Міністерство освіти і науки України
Харківський національний університет радіоелектроніки

Факультет Комп'ютерної інженерії та управління
(повна назва)

Кафедра Автоматизації проектування обчислювальної техніки
(повна назва)

КВАЛІФІКАЦІЙНА РОБОТА
Пояснювальна записка

рівень вищої освіти другий (магістерський)

Модель використання сопрограм при програмуванні вбудованих систем
(тема)

Виконав:
студент _2_ курсу, групи *СКСм-20-1*
Корнієнко В.Р.
(прізвище, ініціали)

Спеціальність
123 Комп'ютерна інженерія
(код і повна назва спеціальності)
Тип програми освітньо-професійна
(освітньо-професійна або освітньо-наукова)
Освітня програма Спеціалізовані комп'ютерні системи
(повна назва освітньої програми)

Керівник доц. Філіппенко І.В.
(посада, прізвище, ініціали)

Допускається до захисту
Зав. кафедри АПОТ _____ Чумаченко С.В.
(підпис) (прізвище, ініціали)

2021 р.

Харківський національний університет радіоелектроніки

Факультет Комп'ютерної інженерії та управління

Кафедра Автоматизації проектування обчислювальної техніки

Рівень вищої освіти другий (магістерський)

Спеціальність 123 Комп'ютерна інженерія
(код і повна назва)

Тип програми освітньо-професійна
(освітньо-професійна або освітньо-наукова)

Освітня програма Спеціалізовані комп'ютерні системи
(повна назва)

ЗАТВЕРДЖУЮ:

Зав. кафедри _____
(підпис)

«____»_____ 20 ___ р.

ЗАВДАННЯ

НА КВАЛІФІКАЦІЙНУ РОБОТУ

студентові_____
(прізвище, ім'я, по батькові)

1. Тема роботи Модель використання сопрограм при програмуванні вбудованих систем

затверджена наказом університету від «_04_» __11__2021 р. № 1635Ст_

2. Термін подання студентом роботи до екзаменаційної комісії ____ _____ 20__ р.

3. Вихідні дані до роботи _____

4. Перелік питань, що потрібно опрацювати в роботі _____

Аналіз предметної області та постановка задачі.

Розробка моделі

Програмна реалізація тестових прикладів

Аналіз отриманих даних.

5. Перелік графічного матеріалу із зазначенням креслеників, схем, плакатів, комп'ютерних ілюстрацій (п.5 включається до завдання за рішенням випускової кафедри) _____
_____19 слайдів_____
_____
_____
_____
_____
_____
_____
_____
_____

КАЛЕНДАРНИЙ ПЛАН

№	Назва етапів роботи	Терміни виконання етапів роботи	Примітка
1	Отримання завдання	23.09.2020-02.10.2020	
2	Аналіз літератури по темі	02.10.2020-22.10.2020-	
3	Розробка моделі	22.10.2020-01.02.2021	
4	Реалізація моделі	01.02.2021-04.06.2021	
5	Написання тестового програмного забезпечення	04.06.2021-11.08.2021	
6	Оформлення пояснювальної записки	11.08.2021-10.12.2021	

Дата видачі завдання __23_ __09_____2020 р.

Студент _____
(підпис)

Керівник роботи _____ _____
(підпис) (посада, прізвище, ініціали)

РЕФЕРАТ

Пояснювальна записка містить 51 сторінок, 7 рисунків, 3 таблиці, 3 додатки, 29 джерел за переліком посилань.

СОПРОГРАМИ, ДРУКОВАНА ПЛАТА,C++20, ІНТЕРФЕЙС ПЕРЕДАЧІ ДАНИХ, МІКРОКОНТРОЛЕРИ, ВБУДОВАНІ СИСТЕМИ

Задачею дослідження є створення моделі використання сопрограм на вбудованих системах з метою поліпшення швидкісних характеристик та зменшення споживання ресурсів FLASH та ROM пам’яті.

Метою дослідження є створення моделі, написання прикладів використання та порівняння результатів на схожих задачах що були вирішені за допомогою ОСРЧ.

Результатом дослідження є розроблена модель використання сопрограм для роботи на вбудованих системах з можливістю подальшого розвитку у рамках додання користувацьких алокаторів та роботи з сторонніми бібліотеками для вбудованих систем.

ABSTRACT

The project contains 51 pages, 7 figures, 3 tables, 3 annexes, 29 sources according to the list of links.

COROUTINE, PCB, C++20, COMMUNICATION INTERFACE, MCUs, EMBEDDED SYSTEMS

The project is related to creation the coroutines usage model for embedded systems programming along with the reducing of FLASH and ROM memory consumption.

The goal of the project is the creation of coroutines model usage with the corresponding programming examples. Also, the comparison with the similar tasks which were based on RTOS should be completed.

The result of the project is the developed model of coroutines usage for embedded systems development with the future possibility of extending it by user-provided allocators and 3rd party libraries integration.

СОДЕРЖАНИЕ

ВСТУП

На сьогоднішній день контролери використовують повсюдно для реалізації завдань управління в різних пристроях. Вони призначені для управління різними електронними пристроями та здійснення взаємодії між ними відповідно до означеної в мікроконтролері програмі. Кількість задач, що мають бути виконані паралельно, постійно збільшується. Програми управління зростають як у складності написання, так і їх налагодження. З зростанням числа завдань виникає необхідність використання операційної системи для організації управління задачами та загальної координації роботи вбудованого пристрою. Однак, це накладає свої обмеження на пам'ять і знижує продуктивність системи в цілому.

Будь-яка ОСРВ вимагає принаймні апаратних засобів для роботи основних механізмів з управлінням пам’яттю, обробкою черг повідомлень і перемиканням контексту потоків. Таким чином, будь-яка мінімальна конфігурація ОСРВ займає RAM та FLASH, навіть якщо запущено лише планувальник. Типове програмне забезпечення на основі ОСРВ містить цикл планувальника завдань із створеними чергами повідомлень. Вбудована програма зазвичай включає в себе кілька завдань ОС для взаємодії з мережею, керування зовнішніми периферійними пристроями і, у разі наявності користувальницького інтерфейсу, потік UI для підготовки завдань, пов’язаних з візуалізацією. Крім того, іноді ОСРВ може використовуватися з комбінацією деяких комунікаційних стеків, таких як MQTT або LWIP. Усі ОСРВ мають власні вимоги до цільового обладнання n, і зазвичай внутрішні механізми суворо засновані на цільовій платформі. Іноді використання ОСРВ може бути джерелом надмірної інженерії для вбудованого програмного забезпечення через обмежені доступні ресурси на цільовій платформі.

Таким чином, використання сопрограм для реалізації кооперативної багатозадачності дає можливість використання легких моделей переключення контексту між завданнями.

1 АНАЛІЗ ПРДМЕТНОЇ ОБЛАСТІ ТА ПОСТАНОВКА ЗАДАЧІ

1.1 Загальні відомості про проектування архітектури вбудованої системи

Розглянемо побудову архітектури вбудованого програмного забезпечення (ПЗ) з використанням компонентів, що мають асинхронну складову.

В [1,2] наведено побудову архітектури з точки з точки зору розподілу компонентів периферійного пристрою, драйвера і компонента як самостійної архітектури. Детально розглядається один з варіантів реалізації взаємодії на підставі моделі сигналів-слотів, що є різновидом архітектурного патерну проектування "Спостерігач" з розглядом проблеми збереження контексту. Пропонується використовувати набір методів у компонента, які мають сигнатури виду onEventCompleted з можливістю трансляції події в головний контекст програми, в разі виникнення події в іншому потоці виконання.

В [3] продемонстровано підхід побудови архітектури програмного забезпечення для вбудованих систем, де передбачається використання event-driven підходу, в якому основним елементом є подія, що виникає на певному рівні [EventDriven].

При такому рішенні при проектуванні вбудованого програмного забезпечення необхідним є формування машини станів, в яких може перебувати система і правил переходу між цими станами. При додаванні асинхронної складової, таких як периферійні компоненти, додаються проміжні стани, пов'язані з очікуванням події і збереження безпосереднього контексту операції.

В [2] представлені варіанти і сценарії використання мови C ++ для створення ПЗ як для мікроконтролерів з обмеженими ресурсами, а саме на базі архітектури AVR, так і процесорів на базі архітектури ARM-Cortex з

використанням самостійно реалізованих примітивів на базі функціоналу C ++ 14. В роботі показано, що поряд з традиційним використанням мови C для програмування вбудованих систем, мова C ++ дає ряд переваг, таких як необхідний рівень абстракції від апаратної частини, зберігаючи сумісність з мовою C.

В [2, 3] розглядаються наступні особливості побудови ПЗ: побудова основних компонентів і використання примітивів ОС, а саме черг, механізмів синхронізації ОС, реалізації перемикання контексту, виділення і менеджменту пам'яті в системах з обмеженими ресурсами. В [3, 4] зазначається, що використання поширених ОСРЧ для вбудованих платформ накладає певні обмеження на обсяги пам'яті і часу на перемикання контексту в межах завдання.

Також механізми, використовувані в RTOS бувають від платформи залежними, що викликає певний ряд проблем при портуванні програм між різними процесорами від масових вендорів і виникає необхідність переписування частини коду.

При використанні RTOS, актуальним залишається збереження архітектури ПО без явного надлишкового ускладнення через використання примітивів конкретної ОС.

У разі RTOS також необхідна передача управління планувальником завдань (в разі використання кооперативної багатозадачності) і відновлення контексту виконання завдання.

Існує існуючий відомий механізм для реалізації поведінки, подібної до сопрограм, у рідному C-API. Він називається protothreads і постачається як стороння бібліотека. Основні недоліки протопотоків пов'язані з їх обмеженнями з продовженням життя змінних області видимості. Крім того, через нестандартність він має деякі обмеження щодо використання в класах C++. Основний недолік у забрудненні вихідного коду протопотоками макровизначеннями [22].

Публікації [4,16,23] описують основні причини, які є базовими для додавання сопрограм як концепції в стандарт мови програмування C ++ 20. З основних розглянутих причин згадані особливості в написанні асинхронного коду з використанням примітивів взаємодії, доданих в стандарті C ++ 11 і існуючих до C ++ 17 включно.

Розглянуто основні проблеми використання future і promise в разі необхідності додавання обробки помилок і реалізації взаємодії producer-consumer з асинхронної складовою. Основною проблемою, яка розглядається, є питання передачі і збереження контексту між різними елементами асинхронного конвеєра, які в разі використання існуючих механізмів не дозволяють реалізовувати ефективний асинхронний код без надмірного опису перехідних точок, де здійснюється передача контексту управління між завданнями.

Сопрограми в C ++ 20 реалізовані на рівні компілятора. Створення Сопрограми відбувається при додаванні ключового слова co_await або co_yield. На момент 2019го року, згідно з даними з пропозиції Гора Нішанова в стандарт C ++ 20, програмне забезпечення, побудоване з використанням сопрограм, було продано / реалізовано більш ніж на 400 000 пристроях [4].

Як приклад в [16] приведена реалізація інтеграції з бібліотекою мережевої взаємодії boost::asio для забезпечення неблокуючим читання буфера з сокета. Також, приділено увагу недолікам і правкам, які повинні бути додані в існуючу пропозицію в стандарт, а саме про використання сопрограм і менеджмент пам'яті та обробку виняткових ситуацій в системах, де генерація і використання винятків є неможливою.

Основні базові приклади використання сопрограми з початкових моментів розглядаються детально в [5.6]. З основних аспектів приділено увагу точкам призупинення сопрограми, нововведених операторів в мову програмування і виразах, які додатково генеруються компілятором при використанні одного з ключових слів. Як приклади наведено реалізацію інтеграції з системно-залежними API Windows / POSIX. Наведено реалізацію

joinable thread для перемикання в новий контекст виконання з використанням оператора co_await і підтримкою pthread_join для коректного завершення створеного потоку після виконання операції.

У разі використання механізмів, доступних в стандарті C ++ 20 для створення сопрограм необхідний один з компіляторів, що підтримують даний функціонал. Серед них GCC 10.2, MSVC 16.8. В ході розробки були виявлені незначні помилки в реалізації компіляторів, які виправлені в останніх ревізіях, а саме особливість роботи co_await оператора в constexpr-контексті [8].

Сопрограми підтримуються в Clang (частково), MSVC19.28 (повна підтримка), GCC 10.2 (повна підтримка), Apple Clang (повна підтримка) [16].

1.2 Різновиди сопрограм

Розрізняють два типи сопрограм, які можуть бути реалізовані в системі. Stackfull мають принципову відмінність від stackless в можливостях призупинення сопрограми. Stackfull сопрограми можуть бути призупинені в будь-якому місці виклику, відновлення контексту відбувається з місця, де програма була припинена. У свою чергу, stackless сопрограми можуть бути призупинені тільки з top-level function [9].

Також, є різновиди в знанні про викликаючу сторону. Існує два види - асиметричні і симетричні сопрограми. Асиметричні сопрограми мають дві операції для передачі управління, де відбувається передача управління призупиненої сопрограми і операція для припинення виконання, яка повертає контроль стороні, звідки було здійснено виклик. Симетричні сопрограми забезпечують єдиний механізм передачі управління між один одним, будучи рівноправними. В цьому випадку необхідно явно вказувати сопрограму, в яку буде передано управління [10]. Реалізація сопрограм в C ++ 20 являє собою compile-internal asymmetric stackless coroutines.

Аспекти тестування асинхронного коду в багатопотоковому середовищі висвітлені в [17,18]. З них є можливим вияснити, як саме може

виглядати реалізація середовища модульного тестування для програми з використанням набору заглушок для спрощення поведінки компонентів.

Приклад практичного використання сопрограм для розробки додатків наведено в [19], де реалізовано величезний набір сопрограмних примітивів з апаратною взаємодією для спеціальної ОС «Манагарм».

Проведений аналіз показав, що використання ОСРЧ накладає свої обмеження у вигляді вимог до оперативної пам’яті і часових обмежень. Існуючі реалізації сопрограм у вигляді сторонніх компонентів не є крос-платформеними рішеннями. Розробка моделі кроссплатформенного застосування співпрограми в при програмуванні вбудованих систем для реалізації легкої за споживанням ресурсів кооперативної багатозадачності є актуальним завданням.

Таблиця 1.1 Порівняльна таблиця технологій, що розглядаються

Технологія	Чи є кросс-платформенною	Стандартність механізму	Низькі накладні втрати	Обмеження
Free RTOS	+	-	-	Є важкою дял простих додатків
Protothreads	+	-	+	Обмеження з життевим циклом змінних. Обмеження використання оператору switch
Coroutines C++20	+	+	+	Не має обмежень. Може бути використано у режимі bare-metal.

1.3 Мета дослідження

Метою дослідження є розробка моделі використання сопрограми для реалізації кооперативної багатозадачності на системах з обмеженими ресурсами з використанням стандарту З С++ 20 для мінімізації використовуваних ресурсів пам'яті, тимчасових втрат і забезпечення продуктивності системи в цілому. Основною метою архітектури, яка базується на взаємодії сопрограм, є спрощення написання асинхронного коду без накладних витрат для системного фону.

Для цього необхідно вирішити наступні завдання.

1. Провести аналіз архітектури сучасних ОСРЧ для виявлення ресурсозатрат в мікроконтролері.
2. Проаналізувати залежність необхідного обсягу виділеної пам'яті контролера і споживання процесорного часу в залежності кількості сопрограм.
3. Розробити модель вдосконаленого мінімального кооперативного планувальника без пріоритетного механізму.
4. Розробити модель використання сопрограм на системах з обмеженими ресурсами.
5. Провести тестування і аналіз програмно реалізованих компонентів на контролері.

2 РОЗРОБКА МОДЕЛІ ВИКОРИСТАННЯ СОПРОГРАМ

2.1 Огляд існуючих задач в області проектування вбудованих рішень

Для розгляду API конкретної системи та особливостей її роботи було обрано ОСРВ з реалізацією на мікроконтролерах архітектури ARM.

Однією з вимог до ОС MCU є реалізація механізмів багатозадачності. Типовий варіант розподілу завдань для даної ОС показаний на рисунку 2.1.

У м’яких системах реального часу можна використовувати кооперативні механізми багатозадачності, щоб мінімізувати накладні витрати. Кооперативна багатозадачність передбачає «добровільну» передачу управління від однієї задачі до іншої шляхом виклику системної функції. Для спільної багатозадачності передбачена гібридна схема роботи, тобто добровільна передача керування завданнями з однаковим пріоритетом і правильне випередження при запуску завдань з більшим пріоритетом.

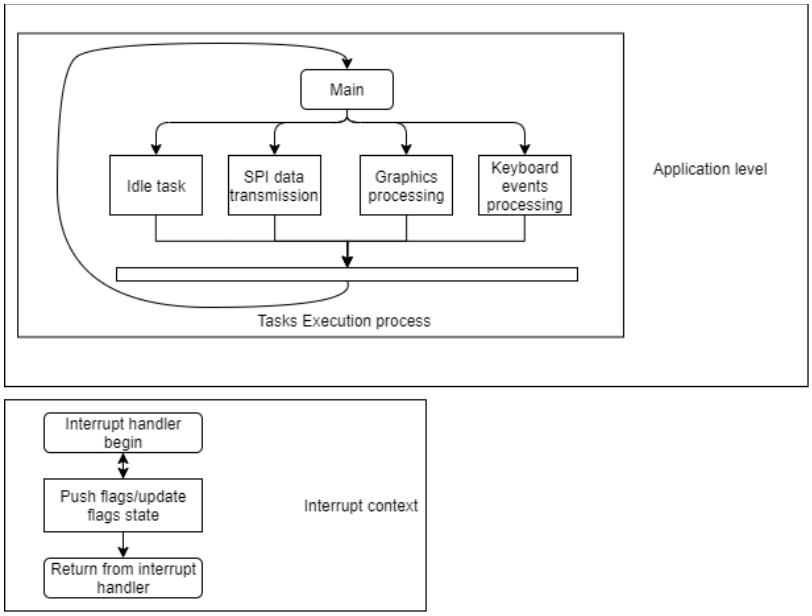


Рисунок 2.1 – Типовий варіант розподілу завдань для заданої ОС

Кооперативний режим багатозадачності позбавляє можливості виконувати завдання відповідно до виділеної кількості часу.

Як приклад кооперативних багатозадачних завдань було взято завдання передачі потоку даних за допомогою комунікаційних інтерфейсів за допомогою DMA.

Приклад DMA був наданий для показу проблеми збереження контексту виконання. Зазвичай результат використання DMA не можна отримати відразу. Для типових платформ розмір буфера DMA обмежений фіксованим значенням, тоді як обсяги передачі даних значно перевищують його. Іноді зв'язок DMA може бути строго пов'язана з периферійними маніпуляціями.

Розглянемо роботу з механізмом транзакції в рамках завдання RTOS під час виконання цих завдань.

Типовий варіант використання включає попереднє формування пулу транзакцій і додавання його до черги надсилання, а потім послідовне отримання транзакцій і їх надсилання через певний інтерфейс даних. Рис. 2.2 ілюструє транзакційний підхід на ОСРВ.

У разі такого підходу необхідно попередньо виділити пам'ять для черги транзакцій. Процес роботи з чергою включає роботу над контекстом переривання з вилученням наступної транзакції. Це спричиняє втрату часу та збільшує накладні витрати на формування черги.

У роботі представлена модель використання сопрограм у завданнях, пов’язаних з передачею даних з поєднанням полегшеної багатозадачності. Основні аспекти охоплюють як оптимальне використання ресурсів процесора, так і лінеаризацію коду мікропрограми.

У модель входять такі деталі:

планувальник сопрограм;

дескриптори завдань, представлені у вигляді призупинених сопрограм.

Планувальник завдань відповідає за керування станами завдань і відновлення виконання призупинених сопрограм у головному циклі виконання. Дескриптор завдання являє собою єдине системне завдання з можливістю призупинення без будь-яких блоків для отримання результату асинхронної операції. У системах з обмеженою пам’яттю його можна

реалізувати як чергу, де розташовані завдання, які очікують на запуск для виконання.

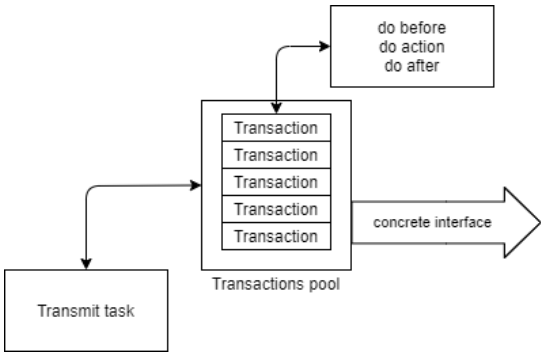


Рисунок 2.2 – Транзакційний підхід на ОСРВ.

Планувальник повиннен відповідати за взаємодію між усіма внутрішніми та зовнішніми компонентами, визначення порядку виконання завдання, обробку переривань.

Після активації системи планувальник повинен відповідати за запуск системи.

Кожна задача представлена як спрограмма, яка може зв’язуватися з доступними спрограмами, викликати їх, призупиняти себе або продовжувати виконання. Робочий потік сопрограм представлено на рис. 2.3.

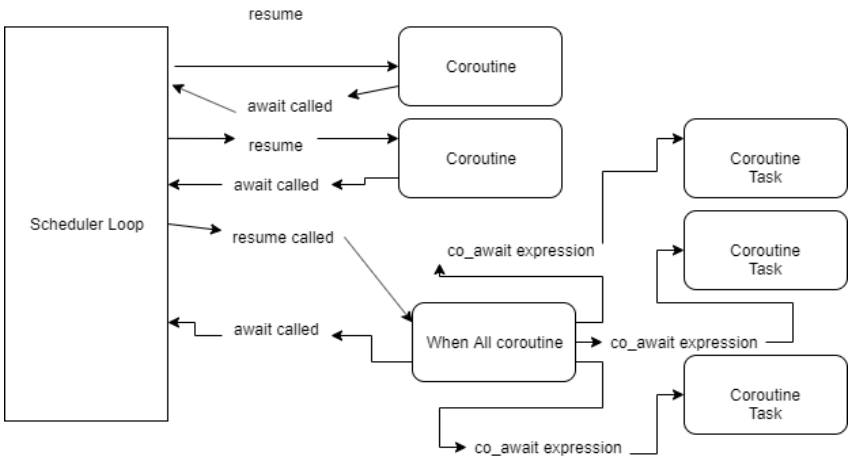


Рисунок 2.3 – Модель взаємодії сопрограм

У разі виконання операції блокування, сопрограму можна призупинити із збереженням контексту виконання, тоді дескриптор сопрограми передається до блокуючого виклику. Після змагання блокування виклику дескриптор сопрограми передається планувальнику, який відновлює стан сопрограми після досягнення конкретного кроку виконання. Також у моделі використовується метод диспетчеризації FIFO. Підхід FIFO був обраний через простоту для невеликої системи. Для прошивки з вимогами програмного забезпечення реального часу можна використовувати досить спрощені планувальники. Основними перевагами є використання ресурсів і спрощення тестування. Усі точки для зберігання стану сопрограми контролюються компілятором з відповідними точками налаштування. Основним механізмом виділення та звільнення фрейму сопрограми можна керувати шляхом перевантаження для нього відповідного оператора new та delete. Крім того, з увімкненою оптимізацією конкретний компілятор може оптимізувати розподіл кадрів сопрограми та перемістити сховище стану на сторону абонента без використання динамічної пам’яті.

Виконується перше призупинене завдання, потім у порядку черги відновлюються решта доступних завдань. Призупинення відбувається, коли визначений користувачем тип Awaitable повертає хибне значення з методу await_ready. Для випадку помилкового повернення генерується додатковий код для призупинення функції із збереженням значень локальної змінної та поверненням до викликаючої сторони.

Виконання завдання передачі даних з використанням асинхронного сповіщення про змагання містить декілька викликів ініціалізації сопрограми, виконання операції await_suspend з передачею збереженого стану сопрограми обробнику. Наступний крок включає передачу дескриптора coroutine_handle до основного циклу планувальника для відновлення стану сопрограми. Рис.4 коротко ілюструє набір станів життєвого циклу сопрограми у разі використання її для передачі даних через інтерфейс зв'язку SPI.

Надану модель сопрограм можна використовувати як у завданнях, пов’язаних зі зв’язком, так і в загальному програмуванні мікропрограми вбудованих пристроїв. Основні деталі, пов’язані з виділенням фрейму сопрограми, розміщуються в частині компілятора. Рисунок 2.4 пов'язаний з розподілом завдань у системі з описом їх життєвого циклу в середовищі RTOS і сопрограм.

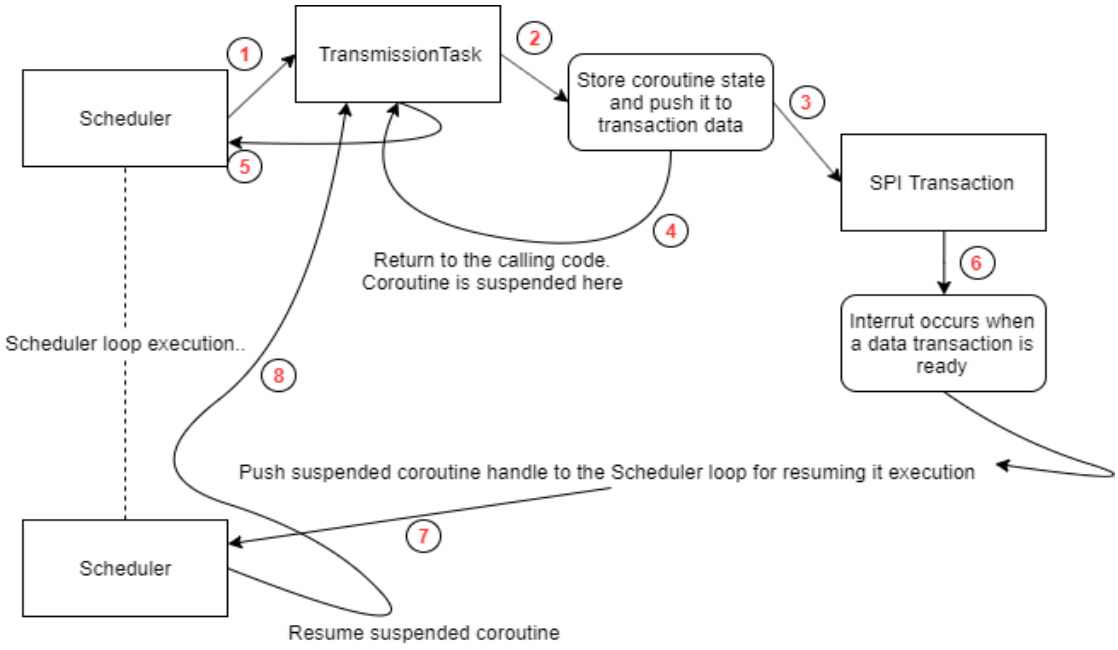


Рисунок 2.4 – Стани сопрограми для завдання передачі даних

Стани сопрограми для завдання передачі даних наведені на рисунку 2.4, де 1 – початок передачі даних; 2 – формування запиту на шину SPI з передачею стану сопрограми; 3 – формування запиту DMA; 4 – повертання на викликаючу сторон; 5 – повернення керування планувальнику, на цьому етапі сопрограма призупинена, планувальник може виконувати інші завдання; 7 – після закінчення передачі даних coroutine_handle додається до черги планувальника; 8 – планувальник виконує відновлення завдання передачі даних з останньої точки призупинення, на цьому етапі відновлюються локальні змінні та стан завдання.

На підставі запропонованої моделі були розроблені такі програмні модулі – кооперативний планувальник сопрограм, дескриптори задач для передачі даних і заготовки для Awaitable-типів. Також, виконано розробку та автоматизоване тестування драйверу SPI шини з підтримкою coroutines та драйверу для мікросхеми FLASH-пам’яті з підтримкою сопрограм.

Для реалізації мінімальної співпрограми необхідно визначити Awaitable-тип, в якому визначити promise_type який буде повернений на викликаючу сторону і визначить поведінку співпрограми при першому призупиненні, при фінальному зупиненні, тип повертаємого результату або ж return_void при його відсутності а також поведінку під час перехоплення необробленого виключення [5,6,7]. Орієнтовна реалізація Resumable типу наведена в лістингу 1. Корисна задача-вивід повідомлень для налагодження з можливістю призупинення та відновлення.

Лістинг 2.1 – Примітивна реалізація awaitable-типу для підтримки оператора co_await

```
#include <string>
#include <coroutine>

#include "SEGGER_RTT.h"

class Resumable
{
public:
    struct promise_type;
    using coro_handle = std::coroutine_handle<promise_type>;

    Resumable(coro_handle coroutineHandle) noexcept
        : m_handle{coroutineHandle}
    {
    }
    ~Resumable() { m_handle.destroy(); }

public:
    void resume(){ if (!m_handle.done()) m_handle.resume();}
    struct promise_type
    {
        auto get_return_object() noexcept { return coro_hand
le::from_promise(*this);}
        auto initial_suspend() noexcept { return std::suspen
d_always{};}
```

```
            auto final_suspend() noexcept{ return std::suspend_alw
ays{};}
            void return_void() noexcept {}
            void unhandled_exception() noexcept{}
        };

    private:
        coro_handle m_handle;
    };

    constexpr inline auto FirstMessage = "Before suspending";
    constexpr inline auto SecondMessage = "After restoring from
suspended state";

    Resumable suspendableFunction() noexcept
    {
        SEGGER_RTT_WriteString(0, FirstMessage);
        co_await std::suspend_always{};
        SEGGER_RTT_WriteString(0, SecondMessage);
    }

    int main(void)
    {
        auto resumable = suspendableFunction();
        resumable.resume();
        resumable.resume();

        while (true)
        {
        }
        return 0;
    }
```

Для мінімальної конфігурації FreeRTOS на цільову платформу необхідно описати конфігурацію у файлі FreeRTOSCOnfig.h де вибрати використовуваний функціонал, визначити стратегію розподілу пам'яті шляхом підстановки одного з файлів heap_n.c, де n- один з підходів менеджменту. Так-же, необхідно додати платформо-залежні модулі для реалізації xPortSysTickHandler, vPortSetupTimerInterrupt, vPortSupressTicksAndSpleep [12,13]. В якості мінімального прикладу був реалізований аналогічний функціонал, що наведено у лістингу 2.2.

Лістинг 2.2 – Мінімальний приклад роботи з FreeRTOS

```
#include <FreeRTOS.h>
#include <task.h>
```

```
#include <timers.h>

#include "SEGGER_RTT.h"

static void loggerTaskFunction (void * pvParameter)
{
    UNUSED_PARAMETER(pvParameter);
    while (true)
    {
        SEGGER_RTT_WriteString(0, FirstMessage);
        taskYIELD();
        SEGGER_RTT_WriteString(0, SecondMessage);
    }
}
int main(void)
{

    constexpr std::uint8_t Priority = 2;
    UNUSED_PARAMETER(
        xTaskCreate(
            loggerTaskFunction,
            "LogggerTask",
            configMINIMAL_STACK_SIZE + 200,nullptr, Priority
, &seggerLoggerTaskHandle)
    );
    vTaskStartScheduler();

    while (true)
    {
    }

    return 0;
}
```

При розгляді підходів до передачі даних традиційно виступає основним транзакційний варіант з формуванням черги транзакцій на шину. Для розгляду транзакційного підходу необхідно реалізувати мінімальну одиницю-транзакцію і реалізувати чергу виконання транзакцій. Транзакційна черга може бути реалізована на базі etl::queue з фіксованим розміром і примітиву транзакції, який може бути реалізований як std::variant для зберігання типу даних, закладених в транзакцію + дій, які повинні бути виконані перед і після транзакції. Приклад наведено у лістингу 2.3.

Лістинг 2.3 – Приклад реалізації елемента транзакції

```
struct TransactionDescriptor
```

```
    {
        std::function<void()> beforeTransaction;
        std::function<void()> afterTransaction;

        struct DataSequence
        {
            const std::uint8_t* dataBlock;
            const size_t blockSize;
        };

        std::variant<std::uint8_t,DataSequence> transactionData;
    };
```

Таким чином, транзакційний дескриптор може бути використано за допомогою додавання допоміжних функцій, що наведені у лістингу 2.4.

Лістинг 2.4 – Реалізація допоміжних функцій для транзакційного дескриптора

```
    template <typename... Args> void sendCommand(std::uint8_t
command, Args... commandArgs) noexcept
    {
        sendCommand(command);
        sendChunk(static_cast<std::uint8_t>(commandArgs)...);
    }

    template <typename... Args> void sendChunk(Args...
_chunkArgs) noexcept
    {
                                    std::array chunk =
{static_cast<std::uint8_t>(_chunkArgs)...};
        Interface::Spi::Transaction chunkTransaction{};

                  chunkTransaction.beforeTransaction = [this]
{ setDcPin(); };

         chunkTransaction.transactionAction = [this, chunkToSend
= std::move(chunk)] {
            m_pBusPtr->sendChunk(
                                          reinterpret_cast<const
std::uint8_t*>(chunkToSend.data()), chunkToSend.size());
        };

                   chunkTransaction.afterTransaction = [this]
{ resetDcPin(); };

        m_pBusPtr->addTransaction(std::move(chunkTransaction));
    }
```

Головна ідея полягає у можливості заданні трьох основних параметрів-дії перед транзакцією, транзакціїї та дії для завершення транзакції на шині. За рахунок використання variadic-шаблонів є можливим передавати не фіксовану кількість аргументів до функції передачі команди та/або її даних. Приклад виконання передачі даних за рахунок розробленого API наведено у лістингу 2.5

Лістинг 2.5 – Процедура відправки ініціюючих команд з використанням транзакційного підходу

```
sendCommand(DisplayReg::SLPOUT);
sendCommand(DisplayReg::COLMOD, 0x55);
sendCommand(DisplayReg::MADCTL, 0x08);
sendCommand(DisplayReg::CASET, 0x00, 0, 0, 240);
```

При реалізації сопрограммного підходу для планувальника завдань-розглянемо спрощений варіант планувальника, при якому попередньо описуються завдання і відповідні awaitable-типи для них, після чого виконується resume всім доступним для виконання завдань в порядку черги. Зберігання завдань реалізовано в векторі фіксованого розміру, що надає інтерфейс std::vector. З метою оптимального використання ресурсів був узятий контейнер etl::vector (ETL -Embedded Template Library).

Використання сопрограми при реалізації відновлення контексту при передачі даних має два підходи.

Перший з використанням std::tuple для формування послідовності команд зі збереженням їх у std::tuple. Другий варіант буде представлений in-memory таблицею команд ініціалізації + статичними буферами. У разі std::tuple підходу реалізуємо відправку з використанням std::apply + when_all концепції. Для табличного варіанту застосуємо спрощений варіант. Для роботи з відправкою даних з використанням co_await оператора необхідно реалізувати відповідний Awaitable- тип.

У разі роботи з std::tuple-варіантом необхідно використовувати fold expressions (були додані у C ++ 17) з оператором co_await, який буде

викликаний для кожного елемента ланцюжка команд. Варіант з використанням std::apply наведено у лістингу 2.6.

Лістинг 2.6 – Процедура відправки ініціюючих команд

```
    // Implementation of launchInit function for generating
co_await Expression for the each Command Descriptor node
    template<typename Tuple, std::size_t... Indexes>
        CoroUtils::VoidTask launchInit(Tuple&& commandSequence,s
td::integer_sequence<std::size_t,Indexes...> ) noexcept
        {
            (co_await CoroUtils::when_all_sequence(
                        sendCommand(
                                std::get<Indexes>(commandSequenc
e).command.data()
                            ,   std::get<Indexes>(commandSequenc
e).command.size()
                        )
                    ), ...
           );
        }
```

При цьому функція передачі даних буде представлена у вигляді послідовності викликів co_await для відповідних компонентів. Для передачі команди необхідний виклик sendCommand, для блоку даних-sendChunk відповідно до лістингу 2.7.

Лістинг 2.7– Приклад процедури відправки даних

```
// Data transmission
    co_await setAddrWindow(_x,_y,_width,_height);

    static CommandDescriptor<0x2c> RamWrite{};

    co_await sendCommand(RamWrite.command.data(), RamWrite.c
ommand.size() );  //LCD_WriteCMD(GRAMWR);
    co_await sendChunk(reinterpret_cast<const std::uint8_t*>
( _colorToFill ),TransferBufferSize);
```

У варіанті з використанням транзакційності – необхідно виконати формування черги команд / даних для відправки, після чого ініціювати

запуск обробки черги. Для цього необхідно реалізувати методи додавання команди / даних в чергу виконання та реалізувати методи запуску черги.

Приклад реалізації методу відправки команди з N кількістю аргументів які повинні бути відправлені окремо наведено у лістингу 2.8.

Лістинг 2.8 – Приклад процедури відправки даних в разі використання транзакційного підходу

```
        template< typename ... Args >
        void sendCommand(std::uint8_t _command,Args... _commandA
rgs)noexcept
        {
            sendCommand( _command );
            sendChunk( static_cast<std::uint8_t>(_commandArgs)...);
        }

        template< typename ... Args >
        void sendChunk( Args... _chunkArgs )noexcept
        {
            std::array chunk = { static_cast<std::uint8_t>(_chun
kArgs)... };
            Interface::Spi::Transaction chunkTransaction{};

            chunkTransaction.beforeTransaction =[this] {setDcPin
();};
            chunkTransaction.transactionAction = [this, chunkToS
end = std::move(chunk)]{
                    m_pBusPtr->sendChunk(
                            reinterpret_cast<const std::uint8_t*
>( chunkToSend.data() ),
                            chunkToSend.size()
                    );
                };
            chunkTransaction.afterTransaction = [this]
{resetDcPin();};
            m_pBusPtr-
>addTransaction(std::move(chunkTransaction));
        }
```

Для варіанту реалізації з використанням FreeRTOS – можливо використовувати транзакційний підхід плюс чергу повідомлень для створеної завдання, однак, буде порушена лінійність вихідного коду, тому що буде необхідність постійного виклику taskYEILD (); на час виконання неблокуючих операцій, тому що подальші операції є залежними-по-даними

від попередніх виконаних. Також, виникне необхідність реалізації статичного сховища даних в рамках завдання.

2.2 Реалізація Awaitable-типів для роботи з SPI-інтерфейсом

Для роботи з SPI інтерфейсом необхідно реалізувати невеликий набір Awaitable-типів, на яких може бути викликаний оператор co_await. Найпростіший варіант має містити контекст, у якому буде виконано відновлення сопрограми, процедуру обробки GPIO Data/Command, параметри буфера, що буде передано:

Також, необхідно реалізувати процедуру, що буде викликана по завершенню передачі з використанням DMA. У загальному випадку, по завершенню транзакції є можливим декілька шляхів виконання обробника. Першим є перевірка, що транзакції розміром з буфер DMA є можливими(в залежності від моделі процесору, DMA транзакції можуть бути обмежені за розміром). Іншим можливим варіантом є ситуація, коли залишись данні, розмір яких є меншим за DMA буфер. Третім варіантом є завершення передачі, при якому необхідно перевірити контекст, у якому може бути виконано відновлення сопрограми. Через те, що обробник завершення передачі працює у контексті переривання, можливо або виконати безпосередньо відновлення виконання сопрограми або відкласти її виконання до глобальної черги сопрограм, таким чином у наступному циклі виконання головного потоку буде її відновлено. У скороченому вигляді обробник завершення передачі даних наведено у лістингу 2.9.

Лістинг 2.9 – Скорочена реалізація обробника завершення передачі даних

```
    void transmitCompleted() noexcept
    {
                const   bool   isAllDmaTransactionsProceeded   =
m_transmitContext.fullDmaTransactionsCount == 0;
         const bool isAllChunckedTransactionsCompleted =
```

```
            m_transmitContext.chunkedTransactionBufSize == 0;

        if (!isAllDmaTransactionsProceeded)
        {
              //Якщко є можливим передача даних розміром з DMA
буфер
        }
        else if (!isAllChunckedTransactionsCompleted)
        {
            // Якщо є залишок з непереданих байтів
        }
        else
        {
             // У випадку, коли всі блоки були передані- перевіряємо,
у якому контексті необхідно відновити сопрограму.
              // Цей метод виконується у контексті переривання, таким
чином відновлення контексту важкої сопрограми може призвести до
«провалу» стеку. На цьому моменті є можливим перенести виконання
сопрограми або до головного потоку програми або виконати її
відновлення у контексті переривання . Черга сопрограм буде оброблена у
головному потоку програми.

            if (m_transmitContext.restoreInSpiCtx)
            {
                m_coroHandle.resume();
            }
            else
            {
                CoroUtils::CoroQueueMainLoop::GetInstance().push
ToLater(m_coroHandle);
            }
        }
    }
```

Як приклад, розглянемо використання подібного компоненту для ініціалізації та роботи з дисплейним модулем. Для кожної команди необхідно виконати передачу команди та передачу її аргументів, якщо вони є і виконати очікування N мілісекунд (після команди DISP_ON, як приклад).

У випадку використання розроблених сопрограмних примітивів, нам необхідно реалізувати лише функцію запису буферу та функцію читання даних з масиву команд ініціалізації. Лістинг функції, що виконує ініціалізацію дисплею, наведено у лістингу 2.10

Лістинг 2.10 – Реалізація функції ініціалізації дисплею

```
    void initDisplay() noexcept
    {
        TBaseSpiDisplay::resetResetPin();
        Delay::waitFor(100);
        TBaseSpiDisplay::setResetPin();
                                size_t       CommandCount       =
InitializationCommands::CommandsTransactionsCount;
                          const     std::uint8_t*     pBuffer     =
InitializationCommands::Commands.data();
        while (CommandCount--)
        {
            const std::uint8_t* Command = pBuffer++;
            const std::uint8_t NumArgs = *pBuffer++;
            const std::uint8_t Delay = *pBuffer++;

            co_await TBaseSpiDisplay::sendCommandImpl(Command);
            if (NumArgs)
            {
                  co_await TBaseSpiDisplay::sendChunk(pBuffer++,
NumArgs);
                pBuffer += (NumArgs - 1);
            }
            if (Delay)
                Delay::waitFor(Delay);
        }
        TBaseSpiDisplay::m_displayInitialized.set();
    }
```

Кожен виклик sendCommandImpl виконується ассихронно. При виконанні co_await TBaseSpiDisplay :: sendCommandImpl або co_await TBaseSpiDisplay::sendChunk - виконується припинення функції initDisplay, при цьому управління повертається викликаючій стороні. Після закінчення процедури виконується установка події про закінчення ініціалізації дисплею.

Для інтеграції з графічною бібліотекою типовим рішенням є реалізація методу відправки фреймбуферу до дисплею. Іноді, є можливим використовувати кадровий буфер не повного розміру, таким чином зменшуючи кількість необхідної RAM у проекті. Інтеграція може бути виконана шляхом реалізації методу заповнення екранної області заданого розміру. У нашому випадку є необхідність реалізувати асинхронну взаємодію компонентів. Лістинг 2.11 демонструє фрагмент реалізації метода заповнення дисплею у синхронному виконанні.

Лістинг 2.11 – Реалізація функції заповнення екранної області дисплею

```
//Формуємо екранну область для оновлення
        setAddrWindow(x, y, width, height);

        static std::uint8_t RamWriteCmd{0x2C};
        // Команда запису до екранного буферу
        TBaseSpiDisplay::sendCommandImplFast(&RamWriteCmd);

            //Встановлення порту Data/Command для переводу
дисплею у режим роботи з даними (все що буде передано у дисплей
        //буде прийнято як данні, а не як команди керування.
        TBaseSpiDisplay::setDcPin();
        // Відправляємо дисплейний буфер
        TBaseSpiDisplay::sendChunk(
                                        reinterpret_cast<const
std::uint8_t*>(colorToFill), TransferBufferSize);
        // Повертаємо порт Data/Command у початковий стан
        TBaseSpiDisplay::resetDcPin();
        // Сигналізуємо про завершення заповнення області
        TBaseSpiDisplay::onRectArreaFilled.emitLater();
```

За допомогою використання сопрограм та їх безшовної інтеграції до системи є можливим додати у декілька точок ключове слово co_await та змінити повертаємий типи у функцій. Подальша генерація коду, що буде виконувати роботу зі збереження контексту, буде виконана компілятором. Питання що до роботи без динамічної пам’яті можуть бути частково вирішені при встановлення життєвого циклу сопрограми, тобто її стек може бути збережено на викликаючій стороні.

2.3 Розробка допоміжних примітивів

Для вирішення задач з використанням сопрограм є необхідним реалізувати набір примітивів, що може бути використаний у якості повертаємого типу, для коректної взаємодії з операторами co_await/co_yield/co_return. Однією з задач є повертання до коретного контексту виконання програми під час асинхронної взаємодії у випадку наявності вкладених викликів сопрограми. У такому випадку є необхідність у створенні типу, що може бути повернено з функції та використано у якості базового примітиву задачі у системі. Одним з моментів, якому треба

приділити увагу, є відновлення коректної сопрограми. У Лістингу 2.12 наведено фрагмент реалізації типу VoidTask у якості lazy-computation примітиву, що може бути створено та призупинено до початку виконання, а саме до використання оператора co_await або co_yield. Наведена реалізація демонструє як саме може бути використано механізм передачі керування symmetric transfer для повертання корректного дескриптора сопрограми з фунції.

Лістинг 2.12 – Фрагмент реалізації типу VoidTask для демонстрації механізму symmertic transfer.

```
     struct task_awaitable
         {
               task_awaitable(std::coroutine_handle<promise_type>
coroutine) : m_coroutine{coroutine}
              {
              }
              bool await_ready() const noexcept
              {
                  return !m_coroutine || m_coroutine.done();
              }
                       auto  await_suspend(std::coroutine_handle<>
awaitingRoutine)
              {
                  // Використовуємо symmetric-transfer
                  // Більш детальне пояснення:
                  // https://lewissbaker.github.io/2020/05/11/
     understanding_symmetric_transfer
                  m_coroutine.promise().set_continuation(awaitingR
outine);
                  return m_coroutine;
              }

              void await_resume()
              {
              }

              std::coroutine_handle<promise_type> m_coroutine;
         };
```

2.4 Модульне та інтеграційне тестування розроблених компонентів

Під час програмування вбудованих систем актуальним залишається питання проведення інтеграційного та модульного тестування розроблених

модулів. Для вбудованого програмного забезпечення є можливість використовувати емулятори процесорних архітектур для розгортання оточення з необхідною операційною системою. Для bare-metal рішень не завжди є можливим використовувати емулятор системи. Також, при розробці прошивки може виникнути необхідність виконати часткове тестування розробленого драйверу високого рівня на базі host-оточення, без наявності цільовою платформи. У такому випадку доцільним є використання фреймворків для розробки модульних тестів та написання шару емуляції апаратних блоків з використанням Mock та Stub компонентів.

Розглянемо підхід для тестування драйвера SPI високого рівня з сопрограмними компонентами та його інтеграцію у середовище з апаратно-залежними модулями.

Реалізація драйверу SPI виконана як шаблонний клас, до якого можна передати тип, з яким цей шаблон має бути інстанційований. Наприклад, це рішення можна використовувати для передачі платформо-залежних компонентів від Nordic SDK, а з метою тестування передавати Mock-об’єкт що відтворює інтерфейс конкретного типу драйвера. У загальному випадку, користувач драйверу SPI модуля працює з методами transmitBuffer та exchangeBuffer, тобто або виконати передачу або виконати прийом та передачу.

У лістингу 2.13 наведено фрагмент реалізації методу transmitBuffer який виконує формування транзакцій та зберігає coroutine_handle для подальшого відновлення у заданому контексті виконання. У свою чергу, конкретний драйвер на платформі має реалізувати методи sendChunk для відправки заданого фрагменту та callback-виклик функції для сповіщення драйвера верхнього рівня про завершення транзакції.

Лістинг 2.13 – Фрагмент реалізації драйверу верхнього рівня та його інтеграції з платформо-залежним компонентом

```
void transmitBuffer(
    const std::uint8_t* pBuffer,
```

```
        std::uint16_t pBufferSize,
        void* pUserData,
        bool restoreInSpiCtx) noexcept
    {
        // встановили coroutine_handle
                                            m_coroHandle         =
std::coroutine_handle<>::from_address(pUserData);
            // встановили callback-метод що буде викликано у
драйвера низького рівня SPI кокнретної платформи
             m_backendImpl.setTransactionCompletedHandler([this]
{ transmitCompleted(); });

        const size_t TransferBufferSize = pBufferSize;
                  const   size_t   FullDmaTransactionsCount   =
TransferBufferSize / DmaBufferSize;
                 const   size_t   ChunkedTransactionsBufSize   =
TransferBufferSize % DmaBufferSize;
                   const   bool   ComputeChunkOffsetWithDma   =
FullDmaTransactionsCount >= 1;

        // сформували контекст транзакції
             TransactionContext  newContext{.restoreInSpiCtx  =
restoreInSpiCtx,
                                          .computeChunkOffsetWithDma
= ComputeChunkOffsetWithDma,
                                                  .pDataToTransmit =
pBuffer,
                                           .fullDmaTransactionsCount
= FullDmaTransactionsCount,
                                          .chunkedTransactionBufSize
= ChunkedTransactionsBufSize,
                                          .completedTransactionsCoun
t = 0};

        m_transmitContext = std::move(newContext);

        // виконали передачу даних
        if (FullDmaTransactionsCount)
        {
            --m_transmitContext.fullDmaTransactionsCount;
            m_backendImpl.sendChunk(pBuffer, DmaBufferSize);
        }
        else
        {
            m_transmitContext.chunkedTransactionBufSize = 0;
            m_backendImpl.sendChunk(pBuffer, pBufferSize);
        }
    }
```

Реалізація методу на конкретній платформі може бути як синхронною, так і асинхронною у випадку використання DMA або переривань для

обробки завершення передачі. Nordic процесори у своєму складі мають DMA модуль що дозволяє за одну транзакцію відправити не більше 255 байтів, таким чином є необхідним реалізувати функцію формування DMA-виклику на шину та обробку переривання по завершенню передачі буфера DMA. Фрагмент реалізації платформо-залежного драйверу наведено у лістингу 2.14.

Лістинг 2.14 – Фрагмент реалізації платформо-залежного компоненту для роботи з DMA модулем SPI

```
        void xferChunk(
             std::span<const std::uint8_t> _transmitArray,
             std::span<std::uint8_t> _receiveArray)
         {
             // формування transmit-receive транзакції
             nrfx_spimxfer_desc_t xferDesc = NRFX_SPIMXFER_TRX(
                 _transmitArray.data(),
                 _transmitArray.size(),
                 _receiveArray.data(),
                 _receiveArray.size());

             nrfx_err_t transmissionError = nrfx_spimxfer(
                 &SpiInstance::HandleStorage[PeripheralInstance::
HandleIdx], &xferDesc, 0);
             APP_ERROR_CHECK(transmissionError);
         }

                        using    TTransactionCompletedHandler    =
std::function<void()>;
                      void    setTransactionCompletedHandler(const
TTransactionCompletedHandler& _handler) noexcept
         {
             m_transactionCompleted = _handler;
         }

     private:
             static  void  spimEventHandlerThisOne(nrfx_spim_evt_t
const* _pEvent, void* _pContext) noexcept
         {
                // даний обробник буде виклиакано по завершенню
роботи  EasyDMA
             if (_pEvent->type == NRFX_SPIM_EVENT_DONE)
             {
                     // викик встановленого callback з драйверу
верхнього рівня
                                                 auto    pThis    =
reinterpret_cast<This_t*>(_pContext);
```

```
            pThis->m_transactionCompleted();
        }
    }
```

Для тестування драйверу верхнього рівня є необхідним розробити Mock, що буде повторювати інтерфейс платформо-залежного драфверу та реалізувати набір тестових сценаріїв.

Тестові сценарії, що можуть бути розглянуті, містять в собі передачу блоку даних менше, ніж розмір DMA буферу, передачу блоку даних розміром з DMA буфер, передачу декількох блоків розміром з DMA буфер та неповного блоку.

Гнучким механізмом створення сценаріїв тестування є параметричні тести фреймворку GTest, що дозволяють створити єдині шаги, що мають бути виконані, після чого цей тест може бути параметризований заданими аргументами.

Для підтримки параметричних тестів в фікстур необхідно додати ще однин клас, від якого йде спадкування тестової фікстури крім базового ::testing::Test. Підтримка параметрчних тестів вимагає успадкування від :: testing::WithParamInterface. Для реалізації тестових сценаріїв SPI інтерфейсу спадкування буде йти від

```
::testing::WithParamInterface<std::tuple<std::uint16_t,
std::string_view>>
```

де std::string_view буде відповідати назві тесту, а std::uint16 буде використана для задавання розміру буферу відаправки.

Реалізація драйвера низького рівня представлена у вигляді сховища транзакцій, що були виконані на шині SPI, кожний виклик sendChunk автоматично додається до контейнеру. По завершенню роботи є можливим виконати порівняння відправленого буферу з переданим шляхом перетворення сховища транзакцій до повного буферу. Реалізація тестового методу для збереження транзакцій наведена у лістингу 2.15.

Лістинг 2.15 – Реалізація методу sendChunk у класу, що являє собою імітацію драйвера низького рівня

```
    void sendChunk(const std::uint8_t* pBuffer, const size_t
bufferSize) noexcept
    {
                    BusTransactionsTransmit.emplace_back(pBuffer,
bufferSize);
        m_completedTransaction();
    }
    using TDataStream = std::vector<std::byte>;
    TDataStream getTransmittedData() const
    {
        TDataStream stream;
        std::ranges::for_each(BusTransactionsTransmit, [&stream]
(const auto& _transaction) {
            const auto& [pArray, blockSize] = _transaction;
            auto arraySpan = std::span{pArray, blockSize};
                                std::ranges::transform(arraySpan,
std::back_inserter(stream), [](std::uint8_t _dataByte) {
                return std::byte{_dataByte};
            });
        });
        return stream;
    }
```

Тестовий фрагмент буде створено як послідовність дій, що будуть нагадувати реальне використання драйвера високого рівня. У першу чергу є необхідним отримати розмір послідовності, що має бути передано. Після цього виконати ініціалізацію генератора випадкових послідовностей та сформувати випадковий буфер передачі даних заданого розміру. По завершенню тестування передачі необхідно порівняти буфер, що було передано з початковим. Також, є необхідним виконати очікування завершення сопрограми перед перевіркою результатів. Реалізація тестового сценарію та інстанціанювання параметричних тестів наведена у лістингу 2.16

Лістинг 2.16 – Реалізація тестового сценарію для перевірки коректності передачі даних драйвером верхнього рівня

```
    TEST_P(SpiDriverTest,
CheckRandomSequenceWithLengthTransmissionCorrect)
    {
        // Отримуємо розмір послідовності
```

```
                                        const        std::uint16_t
TransactionLength{std::get<0>(GetParam())};
        TDataStream ExpectedStream{TransactionLength};

        std::random_device randomDevice;
        std::mt19937 generator(randomDevice());
             std::uniform_int_distribution<>  distribution(0x00,
0xFF);
        // формуємо випадкову посилку
           std::ranges::generate(ExpectedStream,  [&distribution,
generator]() mutable {
             return std::byte(distribution(generator));
        });

        CoroUtils::syncWait(sendChunk(
                                             reinterpret_cast<const
std::uint8_t*>(ExpectedStream.data()), ExpectedStream.size()));
        // Звіряємо відправлені та прийняті данні у spi-backend
        EXPECT_EQ(TransactionsToDataStream(), ExpectedStream);
    }
    INSTANTIATE_TEST_SUITE_P(
        SpiDriverTesting,
        SpiDriverTest,
        ::testing::Values(
            std::make_tuple(Null, "Null"),
                std::make_tuple(SmallerThanSingleDmaTransaction,
"SmallerThanSingleDmaTransaction"),
                   std::make_tuple(EqualsToSingleDmaTransaction,
"EqualsToSingleDmaTransaction"),
                           std::make_tuple(ThreeDmaTransactions,
"ThreeDmaTransactions"),
                              std::make_tuple(SingleDmaAndChunk,
"SingleDmaAndChunk"),
                               std::make_tuple(ThreeDmaAndChunk,
"ThreeDmaAndChunk"),
            std::make_tuple(StressChunked, "StressChunked")),
        [](const auto& testParam) {
                                 const    auto&    suiteName    =
std::get<1>(testParam.param);
            return std::string(suiteName.data());
        });
```

2.5 Розробка та тестування драйверу SPI-FLASH з використанням сопрограмних примітивів

Розглянемо варіант використання сопрограмних примітивів для розробки драйверу SPI-FLASH пам’яті. Для першої ітерації виконаємо реалізацію зчитування JEDECID, реазіуємо unit-тести для цього модуля та

додамо реалізації запису та зчитування сторінки з пам’яті. Умови реалізації є такими самими, як і у випадку SPI-дисплею, а саме реалізація має бути асинхронною, використовувати максимально ефективно DMA та не блокуватися на очікуванні передачі.

Результат, що має бути отримано у ході розробки драйверу наведено у лістингу 2.17.

Лістинг 2.17 – Запланований вигляд розроблюємого компоненту під час використання

```
    void Board::initBoardSpiFlash() noexcept
    {
        m_pFlashDriver = std::make_unique<Hal::TFlashDriver>();
        if (m_pFlashDriver)
        {
                    const std::uint32_t JedecId = co_await
m_pFlashDriver->requestJEDEDCId();
            LOG_DEBUG("Jedec Id is:");
            LOG_DEBUG_ENDL(fmt::format("{:#04x}", JedecId));

              const std::span<std::uint8_t> DeviceId = co_await
m_pFlashDriver->requestDeviceId();
                            LOG_DEBUG_ENDL(fmt::format("{:02X}",
fmt::join(DeviceId, "")));
        }
    }
```

Для цього нам необхідно реалізувати примітив, що дозволить повернути на викликаючу сторону значення с сопрограми, що виконалась. Розглянемо розширений інтерфейс VoidTask, а саме зміни, що мають бути внесені до реалізації.

У першу чергу необхідно змінити тип get_return_object у task_awaitable, для підтримки повертання значення. Також необхідно додати змінну, що буде зберігати значення з сопрограми та повертати його.

Для реалізації зчитування JDECID необхідно реалізувати сопрограму, що буде мати повертаючий тип Task<uint32_t> та виконувати читання з заданого регістру чипу SPI-FLASH пам’яті. З урахованням асинхронної природи взаємодії компонентів, реалізація буде використовувати інтерфейс

верхнього рівня, який було розглянуто раніше. Ідея з std :: forward_as_tuple наступна – все, що передано в аргументах - в функції prepareXferTransaction буде розглянуто як команда та кількість dummy-bytes які потрібні для прийняття команди мікросхемою пам’яті. Все що після forward_as_tuple - кількість dummy-bytes для вичитки даних.

Реалізація зчитування JEDECID наведена у лістингу 2.18.

Лістинг 2.18 – Реалізація зчитування JEDEC ID з використанням розроблених сопрограмних примітивів

```
    CoroUtils::Task<std::uint32_t> requestJEDEDCId() noexcept
    {
        auto receivedData = co_await prepareXferTransaction(
            std::forward_as_tuple(WindbondCommandSet::ReadJedecI
d),
            WindbondCommandSet::DummyByte,
            WindbondCommandSet::DummyByte,
            WindbondCommandSet::DummyByte);

        std::uint32_t JedecDeviceId{};
                        for     (std::size_t     i{};     i     <
WindbondCommandSet::JedecIdLength; ++i)
        {
            JedecDeviceId |= (receivedData[i] << (16 - i * 8));
        }

        co_return JedecDeviceId;
    }
```

Реалізація функції prepareXferTransaction наступна: отримуємо на вхід команду у вигляді tuple та кількості порожніх посилок. Далі, заповнюємо буфер передачі спочатку командою та її аргументами, передаємо, після виконуємо передачу байті 0x00. Для значення, що повертається повертаємо slice на буфер з прийнятими даними. Реалізація функції prepareXferTransaction наведена у лістингу 2.19.

Лістинг 2.19 – Реалізація функції prepareXferTransaction на базі std::tuple з командою та її аргументами

```
template <typename TCommand, typename... Args>
```

```
                        CoroUtils::Task<std::span<std::uint8_t>>
prepareXferTransaction(
            TCommand&& command,
            Args&&... argList)
        {
                         auto&   transmitBuffer   =   getSpiBus()-
>getDmaBufferTransmit();
                          auto&   receiveBuffer   =   getSpiBus()-
>getDmaBufferReceive();

            getSpiBus()->setCsPinLow();

                           processTransmitBuffer(transmitBuffer,
std::forward<TCommand&&>(command));

                         constexpr   std::size_t   CommandSize   =
std::tuple_size_v<TCommand>;

                                                          co_await
transmitChunk(std::span(transmitBuffer.data(), CommandSize));

                constexpr std::size_t DummyListSize = sizeof...
(argList);

                           processTransmitBuffer(transmitBuffer,
std::forward_as_tuple(argList...));

            auto receivedBlockSpan = co_await xferTransaction(
                std::span(transmitBuffer.data(), DummyListSize),
                std::span(receiveBuffer.data(), DummyListSize));

            getSpiBus()->setCsPinHigh();
                     co_return  std::span(receiveBuffer.data(),
DummyListSize);
        }
```

Розглянемо процесс тестування запису даних до флеш-пам’яті. Для виконання запису необхідно виконати запис команди write enable, після чого виконати запис команди program та відправити данні, що мають бути записані до чіпу. Ля перевірки коректності запису є необхідним використовувати механізм тестування testing::Sequence з макросом EXPECT_CALL де можливо встановити очікувану послідовність викликів записів до драйверу SPI та перевірити коректність переданих аргументів до функції. Фрагмент реалізації процессу тестування запису наведено у

лістингу 2.20. Реалізація примітиву SyncWait з його детальним розгляданям наведена у [27].

Лістинг 2.20 – Фрагмент реалізації процесу тестування запису даних до мікросхеми флеш-пам’яті.

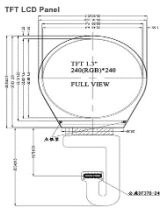
```
        testing::Sequence sequence;

                                                                //
https://gist.github.com/cppengineer/f1b6bc0f04ac7c29e963364f2c56
4a5e

        const auto writeEnableSpan =
                                                  std::span<const
std::uint8_t>(writeEnableCommand.data(),
writeEnableCommand.size());

             const auto ProgramCommandSpan = std::span<const
std::uint8_t>(pageProgramCommand.data(),
pageProgramCommand.size());
        EXPECT_CALL(spiMockAccess(), sentData)
            .With(SpanChecker(writeEnableSpan))
            .Times(1)
            .InSequence(sequence);

        EXPECT_CALL(spiMockAccess(), sentData)
            .With(SpanChecker(ProgramCommandSpan))
            .Times(1)
            .InSequence(sequence);
        EXPECT_CALL(spiMockAccess(), sentData)
                 .With(SpanChecker(std::span(TransmitData.data(),
TransmitData.size())))
            .Times(1)
            .InSequence(sequence);

                auto task = flashDriver.pageWrite(address,
std::span(TransmitData.data(), TransmitData.size()));
        CoroUtils::syncWait(task);
```

2.6 Тестування на апаратній платформі розроблених компонентів

Розробка і тестування були проведені на платі налагодження, розробленої під дослідження, а саме на базі рішення з використанням процесора сімейства Nordic. Цільова платформа була обрана відповідно до необхідних засобів стандарту C++. Для експерименту необхідна підтримка

стандарту C++20. Для практичного експерименту вибрано порт FreeRTOS для ARM Cortex M4F. На платі присутні дисплейний модуль з інтерфейсом SPI, SPI-FLASH пам'ять, процесор з підтримкою BLE стека. Операційна система буде використана тільки в порівнянні мінімальних витрат на її розгортання. У частині порівняльних прикладів буде використано bare-metal оточення. Детальна специфікація приведена в таблиці 2.1.

Таблиця 2.1 – Використані компоненти на платі налагодження

Назва компоненту	Вигляд компоненту	Основні характеристики компоненту
NRF52832		-Bluetooth 5 - 2Mbs mode - 32-bit ARM Cortex M4F processor - 512kB flash + 64kB RAM -RAM mapped FIFOs using EasyDMA -128-bit AES ECB/CCM/AAR co-processor
ST7789V Display		- Resolution 240*240px - IPS Matrix - 1.3” - SPI/I2C/Parallel interfaces
W25Q16BVS		-8M-BIT, 16M-BIT AND 32M-BIT SERIAL FLASH MEMORY -DUAL AND QUAD SPI
Saleae Logic Analyzer		8-channel logic analyzer with up to 24MHz sampling rate.

Апаратна частина плати і схемотехніка реалізовані з використанням САПР KiCAD. Додатково виведені коннектори для підключення логічного аналізатора та / або інших периферійних модулів. Трасування плати наведено на рис 2.5.

Дана плата розроблена для експерименту з можливостями швидкого підключення логічного аналізатора. Плата містить інтерфейсні роз'єми SPI і

I2C для різних периферійних пристроїв. Доступні бортові пристрої включають дисплей, роз'єм для датчика пульсоксиметра I2C, акселерометр IMU I2C з вбудованим модулем DMP. SPI-FLASH розведена для подальших експериментів із сопрограмами інтеграції до існуючих бібліотек на основі C для гнучкого використання як компонентів C++ з асинхронною взаємодією.

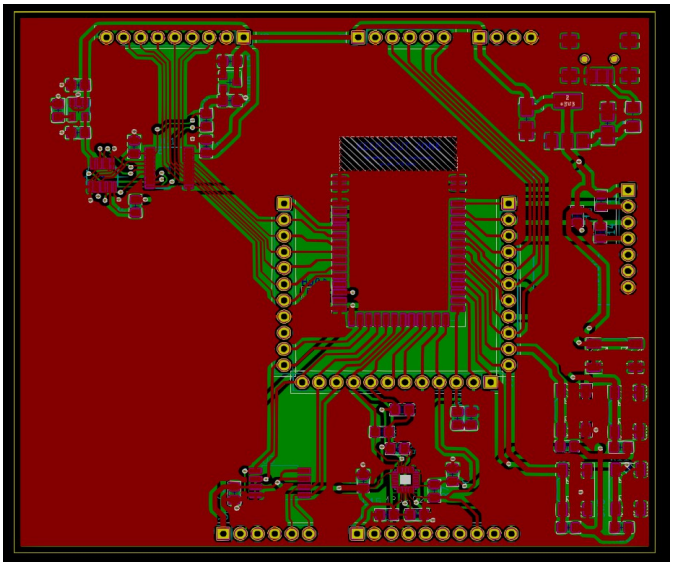


Рисунок. 2.5 – Трасування плати для проведення дослідження

Цільовий MCU ARM був обраний для експерименту через доступні інструменти для нього. Величезний набір доступних як відкритих, так і комерційних налагоджувачів дозволяє вибрати винятковий набір для розробки вбудованих додатків. Крім того, відсутня підтримка функцій C++20 для архітектур PIC або AVR, які були актуальними на початку.

Існують також інші платформи, які можна використовувати для перевірки результатів експерименту. Для цього може бути використана архітектура x86 з портом x86-FreeRTOS для симулятора. Крім того, відповідно можна використовувати сімейства процесорів ARM-Cortex A або ARM-Cortex-R.

Для компіляції прошивки з підтримкою сопрограм з стандарту C++ 20 необхідно використовувати компілятор GCC версії не нижче 10.1. Повна підтримка реалізована у версії 10.2. Для роботи в bare-metal оточенні

використовується версія arm-gcc-none-eabi з бібліотекою newlib-nano для використання C++-runtime. Для побудови та тестування проекту використовується CMake версії 3.14.

Також, проект включає в себе частину статичних бібліотек з NRFSDK для роботи з BLE сервісами, з метою забезпечення мінімального споживання пам’яті вони виключені на етапі компонування прошивки. Розроблені програмні модулі сумісні з компіляторами, що підтримуються C++20, і можуть використовуватися в кросплатформному середовищі. Приклади сопрограм завершуються як набір драйверів для дисплея для вимірювання часу передачі та проміжків між транзакціями. Опис обладнання прикладу EasyDMA наведено на сторінці Nordic інформаційного центру з детальною інформацією про структуру структури ArrayList. Також наведено приклади NRFSDK для реалізації інтерфейсу SPI з інтеграцією EasyDMA.[20]

Компілювання прикладів буде приведено з повністю виключеним оптимізаціями (O0), а так-же порівняльний варіант з використанням O3, Os, і з включеними межпроцедурнимі оптимізаціями лінковщіку.

Деякі опції збірки залишаються однаковими для всіх проектів, а саме,:
для лінковщіку:

```
    -Wl,--print-memory-usage,--gc-sections    --specs=nano.specs
-lc -lnosys -lm
    Для компилятора:
    -MP -MD -mthumb -mabi=aapcs -Wall -g0 -ffunction-sections
-fdata-sections -fno-strict-aliasing -fno-builtin --short-enums
-fno-exceptions
```

Специфічні для використовуваного процесору:

```
    -mcpu=cortex-m4 -mfloat-abi=hard -mfpu=fpv4-sp-d16
```

3 РЕЗУЛЬТАТИ ДОСЛІДЖЕННЯ

Для перевірки запропонованої моделі були виконані експериментальні перевірки передачі даних.

Так само як приклад таких завдань можна розглядати приклад реалізації неблокуючих затримок на апаратних / програмних таймерах.
Передача даних по шині здійснюється послідовно, після передачі одного блоку необхідне відновлення стану передавальної функції для вибірки нового блоку даних., Що може бути актуальним при використанні інтерфейсів з блочними DMA транзакціями.

Більш ускладненим прикладом може бути розглядатися задача обміну даними із зовнішньою пам'яттю, вичитування блоку даних з неї, обробкою і записом назад. Подібна задача має місце і в разі використання дисплейних модулів з внутрішнім кадровим буфером, коли необхідно передати кілька команд для вказівки початкового і кінцевого адреси, відтворення і після передати потік байт, відповідних зображенню в заданому колірному форматі.

У разі деяких особливостей DMA периферії, можуть бути обмеження по утилізації периферійної шини, особливостей настройки і максимального переданого блоку за одну транзакцію [14]. Наприклад, налаштування модуля EasyDMA для процесорів сімейства Nordic, згідно [17], має можливість використання PPI. Але, при доступних можливостях є обмеження за розміром транзакції для периферії сімейства NRF52832, а саме у вигляді 255 байт за одну транзакцію. З використанням автоінкремента адреси і ArrayList розмір може бути збільшений.

Не дивлячись на наявність в типових RTOS сопрограм, як окремого компонента, вони мають обмеження, пов'язані з використанням локальних змінних, окремими наборами викликів для використання компонентів ОС, обмеженим набором примітивів синхронізації або їх повною відсутністю.

Розглянуті приклади були скомпільовані з різними оптимізаційними настройками, а саме Os / Os + LTO для мінімальної конфігурації і конфігурація з O0 для неоптимізованого варіанту[28].

У таблиці 2 наведені результати споживання пам'яті та часу виконання деяких з наведених прикладів скомпільованими на різних конфігураціях оптимізацій.

Таблиця 3.1 – Результати споживання пам'яті та часу виконання

Приклад	Опції компіляції	Використання пам’яті	Середній час виконання
Мін. Приклад FreeRTOS	O0	Flash: 10324B RAM: 2436	Не вимірювалось
Мін. Приклад FreeRTOS	Os	Flash: 10324B RAM: 2436	Не вимірювалось
Мін. Приклад Сопрограми	O0	FLASH: 3192 B RAM:824 B	Не вимірювалось
Мін. Приклад Сопрограми	Os + LTO	FLASH:3036 B RAM:824 B	Не вимірювалось
Мін.приклад кооперативний планувальник+ FreeRTOS+дві задачі	Os	FLASH:10428B RAM: 2440B	8.01us
Мін.приклад кооперативний планувальник+ дві задачі	Os+LTO	FLASH:3260 RAM: 824B	4.28us
Передача даних, транзакції	Os+LTO	FLASH: 19740B RAM: 1592B	74.25us
Передача даних, сопрограми+ std::tuple	Os+LTO	FLASH: 12696B RAM:1884B	71.9us
Передача даних, сопрограми + масив	Os+LTO	FLASH: 9684B RAM: 2596B	41.5 us

Скриншоти з Saleae Logic Analyzer наведені на рис. 3.1 – 3.2.

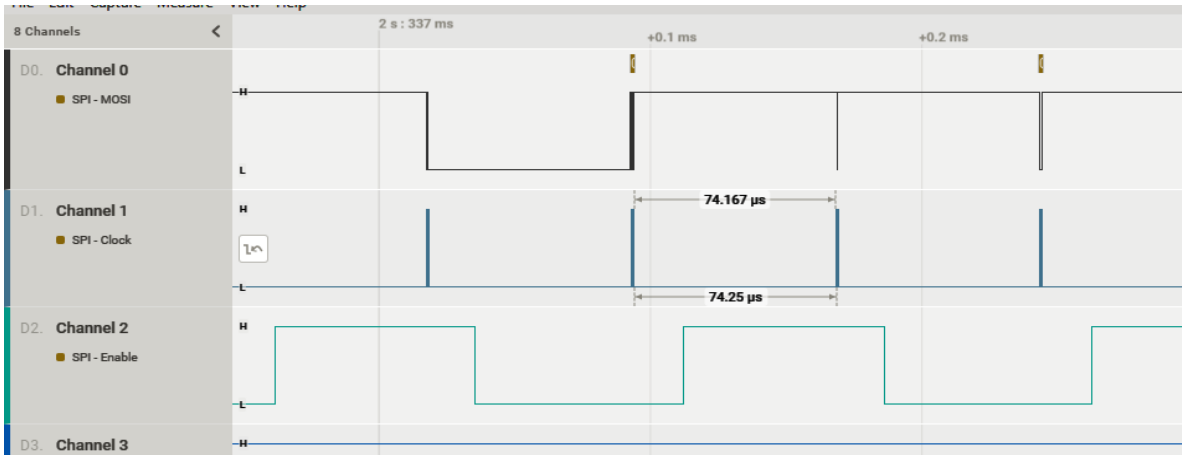


Рисунок 3.1 – Передача даних SPI з використанням транзакційного підходу

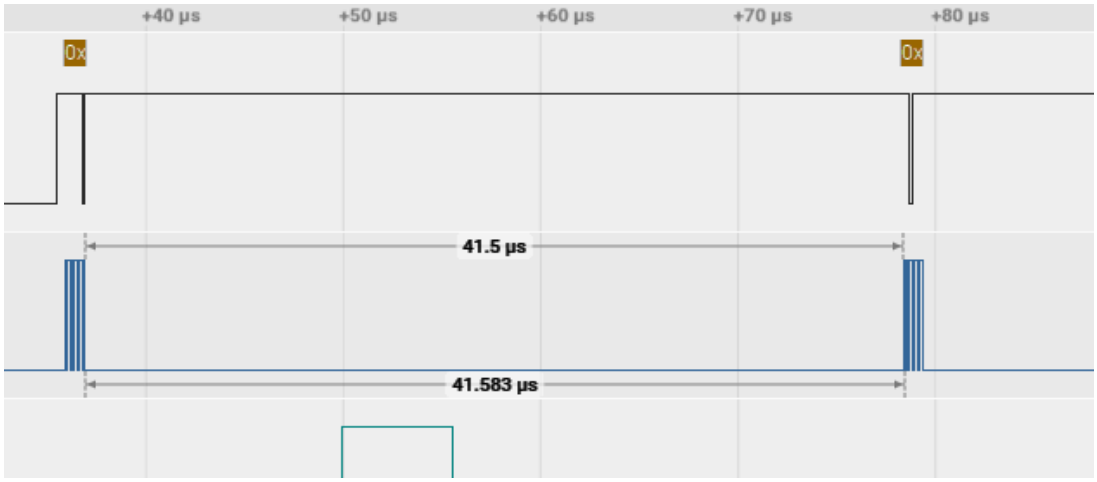


Рисунок 3.2 – Передача даних SPI з використанням сопрограммного підходу

4 АНАЛІЗ ОТРИМАНИХ РЕЗУЛЬТАТІВ

В результаті виконання курсового проекту була розроблена модель використання сопрограм для програмування вбудованих систем.

В ході дослідження виконано аналіз споживаних ресурсів компонентами на базі ОСРЧ і сопрограм. Реалізовано мінімальні примітиви роботи з сопрограмами в задачах забезпечення кооперативної багатозадачності, архітектурних фрагментів для реалізації неблокуючої передачі даних.

Запропоновано модель мінімального планувальника з інтеграцією в компонент передачі даних з сопрограммним збереженням контексту.

Дослідження показало перспективність використання співпрограми в системах з обмеженими ресурсами через значний виграшу як по використанню FLASH, так і RAM пам'яті. При відсутності необхідності в механізмі жорстких пріоритетів і детермінованою реакції системи на зовнішнє подія сопрограмна кооперативна багатозадачність може бути вирішенням проблеми забезпечення багатозадачності на невеликій системі на базі МК. Порівняльний аналіз також показав, що сопрограммний підхід дозволяє забезпечити необхідний рівень стиснення і оптимізації коду компілятором з можливістю розміщення стека сопрограми на стороні виклику при невеликій глибині стека викликів. Середні результати для моделі сопрограм показують підвищення продуктивності на 30% за час виконання та в 3 рази менше споживання пам’яті, ніж підхід RTOS.

Подальші дослідження в області застосування нового функціоналу C + + 20 для вирішення завдань в сфері програмування вбудованих систем можуть включати в себе спектр завдань, що відносяться до областям

типобезпечного програмування, удосконалення та використання constexpr new operator, проектування і розгляду користувальницьких аллокаторів для сопрограм і завдань, пов'язаних з проектуванням високонавантажених систем обробки даних з мінімальним енергоспоживанням. Є можливою розробка інтеграційних компонентів для спрощення інтеграції з існуючими бібліотеками графічного інтерфейсу для користувача [29]. Типові завдання сопрограм у вбудованому програмному забезпеченнні можуть включати завдання перемикання між мережею та потоком інтерфейсу користувача в програмі з обробкою отриманих даних через канал зв'язку.

Зміни, які можуть бути включені в стандарт C ++ 23 включають в себе поліпшення стандарту, пов'язаними з freestanding реалізацією компонентів стандартної бібліотеки з метою поліпшення інтеграції в системи, де відсутня можливість динамічного виділення пам'яті, використання мінімально необхідного функціоналу стандартної бібліотеки

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

Practical Guide to Bare Metal C++ [Електронний ресурс] // arobenko. – 2018. – Режим доступу до ресурсу: https://legacy.gitbook.com/book/arobenko/bare_metal_cpp/details.

Real-Time C++ : Efficient Object-Oriented and Template Microcontroller Programming – Berlin, Germany: Springer-Verlag Berlin and Heidelberg GmbH & Co. KG, 14. – 426 c. – (Springer-Verlag Berlin and Heidelberg GmbH & Co. KG).

Samek M. Practical UML Statecharts in C/C++ Event-Driven Programming for Embedded Systems / Miro Samek., 2009. – 728 c.

Gor N. A proposal to add coroutines to the C++ standard library (Revision 1) [Електронний ресурс] / Nishanov Gor // Programming Language C++, SG1. – 2014. – Режим доступу до ресурсу: https://isocpp.org/files/papers/n3985.pdf.

1. P.Long, G. Fei, P. Luc, T. Martin.: Behaviour and performance comparison between FreeRTOS and µC/OS-III. International Journal of Embedded Systems, 2016, DOI: 10.1504/IJES.2016.077774.

Pilarski D. YOUR FIRST COROUTINE [Електронний ресурс] / Dawid Pilarski // PanicSoftware blog. – 2019. – Режим доступу до ресурсу: https://blog.panicsoftware.com/your-first-coroutine/.

Lewiss B. C++ Coroutines: Understanding operator co_await [Електронний ресурс] / Baker Lewiss // https://lewissbaker.github.io/2017/11/17/understanding-operator-co-await. – 2019.

luncliff. Exploring MSVC Coroutine¶ [Електронний ресурс] / luncliff. – 2019. – Режим доступу до ресурсу: https://luncliff.github.io/coroutine/articles/exploring-msvc-coroutine/.

Korniienko V. R. co_await issue in constexpr functions(Non-standard behavior) [Електронний ресурс] / Valentyn Ruslanovich Korniienko – Режим доступу до ресурсу: https://developercommunity2.visualstudio.com/t/co_await-issue-in-constexpr-functions-N/1301534.

Boost coroutine introduction [Електронний ресурс] – Режим доступу до ресурсу: https://www.boost.org/doc/libs/1_57_0/libs/ coroutine/doc/html/coroutine/intro.html#coroutine.intro.stackfulness.

C++20. Coroutines [Електронний ресурс]. – 2020. – Режим доступу до ресурсу: https://habr.com/ru/post/519464/.

luncliff. Combining C++ Coroutines and pthread_create [Електронний ресурс] / luncliff. – 2019. – Режим доступу до ресурсу: https://luncliff.github.io/coroutine/articles/combining-coroutines-and-pthread_create/.

Barry R. Mastering the FreeRTOS™ Real Time Kernel [Електронний ресурс] / Richard Barry // © Real Time Engineers Ltd. – 2016. – Режим доступу до ресурсу: https://www.freertos.org/fr-content-src/uploads/2018/07/ 161204_ Mastering_the_FreeRTOS_Real_Time_Kernel-A_Hands-On_Tutorial_Guide.pdf.

Dogan I. Arm-Based Microcontroller Multitasking Projects Using the FreeRTOS Multitasking Kernel / Ibrahim Dogan., 2020. – 524 с.

ISOCPP. Resumable Functions (revision 4) [Електронний ресурс] / ISOCPP – Режим доступу до ресурсу: https://isocpp.org/files/papers/ N4402.pdf.

Compiler support [Електронний ресурс] // cppreference – Режим доступу до ресурсу: https://en.cppreference.com/w/cpp/compiler_support.

2. Anthony Williams C++ Concurrency in Action.pdf, 2019 ISBN 9781617294693

3. Jeff Langr, Modern C++ Programming with Test-Driven Development, ISBN-10: 1937785483

4. Alexander van der Grinten. Managarm: A Fully Asynchronous OS Based on Modern C++ [Електронний ресурс] / Alexander van der Grinten – Режим доступу до ресурсу: https://managarm.org/.

EasyDMA interface [Електронний ресурс] // Nordic Semiconductor Infocenter – Режим доступу до ресурсу: https://infocenter.nordicsemi.com/index.jsp?topic=%2Fcom.nordic.infocenter.nrf52832.ps.v1.1%2Feasydma.html.

5. Yiu J. –Elsevier Inc, The Definitive Guide to the ARM Cortex-M3 / Joseph Yiu., 2010.

6. Dunkels A. protothreads [Електронний ресурс] / Adam Dunkels – Режим доступу до ресурсу: http://dunkels.com/adam/pt/.

7. HandWiki Coroutines [Електронний ресурс] // HandWiki – Режим доступу до ресурсу: https://handwiki.org/wiki/Coroutine#Implementations_for_C.2B.2B.

8. Coroutines documentation [Електронний ресурс] – Режим доступу до ресурсу: https://en.cppreference.com/w/cpp/language/coroutines.

9. Mazière D. My tutorial and take on C++20 coroutines [Електронний ресурс] / David Mazière – Режим доступу до ресурсу: https://www.scs.stanford.edu/~dm/blog/c++-coroutines.html.

10. Bond M. C++20 Coroutines [Електронний ресурс] / Martin Bond – Режим доступу до ресурсу: https://blog.feabhas.com/2021/09/c20-coroutines.

11. Korniienko V. R. C++20 Coroutines with the NRF52832 MCU and GTest [Електронний ресурс] / Valentyn Ruslanovich Korniienko // Habrahabr. – 2021. – Режим доступу до ресурсу: https://habr.com/ru/post/566070/.

12. Development of coroutines usage model for cooperative multitasking implementation on the systems with limited resources [Електронний ресурс] / Amin Salih Mohammed, I. Filippenko, B. B. Saravana та ін.] // Springer. – 2021. – Режим доступу до ресурсу: https://link.springer.com/article/10.1007%2Fs10479-021-04417-1.

13. Філіппенко І. В. ОГЛЯД ГРАФІЧНИХ БІБЛІОТЕК ДЛЯ ВБУДОВАНИХ ПЛАТФОРМ / І. В. Філіппенко, В. Р. Корнієнко, Г. К. Кулак. // RI. – 2020. – С. 47–53.