(OBD2.ecuNameRead());
Serial.println();
```

Перевірка доступних PID-параметрів

```
for (int pid = 0; pid < 96; pid++) {
  if (OBD2.pidSupported(pid)) {
```

```
      Serial.println(OBD2.pidName(pid));
    }
  }
```

Створення WiFi точки доступу

```
  WiFi.mode(WIFI_AP);
  uint8_t mac[6/*WL_MAC_ADDR_LENGTH*/]={123,23,34,32,78,98};
  WiFi.softAPmacAddress(mac);
  WiFi.softAP(ssid, password);
  Serial.println("");
  Serial.print("IP address: ");
  Serial.println(WiFi.localIP());
```

Ініціалізація Web-серверу

```
  server.on("/", HTTP_GET, handleGET);
  server.on("/index.html", HTTP_GET, handleGET);
  server.onNotFound(handleNotFound);
```

Запуск Web-серверу

```
  server.begin();
  Serial.println("HTTP server started");
}
```

В головному циклі програми виконується обробка запитів від http-клієнтів та опитування PID-параметрів з ЕБУ електромобіля

```
void loop()
{
  server.handleClient();
  delay(2);
```

Опитування PID-параметрів виконується з неблокуючим очикуванням

```
  if(++count_time>=50)
  {
    count_time=0;
    processPid( count_pid_idx );
    if( ++count_pid_idx >= NUM_PIDS )  count_pid_idx=0;
  }
}
```

Розроблена програма дозволяє виконувати зчитування визначених параметрів та відображення їх у вигляді web-сторінки.

ВИСНОВКИ

Роль IoT в моніторингу систем електромобілів досить широке: моніторинг системи керування акумулятором, аналіз безпеки та розумне водіння, можна вимірювати абсолютні та відносні параметри якості керування, такі як швидкість, прискорення та інші, для надання підказок у реальному часі для забезпечення кращої продуктивності, система оповіщення про несправності та профілактичне обслуговування.

Максимальне використання ресурсів і ефективність електричної інфраструктури зі зниженими витратами є основними прагненнями для прийняття до використання електромобілів.

Розроблена система дозволяє виконати функції моніторингу. Розроблена система побудована на основі мікроконтролеру ESP32 з інтегрованими модулями Wi-Fi і Bluetooth, ядром якого є малопотужний процесор TSMC. Мікроконтролер ESP32 забезпечує високоінтегроване рішення із приблизно 20 зовнішніми компонентами. Для задач IoT мікроконтролер дозволяє використовувати кілька режимів живлення, і динамічне масштабування потужності.

Обмін даними з електромобілем виконується через шину CAN з використанням бібліотека OBD2 API з деякими змінами для підтримки специфічних запитів електромобіля. Розроблена програма реалізує точку доступа WiFi до якої можливо під’єднатися за допомогою будь-якого пристрою і у Web-браузері відкрити сторінку з результатами роботи програми.

ПЕРЕЛІК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Arduino Home page [Електронний ресурс]. – Режим доступу: www. URL: https://www.arduino.cc/ (дата звернення:10.02.23)
2. Arduino Libraries [Електронний ресурс]. – Режим доступу: www. URL: https://www.arduino.cc/reference/en/libraries/ (дата звернення:12.02.23)
3. Arduino core for the ESP32 [Електронний ресурс]. – Режим доступу: www. URL: https://github.com/espressif/arduino-esp32 (дата звернення:12.02.23)
4. CAN bus data loggers [Електронний ресурс]. – Режим доступу: www. URL: https://www.csselectronics.com/ (дата звернення:15.02.23)
5. OBD2 Explained - A Simple Intro [Електронний ресурс]. – Режим доступу: www. URL: https://www.csselectronics.com/pages/obd2-explained-simple-intro (дата звернення:16.02.23)
6. OBD-II Protocols [Електронний ресурс]. – Режим доступу: www. URL: http://www.obdtester.com/obd2_protocols (дата звернення:16.02.23)

ДОДАТКИ

ДОДАТОК А

Вихідний код програми

```
#include <WiFi.h>
#include <WiFiClient.h>
#include <WebServer.h>
#include <ESPmDNS.h>

const char* ssid = ".......";
const char* password = ".......";
WebServer server(80);
#include <CAN.h>
#include <OBD2.h>
typedef struct
{
  int pid;
  float value;
  unsigned long value_raw;
} data_array_t;
data_array_t PIDS[] = {
  {CALCULATED_ENGINE_LOAD, 0, 0},
  {ENGINE_COOLANT_TEMPERATURE, 0, 0},
  {ENGINE_RPM, 0, 0},
  {VEHICLE_SPEED, 0, 0},
  {AIR_INTAKE_TEMPERATURE, 0, 0},
  {MAF_AIR_FLOW_RATE, 0, 0},
  {THROTTLE_POSITION, 0, 0},
  {RUN_TIME_SINCE_ENGINE_START, 0, 0},
  {FUEL_TANK_LEVEL_INPUT, 0, 0},
  {ABSOLULTE_BAROMETRIC_PRESSURE, 0, 0},
  {ABSOLUTE_LOAD_VALUE, 0, 0},
  {RELATIVE_THROTTLE_POSITION 0, 0}
};
const int NUM_PIDS = sizeof(PIDS) / sizeof(PIDS[0]);
void processPid(int idx)
```

```
{
  if (!OBD2.pidSupported(PIDS[idx].pid))
  {    return;  }
  if (OBD2.pidValueRaw(PIDS[idx].pid))
  {
    PIDS[idx].value_raw = OBD2.pidReadRaw(PIDS[idx].pid);
    res = "0x"+String(pidRawValue, HEX);
  }
  else
  {
    // read the PID value
    PIDS[idx].value = OBD2.pidRead(PIDS[idx].pid);
  }
}
String getPid(int idx)
{
  String res;
  if (!OBD2.pidSupported(PIDS[idx].pid))
  {
    return res;
  }
  if (OBD2.pidValueRaw(PIDS[idx].pid)) {
    res = "0x"+String(PIDS[idx].value_raw, HEX);
  } else {
    if (isnan(PIDS[idx].value)) {
      res = "error";
    } else {
      res = String(PIDS[idx].value);
    }
  }
}
const char HTTP_HEAD[] PROGMEM = "<!DOCTYPE
html><html><head><meta http-equiv=\"Content-Type\"
content=\"text/html; charset=UTF-8\" name=\"viewport\"
content=\"width=device-width, initial-scale=1, user-
scalable=yes\"/>"
```

```
    //<script>setTimeout(function(){          location.reload();},
3000);</script>
    "</head><body>";
    const char HTTP_DATA_TABLE[] PROGMEM = "<table><tr><th>Param
Name</th><th>Value</th><th>Unit</th></tr>";
    const char HTTP_DATA_TABLE_END[] PROGMEM = "</table>";
    const char HTTP_END[] PROGMEM             = "</body></html>";
    void handleGET()
    {
      String page;
      page += FPSTR(HTTP_HEAD);
      page += FPSTR(HTTP_DATA_TABLE);
      int i;
      for(i=0; i<NUM_PIDS; i++ )
      {
        page += "<tr><td>" +OBD2.pidName(PIDS[i].pid)+"</td>";
        page += "<td>"+ getPid(i) +"</td>";
        page += "<td>" +OBD2.pidUnits(PIDS[i].pid) +"</td></tr>";
      }
      page += FPSTR(HTTP_DATA_TABLE_END);
      page += FPSTR(HTTP_END);
      server.send(200, "text/html; charset=utf-8", page);
    }
    void handleNotFound() {
      digitalWrite(led, 1);
      String message = "File Not Found\n\n";
      message += "URI: ";
      message += server.uri();
      message += "\nMethod: ";
      message += (server.method() == HTTP_GET) ? "GET" : "POST";
      message += "\nArguments: ";
      message += server.args();
      message += "\n";
      for (uint8_t i = 0; i < server.args(); i++) {
        message += " " + server.argName(i) + ": " + server.arg(i)
+ "\n";
```

```
    }
    server.send(404, "text/plain", message);
    digitalWrite(led, 0);
  }
  long count_time;
  long count_pid_idx;
  void setup()
  {
    count_time=0;
    count_pid_idx=0;
    Serial.begin(9600);
    while (!Serial);
    Serial.println(F("OBD2 Supported PIDs"));
    Serial.println(F("OBD2 Key Stats"));
    while (true) {
      Serial.print(F("Attempting to connect to OBD2 CAN bus ...
"));
      if (!OBD2.begin()) {
        Serial.println(F("failed!"));
        delay(1000);
      } else {
        Serial.println(F("success"));
        break;
      }
    }
    Serial.println();
    Serial.print("VIN = ");
    Serial.println(OBD2.vinRead());
    Serial.print("ECU Name = ");
    Serial.println(OBD2.ecuNameRead());
    Serial.println();
    Serial.println();
    // loop through PIDs 0 to 95, reading and printing the names
of the supported PIDs
    for (int pid = 0; pid < 96; pid++) {
      if (OBD2.pidSupported(pid)) {
```

```
      Serial.println(OBD2.pidName(pid));
    }
  }
  pinMode(led, OUTPUT);
  digitalWrite(led, 0);
  Serial.begin(115200);
  WiFi.mode(WIFI_STA);
  WiFi.begin(ssid, password);
  Serial.println("");
  // Wait for connection
  while (WiFi.status() != WL_CONNECTED) {
    delay(500);
    Serial.print(".");
  }
  Serial.println("");
  Serial.print("Connected to ");
  Serial.println(ssid);
  Serial.print("IP address: ");
  Serial.println(WiFi.localIP());
  server.on("/", HTTP_GET, handleGET);
  server.on("/index.html", HTTP_GET, handleGET);
  server.onNotFound(handleNotFound);
  server.begin();
  Serial.println("HTTP server started");
}
void loop()
{
  server.handleClient();
  delay(2);//allow the cpu to switch to other tasks
  if(++count_time>=50)
  {
    count_time=0;
    processPid( count_pid_idx );
    if( ++count_pid_idx >= NUM_PIDS )  count_pid_idx=0;
  }}
```