МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Західноукраїнський національний університет
Факультет комп'ютерних інформаційних технологій
Кафедра інформаційно-обчислювальних систем і управління

Гулько Юлія Віталіївна

Програмний модуль ядра Linux для нестандартного Arduino-сумісного сенсора / A Linux kernel software module for a non-standard Arduino-compatible sensor

Спеціальність 122 – Комп'ютерні науки
Освітньо-професійна програма – Комп'ютерні науки

Кваліфікаційна робота

Виконала студентка групи КН-41
Ю. В. Гулько

_____

Науковий керівник:
к.т.н., доцент, О.Р. Осолінський

_____

Кваліфікаційну роботу допущено до захисту
«___»_____ 2026 р.

Завідувач кафедри
_____ Н.В. Дзюбановська

Тернопіль – 2026

Факультет комп'ютерних інформаційних технологій
Кафедра інформаційно-обчислювальних систем і управління
Освітній ступінь «бакалавр»
спеціальність 122 – Комп'ютерні науки
освітньо-професійна програма – Комп'ютерні науки

ЗАТВЕРДЖУЮ
В.о. завідувача кафедри
_____ Н.В. Дзюбановська
«____» _____2025 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ СТУДЕНТЦІ
Гулько Юлії Віталіївні
(прізвище, ім'я, по батькові)

1. Тема кваліфікаційної роботи: Програмний модуль ядра Linux для нестандартного Arduino-сумісного сенсора / A Linux kernel software module for a non-standard Arduino-compatible sensor
керівник роботи О.Р. Осолінський, к.т.н., доцент
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)
затверджений наказом по університету від 01 грудня 2025 року № 894.
2. Строк подання студентом закінченої кваліфікаційної роботи 25 травня 2026 р.
3. Вихідні дані до кваліфікаційної роботи: завдання на кваліфікаційну роботу студента, наукові статті, технічна література.
4. Основні питання, які потрібно розробити:
- проаналізувати особливості програмної взаємодії Linux із зовнішніми сенсорами;
- провести огляд і аналіз існуючих підходів підтримки сенсорів в Linux;
- сформувати постановку задачі;
- розробити загальну структуру проектованого модуля;
- розробити алгоритми функціонування модуля ядра Linux;
- розробити інформаційне забезпечення модуля;
- реалізувати програмний модуль для ядра Linux;
- розробити засоби взаємодії з модулем у просторі користувача;
- провести тестування розробленого модуля.

5. Перелік графічного матеріалу в роботі:
- структурна карта файлів і компонентів програмної реалізації модуля;
- візуальна карта сценаріїв тестування
- схема консольного інтерфейсу користувача модуля;

6. Консультанти розділів кваліфікаційної роботи

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

7. Дата видачі завдання 01 грудня 2025 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Строк виконання етапів кваліфікаційної роботи	Примітка
1	Огляд літературних джерел відповідно до предметної області та складання плану роботи	до 01.01.2026 р.	
2	Написання 1 розділу кваліфікаційної роботи	до 01.02.2026 р.	
3	Написання 2 розділу кваліфікаційної роботи	до 15.03.2026 р.	
4	Написання 3 розділу кваліфікаційної роботи	до 01.05.2026 р.	
5	Представлення попереднього варіанту кваліфікаційної роботи, перевірка та внесення змін керівником	до 15.05.2026 р.	
6	Опрацювання зауважень та представлення завершеного варіанту кваліфікаційної роботи. Підготовка супроводжуючих документів	до 25.05.2026 р.	
7	Перевірка кваліфікаційної роботи на оригінальність тексту	до 30.05.2026 р.	
8	Оформлення кваліфікаційної роботи та отримання допуску до захисту	до 10.06.2026 р.	
9	Подання кваліфікаційної роботи до захисту на засіданні атестаційної комісії	до 10.06.2026 р.	

Студент_____ Ю. В. Гулько_____
(підпис) (прізвище та ініціали)

Керівник кваліфікаційної роботи_____ О.Р. Осолінський_____
(підпис) (прізвище та ініціали)

АНОТАЦІЯ

Пояснювальна записка до кваліфікаційної роботи: 72 с., 34 рис., 1 таблиця, 4 додатки, 29 джерел.

Метою кваліфікаційної роботи є розробка програмного модуля ядра Linux для нестандартного Arduino-сумісного сенсора, який працює на платформі Raspberry Pi та забезпечує приймання, обробку, нормалізацію і системне подання даних засобами Linux.

В роботі проведено аналіз драйверів ядра Linux для вбудованих сенсорів, розглянуто особливості програмної взаємодії Linux із зовнішніми сенсорами та досліджено існуючі підходи до підтримки сенсорів у Linux, зокрема sysfs, udev, Industrial I/O та Hardware Monitoring. Сформульовано постановку задачі, визначено основні вимоги до програмного модуля та обґрунтовано доцільність створення власного модуля ядра для нестандартного Arduino-сумісного пристрою.

Розроблено загальну структуру проектованого модуля, описано алгоритми його функціонування та інформаційне забезпечення. Визначено склад вхідних і вихідних даних, службові параметри, логіку подання метрик через sysfs, правила взаємодії з udev, а також засоби журналювання й діагностики. Практична реалізація виконана мовою C у вигляді модуля ядра Linux з підтримкою sysfs-атрибутів value, raw, status, error_code, last_update та inject_raw. Додатково реалізовано правило udev, shell-скрипти для читання метрик і тестування, а також консольний інтерфейс користувача.

Проведено тестування розробленого модуля в середовищі Debian Linux. Перевірено складання, завантаження і вивантаження модуля, створення sysfs-атрибутів, коректність оновлення метрик при допустимих вхідних даних, обробку помилок виходу за діапазон і помилок розбору, спрацювання udev та роботу користувацького консольного інтерфейсу.

Ключові слова: LINUX, ЯДРО LINUX, RASPBERRY PI, ARDUINO, СУМІСНИЙ СЕНСОР, ДРАЙВЕР, ПРОГРАМНИЙ МОДУЛЬ, SYSFS, UDEV, ВБУДОВАНА СИСТЕМА, МОНІТОРИНГ.

ANNOTATION

Explanatory note to the qualification thesis: 72 pages, 34 figures, 1 table, 4 annexes, 29 references.

The purpose of the qualification thesis is to develop a Linux kernel software module for a non-standard Arduino-compatible sensor operating on the Raspberry Pi platform and providing data acquisition, processing, normalization, and system-level representation by means of Linux.

The thesis analyzes Linux kernel drivers for embedded sensors, examines the specific features of software interaction between Linux and external sensors, and studies existing approaches to sensor support in Linux, including sysfs, udev, Industrial I/O, and Hardware Monitoring. The problem statement is formulated, the main requirements for the software module are defined, and the feasibility of creating a dedicated kernel module for a non-standard Arduino-compatible device is substantiated.

The general structure of the designed module has been developed, and its operating algorithms and information support have been described. The composition of input and output data, service parameters, the logic of presenting metrics through sysfs, the rules of interaction with udev, as well as logging and diagnostic tools have been defined. The practical implementation was carried out in the C language in the form of a Linux kernel module supporting the sysfs attributes value, raw, status, error_code, last_update, and inject_raw. In addition, a udev rule, shell scripts for metric reading and testing, and a console user interface were implemented.

The developed module was tested in the Debian Linux environment. The following were verified: module compilation, loading and unloading, creation of sysfs attributes, correctness of metric updates for valid input data, handling of out-of-range errors and parsing errors, udev triggering, and the operation of the console user interface.

Keywords: LINUX, LINUX KERNEL, RASPBERRY PI, ARDUINO, COMPATIBLE SENSOR, DRIVER, SOFTWARE MODULE, SYSFS, UDEV, EMBEDDED SYSTEM, MONITORING.

ЗМІСТ

ВСТУП

В сучасних вбудованих системах використовуються одноплатні комп’ютери, мікроконтролерні вузли та зовнішні сенсори, які забезпечують збір, передавання та обробку даних в реальному часі. Такі рішення застосовуються у вимірювальних системах, локальних системах моніторингу, експериментальних стендах, автоматизації та навчальних проектах. Однією з найпоширеніших платформ для побудови подібних систем є Raspberry Pi, так як вона поєднує відносну доступність, підтримку операційної системи Linux, наявність стандартних цифрових інтерфейсів та можливість одночасного використання системного і прикладного програмного забезпечення.

Але розробники стикаються з проблемою, тому що ефективна робота зовнішнього сенсора в середовищі Linux залежить не тільки від фізичного підключення, а і від правильної програмної інтеграції на рівні операційної системи. Якщо пристрій належить до поширених типів обладнання, для нього часто існують готові драйвери ядра та стандартні механізми доступу. Але на практиці дуже часто також використовуються нестандартні Arduino-сумісні сенсори або проміжні мікроконтролерні вузли, які мають власний формат даних, особливу логіку обміну, нетипову ініціалізацію або нестандартний набір службових параметрів.

Актуальність теми полягає в створенні програмного модуля ядра Linux, який забезпечить повноцінну підтримку нестандартного Arduino-сумісного сенсора на платформі Raspberry Pi. Такий модуль повинен крім того, що зчитувати сирі дані, виконувати ідентифікацію пристрою, перевірку коректності повідомлень, перетворення сирих значень в системні метрики, експорт параметрів через sysfs, взаємодію із системними подіями через udev, а також підтримувати журналювання і засоби діагностики. Реалізація такого підходу дозволяє зробити сенсор повноцінною частиною Linux-системи.

Крім того стан сучасних вбудованих рішень показує, що Linux є гнучким середовищем для роботи з сенсорами, але стандартні підсистеми найбільш ефективні тоді, коли пристрій відповідає типовим моделям інтеграції. Для нестандартних модулів часто виникає ситуація, коли готова підтримка або відсутня, або не покриває всіх особливостей роботи конкретного сенсора. Тому виникає потреба у власному модулі ядра, який упорядковує обмін із пристроєм,

подає метрики в єдиному форматі та забезпечує стабільний доступ до них для користувацьких програм і системних сервісів.

Подібні модулі можуть використовуватися не тільки в експериментальних або навчальних роботах, а і у прикладних системах моніторингу, локального керування та автоматизації.

Метою кваліфікаційної роботи є розробка програмного модуля ядра Linux для нестандартного Arduino-сумісного сенсора, який працює на платформі Raspberry Pi та забезпечує приймання, обробку, нормалізацію і системне подання даних засобами Linux.

Для досягнення поставленої мети необхідно вирішити такі завдання:

1. Провести аналіз предметної області драйверів ядра Linux для вбудованих сенсорів.

2. Розглянути особливості програмної взаємодії Linux з зовнішніми сенсорами.

3. Проаналізувати існуючі підходи до підтримки сенсорів в Linux, зокрема sysfs, udev і типові драйверні механізми.

4. Сформулювати постановку задачі та вимоги до програмного модуля.

5. Розробити загальну структуру модуля та визначити його місце у системі Raspberry Pi.

6. Описати алгоритм функціонування модуля ядра Linux.

7. Розробити інформаційне забезпечення модуля, включаючи модель даних, структуру атрибутів sysfs, взаємодію з udev та засоби діагностики.

8. Здійснити програмну реалізацію та тестування працездатності модуля.

Об’єктом дослідження є процеси програмної взаємодії операційної системи Linux із зовнішніми сенсорними пристроями у вбудованих системах на базі Raspberry Pi.

Предметом дослідження є методи, алгоритми та програмно-технологічні засоби побудови програмного модуля ядра Linux, а також способи подання метрик через sysfs, формування подій через udev і організації доступу до них у просторі користувача.

Результати кваліфікаційної роботи апробовані та опубліковані у матеріалах IX Всеукраїнської студентської наукової конференції «Науковий простір: аналіз, сучасний стан, тренди та перспективи» (м. Київ, Україна).

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ДРАЙВЕРІВ ЯДРА LINUX ДЛЯ ВБУДОВАНИХ СЕНСОРІВ

1.1 Особливості програмної взаємодії Linux із зовнішніми сенсорами

У вбудованих системах на базі Linux робота з зовнішніми сенсорами будується не тільки на рівні апаратного підключення, а і на рівні програмної організації доступу до пристрою. Через це драйвер ядра є одним із ключових елементів такої системи. Він зв’язує конкретний сенсор із програмним середовищем Linux. Драйвер приймає дані від фізичного пристрою, виконує початкову обробку, контролює доступ до апаратних ресурсів, а також надає стандартний інтерфейс для інших частин системи. В Linux для підсистеми Industrial I/O зазначено, що ядро надає єдиний каркас для підтримки багатьох типів вбудованих сенсорів і водночас формує стандартний інтерфейс для доступу з боку прикладних програм [1, 2]. Отже, драйвер зчитує байти з шини та вводить сенсор у загальну модель пристроїв Linux.

Для зовнішніх сенсорів така роль драйвера важлива, тому що один і той самий клас сенсорів може мати різні фізичні інтерфейси, різні формати регістрів і різні режими роботи. На практиці датчики температури, тиску, освітленості, відстані або прискорення часто підключаються через I2C або SPI [1]. В Linux типовий драйвер для такого пристрою реєструється як I2C або SPI драйвер, виконує ініціалізацію під час підключення пристрою, а потім відкриває доступ до значень і параметрів через системні інтерфейси [2, 3]. Якщо сенсор нестандартний або Arduino-сумісний, то його підтримка часто не зводиться до використання готової бібліотеки. У такому випадку доводиться створювати власний модуль ядра, який враховує особливості протоколу обміну, формат сирих значень, порядок ініціалізації, а також вимоги до стабільності роботи на цільовій платформі.

Розуміння ролі драйвера пов’язане з поділом Linux на простір ядра і простір користувача. Простір ядра відповідає за безпосередню роботу з апаратурою, обробку переривань, доступ до шин, синхронізацію і керування системними ресурсами. Простір користувача, навпаки, працює через вже готові інтерфейси і не

має прямого доступу до критичних механізмів ядра. На рівні практики це означає, що програма користувача може читати готові значення із sysfs, працювати з вузлами в каталозі dev або взаємодіяти з даними через бібліотеки, але не повинна заміняти собою повноцінний драйвер там, де для цього вже існує відповідний механізм ядра [2, 4, 5]. В GPIO character device API наголошено, що не слід зловживати userspace API для керування апаратурою, якщо для цього існують правильні драйвери ядра [4].

Для Raspberry Pi вибір між реалізацією в просторі ядра та в просторі користувача має не тільки архітектурне, а і практичне значення. У дослідженні [7], де порівнювали user space і kernel space під час збору даних з сенсора на Raspberry Pi, було показано, що реалізація на рівні користувацьких скриптів зручна для швидкого прототипування, однак може створювати додаткове навантаження на систему коли дані зчитуються часто і паралельно зберігаються або передаються далі. Варіант з кодом на рівні ядра виявився стабільнішим щодо частоти вибірок і краще підходив для безперервного збору даних. В описі SPI userspace API також вказується, що простір користувача підходить для прототипування і простих протоколів, але існують задачі, які неможливо коректно реалізувати без доступу до можливостей ядра, наприклад до обробників переривань або внутрішніх шарів драйверної підсистеми [5]. Тому при побудові модуля для нестандартного Arduino-сумісного сенсора доцільно розглядати user space як початковий етап перевірки ідей, а kernel space як основний шлях для повноцінної реалізації.

Окреме місце займає Raspberry Pi як цільова платформа. Вона стала популярною через невисоку вартість та поєднання Linux-середовища з доступом до фізичних інтерфейсів плати. В описі Raspberry Pi відмічено, що GPIO лінії можуть працювати як звичайні входи і виходи, а також як альтернативні функції для SPI, I2C і Serial [6]. Для практичної розробки це зручно, томущо одна плата може одночасно виконувати роль вузла збору даних і платформи для запуску власного драйвера. Крім того Raspberry Pi має низку обмежень, які прямо впливають на роботу з сенсорами. В дослідженні [7] про збір даних на Raspberry Pi підкреслено, що сильними сторонами плати є стек Linux, підтримка сучасного ядра і гнучкість у

підключенні периферії, але серед слабких сторін названо відсутність вбудованого АЦП, обмеження з боку SD-карти і неорієнтованість плати на жорсткі вимоги промислових систем.

Однією з найважливіших особливостей Raspberry Pi є те, що її GPIO працюють на рівні 3.3 В, а не 5 В [6]. Для Arduino-сумісних сенсорів це важливо, оскільки частина модулів допускає живлення і логіку 5 В. Якщо підключити такий модуль без узгодження рівнів, це може призвести до некоректної роботи або навіть пошкодження плати. Ще одна особливість Raspberry Pi — відсутність аналогових входів. Це не дозволяє напряму працювати з багатьма простими сенсорами, які видають аналоговий сигнал. В декількох роботах відзначено, що для такої плати необхідно або використовувати цифрові сенсори, або додавати зовнішній АЦП, або залучати окремий мікроконтролер як проміжну ланку [8, 9, 13]. В роботі [8] для системи на Raspberry Pi саме з цієї причини був обраний цифровий датчик DS18B20 (рисунок 1.1) з інтерфейсом 1 Wire, а не аналоговий температурний сенсор.

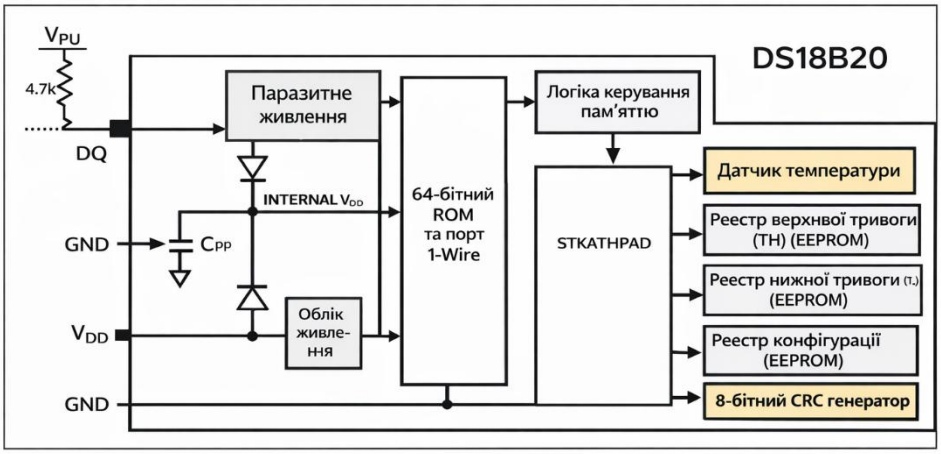


Рисунок 1.1 - Блок-схема датчика DS18B20

Способи підключення Arduino-сумісних сенсорів до Linux-платформи можуть бути різними. Найпростіший варіант полягає у використанні цифрових сенсорів, які Raspberry Pi може обслуговувати напряму через I2C, SPI або 1 Wire. Це зменшує кількість проміжних вузлів і спрощує програмну частину. Наприклад, у випадку DS18B20 сенсор має унікальний ідентифікатор, підтримує 1 Wire і добре пристосований до підключення кількох пристроїв на одній шині [8].

Інший варіант пов’язаний з використанням зовнішнього мікроконтролера або Arduino-сумісної плати, як фронтального вузла збору даних. В даному випадку мікроконтролер опитує сенсор, виконує базову обробку, а Raspberry Pi приймає вже підготовлені значення через UART, USB serial, TCP або MQTT. В роботі [15] показано практичний підхід до роботи з послідовним портом, де Raspberry Pi використовує шлях dev ttyACM0 для обміну з підключеним модулем. В Іншій роботі [11] ESP32 разом із LiDAR-сенсором збирає дані, а Raspberry Pi приймає їх по TCP через Wi Fi і зберігає на серверній частині. Такий варіант доцільний, коли сенсор передає дані у нетиповому або власному форматі.

Для Arduino-сумісних сенсорів також характерний сценарій, коли кілька цифрових датчиків підключаються до одного мікроконтролера по спільній шині, а Linux-вузол виконує роль головного пристрою або центрального обробника. В роботі [10] усі сенсори були цифровими і працювали через спільну шину I2C, а Raspberry Pi виступав головним вузлом, який отримував дані з хмари (рисунок 1.2), зберігав їх локально, перевіряв часові мітки і усереднював значення перед подальшою передачею.

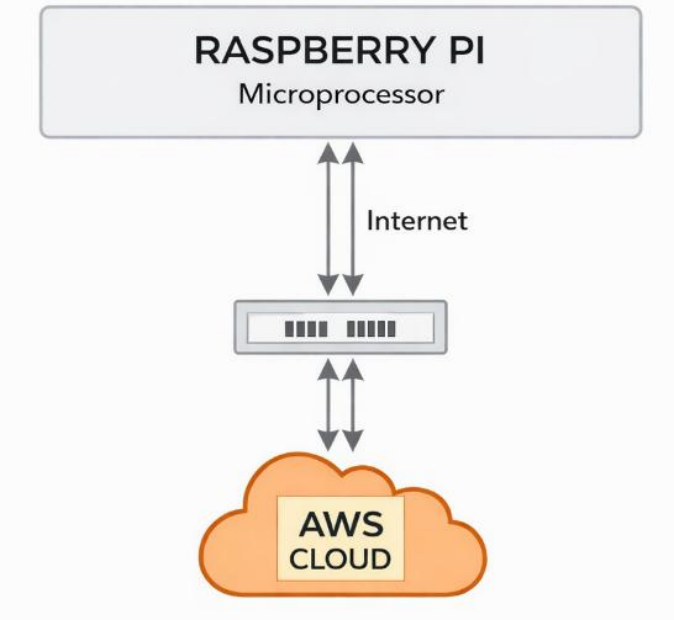


Рисунок 1.2 - Схема головного пристрою на базі Raspberry Pi

Такий приклад підтверджує, що Linux-платформа може виконувати як низькорівневе читання регістрів, так і повний цикл збору та агрегації даних. Це важливо, оскільки навіть простий на вигляд нестандартний сенсор в системі може бути частиною складнішого ланцюга обробки даних, де необхідно враховувати синхронізацію, форматування, локальне зберігання і експорт даних для інших компонентів.

Ще один сценарій який використовується для аналогових або напівпромислових сенсорів, коли Raspberry Pi працює не безпосередньо з датчиком, а через допоміжні модулі. В роботі [9] показано, що через відсутність аналогових входів на Raspberry Pi для вимірювання струму потрібно використовувати зовнішній АЦП на базі SPI, а для обміну з вимірювальним модулем по Modbus потрібно додатковий RS485 трансивер (рисунок 1.3), підключений до UART плати.

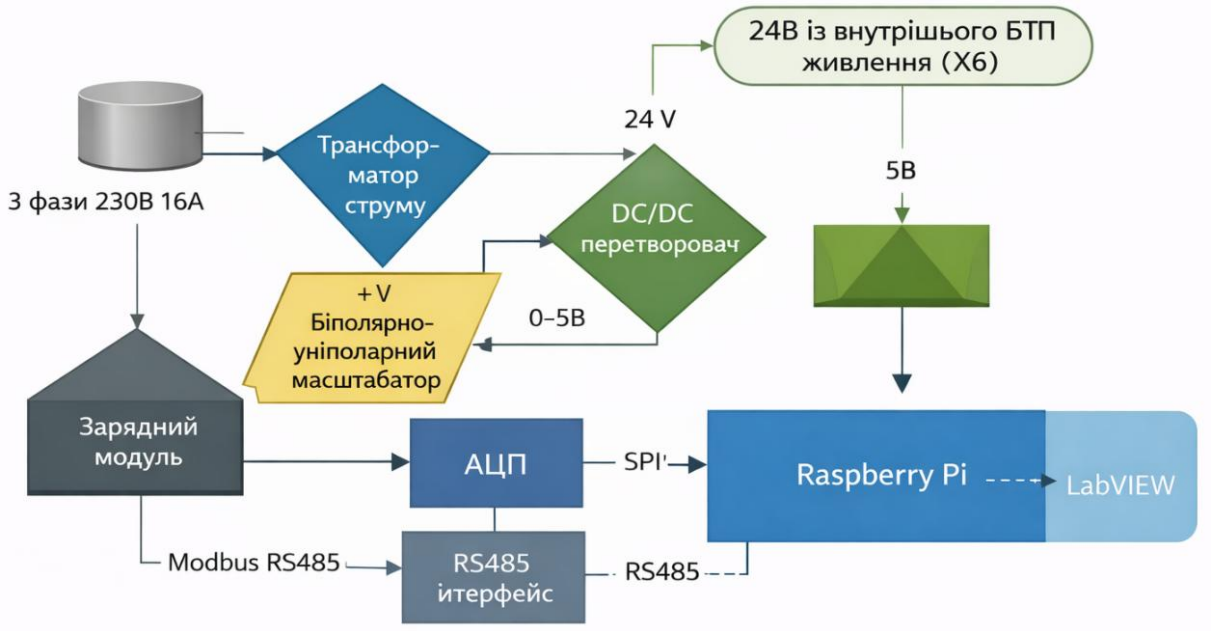


Рисунок 1.3 - Апаратна блок-схема вузла збору даних

Цей приклад демонструє, що під час проектування системи збору даних необхідно окремо враховувати три аспекти: як фізично зчитати сигнал, як передати його в Linux і як привести значення до форми, придатної для подальшої обробки. В ряді випадків саме модуль ядра бере на себе частину цієї роботи, тому що він

може приховати від простору користувача особливості конкретного протоколу та дати стабільний уніфікований інтерфейс.

В Linux робота з сенсором не завершується простим отриманням значення. Після читання сирих даних треба виконати нормалізацію - перетворення первинного коду, що надходить з регістра, АЦП або буфера, у фізично зрозумілу величину. У підсистемі I/O це розділення прямо закладено в модель пристрою, де для сенсорів можуть існувати окремі атрибути для сирих значень і для вже оброблених величин [2]. Тобто система може окремо зберігати raw значення і окремо подавати processed або input значення, які вже враховують масштаб, зсув або інші параметри перерахунку. В роботі [9] також показано, що в реальних системах перед нормалізацією потрібно навіть зміщувати сигнал до допустимого діапазону АЦП, а вже після цього виконувати дискретизацію і масштабування.

Після нормалізації дані потрібно передати далі в придатному для системи вигляді. В Linux це може бути організовано декількома способами. Найбільш поширений підхід для сенсорного драйвера полягає в створенні атрибутів sysfs, через які можна прочитати поточні значення, параметри дискретизації або службовий стан пристрою [2]. Якщо пристрій потребує потокової передачі або роботи з подіями, можуть використовуватися символьні вузли в каталозі dev [2], [5]. В документації для spidev зазначено, що після прив’язки драйвера з’являються вузли, які автоматично створюються і обслуговуються через udev або сумісні засоби [5]. Експорт метрик через sysfs і взаємодія з udev фактично є частиною функціональних вимог до модуля. Це робить сенсор видимим для всієї системи, а не лише для однієї прикладної програми.

На рівні прикладної логіки передача даних включає часове позначення, буферизацію, усереднення і пересилання далі в локальне сховище або зовнішню систему. В роботі [10] показано, що після отримання даних вони зберігаються і перевіряється часова близькість вимірювань. В роботі [11] дані від ESP32 потрапляли на Raspberry Pi сервер, де зберігалися в базі даних разом з часовими мітками. В роботі [12] Raspberry Pi виступав хостом для системи, яка одночасно працювала з I2C, SPI і RS232 сенсорами, а далі передавала результати через веб

інтерфейс, API або MQTT. Отже, в сучасній вбудованій системі важливо не лише зчитати показник сенсора, а і побудувати зрозумілий шлях даних від фізичного вимірювання до користувача або сервісу. Тому при проектуванні модуля ядра потрібно враховувати не лише драйверну частину, а і спосіб подання метрик назовні.

Отже, програмна взаємодія Linux із зовнішніми сенсорами спирається на чіткий розподіл ролей між апаратним рівнем, драйвером ядра і прикладними компонентами. Драйвер ядра виконує низькорівневу інтеграцію пристрою в систему, простір користувача отримує вже стандартизований доступ до даних, а Raspberry Pi виступає платформою, яка має набір цифрових інтерфейсів і водночас не має вбудованого АЦП.

1.2 Огляд і аналіз існуючих підходів підтримки сенсорів в Linux

Підтримка сенсорів у Linux реалізована як багаторівнева система, в якій кожен рівень виконує окрему функцію. На нижньому рівні система має шини та контролери, через які пристрій фізично підключається до плати. Найчастіше для вбудованих сенсорів це I2C, SPI, 1-Wire, GPIO або послідовний інтерфейс. Вище працює модель пристроїв ядра, яка відповідає за реєстрацію пристрою, пошук відповідного драйвера і зв’язування апаратного вузла з програмною логікою. Ще вище розташовані підсистеми, які дають уніфікований спосіб роботи саме з класом сенсорів. Для вимірювальних пристроїв в Linux найбільш важливими є Industrial I O та Hardware Monitoring. Завдяки такому поділу особливості конкретної шини не змішуються з логікою доступу до даних сенсора [16, 17, 18].

Тому драйверу не потрібно щоразу створювати окрему схему взаємодії з користувацькими програмами. Якщо сенсор вписується в стандартну модель Linux, драйвер реєструється в потрібній підсистемі та одразу отримує узгоджений спосіб експорту атрибутів, параметрів і значень. Драйвери в Linux є рівнем абстракції (рисунок 1.4) між програмним забезпеченням та апаратурою.

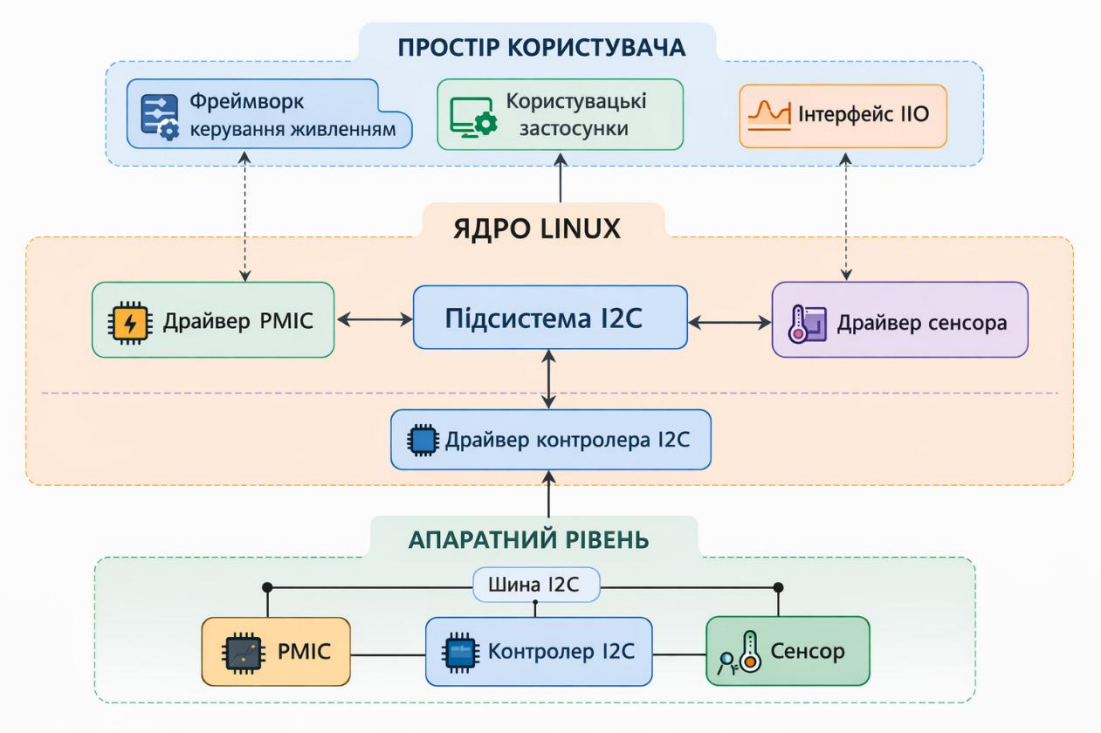


Рисунок 1.4 – Взаємодія апаратури, драйвера та підсистем ядра Linux

Вони повинні підлаштуватися під внутрішній інтерфейс певної підсистеми, а вже ця підсистема забезпечує зрозумілу модель доступу для інших частин системи [19]. Ядро дає єдиний framework для написання драйверів багатьох типів сенсорів і стандартний інтерфейс до user space [16].

Для сенсорів, що видають числові вимірювання, в Linux існують два найбільш поширені підходи. Перший пов’язаний з Hardware Monitoring. Він орієнтований на температуру, напругу, струм, потужність, швидкість обертання вентиляторів та подібні величини. Він пропонує просту та передбачувану схему імен атрибутів і дозволяє користувацьким програмам працювати з різними чипами майже однаково. Бібліотека libsensors працює через sysfs і вважає драйвер чип-незалежним саме тоді, коли драйвер дотримується стандартного sysfs-інтерфейсу [20].

Другий підхід пов’язаний з Industrial I/O. Ця підсистема більше орієнтована на датчики, де крім простого поточного значення потрібні канали, буфери, частота дискретизації, події, часові мітки або робота з потоками вибірок. IIO device відповідає одному апаратному сенсору і має набір sysfs-атрибутів для конфігурації та зчитування, а також може надавати буферний інтерфейс через вузол у каталозі

dev [16]. Це більш гнучка модель, ніж класичний hwmon, і вона часто використовується для акселерометрів, гіроскопів, АЦП, датчиків освітленості чи комбінованих модулів.

Механізми підтримки сенсорів у Linux покривають значну частину типових задач. Якщо є датчик на I2C або SPI з відомим набором регістрів, під нього можна знайти готовий mainline драйвер. В [21] зазначено, що Linux drivers для MEMS і environmental sensors побудовані на IIO framework, підтримують SPI та I2C і доступні як відкритий код для кількох версій ядра.

В цій моделі важливу роль відіграє sysfs. Це файловий інтерфейс, через який ядро показує користувачеві параметри пристрою. Для сенсорів він зручний тим, що дає простий доступ до поточних значень, коефіцієнтів масштабування, частоти опитування, меж тривоги та інших атрибутів. В [20] представлено, що всі sysfs-значення подаються у форматі fixed point, а сам інтерфейс має стандарт іменування, який спрощує роботу універсальних програм. В [17] вказано, що під час реєстрації пристрою через hwmon_device_register_with_info стандартні sysfs-атрибути створюються самим ядром. Вручну додаються переважно лише нестандартні атрибути, якщо вони потрібні.

Такий підхід має кілька переваг:

– sysfs робить пристрій видимим для всієї системи, а не тільки для однієї програми;

– добре працює для конфігураційних параметрів і для зчитування окремих значень;

– через sysfs будується багато стандартних інструментів Linux, що спрощує перевірку драйвера на ранніх етапах.

В роботі [22] зазначено, що взаємодія з сенсорами демонструється через sysfs, а підтримка I2C датчиків реалізується за допомогою Linux kernel drivers та devicetree overlays. Це показує, що sysfs є стандартним інструментом в реальних вбудованих системах.

Водночас sysfs не покриває всі можливі сценарії роботи з сенсором. Він добре підходить для атрибутів і керуючих параметрів, але менш зручний там, де потрібен

швидкий потік даних або складніший обмін. Саме тому IIO окрім sysfs використовує ще і buffer devices у каталозі dev [16].

Окреме місце займає udev. Драйвер працює в ядрі і спілкується з пристроєм, а udev працює на рівні user space і реагує на події, які надсилає ядро. В [23] представлено, що udev постачає системі software device events, керує дозволами для device nodes і може створювати додаткові symlink у каталозі dev. Це корисно, коли пристрій підключається як USB Serial або інший вузол, ім'я якого може змінюватися після перезавантаження чи перепідключення. Ядро може видати ttyACM0 або ttyUSB0 залежно від порядку виявлення, а udev допомагає дати такому пристрою стабільне ім'я за певною ознакою.

Для сенсорних систем це потрібно, бо якщо Arduino-сумісний модуль підключається через USB і в системі з'являється як послідовний порт, то udev може автоматично визначити його за vendor id, product id або іншими властивостями, створити стабільне символьне ім'я і запустити додаткову дію. Але сам по собі udev не нормалізує дані сенсора і не створює модель каналів чи атрибутів. Він тільки допомагає системі правильно відреагувати на появу або зникнення пристрою. В документації device model [18] також підкреслюється, що атрибути повинні бути створені до генерації KOBJ_ADD uevent. Якщо додати їх пізніше, userspace може не ідентифікувати їх.

Типові драйверні підходи для вбудованих сенсорів у Linux можна умовно поділити на кілька сценаріїв. Найпростіший сценарій виникає тоді, коли сенсор вже підтримується ядром. В такому випадку достатньо правильно описати його у Device Tree або підключити на відповідну шину, після чого драйвер сам зв'яжеться з пристроєм через механізм probe. В роботі [19] показано, що драйвери реєструються в bus subsystem, а після виявлення сумісного пристрою викликається probe, де і відбувається прив'язка та ініціалізація. В [24] цей підхід показано на прикладі ADXL313, де наведено приклад вузла Device Tree (рисунок 1.5) для цифрового акселерометра на SPI. Такий шлях є найбільш оптимальним для I2C і SPI сенсорів.

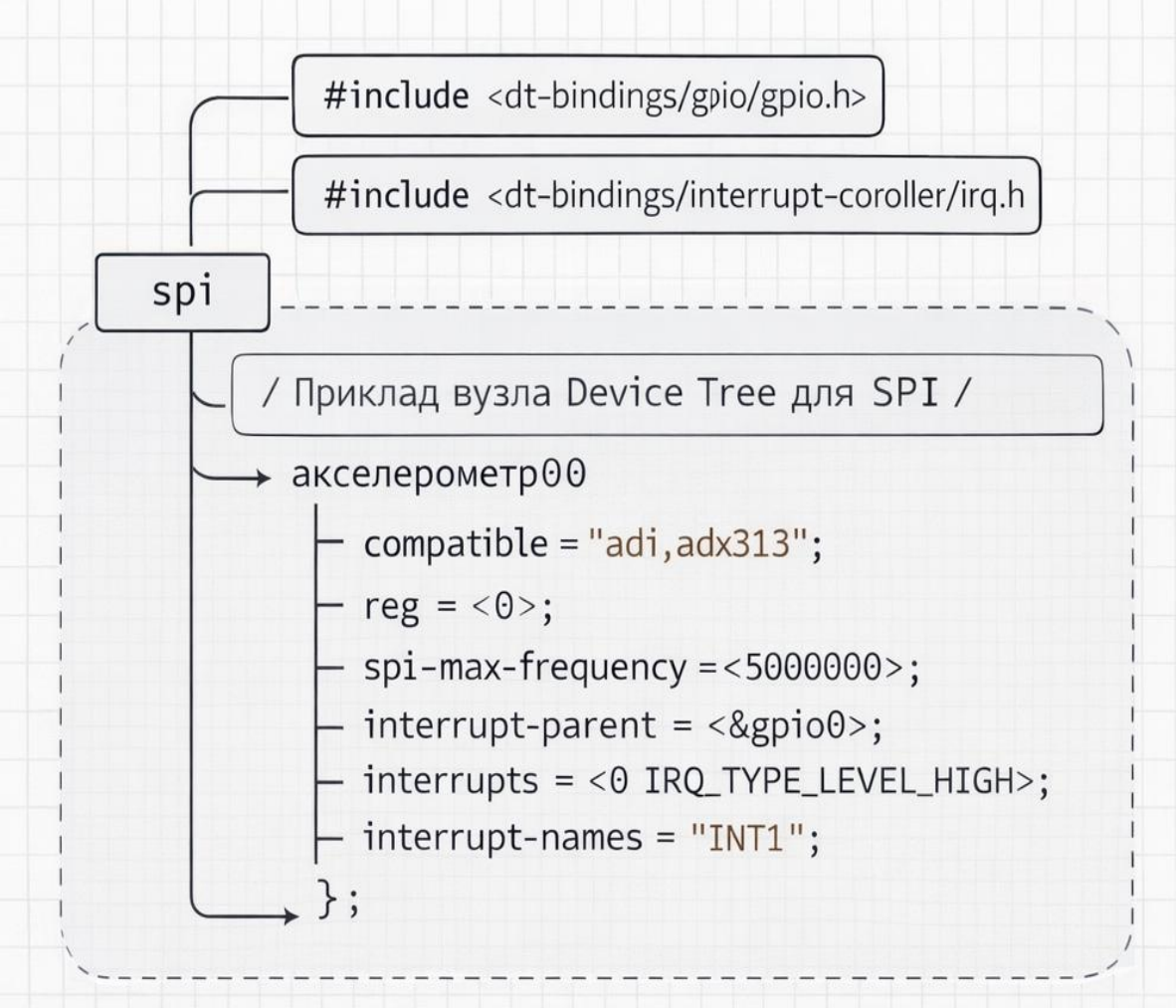


Рисунок 1.5 – Приклад опису цифрового сенсора у Device Tree для SPI

Інший типовий підхід полягає у використанні platform driver. Він більш характерний для периферії, яка не приходить через зовнішню стандартну шину, а є частиною самої платформи або описується на рівні апаратних ресурсів системи. В статті [25] зазначається, що platform driver architecture має сильну незалежність ресурсів пристрою та драйвера, повторно використовує framework code і забезпечує кращу підтримуваність та розширюваність. Для зовнішнього Arduino-сумісного сенсора цей підхід важливий тоді, коли нестандартний пристрій інтегрується глибше в платформу або використовує спеціальні ресурси SoC.

Ще один поширений варіант стосується готових стандартних протоколів, для яких підтримка вже є в ядрі. В роботі [8] та на рисунку 1.6 показано, що для нього достатньо увімкнути 1-Wire, завантажити модулі w1_therm і w1-gpio та читати дані через шлях /sys/bus/w1/devices.

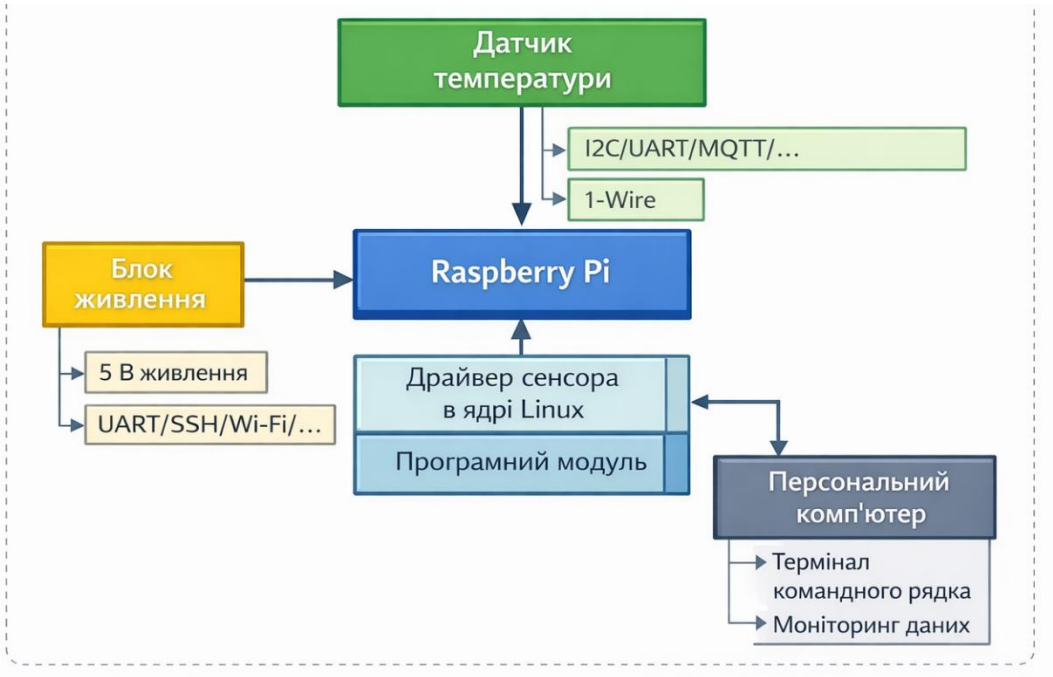


Рисунок 1.6 – Структурна схема системи на Raspberry Pi

З цього видно, що для сенсора з поширеним протоколом і готовою підтримкою в ядрі інтеграція є відносно простою та надійною. Але вона працює лише тоді, коли пристрій добре описаний в існуючій моделі.

У випадку нестандартного Arduino-сумісного сенсора ситуація зазвичай складніша. Такий модуль нерідко віддає дані не як класичний I2C або SPI чип з реєстрами, а як набір пакетів через UART або USB CDC. Іноді там використовується власний текстовий або бінарний протокол, службові префікси, контрольні суми, несталі поля чи асинхронні повідомлення. В такому випадку стандартний hwmon або IIO не може бути використаний без додаткового шару адаптації. Сам по собі sysfs не опрацьовує користувацький протокол. udev вміє знайти пристрій і дати йому стабільне ім’я, але не перетворює пакетні дані на стандартизовані метрики. Саме тому для таких сенсорів часто з’являється потреба у власному модулі ядра або в окремому проміжному драйвері.

Такий приклад цього обмеження наведено в роботі [26] де описано, що в Linux kernel не було підтримки INA260, тому для повного ланцюга від сенсора до кінцевого користувача було розроблено власний Linux driver, який експортує регістри та поля INA260 через sysfs Virtual File System.

Практика показує, що навколо стандартних підсистем уже сформувалося і стандартне userspace середовище. Для IIO існує libiio, яка спрощує розробку програм для роботи з Industrial IO devices і приховує частину низькорівневих деталей [27]. Для налагодження доступні iio_attr, iio_info, iio_readdev, iio_writedev і iio_reg, які дозволяють читати атрибути, працювати з buffer device і напряму звертатися до SPI або I2C регістрів для debugging [28]. Перевага стандартного підходу полягає в тому, що після коректної інтеграції сенсора в IIO або hwmon розробник отримує драйвер та готові інструменти для тестування і експлуатації.

Отже, в Linux існує повноцінна система підтримки сенсорів, яка базується на bus subsystems, driver model, IIO, hwmon, sysfs і udev. Для звичайних цифрових датчиків це дає рівень стандартизації, повторного використання коду і сумісності з користувацькими інструментами. Але для нестандартного Arduino-сумісного сенсора ці механізми працюють лише частково. Якщо пристрій використовує стандартну шину і має типову модель каналів, його можна інтегрувати в IIO або hwmon. Якщо ж він працює через власний протокол, повертає нестандартні пакети або потребує особливої обробки подій, можна використати sysfs і udev як інтерфейс експорту і механізм автоматичного виявлення, але центральною частиною рішення стає власний драйвер або модуль ядра, який переводить дані сенсора у зрозумілу для Linux форму.

1.3 Постановка задачі

Сучасні вбудовані системи на базі Linux потребують надійної підтримки драйверів на рівні операційної системи. Особливо це стосується тих випадків, коли використовується нестандартний Arduino-сумісний сенсор, для якого немає готового драйвера в ядрі Linux. В такій ситуації підключення пристрою або простого користувацького скрипта вже недостатньо. Система повинна не лише отримати дані від сенсора, а й правильно їх розпізнати, перевірити, перетворити у зручний формат та надати іншим програмам через стандартні механізми Linux.

Проблема полягає в тому, що нестандартний сенсор не завжди відповідає типовим схемам роботи, які вже підтримуються ядром Linux. Якщо звичайний цифровий датчик може бути підключений через стандартний інтерфейс і обслуговуватись готовим модулем системи, то нестандартний Arduino-сумісний пристрій часто використовує власний формат даних, власну послідовність ініціалізації або нетиповий спосіб передавання вимірювань. Через це система не може автоматично надати стабільний і зручний доступ до його показників. Тому виникає потреба у створенні спеціального програмного модуля ядра, який виступатиме проміжною ланкою між фізичним сенсором і програмним середовищем Raspberry Pi.

Основою роботи є створення модуля ядра Linux, який забезпечує коректну взаємодію Raspberry Pi з нестандартним Arduino-сумісним сенсором. Такий модуль повинен виконувати кілька важливих ролей. Він має забезпечити виявлення пристрою, запуск процедури ініціалізації, приймання даних, перевірку їхньої коректності, нормалізацію значень і подання результату в стандартному вигляді для простору користувача. Фактично модуль має перетворити сенсор з зовнішнього нестандартного пристрою на кероване джерело вимірювальних даних в Linux.

Отже, метою роботи є розробка програмного модуля ядра Linux для нестандартного Arduino-сумісного сенсора, який працює на Raspberry Pi та забезпечує приймання, обробку, нормалізацію й експорт даних у стандартне програмне середовище Linux. Для досягнення цієї мети необхідно вирішити такі завдання:

1. Розробити загальну структуру модуля та визначити його місце у системі Raspberry Pi.

2. Описати алгоритм функціонування модуля ядра Linux.

3. Розробити інформаційне забезпечення модуля, включаючи модель даних, структуру атрибутів sysfs, взаємодію з udev та засоби діагностики.

4. Здійснити програмну реалізацію та тестування працездатності модуля.

Розроблюваний модуль має забезпечити зрозумілий шлях від фізичного вимірювання до готової системної метрики, доступної для подальшого використання в програмах моніторингу, керування та обробки даних.

2 АЛГОРИТМІЧНЕ ТА ІНФОРМАЦІЙНЕ ЗАБЕЗПЕЧЕННЯ ПРОГРАМНОГО МОДУЛЯ

2.1 Загальна структура проектованого модуля

Модуль повинен розглядатися як частина всієї системи Raspberry Pi, в якій є апаратний рівень, рівень ядра Linux, системні інтерфейси експорту параметрів і засоби користувача, що отримують доступ до метрик. Загальну структуру модуля краще будувати так, щоб кожен рівень системи мав чітке призначення, а передавання даних між ними залишалося стабільним і передбачуваним. (рисунок 2.1).

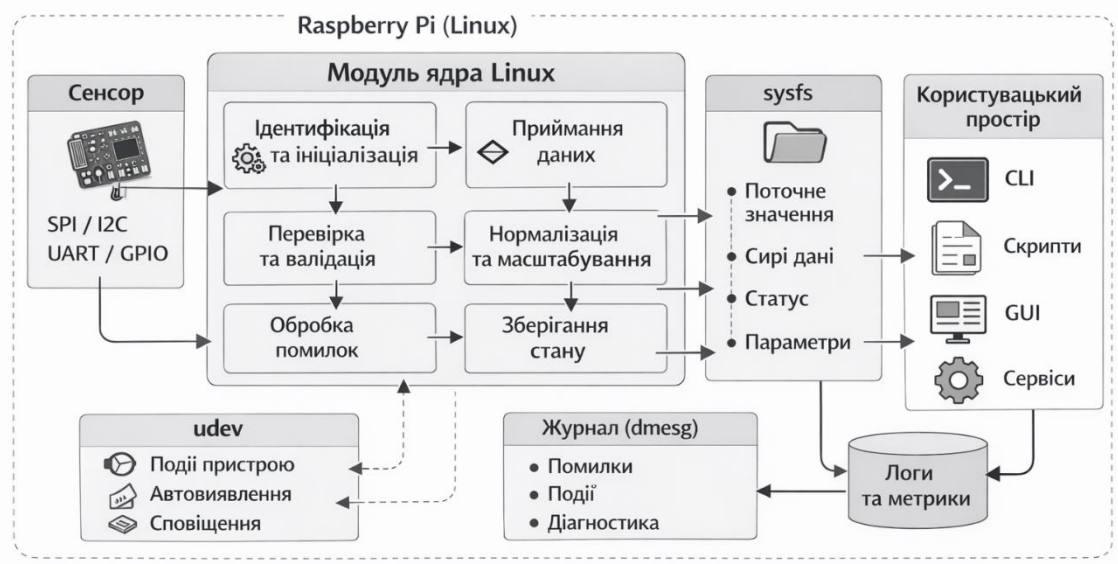


Рисунок 2.1 – Загальна структура проектованого модуля

Raspberry Pi виступає центральною платформою, на якій поєднуються зовнішній сенсор, програмний модуль ядра та прикладні засоби доступу до інформації. Сенсор є джерелом первинних даних. Він формує значення, які надходять до плати через відповідний цифровий інтерфейс. Далі вступає в роботу модуль ядра Linux, який приймає ці значення, виконує розбір формату, перевіряє їхню коректність, нормалізує отриману інформацію та готує її до подальшого подання системі. Після цього результати стають доступними через системні механізми, такі як sysfs, а також можуть супроводжуватися подіями, на які реагує

udev. В просторі користувача до цих даних вже звертаються скрипти, консольні утиліти, діагностичні програми або власні прикладні засоби моніторингу. Модуль ядра в цій структурі забезпечує взаємозв'язок між пристроєм і програмами користувача.

Якщо дивитися на місце модуля в структурі всієї системи Raspberry Pi, то він належить до простору ядра і взаємодіє одночасно у двох напрямах. Знизу він працює з апаратним пристроєм, а зверху надає доступ до результатів роботи для інших компонентів Linux, тобто модуль не повинен поводитися як окрема ізольована програма. Його основне завдання — включити нестандартний сенсор в звичну логіку роботи Linux-системи. Для користувача або прикладного програмного забезпечення сенсор повинен виглядати як звичне джерело метрик, а не як набір нестандартних байтів або нестабільний потік повідомлень

Загальна структура проєктованого модуля повинна враховувати те, що нестандартний Arduino-сумісний сенсор може не відповідати повністю типовим моделям Linux-підтримки. Частина таких сенсорів передає дані у власному форматі, частина вимагає спеціальної послідовності запуску, а частина може повертати не тільки корисні дані, а й службові стани, попередження або пакети з помилками. Саме тому всередині модуля потрібна внутрішня структуризація, за якої окремі функціональні блоки відповідають за різні етапи роботи з пристроєм.

У запропонованій структурі можна виділити п'ять основних шарів.

1. Перша частина — це сам сенсор, який формує сирі дані.

2. Друга частина — модуль ядра, в якому виконується ідентифікація пристрою, ініціалізація та первинна обробка даних.

3. Третя частина — блок експорту метрик у sysfs.

4. Четверта частина — механізм системних подій, пов'язаний із udev.

5. П'ята частина — користувацькі засоби, які читають значення сенсора або реагують на зміну його стану.

В такій схемі сенсор і прикладні програми не взаємодіють напряму. Між ними завжди є модуль ядра, який забезпечує керований і стандартизований доступ до інформації.

Початковим функціональним блоком в структурі модуля є блок ідентифікації та ініціалізації сенсора. Його роль полягає в тому, щоб під час завантаження модуля або при появі пристрою визначити, чи доступний сенсор, чи відповідає він очікуваному формату, і чи можна перейти до робочого режиму. На цьому етапі модуль готує внутрішні структури даних, налаштовує параметри обміну, встановлює початковий стан пристрою та перевіряє, чи сенсор готовий до передавання корисної інформації. Якщо на цій стадії виявлено помилку, наприклад сенсор не відповідає або повертає неправильний ідентифікатор, модуль повинен завершити ініціалізацію коректно і не допустити переходу системи до помилкового робочого стану.

Після успішної ініціалізації починає працювати блок приймання, розбору та перевірки даних. Це один із ключових елементів модуля. Він отримує сирі значення від сенсора через відповідний інтерфейс, аналізує структуру пакета, виділяє корисні поля та перевіряє їх на коректність. Якщо дані надходять у текстовому вигляді, цей блок повинен виконати синтаксичний розбір рядка та перетворити його на числові величини. Якщо використовується бінарний протокол, потрібно виділити окремі байти або слова, зібрати з них значення і переконатися, що пакет не пошкоджений. Тут також перевіряється довжина повідомлення, контрольні ознаки та допустимі межі значень.

Наступним блоком є нормалізація та внутрішнє подання метрик. Його завдання полягає в тому, щоб перетворити сирі дані сенсора у зрозумілі системі величини. Якщо сенсор передає код або умовну одиницю, модуль повинен застосувати коефіцієнт масштабування, зміщення або інше правило перерахунку. Якщо вимірювання вже передане у готовому вигляді, модуль усе одно має привести його до єдиного внутрішнього формату, з яким надалі зручно працювати. У цьому ж блоці доцільно підтримувати не лише поточне значення, а й пов’язаний із ним службовий стан, час останнього оновлення, сире значення та ознаку помилки. Такий підхід робить модуль придатним не тільки для отримання одного числа, а й для повноцінного моніторингу поведінки пристрою.

Окремим блоком в структурі проектованого модуля є блок експорту параметрів через sysfs. Через нього результат роботи драйвера стає видимим для системи та користувача. Через sysfs можна подати поточні метрики, службові параметри, ознаки помилок і, якщо потрібно, налаштовувані характеристики. Завдяки цьому користувачеві не потрібно знати внутрішній формат пакета сенсора або будову самого драйвера. Для нього важливо лише те, що потрібні атрибути доступні у звичайному файловому вигляді. В загальній структурі це означає, що sysfs є інтерфейсом між простором ядра і простором користувача. Сам sysfs не обробляє дані, проте надає зручний доступ до вже підготовлених результатів.

Разом з sysfs в структурі є і механізм udev. Якщо sysfs відповідає за подання атрибутів, то udev допомагає системі автоматично реагувати на появу, зміну стану або зникнення пристрою.

Модуль ядра повинен формувати таку взаємодію із системними подіями, щоб користувацькі засоби могли реагувати на зміну стану сенсора не лише опитуванням, а й подієвим способом.

До блоку користувацьких засобів можна віднести командний рядок, тестові утиліти, скрипти моніторингу, програми для зчитування і фіксації метрик або допоміжні сервіси, що реагують на події udev. Ці засоби не повинні звертатися до сенсора напряму. Вони повинні працювати лише через ті інтерфейси, які надає система.

Для Raspberry Pi така структура є робочою, тому що ця платформа застосовується як проміжний обчислювальний вузол між зовнішніми сенсорами та програмним середовищем моніторингу. Вона достатньо гнучка, щоб підключати різні типи пристроїв, але при цьому її програмна логіка повинна бути чітко впорядкованою.

Крім того така структура розрахована на розширення. На першому етапі модуль може обслуговувати лише один сенсор і надавати обмежений набір атрибутів. Надалі модуль можна розширити додавши нові параметри, кілька режимів роботи, ширшу реакцію на події та додаткові канали діагностики.

2.2 Алгоритми функціонування модуля ядра Linux

Роботу модуля ядра Linux для нестандартного Arduino-сумісного сенсора доцільно розглядати як послідовність взаємопов’язаних етапів, які починаються із завантаження модуля в ядро і завершуються його коректним вивантаженням з системи (рисунок 2.2). Такий модуль працює всередині ядра Linux, тому він повинен приймати дані від сенсора та робити це в спосіб, який не порушує стабільність системи, не створює конфліктів з підсистемами ядра та забезпечує передбачуваний доступ до метрик із простору користувача.

Початковим етапом алгоритму є ініціалізація та завантаження модуля.

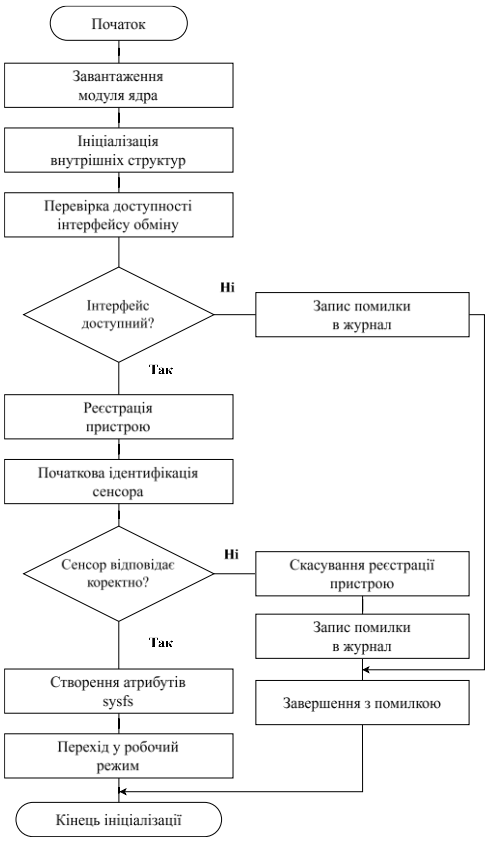


Рисунок 2.2 - Схема алгоритму ініціалізації та завантаження модуля

На початку ядро підключає модуль, виконує його початкову функцію і передає йому керування для підготовки до роботи. На цьому етапі модуль перевіряє, чи доступні всі ресурси, необхідні для взаємодії з сенсором. Якщо пристрій працює через I2C, SPI, UART або GPIO, модуль підбирає відповідний інтерфейс. Далі виконується створення внутрішніх структур даних, в яких будуть

зберігатися службовий стан пристрою, останні отримані значення, сирі дані, ознаки помилок та допоміжні параметри.

Після підготовки внутрішніх структур модуль переходить до етапу реєстрації пристрою. Якщо сенсор підключений через стандартну шину, модуль повинен зареєструвати його відповідно до вибраного механізму ядра. Якщо використовується власна схема зв'язку або нестандартний адаптер, логіка реєстрації може бути спрощеною, але навіть у такому випадку модуль має створити контрольовану точку доступу до пристрою. Під час реєстрації встановлюється, що саме система вважатиме активним сенсором, у якому стані він перебуває на старті, які функції будуть доступні надалі та які атрибути потрібно створити в sysfs.

Наступним етапом алгоритму є перевірка доступності сенсора та його початкова ідентифікація. У випадку нестандартного Arduino-сумісного пристрою це потрібно, тому що ядро не має вбудованого знання про його формат, службові команди або особливості відповіді. Модуль повинен ініціювати перший обмін із сенсором, одержати відповідь і перевірити, що з'єднання справді встановлено з правильним пристроєм.

Якщо сенсор повертає ідентифікатор, службове повідомлення або початковий пакет стану, ці дані перевіряються до переходу в основний режим роботи. Якщо пристрій не відповідає, відповідає із затримкою або надсилає дані, що не відповідають очікуваному шаблону, алгоритм має зупинити запуск робочої частини модуля та перевести його в безпечний стан із повідомленням про помилку.

Після успішної ідентифікації та реєстрації починається основний робочий цикл модуля (рисунок 2.3).

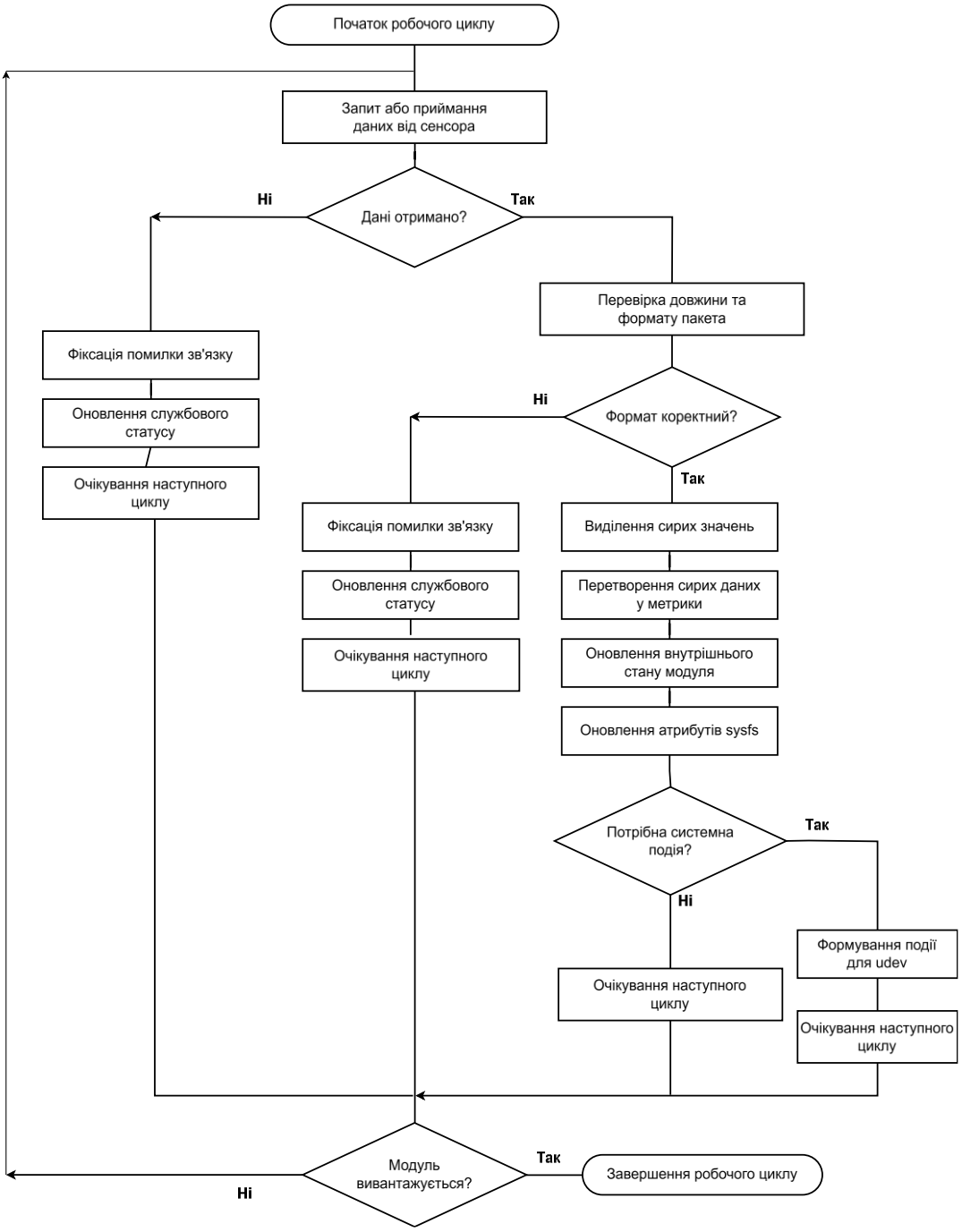


Рисунок 2.3 - Схема алгоритму основного робочого циклу модуля

На цьому етапі виконується зчитування даних з сенсора. В найпростішому варіанті це періодичне опитування пристрою через певний проміжок часу. В складнішому випадку модуль може працювати подієво, коли зчитування відбувається після певного сигналу, переривання або службової події. Незалежно від способу роботи модуль має приймати пакет даних і перевіряти його базову коректність.

Після приймання даних модуль переходить до етапу розбору сирого повідомлення, тому що нестандартний сенсор деколи не повертає готову системну метрику у вигляді, зручному для Linux. Він може передавати числа у власному форматі, кілька значень в одному пакеті, текстові рядки, байтові послідовності або змішані службові та корисні поля. Тому модуль повинен відокремити корисну частину повідомлення від службової, перевірити правильність полів і виділити сирі значення для подальшої обробки. Якщо формат не відповідає очікуваному шаблону, алгоритм не переходить до етапу оновлення метрик, а фіксує помилку, записує її в журнал і пробує повторне зчитування, або чекає наступного циклу роботи.

Після успішного розбору сирих даних модуль перетворює їх у метрики. Сенсор може передавати значення у вигляді кодів, імпульсів, байтових слів або внутрішніх одиниць вимірювання, які ще не є зручними для прикладних програм. Модуль повинен застосувати необхідне правило перерахунку. Якщо сенсор передає кілька пов’язаних параметрів, алгоритм має сформувати окремі метрики для кожного з них.

Після перетворення сирих даних результати зберігаються у внутрішньому стані модуля. З цих внутрішніх структур інформація потрапляє до sysfs або використовується для реакції на події. У нормальному режимі роботи модуль тримає останнє успішне отримане значення, сире значення, інформацію про стан сенсора і код останньої помилки, якщо така виникала.

Далі модуль оновлює атрибути sysfs. Цей етап робить результати роботи модуля видимими для системи та простору користувача. Тобто після кожного успішного циклу зчитування і перетворення оновляються атрибути, які містять поточне значення, сирі дані, службовий статус. Якщо відбулося успішне зчитування, атрибут із поточним значенням повинен містити нову метрику. Якщо виникла помилка, sysfs може не оновлювати значення метрики, але має змінити службовий стан або надати ознаку помилки.

Паралельно з оновленням sysfs алгоритм може формувати події для udev. Для загального алгоритму це означає, що модуль підтримує значення в sysfs та у разі потреби сигналізує системі про суттєву зміну стану сенсора.

Окремий великий блок алгоритму становить обробка помилок (рисунок 2.4) зв’язку та аварійних ситуацій.

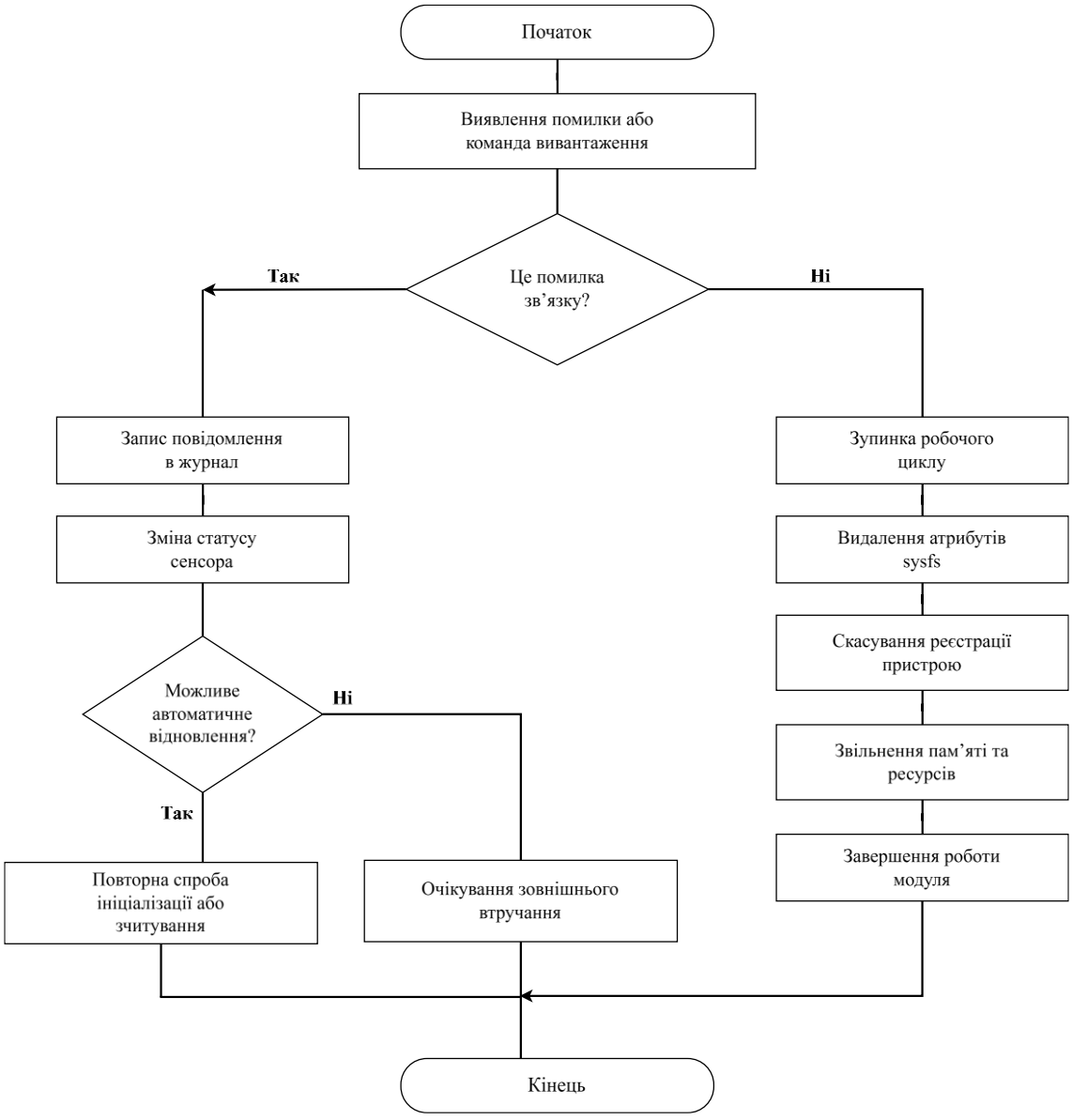


Рисунок 2.4 Схема алгоритму обробки помилок зв’язку та аварійних ситуацій

В реальній роботі сенсор може перестати відповідати, передати пошкоджене повідомлення, повернути недопустиме значення або тимчасово бути відключеним. Тому алгоритм повинен містити перевірку всіх критичних точок. Спочатку перевіряється сам факт отримання відповіді. Далі перевіряється коректність

довжини повідомлення. Після цього контролюється формат полів і допустимість значень. Якщо будь-яка з цих умов не виконується, модуль не повинен передавати результат далі як правильну метрику. Але він змінює службовий статус, і фіксує помилку у внутрішньому стані, повідомляє про неї в системний журнал і переходить до повторної спроби, або до очікування наступного циклу.

Ще одним важливим елементом алгоритму є журналювання подій. Він є основним джерелом діагностики під час розробки і тестування модуля. Тому алгоритм повинен передбачати запис основних подій, таких як успішне завантаження модуля, початкова ініціалізація сенсора, виявлення помилки, відновлення зв'язку та вивантаження модуля. Повідомлення журналу мають бути короткими, змістовними та не повторюватися без потреби.

Завершальним етапом алгоритму є вивантаження модуля і звільнення ресурсів. Якщо модуль коректно працює під час зчитування, але неправильно завершує роботу, в системі можуть залишитися неактуальні атрибути, зайняті дескриптори, невивільнена пам'ять або некоректний стан пристрою. Тому під час вивантаження модуль повинен зупинити цикл зчитування, скасувати відкладені операції, видалити створені атрибути sysfs, завершити роботу з інтерфейсом обміну, відв'язати пристрій від своїх внутрішніх структур і звільнити всю пам'ять, яка була виділена на етапі ініціалізації. Якщо сенсор підтримує завершальну команду або перехід у безпечний режим, її також варто виконати до повного завершення модуля.

2.3 Інформаційне забезпечення модуля

Для модуля ядра Linux який буде працювати з Arduino-сумісним сенсором потрібно забезпечити цілісний шлях руху інформації від моменту її надходження з сенсора до моменту, коли ця інформація стає доступною для системи Raspberry Pi, користувацьких засобів, сценаріїв автоматизації та засобів діагностики.

Вхідна інформація для модуля формується самим сенсором або проміжним Arduino-сумісним вузлом, який передає дані у цифровому вигляді. В загальному

випадку це може бути один пакет даних, окреме числове значення або серія коротких повідомлень, що надходять через цифровий інтерфейс. Для нестандартного сенсора потрібно, тому що на рівні модуля вхідна інформація має не готовий системний формат, а власну внутрішню структуру пристрою. Тому вхідні дані розглядаються не як готові метрики, а як первинні повідомлення, які потрібно розібрати й перевірити.

В модулі вхідна інформація складається з кількох інформаційних шарів. Перший шар — це сире значення, тобто те, що фактично надійшло від сенсора без перетворення. Другий шар — це технічна інформація про сам факт отримання даних. Сюди належать довжина пакета, час приймання, номер циклу зчитування, ознака коректності структури або помітка про втрату кадру. Третій шар — це службова інформація від самого сенсора, наприклад код режиму, стан калібрування, маркер готовності або помилка вимірювання. Таке розділення дозволяє відокремити корисну частину даних від діагностичної та допоміжної інформації (рисунок 2.5).

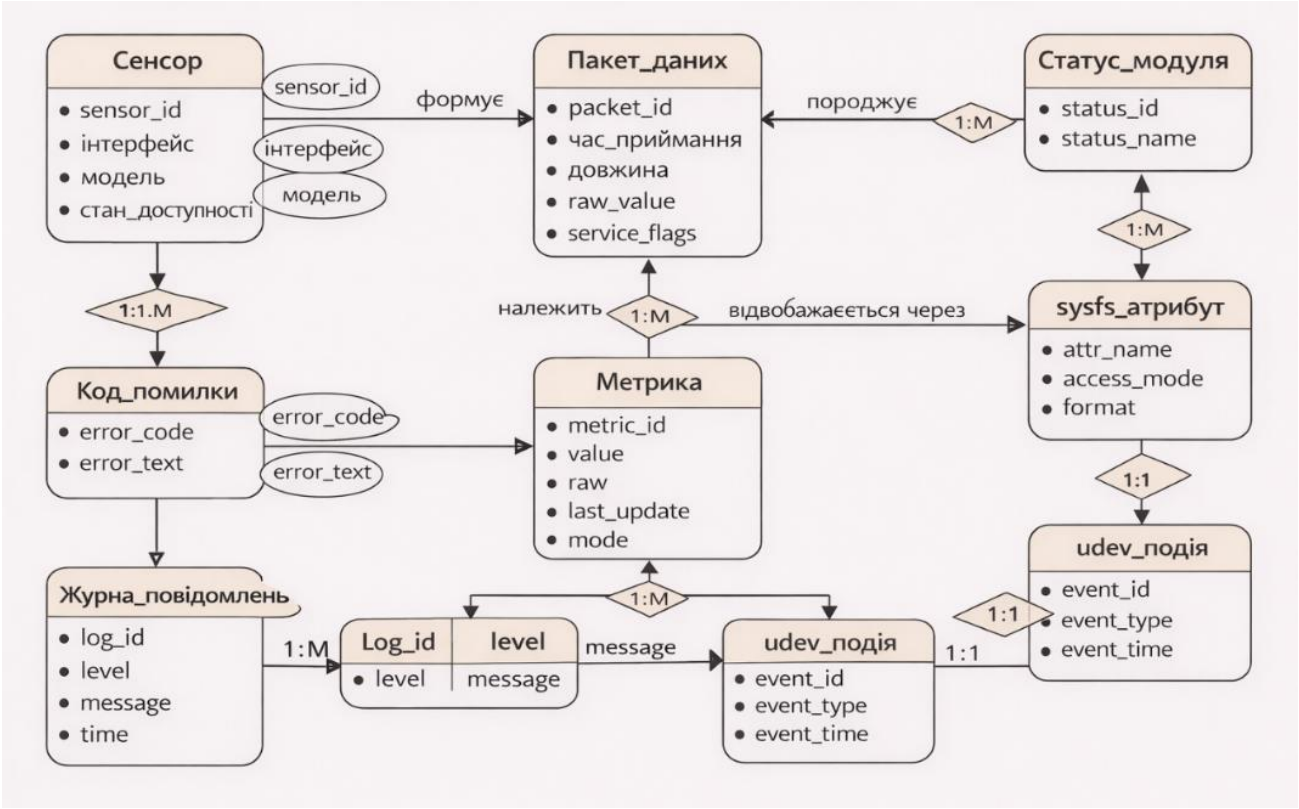


Рисунок 2.5 — Концептуальна інфологічна модель даних модуля

Службові параметри самого модуля не надходять ззовні, а формуються всередині драйвера в процесі його роботи. До них можна віднести стан ініціалізації, режим доступу до пристрою, час останнього успішного зчитування, кількість невдалих спроб доступу, прапор доступності сенсора, внутрішній код помилки та мітку працездатності. Ці параметри використовуються як у внутрішній логіці модуля, так і для коректного відображення стану пристрою назовні.

Вихідна інформація формується вже після того, як модуль розібрав сирі дані, перевірив їх і привів до зручного вигляду. Основу вихідної інформації становлять метрики, які далі використовуються в Linux-середовищі. Також потрібно формувати кілька пов’язаних вихідних параметрів:

– поточне нормалізоване значення;

– сире значення, якщо воно потрібне для діагностики або перевірки масштабування;

– інформація про стан вимірювання, наприклад ознака того, що значення актуальне;

– службові параметри, які пояснюють режим роботи сенсора або частоту оновлення.

Окремо є статусні повідомлення та коди помилок, тому що помилки можуть виникати на різних етапах. Пристрій може бути фізично відсутнім, може не пройти ідентифікацію, може передати пошкоджений пакет. Якщо всі ці ситуації зводити до одного загального повідомлення, діагностика модуля ускладнюється. Тому використано кілька чітко визначених станів. Стан ініціалізації, стан нормальної роботи, стан тимчасової недоступності, стан помилки формату даних, стан помилки зчитування та стан аварійного завершення. Кожен з таких станів супроводжується власним кодом або текстовим поясненням, щоб прикладні програми і користувач могли зрозуміти причину проблеми.

Логічна модель подання метрик визначає, в якому вигляді результати роботи модуля стають доступними для системи та користувача. Для Linux способом такого подання є sysfs. Саме через нього дані сенсора можуть бути представлені як набір

стабільних атрибутів, які читаються звичайними системними засобами (рисунок 2.6).

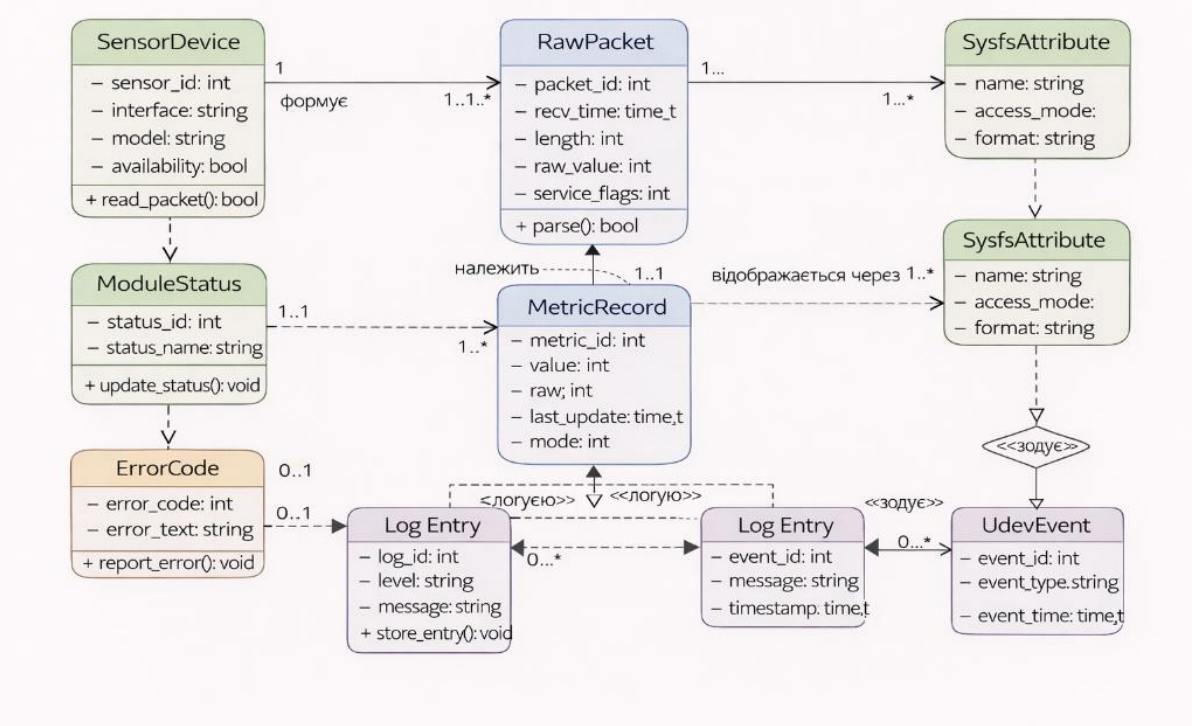


Рисунок 2.6 — UML-діаграма класів інформаційних об’єктів модуля

Метрики пристрою подаються у вигляді окремої директорії пристрою, де кожен атрибут відповідає певному типу інформації. Така структура розділяє корисні значення, службові параметри та діагностичні ознаки. Тобто, окремий атрибут для нормалізованого значення, окремий — для сирого значення, окремий — для статусу, окремий — для коду помилки, окремий — для часу останнього оновлення.

Для основного значення використовується value, для сирих даних — raw, для стану — status, для коду помилки — error_code, для часу останнього успішного вимірювання — last_update. Якщо сенсор надає більше однієї метрики, тоді імена уточнюються, наприклад temperature_value, humidity_value, signal_raw.

Для числових величин використано простий текстовий формат без зайвих службових символів, тобто одне значення — один рядок. Це узгоджується з логікою sysfs. Для статусу використано короткі текстові позначення ready, init, error, offline. Для коду помилки використано числове значення, а пояснення до нього вноситься в документацію модуля або дублюється в журналі ядра. Для часу

останнього оновлення використано часову позначку, щоб її можна було порівнювати між циклами роботи.

Правила оновлення та читання значень є однаковими для всіх атрибутів. Якщо нові дані від сенсора отримано успішно, спочатку оновлюється внутрішній стан модуля, а вже потім значення стають видимими через sysfs.

Атрибути sysfs мають бути доступними для безпечного читання у будь-який момент роботи модуля. Тобто, внутрішні дані зберігаються так, щоб користувачеві не потрапляло пошкоджене або напівоновлене значення.

Для повноцінної роботи в системі Linux ще використовується udev, який обробляє події ядра і дозволяє зв’язати ці події з користувацькими сценаріями.

В модулі виділено наступні події:

– подія появи пристрою, коли в момент завантаження модуля або підключення сенсора, система повинна зрозуміти, що доступне нове джерело даних.

– подія зміни стану включається, якщо сенсор переходить в режим помилки, відновлює зв’язок або змінює режим роботи.

– подія відключення пристрою або завершення роботи модуля.

Зв’язок між ядром, udev і користувацькими скриптами має бути послідовним. Тобто модуль ядра змінює стан пристрою або його атрибути. Якщо ця зміна є суттєвою, формується системна подія. Її отримує udev і на основі правил вирішує, чи потрібно створити додатковий запис, символьне ім’я або запустити певний сценарій. Далі користувацький скрипт може звернутися до sysfs і отримати потрібні параметри вже у зрозумілому вигляді (рисунок 2.7).

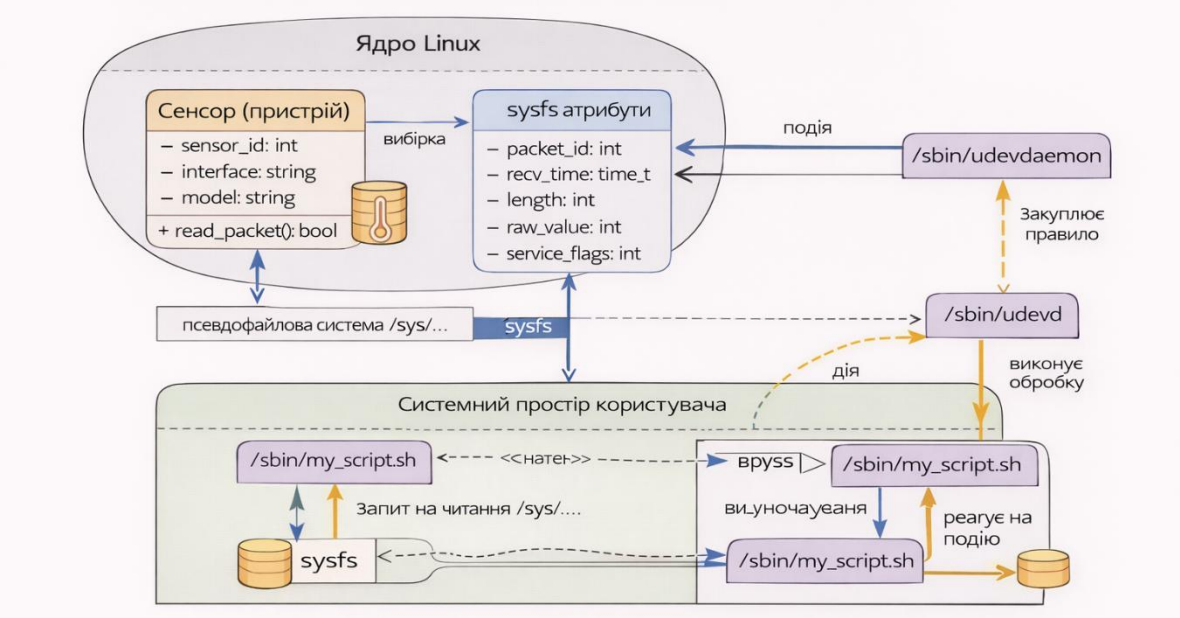


Рисунок 2.7 — Фізична схема подання даних через sysfs та udev

Журналювання та діагностика потрібні для того, щоб модуль можна було перевіряти, налагоджувати та супроводжувати.

Крім того, модуль формує повідомлення ядра на основних етапах роботи. Перший блок відповідає за завантаження та ініціалізацію. Другий блок пов’язаний із робочим циклом. Третій блок повідомлень - помилки.

Діагностичними ознаками правильної роботи модуля є успішне створення атрибутів sysfs і доступність їх для читання. Окремо перевіряється коректне оновлення статусу під час переходу сенсора між режимами роботи. Якщо модуль працює правильно, журнал ядра містить зрозумілі повідомлення без постійного повторення однакових помилок, а дані в sysfs змінюються відповідно до реального стану сенсора.

Інформаційне забезпечення модуля покриває весь шлях даних: від сирого повідомлення сенсора до системно впорядкованої метрики, доступної через sysfs і придатної для автоматичної обробки через udev та користувацькі сценарії. У вхідній частині відокремлюються корисні дані від службових полів. У внутрішній частині підтримуються службові параметри модуля, статусні ознаки та коди помилок. У вихідній частині надається логічна модель атрибутів sysfs, де значення, стан і діагностична інформація подані у простому і стабільному вигляді.

Доповнення цієї моделі механізмом udev і засобами журналювання робить модуль функціональним та придатним до реальної експлуатації на платформі Raspberry Pi.

3 ПРОГРАМНО-ТЕХНОЛОГІЧНЕ ЗАБЕЗПЕЧЕННЯ МОДУЛЯ

3.1 Реалізація програмного модуля ядра Linux

На етапі реалізації, розроблення та первинного налагодження модуля виконувалась у віртуальній машині Debian Linux. Такий підхід дав змогу відпрацювати логіку модуля, механізми sysfs, журналювання, інтеграцію з udev і базові сценарії роботи. Програмна реалізація побудована так, щоб її можна було надалі адаптувати до Raspberry Pi, замінивши тестовий спосіб подання вхідних даних на реальний апаратний обмін через відповідний цифровий інтерфейс.

Для реалізації модуля використано мову C, так як вона є основною мовою розробки модулів ядра Linux, забезпечує прямий доступ до структур ядра, дозволяє використовувати внутрішні механізми роботи з пам’яттю, файловими атрибутами, mutex-блокуваннями та системними подіями. Також використання C забезпечує сумісність із типовими засобами складання модулів ядра. Для цього були використані компілятор GCC, утиліта make, пакет build-essential та заголовкові файли поточного ядра Linux.

Структура реалізованого проєкту була побудована як невеликий набір взаємопов’язаних файлів.

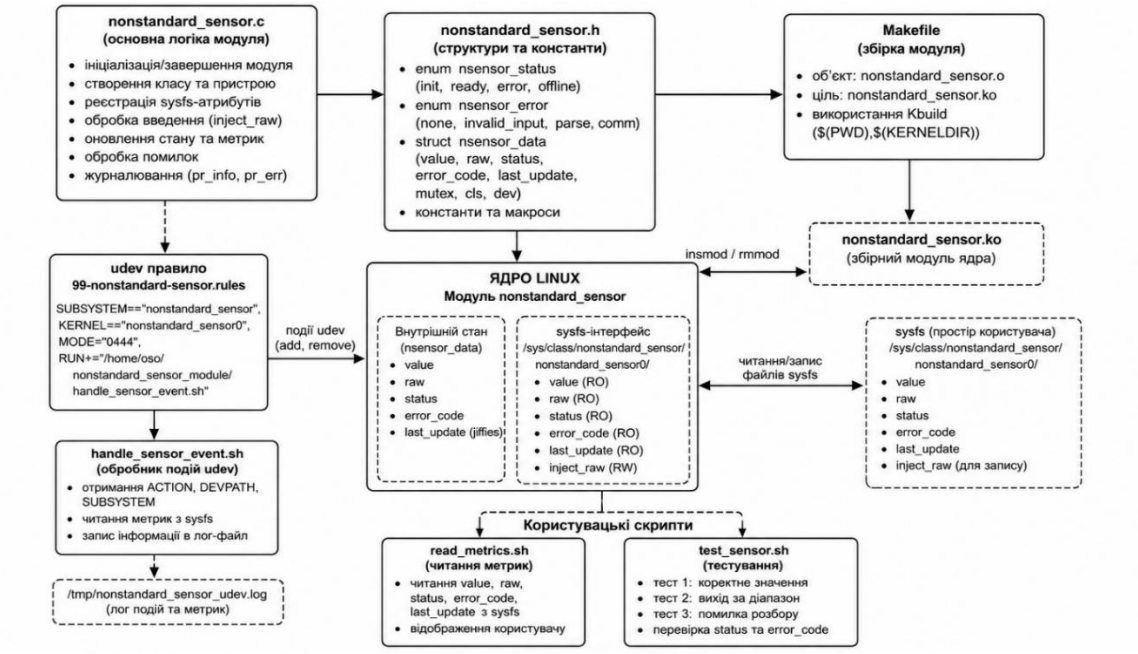


Рисунок 3.1 – Структурна схема програмної реалізації модуля

Основний вихідний код модуля розміщено у файлі nonstandard_sensor.c. В ньому реалізовано точки входу та завершення модуля, створення класу пристрою, формування sysfs-атрибутів, обробку тестових вхідних даних, оновлення внутрішнього стану модуля та журналювання. Заголовковий файл nonstandard_sensor.h містить основні константи, перелік статусів, перелік кодів помилок та опис внутрішньої структури nsensor_data, у якій зберігається актуальний стан пристрою.

Таблиця 3.1 – Склад файлів програмної реалізації модуля та їх призначення

Назва файла	Тип файла	Призначення	Основний вміст
nonstandard_sensor.c	вихідний файл модуля ядра	реалізація основної логіки модуля	ініціалізація і завершення модуля, створення класу і пристрою, формування sysfs-атрибутів, обробка введення через inject_raw, оновлення метрик, обробка помилок, журналювання
nonstandard_sensor.h	заголовковий файл	опис структур даних і службових констант	enum nsensor_status, enum nsensor_error, struct nsensor_data, макроси і прототипи допоміжних функцій
Makefile	файл складання	збирання зовнішнього модуля ядра Linux	опис об’єкта nonstandard_sensor.o, використання Kbuild, команди all і clean
99-nonstandard-sensor.rules	файл правил udev	налаштування реакції системи на появу пристрою	правило для підсистеми nonstandard_sensor, задання параметрів доступу, запуск скрипта-обробника події
read_metrics.sh	shell-скрипт	читання метрик модуля в просторі користувача	зчитування значень value, raw, status, error_code, last_update з каталогу /sys/class/nonstandard_sensor/nonstandard_sensor0/
test_sensor.sh	shell-скрипт	функціональна перевірка	тест коректного введення, тест помилки діапазону, тест помилки розбору, перевірка оновлення status та error_code

		роботи модуля	
handle_sensor_event.sh	shell-скрипт	обробка подій udev	отримання змінних середовища події, читання sysfs-атрибутів, запис інформації про подію та метрики у лог-файл /tmp/nonstandard_sensor_udev.log

Файл Makefile використовується для складання модуля командою make через стандартний механізм зовнішніх модулів ядра Linux. Окремо створено файл 99-nonstandard-sensor.rules, який містить правило udev, а також кілька допоміжних shell-скриптів для взаємодії з модулем.

Центральним програмним елементом реалізації є структура nsensor_data (рисунок 3.2).

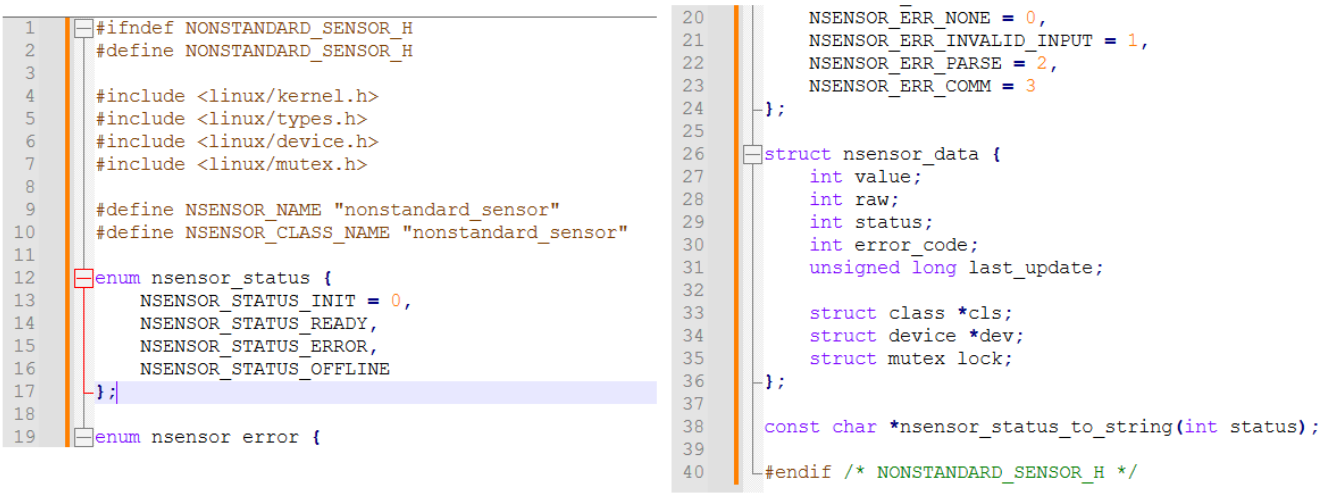


Рисунок 3.2 – Фрагмент коду заголовкового файлу nonstandard_sensor.h

Вона використовується для зберігання основної інформації про поточний стан модуля. До цієї структури включено поля value, raw, status, error_code і last_update, а також службові поля cls, dev і lock.

Для опису станів модуля було введено перелік nsensor_status, який містить значення NSENSOR_STATUS_INIT, NSENSOR_STATUS_READY, NSENSOR_STATUS_ERROR і NSENSOR_STATUS_OFFLINE. Для опису причин збою введено перелік nsensor_error, який містить коди NSENSOR_ERR_NONE, NSENSOR_ERR_INVALID_INPUT, NSENSOR_ERR_PARSE і

NSENSOR_ERR_COMM. Така структура дає змогу окремо фіксувати сам факт помилки та її тип. Окрема допоміжна функція nsensor_status_to_string реалізує перетворення внутрішнього числового коду статусу на текстове представлення, яке відображається через sysfs.

Початковою точкою роботи драйвера є функція nsensor_init, яка виконується під час завантаження модуля в ядро Linux. У цій функції спочатку очищується структура g_sensor, ініціалізується mutex і встановлюються початкові значення полів. На цьому етапі статус встановлюється в init, код помилки дорівнює нулю, а значення метрик мають початкове нульове значення. Після цього створюється клас nonstandard_sensor і пристрій nonstandard_sensor0. Далі для пристрою створюється група sysfs-атрибутів. Якщо будь-який із цих етапів завершується помилкою, функція коректно повертає код помилки і звільняє вже створені ресурси. Якщо ж усі етапи проходять успішно, статус змінюється на ready, оновлюється поле last_update, а в журнал ядра записуються повідомлення про успішне завантаження модуля і шлях до sysfs-каталогу.

Завершення роботи модуля реалізоване у функції nsensor_exit. В ній виконується видалення групи sysfs-атрибутів, знищення пристрою, знищення класу та запис повідомлення до журналу ядра.

Реалізація sysfs-інтерфейсу. Для кожного атрибута створено окрему функцію читання. Так реалізовано атрибути value, raw, status, error_code і last_update.

```c
static ssize_t value_show(struct device *dev,
                          struct device_attribute *attr, char *buf)
{
    ssize_t len;

    mutex_lock(&g_sensor.lock);
    len = sysfs_emit(buf, "%d\n", g_sensor.value);
    mutex_unlock(&g_sensor.lock);

    return len;
}
static ssize_t status_show(struct device *dev,
                           struct device_attribute *attr, char *buf)
{
    ssize_t len;

    mutex_lock(&g_sensor.lock);
    len = sysfs_emit(buf, "%s\n", nsensor_status_to_string(g_sensor.status));
    mutex_unlock(&g_sensor.lock);

    return len;
}
```

```c
static ssize_t error_code_show(struct device *dev,
                               struct device_attribute *attr, char *buf)
{
    ssize_t len;

    mutex_lock(&g_sensor.lock);
    len = sysfs_emit(buf, "%d\n", g_sensor.error_code);
    mutex_unlock(&g_sensor.lock);

    return len;
}
static DEVICE_ATTR_RO(value);
static DEVICE_ATTR_RO(status);
static DEVICE_ATTR_RO(error_code);
static struct attribute *nsensor_attrs[] = {
    &dev_attr_value.attr,
    &dev_attr_status.attr,
    &dev_attr_error_code.attr,
    NULL
};
```

Рисунок 3.3 – Код реалізації sysfs-атрибутів модуля

Кожна функція використовує mutex-блокування, щоб запобігти некоректному доступу до внутрішнього стану модуля в момент одночасного

читання і зміни значень. Для генерації текстового представлення результатів використовується sysfs_emit. Реалізовані атрибути є тільки для читання і надають користувачеві доступ до актуального стану модуля без необхідності звертатися до внутрішньої логіки драйвера.

Для моделювання роботи нестандартного сенсора на випадок відсутності реального апаратного пристрою було реалізовано спеціальний атрибут inject_raw.

```
1  static ssize_t inject_raw_store(struct device *dev,
2                                  struct device_attribute *attr,
3                                  const char *buf, size_t count)
4  {
5      int ret;
6      int raw_value;
7      ret = kstrtoint(buf, 10, &raw_value);
8      mutex_lock(&g_sensor.lock);
9      if (ret) {
10         nsensor_set_error(NSENSOR_ERR_PARSE);
11         mutex_unlock(&g_sensor.lock);
12         pr_err("%s: failed to parse input\n", NSENSOR_NAME);
13         return -EINVAL;
14     }
15     if (raw_value < 0 || raw_value > 1000) {
16         nsensor_set_error(NSENSOR_ERR_INVALID_INPUT);
17         mutex_unlock(&g_sensor.lock);
18         pr_err("%s: invalid raw input: %d\n", NSENSOR_NAME, raw_value);
19         return -EINVAL;
20     }
21     nsensor_update_from_raw(raw_value);
22     mutex_unlock(&g_sensor.lock);
23
24     pr_info("%s: injected raw=%d value=%d\n",
25             NSENSOR_NAME, raw_value, raw_value * 2);
26     return count;
27 }
```

Рисунок 3.4 – Функція inject_raw_store для обробки тестових вхідних даних

Цей атрибут має і функцію читання, і функцію запису. Через нього в модуль подаються тестові сирі значення. Запис виконується через функцію inject_raw_store, в якій спочатку виконується спроба перетворення текстового вводу в ціле число за допомогою kstrtoint. Якщо розбір завершився невдачею, модуль фіксує помилку типу parse, встановлює відповідний код помилки, переводить статус у стан error і повертає помилку EINVAL. Якщо розбір пройшов успішно, але значення виходить за допустимі межі, модуль фіксує помилку некоректного вводу і також повертає EINVAL. Якщо ж введене значення допустиме, викликається функція nsensor_update_from_raw, яка оновлює поле raw,

обчислює value, встановлює стан ready, очищає код помилки і оновлює часову мітку.

У реалізованому модулі нормалізацію виконано за формулою value = raw * 2. Такий варіант був обраний як контрольована логіка для перевірки коректності оновлення метрик.

Для обробки помилок використано допоміжну функцію nsensor_set_error. Вона переводить модуль у стан error, встановлює відповідний код помилки і оновлює поле last_update. В такому разі модуль зберігає останні коректні значення raw і value, але водночас показує, що останній цикл завершився помилкою.

Журналювання реалізовано через pr_info і pr_err. У журналі ядра фіксуються події завантаження модуля, створення sysfs-шляху, успішного приймання тестового сирого значення, помилки некоректного діапазону та помилки розбору вхідних даних.

Інтеграція з udev виконана на базовому рівні. Для цього створено правило 99-nonstandard-sensor.rules, яке орієнтується на підсистему nonstandard_sensor і пристрій nonstandard_sensor0.

Для взаємодії з модулем у просторі користувача було підготовлено два shell-скрипти. Перший скрипт read_metrics.sh реалізує просте читання атрибутів value, raw, status, error_code і last_update з каталогу /sys/class/nonstandard_sensor/nonstandard_sensor0. Другий скрипт test_sensor.sh виконує серію автоматизованих перевірок. У ньому послідовно тестуються три сценарії: коректне введення значення, введення числа поза допустимим діапазоном і введення нечислового рядка.

3.2 Засоби взаємодії з модулем у просторі користувача

Після реалізації програмного модуля ядра Linux було розроблено засоби взаємодії з ним у просторі користувача. Оскільки взаємодія не передбачає графічного інтерфейсу, основну увагу зосереджено на системних засобах Linux,

читанню атрибутів sysfs, використанню правил udev і простим shell-скриптам для контролю стану модуля.

Головним механізмом взаємодії з модулем у просторі користувача є каталог /sys/class/nonstandard_sensor/nonstandard_sensor0/ (рисунок 3.5).

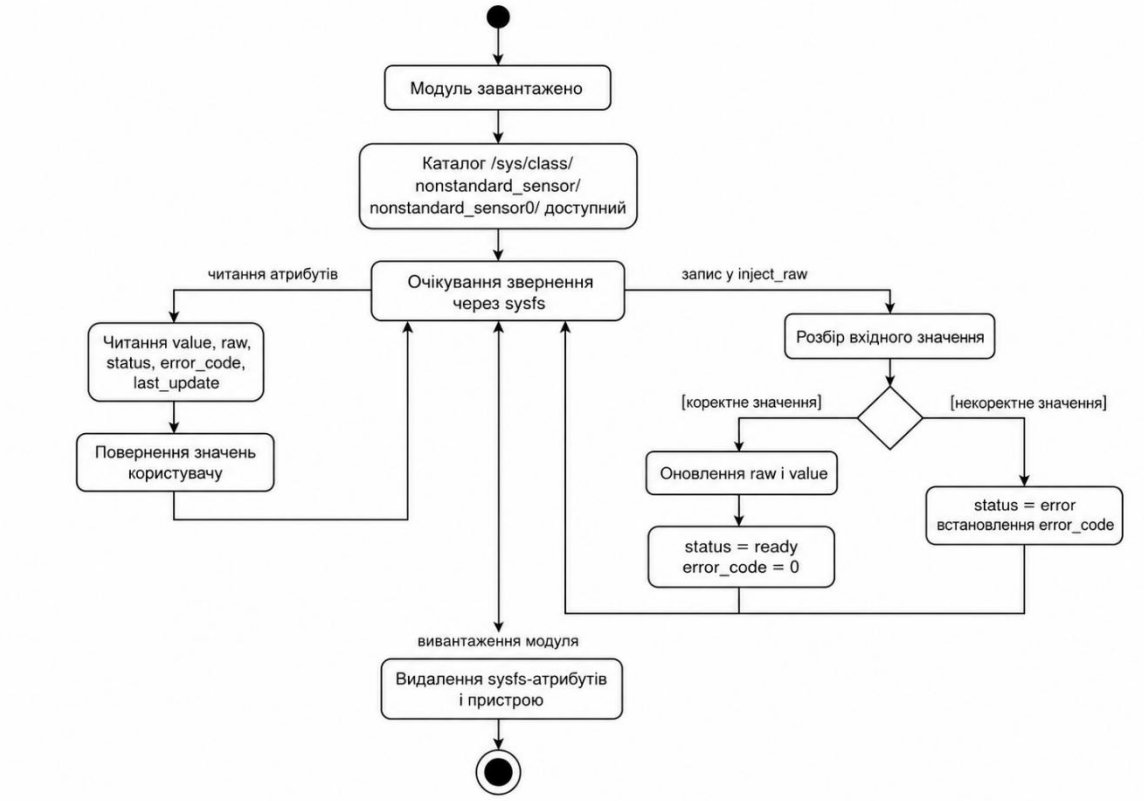


Рисунок 3.5 – Діаграма станів взаємодії користувача з модулем через sysfs

Через нього користувач або скрипт може отримати доступ до даних, які формує модуль ядра. Також були створені атрибути value, raw, status, error_code, last_update та inject_raw. Перші п’ять атрибутів призначені для читання стану пристрою, а атрибут inject_raw використовується як службовий засіб подання тестових вхідних даних у модуль.

Читання атрибутів sysfs здійснюється стандартними засобами командного рядка, зокрема через cat. Такий спосіб є зручним під час налагодження, оскільки дозволяє безпосередньо перевіряти поточне значення метрики, сирі дані, стан модуля та код помилки. В результаті користувач бачить уже підготовлені значення, а не сирі дані модуля.

Поділ атрибутів на атрибути читання і запису робить роботу з модулем безпечнішою та передбачуваною.

Можливість подання тестових даних в просторі користувача реалізовано через inject_raw. Завдяки цьому атрибуту стало можливим моделювати як коректне надходження даних, так і різні типи помилок. Цей атрибут у поточній реалізації виконує роль тестового каналу взаємодії з модулем.

Для спрощення читання основних метрик в просторі користувача було створено shell-скрипт read_metrics.sh. (рисунок 3.6)

```
#!/bin/bash

BASE_PATH="/sys/class/nonstandard_sensor/nonstandard_sensor0"

echo "=== NONSTANDARD SENSOR METRICS ==="

if [ ! -d "$BASE_PATH" ]; then
    echo "Device path not found: $BASE_PATH"
    exit 1
fi

echo "value:       $(cat "$BASE_PATH/value")"
echo "raw:         $(cat "$BASE_PATH/raw")"
echo "status:      $(cat "$BASE_PATH/status")"
echo "error_code:  $(cat "$BASE_PATH/error_code")"
echo "last_update: $(cat "$BASE_PATH/last_update")"
```

Рисунок 3.6 – Скрипт read_metrics.sh

Його призначення — зібрати основні атрибути в одному виводі та подати їх у зручному для перегляду вигляді. Скрипт звертається до каталогу /sys/class/nonstandard_sensor/nonstandard_sensor0/, перевіряє наявність пристрою і далі послідовно зчитує value, raw, status, error_code і last_update. В результаті користувач отримує готовий короткий звіт про поточний стан модуля без потреби вручну вводити кілька окремих команд.

Крім засобів ручного читання, було реалізовано правило udev, яке забезпечує реакцію системи на події, пов’язані з підсистемою nonstandard_sensor. В реалізованому варіанті цей механізм реалізовано як окремий shell-скрипт (рисунок 3.7).

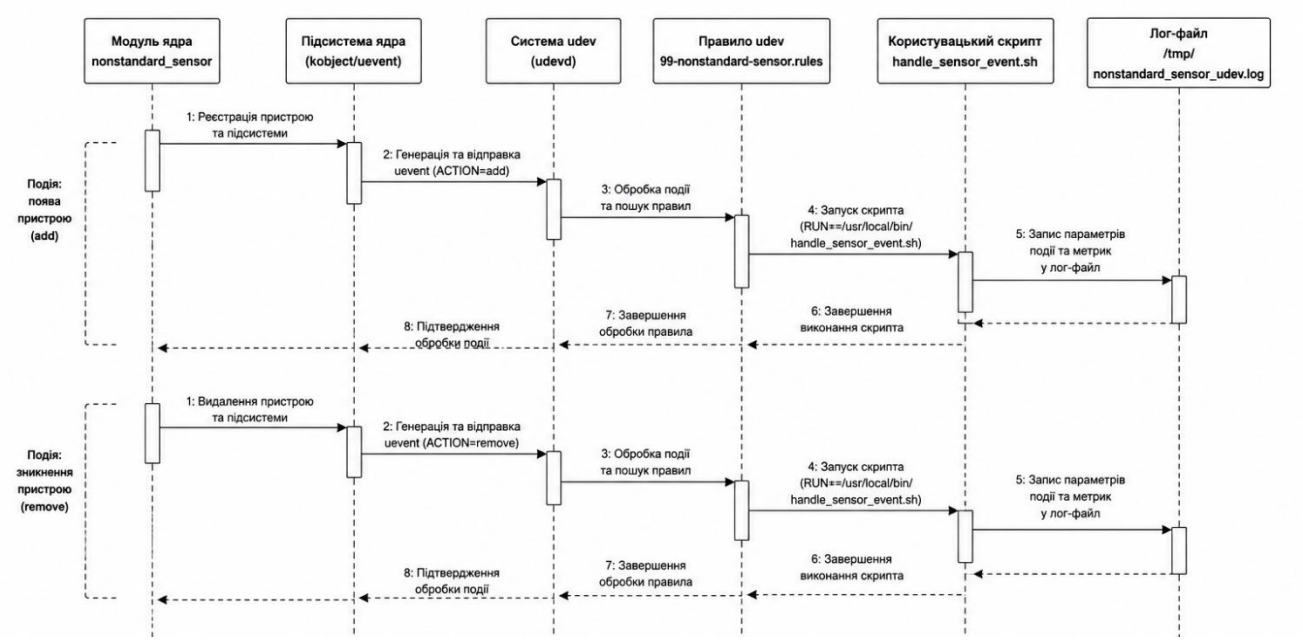


Рисунок 3.7 – Взаємодія ядра, udev і користувацьких скриптів

Завдяки цьому події ядра можна пов’язати з діями користувацьких скриптів.

Правило 99-nonstandard-sensor.rules орієнтується на підсистему nonstandard_sensor і пристрій nonstandard_sensor0.

```
SUBSYSTEM=="nonstandard_sensor",
KERNEL=="nonstandard_sensor0",
MODE="0444",
RUN+="/home/----/nonstandard_sensor_module/handle_sensor_event.sh"
```

Рисунок 3.8 – Скрипт реалізації правила 99-nonstandard-sensor.rules

Після його встановлення в каталог /etc/udev/rules.d/ і перезавантаження правил система починає враховувати це правило під час обробки відповідних подій.

Для автоматичної реакції на події був створений скрипт handle_sensor_event.sh.

```
#!/bin/bash
LOG_FILE="/tmp/nonstandard_sensor_udev.log"
BASE_PATH="/sys/class/nonstandard_sensor/nonstandard_sensor0"
{
    echo "=== EVENT ==="
    date
    echo "ACTION=$ACTION"
    echo "DEVPATH=$DEVPATH"
    echo "SUBSYSTEM=$SUBSYSTEM"
    if [ -d "$BASE_PATH" ]; then
        echo "value=$(cat "$BASE_PATH/value" 2>/dev/null)"
        echo "raw=$(cat "$BASE_PATH/raw" 2>/dev/null)"
        echo "status=$(cat "$BASE_PATH/status" 2>/dev/null)"
        echo "error_code=$(cat "$BASE_PATH/error_code" 2>/dev/null)"
        echo "last_update=$(cat "$BASE_PATH/last_update" 2>/dev/null)"
    else
        echo "device path not available"
    fi
    echo
} >> "$LOG_FILE"
```

Рисунок 3.9 – Скрипт handle_sensor_event.sh

Скрипт під час події зчитує основні параметри модуля та записує їх у лог-файл /tmp/nonstandard_sensor_udev.log. Скрипт фіксує тип події, системний шлях до пристрою, значення змінних ACTION, DEVPATH, SUBSYSTEM, а також намагається зчитати value, raw, status, error_code і last_update. Якщо на момент обробки події каталог пристрою вже відсутній, у лог записується повідомлення про недоступність шляху.

Ще одним засобом взаємодії з модулем став тестовий сценарій test_sensor.sh.

```
#!/bin/bash
BASE_PATH="/sys/class/nonstandard_sensor/nonstandard_sensor0"
echo "=== TEST 1: valid input ==="
echo 100 > "$BASE_PATH/inject_raw"
echo "raw=$(cat "$BASE_PATH/raw")"
echo "value=$(cat "$BASE_PATH/value")"
echo "status=$(cat "$BASE_PATH/status")"
echo "error_code=$(cat "$BASE_PATH/error_code")"
echo
echo "=== TEST 2: invalid range ==="
if echo 2001 > "$BASE_PATH/inject_raw"; then
    echo "unexpected success"
else
    echo "write failed as expected"
fi
echo "raw=$(cat "$BASE_PATH/raw")"
echo "value=$(cat "$BASE_PATH/value")"
echo "status=$(cat "$BASE_PATH/status")"
echo "error_code=$(cat "$BASE_PATH/error_code")"
echo

echo "=== TEST 3: parse error ==="
if echo abc > "$BASE_PATH/inject_raw"; then
    echo "unexpected success"
else
    echo "write failed as expected"
fi
echo "raw=$(cat "$BASE_PATH/raw")"
echo "value=$(cat "$BASE_PATH/value")"
echo "status=$(cat "$BASE_PATH/status")"
echo "error_code=$(cat "$BASE_PATH/error_code")"
```

Рисунок 3.10 – Скрипт test_sensor.sh

Він відрізняється від read_metrics.sh, тим що перший виконує тільки читання, а новий призначений для перевірки кількох режимів роботи модуля. В ньому реалізовано подання коректного значення в inject_raw, перевірка результату, подання значення поза допустимим діапазоном і подання нечислового рядка. Після кожної такої дії скрипт зчитує raw, value, status і error_code, що дозволяє побачити, як саме модуль реагує на різні типи вводу.

3.3 Тестування розробленого модуля

Після реалізації модуля ядра Linux, налаштування sysfs, інтеграції з udev і створення допоміжних користувацьких засобів було проведено тестування розробленого рішення, було проведено перевірку працездатності модуля в умовах реального виконання, оцінювання правильності експорту метрик, перевірку обробки коректних і некоректних даних.

Тестування було розділено на декілька основних груп. До першої групи належать перевірки, пов’язані зі складанням і базовим запуском модуля. До другої групи належать перевірки правильності створення sysfs-інтерфейсу та читання метрик. До третьої групи входять сценарії функціональної перевірки на коректних і некоректних вхідних даних. До четвертої групи належить перевірка інтеграції з udev. До п’ятої групи належить перевірка консольного інтерфейсу користувача, який працює поверх sysfs-атрибутів модуля.

Першим етапом тестування була перевірка можливості складання модуля в системі Linux. Для цього використовувалися Makefile, встановлені заголовкові файли ядра та стандартні інструменти компіляції. Після виконання команди складання було сформовано файл nonstandard_sensor.ko, (рисунок 3.11) що підтвердило коректність побудови структури проекту та відсутність помилок у коді.

```
root@Debain-   :/home/   /nonstandard_sensor_module# make
make -C /lib/modules/6.12.63+deb13-amd64/build M=/home/   ɔ/nonstandard_sensor_module modules
make[1]: Entering directory '/usr/src/linux-headers-6.12.63+deb13-amd64'
  CC [M]  /home/oso/nonstandard_sensor_module/nonstandard_sensor.o
  MODPOST /home/oso/nonstandard_sensor_module/Module.symvers
  CC [M]  /home/oso/nonstandard_sensor_module/nonstandard_sensor.mod.o
  CC [M]  /home/oso/nonstandard_sensor_module/.module-common.o
  LD [M]  /home/oso/nonstandard_sensor_module/nonstandard_sensor.ko
  BTF [M] /home/oso/nonstandard_sensor_module/nonstandard_sensor.ko
make[1]: Leaving directory '/usr/src/linux-headers-6.12.63+deb13-amd64'
```

Рисунок 3.11 – Результат складання модуля

Другим етапом була перевірка завантаження і вивантаження модуля. Після виконання команди insmod модуль з’являвся у списку завантажених модулів, а в журналі ядра з’являлося повідомлення про успішне завантаження.

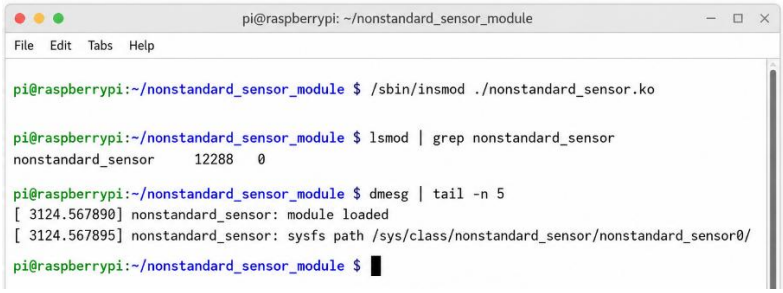


Рисунок 3.12 – Результат завантаження модуля

Додатково модуль виводив шлях до sysfs-каталогу, через який далі здійснювався доступ до метрик. Перевірка вивантаження через rmmod показала, що модуль коректно завершує роботу, виводить повідомлення про вивантаження і після цього зникає зі списку завантажених модулів.

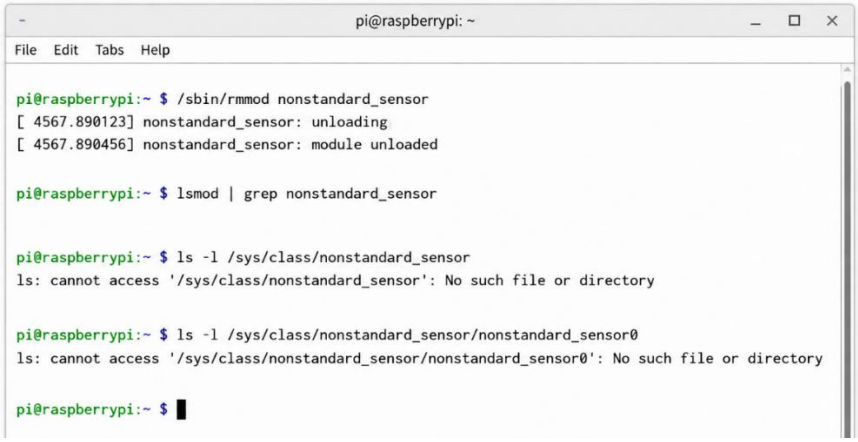


Рисунок 3.13 – Результат вивантаження модуля

Після вивантаження каталог /sys/class/nonstandard_sensor ставав недоступним, що підтверджує коректне звільнення створених системних об’єктів.

Далі була перевірка sysfs-інтерфейсу. Після завантаження модуля в системі створювався каталог /sys/class/nonstandard_sensor/nonstandard_sensor0/, в якому розміщувалися атрибути value, raw, status, error_code, last_update та inject_raw.

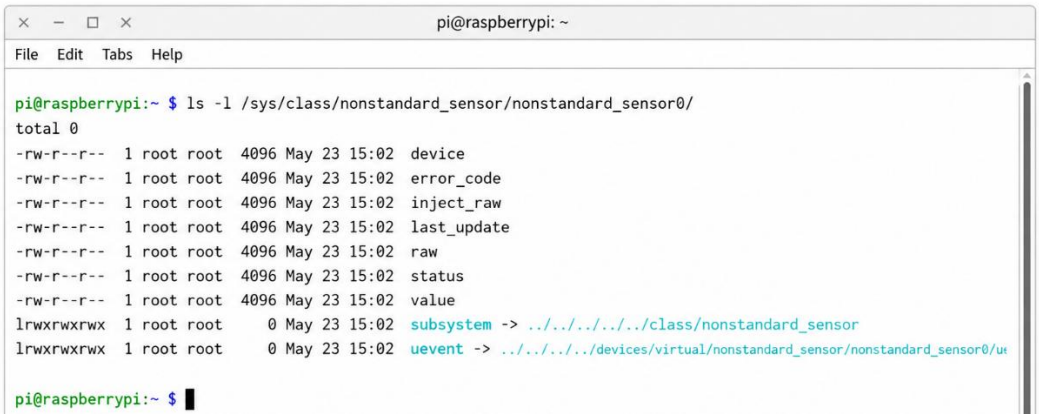


Рисунок 3.13 – Результат перевірки sysfs-атрибутів модуля

Перевірка значень атрибутів у початковому стані показала, що модуль після завантаження встановлює value = 0, raw = 0, status = ready, error_code = 0, а також фіксує часову мітку останнього оновлення.

Основним функціональним тестом була перевірка коректного запису тестового сирого значення через атрибут inject_raw. Під час введення допустимого числового значення модуль зчитував його, виконував перевірку коректності, оновлював поле raw, формував нове значення value за формулою нормалізації та переводив свій стан у ready. Перевірка після введення числа 123 показала, що raw приймає значення 123, а value обчислюється як 246.

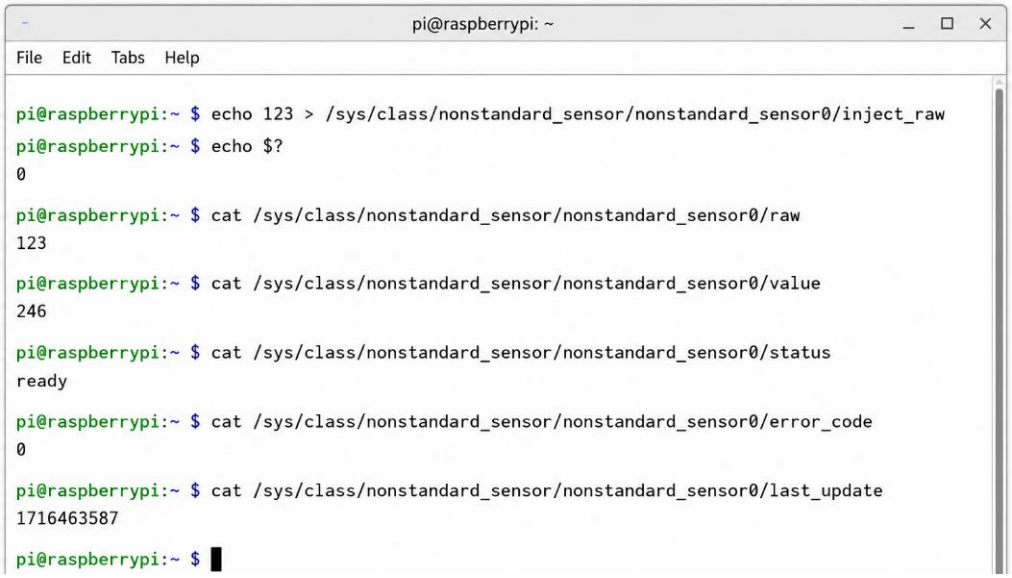


Рисунок 3.14 – Результат коректного запису тестового значення в атрибут

Наступний етап тестування стосувався обробки некоректних вхідних даних. Спочатку було перевірено сценарій, коли в inject_raw подається число, що виходить за допустимий діапазон. В реалізованому модулі межі допустимого значення були встановлені від 0 до 1000. Після спроби запису числа 2001 система повертала помилку запису, статус пристрою переходив у стан error, а код помилки встановлювався в значення, що відповідає некоректному вводу.

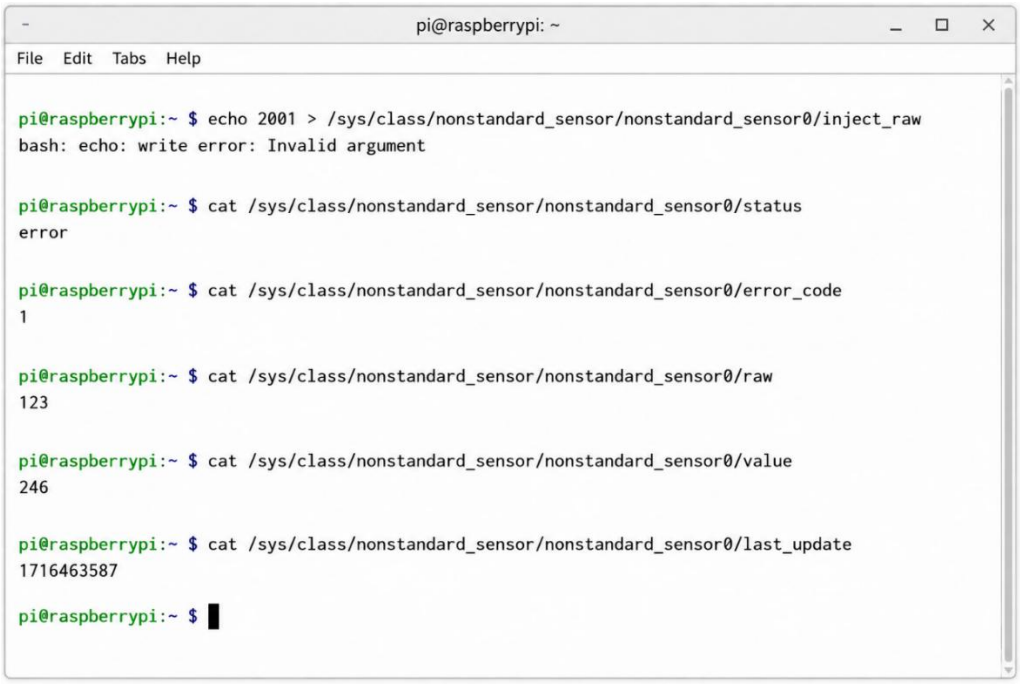


Рисунок 3.15 – Результат перевірки помилки виходу за допустимий діапазон

При цьому попередні коректні значення не змінювалися.

Далі було перевірено сценарій помилки розбору, коли в атрибут inject_raw подається нечисловий рядок. У цьому випадку модуль також повертав помилку запису, переводив себе у стан error, але вже з іншим кодом помилки, який відповідає неможливості розбору введеного значення.

Рисунок 3.16 – Результат перевірки помилки розбору вхідного значення

Перевірка показала, що цей сценарій також обробляється правильно. В журналі ядра з’являлося повідомлення про помилку розбору.

Наступним етапом стала перевірка взаємодії з udev. Для цього було створене правило 99-nonstandard-sensor.rules, яке реагувало на події, пов’язані з підсистемою nonstandard_sensor, і запускало скрипт handle_sensor_event.sh. Перевірка за допомогою udevadm підтвердила, що правило зчитується системою і застосовується до створеного пристрою. Практично це перевірялося шляхом вивантаження та повторного завантаження модуля. В результаті в лог-файлі /tmp/nonstandard_sensor_udev.log (рисунок 3.17) фіксувалися події типу remove і add, а разом із ними основні метрики пристрою.

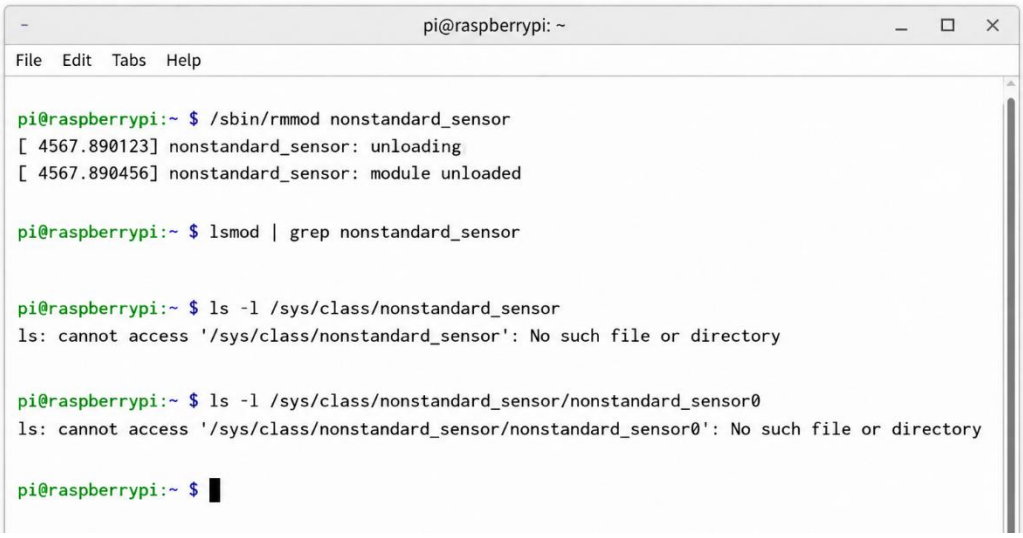


Рисунок 3.17 – Результат спрацювання udev та формування запису у /tmp/nonstandard_sensor_udev.log

Окремо тестувались shell-скрипти, які були створені як допоміжні засоби взаємодії з модулем. Скрипт read_metrics.sh перевіряв можливість зручного читання основних sysfs-атрибутів у просторі користувача. Скрипт test_sensor.sh використовувався як сценарій автоматизованої перевірки. Він послідовно виконував тест коректного значення, тест виходу за допустимий діапазон і тест помилки розбору.

Окремою частиною тестування стала перевірка консольного інтерфейсу користувача, реалізованого у файлі sensor_tui.py. Інтерфейс було побудовано на основі бібліотеки Textual і розгорнуто у віртуальному середовищі Python. Його призначення — надати наочніший спосіб роботи з модулем у просторі користувача. В процесі перевірки було підтверджено, що інтерфейс коректно визначає наявність sysfs-каталогу модуля, відображає значення value, raw, status, error_code і last_update, дозволяє вводити тестові значення для атрибута inject_raw, а також показує журнал подій udev.

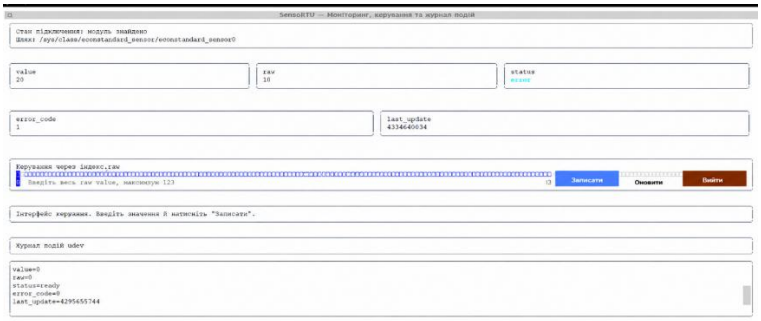


Рисунок 3.18 – Початковий стан консольного інтерфейсу користувача для роботи з модулем

Під час роботи з TUI були перевірені два типи сценаріїв. В першому сценарії вводилось коректне значення, після чого інтерфейс одразу оновлював raw, value, status і error_code.

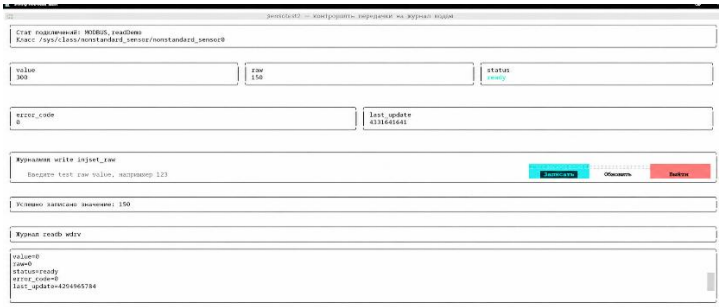


Рисунок 3.19 – Консольний інтерфейс користувача після коректного введення тестового значення

В другому сценарії вводилось некоректне значення, яке призводило до переходу модуля в стан error, що одразу відображалося у TUI.

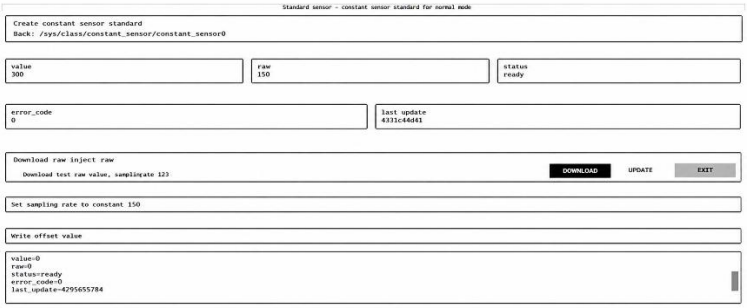


Рисунок 3.20 – Консольний інтерфейс користувача при помилковому введенні тестового значення

Результати тестування підтвердили коректну роботу розробленого модуля ядра Linux. Було підтверджено можливість його складання, завантаження та вивантаження, створення sysfs-атрибутів, коректного оновлення метрик, обробки помилок, інтеграції з udev та роботи через консольний інтерфейс користувача. Отримані результати дають підстави використовувати модуль як основу для підключення реального нестандартного Arduino-сумісного сенсора.

ВИСНОВКИ

Під час виконання кваліфікаційної роботи було розроблено програмний модуль ядра Linux для нестандартного Arduino-сумісного сенсора, орієнтований на використання у вбудованих системах на базі Raspberry Pi.

Отримано такі основні результати:

1. Проведено аналіз предметної області драйверів ядра Linux для вбудованих сенсорів.

2. Досліджено особливості програмної взаємодії Linux із зовнішніми сенсорами. Встановлено, що для нестандартних Arduino-сумісних пристроїв використання користувацьких скриптів не забезпечує достатньої стабільності, контрольованості та системної інтеграції.

3. Проаналізовано існуючі підходи до підтримки сенсорів в Linux, зокрема використання sysfs, udev, Industrial I O та Hardware Monitoring. Визначено, що стандартні підсистеми ефективні для типових сенсорів, але для нестандартного сенсора часто потрібна власна реалізація модуля з індивідуальною логікою обробки даних.

4. Сформульовано постановку задачі та визначено основні вимоги до програмного модуля.

5. Розроблено загальну структуру проектованого модуля та визначено його місце в системі Raspberry Pi.

6. Описано алгоритм функціонування модуля, який охоплює етапи ініціалізації, реєстрації пристрою, зчитування і перевірки даних, нормалізації сирих значень, оновлення sysfs-атрибутів, формування подій для udev, обробки аварійних ситуацій та вивантаження модуля.

7. Розроблено інформаційне забезпечення модуля. Визначено склад вхідної та вихідної інформації, службові параметри, вихідні метрики, статусні повідомлення та коди помилок.

8. Виконано програмну реалізацію модуля ядра Linux мовою C. Створено основний файл модуля, заголовковий файл зі структурами та службовими

константами, Makefile для складання, правило udev, shell-скрипти для читання метрик і тестування, а також консольний інтерфейс користувача на Python із використанням Textual.

9. Реалізовано sysfs-інтерфейс модуля, який надає доступ до атрибутів value, raw, status, error_code, last_update та inject_raw.

10. Проведено тестування розробленого модуля. Під час тестування підтверджено можливість складання модуля, його завантаження та вивантаження, створення sysfs-атрибутів, коректного оновлення метрик при допустимих вхідних значеннях, обробки помилок виходу за діапазон і помилок розбору, а також спрацювання udev та формування журналу подій.

11. Перевірено роботу консольного інтерфейсу користувача, який відображає поточний стан модуля, дозволяє вводити тестові значення в inject_raw, відстежує зміну метрик і показує журнал подій.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Introduction [Електронний ресурс] // The Linux Kernel documentation. — Режим доступу: https://docs.kernel.org/driver-api/iio/intro.html (дата звернення: 11.03.2026).

2. Core elements [Електронний ресурс] // The Linux Kernel documentation. — Режим доступу: https://docs.kernel.org/driver-api/iio/core.html (дата звернення: 11.03.2026).

3. Implementing I2C device drivers [Електронний ресурс] // The Linux Kernel documentation. — Режим доступу: https://docs.kernel.org/i2c/writing-clients.html (дата звернення: 11.03.2026).

4. GPIO Character Device Userspace API [Електронний ресурс] // The Linux Kernel documentation. — Режим доступу: https://docs.kernel.org/userspace-api/gpio/chardev.html (дата звернення: 11.03.2026).

5. SPI userspace API [Електронний ресурс] // The Linux Kernel documentation. — Режим доступу: https://docs.kernel.org/spi/spidev.html (дата звернення: 11.03.2026).

6. Raspberry Pi computer hardware [Електронний ресурс] // Raspberry Pi Documentation. — Режим доступу: https://www.raspberrypi.com/documentation/computers/raspberry-pi.html (дата звернення: 11.03.2026).

7. Vélez D., Martínez Madrid N., Seepold R. Software Scripts for Sensor Data Extraction in Rasberry Pi: user-space and kernel-space comparison [Електронний ресурс] // *Models and Applications for Embedded Systems*. — 2023. — P. 5–9. — Режим доступу: https://www.researchgate.net/profile/Daniel-Velez-16/publication/381401693_Software_Scripts_for_Sensor_Data_Extraction_in_Rasberry_Pi_user-space_and_kernel-space_comparison/links/666b3bebb769e769193385b7/Software-Scripts-for-Sensor-Data-Extraction-in-Rasberry-Pi-user-space-and-kernel-space-comparison.pdf (дата звернення: 11.03.2026).

8. Яременко А. В. Термочутливий елемент датчика температури [Електронний ресурс] : бакалаврська робота. — Київ, 2024. — Режим доступу: https://ela.kpi.ua/bitstreams/2dc93e52-b0dc-4bf0-ae54-710a18e6cc7d/download (дата звернення: 11.03.2026).

9. Rantalainen A. A. Enhancing Intelligence in Test Systems' Power Distribution Units [Електронний ресурс] : bachelor's thesis. — 2024. — Режим доступу: https://www.theseus.fi/bitstream/handle/10024/877721/Rantalainen_Amos.pdf?sequence=2&isAllowed=y (дата звернення: 11.03.2026).

10. Thapa S., K. C. S. C. Raspberry Pi and ESP32-Based Smart Sensor Network for IoT Platform Integration and Real-Time Environmental Data Monitoring [Електронний ресурс] : bachelor's thesis. — 2023. — Режим доступу: https://www.theseus.fi/bitstream/handle/10024/813437/Thapa_KC.pdf?sequence=2&isAllowed=y (дата звернення: 11.03.2026).

11. Dimovski D., Hammargren Andersson J. Validation of a real-time automated production-monitoring system [Електронний ресурс] : thesis. — 2021. — Режим доступу: https://www.diva-portal.org/smash/get/diva2:1568657/FULLTEXT02.pdf (дата звернення: 11.03.2026).

12. Niehaus K. A., Westhoff A. An open-source data acquisition system for laboratory and industrial scale applications [Електронний ресурс] // Measurement Science and Technology. — 2023. — Vol. 34, no. 2. — Art. 027001. — Режим доступу: https://elib.dlr.de/186191/2/ELIB-Eintrag-2022-NiehausK-186191-PaperPublished.pdf (дата звернення: 11.03.2026).

13. Soto-Ocampo R., Múnera M., Cañón C. et al. Low-Cost, High-Frequency, Data Acquisition System for Condition Monitoring of Rotating Machinery through Vibration Analysis—Case Study [Електронний ресурс] // *Sensors*. — 2020. — Vol. 20, no. 12. — Art. 3493. — Режим доступу: https://www.mdpi.com/1424-8220/20/12/3493 (дата звернення: 11.03.2026).

14. Pham Q.-Q., Ta Q.-B., Park J.-H., Kim J.-T. Raspberry Pi Platform Wireless Sensor Node for Low-Frequency Impedance Responses of PZT Interface [Електронний

ресурс] // *Sensors*. — 2022. — Vol. 22, no. 24. — Art. 9592. — Режим доступу: https://www.mdpi.com/1424-8220/22/24/9592 (дата звернення: 11.03.2026).

15. Калюжна В. В. Інтелектуальна інформаційно-вимірювальна система агромоніторингу [Електронний ресурс] : магістерська дисертація. — Київ, 2020. — Режим доступу: https://ela.kpi.ua/bitstream/123456789/38279/1/Kaliuzhna_magistr.pdf (дата звернення: 11.03.2026).

16. The Linux Hardware Monitoring kernel API [Електронний ресурс] // The Linux Kernel documentation. — Режим доступу: https://docs.kernel.org/hwmon/hwmon-kernel-api.html (дата звернення: 11.03.2026).

17. The Basic Device Structure [Електронний ресурс] // The Linux Kernel documentation. — Режим доступу: https://docs.kernel.org/driver-api/driver-model/device.html (дата звернення: 11.03.2026).

18. Niemi K. Automated Linux Driver Test System [Електронний ресурс] : bachelor’s thesis. — 2025. — Режим доступу: https://www.theseus.fi/bitstream/10024/880244/2/Kalle_Niemi.pdf (дата звернення: 11.03.2026).

19. Naming and data format standards for sysfs files [Електронний ресурс] // The Linux Kernel documentation. — Режим доступу: https://docs.kernel.org/hwmon/sysfs-interface.html (дата звернення: 11.03.2026).

20. LinuxDriverIIO [Електронний ресурс] // STMicroelectronics. — Режим доступу: https://www.st.com/en/embedded-software/linuxdriveriio.html (дата звернення: 11.03.2026).

21. Müller M. An Embedded Linux Solution to Monitor Laboratory Environmental Parameters [Електронний ресурс] : semester thesis. — Zürich, 2022. — Режим доступу: https://ethz.ch/content/dam/ethz/special-interest/phys/quantum-electronics/tiqi-dam/documents/semester_theses/semesterthesis-Mose-Mueller.pdf (дата звернення: 11.03.2026).

22. udev(7) [Електронний ресурс] // Linux manual page. — Режим доступу: https://man7.org/linux/man-pages/man7/udev.7.html (дата звернення: 11.03.2026).

23. Stankus L. P. Testing Linux Drivers with Device Emulation Aid [Електронний ресурс] : bachelor capstone project report. — São Paulo, 2021. — Режим доступу: https://bcc.ime.usp.br/tccs/2021/lpstankus/monografia.pdf (дата звернення: 11.03.2026).

24. Mei R., Xiao L. Design on Linux Platform Driver for Embedded Systems [Електронний ресурс] // American Journal of Embedded Systems and Applications. — 2022. — Vol. 9, no. 1. — P. 1–5. — Режим доступу: https://article.sciencepublishinggroup.com/pdf/ajesa.20220901.11 (дата звернення: 11.03.2026).

25. Guégan L., Tofaily S., Raïs I. Design and Evaluation of Single-Board Computer Based Power Monitoring for IoT and Edge Systems [Електронний ресурс] // *GreenCom* 2023. — 2023. — Режим доступу: https://www.cs.uit.no/~issam/Papers/2023/TestbedEnergyMonitoring_GreenCom2023.pdf (дата звернення: 11.03.2026).

26. Libiio [Електронний ресурс] // System Level Documentation. — Режим доступу: https://analogdevicesinc.github.io/documentation/software/libiio/index.html (дата звернення: 11.03.2026).

27. Libiio command line utility [Електронний ресурс] // System Level Documentation. — Режим доступу: https://analogdevicesinc.github.io/documentation/software/libiio/cli.html (дата звернення: 11.03.2026).

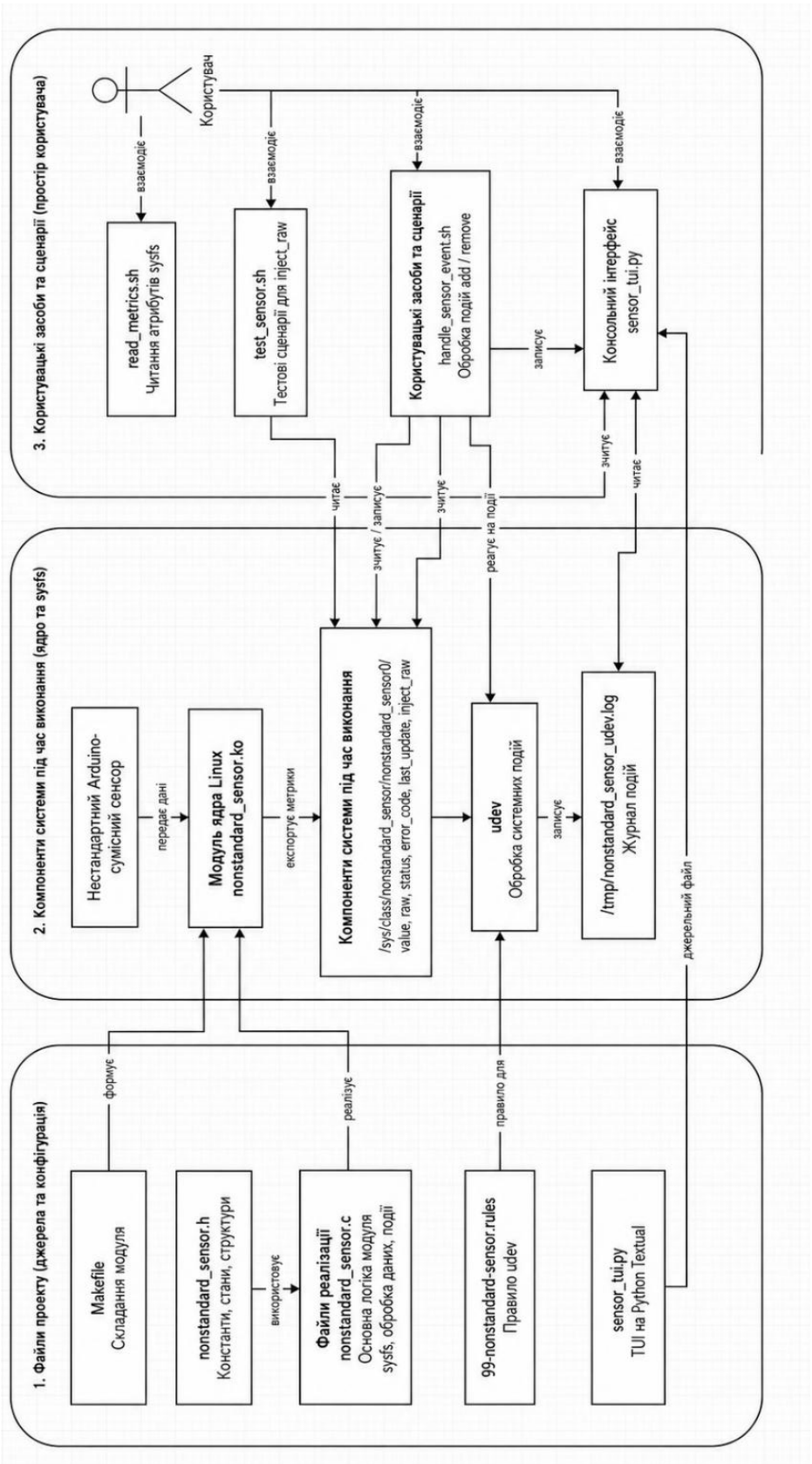
28. Комар М.П., Саченко А.О., Васильків Н.М., Гладій Г.М., Коваль В.С., Ліп’яніна-Гончаренко Х.В. Методичні рекомендації до виконання кваліфікаційної роботи з освітньо-професійної програми «Комп’ютерні науки» спеціальності 122 «Комп’ютерні науки» за першим (бакалаврським) рівнем вищої освіти. Тернопіль: ЗУНУ, 2024. 52 с.

29. Островерхов В. М., Біловус Л. І., Возьний К. З., Луцишин О. О., Монастирський Г. Л., Надвиничний С. А., Питель С. В., Шандрук С. К. Загальні методичні рекомендації з підготовки, оформлення, захисту та оцінювання

кваліфікаційних робіт здобувачів вищої освіти першого (бакалаврського) і другого (магістерського) рівнів. Тернопіль: ЗУНУ, 2024. 83 с.

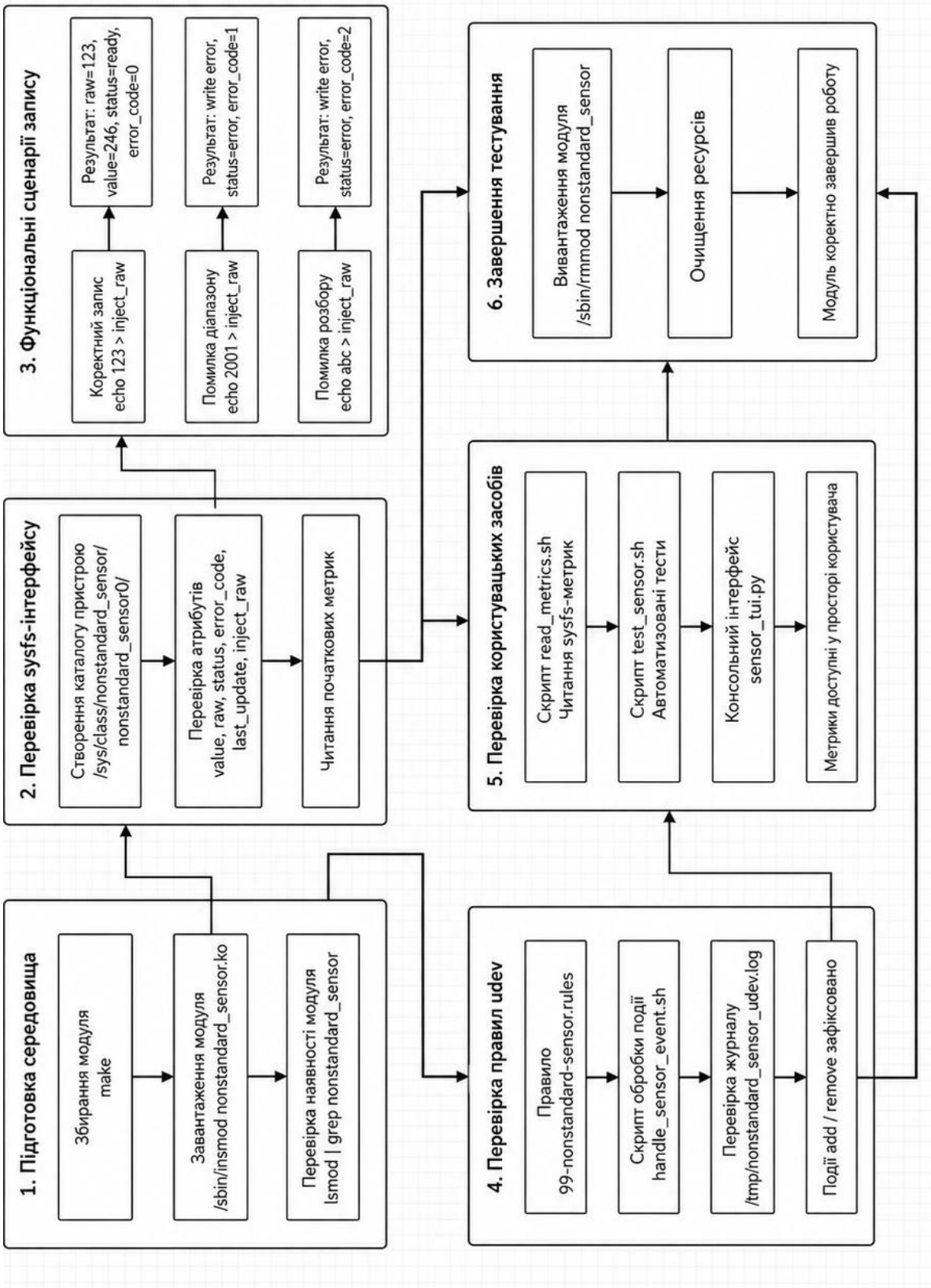
Додаток А

Структурна карта файлів і компонентів програмної реалізації модуля



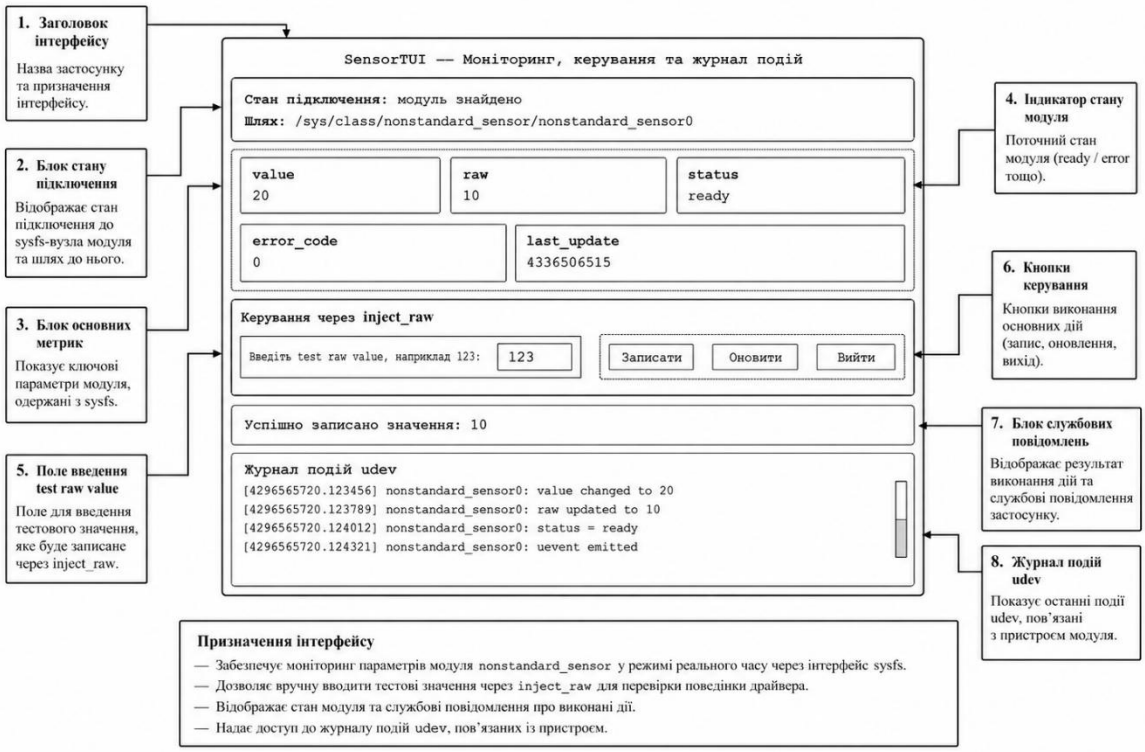
Додаток Б

Візуальна карта сценаріїв тестування



Додаток В

Схема консольного інтерфейсу користувача модуля



Додаток Г
Апробація отриманих результатів

ДОВІДКА

ПРО ПРИЙНЯТТЯ ТЕЗ ДО ПУБЛІКАЦІЇ

11.06.2026

Шановний(і) авторе(и):
Гулько Юлія Віталіївна,

Організаційний комітет з радістю повідомляє, що тези «ПРОГРАМНИЙ МОДУЛЬ ЯДРА LINUX ДЛЯ НЕСТАНДАРТНОГО ARDUINO-СУМІСНОГО СЕНСОРА» прийняті до публікації в збірнику за матеріалами IX Всеукраїнської студентської наукової конференції «Науковий простір: аналіз, сучасний стан, тренди та перспективи» (19.06.2026, м. Київ, Україна).

Опубліковані тези будуть доступні з 19.06.2026 за посиланням:
https://archive.liga.science/index.php/conference-proceedings/issue/view/ukr-19.06.2026

. . . . . .

Електронні сертифікати учасників конференції та подяки науковим керівникам будуть доступні з 19 червня. Розсилка замовлених друкованих примірників, сертифікатів та подяк відбудеться до 3 липня.

Конференцію зареєстровано Державною науковою установою «УкрІНТЕІ» в базі даних науково-технічних заходів України та інформаційному бюлетені «План проведення наукових, науково-технічних заходів в Україні» (Посвідчення № 124 від 26.01.2026).

З повагою,

Директор Молодіжної наукової ліги
Голова оргкомітету конференції
ІГОР КОРЕНЮК

ПРОГРАМНИЙ МОДУЛЬ ЯДРА LINUX ДЛЯ НЕСТАНДАРТНОГО ARDUINO-СУМІСНОГО СЕНСОРА

Гулько Юлія Віталіївна
Студентка спеціальності 122 – Комп'ютерні науки
Західноукраїнський національний університет, Україна
Науковий керівник: Осолінський Олександр Романович
к.т.н., доцент кафедри інформаційно-обчислювальних систем і управління
Західноукраїнський національний університет, Україна

Вступ

В сучасних вбудованих системах часто використовуються одноплатні комп'ютери, мікроконтролерні вузли та зовнішні сенсори, які мають збирати і передавати дані в реальному часі. Raspberry Pi є зручною платформою для таких рішень, так як має підтримку Linux, доступні цифрові інтерфейси та можливість запуску системного і прикладного програмного забезпечення.

Типові сенсори часто підтримуються готовими драйверами або стандартними підсистемами Linux, зокрема Industrial I/O, Hardware Monitoring, sysfs та udev [1–4]. Проблема виникає тоді, коли застосовується нестандартний Arduino-сумісний сенсор або проміжний мікроконтролерний вузол з власним форматом даних, нетиповою логікою обміну чи додатковими службовими параметрами. У такій ситуації простого користувацького скрипта недостатньо, оскільки система має не тільки прийняти дані, а й перевірити їх, нормалізувати, надати стабільний інтерфейс доступу і забезпечити діагностику.

Метою роботи є розробка програмного модуля ядра Linux для нестандартного Arduino-сумісного сенсора, який працює на платформі Raspberry Pi та забезпечує приймання, обробку, нормалізацію і системне подання вимірювальних даних засобами Linux.

Архітектура та логіка роботи модуля

Запропонований модуль розглядається як проміжний рівень між апаратним сенсором і простором користувача. З боку пристрою він приймає первинні дані через відповідний цифровий інтерфейс, а з боку Linux надає вже впорядковані метрики. Це дозволяє представити сенсор для системи, як кероване джерело даних.

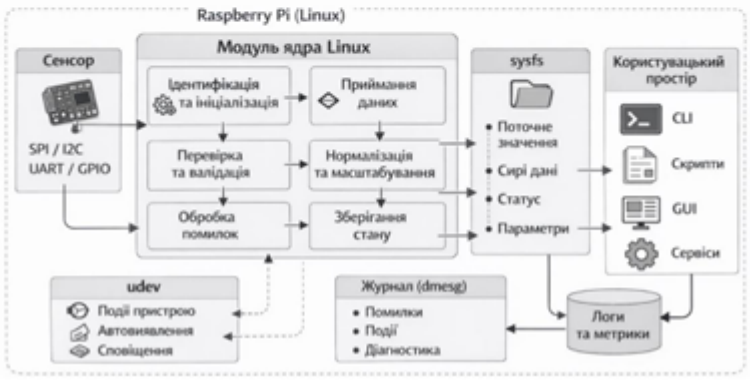


Рис. 1. **Загальна структура проектованого модуля**

Основними функціональними блоками модуля є ідентифікація та ініціалізація сенсора, приймання даних, перевірка і валідація, нормалізація значень, зберігання поточного стану та обробка помилок. Після успішного розбору повідомлення сире значення перетворюється у системну метрику, а службові параметри зберігаються у внутрішньому стані модуля. Це дає змогу розділити корисні дані, діагностичну інформацію і коди помилок.

Для експорту результатів використано sysfs. Через нього простір користувача отримує доступ до атрибутів value, raw, status, error_code і last_update. Окремий атрибут inject_raw передбачено як тестовий канал введення даних, що дає змогу перевіряти логіку модуля без підключення реального сенсора. Механізм udev використовується для реакції системи на появу, зміну стану або вилучення пристрою, а журнал ядра застосовується для фіксації діагностичних повідомлень.

Програмна реалізація

Практична реалізація виконана мовою C у вигляді зовнішнього модуля ядра Linux. Проект містить основний файл nonstandard_sensor.c, заголовковий файл nonstandard_sensor.h, Makefile, правило udev і користувацькі shell-скрипти для читання метрик та тестування. Центральною структурою є nsensor_data в якій зберігаються поточне значення, сирі дані, статус, код помилки, час останнього оновлення та службові об'єкти для взаємодії з ядром.

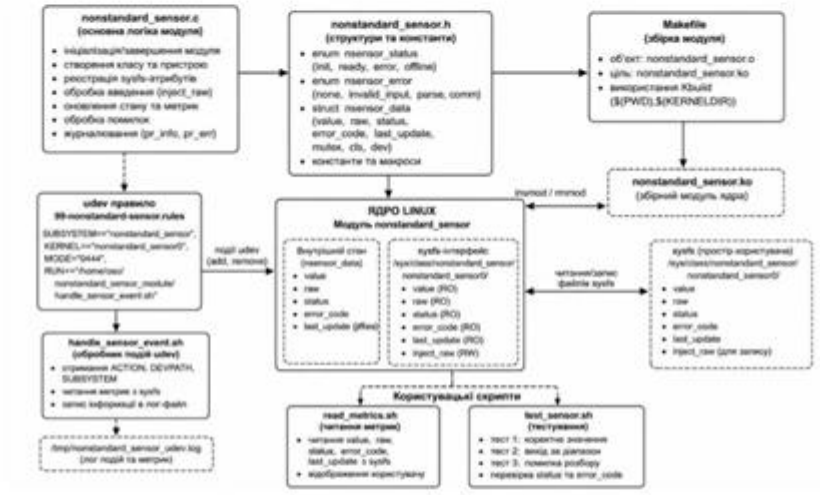


Рис. 2. **Структурна схема програмної реалізації модуля**

Під час завантаження модуль створює клас nonstandard_sensor, пристрій nonstandard_sensor0 та групу sysfs-атрибутів. Функції читання атрибутів повертають поточні значення в простому текстовому форматі, зручному для командного рядка, скриптів і служб моніторингу.

Для обробки тестових даних реалізовано функцію inject_raw_store. Вона перетворює введений рядок у число, перевіряє допустимий діапазон, оновлює raw і value або встановлює відповідний стан помилки. В реалізації нормалізація виконується за правилом value = raw * 2, що дозволяє однозначно перевірити правильність оновлення метрик. Для захисту внутрішнього стану під час одночасного читання і запису використано mutex-блокування.

Тестування розробленого модуля

Спочатку було перевірено складання модуля за допомогою Makefile та формування файлу nonstandard_sensor.ko. Далі перевірено завантаження через insmod, наявність модуля у списку активних модулів, створення каталогу /sys/class/nonstandard_sensor/nonstandard_sensor0/ та коректне вивантаження через rmmod.

Функціональна перевірка включала три основні сценарії. В першому сценарії в inject_raw передавалося коректне числове значення, після чого модуль оновлював raw, value, status і error_code. В другому сценарії подавалося число поза допустимим діапазоном, що переводило модуль у стан error без зміни попередніх коректних метрик. В третьому сценарії вводився нечисловий рядок, і модуль фіксував помилку розбору.

Висновки. Було розроблено програмний модуль ядра Linux для нестандартного Arduino-сумісного сенсора. Модуль забезпечує приймання сирих даних, їх перевірку, нормалізацію, збереження стану, експорт параметрів через sysfs, базову інтеграцію з udev та журналювання подій. Даний модуль може бути використаний як основа для підключення реального нестандартного Arduino-сумісного сенсора до Raspberry Pi та подальшого розширення системи моніторингу.

Список використаних джерел:

1. Introduction [Електронний ресурс] // The Linux Kernel documentation. – Режим доступу: https://docs.kernel.org/driver-api/iio/intro.html (дата звернення: 11.03.2026).
2. Core elements [Електронний ресурс] // The Linux Kernel documentation. – Режим доступу: https://docs.kernel.org/driver-api/iio/core.html (дата звернення: 11.03.2026).
3. The Linux Hardware Monitoring kernel API [Електронний ресурс] // The Linux Kernel documentation. – Режим доступу: https://docs.kernel.org/hwmon/hwmon-kernel-api.html (дата звернення: 11.03.2026).