Хмельницький національний університет
Факультет інформаційних технологій
Кафедра комп'ютерної інженерії та інформаційних систем

КВАЛІФІКАЦІЙНА РОБОТА

Багатозадачна операційна система реального часу для пристроїв IoT
Назва теми

Рівень вищої освіти *перший (бакалаврський)*

Галузь знань *12 «Інформаційні технології»*
Шифр, назва

Спеціальність *123 «Комп'ютерна інженерія»*
Шифр, назва

Освітня програма *«Комп'ютерна інженерія та програмування»*
Назва

Шифр КвРКІ 22005.22.01.95 ПЗ

Виконав здобувач IV курсу, група КІ2-22-1 — Підпис — Олеся БОСА (Ініціали, прізвище)

Керівник — Науковий ступінь, учене звання — Підпис — Сергій ЛИСЕНКО (Ініціали, прізвище)

Нормоконтролер — Науковий ступінь, учене звання — Підпис — Сергій ЛИСЕНКО (Ініціали, прізвище)

До захисту допускаю:
завідувач кафедри КІІС — Підпис — Ольга ПАВЛОВА (Ініціали, прізвище)
«___» _червня_ 2026 р.
дата

Хмельницький 2026

ХМЕЛЬНИЦЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

Факультет ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

Кафедра КОМП'ЮТЕРНОЇ ІНЖЕНЕРІЇ ТА ІНФОРМАЦІЙНИХ СИСТЕМ

Рівень вищої освіти ПЕРШИЙ (БАКАЛАВРСЬКИЙ)

Галузь знань 12 ІНФОРМАЦІЙНІ ТЕХНОЛОГІЇ

Спеціальність 123 КОМП'ЮТЕРНА ІНЖЕНЕРІЯ

Освітня програма «КОМП'ЮТЕРНА ІНЖЕНЕРІЯ ТА ПРОГРАМУВАННЯ»

ЗАТВЕРДЖУЮ
Завідувачка кафедри КПС
_____ Ольга ПАВЛОВА
"10" 01 2026 р.

ЗАВДАННЯ НА КВАЛІФІКАЦІЙНУ РОБОТУ

Босій Олеся Андріївні
Прізвище, ім'я, по батькові студента

1. Тема проекту (роботи) Багатозадачна операційна система реального часу для пристроїв IoT

Керівник проекту (роботи) Лисенко Сергій Миколайович, д.т.н., проф.
Прізвище, ім'я, по батькові, науковий ступінь, вчене звання

Затверджена наказом ректора університету від 20.01.2026 р. № 7

2. Термін подання здобувачем роботи на кафедру 01.06.2026 р.

3. Вихідні дані до роботи Завдання на кваліфікаційну роботу

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

Аналіз характеристик операційних систем для пристроїв IoT

Компонентне проєктування механізмів роботи операційної системи реального часу для пристроїв IoT

Системне програмне забезпечення операційної системи реального часу для пристроїв IoT

5. Перелік графічного матеріалу (із зазначенням обов'язкових креслень)

Апаратне забезпечення проєкту

Алгоритми роботи розробленої ОС реального часу

Структурна схема системного програмного забезпечення

6. Консультанти розділів кваліфікаційної роботи

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

7. Дата видачі завдання « 10 » 01 2026 р.

КАЛЕНДАРНИЙ ПЛАН

№з/п	Назва етапів (розділів) дипломного проекту (роботи)	Термін виконання етапів проекту (роботи)	Примітка
1	Вибір напряму дослідження та узгодження тематики кваліфікаційної роботи з керівником	10.01.2026	виконано
2	Ознайомлення з предметною областю; формулювання мети та задач дослідження; визначення об'єкта та предмета дослідження	01.02.2026	виконано
3	Робота над розділом 1 – дослідження предметної області та постановка задачі	01.03.2026	виконано
4	Робота над розділом 2 – вибір компонентів для проєктування операційної системи реального часу для пристроїв IoT	01.04.2026	виконано
5	Робота над розділом 3 – проєктування операційної системи реального часу для пристроїв IoT	29.04.2026	виконано
6	Оформлення пояснювальної записки згідно вимог	25.05.2026	виконано
7	Попередній захист ВКР	26.05.2026	виконано
8	Захист ВКР на засіданні ЕК	Червень 2026 року	

Здобувач _____ Олеся БОСА
Підпис Ім'я, ПРІЗВИЩЕ

Керівник кваліфікаційної роботи _____ Сергій ЛИСЕНКО
Підпис Ім'я, ПРІЗВИЩЕ

№ рядка	формат	Позначення	Найменування	Кіл. листів	№ екз	Примітка
			Текстові документи			
1		КвРКІ 22005.22.01.95 ПЗ	Пояснювальна записка	60		
			Графічні матеріали			
2		КвРКІ 22005.22.01.95 Е8	Апаратне забезпечення	1		
			проєкту			
3		КвРКІ 22005.22.01.95 Е8	Алгоритми роботи	1		
			розробленої ОС			
			реального часу			
4		КвРКІ 22005.22.01.95 Е8	Структурна схема	1		
			системного програмного			
			забезпечення			

Зм	Арк	№ докум	Підпис	Дата	КвРКІ 22005.22.01.95 ВП		
Розробив		Боса		01.06.26	Відомість проекту	Літера	Аркуш	Аркушів
Перевір.		Лисенко		01.06.26		У	1	1
Н. контр.		Лисенко		01.06.26		ХНУ, КІ2-22-1		
Затв.		Павлова		01.06.26				

АНОТАЦІЯ

Тема кваліфікаційної роботи: «Багатозадачна операційна система реального часу для пристроїв IoT».

Автор роботи: Олеся БОСА.

Керівник роботи: Сергій ЛИСЕНКО.

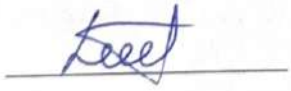
Пояснювальна записка: 60 с., 16 рис., 2 табл., 4 дод., 62 джерел.

Графічна частина: 3 креслення.

БАГАТОЗАДАЧНІСТЬ, ОПЕРАЦІЙНА СИСТЕМА, ІНТЕРНЕТ РЕЧЕЙ, М'ЮТЕКС, СЕМАФОР, АРХІТЕКТУРА, СИСТЕМА

Кваліфікаційна робота бакалавра присвячена розробці та дослідженню багатозадачної операційної системи реального часу для пристроїв Інтернету речей. Актуальність теми зумовлена потребою у створенні системного програмного забезпечення для вбудованих пристроїв, які працюють з обмеженими апаратними ресурсами, повинні вчасно реагувати на зовнішні події, підтримувати роботу з периферією, мережеву взаємодію та стабільне виконання кількох задач одночасно. Використання операційної системи реального часу дає змогу впорядкувати роботу потоків, таймерів, пам'яті, механізмів синхронізації та драйверного рівня, що є важливим для надійного функціонування IoT-пристроїв.

Метою роботи є синтез багатозадачної операційної системи реального часу для пристроїв IoT. Для досягнення поставленої мети було виконано аналіз характеристик операційних систем і пристроїв IoT, розглянуто принципи багатозадачності, планування потоків, керування пам'яттю та міжпотокової взаємодії, визначено основні компоненти системного програмного забезпечення, підготовлено апаратне середовище для тестування, виконано завантаження програмного образу та перевірено працездатність.



Підпис здобувача

30.05.2026

Дата

ЗМІСТ

ВСТУП

Сучасний розвиток інформаційних технологій пов’язаний із поширенням вбудованих обчислювальних систем, які працюють безпосередньо у фізичному середовищі і застосовуються у безлічі напрямів Інтернету речей. Для таких пристроїв характерними є обмежені апаратні ресурси, потреба в енергоефективності, постійна взаємодія з периферійними пристроями та необхідність швидкої реакції на зовнішні події.

Для розумних пристроїв важлива правильна організація системного програмного забезпечення. Потрібно враховувати, що йому потрібно одночасно обробляти сигнали від датчиків, передавати дані мережею, реагувати на переривання та виконувати різні задачі. Саме тому для розумних девайсів варто використовувати операційні систем реального часу, які забезпечують багатозадачність, підтримують планування виконання, синхронізацію, обмін повідомленнями та контроль використання пам’яті. Така ОС дозволяє впорядкувати виконання задач, забезпечити передбачувану реакцію на події і спростити роботу з периферією. Для IoT-пристроїв це має особливе значення, адже вони часто працюють у нестабільних умовах, часто мають обмежені характеристики і повинні підтримувати безпечний обмін даними.

Метою дипломної роботи є синтез операційної системи реального часу для пристроїв IoT. Поставленими задачами є дослідження архітектури та особливостей функціонування операційної системи реального часу для пристроїв Інтернету речей, а також практична перевірка її запуску, базових системних механізмів і взаємодії з апаратною платформою. Об’єктом дослідження є процес функціонування операційної системи реального часу у вбудованому пристрої Інтернету речей. Предметом дослідження є архітектура, програмні механізми та засоби практичного запуску операційної системи реального часу на мікроконтролерній платформі.

1 АНАЛІЗ ІСНУЮЧИХ РІШЕНЬ ТА ХАРАКТЕРИСТИК ОПЕРАЦІЙНИХ СИСТЕМ ДЛЯ ПРИСТРОЇВ ІОТ

1.1 Поняття та функціональні особливості

Операційна система є фактично фундаментальним комплексом системного програмного забезпечення, вона не просто управляє ресурсами, але також відповідає за взаємодію із процесором, пам’яттю і периферійним пристроями [1]. Роль операційної системи є двоякою і складною. Фактично з одного боку її можна вважати розширенням апаратної частини, але з іншого боку, це також і менеджер ресурсів. Операційна система впорядковує запити і оптимізує продуктивність системи, наприклад у випадку, коли кілька програм одночасно потребують доступу до оперативної пам’яті або процесору. Така оптимізація можлива завдяки реалізації складних алгоритмів планування та диспетчеризації [2, 5]. Ці процеси гарантують отримання критичними системними процесами пріоритету у виконанні, а користувацькі завдання виконуються без суттєвих затримок.

Функціонування операційної системи базується на тісній взаємодії з апаратним забезпеченням, а саме з центральним процесором. Однією з властивостей, що цьому відповідають, є механізм переривань та поділ режимів роботи процесора. Надійність більшості сучасних архітектур є можливою завдяки двом режимам, зокрема користувацькому і привілейованому. Привілейований режим, або режим ярда, дозволяє операційній системі мати повний доступ до всіх інструкцій процесора та апаратних ресурсів [3, 4, 7]. У випадку режиму користувача прикладні програми виконуватимуть з обмеженими правами. Узагалі якщо програмі необхідно виконати операцію, яка здійснює вплив на стан системи, наприклад, запис даних на диск, то вона здійснить запит до ОС через механізм системних викликів, а не робитиме це напряму [5, 6]. Як наслідок, такий механізм запобігає потенційному краху системи і контролює коректність запитів.

Центральною, резидентною частиною операційної системи є її ядро, яке постійно знаходить в оперативній пам’яті. Залежно від архітектури розділяють два типи ярда. При монолітному ядрі, всі основні функції, зокрема управління пам’яттю, файловою системою, драйверами, реалізовані в єдиному адресному просторі цього ядра [2, 5]. Мікроядерні ж забезпечують в привілейованому режимі лише мінімально необхідний набір функцій, а решта сервісів винесена в простір користувача. У будь-якому разі, незалежно від обраної архітектури, взаємодія з периферійними пристроями відбувається через спеціалізовані модулі, драйвери. Вони перетворюють універсальні команди операційної системи в сигнали.

Однією з найважливіших функцій операційної системи є управління процесами. Процес можна визначити як програму в стадії виконання, як активну сутність, що має лічильник команд, регістри, змінні та стек. Оскільки більшість сучасних систем є багатозадачними, ОС має створювати ілюзію одночасного виконання багатьох програм на обмеженій кількості фізичних ядер процесора. Це досягається тим, що процесор зупиняє виконання одного процесу, зберігаючи його стан, і починає інший [8, 21, 22].

Одним із компонентів ОС є планувальник процесів, який визначає, який саме процес далі отримає доступ до процесора. Алгоритмів планування є досить багато, наприклад, карусельне планування, де на кожний процес виділяється фіксований квант часу [14, 15]. Крім цього операційна система також відповідає за синхронізацію та міжпроцесорну взаємодію. Коли є декілька процесів, які одночасно звертаються до одних і тих самих даних, виникає ризик станів гонитви. Це та ситуація, де результат стає абсолютно непередбачуваним і залежить суто від того, хто встиг першим, що для стабільної системи неприпустимо. ОС пропонує певний набір інструментів синхронізації для таких ситуацій, зокрема м’ютекси, семафори чи високорівневі монітори, які допомагають програмісту розмежувати чергу та організувати безпечний доступ до розділюваних ресурсів [9]. Але тут з’являється інша загроза, а саме взаємні

блокування. Процеси просто завмирають у постійному очікуванні один на одного.

Ефективна робота комп'ютерної системи неможлива без раціонального управління оперативною пам'яттю. Операційна система повинна відслідковувати, які частини пам'яті наразі використовуються, а які є вільними, виділяти пам'ять процесам за запитом та звільняти її після завершення роботи. Ключовим концептом у сучасних ОС є віртуальна пам'ять. Ця технологія дозволяє використовувати більше пам'яті, ніж фізично встановлено в комп'ютері, за рахунок використання частини жорсткого диска як розширення оперативної пам'яті, тобто файл підкачки [2, 5, 22].

Механізм віртуальної пам'яті також забезпечує ізоляцію адресних просторів процесів. Кожен процес бачить свій власний безперервний простір пам'яті, який транслюється блоком управління пам'яттю у фізичні адреси. Це означає, що помилка в одній програмі, яка намагається записати дані за некоректною адресою, не пошкодить дані іншої програми або ядра системи. Для реалізації цього механізму найчастіше використовується сторінкова організація пам'яті, де адресний простір розбивається на блоки фіксованого розміру, сторінки. Операційна система динамічно завантажує необхідні сторінки в фізичну пам'ять і вивантажує на диск ті, що рідко використовуються [2, 5, 13].

Якщо управління пам'яттю стосується тимчасового зберігання даних під час обробки, то файлова система відповідає за довготривале збереження інформації на енергонезалежних носіях. Фізично накопичувачі інформації, такі як жорсткі диски або твердотільні накопичувачі, оперують поняттями секторів, блоків та сторінок. Операційна система створює логічну структуру у вигляді файлів та каталогів чи папок.

Файлова система визначає спосіб іменування файлів, їх зберігання та організації доступу до них. Вона підтримує ієрархічну структуру, дозволяючи групувати дані логічним чином. Також файлова система зберігає інформацію про власника файлу, права доступу, час створення та модифікації, розмір і забезпечує

їх цілісність. У разі вимкнення живлення сучасні журнальовані файлові системи залишають можливим відновлення структури даних через аналіз спеціального журналу транзакцій [2, 5, 22]. Операційна система також керує вільним простором на диску, вирішуючи, в які саме фізичні блоки записати новий файл, і намагаючись мінімізувати фрагментацію, яка може сповільнити до даних доступ.

Буферизації та кешуванні дискових операцій теж є одним із провідних функцій операційних систем. Швидкість роботи накопичувачів значно нижча за швидкість процесора чи оперативної пам’яті, тому ОС створює дисковий кеш, а це дозволяє відкладати операції запису та пришвидшувати читання, якщо дані вже є в кеші [2, 5].

Підсистема вводу-виводу операційної системи відповідає за обмін даними між комп’ютером та периферійними пристроями. ОС уніфікує взаємодію з ними.

Коли периферійний пристрій готовий передати дані або завершив операцію, він надсилає сигнал переривання процесору. Операційна система призупиняє поточний процес, зберігає його стан, обробляє дані за допомогою драйвера, а потім повертає керування перерваному процесу. Використання технологій прямого доступу до пам’яті дозволяє пристроям обмінюватися даними з оперативною пам’яттю без постійної участі центрального процесора, що розвантажує останній для виконання обчислювальних завдань [12].

У багатокористувацьких системах важливо забезпечити автентифікацію користувачів та розмежовувати права доступу. ОС зберігає інформацію про облікові записи та перевіряє повноваження кожного користувача при спробі отримати доступ до об’єкта, а це може бути файл, процес, чи навіть пристрій. Сучасні операційні системи реалізують складністратегії безпеки, включаючи шифрування даних, контроль цілісності системних файлів та захист від шкідливого програмного забезпечення на рівні ядра [10, 11]. Це, відповідно, робить реальним безпечне зберігання, оброблення й передавання інформації

навіть якщо ризики несанкціонованого доступу або зовнішнього втручання підвищені.

1.2 Особливості функціонування багатозадачної ОС

Багатозадачність є фундаментальною характеристикою сучасних операційних систем, що визначає їхню здатність забезпечувати одночасне виконання декількох обчислювальних процесів. У основі цієї концепції лежить ідея ефективного використання ресурсів центрального процесора, який завдяки своїй високій швидкодії здатний обслуговувати велику кількість запитів користувачів та системних програм, перемикаючись між ними настільки швидко, що створюється ілюзія їхньої безперервної та одночасної роботи. Є два ключових поняття, паралелізм та конкурентність [14, 17]. На одноядерних системах багатозадачність реалізується через конкурентність. У цьому випадку процеси чергуються у часі, користуючись одним ресурсом. А у багатопроцесорних або багатоядерних системах є істинний апаратний паралелізм, коли різні потоки інструкцій фізично виконуються одночасно на різних ядрах, при цьому операційна система продовжує застосовувати алгоритми диспетчеризації для управління сотнями активних потоків.

Еволюція підходів до багатозадачності пройшла шлях від систем пакетної обробки, де програми виконувалися послідовно одна за одною, до систем поділу часу. Критичним етапом розвитку став перехід від кооперативної багатозадачності до витискальної. У моделі кооперативної багатозадачності, яка використовувалася, наприклад, у ранніх версіях Windows та Mac OS, операційна система передавала управління процесу. Саме процес мав сам повернути це управління системі [14, 15, 16]. Це виливалось в ризики для стабільності, бо якщо прикладна програма зависала або входила у нескінченний цикл, то блокувала роботу всього комп’ютера. Механізму примусового переривання тоді не існувало. Сучасні системи базуються виключно на витискальній

багатозадачності. У цьому режимі планувальник операційної системи повністю контролює розподіл процесорного часу [17, 19]. Ядро виділяє кожному процесу певний квант часу, а після закінчення апаратний таймер генерує переривання. Далі управління примусово повертається операційній системі, яка визначає, який процес буде виконуватись наступним [18, 20]. Таким чином можливо уникнути монополізації процесор.

Центральним елементом реалізації багатозадачності є поняття процесу та його життєвого циклу. Процес в операційній системі є динамічною структурою, що містить поточні значення регістрів процесора, стек, виділену пам’ять та ресурси. Для управління цими сутностями ядро підтримує спеціальну структуру даних – блок управління процесом [21, 22]. Він містить всю необхідну інформацію для відновлення виконання процесу після переривання, зокрема ідентифікатор процесу, стан (готовий, виконується, очікує), пріоритет, вміст програмного лічильника та вказівники на виділені ресурси пам’яті [23]. Кожен активний процес постійно мігрує між станами. Наприклад, процес, що виконується, може перейти у стан очікування, якщо йому потрібно отримати дані з жорсткого диска (операція вводу-виводу є повільною порівняно з роботою центрально процесора), звільняючи процесор для інших задач.

Механізм, що забезпечує зміну активного завдання, називається перемиканням контексту. Це одна з найбільш ресурсомістких операцій у багатозадачній системі. Коли планувальник вирішує змінити поточний процес, ядро повинно зберегти стан регістрів, прапорів та інших параметрів поточного процесу в його блок управління процесором, а потім завантажити відповідні дані з нього для наступного процесу [21, 22, 23]. Перемикання контексту призводить до непрямих втрат продуктивності через скидання конвеєра процесора та інвалідацію кеш-пам’яті і буфера асоціативної трансляції. Тому, згідно з проведеним дослідженням, розробники операційних систем постійно оптимізують алгоритми планування, намагаючись знайти баланс між частотою перемикань та ефективністю використання процесорного часу.

Планування процесів є інтелектуальним ядром багатозадачності. Алгоритми планування визначають, який із готових до виконання процесів отримає доступ до центрального процесору. Найпростішим є алгоритм карусельного планування, де всім процесам по черзі надається однаковий квант часу [14, 15]. Хоча цей метод є справедливим і запобігає голодуванню процесів, він не враховує специфіку завдань. Більш досконалі системи використовують пріоритетне планування та багаторівневі черги зі зворотним зв’язком [16, 18]. У таких системах процеси, що потребують швидкої реакції, отримують вищий пріоритет, ніж фонові завдання. Операційна система динамічно коригує пріоритети. Тобто, якщо процес довго очікує у черзі, його пріоритет може бути тимчасово підвищений, щоб гарантувати його виконання [18, 23].

Процеси часто використовують спільні ресурси, зокрема області пам’яті, файли, чи глобальні змінні, тому виникає ризик стану гонитви, коли результат виконання залежить від непередбачуваної послідовності операцій різних процесів. Для вирішення цієї проблеми операційні системи мають примітиви синхронізації, а саме м’ютекси, семафори та монітори. М’ютекс дозволяє створити критичну секцію коду, яку в кожен момент часу може виконувати лише один потік [9]. Це запобігає конфліктам доступу, але спричиняє взаємне блокування. Дедлок виникає, коли два або більше процесів очікують на ресурси, які утримуються один одним, створюючи замкнене коло очікування. Варто зазначити, що попри високу інтелектуальність сучасних систем, вони не є панацеєю від помилок у логіці прикладного коду. Операційні системи сьогодні володіють розвиненими механізмами моніторингу, які здатні ідентифікувати критичні стани, наприклад, через аналіз графів очікування ресурсів та втручатися в роботу процесів для запобігання повному зависанню ядра.

Але ці превентивні заходи мають свої межі. Якщо архітектура взаємодії потоків всередині програми побудована некоректно, система може лише констатувати факт виникнення взаємного блокування, але аж ніяк не виправити його без примусового завершення процесу. Таким чином, основна

відповідальність за створення відмовостійких алгоритмів та коректне використання примітивів синхронізації все одно залишається на розробникові. ОС у цьому випадку виступає не стільки своєрідним рятівним засобом, скільки останнім рубежем захисту, що ізолює наслідки програмної помилки від решти користувачів та системних сервісів

Сучасна багатозадачність використовує потоки або легковагові процеси. На відміну від класичних процесів, потоки, що належать одній програмі, поділяють спільний адресний простір пам’яті та таблицю дескрипторів файлів. Створення потоку та перемикання між потоками одного процесу вимагає значно меншої кількості ресурсів ніж створення нового процесу, бо не потребує створення нової віртуальної адресної карти. Підтримка багатопотоковості реалізується як на рівні користувача, так і на рівні ядра [21, 22]. Більшість сучасних ОС використовують модель, де потоки ядра відображаються на потоки користувача, що дозволяє операційній системі розподіляти потоки однієї програми між різними фізичними ядрами процесора, значно підвищуючи продуктивність у важких обчислювальних задачах.

Операційна система повинна гарантувати ізоляцію адресного простору кожного процесу. Використання віртуальної пам’яті дозволяє кожному процесу розуміти, що він володіє всією доступною пам’яттю, починаючи з нульової адреси. Блок управління пам’яттю спільно з операційною системою транслює ці віртуальні адреси у фізичні. Така архітектура захищає систему від збоїв. Якщо один процес спробує записати дані в пам’ять іншого, то це викличе апаратне виключення і ОС просто завершить некоректний процес, не впливаючи при цьому на роботу інших задач та самої системи.

У середовищах віртуалізації та контейнеризації один фізичний хост може обслуговувати тисячі ізольованих екземплярів програм. Операційна система хоста виступає як гіпервізор або менеджер контейнерів, розподіляючи ресурси між ними. Тут у дії складніші алгоритми планування, такі як Completely Fair Scheduler у Linux, який використовує червоно-чорні дерева для відстеження

виконання задач з точністю до наносекунди, забезпечуючи при цьому чесний розподіл процесорного часу навіть при великому навантаженні [1, 18].

Водночас, багатозадачність накладає певні обмеження на системи реального часу. У системах загального призначення, Windows, Linux, затримка у перемиканні задач або непередбачуваність планувальника є прийнятною. Однак у промислових контролерах чи тому ж медичному обладнанні така невизначеність є критичною. Тому системи реального часу використовують детерміновані алгоритми планування, у яких час реакції на переривання є фіксованим.

1.3 Поняття інтернету речей та пристроїв інтернету речей

Концепція Інтернету речей, або IoT, за останні роки остаточно перейшла з від простого прогнозу чи плану у статус звичного елемента сучасної цифрової інфраструктури. У науковому розумінні цю систему доцільно визначати як багаторівневу екосистему фізичних об’єктів, які завдяки інтегрованим сенсорам та обчислювальним модулям отримують здатність взаємодіяти між собою або із зовнішнім середовищем.

Взагалі історично розвиток IoT став можливим завдяки мініатюризації електроніки, здешевленню обчислювальних потужностей та поширенню бездротового зв’язку. Важливу роль відіграло також впровадження протоколу IPv6, який вирішив проблему закінчення адресного простору, дозволивши присвоїти унікальну IP-адресу надзвичайно великій кількості об’єктів [24, 25, 26].

Архітектурною особливістю IoT-систем є їхня трирівнева структура, що включає рівень сприйняття, мережевий рівень та рівень застосувань. На базовому рівні знаходяться, пристрої інтернету речей. Це широкий спектр обладнання, який можна умовно поділити на дві категорії: сенсори та актуатори. Сенсори виступають своєрідними органами чуття системи – вони збирають

інформацію про фізичні параметри середовища. Актуатори дозволяють цифровій системі впливати на фізичний світ, вмикати світло, перекривати клапан водопостачання або змінювати режим роботи двигуна.

Пристрій IoT оснащений мікроконтролером для первинної обробки даних та комунікаційним модулем для їх передачі. Саме спосіб передачі даних визначає специфіку взаємодії в таких системах. На відміну від класичного інтернету, де переважають великі потоки даних, пристрої IoT часто оперують короткими повідомленнями телеметрії, але у величезній кількості. Це зумовило появу специфічних протоколів зв'язку, зокрема MQTT, CoAP, та енергоефективних мереж дальнього радіусу дії, які дозволяють датчикам працювати роками від однієї батарейки, передаючи дані крізь стіни або на значні відстані [33, 39, 40].

Принцип взаємодії в екосистемі IoT базується на концепції міжмашинної взаємодії. Дані, зібрані сенсорами, передаються через шлюзи до хмарних платформ або обробляються на краю мережі. Хмарні технології виступають керуючим елементом системи, де накопичуються масиви великих даних. Тут застосовуються алгоритми машинного навчання та штучного інтелекту для аналізу патернів, виявлення аномалій та прийняття рішень [24, 38]. Наприклад, система розумного будинку не просто показує температуру в кімнаті, а аналізує звички мешканців і погодні умови, автоматично налаштовуючи опалення для економії енергії.

Практичне застосування інтернету речей охоплює майже всі сфери людської життєдіяльності. У приватному секторі це передусім екосистеми «Розумний дім», де побутова техніка, системи безпеки та освітлення працюють як єдиний злагоджений організм [27, 28, 31, 37]. На рівні міста існує концепція «Розумне місто», дозволяє містам адаптувати світлофори, щоб розвантажити трафік, автоматизувати вуличне освітлення так, щоб воно вмикалося лише там, де дійсно є люди.

Найбільш ефективне використання IoT відбувається у промисловому секторі. Впровадження датчиків на виробничих потужностях дозволило перейти

до стратегії предиктивного обслуговування. Замість того, щоб людина була змушена ремонтувати верстат після поломки або проводити планову заміну деталей, які ще могли б служити, система сама за непрямими ознаками, мікро-вібраціями чи зміною температури, виявляє проблему на стадії її появи [27, 30, 40]. Це в свою чергу мінімізує збитки від простої. Аналогічні процеси відбуваються і в логістиці, де IoT-мітки забезпечують повну прозорість ланцюга постачання, дозволяючи контролювати температуру ліків у рефрижераторі і точний час прибуття вантажу в порт.

Для сільського господарства та екології інтернет речей також несе суттєве значення. Датчики вологості ґрунту та пристрої аналізу стану рослин дозволяють кожній рослині отримувати рівно стільки води та добрив, скільки їй потрібно.

1.4 Системне програмне забезпечення для пристроїв IoT

Функціонування екосистеми інтернету речей безпосередньо залежить від ефективності взаємодії апаратних компонентів із програмними оболонками, що керують їхніми ресурсами. Системне програмне забезпечення у цьому контексті виступає посередником, який трансформує обмежені обчислювальні можливості мікроконтролерів та сенсорів у функціональні інтелектуальні вузли [39, 40, 50]. На відміну від традиційних комп'ютерних систем, де програмне забезпечення орієнтоване на взаємодію з користувачем, системне ПЗ для IoT фокусується на автономності, надійності та здатності працювати в умовах суворих апаратних обмежень.

Основою системного рівня IoT-пристроїв є вбудоване програмне забезпечення, тобто прошивка. Воно інтегрується безпосередньо в енергонезалежну пам'ять мікроконтролера і виконує роль основи для всіх подальших операцій. У найпростіших сценаріях, наприклад, у датчиках температури або вологості, прошивка може бути єдиним програмним шаром, що працює за принципом голого заліза [39, 42, 44]. Цей підхід забезпечує

можливість отримати максимально глибокий контроль над апаратним забезпеченням, що є критичним для мінімізації затримки сигналу та раціонального споживання енергії. Однак в міру того, як функціонал IoT-пристроїв стає дедалі складнішим, програмування безпосередньо на залізі починає демонструвати свої обмеження. Коли виникає потреба в одночасному забезпеченні роботи декількох мережевих протоколів, реалізації багатопотоковості або впровадженні надійних методів криптографічного захисту, написання коду з нуля стає об’єктивно недоцільним. Саме в цей момент з’являється об’єктивна необхідність у переході до використання спеціалізованих операційних систем, які беруть на себе управління системними викликами і ресурсами [38, 39].

Операційні системи для інтернету речей суттєво відрізняються від десктопних чи мобільних аналогів своєю архітектурною гнучкістю і модульністю. Головним завданням такої ОС є не графічний інтерфейс, а ефективне керування завданням та пам'яттю. Більшість сучасних IoT-рішень базуються на операційних системах реального часу. Їх особливість полягає в детермінованості. Система гарантує виконання критично важливого завдання протягом чітко визначеного проміжку часу [49]. Це життєво важливо для промислових систем автоматизації, медичних пристроїв або систем «розумного міста», де затримка в обробці сигналу навіть на частку секунди може призвести до системного збою або аварійної ситуації.

Серед архітектурний підходів варто виділити мікроядерні архітектури, які набули значної популярності в розробці системного програмного забезпечення для IoT. У мікроядерній ОС лише найнеобхідніші функції, такі як керування перериваннями та базове планування, виконуються в привілейованому режимі ядра [2, 5, 39]. Всі інші сервіси, файлові системи, мережеві стеки та драйвери, винесені в простір користувача. Якщо один із компонентів системи виходить з ладу, то він не спричиняє критичного падіння всієї системи, що дозволяє пристрою виконати самовідновлення без повного перезавантаження [41, 42, 43].

Значна частина IoT-вузлів живиться від автономних джерел і повинна функціонувати протягом довгого часу без обслуговування. Через це операційна система має володіти механізмами керування живленням. Це реалізується через глибоку інтеграцію з режимами сну мікроконтролера. Системне програмне забезпечення аналізує чергу завдань і, якщо активних процесів немає, переводить процесор у стан наднизького споживання, залишаючи активними лише таймери або зовнішні переривання [27, 38, 39].

Враховуючи, що IoT-пристрої часто працюють у відкритих середовищах і мають доступ до конфіденційних даних, системне програмне забезпечення повинно забезпечувати ізоляцію процесів та захищене зберігання ключів шифрування. Сучасні стандарти вимагають впровадження «кореня довіри» на рівні завантажувача. Тобто пристрій перевіряє цифровий підпис кожного компонента програмного забезпечення перед його запуском. Якщо прошивка була несанкціоновано змінена, то система заблокує завантаження, таким чином запобігаючи перетворенню пристрою на частину інфікованої шкідливим програмним забезпеченням мережі [41, 42, 47].

Невід'ємною частиною системного ПЗ є також механізми бездротового оновлення. Оскільки кількість пристроїв може перевищувати тисячі, то фізичний доступ до кожного з них для оновлення коду є неможливим. Процес OTA-оновлення є полягає в наступному: система повинна завантажити нову версію програмного забезпечення у фоновому режимі, перевірити її цілісність і провести безпечну інсталяцію, маючи при цьому план відкату на випадок невдачі [48]. Цей механізм визначає життєздатність системи в довгостроковій перспективі.

Масштабованість і стабільність будь-якої IoT-системи знаходяться у прямій залежності від якості її системного програмного забезпечення. При розширенні мережі та збільшенні кількості робочих вузлів є неминучою проблема гетерогенності, коли в межах одного проєкту доводиться поєднувати

різні апаратні платформи. У такому разі використовують системне ПЗ, побудоване на принципах абстракції апаратного рівня [39, 50].

Використання абстракції апаратного рівня уніфікує інтерфейси взаємодії з периферією, що дозволяє уникнути зайвих дій із специфічними регістрами конкретного мікроконтролера. На практиці це означає, що програмне забезпечення, спочатку написане для архітектури ARM Cortex-M, може бути портоване на RISC-V з мінімальними правками у вихідному коді. При такому підході фокус направлено на прикладну логіку пристрою, а не на нюанси реалізації інтерфейсів GPIO, I2C чи SPI на низькому рівні.

Окрім менеджменту обчислень, фундаментальною задачею системного програмного забезпечення в IoT-сегменті є підтримка мережевих стеків. У класичних мережевих середовищах стандартом виступає повний стек TCP/IP, проте для енергоефективних пристроїв він часто виявляється надто громіздким. Через це в ОС для Інтернету речей інтегрують полегшені рішення, такі як uIP або lwIP, а також специфічні протоколи рівня 6LoWPAN, Zigbee чи Thread. Головна вимога до операційної системи тут полягає у інкапсуляції даних та фрагментації пакетів [33, 39, 40].

Важливо враховувати і класову диференціацію системного програмного забезпечення. Якщо на кінцевих сенсорних вузлах домінують рішення класу RTOS, то для шлюзів та концентраторів стали стандартом вбудовані версії Embedded Linux, зокрема проєкти Yocto та OpenWrt. Хоча такі системи потребують сильної апаратної бази з блоком MMU та більшим обсягом пам’яті, але багатозадачність і підтримка високорівневих мов на кшталт Python чи Node.js, які вони надають, того варті [39, 40].

Адаптація технологій віртуалізації та контейнеризації для вбудованих платформ зараз теж дуже важлива. Раніше Docker вважався занадто ресурсомістким для IoT, але зараз використання його полегшених аналогів дозволяє надійно ізолювати прикладні сервіси один від одного та від ядра ОС [46].

Сучасні мікроконтролери комплектуються нейропроцесорними прискорювачами або DSP-блоками. Відповідно, операційна система повинна надавати ефективні API для доступу до цих ресурсів, дозволяючи виконувати інференс нейромереж локально. Це вимагає впровадження складніших алгоритмів планування, які збалансовують час виконання задачі з енергетичним бюджетом пристрою.

1.5 Висновки до першого розділу

Завершуючи теоретичний розгляд принципів функціонування операційних систем і системного програмного забезпечення в середовищі інтернету речей, можна зробити висновок, що стабільність сучасних цифрових екосистем базується на поєднанні апаратної адаптивності. Проведений у розділі 1 аналіз показав, що операційна система залишається основним елементом будь-якої обчислювальної платформи, одночасно розширюючи можливості обладнання та керуючи його обмеженими ресурсами. Вибір архітектури та організація ядра мають вирішальне значення, оскільки саме вони формують баланс між швидкодією та надійністю. Поширення мікроядерних підходів і систем реального часу є ключовими в IoT-середовищі, де навіть незначні затримки або збої можуть впливати на роботу всієї інфраструктури.

Дослідження механізмів багатозадачності підтвердило, що витискальне планування та ефективне управління контекстом процесів найкраще відповідають умовам динамічних систем. Розділення адресних просторів і використання віртуальної пам’яті підвищують рівень відмовостійкості, а це особливо важливо для автономних пристроїв, які працюють без постійного контролю з боку користувача. Водночас питання синхронізації процесів і уникнення взаємних блокувань залишаються складними й потребують постійного вдосконалення алгоритмів керування.

Окремо було також проаналізовано особливості прошивок і вбудованих платформ, що функціонують у режимі жорстких енергетичних обмежень. Встановлено, що ефективне управління живленням на рівні операційної системи безпосередньо впливає на життєвий цикл автономних вузлів.

2 КОМПОНЕНТНЕ ПРОЄКТУВАННЯ МЕХАНІЗМІВ РОБОТИ ОПЕРАЦІЙНОЇ СИСТЕМИ РЕАЛЬНОГО ЧАСУ ДЛЯ ПРИСТРОЇВ IoT

2.1 Підсистема ядра та механізми міжпотокової взаємодії

Архітектура ядра розглядається як узгоджений набір детермінованих сервісів і бібліотечних компонентів, які забезпечують керування системними об’єктами, потоками виконання, часом, засобами синхронізації та оперативною пам’яттю. Така організація дає змогу ядру працювати стабільно навіть у середовищі з обмеженими апаратними ресурсами. Для його функціонування достатньо незначного обсягу постійної пам’яті та ще меншого обсягу оперативної пам’яті [20, 22].

Логічна структура системи побудована на уніфікованій об’єктно-орієнтованій моделі керування. Через неї здійснюється робота з потоками, семафорами, програмними таймерами, поштовими скриньками та рештою системних примітив. Усі перелічені елементи подаються як похідні від базового системного об’єкта, який містить текстове ім’я, тип і набір службових прапорців стану, успадковуючи його пластивості. Після створення такі об’єкти автоматично додаються до глобальних двозв’язних списків відповідного типу [9, 21].

Для опису і супроводу кожного окремого потоку в оперативній пам’яті виділяється окремий блок керування, який зберігає основні службові дані про задачу. У ньому міститься покажчик на вершину локального стека. Ця стекова ділянка відведена під зберігання локальних змінних, адрес повернення функції, а також для збереження/відновлення повного контексту регістрів центрального процесора під час процедури перемикання контексту [23].

Диспетчеризація задач підтримує шкалу пріоритетів, яка масштабована гнучко, верхня межа ъъ за наявності достатнього ресурсу пам'яті може сягати близько трьохсот дискретних рівнів. Найвищий ранг у черзі планувальника закріплено за нульовим індексом. Алгоритм планування функціонує

детерміновано, тому тривалість вибору наступної для виконання задачі є константною і не корелює із загальною кількістю активних потоків [14, 15]. Для задач із тотожними пріоритетами реалізовано стратегію кругового обслуговування із динамічним квантуванням процесорного часу.

Життєвий цикл потоку описується моделлю скінченного автомата та регламентується п’ятьма основними станами, зокрема початковий, готовності, виконання, блокування і завершення (рис. 2.1). На етапі сторення новий потік фіксується у вихідній точці життєвого циклу, при цьому в його виділеному стековому просторі завчасно формується первинний контекст, який моделює виклик цільової функції. У момент переходу потоку до реєстру готових задач він маркується як валідний об'єкт для диспетчеризації.

Рисунок 2.1 – Діаграма станів потоку

Якщо під час роботи потік змушений чекати звільнення ресурсів, завершення затримки або сигналу синхронізації, то він переводиться у заблокований стан. Тоді потік вилучається з черги готовності, а відповідний біт у карті пріоритетів скидається за умови, що черга даного рівня більше не містить активних задач. Фонова задача простою системи володіє мінімальним рангом і

перманентно перебуває у стані готовності, забезпечуючи утилізацію та очищення звільнених системних ресурсів. Пізніше асинхронні процедури ядра здійснюють фінальний обхід цього списку для остаточного звільнення виділених під задачу адресних просторів та блоків пам'яті.

Підсистема керування часом базується на системному такті, а такт в свою чергу виступає мінімальним дискретним квантом часу в операційному середовищі. На основі цього реалізовано підсистему програмних таймерів, яка підтримує два режими функціонування. Одноразовий запуск, коли функція зворотного виклику або певна подія спрацьовує лише один раз після завершення заданого інтервалу. Періодичний, що забезпечує циклічний запуск цільових процедур через фіксовані часові проміжки. Для оптимізації обчислювальних витрат ядра є організація черги активних таймерів у формі двозв'язного списку, упорядкованого за зростанням абсолютного часу їхнього найближчого спрацьовування [20, 21, 22].

Залежно від вимог до часових характеристик програмні таймери можуть працювати в режимах апаратного переривання і системного потоку з високим пріоритетом.

Інтегровані в мікроядро механізми міжпроцесної взаємодії та синхронізації призначені для регулювання конкурентного доступу до розділюваних ресурсів і координації паралельних обчислювальних потоків. Системними засобами координації виконання потоків виступають семафори та м’ютекси.

Робота семафора базується на атомарному лічильнику і черзі очікування. Якщо ресурс у певний момент недоступний, потік переводиться у стан блокування й додається до відповідної черги. Натомість реалізація м'ютексів передбачає обов'язкову ідентифікацію потоку-власника, що дозволяє апаратно-програмно підтримувати протокол успадкування пріоритетів для запобігання неконтрольованій інверсії [9, 16, 20].

Окремим гнучким інструментом координації є групи подій, де кожен біт системної змінної функціонує як незалежний бінарний сигнал. Потік може

призупиняти своє виконання до настання потрібної комбінації подій. У такому випадку він або реагуватиме на перший із доступних сигналів за принципом диз'юнктивного вибору, або продовжувати очікування доти, доки не будуть синхронно виконані всі задані умови за логікою кон’юнкції.

Передавання інформації між потоками, а також взаємодія потоків з обробниками переривань базується на використанні механізмів поштових скриньок і черг повідомлень. Застосування поштових скриньок найбільш ефективне, коли потрібно швидко переслати короткі інформаційні пакети утвердженої довжини. Якщо ж розмір даних наперед не є фіксованим, може змінюватися або взагалі невідомий, асинхронне міжпроцесне обмінювання між системними компонентами реалізується через черги. Алгоритмічно і логічно вони працюють подібно до захищеного циклічного сховища. Тобто спочатку дані дублюються з адресного простору потоку-відправника у внутрішній виділений репозиторій черги, після чого асинхронно переміщуються у цільову область пам'яті потоку-отримувача [9].

Підсистема керування пам’яттю базується на габридній дворівневій архітектурі, що поєднує статичні та динамічні алгоритми розподілу адресного простору для зменшення його фрагментації. Статичні пули пам’яті забезпечують високий рівень детермінованості, оскільки на етапі ініціалізації завчасно визначений масив ОЗУ сегментується на блоки ідентичного фіксованого розміру. Вільні блоки формують однозв’язний список, де початкові чотири байти кожного вільного елемента містять вказівник на наступний вузол. Це дозволяє здійснювати операції виділення та звільнення пам’яті за константний час, що є особливо цінним для критичних обробників переривань [5].

Для задач, які потребують більшої гнучкості, використовується динамічне керування купою. Ця послідовність дій працює з різнорозмірними секціями. Структура кожного такого сегмента містить службовий заголовок, куди інтегрується захисна сигнатура для визначення переповнення буфера, прапорець поточного стану виділення, а також посилання на суміжні адреси для підтримки

зв'язності простору. Коли пам’ять вивільняється, система самостійно аналізує сусідні сегменти і, якщо вони теж перебувають у незайнятому стані, зливає їх в єдину область більшого розміру [2, 22].

Підсистема роботи з апаратними перериваннями спроєктована таким чином, що після появи асинхронного сигналу від периферійного пристрою чи комунікаційного інтерфесу процесор автоматично фіксує актуальний стан виконання, після чого здійснює перехід на відповідний вектор переривання. Якщо в цей момент до виконання стає готовим потік із вищим рівнем пріоритету, система вже не продовжує роботу раніше перерваного фрагмента коду, а запускає механізм відкладеного перемикання контексту. Тільки після повного завершення обробки всіх відкладених переривань, процесор відновлює регістровий стан нового, вищого за пріоритетом, потоку виконання.

2.2 Підсистема керування апаратними ресурсами

Підсистема управління ресурсами устаткування представлена п'ять базовими структурними компонентами (рис. 2.2). Віртуальна файлова система формує єдиний спосіб організації та доступу до даних, близький до підходів, які використовуються в ОС уподіблених до UNIX. Для прикладного програмного коду це означає роботу через звичний POSIX-сумісний набір операцій відкриття, читання, запис і закриття файлів. Завдяки такому підходу програмні модулі легше переносити між сумісними платформами, оскільки логіка роботи з даними не прив’язується безпосередньо до конкретного носія.

Накопичувачі з різною апаратною будовою не розглядаються напряму, оскільки їхні фізичні відмінності, варто відзначити, приховує проміжний рівень. Завдяки цьому для операційної системи такі носії виглядають як єдиний впорядкований простір логічних адрес. Усередині цієї частини архітектури можуть застосовуватися кілька різновидів файлових систем, кожна з яких оптимізована для певного способу використання. Зокрема, одна з них передбачає

роботу зі знімними носіями, тоді як інша більш компактна і доступна тільки для читання дозволяє виконувати інструкції безпосередньо з flash-пам’яті без завчасного завантаження коду в ОЗП. Крім того, окремо використовуються засоби віртуальної файлової системи, які займають місце перетворювачів периферійних апаратних модулів на умовні файлові об’єкти, які відображаються у спеціальному службовому каталозі.

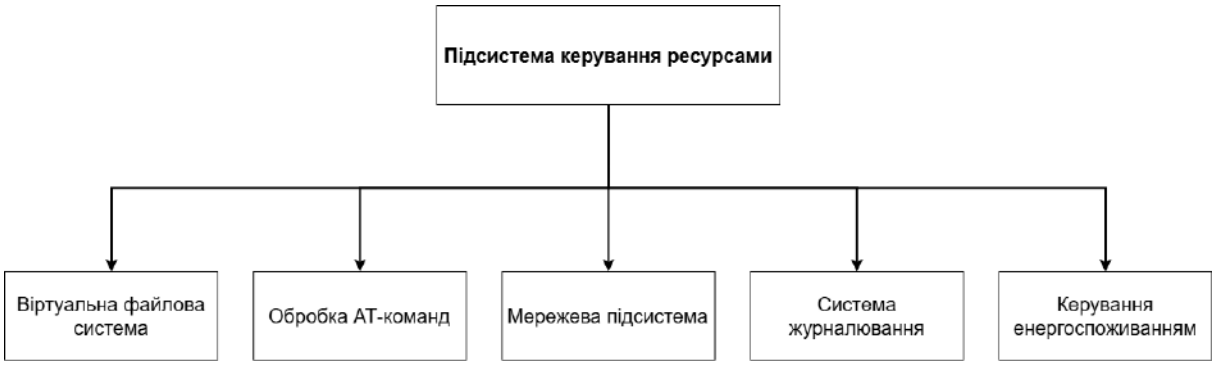


Рисунок 2.2 – Компоненти підсистеми управління апаратними ресурсами

Комунікаційний модуль системи організований навколо сокетного механізму, завдяки чому різні мережеві протоколи можуть використовуватися через уніфікований прикладний інтерфейс. Власний стек мережі розрахований на роботу в середовищі з невеликим обсягом оперативної пам’яті, тому для зменшення витрат ресурсів у ньому застосовуються буферні пули з динамічним розміром і передавання даних без зайвого копіювання. Попри таку оптимізацію, обробка пакетів залишається сумісною з базовими мережевими принципами. Зокрема, підтримується механізм ковзного вікна, а також виконується обчислення контрольних сум з метою верифікації інформаційних кадрів [56, 57].

У випадках, коли передавання інформації здійснюється через зовнішні модемні модулі, звичайні мережеві запити не виконуються напряму, а транслюються у відповідні АТ-команди засобами підсистеми віртуальних сокетів. Для файлових об’єктів і мережевих з’єднань передбачено єдиний простір дескрипторів. Це дає змогу використовувати мультиплексований ввід-

вивід і працювати з кількома каналами зв’язку одночасно, не створюючи для кожного підключення окремий потік. Безпека інформаційного обміну та захист даних під час передавання реалізуються через криптографічне шифрування безпосередньо на цьому ж рівні [61].

Для обміну із зовнішніми комунікаційними модулями в системі використовується окремий механізм опрацювання текстових команд. Його структура поділена на дві частини: клієнтську та серверну. Більш детально це виражається в тому, що клієнтський функцінал зосереджений на генерації запитів в асинхронному режимі, очікує на відповідь через системні семафори та виконує розбір інформації, отриманої від апаратних модемів.

У основі обробки інформаційних потоків покладено модель скінченних автоматів. Завдяки цьому вона може стабільно працювати з асинхронними потоками даних, які не мають чітко впорядкованої структури. Це особливо корисно у випадках раптового розриву мережевого з’єднання або появи нових вхідних пакетів. При цьому окремий фоновий потік ядра безперервно відстежує стан кільцевого буфера і, коли знаходить у ньому службові позначки, передає керування відповідним прикладним сервісам [9, 56].

Серверна частина, у противагу, призначена для приймання команд налаштування від зовнішнього обладнання та діагностичного моніторингу. Через неї зручніше виконувати первинну конфігурацію системи, перевірку окремих модулів і калібрування сенсорів. Обидва компоненти оптимізовано так, що сумарний обсяг бінарного коду клієнтської та серверної складових не перевищує кількох кілобайт, що дозволяє раціонально використовувати ресурс мікроконтролера.

Енергетично відповідальна зона призначена для того, щоб збільшити тривалість автономного функціонування пристрою і водночас не допустити відчутного зниження швидкодії. У її роботі передбачено повноцінний робочий стан, а також декілька варіантів переходу в режим зниженого споживання. Коли пристрій залишається активним, система може змінювати напругу та тактову

частоту залежно від поточного навантаження, завдяки чому за невеликої інтенсивності роботи витрати енергії зменшуються. Чергування режимів сну, в свою чергу, охоплює і тимчасове обмеження роботи ядра, і вимкнення окремих апаратних ділянок.

На рівні операційної системи для кожного потоку або периферійного вузла можуть встановлюватися індивідуальні вимоги щодо найнижчого прийнятного рівня енергозбереження. Ці обмеження аналізує відповідальний за цю частину централізований менеджер живлення, після чого ініціює перехід платформи у максимально глибокий режим сну, не порушуючи коректність роботи критичних елементів системи.

Через те, що під час глибокого сну основний системний таймер припиняє свій відлік, у ядрі реалізовано засіб відновлення часової узгодженості. Алгоритм передбачає, що перед засинанням запускається низькочастотний малопотужний таймер, а після повернення до активного стану система обчислює тривалість перебування в пасивному стані і відповідно коригує глобальний системного часу ядра [20].

Для спостереження за станом системи і верифікації її процесів застосовується розгалужений механізм журналювання, організований так, щоб майже не створювати додаткового навантаження на центральний процесор. Згенеровані ядром та прикладним ПЗ події першочергово перехоплюються клієнтським шаром, після чого центральна частина логування доповнює їх часовими мітками й технічною службовою інформацією, інкапсулюючи [51]. Безпосереднє виведення сформованих записів у фізичні канали зв’язку, наприклад через інтерфейс UART, забезпечує ізольований серверний компонент.

Окремо передбачено асинхронний спосіб роботи, тобто у цьому випадку журнальні записи не передаються на вихід одразу, а тимчасово зберігаються в кільцевому буфері. Після цього спеціальний потік поступово обробляє накопичені дані та виконує необхідні операції інформаційного входу та виходу [52].

2.3 Підсистема мережевої взаємодії та обміну даними для IoT

Мережевий обмін не залежить від характеристик конкретного фізичного носія інформації. Прикладний рівень звертається до ресурсів мережеві через пакет підтримки плати, який відповідає за роботу з апаратною частиною, тоді як вибір конкретного інтерфейсу залишається прихованим у внутрішніх механізмах системи. Завдяки цьому для програмного коду немає суттєвої відмінності, яким саме способом передаються дані, будь це через Ethernet, Wi-Fi чи стільниковий модем [56]. На системному рівні ця частина є своєрідним диспетчером викликів. У пам’яті підтримується таблиця відповідності між файловими дескрипторами та мережевими протоколами, що дозволяє одночасно працювати з різними типами інтерфейсів. Отже, група мережевої взаємодії та обміну даними складається із чотирьох основних компонентів (рис. 2.3).

Базовим елементом цієї групи компонентів виступає універсальна зона, у якій міститься сокетний прошарок. Через нього виконуються операції відкриття комунікаційного каналу, надсилання та отримання інформації, а також відстеження поточного стану мережевих ідентифікаторів. Щоб визначити, які канали вже готові до роботи, застосовуються засоби вибіркового очікування й періодичної перевірки, завдяки чому система може паралельно супроводжувати кілька підключень, не змушуючи потоки надмірно довго перебувати у заблокованому стані [57].

У внутрішній структурі ядра мережевий канал подається не як незрозумілий чи невідомий віддільний об’єкт, а як одна з форм файлового ідентифікатора. Для цього в оперативній пам’яті підтримується таблиця зв’язків між такими ключами, типами протокольної обробки та відповідними комунікаційними адаптерами [57]. Саме тому операції запису і читання можуть виконуватися за схожою схемою і для файлів, і для каналів передавання даних.

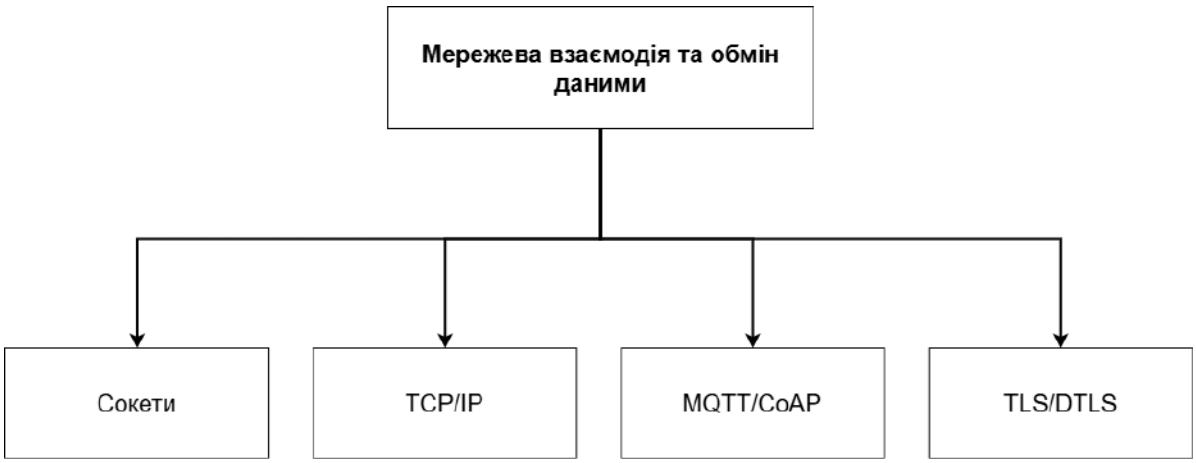


Рисунок 2.3 – Компоненти підсистеми мережевої взаємодії та обміну даними

Подальше спрямування трафіку визначається через внутрішній каталог комунікаційних адаптерів. Кожен його елемент містить набір процедур, пов’язаних із певною реалізацією мережевої обробки або конкретним апаратним інтерфейсом. Коли потік ініціює створення об’єкта в сокеті, система перевіряє задані параметри протокольної групи й обирає пристрій, з яким має бути пов’язаний новий ідентифікатор. Після цього передавання даних може бути направлене або на вбудований Ethernet-модуль, або на зовнішній засіб зв’язку.

У розробленій системі мережевий обмін може реалізовуватися за двома різними підходами. Перший підхід використовується тоді, коли вся обробка TCP/IP виконується безпосередньо на самому пристрої. У такій конфігурації до програмного середовища підключається стек LwIP, який бере на себе основні функції мережевої взаємодії без потреби у додатковому зовнішньому модемі чи окремому процесорі зв’язку. На рівні системного ядра він забезпечує формування, приймання та службову обробку пакетів, зокрема контроль коректності переданих даних, фрагментацію великих IP-повідомлень і регулювання потоку в межах TCP-з’єднань [56]. Обмін між мережевим контролером і програмною частиною протокольної обробки організовано не через пряме безперервне звернення, а за допомогою механізмів взаємодії між задачами. Події, які виникають під час роботи апаратного інтерфейсу, передаються до відповідного обробного потоку через черги повідомлень або

поштові скриньки. Така схема дозволяє безпечно переносити дані з контексту переривання у звичайний режим виконання задачі, не порушуючи послідовності подальшої обробки.

Протокол міжмережевої взаємодії забезпечує адресацію та передавання пакетів між вузлами мережі. Протокол керуючих повідомлень застосовується для діагностики, перевірки доступності вузлів і повідомлення про помилки. Протокол користувацьких датаграм використовується для швидкого передавання невеликих повідомлень без попереднього встановлення з'єднання. Протокол керування передаванням, навпаки, призначений для випадків, коли потрібні підтвердження доставки, контроль цілісності обміну та керування інтенсивністю передавання даних [57, 58]. Інший спосіб мережевої роботи застосовується тоді, коли підтримка стеку відповідної взаємодії уже реалізована всередині зовнішнього комунікаційного модуля. Це характерно для модулів стільникового зв'язку, вузькосмугового інтернету речей і деяких модулів бездротової мережі, які керуються через текстові команди керування [56].

На внутрішньому рівні системи запит транслюється у текстову команду для комунікаційного модуля. Далі вона передається модулю через універсальний асинхронний приймач-передавач, а спеціальний програмний компонент розбирає отриману відповідь і оновлює стан з'єднання. Той самий підхід використовується не лише під час створення сеансу з гарантованим передаванням, а й для повного керування зовнішнім модемом. Підсистема виконує первинну конфігурацію модуля, контролює його реєстрацію в мережі оператора, обробляє сповіщення про появу нових даних і відстежує ситуації, коли канал зв'язку розривається, закривається або стає недоступним.

Для реалізації застосовується компонент клієнта текстових команд керування, винесений в окремий потік із підвищеним пріоритетом виконання. Його робота полягає в безперервному прийманні символів із послідовного інтерфейсу, подальшому розборі сформованих рядків і зіставленні їх із наперед зареєстрованими ознаками подій. Через те, що відповіді модема можуть

надходити із суттєвою затримкою, у цьому модулі використовуються семафорні механізми. Вони синхронізують доступ до спільного фізичного каналу й дають змогу кільком потокам працювати з ним без конфліктів [56].

До складу комунікаційного рівня входять засоби обміну даними, орієнтовані на пристрої з обмеженими ресурсами та періодичною передачею інформації. Одним із таких рішень є MQTT. Його доцільно використовувати тоді, коли кінцевий вузол не повинен постійно підтримувати складну взаємодію з сервером, а передає або отримує дані через посередника, брокер повідомлень [60]. У такій моделі пристрій не звертається напряму до кожного отримувача, а надсилає інформацію у визначену тематичну категорію. Інші учасники системи отримують ці дані лише тоді, коли попередньо підписані на відповідну тему. У розробленій архітектурі MQTT не прив’язаний до конкретного типу мережевого каналу. Його робота винесена на рівень абстракції сокетів, тому той самий програмний компонент може використовувати незахищене TCP-підключення або канал із шифруванням через TLS [60, 61]. Це спрощує перенесення системи між різними варіантами зв’язку, оскільки логіка прикладного обміну не потребує окремої переробки для кожного транспорту.

Особливістю протоколу передавання телеметрії з чергою повідомлень є невеликий обсяг службових даних у пакетах. Завдяки цьому протокол придатний для вузлів, які передають короткі телеметричні повідомлення, працюють від батареї або мають обмежену пропускну здатність каналу. Надійність доставки може налаштовуватися залежно від вимог конкретної задачі, тобто повідомлення можуть надсилатися без підтвердження, з підтвердженням отримання або з гарантією одноразової обробки. Якщо протягом заданого часу між клієнтом і посередником обміну повідомленнями немає звичайного обміну, клієнт надсилає короткий службовий запит для перевірки зв’язку. За відсутності відповіді система може швидко визначити, що підключення втрачено, і виконати повторне з’єднання або іншу передбачену реакцію [60]. Це особливо корисно для

розумних пристроїв, які працюють у нестабільних мережах і мають передавати дані без постійного ручного контролю.

Для обміну даними з вузлами Інтернету речей у системі передбачено підтримку CoAP. Його доцільно застосовувати там, де пристрій має швидко передавати короткі повідомлення і не витрачати зайві ресурси на підтримання складного мережевого сеансу. CoAP працює поверх протоколу користувацьких дейтаграм, тому й краще підходить для вбудованих пристроїв із невеликим обсягом пам’яті так ще й обмеженою пропускною здатністю каналу.

У системі датчики та виконавчі механізми представляються як віртуальні ресурси з унікальними URI. CoAP також підтримує механізм підтверджуваних повідомлень, тобто якщо підтвердження не надходить протягом визначеного часу, пакет повторно передається з поступовим збільшенням інтервалу очікування. Додатково реалізовано механізм спостереження за ресурсами. У цьому режимі сервер автоматично надсилає клієнту оновлення після зміни стану ресурсу, наприклад під час зміни температури сенсора [59].

Захист мережевого обміну в системі винесено на окремий транспортний рівень, який приховує криптографічні операції від прикладної логіки. Для з’єднань, що працюють поверх TCP, використовується TLS, а для протоколів із передаванням через UDP застосовується DTLS.

Під час створення сокета достатньо вибрати режим захищеного підключення, після чого системний рівень самостійно запускає узгодження параметрів сеансу. У межах цього процесу перевіряється чи віддалений вузол є справжнім, обираються криптографічні параметри та формується спільний ключ для подальшого обміну даними. Внутрішньо підсистема захисту транспортного рівня поділена на дві основні частини. Перша відповідає за встановлення сеансу та погодження параметрів захисту між сторонами. Друга обробляє вже прикладні дані, тобто розбиває їх на фрагменти, додає службову інформацію, виконує шифрування та забезпечує перевірку цілісності отриманих повідомлень. Структура дозволяє відокремити етап підготовки захищеного каналу від

безпосереднього передавання інформації. Дейтаграмний протокол захисту транспортного рівня виконує аналогічну роль для обміну, орієнтованого на протокол користувацьких датаграм. Це дає змогу використовувати захищене передавання там, де небажано переходити на протокол керування передаванням через вимоги до швидкодії, простоти або економії ресурсів. У результаті протоколи можуть зберігати властиву їм легкість, але водночас отримувати шифрування, автентифікацію та контроль незмінності даних під час роботи у відкритих мережах [59, 61].

Налаштування мережевого інтерфейсу може виконуватися без ручного введення параметрів. Для цього в системі використовується DHCP-механізм, який дає змогу пристрою самостійно отримати IP-адресу та інші необхідні дані для роботи в мережі. Обслуговування цього процесу винесено в окремий службовий потік. Він у фоновому режимі формує запити до сервера динамічного налаштування вузлів, очікує відповідь і після її отримання оновлює поточну мережеву конфігурацію [62]. Для роботи в локальних Ethernet-мережах додатково застосовується ARP. Його призначення полягає в тому, щоб визначати апаратну адресу пристрою за відомою адресою міжмережевого протоколу. Це необхідно, оскільки на рівні локальної мережі обмін відбувається не за логічними адресами міжмережевого протоколу, а через фізичні MAC-ідентифікатори мережевих вузлів [57].

Для захисту ОЗУ від виснаження під час мережевого обміну в архітектурі реалізовано блочний метод керування пам'яттю. Замість хаотичного виділення адресного простору, під потоки даних відводяться заздалегідь зорієнтовані пули з фіксованим розміром комірок. Це робить використання пам'яті передбачуваним, мінімізуючи її фрагментацію під час трафіку. Додатковим рівнем безпеки є відкидання пакетів у ситуаціях, коли всі буфери заповнені. У випадку вичерпання ліміту доступних блоків активується захисний алгоритм, який блокує прийом нових даних і просто відкидає їх. Завдяки цьому дається

стримати зростання навантаження на систему та захистити ядро від збоїв, спричинених дефіцитом ОЗУ [56].

2.4 Стандартизація інтерфейсів операційної системи

Концепція розробки архітектурно-незалежного системного програмного забезпечення базується на впровадженні абстрактного шару сумісності, що функціонує згідно з регламентами стандарту POSIX. Це рішення спрямоване на уніфікацію методів взаємодії підсистем із внутрішніми ресурсами платформи, включаючи дескриптори, файлові деривативи, комунікаційні інтерфейси та потоки виконання.

Шляхом виключення безпосереднього звернення програмного коду верхнього рівня до низькорівневих структур ядра або периферійних драйверів досягається ізоляція прикладної секції. Вбудовані системи критично залежать від подібного розподілу прав, оскільки він усуває необхідність переконфігурації та повторного проектування базових шарів апаратної підтримки, включаючи пакет підтримки плати, обробники апаратних переривань та системні таймери [2, 5, 39].

Крім того, завдяки тотожній моделі дескрипторів забезпечується сталість функціональних модулів, що дозволяє динамічно перенаправляти інформаційні потоки між різними фізичними каналами зв'язку без втручання у вихідний код прикладного програмного забезпечення [39].

Зважаючи на те, що архітектура базових мережевих, сервісних та файлових компонентів за замовчуванням орієнтована на POSIX-сумісні засоби, оптимізація процесів кросплатформеної адаптації зовнішніх бібліотек виступає основним фактором масштабованості системи. Зниження індексу зв'язаності модулів та збереження недоторканності логіки застосунків при розширенні їхніх можливостей є пріоритетним для якісного проєктування. У підсумку,

стандартизація інтерфейсної взаємодії повністю нівелює апаратну різнорідність обладнання.

2.5 Обґрунтування вибору мікроконтролера ESP32-C3 Super mini і допоміжних засобів

Для реалізації практичного етапу та подальшої оцінки працездатності створеної операційної системи реального часу як апаратну основу використано плату ESP32-C3 Super Mini (рис. 2.4). Такий вибір є виправданим у контексті IoT-рішень, адже цей модуль має компактні розміри, раціональне енергоспоживання, підтримку бездротових інтерфейсів, достатній обсяг пам’яті та сучасне процесорне ядро архітектури RISC-V. У цій роботі ESP32-C3 Super Mini розглядається не як платформа, яка не має альтернатив, а як типовий зразок вузла Інтернету речей, на базі якого можна перевірити запуск ядра, роботу драйверного рівня, системного таймера, послідовного порту та процедури завантаження прошивки [51].

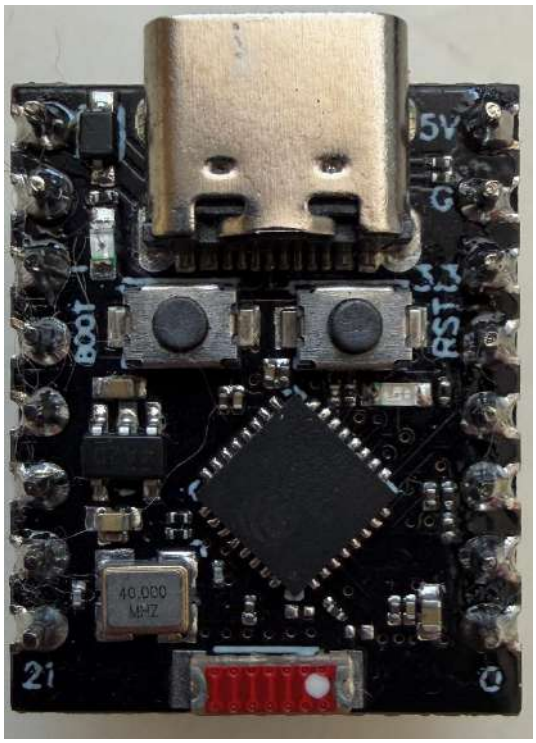


Рисунок 2.4 – Зовнішній вигляд плати ESP32-C3 Super Mini

Форм-фактор Super Mini робить цю плату придатною для застосування в малогабаритних вбудованих системах. Завдяки невеликим фізичним параметрам, зокрема 22,5 мм в довжину і 18 мм в ширину, її можна легко розміщувати в експериментальних макетах, портативних пристроях або корпусах з обмеженим внутрішнім простором [51]. На платі є інтерфейс USB Type-C, який спрощує подавання живлення, завантаження прошивки та початкову взаємодію з мікроконтролером у процесі розробки. Для практичної частини дипломної роботи ця особливість має суттєве значення, оскільки дає змогу записувати прошивку без використання складної зовнішньої апаратної обв'язки.

Таблиця 2.1 – Основні характеристики мікроконтролера ESP32-C3

Характеристика	Значення
Максимальна частота тактів	160 мГц
ОЗП	400 кБ
Вбудована flash-пам'ять	4 мБ

По периферійних можливостей ESP32-C3 Super Mini підтримує UART, SPI, I2C, PWM, ADC та апаратні переривання [51]. При цьому GPIO використовується як базовий інтерфейс керування цифровими виводами, а UART слугує для діагностики, командної взаємодії та роботи системної консолі.

Цей мікроконтролер підтримує системні таймери, апаратні переривання, механізми енергозбереження, контролери бездротового зв'язку та засоби безпечного завантаження. Для операційної системи подібна характеристика дійсно важлива, так ще й відповідає потребам в сфері IoT.

Перетворювач PL2303HX USB-UART TTL використовується як допоміжний апаратний засіб для зв'язку між комп'ютером і UART-інтерфейсом мікроконтролерної плати (рис. 2.5). Його призначення полягає в організації

каналу налагодження, моніторингу та передавання службових повідомлень. Через такий модуль можна контролювати запуск програмного забезпечення, аналізувати консольний вивід і перевіряти взаємодію прикладного коду з низькорівневими компонентами платформи [52].

Рисунок 2.5 – Зовнішній вигляд перетворювача PL2303HX USB-UART TTL

Основою модуля є мікросхема PL2303HX, яка виконує роль мосту між USB та асинхронним послідовним інтерфейсом UART з TTL-рівнями [52]. До комп'ютера перехідник під'єднується через USB Type-A, а після встановлення драйвера визначається операційною системою як віртуальний COM-порт. Це дозволяє працювати з ним через термінальні програми, монітори послідовного порту або інші засоби налагодження. Якщо згадати контакти, то він має 3V3, TXD, RXD, GND і +5V [52].

2.6 Інструментальні засоби розробки, збірки, завантаження та тестування системного ПЗ

Для реалізації системного програмного забезпечення було використано мову програмування C, оскільки вона є базовою мовою для більшості вбудованих операційних систем реального часу. Дуже на це ще вплинуло те, що потрібно працювати з апаратно-залежними структурами, вказівниками, областями пам'яті, регістрами периферійних пристроїв, обробниками переривань і низькорівневими системними API. Тут ця мова використовується для опису логіки, ініціалізації плати, роботи з GPIO, запуску потоків, виклику

функцій затримки, взаємодії з ядром та підключення драйверів. С дозволяє безпосередньо контролювати розміщення даних у пам’яті, передавати адреси структур ядра, працювати з макросами і формувати код, підходящий для виконання на мікроконтролері з обмеженими ресурсами [2, 5, 39].

Збірка ПЗ виконувалася за допомогою системи SCons. Вона практично є основною для автоматизованої компіляції, компонування та формування вихідних файлів прошивки. Тим більше вона враховує параметри, компілює окремі файли, виконує лінкування та створює фінальний виконуваний образ [53].

Також застосовано RISC-V toolchain, оскільки мікроконтролер взятий для тестування побудований на відповідному ядрі. Основним компілятором є riscv32-esp-elf-gcc, який перетворює все, що треба у машинний код, який вже підійде для мікроконтролера ESP32-C3. До складу цього засобу також входять асемблер, компонувальник, утиліти для роботи з ELF-файлами та інструменти діагностики, де останній використовується для розшифрування адрес із логів виконання [54]. За адресою програмного лічильника вдається визначити функцію та рядок коду, з якими пов’язана помилка або переривання виконання.

Для завантаження прошивки у flash-пам’ять мікропроцесора використовувалися консольний засіб esptool і графічна програма Espressif Flash Download Tool. Перший застосовувався для перевірки з’єднання з мікроконтролером, зчитування службової інформації про чип, стирання flash-пам’яті та запису бінарних файлів за визначеними адресами. Було враховано визначення чипа, перевірку MAC-адресу, тип flash-пам’яті, режим USB-Serial/JTAG і параметри підключення [51]. Espressif Flash Download Tool застосовувалась як альтернативний офіційний інструмент завантаження прошивки у середовищі Windows. Її графічним інтерфейсом дозволяє обирати усі потрібні налаштування враховуючи ергономіку використання.

Для контролю виконання прошивки після завантаження застосовувався послідовний монітор, зокрема esp_idf_monitor. Через цей засіб можна було спостерігати службовий вивід мікроконтролера на етапі запуску. У консолі

відображалися дані ROM-завантажувача, параметри основного завантажувача, структура таблиці розділів, адреса розміщення програмного образу, інформація про доступну динамічну пам’ять, а також збережене значення програмного лічильника після попереднього стану виконання [51].

Для додаткової перевірки роботи операційної роботи було задіяно QEMU. Суть його призначення полягала в емуляції апаратної платформи, на якій можна запустити операційну систему без фізичного пристрою [55]. Запуск у цій програму підтвердив працездатність самої розробленою ОС реального часу та її базових механізмів незалежно від конкретного мікроконтролера.

2.4 Висновки до другого розділу

У межах розділу 2 проведено аналіз архітектурної організації ОС реального часу, її базових програмних підсистем, апаратної платформи та інструментальних засобів, використаних для розробки, завантаження і перевірки працездатності програмного образу. Розглянуто підсистему ядра як сукупність сервісів керування потоками, системними об’єктами, часом, синхронізацією, міжпотоковим обміном, пам’яттю та апаратними перериваннями. Визначено, що робота потоків організовується через блоки керування, локальні стеки, пріоритетну модель планування і стани життєвого циклу, а вибір задачі для виконання здійснюється з урахуванням рівня пріоритету та механізму кругового обслуговування для потоків одного рівня. Окремо описано програмні таймери, які працюють на основі системного такту та можуть використовуватися для одноразового або періодичного запуску подій. Для синхронізації потоків розглянуто семафори, м’ютекси, набори подій, поштові скриньки та черги повідомлень, через які забезпечується узгоджений доступ до ресурсів, передавання коротких повідомлень і асинхронний обмін даними між задачами та обробниками переривань. Підсистема пам’яті подана через статичні пули і динамічне керування купою. Також розглянуто підсистему керування

апаратними ресурсами. Встановлено, що файлові об’єкти, пристрої та мережеві канали можуть подаватися через уніфіковану модель дескрипторів, а журнальні повідомлення передаються від клієнтського рівня до центрального модуля логування з подальшим виведенням через серверний компонент, зокрема UART. Мережева взаємодія описана через сокети, TCP/IP-стек LwIP, механізми MQTT і CoAP, а також TLS і DTLS. Визначено, що передавання даних може виконуватися і через вбудовані мережеві засоби, і через зовнішні модемні модулі, для яких запити перетворюються на текстові AT-команди. У розділі також обґрунтовано застосування POSIX-подібного інтерфейсу для стандартизації роботи.

Як апаратну основу практичної частини розглянуто плату ESP32-C3 Super Mini з мікроконтролером архітектури RISC-V. Допоміжним засобом визначено перетворювач PL2303HX USB-UART TTL, який використовується для зв’язку комп’ютера з UART-інтерфейсом плати, моніторингу запуску та аналізу консольного виводу. Для системного ПЗ описано використання мови C і методи збірки, потрібні для успішного виконання проєкту.

3 СИСТЕМНЕ ПРОГРАМНЕ ЗАБЕЗПЕЧЕННЯ ОПЕРАЦІЙНОЇ СИСТЕМИ РЕАЛЬНОГО ЧАСУ ДЛЯ ПРИСТРОЇВ ІОТ

3.1 Алгоритм запуску ядра операційної системи реального часу та ініціалізації системних компонентів

У межах ядра виділено основні функціональні підсистеми, які забезпечують виконання потоків, планування задач, обробку таймерів, міжпотокову взаємодію, керування пам’яттю та роботу з пристроями (рис. 3.1).

Рисунок 3.1 – Структура ядра та його основних компонентів

Завантаження ядра є початковим етапом функціонування операційної системи реального часу. Протягом цього процесу формується цілісне середовище, у якому далі відбудеться виконання потоків, буде працювати системний таймер, обробляються переривання, керується пам’ять і забезпечується взаємодія прикладного програмного коду з апаратною частиною. Тут система готує внутрішні структури ядра, ініціалізує апаратно-залежний рівень, створює основні службові потоки та переводить планувальник у робочий стан.

Алгоритм запуску ядра багаторівневий, при цьому кожен наступний етап в ньому залежить від коректного завершення попереднього. Центральною функцією цього процесу є головна точка входу, яка переймає керування обчислювальними механізмами безпосередньо після того, як зафіксує ініціалізацію апартної платформи.

На початку виконання операційна система переводиться в іозольований режим завантаження. Для цього локальні переривання тимчасово вимикаються за допомогою апаратно-залежної дії. Це потрібно для того, щоб під час формування внутрішнього стану ядра не виникали асинхронні події, які могли б звернутися до ще неініціалізованих структур, унеможливлюючи передчасне звернення до неіціалізованих структур. У момент старту ядра черги потоків, таймери, об’єкти синхронізації і планувальник ще не готові, тому будь-яке передчасне переривання могло б призвести до некоректної поведінки системи. Після стабілізації внутрішнього стану керування передається на рівень ініціалізації плати. Ця функція вже належить до рівня пакетів підтримки платформи і відповідає за її підготовку до роботи під керуванням операційної системи. Вона утворює зв’язок універсального ядра ОС реального часу з конкретним мікроконтролером. На цьому рівні виконується налаштування системного таймера, базових апаратних ресурсів, початкової конфігурації драйверів та консолі. Суттєвим в процесі збору програмного забезпечення виступає налаштування системного такту, оскільки без нього система не може коректно виконувати затримки потоків, облік часу, програмні таймери чи перемикання задач за квантом часу. Далі прив’язується таймер апарату до механізму системних тіків. Після кожного спрацювання таймера обробник переривання викликає збільшення такті і відповідно збільшує системний лічильник часу та запускає перевірку часових подій ядра.

Системні таймери на відміну від апаратних позиціонуються в розробленому проєкті як об’єкти ядра і забезпечують відкладений запуск функцій після завершення визначених інтервалів часу. Під час ініціалізації

створюються внутрішні списки таймерів, налаштовуються структури керування та готується механізм перевірки подій, час яких вже вийшов. Причому зазначені процедури відбуваються строго до активації планувальника, бо після переведення системи в багатопотоковий режим сервіси керування часом мають бути повністю доступними і для самого ядра, і для прикладних компонентів.

Завершення конфігурації таймерів дозволяє перейти до ініціалізації планувальника, що визначає, який потік повинен виконуватися в конкретний момент часу. На етапі ініціалізації створюються черги готових потоків, очищуються службові змінні планувальника, встановлюється початковий стан поточного потоку та готуються структури для подальшого перемикання контексту. До його запуску жоден користувацький або службовий потік ще фактично не виконується як окрема задача. Вони можуть бути створені й додані до системи, але реальне передавання керування між ними починається лише після самого стару в кінці процедури запуску.

Параментри розподілу ресурсів при створенні головного потоку залежать від поточної конфігурації підсистеми пам'яті. За умови активації динамічної пам'ятті, стек і блок керування потоку виділяються з купи. У протилежному випадку, якщо динамічна пам’ять вимкнена, система використовує статичні структури потоку і завчасно зарезервовані фіксовані масиви ОЗУ під стек. Завдяки цьому операційній системі і вдається працювати і у системах із повноцінним динамічним керуванням пам’яттю, і в обмежених вбудованих системах, де всі ресурси мають бути визначені на етапі компіляції

Після завантаження образу в пам’яті присутні сегмент лише для читання та сегмент читання і запису (рис.3.2). Сегмент лише для читання містить код програми й незмінні дані, тому він розміщується в області постійної пам’яті. Сегмент читання і запису містить ініціалізовані змінні, які під час виконання мають бути доступні для зміни, тому на етапі старту відповідні дані переносяться до оперативної пам’яті. Під час виконання також формується сегмент нульової ініціалізації, у якому розміщуються глобальні та статичні змінні з нульовим

початковим значенням. Після завершення цього сегмента починається область динамічної купи пам’яті, з якої ядро може виділяти пам’ять для потоків, об’єктів синхронізації, черг повідомлень, таймерів та інших системних структур.

Після створення головного потоку він переводить його у стан готовності до виконання. Тут ще потік не починає відразу виконувати основну функцію. Він лише додається до черги готових потоків планувальника. Фактичний запуск відбудеться пізніше, аж коли буде запущений планувальник і система обере цей потік як наступний для виконання. Тому запуск доволі впорядкований, адже спочатку ядро створює всі необхідні системні структури, а вже тільки потім дозволяє потокам отримувати процесорний час.

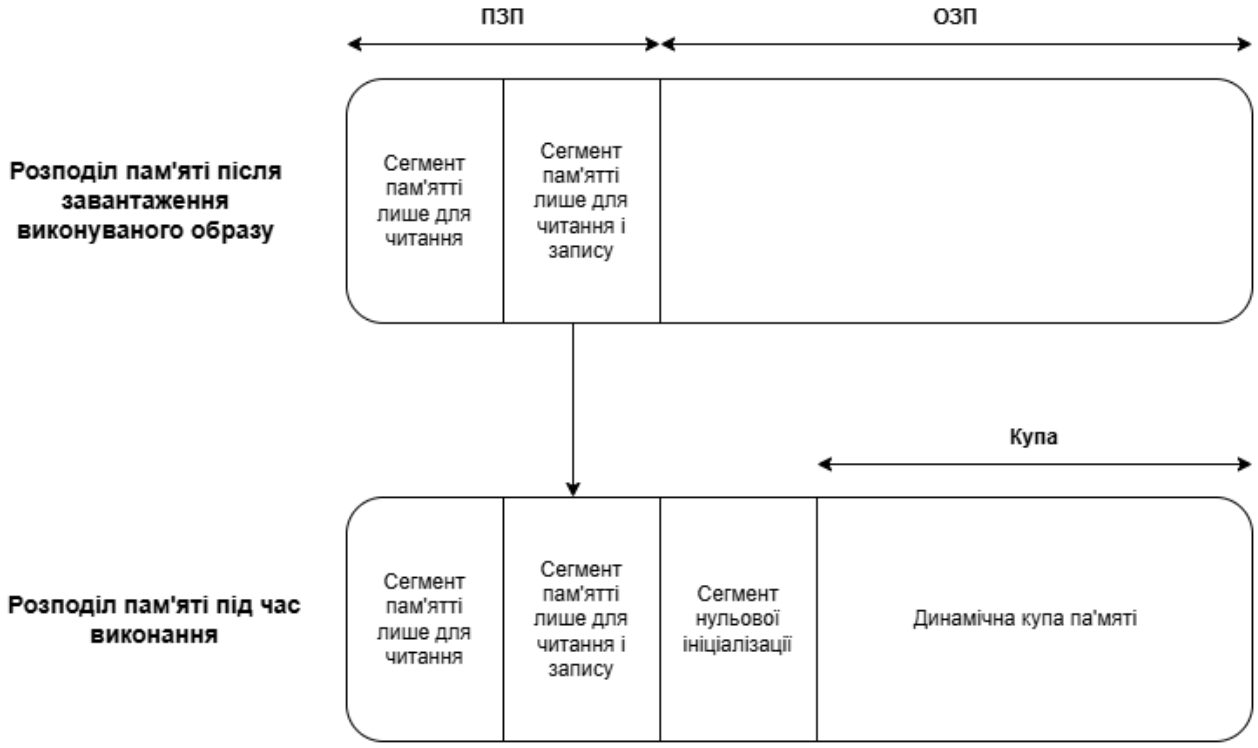


Рисунок 3.2 – Розподіл пам’яті під час виконання операційної системи реального часу

Перед викликом користувацької логіки функція входу головного потоку може ініціалізувати системні компоненти. Таким чином різні модулі системи можуть реєструвати власні функції запуску через спеціальні макроси. Під час

старту ці функції автоматично викликаються у визначеному порядку, що дозволяє підключати драйвери, файлові системи, командні рядки, мережеві компоненти та інші підсистеми без ручного виклику кожної функції в одному центральному файлі.

Після створення головного потоку відбувається ініціалізація службових потоків. Зокрема, за умови використання програмного таймеру, розгортається виділений потік таймерного сервісу. Цей об'єкт дозволяє виконувати функції зворотного виклику в окремому системному потоці. Це ж в свою чергу підвищує безпеку виконання, оскільки в контексті потоку дозволено виконувати більше операцій. Далі створюється фоновий потік простою. Він виконується обчислювальною системою виключно за відсутності інших готових до виконання потоків, забезпечую при цьому не тільки енергозбереження, але й очищення звільнених ресурсів завершених задач і обробку хуків простою.

Суть об’єктної моделі розробленої ОС в тому, що більшість сутностей системи, по типу потоків, таймерів, семафорів, м’ютексів, периферійних пристроїв і черг повідомлень, представлені як об’єкти ядра. Під час ініціалізації системи ці об’єкти реєструються у відповідних контейнерах, що дозволяє ядру здійснювати уніфіковане керування.

Останнім в алгоритмі є завантаження планувальника задач. Його активація спричиняє переведення системи із статичної ініціалізації у стан динамічного багатопотокового виконаня. Диспечер вибирає перший готовий потік з максимальним пріоритетом серед зареєстрованих і передає йому керування. Починаючи з цього моменту розроблена ОС функціонує як повноцінна операційна система реального часу з витісняючою багатозадачністю.

За умови надання головному потоку одного з найвищих рівнів пріоритету, планувальник обере його першим. У межах цього потоку виконується функція переходу до глобальної системної логіки, виконується каскадна ініціалізація компонентів і безпосередній виклик користувацької функції програми. Всі

подальші затримки, таймери, взаємодія з пристроями і перемикання між потоками відбуваються під керуванням планувальника.

3.2 Організація потоків, системного часу та планування виконання задач

Через організацію потоків розроблена ОС реального часу забезпечує багатозадачність, контроль виконання прикладних функцій і передбачувану реакцію на події. На відміну від послідовної програми, де всі дії виконуються в одному потоці керування, операційна система розділяє логіку системи на окремі потоки, кожен з яких має власний стековий простір, пріоритет, стан виконання та контекст процесора. Завдяки цьому вдається ізолювати функціональні частини програми. Наприклад, один потік може обробляти дані з периферійного пристрою, другий виконувати обмін повідомленнями, третій відповідати за індикацію або діагностику. Відповідно, диспетчеризація обчислювальних ресурсів регламентується станом потоків, ієрархією їх пріоритетів та поточними системними подіями.

Потік є об’єктом ядра, тому він має не лише функцію входу, але й службову структуру керування. У цій структурі зберігається символьний ідентифікатор потоку, його поточний стан, пріоритет, базовий покажчик, ліміт виділеного стеку, залишок кванту часу, локальні змінні помилки, а також посилання на двозв’язні списки ядра. Під час створення потоку система формує його початковий контекст так, щоб після першого вибору планувальником він міг почати виконання з визначеної функції входу.

Якщо потік створюється динамічно, ядро виділяє пам’ять саме під два базові компоненти, блок керування потоком та індивідуальний стек. При статичному створенні пам’ять для структур задається ядру заздалегідь, воно фактично лише ініціалізує їх і починає використовувати. Логіка виконання у них обох однакова, але різниця є у способі виділення ресурсів. Після створення потік не починає виконуватися автоматично. Спочатку він перебуває у початковому

стані, оскільки ядро лише підготувало його структуру та стек. Щоб потік став доступним для планувальника, він має бути переведеним у стан готовності і додатись до відповідної черги готових. Потік стає одним із тих, хто може отримати процесорний час. Реальне ж виконання починається вже тоді, коли планувальник обере його серед інших готових потоків. Тому можна створити кілька потоків під час ініціалізації системи, підготувати їх до роботи, а потім делегувати керування планувальнику, який сам визначить порядок виконання відповідно до пріоритетів.

Як вже було згадано, потік може перебувати в початковому стані, стані готовності, стані виконання, стані очікування або завершеному стані. Початковий стан характерний для щойно створеного потоку, який ще не додано до черги готових. Стан готовності означає, що потік має всі необхідні ресурси для виконання, але процесорний час наразі належить іншому потоку. Стан виконання має лише потік, який у цей момент фактично виконується процесором. Стан очікування виникає тоді, коли потік добровільно блокується, якщо він, наприклад, викликає затримку, очікує семафор, подію, повідомлення чи те ж завершення таймера. Завершений стан використовується для потоків, які виконали свою функцію або були примусово видалені. Перехід між цими станами відбувається через чітко визначені виклики ядра, системних тіків або подій синхронізації.

Планувальник працює за пріоритетним витісняючим принципом. Це означає, що серед усіх готових до виконання потоків система обирає потік з найвищим пріоритетом. У розробленій системі менше числове значення пріоритету відповідає вищому логічному пріоритету. Якщо під час роботи поточного потоку в систему переходить у стан готовності потік з вищим пріоритетом, планувальник може негайно виконати перемикання контексту.

Структура готових до виконання задач організована у вигляді масиву двозв'язних списків, де кожен індекс відповідає конкретному пріоритету. Для мінімізації затримок і оптимізації вибору планувальника використовується

бітова карта активних пріоритетів. Під час кожного циклу планування ядро перевіряє наявність готових потоків, визначає найпріоритетніший з них і порівнює його з поточним. Якщо виявлено розбіжності між дескрипторами, запускається механізм перемикання контексту. Під час перемикання зберігається контекст поточного потоку, фіксується вміст загальноцільових і службових регістрів, поточне значення покажчику стеку і можуть бути інші апаратно-залежні дані. Потім завантажується контекст нового потоку, а процесор продовжує виконання вже в межах іншої задачі.

Перемикання контексту має апаратно-залежну частину, оскільки набір регістрів і спосіб їх збереження залежить від архітектури процесора. У загальному ядрі визначається логіка планування, але низькорівневі операції перемикання реалізуються через апаратний порт і функції. Саме тому ядро може залишатися універсальним, а адаптація під конкретну архітектуру мікроконтролера реалізується на рівнях пакету підтримки плати і абстракції архітектури процесора. Тому кожен потік виконує свою функціональну логіку автономно, маючи можливість переходити в стан очікування ресурсу чи викликати затримку. При цьому, міжпотокове перемикання і диспечеризація здійснюються ядром автоматично, а багатозадачність на одному ядрі процесора досягається швидким і контрольованим перемиканням контекстів з мінімальними затримками.

Системний час базується на механізмі системного такту. Апаратний таймер періодично генерує переривання з частотою, визначеною параметром такту на секунду, зокрема один тік це 1 мілісекунда. Кожна ітерація обробки цього сигналу підвищує глобальний лічильник системних тіків ядра. Через нього реалізуються затримки потоків, перевірка таймерів, облік часового кванту і пробудження потоків після завершення очікування.

Коли потік викликає блокуючу функцію затримки, то ядро переводить його у стан очікування на задану кількість системних квантів часу. Апаратний дескриптор потоку вилучається, а потік додається до списку потоків, які мають

бути пробуджені після завершення часу затримки. Після цього викликається планувальник, який обирає інший готовий потік. Коли системний такт досягає потрібного значення, ядро повертає потік до стану готовності і він знову може бути обраний планувальником.

Крім простих затримок, у системі також реалізовано підтримку програмних таймерів. Програмний таймер функціонує як об’єкт ядра, призначений для запуску заданої функції зворотного виклику після завершення певного інтервалу часу. Програмні таймери реалізовано одноразовими або періодичними. Одноразовий після першого спрацювання автоматично зупиняється, а періодичний перезапускається для наступного відліку.

У розробленій ОС вони працюють на лічильнику системних тіків, тому їхня точність залежить від частоти такту на секунду. Під час ініціалізації системи готує структури для роботи таймерів. У цьому режимі функції зворотного виклику таймерів виконуються в окремому потоці, а не безпосередньо в контексті переривання.

Для кожного екземпляра таймера задається функція спрацювання, яка активується після завершення встановленого інтервалу часу (рис. 3.3). Залежно від режиму роботи таймера вона може виконуватися у різному контексті. Коли апаратний таймер працює в середовищі переривань, його функція спрацювання запускається безпосередньо після виникнення відповідної часової події. Натомість при використанні програмного таймеру, функція обробляється у спеціальному системному потоці, який виконує її в ізольованому системному контексті, зопобігаючи блокуванню критичних вузлів ядра.

Для ефективного керування задачами з однаковим рівнем пріоритету в розробленій ОС передбачено використання квантування часу. Кожному такому потоку виділяється певний фіксований інтервал, що вимірюється в кількості системних тактів. Якщо потік повністю вичерпує свій ліміт часу, він переміщується в кінець черги задач свого рівня, а планувальник віддає процесорний час наступному готовому потоку з цієї ж черги. При цьому, все

одно потоки з вищим пріоритетом мають перевагу і можуть витіснити поточний, навіть якщо той ще не встиг витратити свій поточний квант часу.

Рисунок 3.3 – Середовища роботи системного таймера

Фоновий потік простою наділений найнижчим рівнем пріоритетності й активується лише тоді, коли в системі відсутні інші готові до виконання задачі. Його наявність гарантує, що планувальник завжди має хоча б один потік для виконання. Окрім заповнення простою процесора, фоновий потік виконує службові функції по типу очищення ресурсів завершених потоків. Коли якась задача завершується, її пам'ять і структури даних не завжди можна безпечено видалити безпосередньо в її власному контексті. Для цього ядро запрограмовано так, щоб воно передавало ці дії потоку простою, який акуратно очищає залишені ресурси в момент мінімального навантаження на систему задач.

3.3 Організація синхронізації, міжпотокової взаємодії та керування ресурсами

Семафор складається з лічильника вільних ресурсів і черги задач, що очікують його звільнення. Коли потік ініціює запит на захоплення семафора, ядро спочатку перевіряє поточне значення цього лічильника. За умови, що воно перевищує нуль, він зменшиться, а потік продовжуватиме безперешкодне виконання без переходу в блокування. Якщо ж значення лічильника дорівнює нулю, негайне отримання ресурсу потоком неможливе, і розроблена ОС починає перевірку встановленого таймауту. Коли блокування потоку заборонене правилами виклику, функція миттєво повертає помилку. Якщо ж очікування дозволене, потік переводиться у призупинення, додається до списку очікування семафора, а планувальник обирає інший готовий потік.

Прописана логіка звільнення семафора залежить від того, чи є потоки у списку очікування. Якщо ця черга порожня, ядро просто збільшує поточне значення лічильника семафора. Якщо ж хоча б один потік чекає на нього, то відбудеться вилучення одного потоку із списку очікування, потім він переведеться у стан готовності та додасться до черги відповідного пріоритету. Враховуючи, що потік, який щойно прокинувся, може мати вищий пріоритет ніж поточний, у цей момент позачергово викликається планувальник. У цьому випадку одразу спрацьовує механізм витіснення, а керування обчислювальним процесом передається задачі, яка щойно одержала необхідний доступ. Тобто можна зрозуміти, що життєвий цикл семафора включає основні операції створення, отримання, вивільнення та видалення (рис. 3.4).

М’ютекс у розробленій ОС використовується для взаємовиключного доступу до ресурсу. Причому його логіка прописана так, що при успішному захопленні об’єкта зберігається посилання на поточний потік. Якщо цей самий потік знову захоплює цю структуру, тоді вже використовується лічильник

повторного захоплення м'ютеса, що дозволяє уникнути блокування м'ютексу ним же самим при повторному вході в захищену ділянку коду. Звільнення м’ютекса зменшить цей лічильник, а звільнення ресурсу буде тільки тоді, коли лічильник повернеться до нуля.

Механізм подій у розробленій ОС застосувується тоді, коли потоку необхідно очікувати не один ресурс, а певну комбінацію умов чи системних сигналів. Структура об’єкта події базується на бітовій масці, де кожен окремий розряд відповідає за конкретну подію або поточний стан системи. При цьому потік може очікувати на появу будь-якого з указаних бітів або на одночасну появу всіх заданих розрядів. Під час запиту на очікування ядро перевіряє чи умова задовольняється. Якщо потрібні бітові прапорці вже встановлені, то функція повертається одразу. У протилежному випадку потік реєструється у черзі очікування цього об'єкта і просто блокується. Коли інша задача або обробник апаратного переривання встановлює відповідний прапорець, операційна система аналізує список потоків очікування і пробуджує ті, для яких задана логічна умова стала істинною.

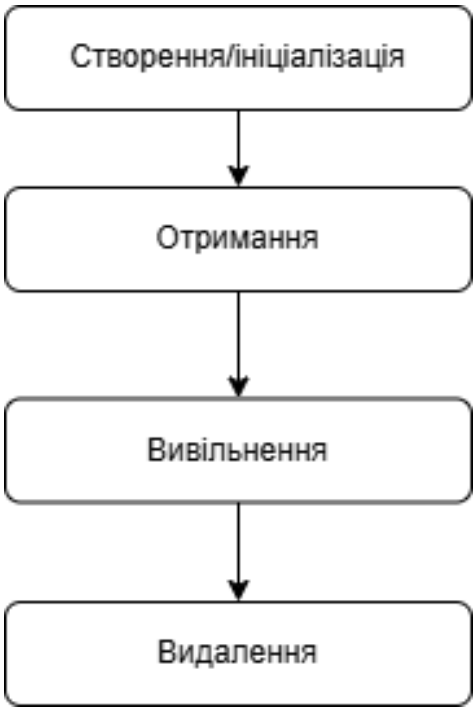


Рисунок 3.4 – Основні операції над семафором

Взаємодія потоків в розробленому ПЗ реалізована через поштові скриньки і черги повідомлень. Перші орієнтовані на передачу малих даних, наприклад, покажчиків, кодів подій або коротких числових значень. Під час відправлення повідомлення через цей механізм ядро перевіряє вільне місце у внутрішньому буфері скриньки. Якщо вільний простір знайшовся, повідомлення записується, якщо ж буфер заповнений, то потік-відправник отримає помилку і буде заблокованих поки не знайдеться вільна для нього пам'ять незалежно від параметрів. Черга ж повідомлень призначена для передачі більш об'ємних пакетів даних і функціонує з блоками фіксованої довжини. Під час її створення задається розмір одного повідомлення та кількість повідомлень, які можуть зберігатися в черзі. Алгоритм роботи схожий на поштову скриньку, але додатково тут ще буде відбуватись копіювання даних у внутрішній буфер черги або з нього відповідно. Якщо потік викликає отримання повідомлення, а черга порожня, він може бути переведений у стан очікування (рис. 3.5). Коли інший потік надсилає повідомлення, ядро записує його в чергу і пробуджує один із потоків-одержувачів.

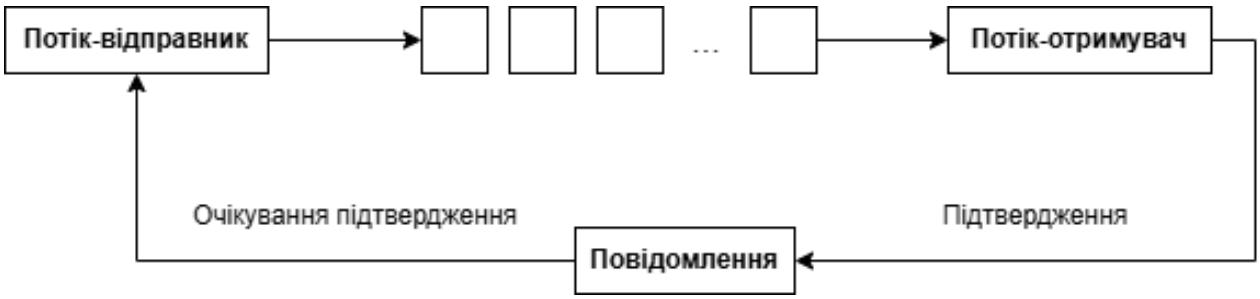


Рисунок 3.5 – Передавання даних між потоками через чергу повідомлень

Усі об’єкти міжпотокової взаємодії у розробленій ОС реального часу працюють через уніфіковані механізми керування списками очікування. Коли потік не може негайно отримати ресурс, він вилучається з переліку готових до виконання потоків і додається до черги призупинених відповідного об’єкта. Черговість розміщення потоків всередині цього списку визначається або за

критерієм їх пріоритетів, або за принципом хто перший зайде, той так само першим і вийде. Якщо використовується перший варіант, то при звільненні ресурсу першим пробуджується потік із найвищим пріоритетом. Якщо використовується другий варіант, то потоки пробуджуються у порядку свого надходження. Після цього потік повертається до черги готовності, а планувальник приймає рішення щодо подальшого виконання. Тобто міжпотокова взаємодія справді не дублює планувальник, а динамічно змінюють множину потоків, які мають бути доступними для планування.

Під час створення об’єкта ядро виділяє пам’ять для його службової структури, ініціалізує поля, задає тип об’єкта і реєструє його в контейнер об’єкту. Під час видалення об’єкта система повинна коректно звільнити пам’ять і прибрати об’єкт із відповідних списків ядра. Таким чином забезпечується організація всіх системних сутностей дра (рис. 3.6).

Рисунок 3.6 – Об’єктна модель ядра

Підсистема керування ресурсами в розробленій системі розроблена з врахуванням потреми створення системних об’єктів і випадків, коли компоненти потребують окремі області ОЗУ.

Для невеликих вбудованих систем використовується простий механізм роботи з купою, яка виділяється під час старту системи у вигляді суцільного масиву оперативної пам'яті. Під час ініціалізації ядро фіксує початкову і кінцеву адреси простору, формує керуючі таблиці стану блоків і відкриває доступ іншим сервісам для розміщення пам'яті під потоки виконання, черги повідомлень, програмні таймери, інструменти синхронізації та системні буфери. Під час обробки запиту, відбувається аналіз наявності вільних сегментів і підбір відповідних розмірів. Якщо знайдений блок більший ніж потрібно, то він розділяється на робочу область і залишковий вільний блок, який повертається назад у пул. Коли життєвий цикл об’єкта закінчується, пам’ять відповідно звільняється теж.

Для складніших реалізовано паралельну взаємодію з кількома незалежними частинами пам'яті. Її алгоритм передбачає, що кожна виділена область сама по собі автономна в функціонанні, маючи власні адресні межі, ізольовані службові структури та внутрішні правила розподілу.

3.4 З’єднання апаратних компонентів і тестування роботи ОС реального часу

Першим етапом практичної перевірки стало підготування апаратури, на яку в подальшому відбудеться завантаження прошивки. Плата ESP32-C3 Super Mini початково була без виводів, що у свою чергу ускладнювало і унеможливлювало з’єднання її з потрібним перехідником. Для вирішення цього за допомогою звичайного паяльника було прикріплено штирьові виводи в кількості шістнадцяти штук. Здійснення цієї маніпуляції уможливило

використання з’єднувальних дротів типу female-female і звичайно забезпечило доступ до потрібних контактів без притискання дротів до тих же контактних майданчиків.

Для організації додаткового каналу обміну було взято PL2303HX перетворювач USB-UART TTL. Його підключення до мікроконтролерної плати було зроблено через сигнальні лінії UART і спільне для обох міні пристроїв заземлення (табл. 3.1). При цьому лінії передавання і приймання потрібно з’єднувати навхрест, тобто TXD перехідника підключається до RXD плати і навпаки. Також обов’язково з’єднується GND, бо цього зв’язок через UART працюватиме некоректно або взагалі не запуститься. Живлення з PL2303HX не підключалось нікуди, оскільки мікроконтролер ESP32-C3 було під’єднано через Type-C шнур, який підтримує і передачу даних, і живлення. Ця характеристика кабелю суттєва, адже інакше він би не дозволив комп’ютеру визначити плату як пристрій і створювати відповідний COM-порт. У середовищі Windows після підключення перевірялася поява відповідного COM-порту, оскільки саме через нього надалі виконувалися команди запису прошивки та перегляд стартових повідомлень.

Таблиця 3.1 – Основні апаратні з’єднання пристроїв

Порт плати ESP32-C3 Super Mini	Порт перетворювача PL2303HX
GND	GND
GPIO 20	TXD
GPIO 31	RXD

Поки мікроконтролер не був прошитий, виникала проблема з його постійним підключенням і відключенням. Її вдалося усунути шляхом утримання кнопки BOOT протягом 7 секунд, подальшого перезапуску плати кнопкою

RESET та відпускання кнопки BOOT із затримкою приблизно 3 секунди. Що ж підсумую, що складені пристрої показано на рисунку 3.7.

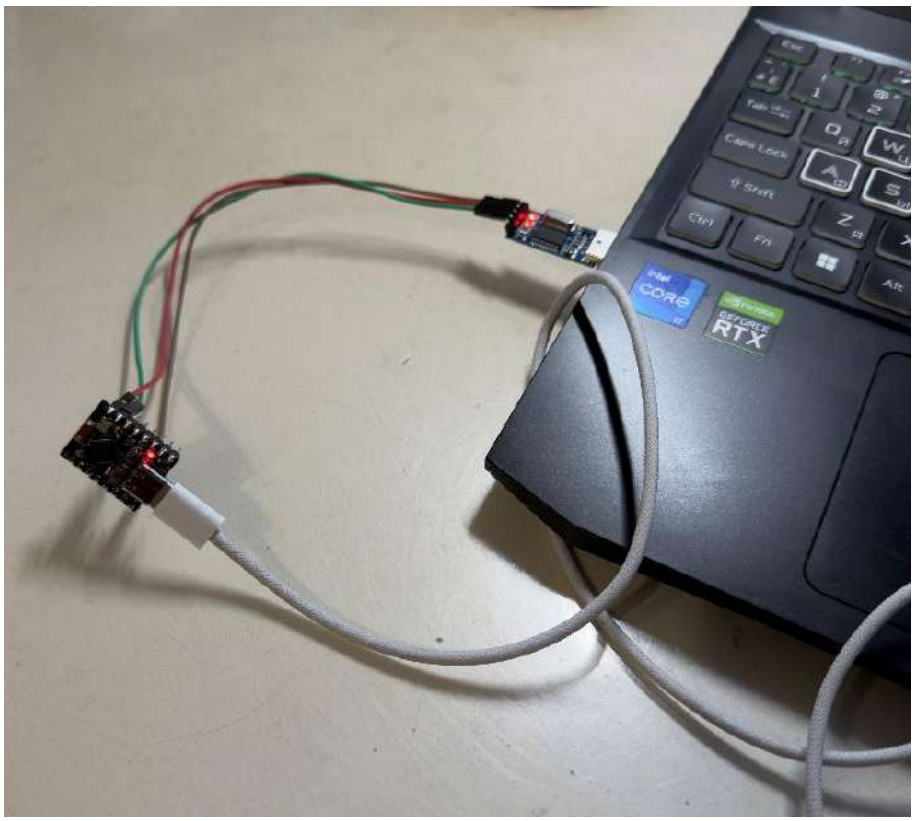


Рисунок 3.7 – З’єднання апаратних пристроїв

Після підключення тестового стенду було перевірено, чи взагалі операційна система ноутбука правильно визначає обидва пристрої USB-UART перехідник і вбудований USB-інтерфейс мікроконтролерної плати. Успішний результат їх виявлення наведено на рисунку 3.8.

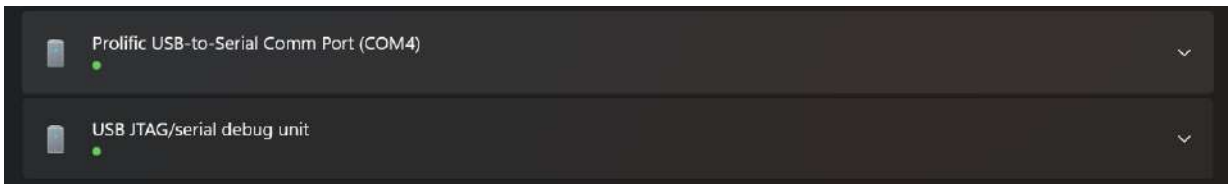


Рисунок 3.8 – Скріншот успішного виявлення ноутбуком підключень девайсів

Взагалі для прошивки використовувалось кілька різних способів. Почати варто з завантаження через консоль esptool, яка хоч і не дає графічний інтерфес, але не менш зручна (рис. 3.9). Звичайно перед виконанням запису спочатку перевірялося чи плата доступна через відповідний COM-порт. Такий крок потрібний для того, аби переконатися, що комп'ютер бачить мікроконтролер, інша програма не займає порт, а сам пристрій може перейти в режим обміну з засобом прошивання. Якщо порт був зайнятий монітором послідовного виводу або іншою програмою, доступ до плати міг блокуватися, тому перед прошиванням такі інструменти закривалися. Після перевірки з'єднання виконувалося завантаження трьох основних бінарних файлів у вбудовану flash-пам'ять. Завантажувач розміщувався за адресою 0x0, оскільки саме з цієї області після скидання починається початковий етап запуску ПЗ. Таблиця розділів записувалася у комірку пам'яті 0x8000. Вона показує, як поділена flash-пам'ять і за якими адресами знаходяться потрібні частини прошивки. А основний файл прошивки записувався у місце 0x10000, тобто в область прикладної програми, з якої передається подальше керування після закінчення роботи завантажувача. Для плати вказувався ключові характеристики, а саме тип чипу esp32c3, третій COM-порт, режим DIO для послідовного виводу і вводу даних.

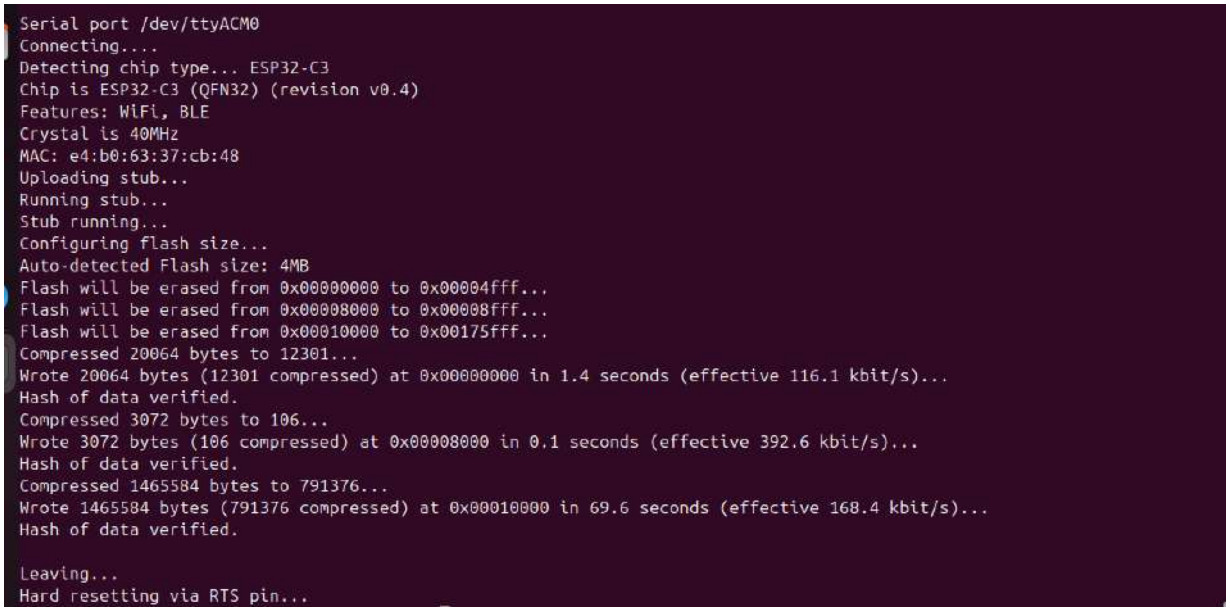


Рисунок 3.9 – Скріншот успішного запису прошивки за допомогою esptool

Щоб перевірити інший варіант завантаження програмного образу, було взято вже графічний інструмент Espressif Flash Download Tool. У противагу попередньому методу, тут не потрібне введення команд у термінал, бо всі параметри задаються безпосередньо у відповідні комірки на екрані. Алгоритм в цілому такий самий. Перед початком завантаження здійснилась перевірка чи стоять усі праворці біля потрібних рядків, тоюто чи активні вони. Після натискання кнопки старту програма послідовно записувала всі обрані бінарні файли, відображався поточний стан, а після завершення з’являлося повідомлення про успішне виконання операції.

Завантаживши програмний образ, вдалось виконати перевірку роботи операційної системи через системну консоль. На цьому етапі треба було підтвердити не лише сам факт успішного запуску, але й чи працюють основні механізми. Тестування коректність обробки завдань ядром включає в себе реакцію на команди та відповідно їх виконання, завдяки цьому можна зрозуміти чи доступні системні сервіси і чи створені необхідні службові потоки.

Спочатку було перевірено роботу команд пов'язаних з мережею. Вдалось успішно підключити плату до точки доступу на телефоні. При цьому для інтерфесу активовано DHCP, завдяки чому плата автоматично отримала IP-адресу, шлюз, маску підмережі і DNS-сервер. Далі виконано перевірку доступності мережі за допомогою пінгування. Спочатку відбулась успішна перевірка зв’язку із шлюзом, після чого було передано 4 пакети, які усі пройшли без втрат. Вдалось підтвердити наявність доступу до зовнішньої мережі інтернету. Крім цього вдалось звернутись до google.com, пінг вдало перетворив доменне ім’я на IP-адресу і підтвердив правильну роботу DNS.

У консольному виводі відображалися основні показники її використання: загальний доступний обсяг, уже зайнята частина, максимальне зафіксоване використання, а також поточний залишок вільної пам’яті. Це, логічно слідуючи, дозволило оцінити, як система розподіляє пам’ять після запуску та виконання

базових команд. Додатково було проведено моніторинг активних потоків, що дозволило проаналізувати їхні ключові параметри: пріоритетність, поточний статус, конфігурацію стеку (загальний обсяг та рівень заповнення), часові ліміти, коди помилок та вказівники на блоки керування. Наявність у сформованому звіті сервісних задач, системної оболонки, а також фонового та таймерного потоків є прямим підтвердженням коректної ініціалізації планувальника та здатності ядра забезпечувати повноцінне багатозадачне середовище. Результати перевірки наведено на рисунках 3.10 та 3.11.

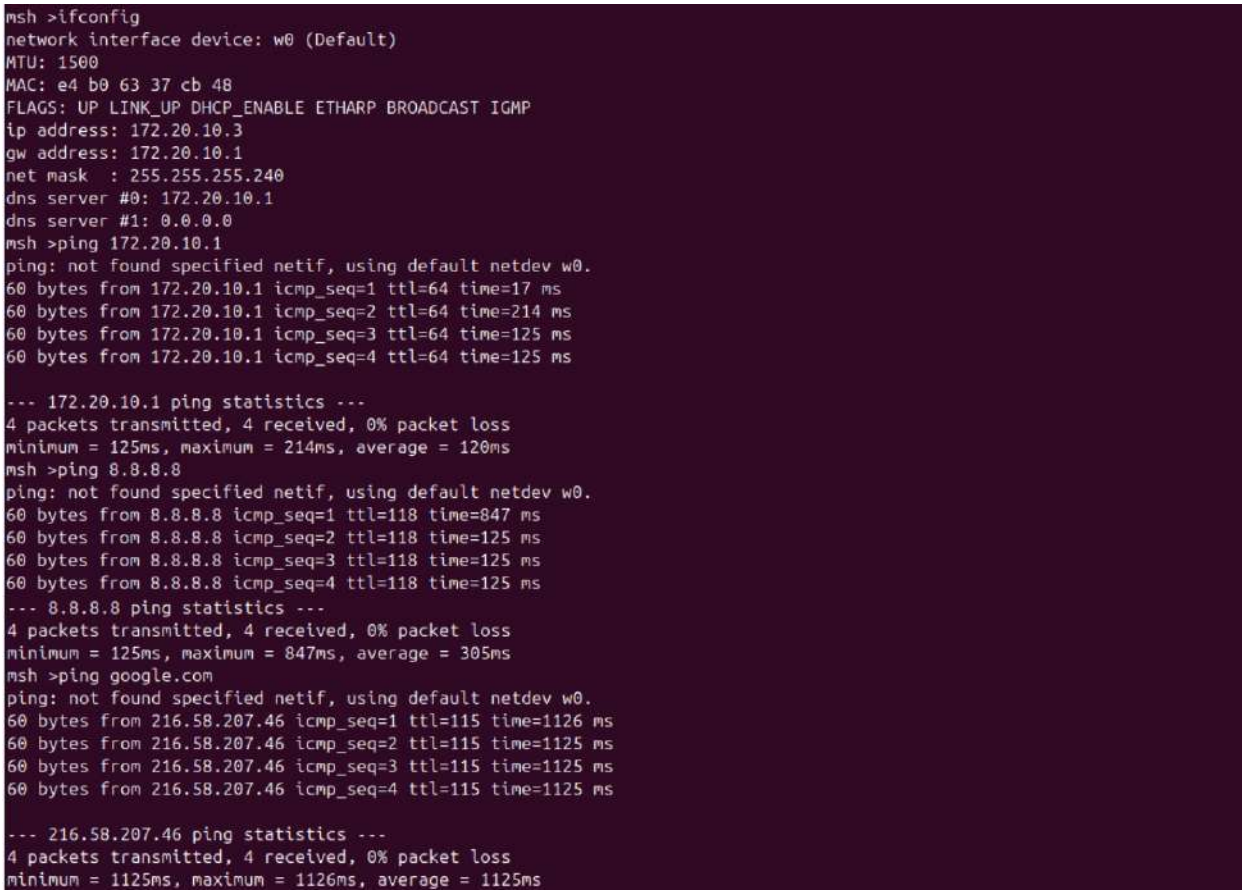


Рисунок 3.10 – Скріншот успішного тестування команд роботи з мережею

З отриманого консольного виводу видно, що операційна система приймає команди користувача, виконує мережеві операції, відображає актуальний стан пам’яті та надає інформацію про активні потоки. Отже, базові механізми

операційної системи реального часу працюють коректно і можуть використовуватися для подальшого тестування.

```
msh >list thread
thread       pri  status      sp     stack size max used left tick   error  tcb addr   usage
------------ ---  ------- ---------- ----------  ------  ---------- ------- ---------- -----
wifi           1  suspend 0x00000160 0x00004000     10%   0x00000001 EINTRPT 0x3fcb2738  N/A
sys_evt        4  suspend 0x00000170 0x00000b00     24%   0x00000001 EINTRPT 0x3fcb0cb8  N/A
esp_timer      2  suspend 0x000000f0 0x00001000     10%   0x00000001 OK      0x3fcaf7e8  N/A
tshell        20  running 0x00000120 0x00003000      9%   0x00000001 OK      0x3fcac728  N/A
wlan          15  suspend 0x00000120 0x00000c00     23%   0x00000008 OK      0x3fcab700  N/A
sys workq     23  suspend 0x00000120 0x00001000      7%   0x0000000a OK      0x3fcaa5e0  N/A
tcpip         10  suspend 0x00000150 0x00000800     35%   0x00000013 EINTRPT 0x3fca9cc0  N/A
etx           12  suspend 0x00000120 0x00000600     33%   0x0000000c EINTRPT 0x3fcc1518  N/A
erx           12  suspend 0x00000120 0x00000600     18%   0x00000010 EINTRPT 0x3fcc0e58  N/A
tidle0        31  ready   0x00000090 0x00001000      4%   0x00000015 OK      0x3fcc53b8  N/A
timer          4  suspend 0x00000100 0x00000400     26%   0x00000009 EINTRPT 0x3fcc6ac0  N/A
main          10  suspend 0x00000130 0x00002000     25%   0x00000014 EINTRPT 0x3fca7b38  N/A
msh >free
total    : 76704
used     : 62744
maximum  : 63272
available: 13960
msh >list device
device              type         ref count
------------ -------------------- ----------
w0           Network Interface    1
wlan1        Network Interface    0
wlan0        Network Interface    1
uart         Character Device     2
pin          Pin Device           0
```

Рисунок 3.11 – Успішно пройдене тестування роботи розробленої ОС реального часу.

3.6. Висновки до третього розділу

У межах розділу 3 відбулась безпосередня розробка системного програмного забезпечення, підготовка апаратних складових і тестування успішності роботи розробленої ОС. Тому вдалось підтвердити, що ядро переходить від етапу початкового налаштування до режиму виконання потоків у контрольованій послідовності. Також було визначено, що організація пам’яті, модель потоків, системний такт, програмні таймери та пріоритетний витісняльний планувальник формують базу для багатозадачного виконання, а механізми семафорів, м’ютексів, подій, поштових скриньок і черг повідомлень забезпечують синхронізацію та обмін даними між потоками. При цьому успішно

вдається використовувати основні базові функції взаємозв’язку із мережею коректно.

Практична перевірка на ESP32-C3 Super Mini із використанням PL2303HX USB-UART TTL показала вдале налагодження через послідовний інтерфейс і консольний вивід. Усі потрібні бінарні файли запуску, вдалось записати, а пізніше підтвердилась робота ОС. Отже, отримані результати дали змогу зробити висновок, що усі компоненти системного ПЗ функціонують узгоджено.

ВИСНОВКИ

За результатами виконаних у роботі теоретичних та практичних досліджень було досліджено архітектуру та особливості функціонування багатозадачної операційної системи реального часу для пристроїв Інтернету речей, а також виконано її розробку і практичну перевірку запуску, системних механізмів і взаємодії з апаратною платформою.

У першому розділі проаналізовано будову операційних систем для IoT-пристроїв і досліджено багатозадачність, перемикання контексту, пріоритети, роботу потоків та вплив обмежених ресурсів IoT-пристроїв.

У другому розділі виконано компонентне проєктування ОС реального часу для IoT. Визначено складові ядра, підсистему керування апаратними ресурсами, файлові операції, мережевий обмін, журналювання та енергозбереження. Окремо розглянуто прикладний інтерфейс, який спрощує взаємодію з різними апаратними платформами.

У третьому розділі описано розробку й тестування системного програмного забезпечення. Подано запуск ядра, ініціалізацію апаратного рівня, таймера, планувальника та потоків. Перевірено роботу системного часу, динамічної пам’яті, синхронізації й міжпотокової взаємодії. У практичній частині зібрано тестові пристрої, зокрема підключено ESP32-C3 Super Mini та USB-UART перехідник, після чого успішно прошито програмний образ.

Отже, у дипломній роботі вдалось виконати синтез багатозадачної операційної системи реального часу для пристроїв IoT, який відповідає вимогам і має практично підтверджену працездатність.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАНЬ

1. Thyagaturu A. S., Shantharama P., Nasrallah A., Reisslein M. Operating systems and hypervisors for network functions: a survey of enabling technologies and research studies. *IEEE Access*. 2022. Vol. 10. P. 1. URL: https://doi.org/10.1109/access.2022.3194913 (дата звернення: 12.02.2026).

2. Comer D. Operating System Design: The Xinu Approach. Boca Raton: Chapman and Hall/CRC, 2025. 547 p.

3. Robot Operating System 2: Design, architecture, and uses in the wild / S. Macenski et al. *Science Robotics*. 2022. Vol. 7, no. 66. URL: https://doi.org/10.1126/scirobotics.abm6074 (дата звернення: 12.02.2026).

4. Folgado F. J., Calderón D., González I., Calderón A. J. Review of Industry 4.0 from the perspective of automation and supervision systems: definitions, architectures and recent trends. *Electronics*. 2024. Vol. 13, No. 4. P. 782. URL: https://doi.org/10.3390/electronics13040782 (дата звернення: 15.02.2026).

5. Chakraborty P. Operating Systems: Evolutionary Concepts and Modern Design Principles. Boca Raton: Chapman and Hall/CRC, 2023. 618 p.

6. Bin Akhtar Z. Operating Systems (OS): An Insight Investigative Research Analysis and Future Directions. *Journal of Technology and Informatics (JoTI)*. 2024. Vol. 6, no. 1. P. 58–69. URL: https://doi.org/10.37802/joti.v6i1.637 (дата звернення: 15.02.2026).

7. Chen L., Zhang Y., Tian B., Ai Y., Cao D., Wang F. Y. Parallel driving OS: a ubiquitous operating system for autonomous driving in CPSS. *IEEE Transactions on Intelligent Vehicles*. 2022. Vol. 7, No. 4. P. 886–889. URL: https://doi.org/10.1109/tiv.2022.3223728 (дата звернення: 15.02.2026).

8. Agal S. Fundamentals of Operating Systems. Dabra, Gwalior, M.P., India: Xoffencer International Publication, 2023. 195 p.

9. Kode O., Oyemade T. *Analysis of synchronization mechanisms in operating systems*. 2024. P. 10–13. URL: https://arxiv.org/pdf/2409.11271. (дата звернення: 15.02.2026).

10. Jaeger T. Operating System Security. Cham: Springer Nature, 2022. 230 p.

11. Akhtar Z. B. Securing operating systems (OS): a comprehensive approach to security with best practices and techniques. *International Journal of Advanced Network, Monitoring and Controls*. 2024. Vol. 9, No. 1. P. 106–107.

12. Ammari H. M. An overview of sensing hardware, standards, operating systems, software development, and applications and systems. *Theory and Practice of Wireless Sensor Networks*: *Cover, Sense, and Inform*. Cham: Springer, 2022. P. 685–727.

13. Xu J., Lin H., Yuan Z., Shen W., Zhou Y., Chang R. et al. RegVault: hardware assisted selective data randomization for operating system kernels. *Proceedings of the 59th ACM/IEEE Design Automation Conference*. 2022. P. 715–720.

14. González-Rodríguez M., Otero-Cerdeira L., González-Rufino E., Rodríguez-Martínez F. J. Study and evaluation of CPU scheduling algorithms. *Heliyon*. 2024. Vol. 10, No. 9. P. 5–7. URL: https://doi.org/10.1016/j.heliyon.2024.e29959 (дата звернення: 15.02.2026).

15. A novel intelligent round robin CPU scheduling algorithm / P. S. Sharma et al. *International Journal of Information Technology*. 2021. URL: https://doi.org/10.1007/s41870-021-00630-0 (дата звернення: 15.02.2026).

16. Abu-Dalbouh H. M. A new combination approach to CPU scheduling based on priority and round-robin algorithms for assigning a priority to a process and eliminating starvation. *International Journal of Advanced Computer Science and Applications*. 2022. Vol. 13, No. 4. P 542–543.

17. Singhal A., Varshney S., Mohanaprakash T. A., Jayavadivel R., Deepti K., Reddy P. C. S., Mulat M. B. Minimization of latency using multitask scheduling in industrial autonomous systems. *Wireless Communications and Mobile Computing*. 2022. Vol. 2022, No. 1. P. 1–10. URL: https://doi.org/10.1155/2022/1671829 (дата звернення: 15.02.2026).

18. Sliwko L. *A taxonomy of schedulers: operating systems, clusters and big data frameworks*. 2025. P. 7. URL: https://arxiv.org/pdf/2511.01860. (дата звернення: 15.02.2026).

19. Chaaban K. A New Algorithm for Real-Time Scheduling and Resource Mapping for Robot Operating Systems (ROS). *Applied Sciences*. 2023. Vol. 13, no. 3. P. 1532. URL: https://doi.org/10.3390/app13031532 (дата звернення: 14.02.2026).

20. An integrated AGV control system using preemptive and non-preemptive mixed RTOS / D. Chen et al. *The Journal of Supercomputing*. 2024. URL: https://doi.org/10.1007/s11227-024-06193-8 (дата звернення: 14.02.2026).

21. Dubey M. K. B., Arora N., Yadav M. R., Tewari M. J. Process Management. *Operating System: Principles and Applications*. Palwal, Haryana, India: Chyren Publication, 2025. P. 43–45.

22. Patil A. P., Kittali M. R. M. Operating System. Chennai, Tamil Nadu, India : Chendur Publishing House, 2022. 283 p.

23. Guo J., Wu S., Chen H., Huang S., Zhang Y. Application analysis of process state control in modern operating system. *Academic Journal of Computing & Information Science*. 2024. Vol. 7, No. 6. P. 13–19.

24. Kong L., Tan J., Huang J., Chen G., Wang S., Jin X. et al. Edge-computing-driven internet of things: a survey. *ACM Computing Surveys*. 2022. Vol. 55, No. 8. P. 1–41.

25. Baras K., Brito L., Hassan Q. F. Internet of Things. *Advances in the Internet of Things: Challenges, Solutions, and Emerging Technologies*. 2025. P. 1–3.

26. Guillemin P., Berens F., Carugi M., Barthel H., Dechamps A., Rees R. et al. Internet of Things global standardisation: state of play. *Internet of Things Applications: From Research and Innovation to Market Deployment*. Aalborg : River Publishers, 2022. P. 143–197.

27. Albreem M. A., Sheikh A. M., Bashir M. J., El-Saleh A. A. Towards green Internet of Things (IoT) for a sustainable future in Gulf Cooperation Council countries:

current practices, challenges and future prospective. *Wireless Networks*. 2023. Vol. 29, No. 2. P. 539–567.

28. Salam A. Internet of things for sustainable community development: introduction and overview. *Internet of Things for Sustainable Community Development: Wireless Communications, Sensing, and Systems*. Cham : Springer International Publishing, 2024. P. 1–31.

29. Paolone G., Paesani R., Camplone J., Piazza A., Di Felice P. Conceptualization of IoT architectures. *International Journal of Informatics and Communication Technology (IJ-ICT).* 2025. Vol. 14, No. 1. P. 334–346.

30. Lechqar K., Errais M. A conventional IoT architecture: precision agriculture as domain of application. *International Journal on Engineering Applications*. 2023. Vol. 11, No. 6. P. 6–8.

31. Yusupbekov N. R., Gulyamov S. M., Muxamedxanov U. T., Kuziev Z. Z. Some features of the modern concept of the Internet of Things. *Journal of Siberian Federal University. Engineering & Technologies*. 2023. Vol. 16, No. 5. P. 636–643.

32. Domínguez-Bolaño T., Campos O., Barral V., Escudero C. J., García-Naya J. A. An overview of IoT architectures, technologies, and existing open-source projects. *Internet of Things*. 2022. Vol. 20. P. 13. URL: https://doi.org/10.1016/j.iot.2022.100626 (дата звернення: 19.04.2026).

33. Muradova A., Normatova D., Khamraeva G. Results of analytical experiments on the operation of internet of things technology protocols. *AIP Conference Proceedings*. 2024. Vol. 3244, No. 1. Article 030008. P. 20–21.

34. Choudhary V., Tanwar S. A Concise Review on Internet of Things: Architecture and Its Enabling Technologies. *Lecture Notes in Electrical Engineering*. Singapore, 2023. P. 443–456. URL: https://doi.org/10.1007/978-981-19-8493-8_34 (дата звернення: 19.04.2026).

35. Elgedawy I., Shoukry L. SANAD: a comprehensive IoT reference architecture for integrated enterprise platforms. *2022 IEEE Global Conference on Artificial Intelligence and Internet of Things (GCAIoT)*. IEEE, 2022. P. 182–187.

36. Shahinzadeh G., Shahinzadeh H., Tanwar S. Security and privacy issues in the internet of things: a comprehensive survey of protocols, standards, and the revolutionary role of blockchain. *2024 8th International Conference on Smart Cities, Internet of Things and Applications (SCIoT)*. IEEE, 2024. P. 59–67.

37. Baras K., Brito L., Hassan Q. F. Internet of things: architectures, technologies, applications, and challenges 1. *Advances in the Internet of Things*. 2025. P. 3–35.

38. Deniziak S., Płaza M., Arcab Ł. Approach for designing real-time IoT systems. *Electronics*. 2022. Vol. 11, No. 24. P. 4120. URL: https://doi.org/10.3390/electronics11244120 (дата звернення: 19.05.2026).

39. Zrelli A. Hardware, software platforms, operating systems and routing protocols for Internet of Things applications. *Wireless Personal Communications*. 2022. Vol. 122, No. 4. P. 3889–3912.

40. Mansour M., Gamal A., Ahmed A. I., Said L. A., Elbaz A., Herencsar N., Soltan A. Internet of things: a comprehensive overview on protocols, architectures, technologies, simulation tools, and future directions. *Energies*. 2023. Vol. 16, No. 8. P. 3465. URL: https://doi.org/10.3390/en16083465 (дата звернення: 19.04.2026).

41. Aziz Al Kabir M., Elmedany W., Sharif M. S. Securing IoT devices against emerging security threats: challenges and mitigation techniques. *Journal of Cyber Security Technology*. 2023. Vol. 7, No. 4. P. 1–25. URL: https://doi.org/10.1080/23742917.2023.2228053 (date of access: 19.05.2026).

42. Bakhshi T., Ghita B., Kuzminykh I. A review of IoT firmware vulnerabilities and auditing techniques. *Sensors*. 2024. Vol. 24, No. 2. P. 708. URL: https://doi.org/10.3390/s24020708 (дата звернення: 19.05.2026).

43. Aqeel M., Ali F., Iqbal M. W., Rana T. A., Arif M., Auwul M. R. A review of security and privacy concerns in the internet of things (IoT). *Journal of Sensors*. 2022. Vol. 2022, No. 1. P. 1–20. URL: https://doi.org/10.1155/2022/5724168 (дата звернення: 15.02.2026).

44. Gangolli A., Mahmoud Q. H., Azim A. A systematic review of fault injection attacks on IoT systems. *Electronics*. 2022. Vol. 11, No. 13. P. 2023. URL: https://doi.org/10.3390/electronics11132023 (дата звернення: 16.02.2026).

45. Baho S. A., Abawajy J. Analysis of consumer IoT device vulnerability quantification frameworks. *Electronics*. 2023. Vol. 12, No. 5. P. 1176. URL: https://doi.org/10.3390/electronics12051176 (дата звернення: 15.02.2026).

46. Gyamfi E., Jurcut A. Intrusion detection in internet of things systems: a review on design approaches leveraging multi-access edge computing, machine learning, and datasets. *Sensors*. 2022. Vol. 22, No. 10. P. 3744. URL: https://doi.org/10.3390/s22103744 (дата звернення: 16.02.2026).

47. Purushothaman G. Integrating Secure Firmware Updates into the Hardware Lifecycle of Smart Devices. 2025. 15 p. URL: https://ssrn.com/abstract=5598770. (дата звернення: 16.02.2026)

48. Wijesundara W. M. A. B., Lee J. S., Tith D., Aloupogianni E., Suzuki H., Obi T. Security-enhanced firmware management scheme for smart home IoT devices using distributed ledger technologies. *International Journal of Information Security*. 2024. Vol. 23, No. 3. P. 1927–1937.

49. Kenaza R., Khemane A., Bendjenna H., Meraoumia A., Laimeche L. Internet of things (IoT): architecture, applications, and security challenges. *2022 4th International Conference on Pattern Analysis and Intelligent Systems (PAIS)*. IEEE, 2022. P. 1–5.

50. Mallegowda M., Sarashetti P., Kanavalli A. SOA-based middleware framework for IoT applications. *Proceedings of Second International Conference on Sustainable Expert Systems: ICSES 2021*. Singapore: Springer Nature Singapore, 2022. P. 315–327.

51. Cameron N. ESP32 Microcontroller. *ESP32 Formats and Communication*. Berkeley, CA, 2023. P. 1–54. URL: https://doi.org/10.1007/978-1-4842-9376-8_1 (дата звернення: 03.04.2026).

52. Merugu S., Kumar A., Ghinea G. Hardware, Component, Description. *Advanced Technologies and Societal Change*. Singapore, 2022. P. 31–48. URL: https://doi.org/10.1007/978-981-19-1264-1_5 (дата звернення: 19.04.2026).

53. statacons: An SCons-based build tool for Stata / R. P. Guiteras et al. *The Stata Journal: Promoting communications on statistics and Stata*. 2023. Vol. 23, no. 1. P. 148–196. URL: https://doi.org/10.1177/1536867x231162032 (дата звернення: 19.04.2026).

54. Gao Y., Qian W., Cui E. RISC-V ISA Extension Toolchain Supports: A Survey. *CNIOT'23: 2023 4th International Conference on Computing, Networks and Internet of Things*, Xiamen China. New York, NY, USA, 2023. URL: https://doi.org/10.1145/3603781.3603942 (дата звернення: 21.04.2026).

55. Bekele Y. B., Limbrick D. B., Kelly J. C. A Survey of QEMU-based Fault Injection Tools & Techniques for Emulating Physical Faults. *IEEE Access*. 2023. P. 1. URL: https://doi.org/10.1109/access.2023.3287503 (дата звернення: 22.04.2026).

56. Research on Embedded Network Security Defense Mechanism Based on the Synergy of Cortex-M MPU and LwIP / L. Zhang et al. *2026 IEEE 8th International Conference on Communications, Information System and Computer Engineering (CISCE)*, Guangzhou, China, 27–29 March 2026. 2026. P. 467–472. URL: https://doi.org/10.1109/cisce69494.2026.11504465 (дата звернення: 19.04.2026).

57. Timothy Murkomen. Performance, privacy, and security issues of TCP/IP at the application layer: A comprehensive survey. *GSC Advanced Research and Reviews*. 2024. Vol. 18, no. 3. P. 234–264. URL: https://doi.org/10.30574/gscarr.2024.18.3.0106 (дата звернення: 23.04.2026).

58. An intelligent sustainable efficient transmission internet protocol to switch between User Datagram Protocol and Transmission Control Protocol in IoT computing / S. Mahmoodi Khaniabadi et al. *Expert Systems*. 2022. URL: https://doi.org/10.1111/exsy.13129 (дата звернення: 23.04.2026).

59. Sarkar C., Das A., Jain R. K. Development of CoAP protocol for communication in mobile robotic systems using IoT technique. *Scientific Reports*. 2025. Vol. 15, no. 1. URL: https://doi.org/10.1038/s41598-024-76713-2 (дата звернення 16.04.2026).

60. Lakshminarayana S., Praseed A., Thilagam P. S. Securing the IoT Application Layer from an MQTT Protocol Perspective: Challenges and Research Prospects. *IEEE Communications Surveys & Tutorials*. 2024. P. 1. URL: https://doi.org/10.1109/comst.2024.3372630 (date of access: 24.04.2026).

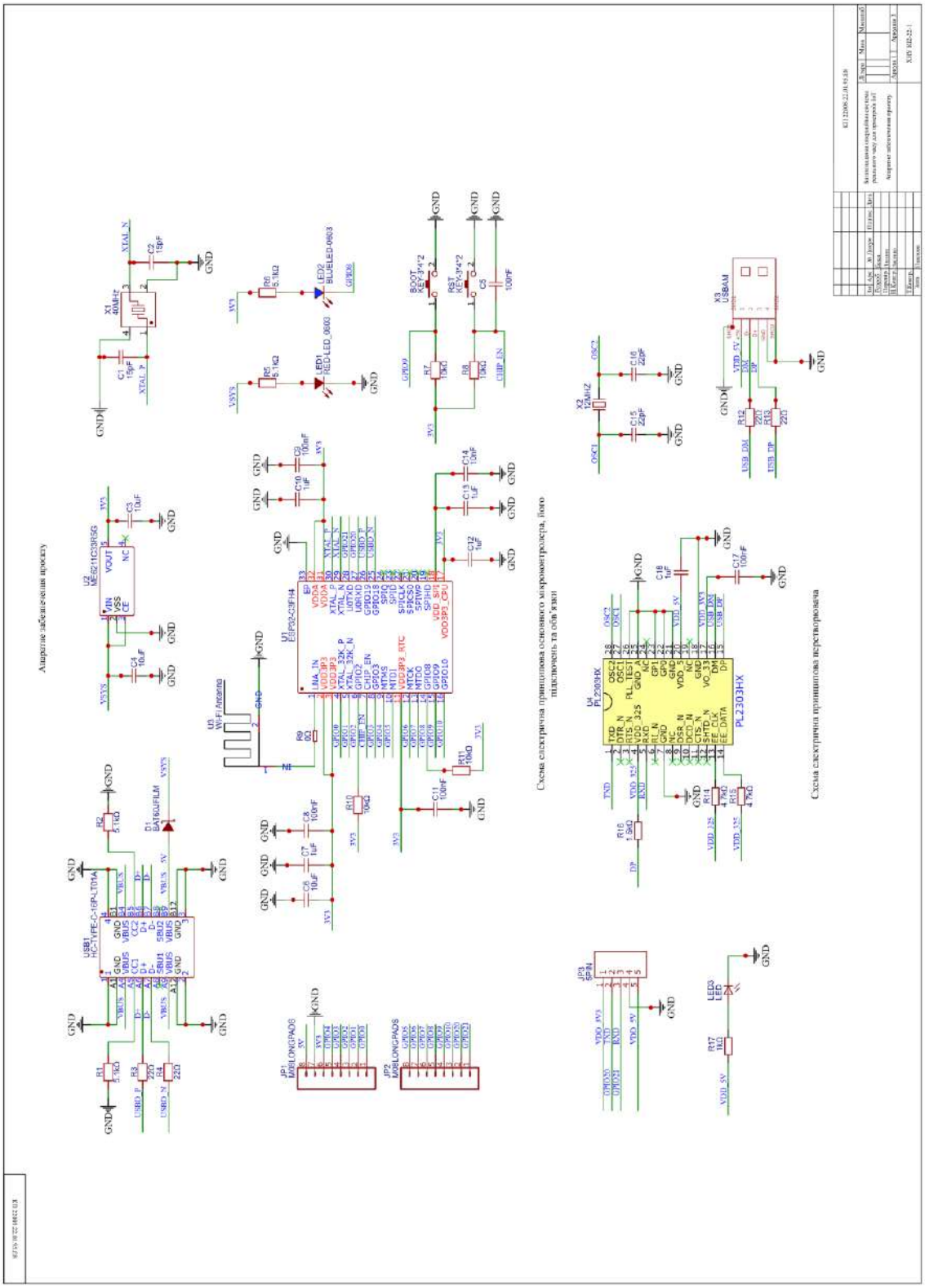
61. Pohlmann N. Transport Layer Security (TLS)/Secure Socket Layer (SSL). *Cyber-Sicherheit*. Wiesbaden, 2022. P. 439–473. URL: https://doi.org/10.1007/978-3-658-36243-0_11 (дата звернення: 23.04.2026).

62. AbdulGhaffar A., Paul S. K., Matrawy A. An Analysis of DHCP Vulnerabilities, Attacks, and Countermeasures. *2023 Biennial Symposium on Communications (BSC)*, Montreal, QC, Canada, 4–7 July 2023. 2023. URL: https://doi.org/10.1109/bsc57238.2023.10201458 (дата звернення: 23.04.2026).

ДОДАТОК А

(обовязковий)

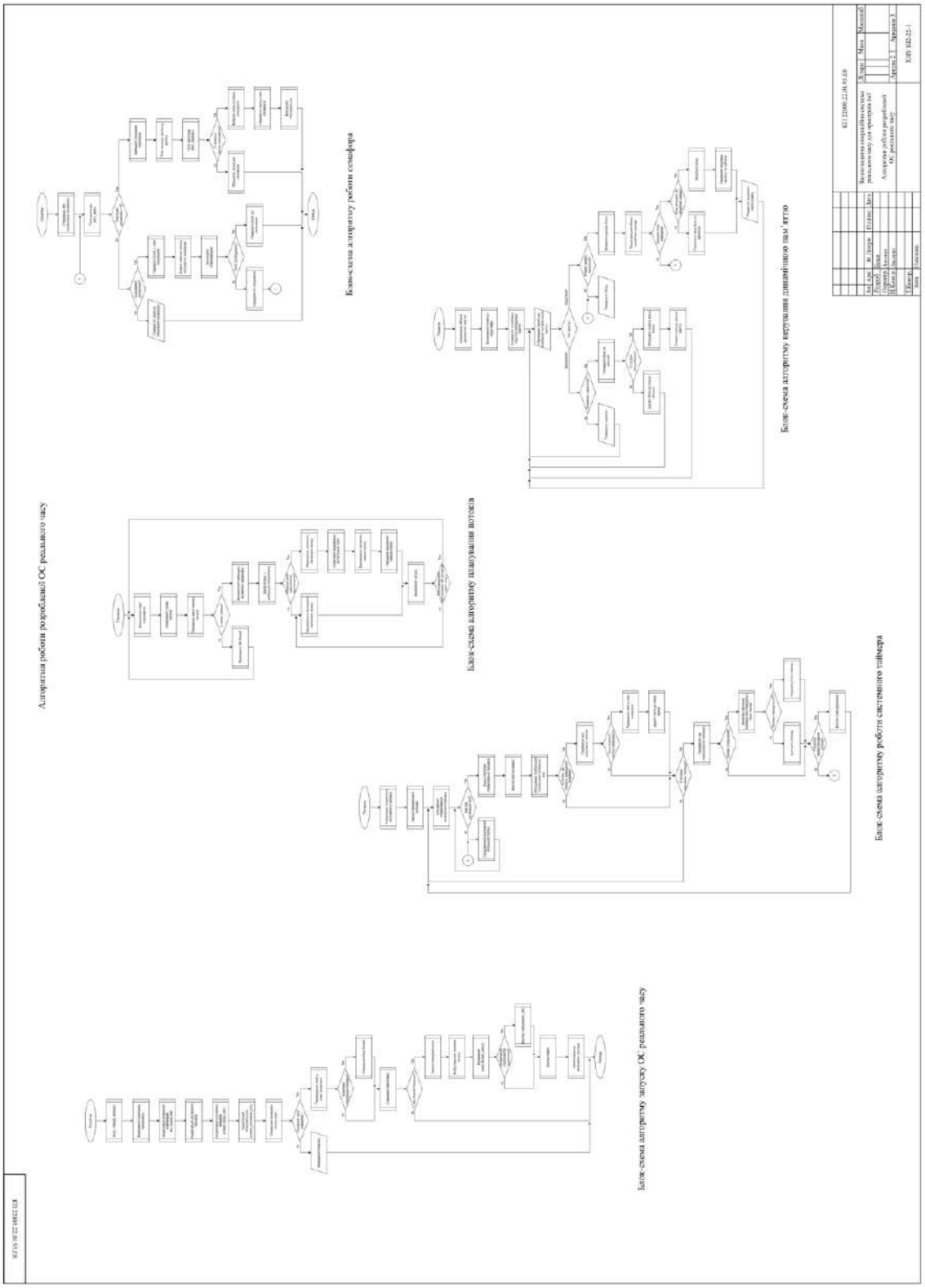
Копія креслення «Апаратне забезпечення проєкту»



ДОДАТОК Б

(обов’язковий)

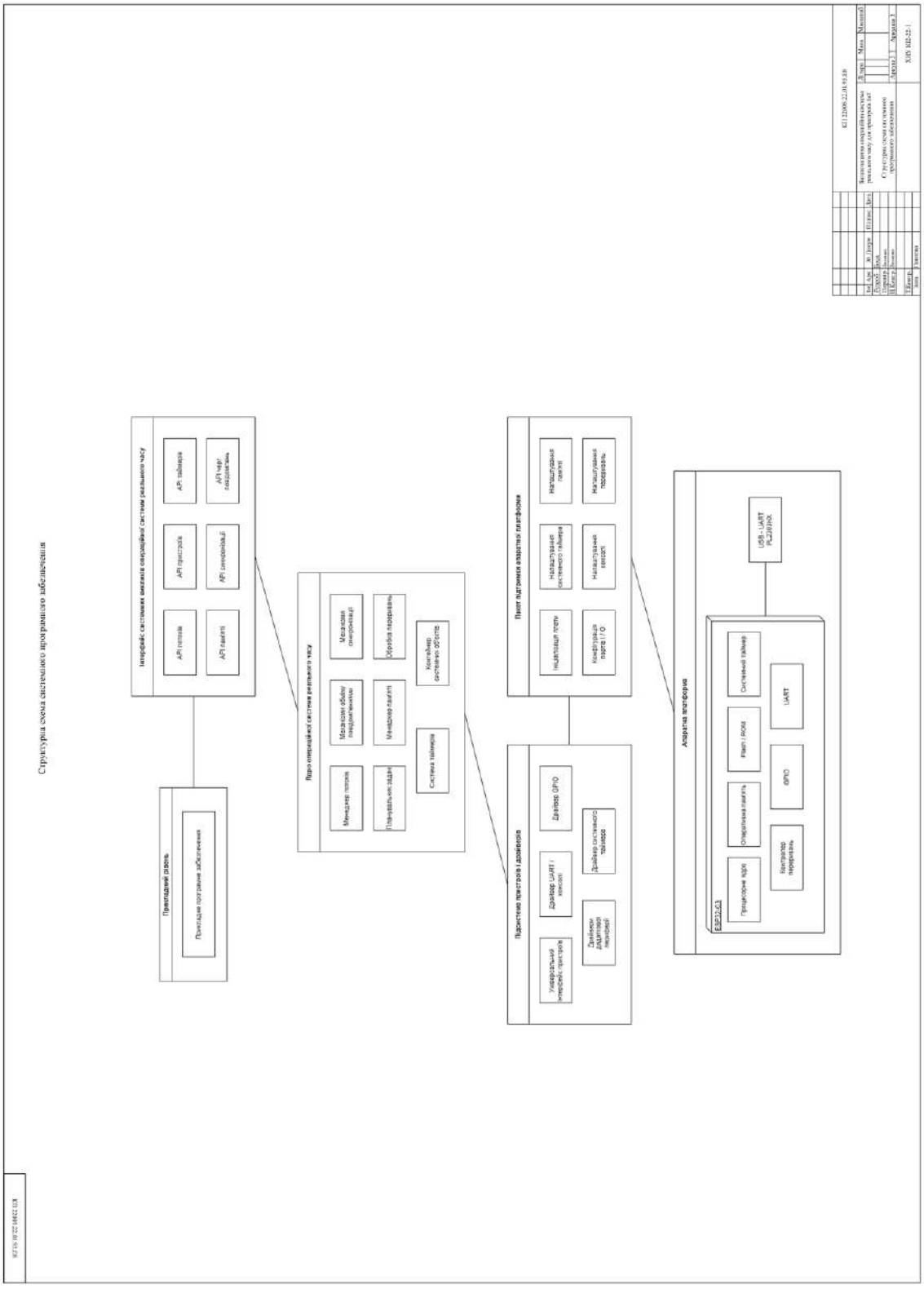
Копія креслення «Алгоритми роботи розробленої ОС реального часу»



ДОДАТОК В

(обов’язковий)

Копія креслення «Структурна схема системного програмного забезпечення»



ДОДАТОК Г

(обов’язковий)

Лістинг вихідного коду програмно-технічного засобу

main.c:

```
#ifdef BSP_USING_BLE
void app_main(){
    while(1)    {
    }
}
#endif
int main(void){
    pin_mode(BSP_LED_PIN, PIN_MODE_OUTPUT);
#ifdef BSP_USING_BLE
    extern void app_main(void);
    esp_timer_init();
    app_main();
#endif
#ifdef USING_WIFI
    wlan_set_mode(WLAN_DEVICE_STA_NAME, WLAN_STATION);
    wlan_set_mode(WLAN_DEVICE_AP_NAME, WLAN_AP);
#endif
    while (1) {
        pin_write(BSP_LED_PIN, PIN_HIGH);
        thread_mdelay(1000);
        pin_write(BSP_LED_PIN, PIN_LOW);
        thread_mdelay(1000);
    }
}
```

board.c:

```
static systimer_hal_context_t systimer_hal;
IRAM_ATTR                void                SysTickIsrHandler(void                *arg){
systimer_ll_clear_alarm_int(systimer_hal.dev, SYSTIMER_LL_ALARM_OS_TICK_CORE0);
    interrupt_enter();
    tick_increase();
    interrupt_leave();
}
void  hw_systick_init(void){         esp_intr_alloc(ETS_SYSTIMER_TARGET0_EDGE_INTR_SOURCE,
ESP_INTR_FLAG_IRAM | ESP_INTR_FLAG_LEVEL1, SysTickIsrHandler, &systimer_hal, NULL);
    periph_module_enable(PERIPH_SYSTIMER_MODULE);
    systimer_hal_init(&systimer_hal);
    systimer_hal_tick_rate_ops_t ops = {
            .ticks_to_us = systimer_ticks_to_us,
            .us_to_ticks = systimer_us_to_ticks,
    };                          systimer_hal_set_tick_rate_ops(&systimer_hal,       &ops);
systimer_ll_set_counter_value(systimer_hal.dev,      SYSTIMER_LL_COUNTER_OS_TICK,      0);
systimer_ll_apply_counter_value(systimer_hal.dev,            SYSTIMER_LL_COUNTER_OS_TICK);
systimer_hal_connect_alarm_counter(&systimer_hal,         SYSTIMER_LL_ALARM_OS_TICK_CORE0,
SYSTIMER_LL_COUNTER_OS_TICK);                 systimer_hal_set_alarm_period(&systimer_hal,
SYSTIMER_LL_ALARM_OS_TICK_CORE0,          1000000UL          /          TICK_PER_SECOND);
systimer_hal_select_alarm_mode(&systimer_hal,             SYSTIMER_LL_ALARM_OS_TICK_CORE0,
SYSTIMER_ALARM_MODE_PERIOD);    systimer_hal_counter_can_stall_by_cpu(&systimer_hal, 1, 0,
true);     systimer_hal_enable_alarm_int(&systimer_hal, SYSTIMER_LL_ALARM_OS_TICK_CORE0);
systimer_hal_enable_counter(&systimer_hal, SYSTIMER_LL_COUNTER_OS_TICK);
```

```
}
void hw_board_init(void){
    hw_systick_init();
#ifdef USING_COMPONENTS_INIT
    components_board_init();
#endif
#if defined(USING_CONSOLE) && defined(USING_DEVICE)
    console_set_device(CONSOLE_DEVICE_NAME);
#endif
}
static gptimer_handle_t gptimer_hw_us = NULL;
static int delay_us_init(void)
{
    gptimer_config_t timer_config = {
        .clk_src = GPTIMER_CLK_SRC_DEFAULT,
        .direction = GPTIMER_COUNT_UP,
        .resolution_hz = 1 * 1000 * 1000,
    };              ESP_ERROR_CHECK(gptimer_new_timer(&timer_config,    &gptimer_hw_us));
ESP_ERROR_CHECK(gptimer_enable(gptimer_hw_us));
    return EOK;
}
INIT_DEVICE_EXPORT(delay_us_init);
void hw_us_delay(uint32_t us){
    uint64_t   count   =   0;             ESP_ERROR_CHECK(gptimer_start(gptimer_hw_us));
ESP_ERROR_CHECK(gptimer_set_raw_count(gptimer_hw_us, 0));
    while(count < (uint64_t)us)
    {        ESP_ERROR_CHECK(gptimer_get_raw_count(gptimer_hw_us, &count));
    }    ESP_ERROR_CHECK(gptimer_stop(gptimer_hw_us));
}
```

ipc.c:

```
#ifdef USING_USER_MAIN
#ifndef MAIN_THREAD_STACK_SIZE
#define MAIN_THREAD_STACK_SIZE     2048
#ifndef MAIN_THREAD_PRIORITY
#define MAIN_THREAD_PRIORITY
#ifdef USING_COMPONENTS_INIT
static int start(void){
    return 0;
}
INIT_EXPORT(start, "0");
static int board_start(void){
    return 0;
}
INIT_EXPORT(board_start, "0.end");
static int board_end(void){
    return 0;
}
INIT_EXPORT(board_end, "1.end");
static int end(void){
    return 0;
}
INIT_EXPORT(end, "6.end");
void components_board_init(void){
#ifdef DEBUGING_AUTO_INIT
    int result;
    const struct init_desc *desc;
    for (desc = &__init_desc_board_start; desc < &__init_desc_board_end; desc ++) {
        kprintf("initialize %s\n", desc->fn_name);
        result = desc->fn();
```

```
        kprintf(":%d done\n", result);
    }
#else
    volatile const init_fn_t *fn_ptr;
    for (fn_ptr = &__init_board_start; fn_ptr < &__init_board_end; fn_ptr++){
        (*fn_ptr)();
    }
#endif
}
void components_init(void){
#ifdef DEBUGING_AUTO_INIT
    int result;
    const struct init_desc *desc;
    kprintf("do components initialization.\n");
    for (desc = &__init_desc_board_end; desc < &__init_desc_end; desc ++){
        kprintf("initialize %s\n", desc->fn_name);
        result = desc->fn();
        kprintf(":%d done\n", result);
    }
#else
    volatile const init_fn_t *fn_ptr;
    for (fn_ptr = &__init_board_end; fn_ptr < &__init_end; fn_ptr ++){
        (*fn_ptr)();
    }
#endif
}
#endif
#ifdef USING_USER_MAIN
void application_init(void);
void hw_board_init(void);
int thread_startup(void);
#ifdef __ARMCC_VERSION
extern int $Super$$main(void);
int $Sub$$main(void){
    thread_startup();
    return 0;
}
#elif defined(__ICCARM__)
extern void __iar_data_init3(void);
int __low_level_init(void){
    __iar_data_init3();
    thread_startup();
    return 0;
}
#elif defined(__GNUC__)
int entry(void){
    thread_startup();
    return 0;
}
#endif
#ifndef USING_HEAP
align(ALIGN_SIZE)
static uint8_t main_thread_stack[MAIN_THREAD_STACK_SIZE];
struct thread main_thread;
#endif
static void main_thread_entry(void *parameter){
    extern int main(void);
    UNUSED(parameter);
#ifdef USING_COMPONENTS_INIT
    components_init();
#endif
```

```
#ifdef USING_SMP
    hw_secondary_cpu_up();
#endif
#ifdef __ARMCC_VERSION
    {
        extern int $Super$$main(void);
        $Super$$main();
    }
#elif    defined(__ICCARM__)    ||    defined(__GNUC__)    ||    defined(__TASKING__)    ||
defined(__TI_COMPILER_VERSION__)
    main();
#endif
}
void application_init(void){
    thread_t tid;
#ifdef USING_HEAP
    tid          =          thread_create("main",          main_thread_entry,          NULL,
MAIN_THREAD_STACK_SIZE, MAIN_THREAD_PRIORITY, 20);
    ASSERT(tid != NULL);
#else
    err_t result;
    tid = &main_thread;
    result       =       thread_init(tid,       "main",       main_thread_entry,       NULL,
main_thread_stack, sizeof(main_thread_stack), MAIN_THREAD_PRIORITY, 20);
    ASSERT(result == EOK);
    (void)result;
#endif
    thread_startup(tid);
}
int thread_startup(void){
#ifdef USING_SMP
    hw_spin_lock_init(&_cpus_lock);
#endif
    hw_local_irq_disable();
    hw_board_init();
    show_version();
    system_timer_init();
    system_scheduler_init();
#ifdef USING_SIGNALS
    system_signal_init();
#endif
    application_init();
    system_timer_thread_init();
#endif
    system_scheduler_start();
    return 0;
}
#endif
```

Протокол аналізу звіту подібності експертом

Заявляю, що я ознайомився (-лась) з Повним звітом подібності, який був згенерований Системою виявлення і запобігання плагіату щодо роботи:

Автор: Олеся БОСА

Співавтор:

Назва: Багатозадачна операційна система реального часу для пристроїв IoT

Експерт: Сергій ЛИСЕНКО

Підрозділ: Кафедра комп'ютерної інженерії та інформаційних систем

Коефіцієнт подібності 1:4.19%

Коефіцієнт подібності 2:1.74%

Мікропробіли: 161

Заміна букв: 2

Інтервали: 0

Білі знаки: 0

Дата створення звіту: 2026-05-20 22:24:10.0

Після аналізу Звіту подібності констатую наступне:

☒ **Запозичення, виявлені в роботі є законними і не є плагіатом. Рівень подібності не перевищує допустимої межі. Таким чином робота незалежна і приймається.**

☐ **Запозичення не є плагіатом, але перевищено граничне значення рівня подібностей. Таким чином робота повертається на доопрацювання.**

☐ **Виявлено запозичення і плагіат або навмисні текстові спотворення (маніпуляції), як передбачувані спроби укриття плагіату, які роблять роботу невідповідною вимогам законодавства (Ст. 32. ЗУ Про вищу освіту, пункт 3.1, Ст. 42. ЗУ Про освіту) та вимог НАЗЯВО (Критерій 5), а також кодексу етики і процедурам. Таким чином робота не приймається.**

Обгрунтування:

2026-05-21 Доцент Андрій Нічепорук

Дата експерт

Wed May 20 23:08:36 EEST 2026, Медзатий Дмитро Миколайович, Хмельницький національний університет, ХНУ

Anti-Plagiarism (http://ap.km.ua) v-15.701

Максимальне співпадіння з одним документом 29.0%

Словники перевірки: en_US, ru_RU, ua_UA. **Помилок в документах: 11%**

ID: 271827 Назва: БКР Багатозадачна операційна система реального часу для пристроїв IoT Додано в БД: 2026-05-20 Автора: Олеся БОСА Керівники: Сергій ЛИСЕНКО Консультанти: Опоненти:	Документ		Сумарний збіг по Базі Даних	
	Символи	Лексеми	Символи	Лексеми
	109502	850	33095 (30%)	271 (32%)

Джерело плагіату

ID	Опис	Наявність плагіату в документі	
		Символи	Лексеми
269573	Назва: Звіт з ПДП Багатозадачна операційна система реального часу для пристроїв IoT Додано в БД: 2026-02-27 Автора: Босої О.А. Керівники: Лисенко С.М Консультанти: Опоненти:	32092 (29.0%)	257 (30.0%)

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ХМЕЛЬНИЦЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

РЕЦЕНЗІЯ НА КВАЛІФІКАЦІЙНУ РОБОТУ

Дипломник: Боса Олеся Андріївна

Тема: Багатозадачна операційна система реального часу для пристроїв IoT

Спеціальність: 123 «Комп'ютерна інженерія»

Обсяг кваліфікаційної роботи:

Кількість листів креслень 3 Кількість сторінок записки 60

1. Короткий зміст роботи та прийнятих рішень: Метою кваліфікаційної роботи є синтез багатоздачної операційної системи реального часу для пристроїв IoT.

2. Висновок про відповідність роботи дипломному завданню: Робота повністю відповідає поставленому завданню.

3. Характеристика виконання кожного розділу, ступінь використання останніх досягнень науки і техніки і передових методів роботи: У першому розділі проаналізовано будову операційних систем для IoT-пристроїв і досліджено багатозадачність, перемикання контексту, пріоритети, роботу потоків та вплив обмежених ресурсів IoT-пристроїв. У другому розділі виконано компонентне проєктування ОС реального часу для IoT. Визначено складові ядра, підсистему керування апаратними ресурсами, файлові операції, мережевий обмін, журналювання та енергозбереження. Окремо розглянуто прикладний інтерфейс, який спрощує взаємодію з різними апаратними платформами. У третьому розділі описано розробку й тестування системного програмного забезпечення. Подано запуск ядра, ініціалізацію апаратного рівня, таймера, планувальника та потоків. Перевірено роботу системного часу, динамічної пам'яті, синхронізації й міжпотокової взаємодії. У практичній частині зібрано тестові пристрої, зокрема підключено ESP32-C3 Super Mini та USB-UART перехідник, після чого успішно прошито програмний образ.

4. Позитивні сторони роботи: висока практична цінність роботи.

5. Негативні сторони роботи: -.

6. Оцінка графічного оформлення та пояснювальної записки роботи: Пояснювальна записка оформлена коректно, згідно чинних стандартів оформлення документації.

7. Відгук про роботу в цілому: робота виконана на високому технічному рівні.

8. Інші зауваження: _____

9. Оцінка дипломної роботи: відмінно (A / 93)

Рецензент (прізвище, ім'я, по батькові, посада, місце роботи) _____

Нещун Віктор Миколайович

канд. техн. наук, доцент, доцент каф. кібербезпеки

"27" 05 2026 р. _____ (підпис)

Зав. кафедри КІІС
д-р. філософії Ользі ПАВЛОВІЙ

Олеся БОСА

ПІБ здобувача вищої освіти

ФІТ, 4 курсу, групи КІ2-22-1

ЗАЯВА

З правилами чинного Положення про систему забезпечення академічної доброчесності у Хмельницькому національному університеті, згідно з яким виявлення академічного плагіату є підставою для відмови в допуску кваліфікаційної роботи до захисту і застосування заходів академічної відповідальності, ознайомлений (а). Про використання спеціалізованих програмних засобів (СПЗ) StrikePlagiarism та Anti-Plagiarism для перевірки кваліфікаційних робіт здобувачів вищої освіти на наявність академічного плагіату оповіщений (а). Надаю університету право на передачу моєї роботи для обробки та збереження в базах даних СПЗ і використання роботи для виявлення академічного плагіату в інших роботах, які перевіряються СПЗ.

Також надаю свою згоду на обробку й збереження університетом моєї роботи в Інституційному репозитарії Хмельницького національного університету.

Робота надається для перевірки в електронному варіанті. Електронна версія моєї роботи збігається (ідентична) з друкованою.

1 травня 2026 року

РІШЕННЯ ЕКСПЕРТНОЇ КОМІСІЇ

КАФЕДРИ КОМП'ЮТЕРНОЇ ІНЖЕНЕРІЇ ТА ІНФОРМАЦІЙНИХ СИСТЕМ ПРО ДОПУСК КВАЛІФІКАЦІЙНОЇ РОБОТИ ДО ЗАХИСТУ

Назва кваліфікаційної роботи Багатозадачна операційна система реального часу для пристроїв IoT
Автор Олеся БОСА
Освітня програма Комп'ютерна інженерія та програмування
Рівень вищої освіти перший (бакалаврський)
Спеціальність 123 Комп'ютерна інженерія
Науковий керівник: д.т.н., професор Сергій ЛИСЕНКО

На основі аналізу кваліфікаційної роботи на дотримання вимог академічної доброчесності (у т.ч. відсутності ознак академічного плагіату) з урахуванням результатів перевірки роботи спеціалізованим програмним засобом(ами) комісія зробила такий висновок:

№	Висновок	Позначка про відповідність
1	Ознаки академічного плагіату	
1.1	Запозичення, виявлені в роботі, є законними і не є академічним плагіатом (далі – зазначаються підстави віднесення запозичень до правомірних, якщо потрібно). Робота приймається до захисту.	відповідає
1.2	Виявлені запозичення не є академічним плагіатом, розміщені в розділах, які не описують безпосередньо авторське дослідження, але кількість цитат перевищує обсяг, виправданий поставленою метою роботи (далі – зазначаються детальні та аргументовані підстави віднесення запозичень до правомірних). Робота приймається до захисту, але має бути відкоригована.	
1.3	Виявлені запозичення не є академічним плагіатом, але частково розміщені в розділах, які описують безпосередньо авторське дослідження, а кількість цитат перевищує обсяг, виправданий поставленою метою роботи. Робота може бути допущена до захисту після того як буде відкоригована та доопрацьована і успішно пройде повторну перевірку на академічний плагіат.	
1.4	Робота містить навмисні текстові спотворення, передбачувані спроби укриття текстових запозичень або інші прояви академічного плагіату. Робота містить фабрикацію або фальсифікацію даних. Робота не допускається до захисту.	
2	Інші види порушень академічної доброчесності	

Підтвердження:
Запозичення, виявлені в роботі, є законними і не є плагіатом, оскільки:

1) усі запозичення фрагментарні, або мають належним чином оформленні посилання;
2) окремі виявлені збіги є загальновживаними фразами або виразами, про що свідчить посилання системи на збіг з джерелами на один фрагмент речення;
3) всі зафіксовані системою ознаки модифікації тексту відносяться до комбінування латинських символів зі україномовними скороченнями індексів в формулах, що не є модифікацією тексту;
4) значна частина знайденого плагіату відноситься до списку використаних джерел;
5) збіг зі звітом з ПДП Багатозадачна операційна система реального часу для пристроїв IoT;

Сумарний обсяг всіх запозичень, визначений системою виявлення збігів/ ідентичності/схожості StrikePlagiarism, складає 4.19%; та системою Anti-Plagiarism складає 29%, що, з урахуванням наведених обґрунтувань, відповідає характеру наукового дослідження і свідчить на користь кваліфікаційної роботи.

01.06.2026

Завідувач кафедри _____ (Підпис) Ольга ПАВЛОВА (Ім'я, ПРІЗВИЩЕ)

Гарант освітньої програми _____ (Підпис) Андрій НІЧЕПОРУК (Ім'я, ПРІЗВИЩЕ)

Керівник кваліфікаційної роботи _____ (Підпис) Сергій ЛИСЕНКО (Ім'я, ПРІЗВИЩЕ)