МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»

Основи проектування систем Інтернету речей. Периферія мікроконтролерів STM32

Конспект лекцій

Навчальний посібник

Рекомендовано Методичною радою КПІ ім. Ігоря Сікорського
як навчальний посібник для здобувачів ступеня бакалавра
за сертифікатною програмою «Електронні охоронні системи та засоби Інтернету речей»
спеціальності 171 «Електроніка»

Укладачі: Ю.О. Оникієнко, А.Р. Рижова

Електронне мережне навчальне видання

Київ
КПІ ім. Ігоря Сікорського
2022

Рецензент *Клен К. С.* доцент, к.т.н., доцент кафедри ЕПС КПІ ім. Ігоря Сікорського

Відповідальний редактор *Попович П.В.*, к.т.н., доцент

Гриф надано Методичною радою КПІ ім. Ігоря Сікорського
(протокол № 6 від 24.06.2022 р.)
за поданням Вченої ради факультету електроніки
(протокол № 5/22 від 31.05.2022 р.)

В роботі наведено основні відомості про архітектуру та особливості мікроконтролерного ядра ARM. Розглянуто особливості побудови та взаємодії периферійних пристроїв та системи переривань мікроконтролерів STM32. Проаналізовано особливості реалізації блоку переривань, портів вводу-виводу, універсального послідовного асинхронного, I^2C та SPI інтерфейсів. Особливу увагу приділено детальному опису програмній реалізації механізмів керування периферійними пристроями та перериваннями. Пояснення супроводжуються прикладами коду. Даний конспект лекцій є частиною курсу по вивченню мікроконтролерів. Посібник призначений для здобувачів ступеня бакалавра та магістра за спеціальністю «Електроніка», які навчаються по сертифікатній програмі «Електронні охоронні системи та засоби Інтернету речей». Також буде корисний розробникам систем на 32-х бітних мікроконтролерах та студентам відповідних спеціальностей.

Реєстр. № НП XX/XX-XXX. Обсяг 5,5 авт. арк.

Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»
проспект Перемоги, 37, м. Київ, 03056
https://kpi.ua
Свідоцтво про внесення до Державного реєстру видавців, виготовлювачів
і розповсюджувачів видавничої продукції ДК № 5354 від 25.05.2017 р.

ЗМІСТ

Вступ

Вбудовані системи на мікроконтролерах є базовою складовою систем безпеки та систем Інтернету речей. Кожен пристрій збору та передачі інформації має у складі, як мінімум, один мікроконтролер, який виконує основні задачі по збору, обробці та передачі інформації. Тому вибір типу мікроконтролеру є критично важливим для створення апаратної частини системи безпеки або Інтернету речей.

Мікроконтролери STM32 є одними з кращих для використання в вбудованих системах завдяки використанню 32-х бітного мікропроцесорного ядра ARM з розвинутою архітектурою, яка забезпечує високу швидкість обчислень, гнучку систему переривань та різні режими енергоспоживання. Також мікроконтролери STM32 мають велику кількість периферійних пристроїв для вирішення різноманітних задач.

В посібнику насамперед приділено увагу основним комунікаційнім інтерфейсам мікроконтролерів STM32, а саме портам вводу-виводу, UART, I^2C, SPI. Також розкрито особливості побудови та роботи системи переривань, які використовуються для забезпечення роботи комунікаційної периферії. Тобто для вивчення пропонується компоненти мікроконтролера, які в першу чергу задіяні в роботі розподілених охоронних систем та систем Інтернету речей. Наведені інтерфейси використовуються для комунікації через канали прийому передачі даних, а також для отримання інформації від різноманітних датчиків. Інші периферійні пристрої, такі як кола осциляції, таймери, DMA, АЦП, ЦАП, RTC та ін. будуть розглянуті в наступних частинах посібника.

Лекція 1. Коротка довідка про процесори ARM

Архітектура ARM — набір специфікацій, який описує роботу мікропроцесорного ядра ARM та містить моделі виконання, набір інструкцій, організацію та розташування пам'яті. Якщо компілятор може генерувати інструкції збірки для мікропроцесора даної архітектури, він може генерувати машинний код для всіх мікропроцесорів, що реалізують цю архітектуру. Під терміном ARM сьогодні мають на увазі кілька сімейств ядер процесорів зі скороченим набором команд (RISC), які є основою великої кількості мікроконтролерів від різних виробників.

ARM Holdings — це британська компанія, яка розробила набір інструкцій та архітектуру для продуктів на основі ARM, але сама не виробляє пристрої. Процесори на базі ARM ядра забезпечують близько 75 відсотків мобільних пристроїв у світі. Багато основних і популярних 64-розрядних і багатоядерних процесорів, які використовуються в пристроях, які стали символами в електронній промисловості (наприклад, iPhone від Apple), засновані на архітектурі ARM (ARMv8-A).

Cortex-M — це сімейство ядер, розроблене для подальшої інтеграції з периферійними пристроями виробника, щоб утворити готовий мікроконтролер. Спосіб роботи ядра визначається не тільки його пов'язаною архітектурою ARM (наприклад, ARMv7-M), але також інтегрованими периферійними та апаратними можливостями, що визначені виробником кремнію. Наприклад, архітектура ядра Cortex-M4 розроблена для підтримки операцій доступу до бітових даних у двох конкретних областях пам'яті з використанням функції, яка називається бітовою смугою, але додавати таку функцію чи ні, залежить від фактичної реалізації. На ядрі Cortex-M4 засновано сімейство мікроконтроллерів STM32F4, що реалізує цю функцію бітового

діапазону. ST Microelectronics наразі є єдиним виробником, який продає повний спектр процесорів на основі Cortex-M.

Завдяки широкому поширенню ARM архітектури, існує багато компіляторів та інструментів, а також операційних систем (Linux є найбільш використовуваною ОС на процесорах Cortex-A), які підтримують це ядро, пропонуючи розробникам безліч можливостей для створення своїх програм.

1. 1. Основні особливості ARM архітектури

1.1.1. Обробка винятків та переривань

Керування перериваннями та винятками є однією з найпотужніших функцій процесорів на основі Cortex-M. Переривання та винятки — це асинхронні події, які змінюють хід програми. Коли виникає виняток або переривання, центральній процесор призупиняє виконання поточного завдання, зберігає його контекст (тобто вказівник стека) і починає виконання підпрограми, призначеної для обробки події переривання. Ця процедура називається обробником винятків у разі винятків і підпрограмою обслуговування переривань *Interrupt Service Routine* (ISR) у разі переривання. Після обробки винятку або переривання центральний процесор відновлює попередній потік виконання, і виконання перерваного завдання продовжується.

В архітектурі ARM переривання є одним із типів винятків. Переривання зазвичай генеруються вбудованими периферійними пристроями (наприклад, таймером) або зовнішніми входами. Наприклад, тактильним перемикачем, підключеним до *General-purpose input/output* (GPIO), а в деяких випадках вони можуть бути викликані програмним забезпеченням. Натомість винятки

пов’язані з виконанням програмного забезпечення, і сам процесор може бути джерелом винятків. Це можуть бути події збою, наприклад спроба отримати доступ до недопустимого місця в пам’яті, або події, згенеровані операційною системою, якщо такі є.

Кожен виняток (і, отже, переривання) має номер, який однозначно ідентифікує його. Це число відображає положення підпрограми обробника винятків у векторній таблиці, де зберігається фактична адреса підпрограми. І винятки, і переривання обробляються спеціальним блоком під назвою *Nested Vectored Interrupt Controller* (NVIC). NVIC має такі функції:

- Гнучкість керування винятками та перериваннями: NVIC здатен обробляти як сигнали/запити переривань, що надходять від периферійних пристроїв, так і винятки, що надходять від ядра процесора, що дозволяє вмикати/виключати їх у програмному забезпеченні (крім немаскованих переривань). Виняток Reset не можна вимкнути, оскільки це перший виняток, створений після скидання MCU. Виняток Reset є фактичною точкою входу для кожної програми STM32.
- Підтримка вкладених винятків/переривань: NVIC дозволяє призначати рівні пріоритету виняткам і перериванням (за винятком перших трьох типів винятків), що дає можливість класифікувати переривання на основі потреб користувача.
- Векторний запис винятку/переривання: NVIC автоматично визначає положення обробника винятків, пов'язаного з винятком/перериванням, без потреби в додатковому коді.
- Маскування переривань: розробники можуть призупинити виконання всіх обробників винятків (крім немаскованих переривань)

або призупинити деякі з них на рівні пріоритету завдяки набору виділених регістрів. Це дозволяє безпечно виконувати критичні завдання, не маючи справу з асинхронними перериваннями.

- Детермінована затримка переривання: однією цікавою особливістю NVIC є детермінована затримка обробки переривань, яка дорівнює 12 циклам для всіх ядер Cortex-M3/4, 15 циклам для Cortex-M0, 16 циклам для Cortex-M0+, незалежно від поточний стан процесора.
- Переміщення обробників винятків: як ми будемо досліджувати далі, обробники винятків можуть бути переміщені в інші місця розташування флеш-пам'яті, а також зовсім іншу (навіть зовнішню) пам'ять, не призначену лише для читання. Це забезпечує велику гнучкість для просунутих додатків.

1.1.2. Системний таймер (SysTimer)

Процесори на основі Cortex-M можуть додатково забезпечувати системний таймер, також відомий як SysTick. SysTick — це 24-розрядний таймер зворотного відліку, який використовується для забезпечення системного інтервалів часу (тіків) для операційних систем реального часу (RTOS), таких як FreeRTOS. Він використовується для генерування періодичних переривань та для виконання запланованих завдань. Програмно можна задати частоту оновлення таймера SysTick, налаштуванням його регістрів. Таймер SysTick також використовується STM32 HAL для створення точних затримок, навіть якщо RTOS не використовується.

1.1.3. Режими енергоспоживання

Сучасна тенденція в електронній промисловості, особливо коли йдеться про розробку мобільних пристроїв, пов’язана з керуванням живленням. Зменшення енергоспоживання до мінімуму є головною метою всіх дизайнерів і програмістів, які займаються розробкою пристроїв, що живляться від батарей. Процесори Cortex-M забезпечують кілька рівнів керування живленням, які можна розділити на дві основні групи: вбудовані функції та режими живлення, визначені користувачем.

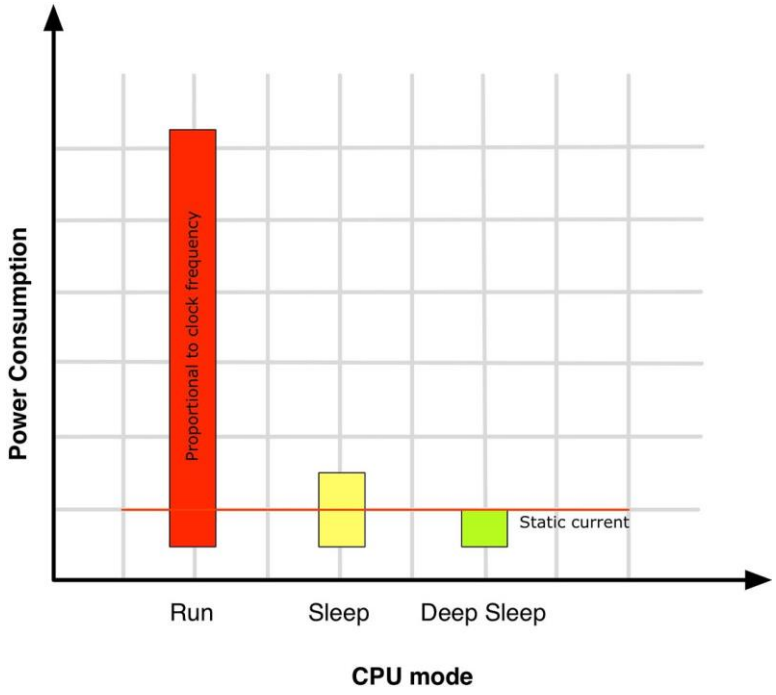

Рисунок 1.1 Споживання енергії Cortex-M при різних режимах живлення

Використання вбудованих функцій реалізовано на власних можливостях, пов’язаних зі споживанням енергії, визначені під час проектування як ядра Cortex-M, так і всього MCU. Наприклад, ядра Cortex-M0+, виконуючи конвеєрну обробку даних, визначають лише два етапи конвеєра, щоб зменшити споживання енергії під час попередньої вибірки

інструкцій. Іншою властивою поведінкою, пов’язаною з керуванням живленням, є висока щільність коду набору інструкцій Thumb-2, що дозволяє розробникам вибирати мікроконтролери з меншою флеш-пам’яттю, щоб зменшити потребу в споживанні енергії.

Традиційно процесори Cortex-M забезпечують визначені користувачем режими живлення через системний реєстр управління (SCR). Перший – це режим *Run mode* (див. Рисунок 1.1), у якому процесор використовує всі свої можливості. У даному режимі роботи споживання енергії залежить від тактової частоти та задіяних периферійних пристроїв.

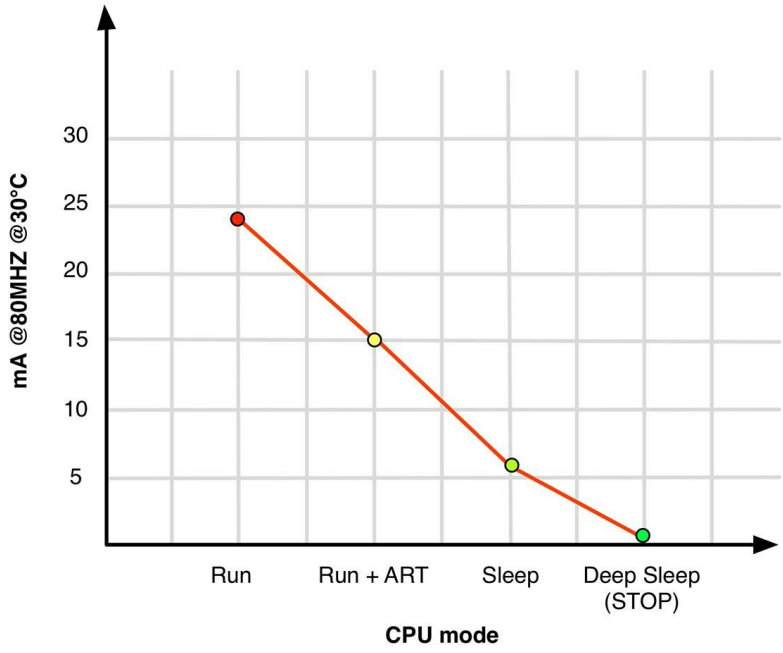

Рисунок 1.2 Споживання електроенергії STM32F2 при різних режимах живлення

Сплячий режим (*Sleep mode*) є першим доступним режимом низького енергоспоживання для зниження споживання енергії. Після активації більшість функціональних можливостей призупиняється, частота процесора знижується, а його діяльність зменшується до необхідної для його

пробудження. У режимі глибокого сну (*Deep sleep mode*) всі тактові сигнали припиняються, і ЦП потребує зовнішньої події, щоб вийти з цього стану.

Однак ці режими живлення є лише загальними моделями, які в подальшому реалізовані у фактичному MCU. Наприклад, на Рисунку 1.2 відображено споживану потужність мікроконтролера STM32F2, що працює на частоті 80 МГц при 30°C. Як видно з рисунка, максимальне споживання енергії досягається в режимі *Run* (тобто в активному режимі) з вимкненим *Adaptive Real-Time* (ART) прискорювачем. Увімкнувши ART прискорювач, можна заощадити до 10 мА, а також досягти кращої обчислювальної продуктивності.

1.1.4. Стандарт програмного інтерфейсу мікроконтролерів Cortex

Однією з ключових переваг платформи ARM (як для виробників мікроконтролерів, так і для розробників додатків) є наявність повного набору інструментів розробки (компілятори, бібліотеки, налагоджувачі тощо), які можуть випускатися кількома постачальниками.

ARM активно працює над тим, щоб стандартизувати інфраструктуру програмного забезпечення серед виробників MCU. Стандарт програмного інтерфейсу мікроконтролера *Cortex Microcontroller Software Interface Standard* (CMSIS) — це незалежний від виробника рівень апаратної абстракції для серії процесорів Cortex-M і визначає інтерфейси налагоджувачів. CMSIS складається з таких компонентів:

- *CMSIS-CORE*: API для процесора Cortex-M і периферійних пристроїв. Він забезпечує стандартизований інтерфейс для Cortex-M0/3/4/7;
- *CMSIS-Driver*: визначає загальні інтерфейси драйверів периферійних пристроїв для проміжного програмного забезпечення, завдяки чому їх можна використовувати на різних пристроях. API є незалежним від

RTOS і підключає периферійні пристрої мікроконтролера до проміжного програмного забезпечення, яке реалізує, серед іншого, стеки зв'язку, файлові системи або графічні інтерфейси користувача;

- *CMSIS-DSP*: колекція бібліотеки DSP з понад 60 функціями для різних типів даних: з фіксованою точкою і з плаваючою комою (32-розрядні). Бібліотека доступна для Cortex-M0, Cortex-M3 і Cortex-M4.
- *CMSIS-RTOS API*: загальний API для операційних систем реального часу. Він забезпечує стандартизований інтерфейс програмування, що переноситься на багато операційних систем реального часу, і, отже, дає змогу використовувати програмні шаблони, проміжне програмне забезпечення, бібліотеки та інші компоненти, які можуть працювати в системах реального часу.
- *CMSIS-Pack*: описує, використовуючи файл опису пакета на основі XML під назвою «PDSC», відповідні для користувача та пристрою частини колекції файлів (а саме «пакет програмного забезпечення»), який включає джерело, заголовок, файли бібліотеки, документацію, алгоритми програмування Flash, шаблони вихідного коду та приклади проектів. Інструменти розробки та веб-інфраструктури використовують файл PDSC для зчитування параметрів пристрою, програмних компонентів та конфігурацій плат оцінки.
- *CMSIS-SVD*: опис системи перегляду *System View Description* (SVD) для периферійних пристроїв. Описує периферійні пристрої пристрою у файлі XML і може використовуватися для створення інформації про периферію в налагоджувачах або файлах заголовків з периферійними регістрами та визначеннями переривань.
- *CMSIS-DAP*: порт доступу для налагодження Debug Access Port (DAP). Стандартизоване програмне забезпечення для налагоджувального

блоку, який підключається до порту доступу до налагодження CoreSight. CMSIS-DAP поширюється як окремий пакет і добре підходить для інтеграції на платах розробки.

CMSIS розвивається, але підтримка всіх компонентів від ST все ще є недостатньою. Офіційний ST HAL – це основний спосіб розробки додатків для платформи STM32, який має багато відмінностей між мікроконтролерами різних сімейств. Більше того, цілком зрозуміло, що головна мета постачальників мікроконтролерів — утримати своїх клієнтів і уникнути їх міграції на інші платформи MCU (навіть якщо вони засновані на тому ж ядрі ARM Cortex). Таким чином, ще далеко від створення універсального коду, який може працювати на всіх мікроконтролерах на базі ARM.

1.2 Загальна інформація про мікроконтролери STM3

STM32 — це широкий спектр мікроконтролерів, розділених на дев’ять підсімейств, кожне з яких має свої особливості. STM розпочав цих сімейств в 2007 році, починаючи з серії STM32F1, яка продовжує розвиватися. Усі мікроконтролери STM32 мають ядро Cortex-M, а також деякі притаманні тільки STM характеристики (наприклад, прискорювач ART). Внутрішньо кожен мікроконтролер складається з ядра процесора, статичної оперативної пам’яті, флеш-пам’яті, інтерфейсу налагодження та різних інших периферійних пристроїв. Деякі мікроконтролери забезпечують додаткові типи пам’яті (EEPROM, CCM тощо). Є ціла лінійка пристроїв, орієнтованих на програми з низьким енергоспоживанням.

Платформа STM32 має кілька суттєвих *переваг* для розробників вбудованих систем:

- Є кілька доступних інструментів для розробки програм, як платних, так і безкоштовних для мікроконтролерів на основі Cortex-M.
- Безкоштовний набір інструментів на основі ARM: завдяки поширенню процесорів на базі ARM можна працювати з безкоштовними засобами розробки, що важливо для початківця.
- Повторне використання кодів: STM32 – це досить великий перелік типів мікроконтролерів, які базуються на спільному знаменнику: їх основна платформа – центральний процесор. Це, наприклад, гарантує, що коди, створені при роботі з CPU STM32Fx, наприклад, можна легко застосувати до інших пристроїв з того ж сімейства або ж від інших виробників.
- Сумісність між корпусами та по виводам: більшість мікроконтроллерів STM32 розроблені таким чином, щоб бути сумісними за монтажем. Особливо це стосується корпусів LQFP64-100, і це великий плюс. Це дозволяє перейти до іншого сімейства, якщо виявиться, що воно більше відповідає вимогам майже не змінюючи дизайн плати.
- Стійкість до напруги 5 В: виводи деяких мікроконтролерів STM32 витримують 5 В. Це означає, що можна взаємодіяти з іншими пристроями, які не забезпечують введення-виведення 3,3 В, без використання узгоджувачів рівня.
- Невелика ціна за 32-розрядну версію: STM32F0 є правильним вибором, якщо потрібно перейти з 8/16-розрядних мікроконтроллерів на потужну та цілісну платформу, зберігаючи при цьому порівнянну низьку ціну.
- Вбудований завантажувач: мікроконтролери STM32 постачаються з інтегрованим завантажувачем, який дозволяє перепрограмувати внутрішню флеш-пам'ять за допомогою деяких комунікаційних периферійних пристроїв (USART, I^2C тощо).

Недоліки платформи STM32:

- Для новачків процес навчання розробці додатків STM32 може бути достатньо складним. Історично, ST-документація не була найкращою для недосвідчених людей, оскільки була надто заплутаною і не мала чітких прикладів.
- Фрагментована та розсіяна документація: ST активно працює над удосконаленням своєї офіційної документації для платформи STM32. Можна знайти багато дійсно величезних таблиць на веб-сайті ST, але все ще бракує хорошої документації, особливо для його HAL. А цього може бути недостатньо, щоб почати програмування з цією платформою, особливо якщо ви новачок в екосистемі STM32 і світі Cortex-M.
- Помилки та недоробки HAL: на жаль, офіційний HAL від ST містить помилки, і деякі з них дійсно серйозні, що призводять до плутанини у новачків. ST активно працює над виправленням помилок HAL, але ще далеко до «стабільного релізу». Більше того, життєвий цикл випуску програмного забезпечення занадто довгий і не відповідає часу: виправлення помилок випускаються через кілька місяців, і іноді виправлення викликає більше проблем, ніж сам помилковий код.

Контрольні запитання

1. Яке призначення та склад CMSIS (Cortex Microcontroller Software Interface Standard)?
2. Для чого використовується SysTick?
3. Які переваги та недоліки має платформа STM32

Лекція 2 Інструменти проектування пристроїв на STM32

2.1 Плати розробника STM32

Існує велике різноманіття плат розробки на основі мікроконтролерів STM32. Вони відрізняються типом вибраного мікроконтролера, а відповідно функціоналом, складністю та ціною. Для початкового ознайомлення, вивчення та здобуття навичок програмування 32-х бітних мікроконтролерів вибрано STM32F103C8T6. Сімейство ліній продуктивності середньої щільності STM32F103xx включає в себе високопродуктивне 32-розрядне ядро ARM Cortex-M3, що працює на частоті 72 МГц та високошвидкісну вбудовану пам'ять, а також широкий спектр розширених вводів-виводів і периферійних пристроїв.

Мікроконтролери STM32F103C8T6 мають наступні характеристики:

- 68 Кбайт флеш-пам'ять і 20 Кбайт SRAM
- два 12-розрядних АЦП,
- три 16-розрядних таймера загального призначення плюс
- один вдосконалений таймер ШІМ
- два I2C
- два SPI,
- три USART
- USB
- CAN.

Пристрої працюють від джерела живлення від 2,0 до 3,6 В. Діапазон робочих температур від –40 до +85 °C. Повний набір режимів енергозбереження дозволяє розробляти програми з низьким енергоспоживанням.

Завдяки наведеним характеристикам сімейство лінійних мікроконтролерів середньої щільності STM32F103xx підходить для широкого кола застосувань, таких як приводи двигунів, керування додатками, медичне та портативне обладнання, ПК та ігрові периферійні пристрої, платформи GPS, промислові програми, ПЛК, інвертори, принтери, сканери, системи сигналізації, відеодомофони та кліматичні системи.

На основі мікроконтролерів STM32F103C8T6 побудовано однойменну плату розробника ARM STM32 STM32F103C8T6 та плату STM32 Smart V2. Зовнішній вигляд плати STM32 Smart V2 наведено на Рисунку 2.1.

Рисунок 2.1. Плата STM32 Smart V2

На платі є всі необхідні елементи для початку роботи з даними мікроконтролером:

- 34 порта GPIO, розведених на контактні майданчики
- мікросхема EEPROM пам'яті на 4 кбіт AT24C04
- два кварци - 8 МГц для тактування ядра і 32768 Гц для тактування RTC

- стабілізатор напруги 3,3 В для забезпечення можливості живлення плат від 5 В
- роз'єм для підключення SWD програматора або JTAG-налагоджувача
- порт Mini-USB (з'єднаний з апаратним USB інтерфейсом)
- кнопка перезавантаження Reset S1
- користувальницька кнопка S2 (вивід PA0)
- два світлодіоди (один з них - індикатор подачі живлення, другий - підключений до порту PC13)
- роз’єм SPI для підключення OLED дисплея

Для програмування мікроконтролера використовується програматор ST-Link або JTAG-програматор. Є перемикач (перемичка) режиму запуску завантажувача (bootloader).

Також варто зауважити, що плата STM32 Smart V2 є однією з найдешевших і доступніших на ринку проміж плат розробки на основі STM32. Таким чином, STM32 Smart V2 є хорошим вибором для початку роботи з мікроконтролерами серії STM32.

2.2 Засоби програмування мікроконтролерів STM32

Перш ніж почати розробку додатків для платформи STM32, необхідно визначитися з набором інструментів для програмування, які дозволяють:

- записувати код і переміщуватись всередині вихідних файлів створюваної програми;
- здійснювати навігацію всередині коду програми, що дозволяє перевіряти змінні, визначення функцій/оголошення тощо;

- компілювати вихідний код за допомогою кросплатформного компілятора;
- завантажувати та налагоджувати програму на цільовій платі розробки (або на платі, яка розробляється).

Для виконання цих дій потрібно:

- Середовище розробки *Integrated Development Environment* (IDE) з інтегрованим редактором джерела та навігатором;
- кросплатформний компілятор, здатний компілювати вихідний код для платформи ARM Cortex-M;
- налагоджувач, який дозволяє виконувати покрокове налагодження програми на цільовій платі;
- інструмент, який дозволяє взаємодіяти з інтегрованим апаратним налагоджувачем (інтерфейс ST-LINK) або спеціальним програматором (наприклад, адаптером JTAG).

Існує кілька інструментів для розробки програмного забезпечення сімейства STM32 Cortex-M, як безкоштовних, так і комерційних. IAR і Keil є одними з найбільш використовуваних комерційних інструментів для мікроконтролерів Cortex-M. Вони є повним рішенням для розробки додатків для платформи STM32, але, будучи комерційними продуктами, вони мають достатньо високу ціну, яка може бути неприйнятною для невеликих компаній або студентів (вони можуть коштувати більше 5000 доларів США в залежності від функціоналу).

STM32CubeIDE від ST— це безкоштовне середовище розробки для платформи STM32. IDE засновано на Eclipse і GCC. Воно поставляється з

усіма необхідними інструментами, попередньо встановленими та налаштованими.

2.2.1 Програма початкових налаштувань STM32CubeMX

STM32CubeMX — це базовий інструмент для створення коду ініціалізації, особливо корисним є для новачків у програмуванні на платформі STM32. Дане програмне забезпечення вільно поширюється і є частиною ініціативи STCube, метою якої є надання розробникам повного набору інструментів і бібліотек для прискорення процесу розробки.

STM32CubeMX використовується для налаштування обраного для проекту мікроконтролера або плати розробки. Він використовується як для вибору правильних апаратних підключень, так і для створення коду, необхідного для налаштування ST HAL.

STM32CubeMX — це програма, орієнтована на мікроконтролер, яка дозволяє вибрати та налаштувати основні його параметри, а саме:

- Сімейство мікроконтролера STM32 (F0, F1 і так далі).
- Тип корпусу вибраного пристрою (LQFP48, BGA144 тощо).
- Апаратні периферійні пристрої, необхідні для проекту (USART, SPI тощо).
- Відображення розміщення периферійних пристроїв на виводах мікроконтролера.
- Загальні конфігурації MCU (наприклад, годинник, керування живленням, контролер NVIC тощо).
- Приклади реалізації коду для периферійних пристроїв.

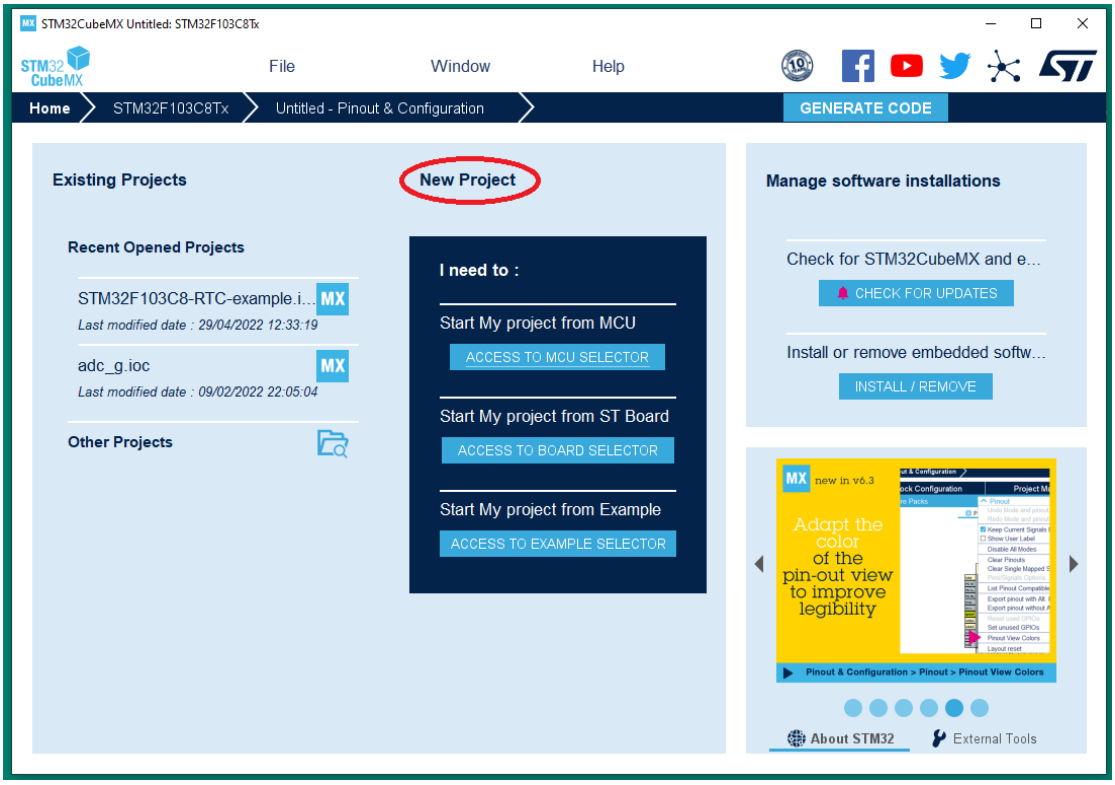

Рисунок 2.2 Головне вікно програми STM32CubeMX

На додаток до функцій, пов'язаних з апаратним забезпеченням, STM32CubeMX також може працювати з наступними аспектами програмного забезпечення:

- Керування ST HAL для вибраного сімейства MCU (CubeF0, CubeF1 тощо).
- Додаткові функції бібліотеки програмного забезпечення, необхідні для проекту (бібліотека FatFs, FreeRTOS тощо).
- Середовище розробки, яке буде використовуватись для створення програми (IAR, TrueSTUDIO тощо).

Після запуску STM32CubeMX з'являється головне вікно програми з основним меню, закладками та кнопками та панелями (Рисунок 2.2).

Для створення **нового проекту** необхідно на панелі “New Project” вибрати початок роботи або з серії потрібного мікроконтролера, або з потрібної плати розробки, або з готового прикладу проекту. Після вибору бажаного початку роботи відкриється вікно нового проекту (Рисунок 2.3).

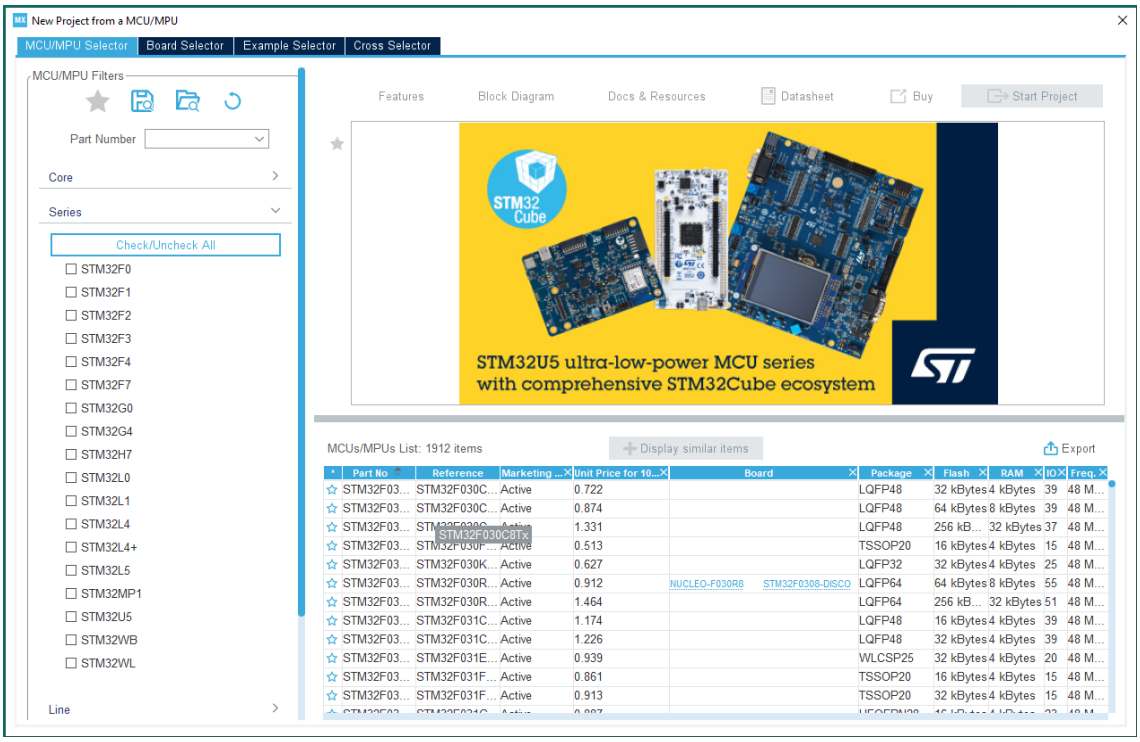

Рисунок 2.3 Вікно нового проекту з вибором типу мікроконтролера

Вікно нового проекту містить чотири закладки: MCU/MPU Selector, *Board Selector*, *Example Selector* та *Cross Selector*.

Перша закладка дозволяє вибрати мікроконтролер з усього портфоліо STM32. Використовуючи поле зі списком *Series*, можна відфільтрувати всі MCU, що належать до даної серії. Поле зі списком *Lines* дозволяє додатково фільтрувати MCU, що належать до підсімейства (рядок значення тощо). Поле зі списком *Package* дозволяє вибрати всі MCU, які мають потрібний пакет. Поле *Peripheral* дозволяє вибрати мікроконтролер, який підтримує потрібну периферію.

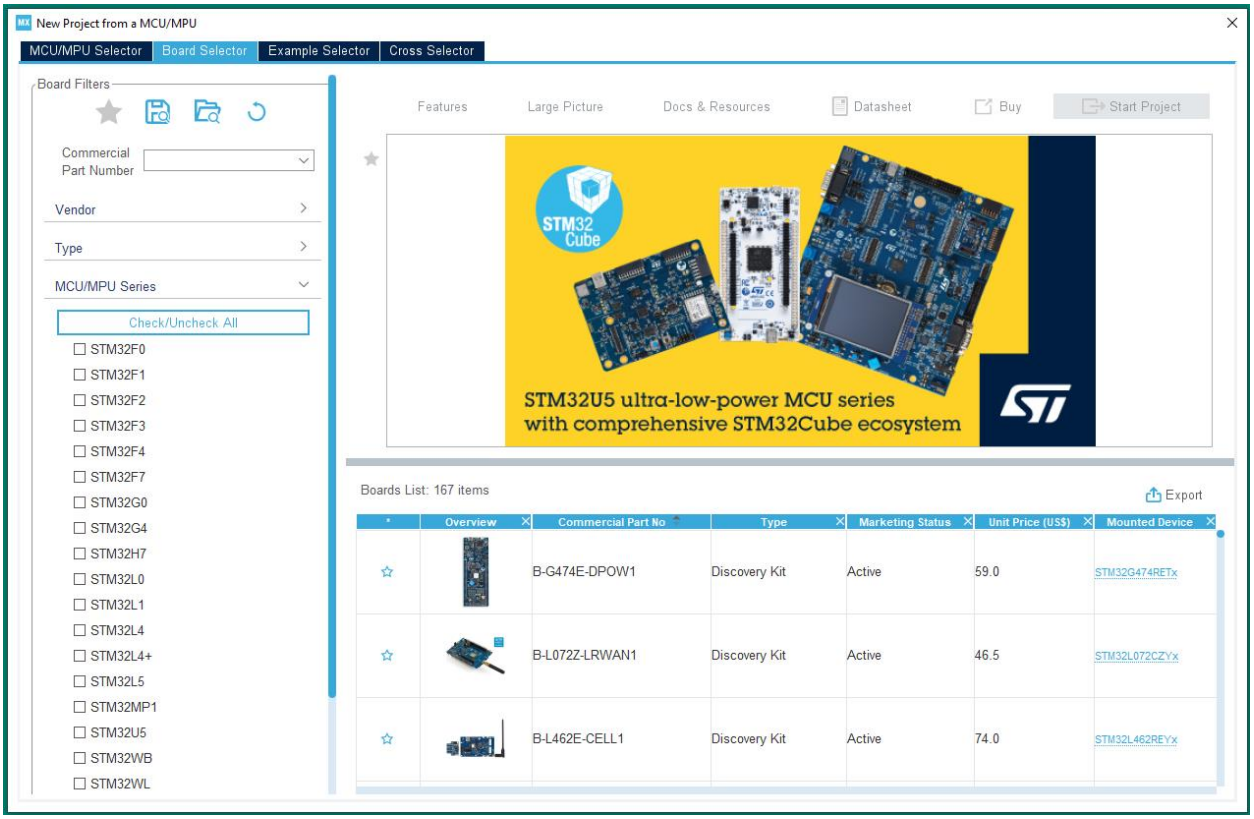

Рисунок 2.4 Вибір плати розробки STM32

Вкладка *Board Selector* дозволяє фільтрувати серед усіх офіційних плат розробки ST (див. Рисунок 2.4). На вибір є три типи плат розробника: *Nucleo*, *Discovery* та *EvalBoard*, які є найповнішими (і найдорожчими) платами розробників для експериментів із STM32 MCU.

Вибір плати з ініціалізацією всіх периферійних пристроїв у режимі за замовчуванням автоматично встановлює як розміщення виводів, так і режими за замовчуванням для периферійних пристроїв, доступних на платі. Це означає, що STM32CubeMX генерує код ініціалізації C для всіх периферійних пристроїв, доступних на платі, а не лише для тих, що стосуються програми користувача.

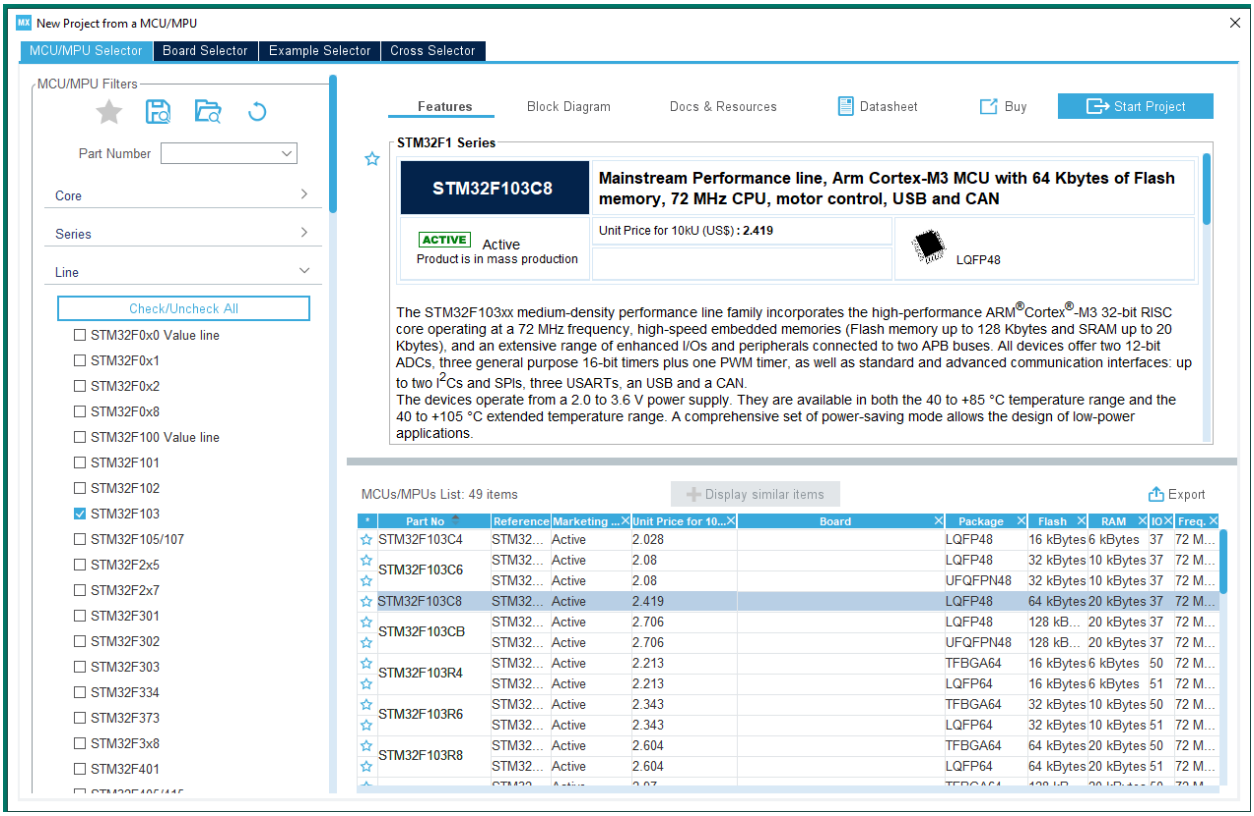

Рисунок 2.5 Результат вибору потрібного MCU в STM32CubeMX

Після того, як потрібний мікроконтролер вибрано у вікні нового проекту з'явиться панель з короткими характеристиками мікроконтролера та меню вибору додаткової інформації по ньому, як показано на Рисунку 2.5 Можна подивитися структуру мікроконтролера, отримати додаткову літературу, необхідну для розробки, та довідатись про ціну. Для продовження роботи необхідно натиснути кнопку *Start Project*. Після завантаження необхідних бібліотек відкриється вікно проекту, як показано на Рисунку 2.6. Вікно проекту містить декілька закладок та панелей. На закладці *Pinout & Configuration* розміщується панель *Categories*. Основні налаштування мікроконтролера розподілені в наступних категоріях: *System Core*, *Analog*, *Timers*, *Connectivity*, *Computing*, *MiddleWare*. Детально вказані категорії буде

розглянуто далі у конкретних прикладах ініціалізації відповідних складових та периферії мікроконтролера.

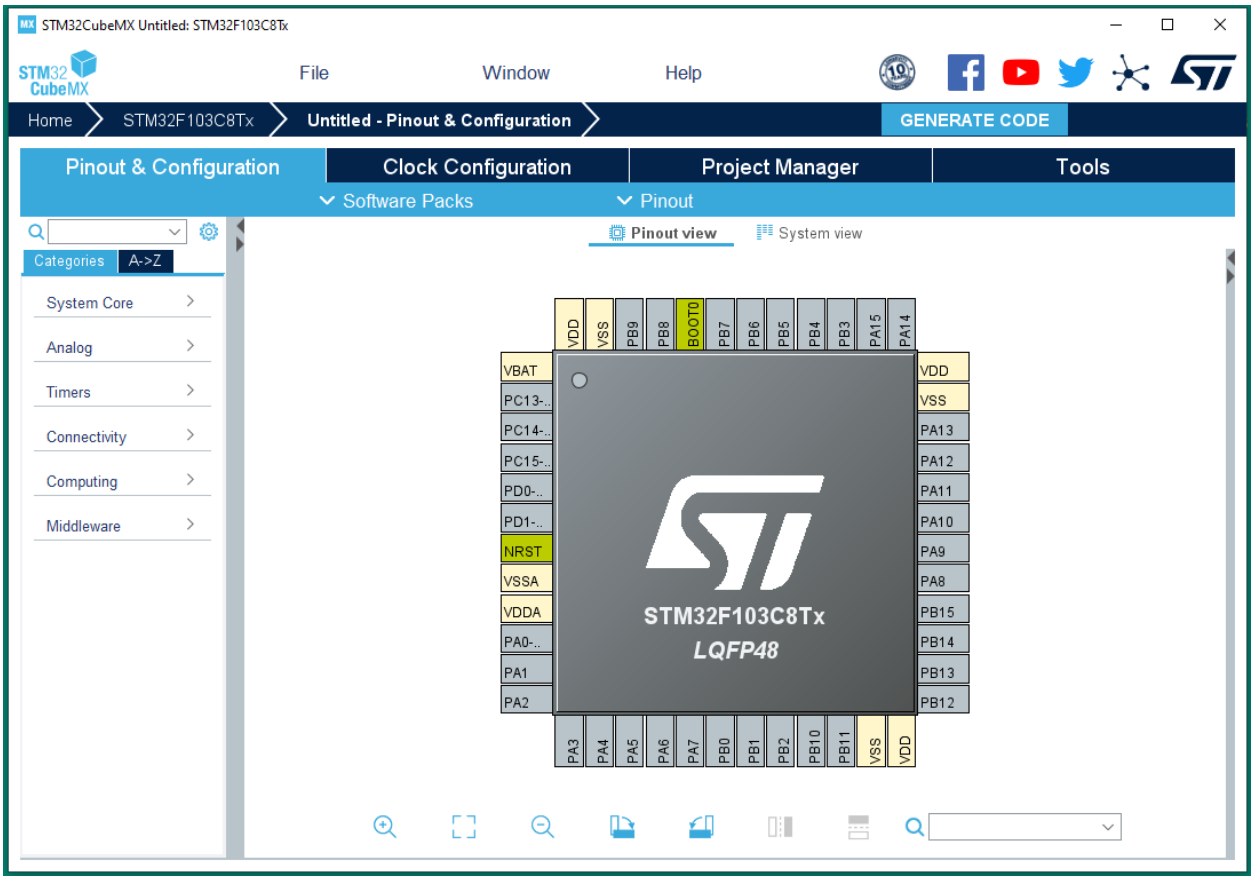

Рисунок 2.6 Вигляд закладки *Pinout & Configuration*

Закладка *Clock Configuration* містить інтерактивну функціональну схему кіл осциляції, годинника реального часу, дільників та множників частоти. Закладка використовується для вибору типу осциляторів та для налаштування тактових сигналів периферії.

Закладка *Project Manager* потрібна для вибору назви проекту, шляху його зберігання, вибору середовища розробки, під яке буде згенеровано програмний код, та параметрів налаштування програмних інструментів при генерації коду.

Закладка *Tools* містить інструменти для оцінки споживання енергії мікроконтролером в залежності від тактової частоти системної шини та підключених периферійних пристроїв.

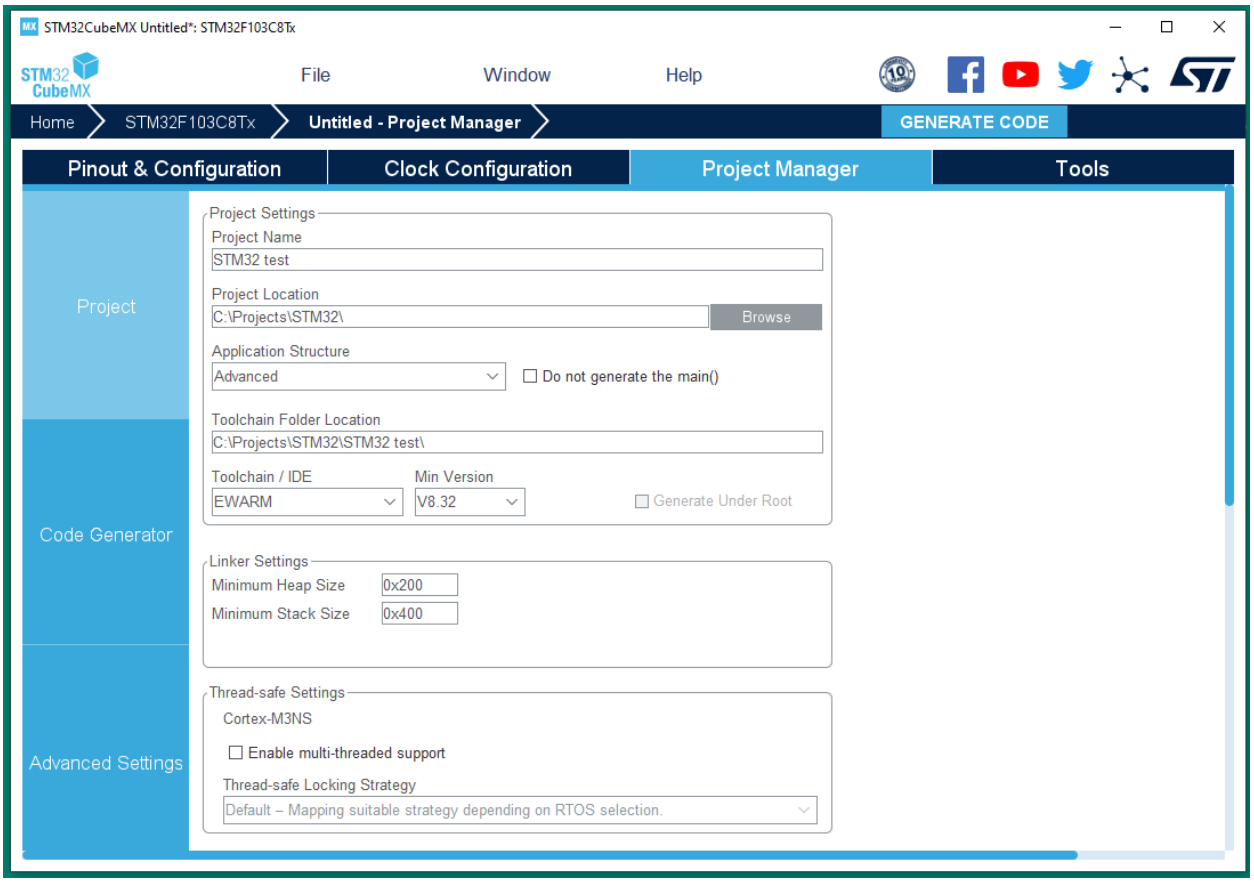

Рисунок 2.7 Вигляд закладки *Project Manager*

Після задання потрібних налаштувань мікроконтролера, вибору назви проекту, шляху його збереження та середовища розробки EWARM (програми будуть створюватись у середовищі IAR Embedded Workbench) можна створити початковий код програми. Для цього потрібно натиснути кнопку *Generate Code*. Вигляд вікна закладки *Project Manager* показано на Рисунку 2.7.

2.2.2 Інтегроване середовище розробки IAR Embedded Workbench

IAR Embedded Workbench (IAR EW) - багатофункціональне середовище розробки додатків мовами C, C++ та асемблері для цілого ряду мікроконтролерів від різних виробників.

Основні переваги пакету – можливість гнучкого налаштування для розробників будь-якого рівня підготовки і відмінна оптимізація генерованого коду. Крім цього реалізована підтримка операційних систем реального часу та JTAG-адаптерів і завантажувачів сторонніх компаній. В даний час IAR Embedded Workbench підтримує роботу з 8-, 16-, 32-розрядними мікроконтролерами від Atmel, ARM, Texas Instruments, STMicroelectronics та багато інших. Для кожної платформи існує своє середовище розробки, зокрема, мікроконтролерам STM32 відповідає версія пакету IAR Embedded Workbench for ARM.

Програмне середовище включає:

1. C/C++ високоякісний компілятор з повною підтримкою ANSI C.

2. Транслятор асемблера, що включає макроассемблер для програм реального часу і препроцесор для C/C++компілятора.

3. Компонувальник, який підтримує понад тридцять різних вихідних форматів для спільного використання з внутрішньосхемними емуляторами.

4. Текстовий редактор, який налаштований на синтаксис мови C, має зручний інтерфейс користувача, автоматичне виділення помилок програмного коду, інструментальну панель з великою кількістю налаштувань, підсвічування блоків, а також зручну навігацію за іменами підпрограм, макросів і змінних.

5. Симулятор та налагоджувач у кодах С та асемблера. Налагоджувач дозволяє переглядати області EEPROM, DATA, CODE, а також регістри введення/виводу, встановлювати точки зупинки та апаратні прапори, обробляти переривання з передбаченням. Крім цього передбачений контроль стека та будь-яких локальних змінних, режим покрокового виконання програми.

6. Менеджер проектів, що полегшує контроль та управління робочими модулями.

Розміщення компонентів в основному вікні програми IAR EW наведено на Рисунку 2.8. Докладний опис компонентів програми наведено в [f].

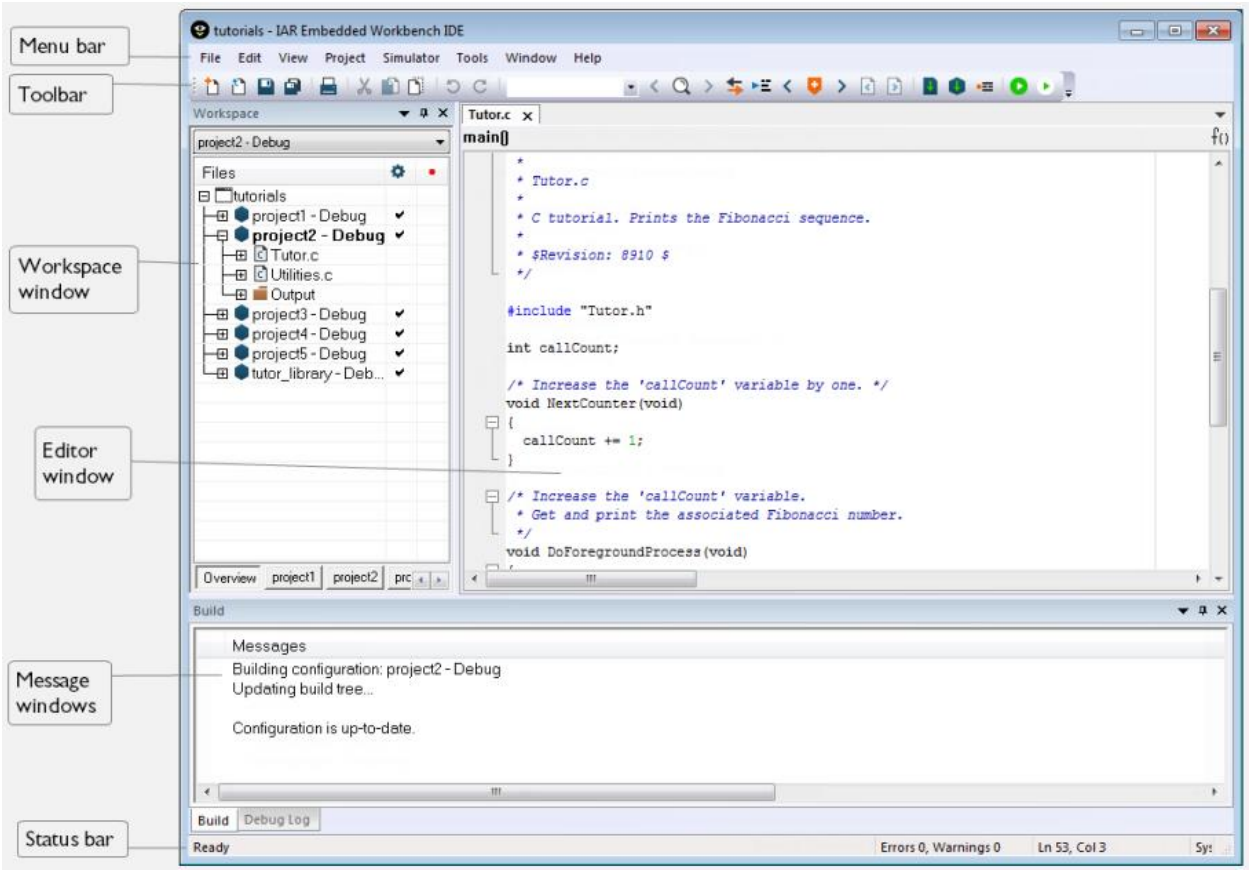

Рисунок 2.8. Складові вікна проекту програми IAR EW

2.3 Робота з проектом STM32CubeMX в IAR Embedded Workbench

Програма ініціалізації STM32CubeMX створює повноцінній проект в середовищі IAR EW, який містить дерево проекту, необхідні бібліотеки та файли коду. Зображення вікна IAR EW проекту, згенерованого в STM32CubeMX наведено на Рисунку 2.9.

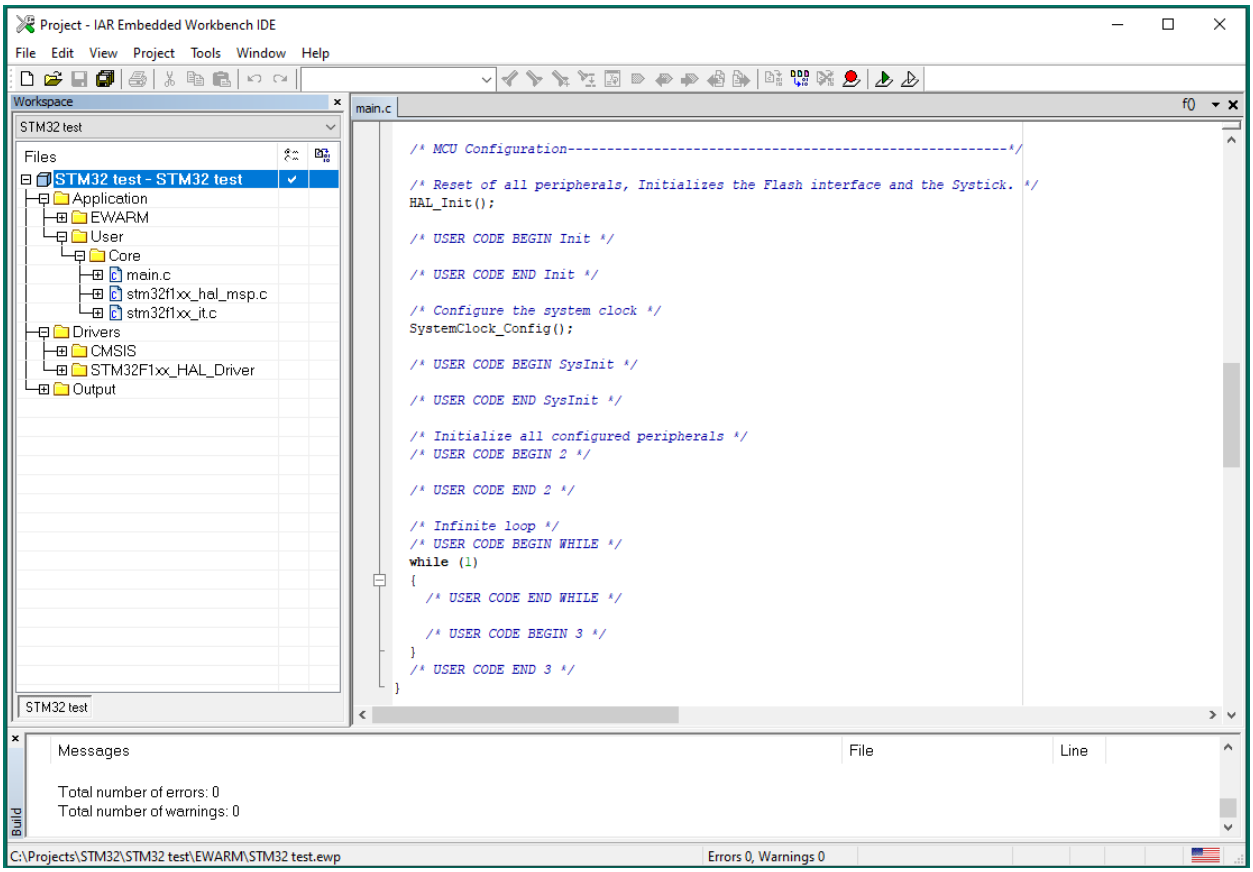

Рисунок 2.9 Вікно проекту, згенерованого в STM32CubeMX та відкритого в IAR EW

Вікно проекту (Workspace window) відображає в папках основні файли та бібліотеки, використані для створення коду програми. В проекті є три основні папки *Application*, *Drivers* та *Output*. В папці *Drivers* є папка *STM32F1xx_HAL_Driver*, в якій знаходяться файли бібліотек функцій для роботи з периферією мікроконтролера. Ці функції можна використовувати в

проекті. В папці *Application* є папка *Core*, в якій знаходяться файли призначені для користувача: *main.c*, *stm32f1xx_hal_msp.c*, *stm32f1xx_it.c*. В цих файлах користувач може додавати свій код, причому обов'язково в межах захищених від видалення областей від `/* USER CODE BEGIN … */` до `/* USER CODE END … */`. У іншому випадку після чергової генерації коду програмою STM32CubeMX користувацький код буде видалено.

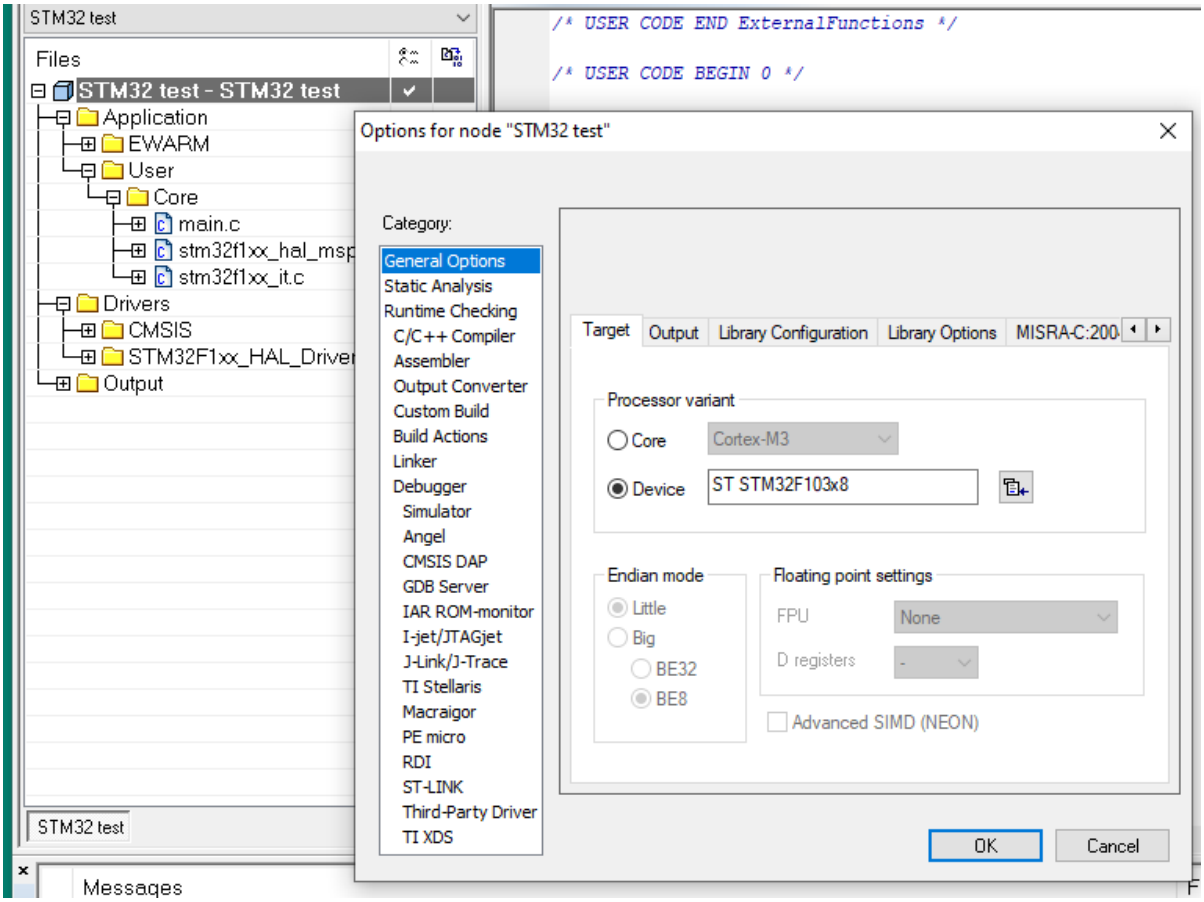

Рисунок 2.10 Вибір типу мікроконтролера

Для уникнення помилок при створенні програми в проекті, згенерованому в STM32CubeMX, необхідно в опціях проекту вказати тип мікроконтролера. Зображення вікна для налаштувань опцій проекту і вибором типу мікроконтролера наведено на рисунку 2.10.

Призначення файлів користувача наступне:

- *main.c* - для розміщення основного коду програми користувача;
- *stm32f1xx_hal_msp.c* – для включення/виключення системних компонентів мікроконтролера;
- *stm32f1xx_it.c* – для розміщення коду роботи з перериваннями.

Для завантаження коду програми в мікроконтролер використовується програматор ST-Link V2, який підключається до плати STM32 Smart V2 та до порту USB комп'ютера. Зовнішній вигляд одного з варіантів програматора ST-Link V2 наведено на Рисунку 2.11.

Рисунок 2.11 Програматор ST-Link V2

Контрольні запитання

1. Наведіть приклади застосування мікроконтролерів сімейства STM32F103xx
2. Для чого використовується STM32CubeMX?
3. Яке призначення файлів `main.c`, `stm32f1xx_hal_msp.c`, `stm32f1xx_it.c` ?

Лекція 3 Керування портами вводу-виводу

Програма ініціалізації STM32CubeMX пропонує стандартній рівень апаратної абстракції (HAL) для мікроконтролерів STM32. Крім того, компанія ST офіційно підтримуватиме його в майбутньому. HAL значно спрощує написання коду, перенесення його між підсімействами STM32 (F0, F1 тощо), зменшуючи зусилля, необхідні для адаптації існуючої програми до іншого MCU (без хорошого рівня абстракції, pin-to-pin сумісність контактів – це лише перевага з точки зору маркетингу). З цієї та кількох інших причин далі буде розглянуто бібліотеку HAL, яку встановлює STM32CubeMX.

Для початку необхідно розглянути як периферійні пристрої STM32 відображаються на адресній карті мікроконтролера та як вони представлені в бібліотеці HAL.

3.1 Периферійні пристрої STM32 та обробники HAL

Кожен периферійний пристрій STM32 з’єднано з ядром MCU кількома шинами, як показано на Рисунку 3.1.

- Системна шина (*System bus*) з'єднує системну шину ядра Cortex-M з BusMatrix, яка керує арбітражем між ядром і DMA. І ядро, і DMA діють як мастери.
- Шина DMA (*DMA bus*) з'єднує головний інтерфейс *Advanced High-Performance Bus* (AHB) DMA з BusMatrix, який керує доступом CPU і DMA до SRAM, флеш-пам'яті та периферійних пристроїв.
- BusMatrix керує арбітражем доступу між основною системною шиною та головною шиною DMA. В арбітражі використовується карусельний алгоритм. BusMatrix складається з двох головних (CPU, DMA) і чотирьох підлеглих (інтерфейс флеш-пам'яті, SRAM, AHB1 з мостом

AHB до *Advanced Peripheral Bus* (APB) і AHB2). Периферійні пристрої AHB підключаються до системної шини через BusMatrix, щоб забезпечити доступ DMA.

- Міст AHB – APB (*AHB to APB bridge*) забезпечує повне синхронне з'єднання між AHB та шиною APB, до якої підключено більшість периферійних пристроїв.

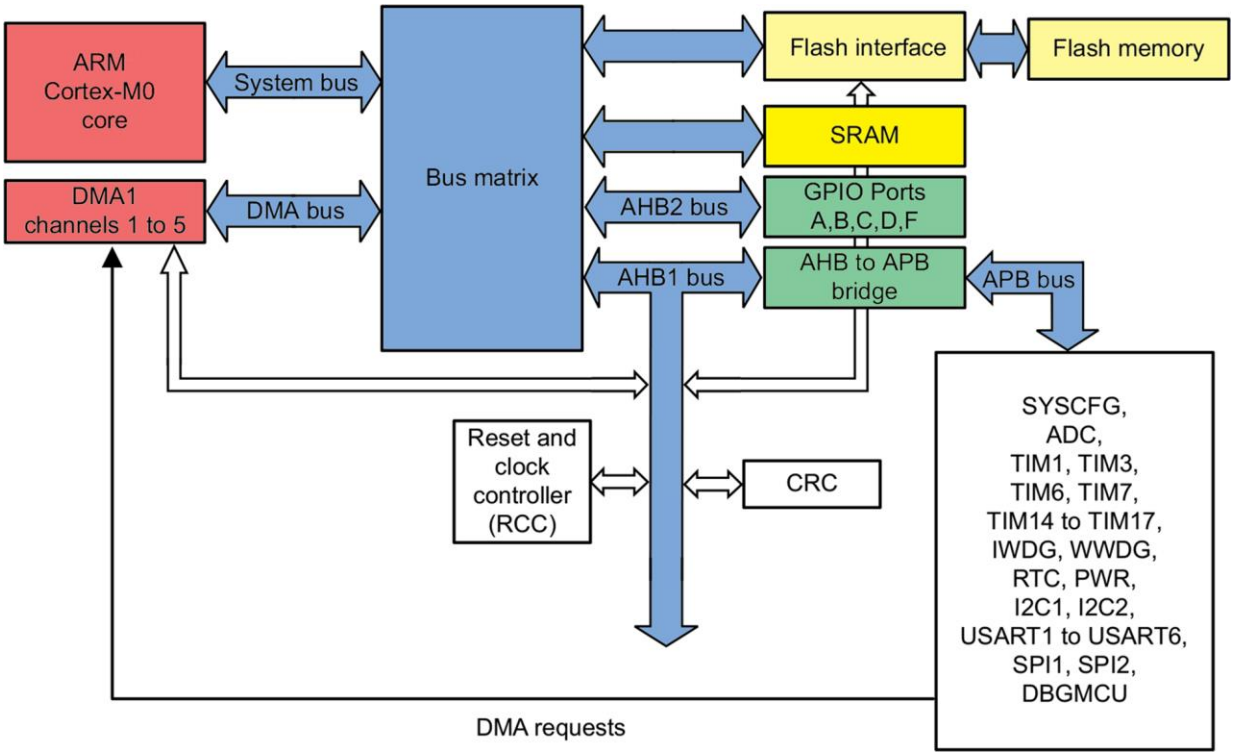

Рисунок 3.1 Архітектура шини мікроконтролера STM32F030

Варто зауважити що кожна з цих шин підключена до різних джерел тактування, які визначають максимальну швидкість для периферійного пристрою, підключеного до цієї шини. Периферійні пристрої відображаються на певну область адресного простору 4 ГБ, починаючи з 0x4000 0000 і тривалістю до 0x5FFF FFFF. Цей регіон далі розділений на кілька субрегіонів, кожен з яких відображено на певному периферійному пристрої, як показано на Рисунку 3.2.

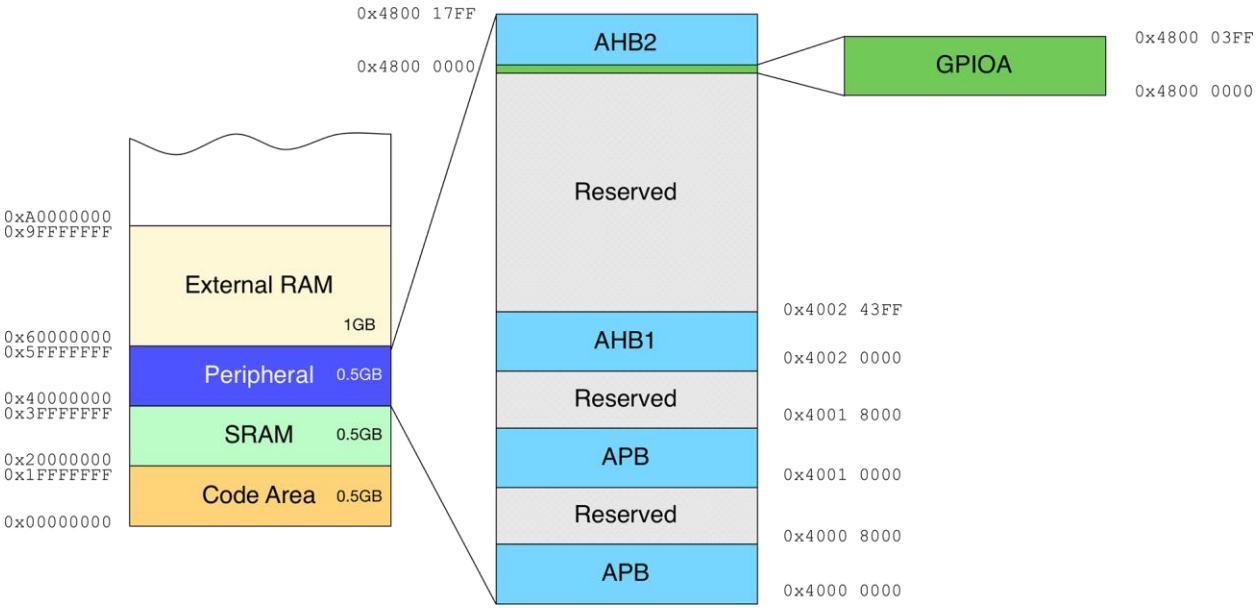

Рисунок 3.2 Карта пам'яті периферійних областей для мікроконтролера STM32F030

Спосіб організації карти пам'яті, а отже, і те, як відображаються периферійні пристрої, є специфічними для даного мікроконтролера STM32. Наприклад, у мікроконтролері STM32F030 шина AHB2 відображається в діапазоні від 0x4800 0000 до 0x4800 17FF. Це означає, що область має ширину 6144 байта. Цей регіон також поділений на кілька субрегіонів, кожен з яких відповідає певній периферії. У свою чергу, те, як цей відображений пам’яттю простір організовано, залежить від конкретного периферійного пристрою.

Важливо ще раз уточнити, що кожне сімейство STM32 (F0, F1 і т. д.) і кожен член даного сімейства (STM32F030, STM32F1 і т. д.) надає свою підмножину периферійних пристроїв, які відображаються на певні адреси. Крім того, спосіб реалізації периферійних пристроїв різниться між серією STM32.

Однією з ролей HAL є абстрагування від конкретного периферійного відображення. Це робиться шляхом визначення кількох обробників для

кожного периферійного пристрою. Обробник — це не що інше, як структура C, посилання якої використовуються для вказівки на реальну периферійну адресу. Давайте подивимося на одну з них.

3.2 Конфігурація виводів портів вводу-виводу

Функціональна схема блоку пристроїв GPIO для одного виводу наведена на Рисунку 3.3.

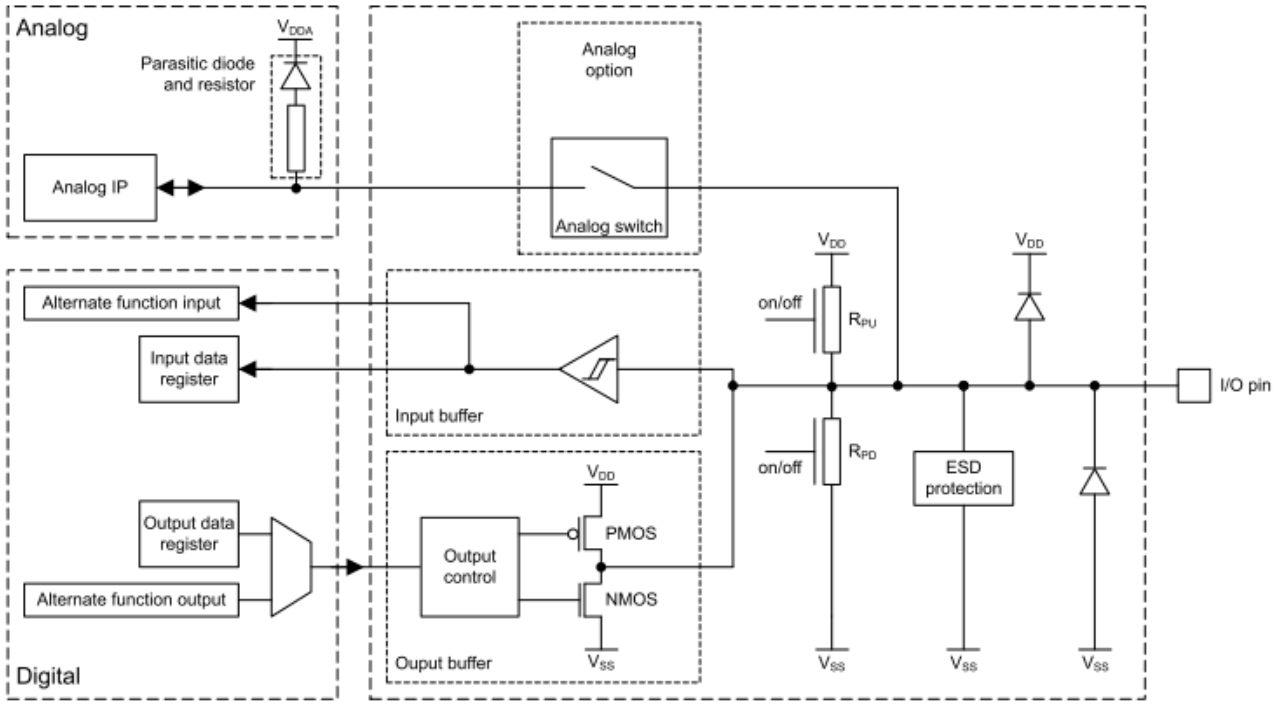

Рисунок 3.3 Функціональна схема блоку пристроїв виводу GPIO

Конфігурація виводів порту вводу-виводу на вхід. Коли вивод I/O пристрою STM32 налаштований як вхід, необхідно вибрати один з трьох варіантів:

- Вхід із внутрішньою прив’язкою (підтягуванням) до напруги живлення. Резистори підтягування використовуються в пристроях STM32 для забезпечення чітко визначеного логічного рівня 1 у випадку плаваючого

вхідного сигналу. Залежно від вимог до застосування, замість нього можна використовувати зовнішнє підтягування.

- Вхід із внутрішньою прив’язкою (опусканням) до нуля. Резистори опускання використовуються в пристроях STM32 для забезпечення чітко визначеного логічного рівня у випадку плаваючого вхідного сигналу. Залежно від вимог програми замість нього можна використовувати зовнішній резистор.
- Плаваючий вхід. Рівень сигналу слідує за зовнішнім сигналом. Коли зовнішній сигнал відсутній, тригер Шмітта випадковим чином перемикається між логічними рівнями, викликаними зовнішнім шумом.

Запрограмований як вхідний, порт вводу-виводу має такі характеристики:

- Вихідний буфер вимкнено
- Активовано вхід тригера Шмітта
- Резистори підтягування або опускання активуються залежно від значення в регістрі GPIOx_PUPDR
- Дані, наявні на контакті вводу/виводу, відбираються в регістр вхідних даних на кожному такті AHB
- Стан введення-виведення отримується шляхом зчитування регістру вхідних даних GPIOx_IDR

Конфігурація виводів порту вводу-виводу на вихід. Коли вивід I/O пристрою STM32 налаштований як вихід, необхідно вибрати один з двох варіантів:

- Push-pull вихідний режим:

Двохактний вихід фактично використовує два транзистори: один PMOS і один NMOS. Кожен транзистор увімкнено, щоб підвести вихід до відповідного рівня:

- Верхній транзистор (PMOS) увімкнено, коли вихід має перейти в стан HIGH
- Нижній транзистор (NMOS) увімкнено, коли вихід має перейти в стан LOW

Управління двома транзисторами здійснюється через регістр типу виходу порту GPIO (GPIOx_OTYPER). Запис відповідного біта вихідного регістра (GPIOx_ODR) в 0 активує транзистор NMOS, щоб примусово заземлити контакт введення-виведення. Запис відповідного біта вихідного регістра (GPIOx_ODR) в 1 активує транзистор PMOS, щоб примусово перевести контакт вводу/виводу до VDD.

- Режим виходу з відкритим стоком:

  У режимі виходу з відкритим стоком транзистор PMOS не використовується, і потрібен підтягуючий резистор.

  Коли вихід має бути високим, транзистор NMOS необхідно вимкнути, підтягуючи лінію високого рівня тільки за допомогою підтягуючого резистора. Цей підтягуючий резистор може бути внутрішнім із типовим значенням 40 кОм і активуватися через регістр підтягування / спаду порту GPIO (GPIOx_PUPDR).

Запрограмований як вихід, порт вводу-виводу має такі характеристики:

- Вихідний буфер може бути налаштований у режимі відкритого стоку або двотактного режиму
- Активовано вхід тригера Шмітта
- Внутрішні резистори підтягування та опускання активуються залежно від значення в регістрі GPIOx_PUPDR.

- Записане значення в регістр вихідних даних GPIOx_ODR встановлює стан контакту вводу/виводу
- Записані дані на GPIOx_ODR можна прочитати з регістра GPIOx_IDR, який оновлюється кожен такт AHB

Вихід з відкритим стоком часто використовується для керування пристроями, які працюють при іншій напрузі, ніж STM32. Режим відкритого стоку також використовується для керування одним або кількома пристроями I2C, коли потрібні спеціальні підтягуючі резистори.

Альтернативні функції виводів порту вводу-виводу. На деяких виводах STM32 GPIO користувач має можливість вибрати альтернативні функції входів/виходів. Кожен вивід мультиплексований з одним з шістнадцяти периферійними функціями, такими як інтерфейси зв'язку (SPI, UART, I2C, USB, CAN, LCD та інші), таймери, інтерфейс налагодження

та інші. Альтернативна функція вибраного контакту налаштовується через два регістри:

- GPIOx_AFRL (для контактів від 0 до 7)
- GPIOx_AFRH (для контактів 8-15)

Коли порт вводу-виводу запрограмований у альтернативному функціональному режимі:

- Вихідний буфер може бути налаштований у режимі відкритого стоку або двотактного режиму
- Вихідний буфер керується сигналами, що надходять від периферійного пристрою (увімкнення передавача та даних)
- Активовано вхід тригера Шмітта

- Активація резисторів підтягування та опускання залежить від значення в регістрі GPIOx_PUPDR

Дані, наявні на контакті вводу/виводу, відбираються в регістр вхідних даних на кожному такті AHB. Доступ для читання до регістру вхідних даних забезпечує стан введення/виводу.

Аналогова конфігурація виводів порту вводу-виводу. Кілька контактів STM32 GPIO можна налаштувати в аналоговому режимі, що дозволяє використовувати внутрішні периферійні пристрої АЦП, ЦАП, OPAMP і COMP. Для використання контакту GPIO в аналоговому режимі враховуються такі регістри:

- GPIOx_MODER для вибору режиму (вхід, вихід, альтернативний, аналоговий)
- GPIOx_ASCR для вибору необхідної функції АЦП, ЦАП, OPAMP або COMP

Якщо порт введення-виводу запрограмований в аналоговій конфігурації:

- Буфер виведення вимкнено
- Вхід тригера Шмітта деактивовано, забезпечуючи нульове споживання для кожного аналогового значення контакту вводу/виводу. На виході тригера Шмітта встановлюється постійне значення (0).
- Підтягуючий і опускаючий резистори відключені апаратно

Доступ на читання до регістру вхідних даних отримує значення 0. Сам аналоговий перемикач не замикається. Аналоговий перемикач замикається тільки тоді, коли аналоговий периферійний пристрій вибрано (або увімкнено) на даному контакті.

3.3 Налаштування виводів порту вводу-виводу

Різні типи мікроконтролерів STM32 мають різну кількість виводів загального призначення *General-Purpose Input /Output* (GPIO). Точна кількість виводів залежить від:

- Вибраного типу корпусу (LQFP48, BGA176 тощо).
- Сімейства мікроконтролерів (F0, F1 та ін.).
- Використання зовнішніх осциляторів для високочастотного та низькочастотного тактових генераторів.

GPIO – це спосіб зв'язку MCU із зовнішніми пристроями. Кожна плата використовує різну кількість вводів/виводів для керування зовнішніми периферійними пристроями (наприклад, світлодіодами) або для обміну даними через кілька типів периферійних пристроїв зв'язку (UART, USB, SPI тощо). Кожен раз, коли потрібно налаштувати периферійний пристрій, який використовує контакти MCU, потрібно налаштувати відповідні GPIO за допомогою модуля HAL_GPIO.

HAL розроблений таким чином, що дозволяє абстрагуватися від конкретного відображення периферійної пам'яті. HAL забезпечує загальний і більш зручний спосіб налаштування периферійного пристрою, не змушуючи програмістів детально налаштувати його регістри.

Для налаштування GPIO використовується функція `HAL_GPIO_Init(GPIO_TypeDef *GPIOx, GPIO_InitTypeDef *GPIO_Init)`. `GPIO_InitTypeDef` - це структура C, яка використовується для налаштування GPIO, і вона визначається таким чином:

```
typedef struct {
```

```
uint32_t Pin;
uint32_t Mode;
uint32_t Pull;
uint32_t Speed;
} GPIO_InitTypeDef;
```

Опис кожного поля структури:

- `Pin`: це номер (починаючи з 0) виводу, який потрібно налаштувати. Наприклад, для контакту PA5 він приймає значення GPIO_PIN_5.
- `Mode`: це режим роботи виводу, і він може приймати одне зі значень у таблиці 3.1.
- `Pull`: визначає наявність прив'язки виводу до землі або живлення, а саме `GPIO_NOPULL, GPIO_PULLUP, GPIO_PULLDOWN`.
- `Speed`: визначає швидкість наростання напруги на виводі, тобто тривалість фронту або спаду сигналу.

Таблиця 3.1 Доступний режим GPIO_InitTypeDef.Mode для GPIO

Pin Mode	**Опис режиму**
GPIO_MODE_INPUT	Режим входу без прив'язки
GPIO_MODE_OUTPUT_PP	Вихідний режим Push-Pull
GPIO_MODE_OUTPUT_OD	Вихід Режим відкритого стоку
GPIO_MODE_AF_PP	Режим Push-Pull для альтернативної функції виводу
GPIO_MODE_AF_OD	Режим відкритого стоку для альтернативної функції виводу

GPIO_MODE_AF_INPUT	Альтернативна функція виводу
GPIO_MODE_ANALOG	Аналоговий режим
GPIO_MODE_IT_RISING	Режим переривання по наростаючому сигналу
GPIO_MODE_IT_FALLING	Режим переривання по спадаючому сигналу
GPIO_MODE_IT_RISING_FALLING	Режим переривання по наростаючому або спадаючому сигналу
GPIO_MODE_EVT_RISING	Режим зовнішньої події по наростаючому сигналу
GPIO_MODE_EVT_FALLING	Режим зовнішньої події по спадаючому сигналу
GPIO_MODE_EVT_RISING_FALLING	Режим зовнішньої події по наростаючому або спадаючому сигналу

3.4 Програмування GPIO в STM32CubeMX та IAR EW

Плата STM32 Smart V2 має два компонента, пов’язаних з GPIO: кнопку, підключену до виводу PA0, та світлодіод, підключений до виводу PC13. Таким чином дуже просто створити простий приклад включення світлодіоду при натисканні кнопки та виключенні при її відпусканні. Конфігурацію виводів в STM32CubeMX виконують наступним чином. На закладці Pinout & Configuration на панелі Categories в категорії System Core зі списку потрібно вибрати GPIO. На Рисунку мікросхеми потрібно клікнути на вивід PC13 та

вибрати зі списку GPIO_Output. Аналогічно для виводу PA0 вибирається GPIO_Input (див. рисунок 3.4).

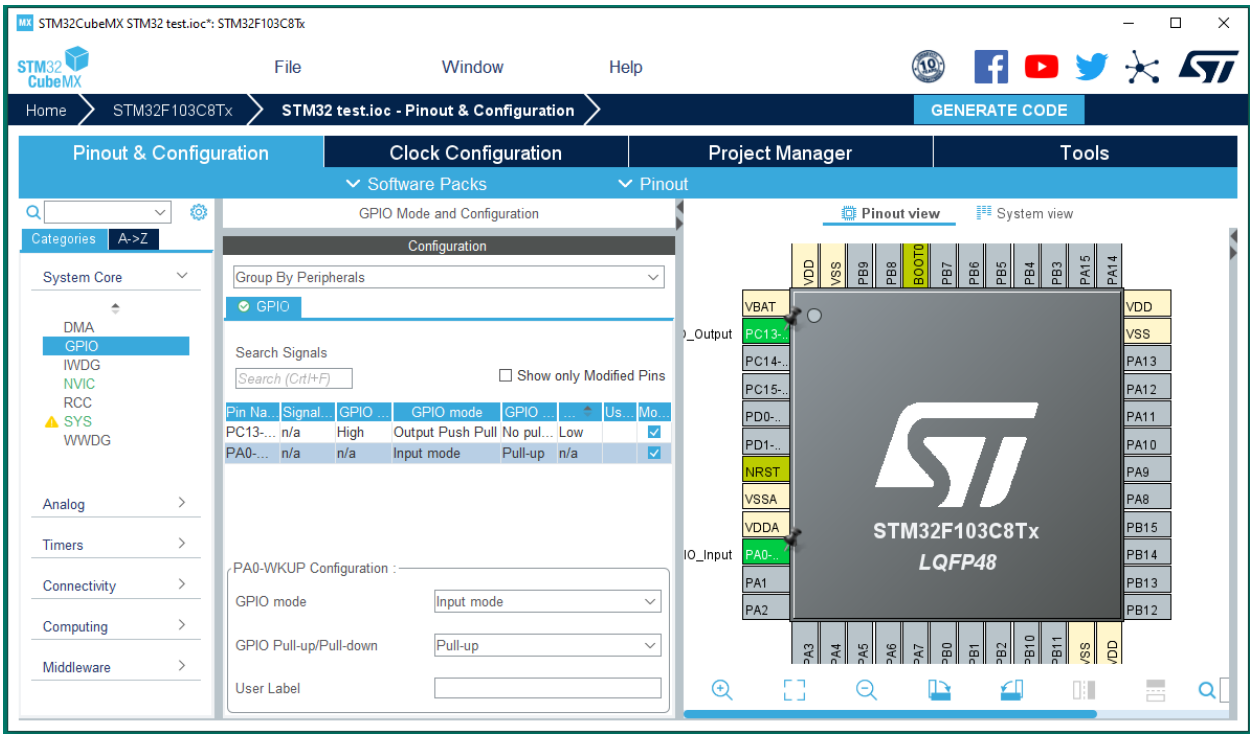

Рисунок 3.4 Налаштування параметрів виводів у STM32CubeMX

Також для виводу PA0 в таблиці на закладці GPIO необхідно вказати параметр *Pull-up*. Це необхідно робити у випадках, коли кнопка підключена між вхідним виводом і мінусовим провід живлення, тому початковий стан виводу має бути підтягнутий до напруги живлення (рівень HIGH) . Початковий стан виводу PC13 має бути HIGH, тому що анод світлодіода під'єднано до напруги живлення. Після натискання кнопки "GENERATE CODE" і створення нового коду ініціалізації проект в IAR EW змінюється і має вигляд, наведений нижче.

```
int main(void)
{
  /* Reset of all peripherals, Initializes the Systick. */
```

```
  HAL_Init();
  /* Configure the system clock */
  SystemClock_Config();
  /* Initialize all configured peripherals */
  MX_GPIO_Init();
  while (1)
  {
  }
}
```

В файлі `main.c`, в функції `main(void)` з'являється функція ініціалізації виводів мікроконтролера `MX_GPIO_Init()`, яка містить задані в STM32CubeMX налаштування виводів PA0 та PC13. Тепер в циклі можна розмістити простий код включення світлодіода при натисканні кнопки:

```
  if(HAL_GPIO_ReadPin(GPIOA, GPIO_PIN_0) == 0)
      HAL_GPIO_WritePin(GPIOC, GPIO_PIN_13, GPIO_PIN_RESET);
  else
      HAL_GPIO_WritePin(GPIOC, GPIO_PIN_13, GPIO_PIN_SET);
```

Контрольні запитання

1. Призначення GPIO. Які режими роботи забезпечує?
2. Які характеристики має порт вводу-виводу запрограмований як вхідний?
3. Що забезпечують режими входу із внутрішньою прив'язкою (підтяжкою) до напруги живлення або нуля?

Лекція 4. Обробка переривань

Робота з апаратним забезпеченням пов’язано з асинхронними подіями. Більшість із них надходить від апаратної периферії. Наприклад, таймер, що досягає налаштованого значення періоду, або UART, який попереджає про надходження даних. Усі мікроконтролери мають функцію, яка називається перериваннями. Переривання — це асинхронна подія, яка викликає зупинку виконання поточного коду на основі пріоритету (чим важливіше переривання, тим вище його пріоритет; це призведе до призупинення переривання з нижчим пріоритетом). Код, який обслуговує переривання, називається процедурою обслуговування переривань *Interrupt Service Routines* (ISR).

Переривання є джерелом мультипрограмування: апаратне забезпечення знає про них, і воно відповідає за збереження поточного контексту виконання (тобто вміст стеку, поточний лічильник програм та інш.) перед перемиканням на ISR. Операційні системи реального часу використовують їх для введення поняття завдань. Без апаратної допомоги неможливо створити надійну систему, яка дозволяє перемикатися між кількома контекстами виконання без непоправної втрати поточного потоку виконання.

Переривання можуть виникати як апаратним, так і програмним забезпеченням. Архітектура ARM розрізняє два типи: переривання виникають через апаратне забезпечення та винятки з боку програмного забезпечення (наприклад, доступ до недопустимого місця в пам’яті). У термінології ARM переривання є типом винятку. Процесори Cortex-M забезпечують блок, призначений для керування винятками. Це називається вбудованим векторним контролером переривань *Nested Vectored Interrupt Controller* (NVIC).

4.1 Вбудований векторний контролер переривань NVIC

NVIC — це спеціальний апаратний блок всередині мікроконтролерів на основі Cortex-M, який відповідає за обробку винятків. На рисунку 4.1 показано співвідношення між блоком NVIC, ядром процесора та периферійними пристроями.

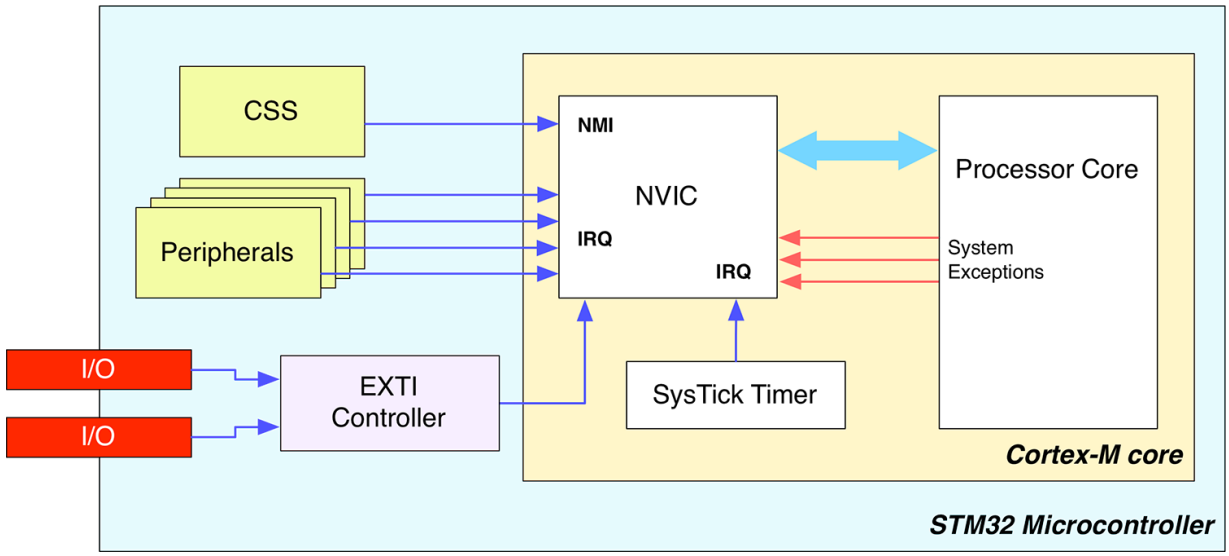

Рисунок 4.1 Взаємодія між контролером NVIC, ядром Cortex-M і периферійними пристроями STM32

Потрібно розрізняти два типи периферійних пристроїв: зовнішні по відношенню до ядра Cortex-M, але внутрішні по відношенню до MCU STM32 (наприклад, таймери, UARTS і так далі), і периферійні пристрої, зовнішні для MCU взагалі. Джерелом переривань, що надходять від периферійних пристроїв останнього типу, є вводи/виводи MCU, які можна налаштувати як введення/виведення загального призначення (наприклад, тактильний перемикач, підключений до контакту, налаштованого як вхід), або для керування зовнішнім периферійним пристроєм. Спеціальний програмований контролер під назвою *External Interrupt/Event Controller* (EXTI) відповідає за

взаємозв’язок між зовнішніми сигналами вводу/виводу та контролером NVIC, як побачимо далі.

Як зазначалося раніше, ARM розрізняє системні винятки, які виникають всередині ядра процесора, і апаратні винятки, що надходять від зовнішніх периферійних пристроїв, які також називаються запитами на переривання (IRQ). Винятки повинні оброблятися за допомогою спеціальних ISR. Процесор знає, де знайти ці підпрограми завдяки непрямій таблиці, що містить адреси в пам'яті обслуговування переривань. Цю таблицю зазвичай називають векторною таблицею, і кожен мікроконтролер STM32 визначає свою власну.

4.2 Таблиця векторів переривань STM32

Усі процесори Cortex-M визначають фіксований набір винятків (п’ятнадцять для ядер Cortex-M3/4/7), які наведено в Таблиці 4.1). Нижче наведено деякі з них:

- `Reset`: цей виняток виникає відразу після скидання процесора. Його обробник є точкою входу в запущену програму. У програмі STM32 все починається з цього винятку. Обробник містить деякі функції, закодовані на асемблері, призначені для ініціалізації середовища виконання, наприклад, основний стек.
- `NMI`: це особливий виняток, який має найвищий пріоритет після скидання. Як і виняток Reset, його не можна замаскувати (тобто вимкнути), і його можна пов’язувати з критичними та невідкладними діями. У мікроконтролерах STM32 він пов'язаний з *Clock Security System* (CSS). CSS — це периферійний пристрій для самодіагностики, який виявляє збій зовнішнього осцилятора HSE. Якщо це станеться, HSE автоматично вимикається (це означає, що внутрішній осцилятор HSI

автоматично вмикається) і виникає переривання NMI, щоб повідомити програмне забезпечення, що щось не так з HSE.

Таблиця 4.1 Типи винятків Cortex-M

Number	Exception type	Priority	Function
1	Reset	-3	Reset
2	NMI	-2	Non-Maskable Interrupt
3	Hard Fault	-1	All classes of Fault, when the fault cannot activate because of priority or the Configurable Fault handler has been disabled.
4	Memory Management[c]	Configurable	MPU mismatch, including access violation and no match. This is used even if the MPU is disabled or not present.
5	Bus Fault[c]	Configurable	Pre-fetch fault, memory access fault, and other address/memory related.
6	Usage Fault[c]	Configurable	Usage fault, such as Undefined instruction executed or illegal state transition attempt.
7-10	-	-	RESERVED
11	SVCall	Configurable	System service call with SVC instruction.
12	Debug Monitor	Configurable	Debug monitor – for software based debug.
13	-	-	RESERVED
14	PendSV	Configurable	Pending request for system service.
15	SysTick	Configurable	System tick timer has fired.
16-[47/240]	IRQ	Configurable	IRQ Input

- `Hard Fault`: є загальним винятком помилки, і, отже, пов'язаний з програмними перериваннями. Коли інші винятки помилок вимкнено, він діє як збірник для всіх типів винятків.
- `Memory Management Fault`: виникає, коли виконання коду намагається отримати доступ до недозволеної області пам’яті.
- `Bus Fault`: виникає, коли інтерфейс AHB отримує відповідь про помилку від підпорядкованої шини (також називається перериванням попередньої вибірки, якщо це вибірка інструкції, або зупинкою даних, якщо це доступ до даних). Також може бути викликано іншим помилковим доступом (наприклад, доступом до неіснуючого місця в пам’яті SRAM).

- `Usage Fault`: Помилка використання, яка виникає, коли виникає програмна помилка, наприклад, неправильна інструкція, проблема з вирівнюванням або спроба отримати доступ до неіснуючого співпроцесора.
- SVCCall не є наслідком несправності, і вона виникає при використанні команди виклику супервізора (SVC). Використовується операційними системами реального часу для виконання інструкцій у привілейованому стані (завдання, яке потребує виконання привілейованих операцій, виконує інструкцію SVC, а ОС виконує запитані операції — це така ж поведінка системного виклику в іншій ОС).
- `Debug monitor`: цей виняток виникає, коли відбувається подія налагодження програмного забезпечення, коли ядро процесора знаходиться в режимі налагодження моніторингу. Він також використовується як виняток для подій налагодження, таких як точки зупинки та спостереження, коли використовується програмне рішення для налагодження.
- `PendSV`: це ще один виняток, пов'язаний з операційними системами реального часу. На відміну від винятку SVCall, який виконується відразу після виконання інструкції SVC, PendSV може бути відкладено. Це дозволяє операційній системі виконувати завдання з вищими пріоритетами.
- `SysTick`: цей виняток зазвичай пов'язаний з діяльністю RTOS. Кожній операційній системі реального часу потрібен таймер, щоб періодично переривати виконання поточного коду та перемикатися на інше завдання. Усі мікроконтролери STM32 забезпечують таймер SysTick, внутрішній для ядра Cortex-M. Навіть якщо кожен інший таймер може використовуватися для планування системної діяльності, наявність

спеціального таймера забезпечує переносимість між усіма сімействами STM32 (через причини оптимізації, пов’язані з внутрішнім кристалом MCU, не всі таймери можуть бути доступні як зовнішні периферійні пристрої). CubeHAL використовує таймер SysTick для виконання внутрішніх дій, пов’язаних із часом (він передбачає, що таймер SysTick налаштований на генерацію переривання кожні 1 мс).

Решта винятків, які можна визначити для даного процесора, пов’язані з обробкою IRQ. Ядра Cortex-M3/4/7 дозволяють визначити до 240 переривань. Список переривань можна знайти у векторній таблиці всередині файлу запуску для вибраного MCU - `startup_stm32f103xb.s`. Відкривши цей файл, можна знайти всю векторну таблицю для цього MCU.

Навіть якщо векторна таблиця містить адреси підпрограм-обробників, ядру Cortex-M потрібен спосіб знайти векторну таблицю в пам’яті. За замовченням, векторна таблиця починається з апаратної адреси 0x0000 0000 у всіх процесорах Cortex-M. Якщо векторна таблиця знаходиться у внутрішній флеш-пам’яті (це зазвичай має місце), і оскільки флеш-пам’ять у всіх MCU STM32 відображається з адреси 0x0800 0000, вона розміщується, починаючи з адреси 0x0800 0000, яка змінюється на 0x0000 0000, коли процесор завантажується з пам’яті, відмінної від внутрішньої флеш-пам’яті. Тому, щоб уникнути плутанини, найкраще вважати позицію векторної таблиці фіксованою та прив’язаною до адреси 0x0000 0000.

На Рисунку 4.2 показано, як векторна таблиця організована в пам'яті. Нульовим записом цього масиву є адреса вказівника основного стека (MSP) всередині SRAM. Зазвичай ця адреса відповідає кінцю SRAM, тобто її базовій адресі + її розміру. Починаючи з другого запису цієї таблиці, можна знайти всі винятки та обробники переривань. Це означає, що векторна таблиця має

довжину 48 для мікроконтролерів на основі Cortex-M0/0+ і довжину 256 для Cortex-M3/4/7.

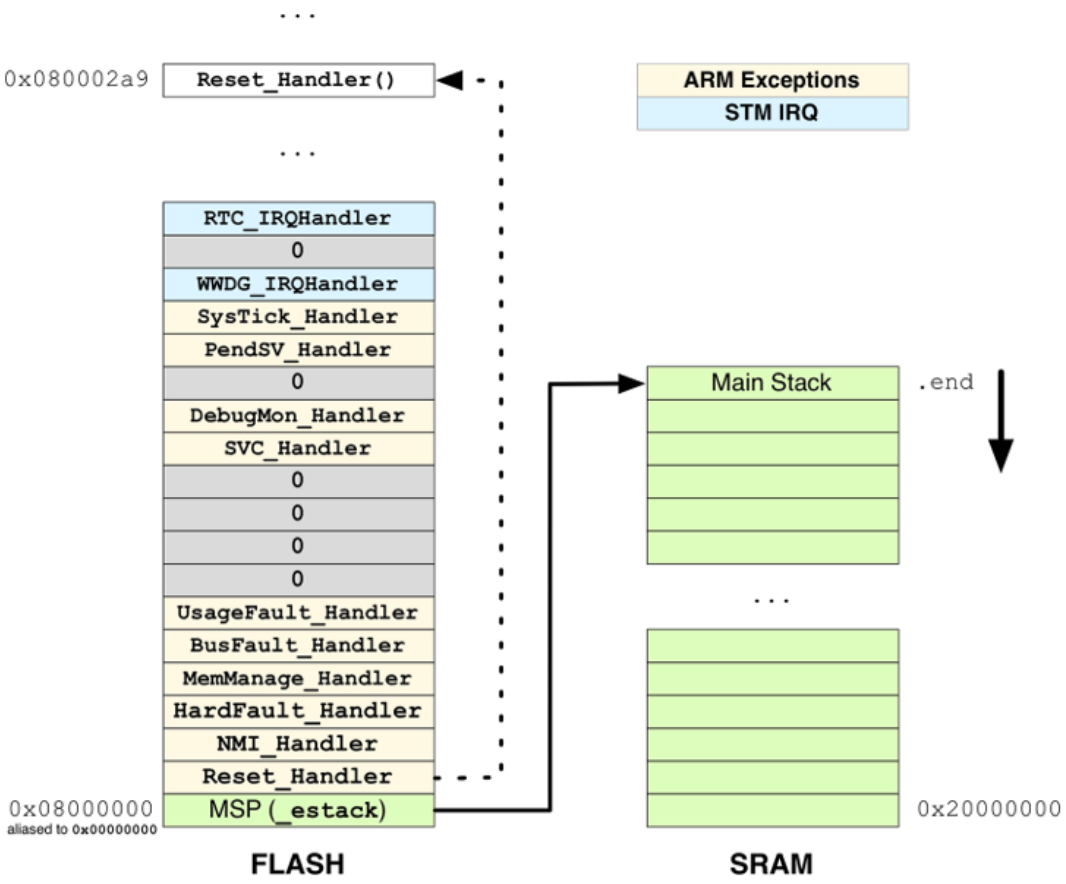

Рисунок 4.2 Розміщення векторної таблиці в STM32 MCU на основі Cortex-M3/4/7 ядра

Важливо прояснити деякі речі щодо векторної таблиці.

1. Назви обробників винятків є лише умовними, і їх можна перейменувати. Це просто символи (як і змінні та функції всередині програми). Однак треба пам'ятати, що програмне забезпечення CubeMX призначене для створення ISR з цими іменами, які є умовою ST. Отже, тоді також доведеться перейменувати ім’я ISR.

2. Як було сказано раніше, векторна таблиця повинна бути розміщена на початку флеш-пам'яті, де процесор очікує її знайти. Це завдання редактора посилань, яке розміщує векторну таблицю на початку даних flash під час генерації абсолютного файлу, який є двійковим файлом, який завантажується на flash.

4.3 Керування перериваннями

Коли завантажується STM32 MCU, за замовчуванням вмикаються лише винятки *Reset, NMI* та *Hard Fault*. Решта винятків і периферійних переривань вимкнені, і їх потрібно ввімкнути за запитом. Щоб увімкнути IRQ, CubeHAL надає таку функцію:

```
void HAL_NVIC_EnableIRQ(IRQn_Type IRQn);
```

де IRQn_Type є перерахуванням усіх винятків і переривань, визначених для цього конкретного MCU. Перерахування IRQn_Type є частиною ST Device HAL, і воно визначено у файлі заголовка, специфічному для даного MCU STM32 `stm32f103xb.h`.

Відповідна функція для вимкнення IRQ:

```
void HAL_NVIC_DisableIRQ(IRQn_Type IRQn);
```

Важливо зауважити, що дві попередні функції дозволяють/виключають переривання на рівні контролера NVIC. Більшість периферійних пристроїв STM32 призначені для роботи у режимі переривань. Використовуючи спеціальні підпрограми HAL, можна ввімкнути переривання на периферійному рівні. Наприклад, за допомогою `HAL_USART_Transmit_IT()` неявно налаштовуємо периферійний пристрій USART у режимі переривань. Зрозуміло, що також потрібно ввімкнути відповідне переривання на рівні NVIC, викликавши `HAL_NVIC_EnableIRQ()`. STM32CubeMX автоматично

прописує цю функцію в коді програми при ввімкненні відповідного переривання під час налаштувань.

4.3 Зовнішні лінії переривань та NVIC

Як видно з Рисунку 1, мікроконтролери STM32 забезпечують різну кількість зовнішніх джерел переривань, підключених до NVIC через контролер EXTI, який, у свою чергу, здатний керувати кількома лініями EXTI. Кількість джерел і ліній переривань залежить від конкретного сімейства STM32.

GPIO підключено до ліній EXTI, що може дозволити переривання для кожного GPIO мікроконтролера, навіть якщо більшість з них використовує одну лінію переривань. На Рисунку 4.3 показані лінії EXTI 0, 10 і 15 в MCU STM32F4. Всі виводи *Px0* підключені до *EXTI0*, всі контакти *Px10* підключені до *EXTI10*, а всі виводи *Px15* підключені до *EXTI15*. Однак лінії EXTI 10 і 15 мають однаковий IRQ всередині NVIC (і, отже, обслуговуються одним і тим же ISR). Це означає що:

- Джерелом переривань може бути лише один вивід PxY, навіть якщо до лінії EXTI їх підключено декілька.
- Для ліній EXTI, які мають однаковий IRQ всередині контролера NVIC, треба програмно визначати відповідний ISR, щоб мати можливість розрізняти, які лінії генерували переривання.

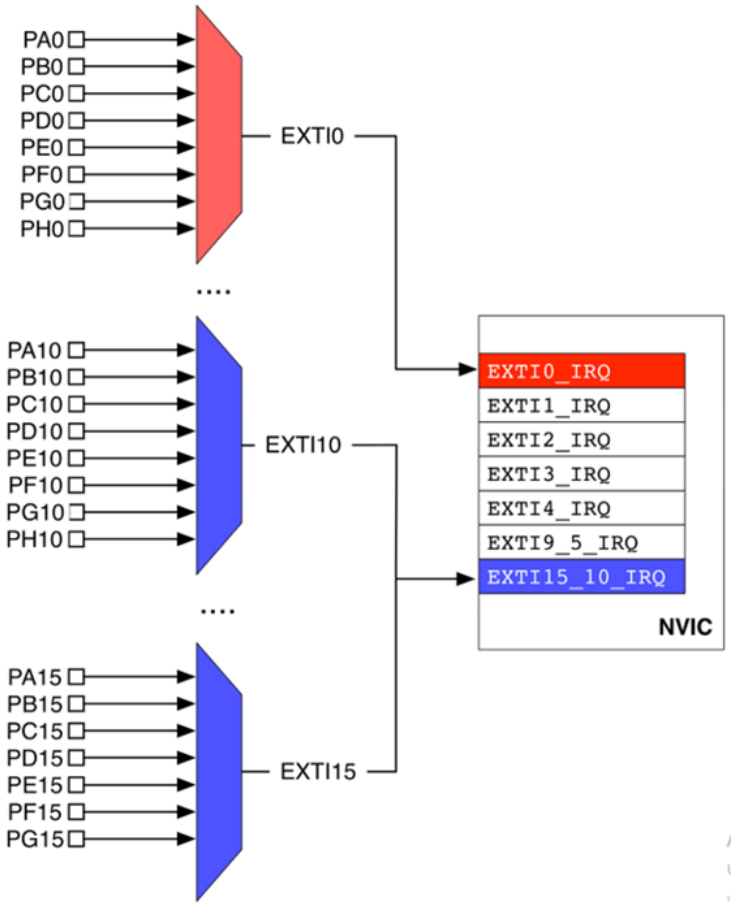

Рисунок 4.3 Зв'язок між лініями GPIO, EXTI та відповідним ISR

Контрольні запитання

1. Що таке переривання та для чого вони використовується?
2. Який апаратний блок відповідає за обробку винятків і як він функціонує ?
3. Для чого використовуються функції `HAL_NVIC_EnableIRQ(IRQn_Type IRQn)` та `HAL_NVIC_DisableIRQ(IRQn_Type IRQn)`?

Лекція 5. Особливості роботи переривань

5.1 Налаштування переривань GPIO в STM32CubeMX

Для ілюстрації налаштування переривань для GPIO можна використати попередній приклад для плати STM32 Smart V2. Використаємо переривання для перемикання світлодіода на виводі PC13 щоразу, коли натискається кнопка, яка під'єднана до контакту PA0. Для цього необхідно задати вивід PA0 як *GPIO_EXTI0*. Потім необхідно налаштувати PA0 так, щоб він запускав переривання щоразу, коли сигнал переходить з високого рівня на нижній при натисканні кнопки, вибором *GPIO Mode* "*External Interrupt Mode with Falling edge trigger detection*" в *PA0-WKUP Configuration*, як показано на Рисунку 5.1.

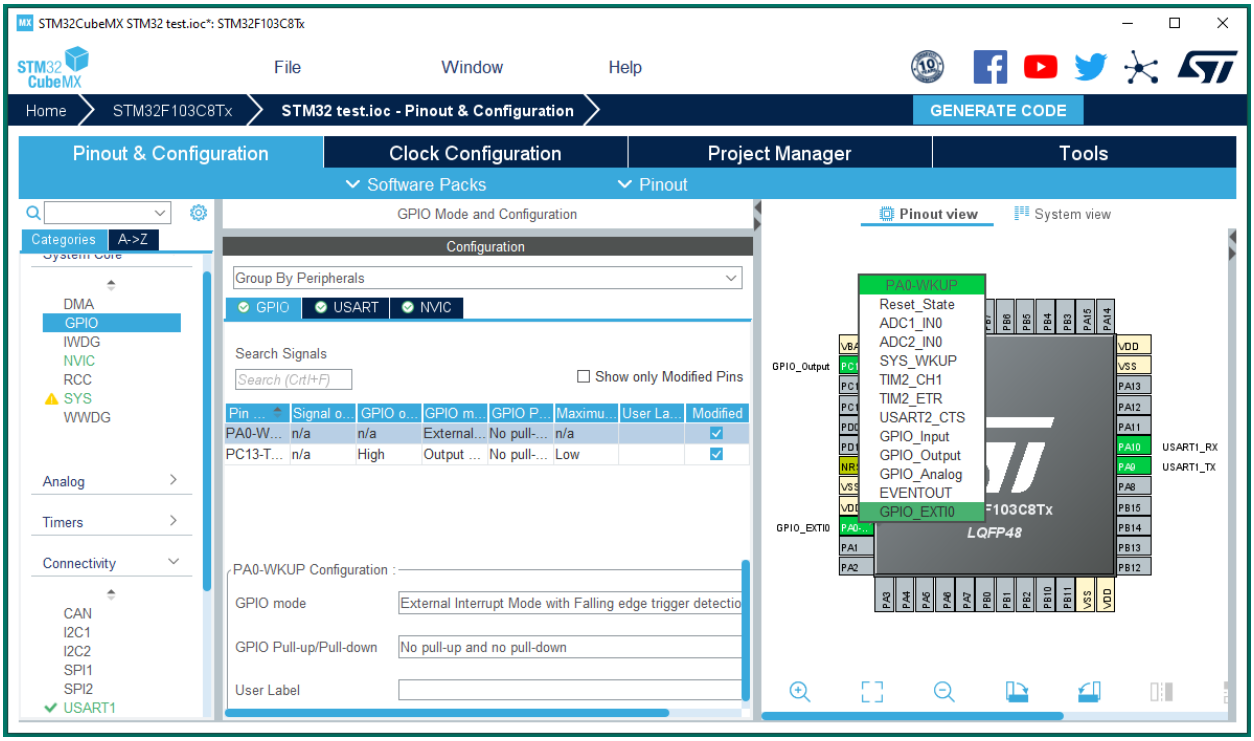

Рисунок 5.1 Налаштування зовнішнього переривання на виводі PA0.

Далі у налаштуваннях NVIC необхідно дозволити переривання лінії *EXTI line 0 interrupt*, пов'язаної з виводом PA0, як показано на Рисунку 5.2.

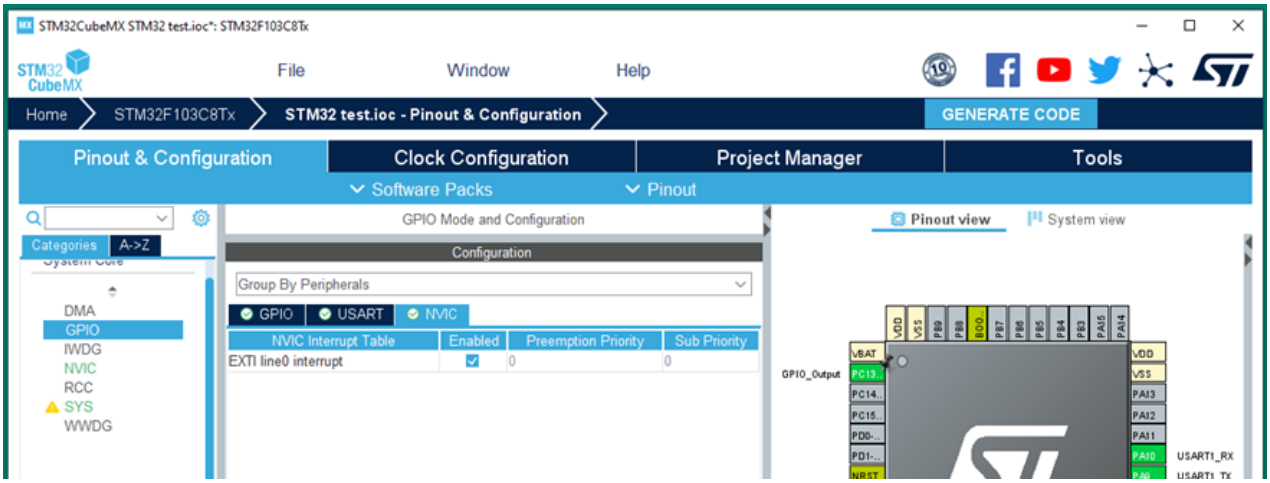

Рисунок 5.2 Підключення зовнішнього переривання *EXTI line 0*.

Код налаштувань `MX_GPIO_Init()` GPIO, згенерований в STM32CubeMX, має наступний вигляд:

```
static void MX_GPIO_Init(void)
{
    GPIO_InitTypeDef GPIO_InitStruct = {0};
  /* GPIO Ports Clock Enable */
  __HAL_RCC_GPIOC_CLK_ENABLE();
  __HAL_RCC_GPIOA_CLK_ENABLE();

  /*Configure GPIO pin Output Level */
  HAL_GPIO_WritePin(GPIOC, GPIO_PIN_13, GPIO_PIN_SET);

  /*Configure GPIO pin : PC13 */
  GPIO_InitStruct.Pin = GPIO_PIN_13;
  GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
  GPIO_InitStruct.Pull = GPIO_NOPULL;
  GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
  HAL_GPIO_Init(GPIOC, &GPIO_InitStruct);
```

```
  /*Configure GPIO pin : PA0 */
  GPIO_InitStruct.Pin = GPIO_PIN_0;
  GPIO_InitStruct.Mode = GPIO_MODE_IT_FALLING;
  GPIO_InitStruct.Pull = GPIO_NOPULL;
  HAL_GPIO_Init(GPIOA, &GPIO_InitStruct);

  /* EXTI interrupt init*/
  HAL_NVIC_SetPriority(EXTI0_IRQn, 0, 0);
  HAL_NVIC_EnableIRQ(EXTI0_IRQn);
}
```

В програмному коді задано режим переривання щоразу, коли сигнал змінюється з високого рівня на нижній `GPIO_MODE_IT_FALLING` та наявна функція включення переривання `HAL_NVIC_EnableIRQ()`. Програма керування світлодіодом по перериванню наведена нижче:

```
int main(void)
{
  HAL_Init();
  SystemClock_Config();
  MX_GPIO_Init();

  while (1);
}
void EXTI0_IRQHandler(void)
{
  HAL_GPIO_EXTI_IRQHandler(GPIO_PIN_0);
  HAL_GPIO_TogglePin(GPIOC, GPIO_PIN_13);
}
```

Функція `EXTI0_IRQHandler()`знаходиться в файлі `stm32f1xx_it.c` і забезпечує обробку переривання. ISR спрацьовує при натисканні кнопки і

змінює стан виводу PC13, також очищується біт наявності переривання, пов'язаний з лінією EXTI0.

Також для обробки переривання можна використати функцію `HAL_GPIO_EXTI_Callback ()`, яка на відміну від `EXTI0_IRQHandler()` не прив'язана до конкретної лінії переривань. Але у цьому випадку може бути потрібна додаткова перевірка того, на якому виводі спрацювало переривання, якщо їх включено декілька. Програмний код у цьому випадку має вигляд:

```
void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin) {
  if(GPIO_Pin == GPIO_PIN_0)
    HAL_GPIO_TogglePin(GPIOC, GPIO_PIN_13);
}
```

Коли викликається `EXTI0_IRQHandler()`, керування передається функції `HAL_GPIO_EXTI_IRQHandler()` всередині HAL. Це виконає усі дії, пов'язані з перериваннями, і викличе функцію `HAL_GPIO_EXTI_Callback()`, передаючи GPIO, який генерував IRQ. На Рисунку 5.3 показано послідовність викликів, яка генерується з IRQ.

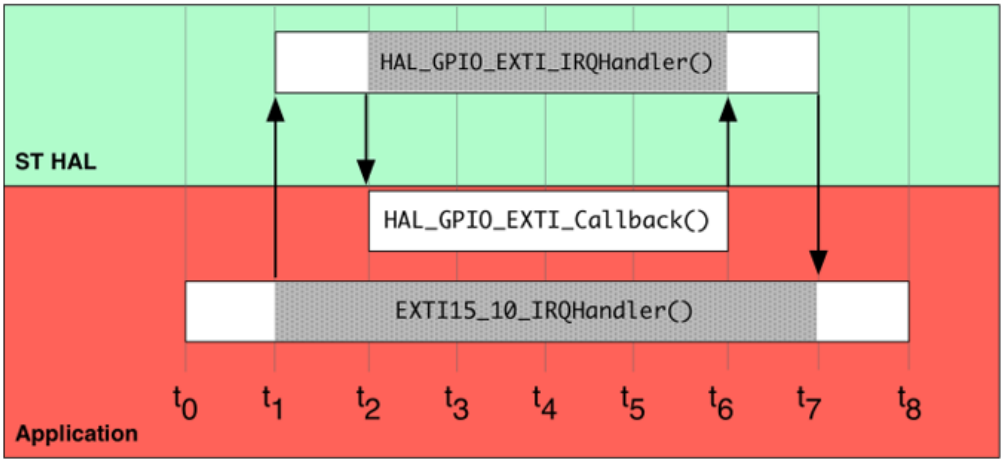

Рисунок 5.3 Порядок обробки IRQ в HAL

Цей механізм використовується майже всіма підпрограмами IRQ всередині HAL. Якщо декілька ліній EXTI підключені до одного IRQ, то потрібно розрізняти в коді, який із контактів генерував переривання. Цю роботу виконує HAL, передаючи параметр GPIO_Pin під час виклику функції зворотного виклику.

5.2 Тривалість дії переривання

Для коректної роботи з перериваннями, дуже важливо зрозуміти їх життєвий цикл. Хоча ядро Cortex-M автоматично виконує більшу частину роботи, треба звернути увагу на деякі аспекти, які можуть стати джерелом ускладнень під час керування перериваннями.

Переривання може знаходитись у наступних станах:

1. бути вимкненим (поведінка за замовчуванням) або увімкненим. Для чого його вмикають / вимикають, викликаючи відповідно функції `HAL_NVIC_EnableIRQ() / HAL_NVIC_DisableIRQ();`

2. перебувати в режимі очікування (запит очікує на обслуговування) або не перебувати;

3. бути в активному стані (обслуговується) або неактивному стані.

Перший випадок вже розглянуто раніше. Важливо також зрозуміти, що відбувається, коли виникає переривання. Коли спрацьовує переривання, воно позначається як очікуване, доки процесор не зможе його обслуговувати. Якщо в даний момент не обробляються жодні інші переривання, процес очікування автоматично очищає його стан, який майже відразу починає його обслуговувати. На Рисунку 5.4 показано, як це працює. Переривання A спрацьовує в момент t0, і оскільки ЦП не обслуговує інше переривання, його

біт очікування очищається, і його виконання починається негайно (переривання стає активним). У момент t1 спрацьовує переривання B, але тут припустимо, що воно має нижчий пріоритет, ніж A. Тому воно залишається в стані очікування, доки A ISR не завершить свої операції. Коли це відбувається, біт очікування автоматично очищається, і ISR стає активним.

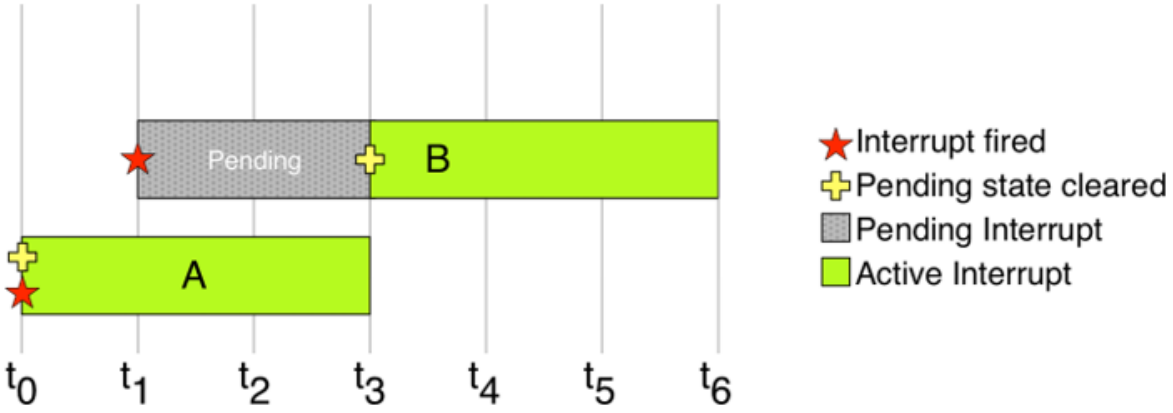

Рисунок 5.4 Відношення між бітом очікування та активним статусом переривання

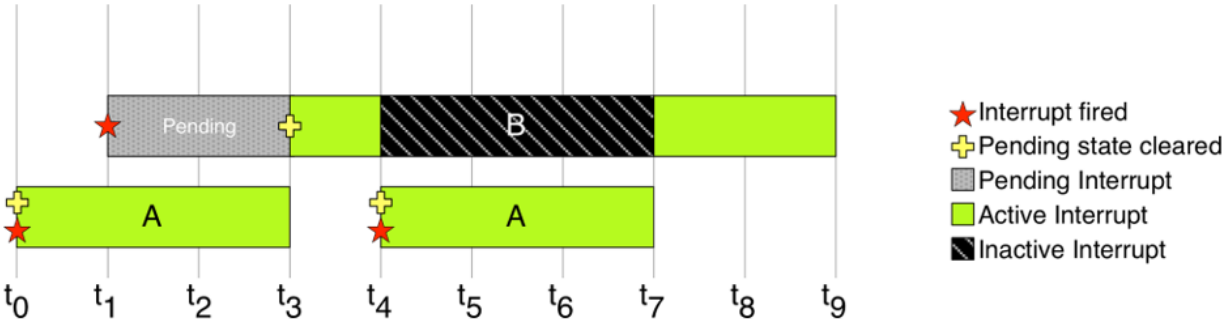

Рисунок 5.5 Зв'язок між активним статусом і пріоритетом переривань

На рисунку 5.5 показаний ще один важливий випадок. Тут спрацьовує переривання A, і центральний процесор може негайно його обслуговувати. Переривання B спрацьовує, поки A обслуговується, тому воно залишається в стані очікування до завершення A. Коли це відбувається, біт переривання B,

який очікує обробки, очищається, і воно стає активним. Однак через деякий час переривання A запускається знову, і оскільки воно має вищий пріоритет, переривання B призупиняється (стає неактивним), і виконання A починається негайно. Коли це закінчиться, переривання B знову стає активним, і воно завершує свою роботу.

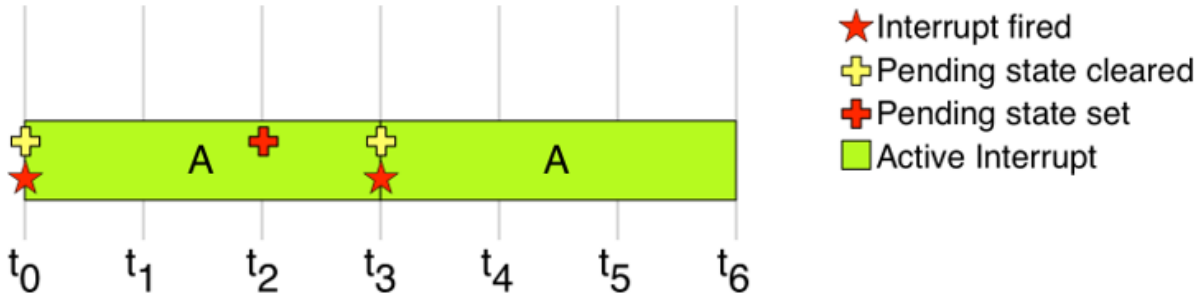

Рисунок 5.6 Як переривання може бути примусово спрацювати знову, встановивши його біт очікування

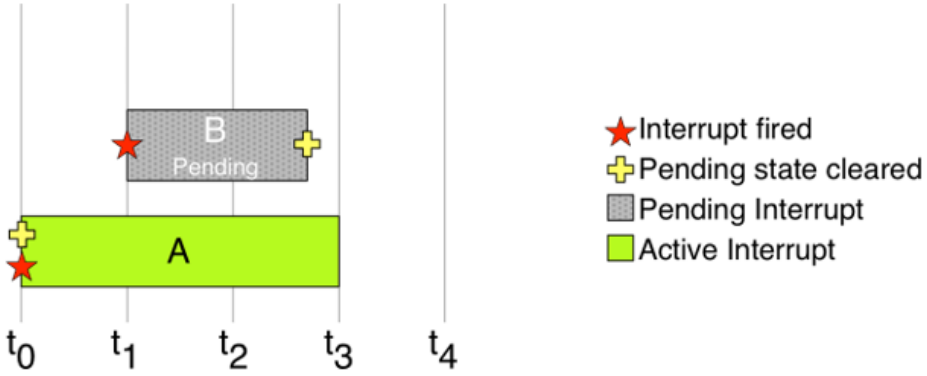

Рисунок 5.7 Обслуговування IRQ можна скасувати, очистивши його біт, що очікує на виконання, до його виконання

NVIC забезпечує високий ступінь гнучкості для програмістів. Переривання можна змусити знову спрацьовувати під час його виконання, просто знову встановивши його біт очікування, як показано на Рисунку 5.6.

Таким же чином виконання переривання може бути скасовано, очищаючи його біт очікування, поки він знаходиться в стані очікування, як показано на Рисунку 5.7.

Тут важливо зрозуміти важливий аспект, пов’язаний з тим, як периферійні пристрої попереджають контролер NVIC про запит на переривання. Коли відбувається переривання, більшість периферійних пристроїв STM32 підтверджують певний сигнал, підключений до NVIC, який відображається в периферійній пам’яті через виділений біт. Цей біт запиту периферійного переривання буде утримуватися на високому рівні, поки він не буде очищений кодом програми. Наприклад, у прикладі 1 треба чітко очистити біт очікування IRQ лінії EXTI за допомогою макросу `HAL_GPIO_EXTI_CLEAR_IT()`. Якщо не скинути цей біт, то буде спрацьовувати нове переривання, поки воно не буде очищено.

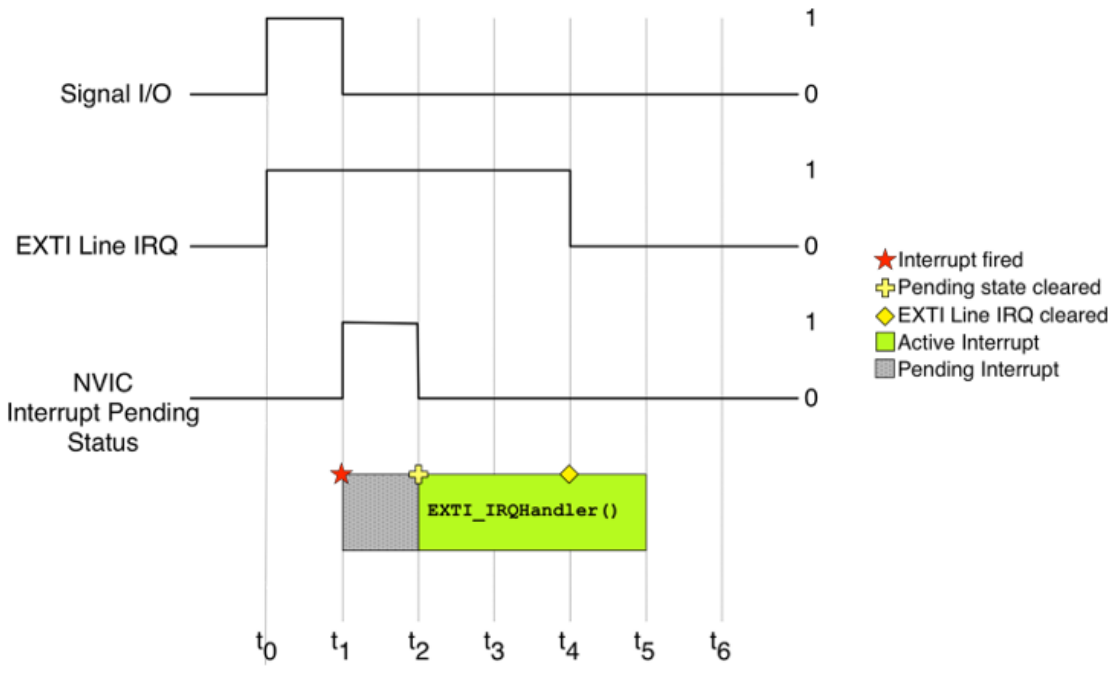

Рисунок 5.8 Зв'язок між периферійним IRQ і відповідним перериванням

На Рисунку 5.8 чітко показано співвідношення між станом очікування периферійного IRQ і станом очікування ISR. Сигнальний вхід-вихід — це зовнішній периферійний пристрій, який управляє вводом-виводом (наприклад, тактильний перемикач, підключений до контакту). Коли рівень сигналу змінюється, лінія EXTI, підключена до цього входу/виводу, генерує IRQ, і встановлюється відповідний біт очікування. Як наслідок, NVIC генерує переривання. Коли процесор починає обслуговувати ISR, біт очікування ISR очищається автоматично, але периферійний біт очікування IRQ буде утримуватися на високому рівні, поки він не буде очищений кодом програми.

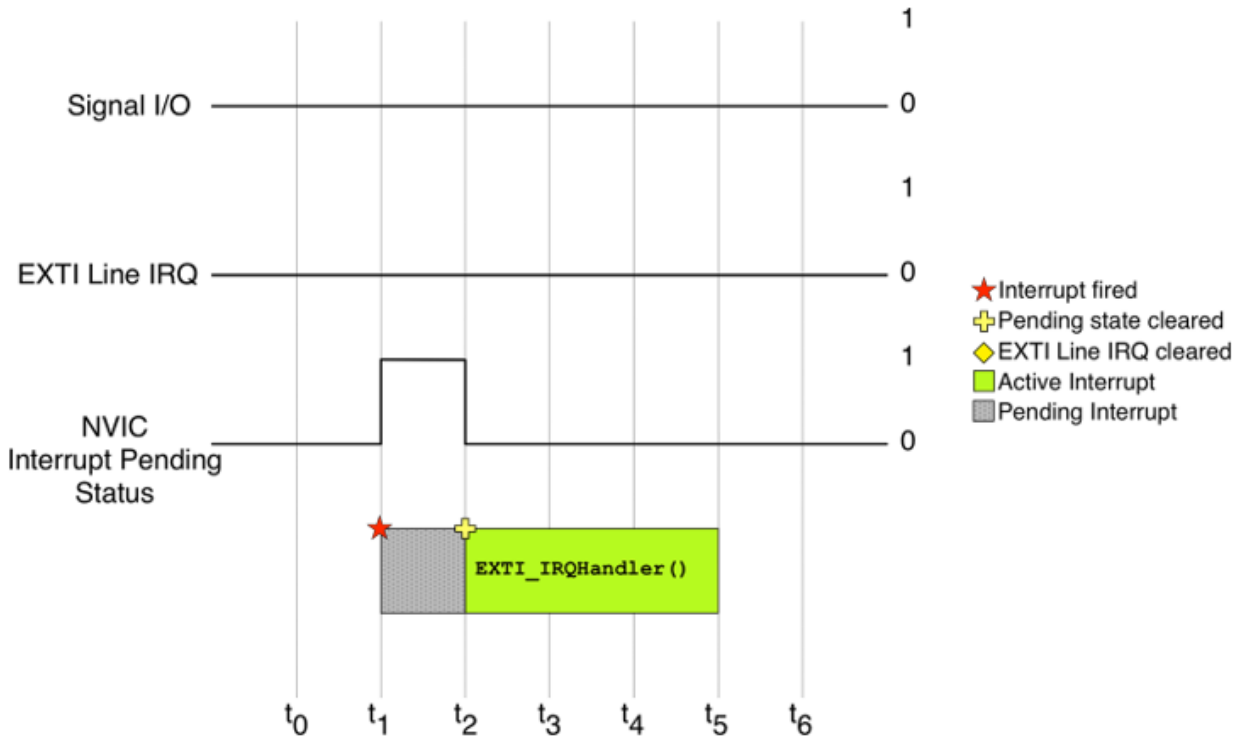

Рисунок 5.9 Коли переривання примусово встановлює свій біт очікування, відповідний периферійний IRQ залишається не встановленим

На Рисунку 5.9 показаний інший випадок. Тут примусово виконується ISR встановленням його біта очікування. Оскільки на цей раз зовнішній

периферійний пристрій не задіяний, немає необхідності очищати відповідний біт очікування IRQ.

Оскільки наявність біта очікування IRQ залежить від периферії, завжди доцільно використовувати функції ST HAL для керування перериваннями, залишаючи всі основні деталі реалізації HAL. Однак потрібно мати на увазі, що для уникнення втрати важливих переривань, гарною практикою проектування є очищення біта стану очікування IRQ периферійних пристроїв, коли їх ISR починає обслуговуватися. Ядро процесора не відстежує численні переривання (воно не ставить в чергу переривань), тому, якщо очистити периферійний біт очікування в кінці ISR, можна втратити важливі IRQ, які запускаються в середині.

Щоб побачити, чи переривання очікує (тобто спрацьовує, але не виконується), можна використовувати функцію HAL:

```
uint32_t HAL_NVIC_GetPendingIRQ(IRQn_Type IRQn);
```

яка повертає 0, якщо IRQ не очікує на розгляд, 1 в іншому випадку.

Щоб програмно встановити біт очікування IRQ, можна використовувати функцію HAL:

```
void HAL_NVIC_SetPendingIRQ(IRQn_Type IRQn);
```

Це призведе до запуску переривання, оскільки воно буде створено апаратним забезпеченням. Відмінною особливістю процесорів Cortex-M є можливість програмно викликати переривання всередині підпрограми ISR іншого переривання.

Замість цього, щоб програмно очистити відкладений біт IRQ, можна використовувати функцію:

```
void HAL_NVIC_ClearPendingIRQ(IRQn_Type IRQn);
```

Знову ж таки, можна також очистити виконання очікуваного переривання всередині ISR, обслуговуючи інше IRQ. Щоб перевірити, чи активний ISR (обслуговується IRQ), можна використовувати функцію:

```
uint32_t HAL_NVIC_GetActive(IRQn_Type IRQn);
```

яка повертає 1, якщо IRQ активний, або 0 в іншому випадку.

5.3 Налаштування рівня пріоритету переривань

Відмінною рисою архітектури ARM Cortex-M є можливість визначення пріоритету переривань (за винятком перших трьох програмних винятків, які мають фіксований пріоритет, як показано в таблиці 1). Пріоритет переривання дозволяє визначити дві речі:

- ISR, які будуть виконуватися першими в разі одночасних переривань;
- ті підпрограми, які можна за бажанням випередити для початку виконання ISR з вищим пріоритетом.

У ядрах Cortex-M3/4/7 пріоритет кожного переривання визначається через регістр *Interrupt Priority Register* (IPR). Це 8-розрядний регістр в архітектурі ядра ARMv7-M, який дозволяє використовувати до 255 різних рівнів пріоритету. Однак на практиці мікроконтролери STM32, що реалізують ці ядра, використовують лише чотири верхні біти цього регістра, а всі інші біти дорівнюють нулю.

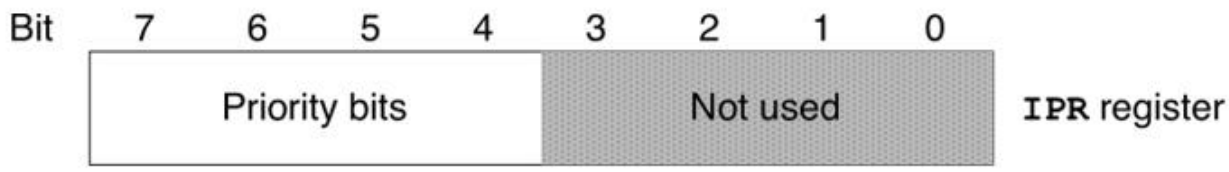

Рисунок 5.10 Вміст регістру IPR в MCU на основі ядра Cortex-M3/4/7

На Рисунку 5.10 показано, як інтерпретується зміст IPR. Це означає, що є лише шістнадцять рівнів пріоритету: 0x00, 0x10, 0x20, 0x30, 0x40, 0x50, 0x60, 0x70, 0x80, 0x90, 0xA0, 0xB0, 0xC0, 0xE0. Чим менше це число, тим вище пріоритет. Тобто IRQ, що має пріоритет, рівний 0x10, має вищий пріоритет, ніж IRQ з рівнем пріоритету, рівним 0xA0. Якщо двоє переривань спрацьовують одночасно, першим буде обслуговуватися те, що має більший пріоритет. Якщо процесор уже обслуговує переривання, а спрацьовує переривання з більш високим пріоритетом, то поточне переривання призупиняється, і управління переходить до переривання з вищим пріоритетом. Коли це буде завершено, виконання повертається до попереднього переривання, якщо тим часом не відбувається жодних інших переривань з вищим пріоритетом.

Регістр IPR можна логічно розділити на дві частини: серію бітів, що визначають *пріоритет випередження*, і серію бітів, що визначають *підпріоритет*. Перший рівень пріоритету регулює пріоритети випередження між ISR. Якщо ISR має пріоритет, вищий, ніж інший, він випередить виконання ISR з нижчим пріоритетом у разі його запуску. Підпріоритет визначає, який ISR буде виконано першим, у разі множинного ISR, що очікує на розгляд, але він не буде діяти на випередження ISR.

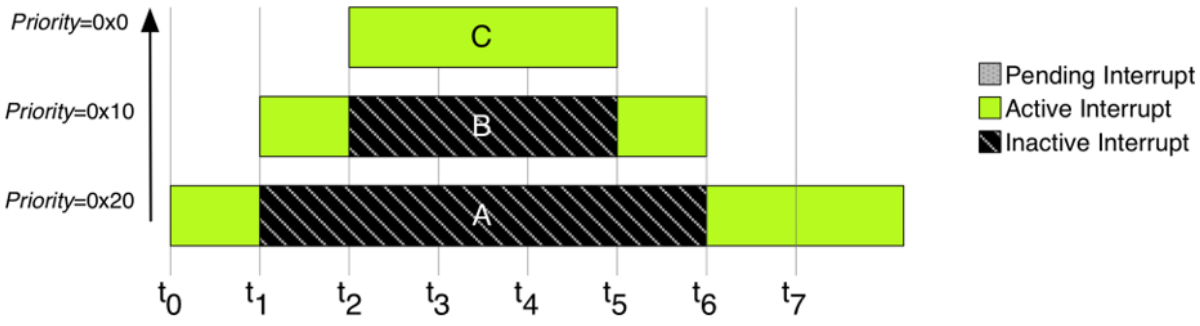

Рисунок 5.11 Виконання переривань в залежності від пріоритетності

На Рисунку 5.11 показаний приклад випередження переривань. A – це IRQ з найнижчим пріоритетом, який запускається в момент t0. ISR починає виконання, але IRQ B, який має вищий пріоритет (нижчий рівень пріоритету), запускається в момент t1, і виконання A ISR зупиняється. Через деякий час C IRQ запускається в момент t2, і B ISR зупиняється, і C ISR починає виконуватися. Коли це закінчиться, виконання B ISR відновлюється, доки не завершиться. Коли це станеться, виконання A ISR відновлюється. Цей «вкладений» механізм, викликаний пріоритетами переривань, веде до назви контролера NVIC, яке є вкладеним векторним контролером переривань.

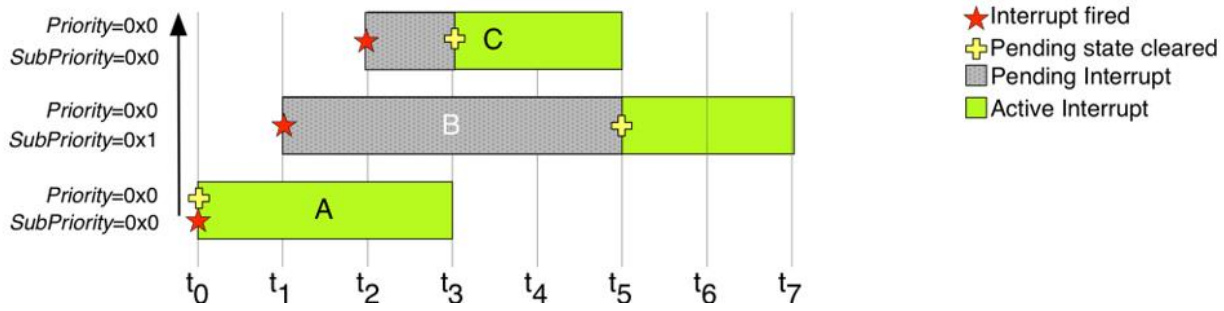

Рисунок 5.12 Якщо два переривання з однаковим пріоритетом очікують, першим виконується переривання з вищим підпріоритетом

На Рисунку 5.12 показано, як підпріоритет впливає на виконання кількох очікуваних ISR. Тут маємо три переривання, усі з однаковим максимальним пріоритетом. У момент t0 спрацьовує IRQ A, і негайно обслуговується. У момент t1 B спрацьовує IRQ, але оскільки воно має той же рівень пріоритету, що й інші IRQ, воно залишається в стані очікування. У момент t2 також спрацьовує C IRQ, але з тієї ж причини, що й раніше, процесор залишає його в стані очікування. Коли A ISR закінчується, C IRQ обслуговується першим, оскільки воно має вищий підпріоритет, ніж B. Лише після завершення C ISR, B IRQ може обслуговуватися.

Спосіб логічного поділу бітів IPR визначається регістром `SCB->AIRCR` (підгрупа бітів регістру System Control Block (SCB), і важливо підкреслити, що цей спосіб інтерпретації зміст реєстру IPR є глобальним для всіх ISR. Після того, як визначено схему пріоритетів (також звану групуванням пріоритетів у HAL), вона є спільною для всіх переривань, що використовуються в системі.

Таблиця 5.1 Кількість доступних рівнів пріоритету випередження на основі поточної схеми групування пріоритетів

NVIC Priority Group	Number of preemption priority levels	Number of sub-priority levels
NVIC_PRIORITYGROUP_0	0	16
NVIC_PRIORITYGROUP_1	2	8
NVIC_PRIORITYGROUP_2	4	4
NVIC_PRIORITYGROUP_3	8	2
NVIC_PRIORITYGROUP_4	16	0

CubeHAL забезпечує таку функцію для призначення пріоритету IRQ:

```
void HAL_NVIC_SetPriority(IRQn_Type IRQn, uint32_t PreemptPriority, uint32_t SubPriority);
```

Бібліотека HAL розроблена таким чином, що `PreemptPriority` і `SubPriority` можна налаштувати з номером рівня пріоритету в діапазоні від 0 до 16. Натомість, щоб визначити пріоритетне групування, тобто розділити регістр IPR між `PreemptPriority` і `SubPriority`, можна використовувати таку функцію:

```
void HAL_NVIC_SetPriorityGrouping(uint32_t PriorityGroup);
```

де параметр `PriorityGroup` є одним із макросів зі стовпця NVIC Priority Group у таблиці 5.1.

Щоб отримати пріоритет переривання, HAL визначає таку функцію:

```
void HAL_NVIC_GetPriority(IRQn_Type IRQn, uint32_t
PriorityGroup, uint32_t* pPreemptPriority, uint32_t*
pSubPriority);
```

Поточне групування пріоритетів можна отримати за допомогою такої функції:

```
uint32_t HAL_NVIC_GetPriorityGrouping(void);
```

Налаштування пріоритетів переривання в STM32CubeMX виконують на панелі *NVIC Mode & Configuration* на закладці *NVIC*. Для увімкненого поля таблиці переривань можна задати значення *Preemption Priority* і *Sub Priority*. На Рисунку 5.14 наведено NVIC таблиці переривань з можливістю включення певного переривання ти його пріоритетів.

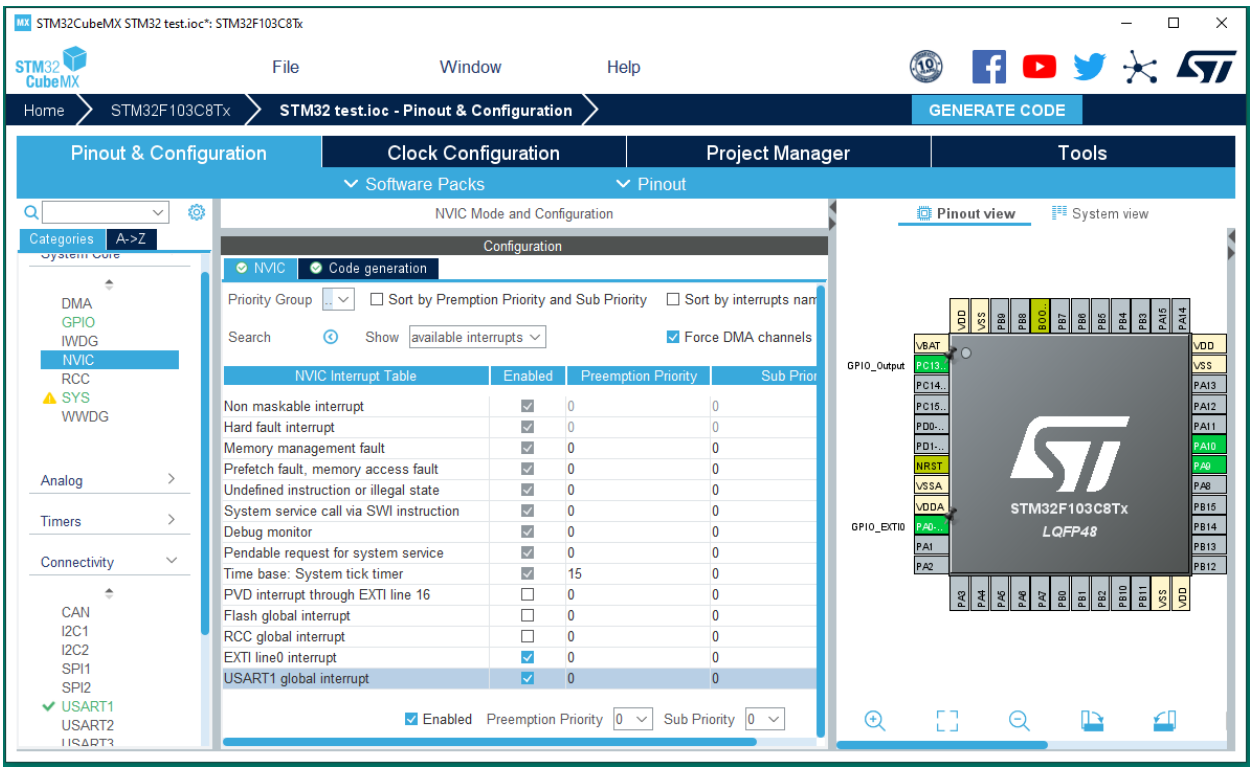

Рисунок 5.14 NVIC таблиці переривань та її налаштування

Використовуючи поле з випадаючим списком *Priority Group*, можна встановити схему групування пріоритетів, а потім призначити окремий пріоритет і підпріоритет кожному перериванню. CubeMX автоматично згенерує відповідний код C для встановлення пріоритету IRQ у функції `MX_GPIO_Init()`. Натомість глобальна схема групування пріоритетів налаштовується всередині функції `HAL_MspInit()`.

Контрольні запитання

1. Які функції можна використати для обробки переривання?
2. У яких станах може знаходитись переривання?
3. Яке переривання виконується першим, якщо два переривання з однаковим пріоритетом очікують?

Лекція 6. Універсальний асинхронний послідовний інтерфейс

Нині в електронній промисловості існує велика кількість протоколів послідовного зв'язку та апаратних інтерфейсів. Більшість із них зосереджені на високій пропускній здатності передачі, як-от новітні стандарти USB 3.0, Firewire (IEEE 1394) тощо. Деякі з цих стандартів прийшли з минулого, але все ще широко поширені, особливо як комунікаційний інтерфейс між модулями на одній платі. Одним з них є інтерфейс універсального синхронного/асинхронного приймача/передавача, також відомий просто як *Universal Synchronous/Asynchronous Receiver/Transmitter* (USART).

Майже кожен мікроконтролер має принаймні один периферійний *Universal Asynchronous Receiver/Transmitter* (UART). Мікроконтролери STM32 забезпечують щонайменше два інтерфейси UART/USART, але більшість із них надають більше двох інтерфейсів (деякі до восьми інтерфейсів) відповідно до кількості вводів-виводів, наявних у корпусі мікроконтролера.

Далі буде показано, як запрограмувати UART за допомогою STM32CubeMX. Крім того, буде розглянуто як розробляти програми з використанням UART як в режимах опитування, так і в режимах переривання.

6.1 Коротко про UART і USART

Для обміну даними між двома (або більше) пристроями, є дві альтернативи: можна передавати дані паралельно, тобто використовуючи задану кількість ліній зв'язку, що дорівнює розміру кожного слова даних (наприклад, вісім незалежних рядків для слова, складеного з восьми бітів), або можна передавати кожен біт, що становить слово, один за іншим. UART/USART – це пристрій, який перетворює паралельну послідовність бітів

(зазвичай згрупованих у байт) у безперервний потік сигналів, що передаються по одному проводу.

Коли інформація передається між двома пристроями всередині спільного каналу, обидва пристрої (і відправник, і одержувач) повинні узгодити час, тобто скільки часу потрібно для передачі кожного окремого біта інформації. При синхронній передачі відправник і одержувач використовують тактові інтервали, створені одним із двох пристроїв (зазвичай пристрій, який виступає в якості ведучого цієї системи з'єднання).

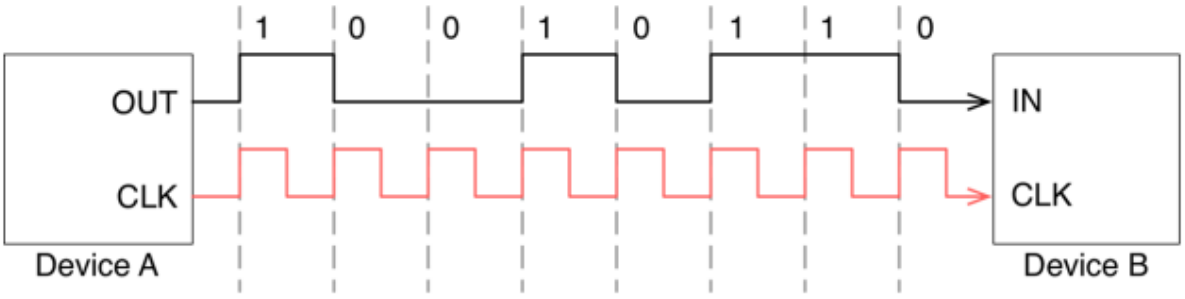

Рисунок 6.1 Послідовний зв'язок між двома пристроями за допомогою спільного джерела тактування

На Рисунку 6.1 наведено типову часову діаграму, яка показує, як Пристрій А відправляє один байт (0b01101001) послідовно до Пристрою В за допомогою спільного тактового сигналу. Тактовий сигнал також використовується для узгодження моменту початку вибірки послідовності бітів: коли головний пристрій видає тактові імпульси, це означає, що він збирається відправити послідовність бітів.

При синхронній передачі швидкість і тривалість передачі визначаються тактовим сигналом: його частота визначає, наскільки швидко можна передати один байт по каналу зв’язку. Але якщо обидва пристрої, які беруть участь у передачі даних, узгодять, скільки часу потрібно для передачі одного біта, а

також коли почати і закінчити вибірку переданих бітів, можна уникнути використання виділеної тактової лінії.

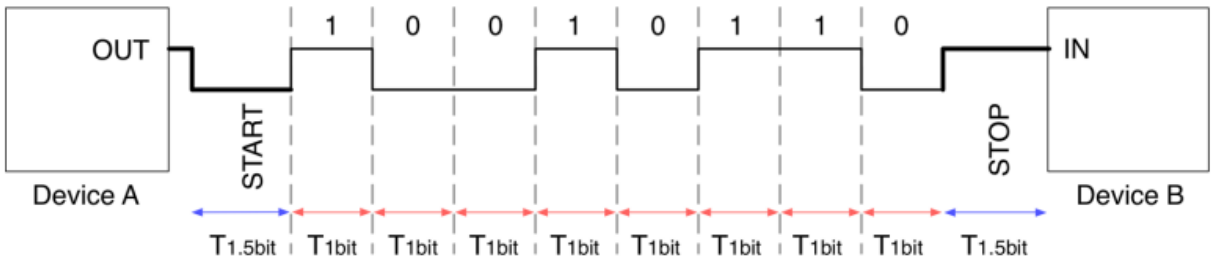

Рисунок 6.2 Часова діаграма послідовного зв'язку без виділеної тактової лінії

На Рисунку 6.2 показана часова діаграма асинхронної передачі. Стан очікування (тобто передача не відбувається) представлена високим сигналом. Передача починається з біта START, який представлений низьким рівнем. Приймач виявляє негативний фронт і 1,5 бітових періодів (T 1.5bit на Рисунку 2) і після цього починається вибірка бітів. Вибирається вісім біт даних. Найменший біт (LSB) зазвичай передається першим. Наприкінці передається необов’язковий біт парності (для перевірки біт даних на помилки). Часто цей біт опускається, якщо передбачається, що канал передачі є безшумним або якщо є перевірка помилок вище на рівнях протоколу. Передача завершується бітом STOP, який триває 1,5 біта.

Універсальний синхронний інтерфейс приймача/передавача – це пристрій, здатний послідовно передавати слова даних, використовуючи два вводу/виводу, один з яких виконує роль передавача (TX), а інший – приймача (RX), а також один додатковий вхід/вихід як одну тактову лінію, як показано на Рисунку 6.3 а). Універсальний асинхронний приймач/передавач використовує лише два RX/TX I/O (див. Рисунок 6.3 б). Традиційно перший інтерфейс називають терміном USART, а другий — терміном UART.

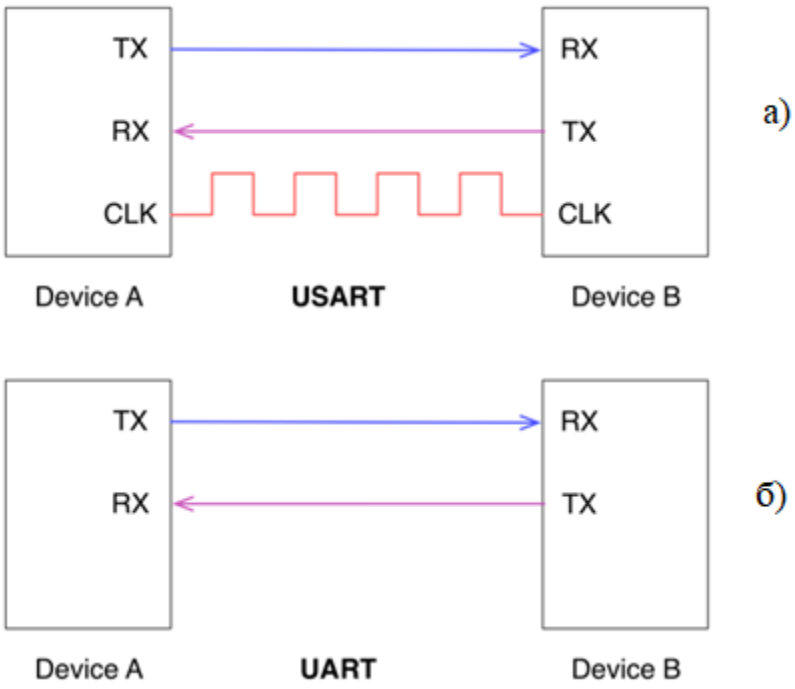

Рисунок 6.3 Різниця в сигналах між USART і UART

UART/USART визначає метод сигналізації, але нічого не говорить про рівні напруги. Це означає, що STM32 UART/USART буде використовувати рівні напруги вводів-виводів MCU, які майже дорівнюють VDD (також ці рівні напруги зазвичай називають рівнями напруги TTL). Спосіб перетворення цих рівнів напруги для забезпечення послідовного зв'язку поза платою вимагає інших стандартів зв'язку. Наприклад, EIA-RS232 або EIA-RS485 — це два популярні стандарти, які визначають напругу сигналізації, на додаток до їхнього часу та значення, а також фізичного розміру та розташування роз'ємів. Крім того, інтерфейси UART/USART можна використовувати для обміну даними за допомогою інших фізичних і логічних послідовних інтерфейсів.

Наявність виділеної тактової лінії або загальна домовленість про частоту передачі не гарантує, що одержувач потоку байтів зможе обробити їх при одній і тій самій передачі від майстра. З цієї причини деякі стандарти зв'язку, такі як RS232 і RS485, надають можливість використовувати виділену лінію

керування потоком обладнання. Наприклад, два пристрої зв'язані за допомогою інтерфейсу RS232 може спільно використовувати дві додаткові лінії, іменовані Request To Send (RTS) і Clear To Send (CTS): відправник встановлює свій RTS, який сигналізує одержувачу почати моніторинг своєї лінії введення даних. Коли одержувач буде готовий до передачі даних, одержувач підніме свою додаткову лінію CTS, яка сигналізує відправнику почати надсилати дані, а відправник розпочне моніторинг лінії виведення даних приймаючого.

Мікроконтролери STM32 мають певну кількість USART, які можна налаштувати на роботу як в синхронному, так і в асинхронному режимі. Деякі мікроконтролери STM32 також забезпечують інтерфейси, здатні діяти лише як UART. Більшість USART також можуть автоматично реалізувати апаратне керування потоком, як для стандартів RS232, так і для RS485.

STM32CubeMX розділяє API для керування інтерфейсами UART та USART. Усі функції та обробники типу C, які використовуються для обробки USART, починаються з префікса HAL_USART і містяться у файлах stm32xxx_hal_usart.{c,h}, а ті, що стосуються керування UART, починаються з префікса HAL_UART і містяться у файлах stm32xxx_hal_uart.{c,h}. Оскільки обидва модулі концептуально ідентичні, і оскільки UART є найпоширенішою формою послідовного взаємозв’язку між різними модулями, далі буде розглянуто лише особливості модуля HAL_UART.

6.2 Ініціалізація UART

Як і всі периферійні пристрої STM32, навіть USART відображаються в відображеній пам’яті периферійній області, яка починається з 0x4000 0000.

STM32CubeMX абстрагує ефективне розташування кожного USART для даного MCU STM32 завдяки дескриптору USART_TypeDef.

Усі функції HAL, пов’язані з керуванням UART, розроблені таким чином, що вони приймають як перший параметр екземпляр структури C UART_HandleTypeDef, яка визначається таким чином:

```
typedef struct {
USART_TypeDef *Instance; /* UART registers base address */
UART_InitTypeDef Init; /* UART communication parameters */
UART_AdvFeatureInitTypeDef AdvancedInit; /* UART Advanced Features initialization parameters */
uint8_t *pTxBuffPtr; /* Pointer to UART Tx transfer Buffer */
uint16_t TxXferSize; /* UART Tx Transfer size */
uint16_t TxXferCount; /* UART Tx Transfer Counter */
uint8_t *pRxBuffPtr; /* Pointer to UART Rx transfer Buffer */
uint16_t RxXferSize; /* UART Rx Transfer size */
uint16_t RxXferCount; /* UART Rx Transfer Counter */
DMA_HandleTypeDef *hdmatx; /* UART Tx DMA Handle parameters */
DMA_HandleTypeDef *hdmarx; /* UART Rx DMA Handle parameters */
HAL_LockTypeDef Lock; /* Locking object */
__IO HAL_UART_StateTypeDef State; /* UART communication state */
__IO HAL_UART_ErrorTypeDef ErrorCode; /* UART Error code */
} UART_HandleTypeDef;
```

Нижче наведено опис основних складових цієї структури:

- `Instance`: це вказівник на дескриптор USART, який буде використовуватися.
- `Init`: є екземпляром структури C UART_InitTypeDef, яка використовується для налаштування інтерфейсу UART.
- `AdvancedInit`: це поле використовується для налаштування більш розширених функцій UART, таких як автоматичне визначення BaudRate і заміна контактів TX/RX. Деякі HAL не передбачають це додаткове поле. Це відбувається тому, що інтерфейси USART не однакові для всіх мікроконтроллерів STM32.
- `pTxBuffPtr i pRxBuffPtr`: ці поля вказують на буфер передачі та прийому відповідно. Вони використовуються як джерело для передачі байтів `TxXferSize` через UART і для отримання `RxXferSize`, коли UART налаштований у повнодуплексному режимі. Поля `TxXferCount` і `RxXferCount` використовуються внутрішньо HAL для підрахунку переданих і отриманих байтів.
- `Lock`: це поле використовується внутрішньо HAL для блокування одночасного доступу різних процесів до інтерфейсів UART.

Усі дії з налаштування UART виконуються за допомогою екземпляра структури C `UART_-InitTypeDef`, яка визначається таким чином:

```
typedef struct {
uint32_t BaudRate;
uint32_t WordLength;
uint32_t StopBits;
uint32_t Parity;
uint32_t Mode;
```

```
uint32_t HwFlowCtl;

uint32_t OverSampling;

} UART_InitTypeDef;
```

- `BaudRate`: цей параметр відноситься до швидкості з'єднання, вираженої в бітах на секунду. Найбільш поширені швидкості передачі даних є 2400, 9600, 19200, 38400, 57600, 115200 б/с та інші.
- `WordLength`: вказує кількість біт даних, що передаються або приймаються в кадрі. Це поле може приймати значення UART_WORDLENGTH_8B або UART_WORDLENGTH_9B, що означає, що можна передавати через UART пакети, що містять 8 або 9 біт даних. Це число не включає передані службові біти, такі як стартовий і кінцевий біти.
- `StopBits`: це поле визначає кількість переданих стоп-бітів. Він може приймати значення UART_STOPBITS_1 або UART_STOPBITS_2, що означає, що можна використовувати один або два стоп-біти для сигналу про кінець кадру.
- `Парність`: вказує на режим парності. Це поле може приймати значення UART_PARITY_NONE, якщо перевірка на парність не ввімкнена. UART_PARITY_EVEN - біт парності встановлюється на 1, якщо кількість бітів, що дорівнює 1, непарна. UART_PARITY_ODD - біт парності встановлюється на 1, якщо кількість бітів, що дорівнює 1, парна. Необхідно звернути увагу, що, коли парність увімкнена, обчислена парність вставляється в позицію MSB переданих даних (9-й біт, якщо довжина слова встановлена на 9 біт даних; 8-й біт, коли довжина слова становить 8 біт даних). Парність — це дуже проста форма перевірки помилок. Він буває двох видів: непарний або парний. Щоб

отримати біт парності, всі біти даних додаються, і від парності суми залежить, встановлено біт чи ні. Парність необов'язкова і не дуже широко використовується. Це може бути корисно для передачі через шумні середовища, але це також трохи уповільнює передачу даних і вимагає від відправника та одержувача впровадити обробку помилок (зазвичай отримані дані, які не вдалися, потрібно надіслати повторно). Коли виникає помилка парності, всі мікроконтролери STM32 генерують конкретне переривання.

- `Режим`: вказує, увімкнено чи вимкнено режим RX або TX. Це поле може приймати одне зі значень: UART_MODE_RX - тільки прийом, UART_MODE_TX – тільки передача, UART_MODE_TX_RX – і прийом, і передача.
- `HwFlowCtl`: вказує, чи увімкнено чи вимкнено режим керування апаратним потоком RS232. Цей параметр може приймати одне зі значень: UART_HWCONTROL_NONE, UART_HWCONTROL_RTS, UART_HWCONTROL_CTS, UART_HWCONTROL_RTS_CTS.
- `OverSampling`: коли UART отримує кадр від віддаленого однорангового пристрою, він відбирає сигнали, щоб обчислити кількість 1 і 0, що становлять повідомлення. Поле OverSampling може приймати значення UART_OVERSAMPLING_16 для виконання 16 вибірок для кожного біта кадру або UART_OVERSAMPLING_8 для виконання 8 вибірок. Це дозволяє знайти компроміс між максимальною швидкістю зв’язку та захищеністю від шуму/неточності тактового генератора.

6.3 Робота UART в режимі опитування

Мікроконтролери STM32, а отже, і CubeHAL пропонують три способи обміну даними між пристроями використовуючи UART: опитування, переривання та режим DMA. Ці режими не лише три різні варіанти обробки данних по UART. Вони є трьома різними підходами програмування до одного завдання, які дають кілька переваг як з точки зору дизайну, так і з точки зору продуктивності. Нижче коротко про кожний.

- У режимі опитування, який також називають режимом блокування, основна програма або один з її потоків синхронно очікує на передачу та отримання даних. Це найпростіша форма передачі даних за допомогою цього периферійного пристрою, і її можна використовувати, коли швидкість передачі не є занадто низькою і коли UART не використовується як критична периферія в програмі (класичним прикладом є використання UART як вихідної консолі для налагодження).
- У режимі переривання, який також називають неблокуючим режимом, основна програма звільняється від очікування завершення передачі та отримання даних. Процедури передачі даних припиняються, щойно вони закінчуються. Коли передача даних закінчиться, переривання повідомить про це основний код. Цей режим більше підходить, коли швидкість зв’язку низька (нижче 38400 біт/с) або коли передача даних відбувається «рідко», порівняно з іншими діями, які виконує MCU, і немає часу на очікування передачі або прийому даних.
- Режим DMA забезпечує найкращу пропускну здатність передачі даних завдяки прямому доступу периферійного пристрою UART до внутрішньої оперативної пам'яті MCU. Цей режим найкраще підходить

для високошвидкісного зв’язку та коли необхідно звільнити MCU від витрат часу на передачу даних. Без режиму DMA майже неможливо досягти найшвидшої швидкості передачі, яку може обробляти периферійний пристрій USART.

Для передачі послідовності байтів через USART в режимі опитування HAL використовується наступна функція:

```
HAL_StatusTypeDef HAL_UART_Transmit(UART_HandleTypeDef *huart, uint8_t *pData, uint16_t Size, uint32_t Timeout);
```

де:

- `huart`: це вказівник на екземпляр структури UART_HandleTypeDef, яку бачили раніше, яка ідентифікує та налаштовує периферійний пристрій UART;
- `pData`: вказівник на масив, довжина якого дорівнює параметру `Size`, що містить послідовність байтів, які необхідно передати;
- `Тайм-аут`: максимальний час, виражений в мілісекундах, протягом якого очікується завершення передачі. Якщо передача не завершується за вказаний час очікування, функція переривається і повертає значення HAL_TIMEOUT; інакше він повертає значення HAL_OK, якщо інших помилок не виникає. Крім того, можна передати тайм-аут, рівний HAL_MAX_DELAY (0xFFFF FFFF), щоб необмежено чекати завершення передачі.

Для отримання послідовності байтів через USART в режимі опитування HAL використовується наступна функція:

```
HAL_StatusTypeDef HAL_UART_Receive(UART_HandleTypeDef *huart, uint8_t *pData, uint16_t Size, uint32_t Timeout);
```

де:

- `huart`: це вказівник на екземпляр структури UART_HandleTypeDef, яка ідентифікує та налаштовує периферійний пристрій UART;
- `pData`: це вказівник на масив, довжина якого принаймні дорівнює параметру Size, що містить послідовність байтів для прийому. Функція буде блокуватися, доки не будуть отримані всі байти, визначені параметром *Size*.
- `Тайм-аут`: максимальний час, виражений в мілісекундах, протягом якого очікується завершення отримання. Якщо передача не завершується за вказаний час очікування, функція переривається і повертає значення HAL_TIMEOUT; інакше він повертає значення HAL_OK, якщо інших помилок не виникає. Крім того, можна передати тайм-аут, рівний HAL_MAX_DELAY (0xFFFF FFFF), щоб необмежено чекати завершення отримання.

6.4 Робота з UART в режимі переривання

У режимі опитування MCU по суті заблокований, очікуючи вхідних даних (значення тайм-ауту HAL_MAX_DELAY блокує `HAL_UART_Receive()`, доки один символ не буде надіслано через UART). Це впливає на виконання інших процесів у режимі реального часу.

У цьому випадку можна вказати набагато коротший тайм-аут функції `HAL_UART_Receive()` щоб інші критичні задачі могли виконуватися. Однак це може призвести до втрати важливих даних, що надходять від периферійного пристрою UART (інтерфейс UART має буфер шириною в один байт).

Для вирішення цієї проблеми HAL пропонує інший спосіб обміну даними через периферійний UART: режим переривання. Щоб скористатися цим режимом, необхідно виконати наступні завдання:

- Увімкнути переривання `USARTx_IRQn` і реалізувати відповідний `USARTx_IRQHandler()ISR`.
- Викликати `HAL_UART_IRQHandler()` всередині `USARTx_IRQHandler()`: це виконуватиме всі дії, пов'язані з керуванням перериваннями, які генеруються периферійним пристроєм UART.
- Використовувати функції `HAL_UART_Transmit_IT()` і `HAL_UART_Receive_IT()` для обміну даними через UART. Ці функції також включають режим переривання периферійного пристрою UART: таким чином периферійний пристрій підтвердить відповідний рядок у контролері NVIC, щоб ISR піднімався, коли відбувається подія.
- Змінити код програми для роботи з асинхронними подіями. Для цього поглянути на доступні переривання UART і на те, як розроблено підпрограми HAL.

Кожен периферійний пристрій STM32 USART забезпечує переривання, наведені в таблиці 6.1. Ці переривання включають як IRQ, пов'язані з передачею даних, так і з помилками зв'язку. Їх можна розділити на дві групи:

- IRQ, які генеруються під час передачі: передача завершена, очищення для відправки або переривання порожнього регістру даних.
- IRQ, які генеруються під час прийому: виявлення простою лінії, помилка переповнення, регістр даних прийому не порожній, помилка парності, виявлення розриву зв'язку, прапорець шуму (тільки при багатобуферному зв'язку) та помилка кадрування (тільки при багатобуферному зв'язку).

Таблиця 6.1 Список переривань, пов'язаних з USART

Interrupt Event	Event Flag	Enable Control Bit
Transmit Data Register Empty	TXE	TXEIE
Clear To Send (CTS) flag	CTS	CTSIE
Transmission Complete	TC	TCIE
Received Data Ready to be Read	RXNE	RXNEIE
Overrun Error Detected	ORE	RXNEIE
Idle Line Detected	IDLE	IDLEIE
Parity Error	PE	PEIE
Break Flag	LBD	LBDIE
Noise Flag, Overrun error and Framing Error in multi buffer communication	NF or ORE or FE	EIE

Ці події генерують переривання, якщо встановлено відповідний біт керування дозволом (третій стовпець таблиці 6.1). Однак мікроконтролери STM32 розроблені таким чином, що всі ці IRQ прив’язані лише до одного ISR для кожного периферійного пристрою USART. Наприклад, USART2 визначає тільки USART2_-IRQn як IRQ для всіх переривань, створених цим периферійним пристроєм. Код користувача має проаналізувати відповідний прапор події, щоб визначити, яке переривання генерувало запит.

CubeHAL призначений для автоматичного виконання цієї роботи. Користувач попереджається про генерацію переривань завдяки серії функцій зворотного виклику, викликаних `HAL_UART_IRQHandler()`, які повинні бути викликані всередині ISR.

З технічної точки зору, немає такої великої різниці між передачею UART в опитуванні і в режимі переривання. Обидва методи передають масив байтів за допомогою *UART DataRegister* (DR) за таким алгоритмом:

- Для передачі даних необхідно помістити байт в регістр USART->DR і зачекати, доки прапор *TransmitData Register Empty* (TXE) не стане істинним.
- Для отримання даних необхідно зачекати, доки не буде підтверджено, що отримані дані готові до читання (RXNE), а потім зберегти вміст регістру USART->DR в пам'яті програми.

Різниця між двома методами полягає в тому, як вони чекають завершення передачі даних. У режимі опитування розроблені функції `HAL_UART_Receive() / HAL_UART_Transmit()`, які чекають встановлення відповідного прапора події для кожного байта, який потрібно прийняти або передати. У режимі переривань функції `HAL_UART_Receive_IT() / HAL_UART_Transmit_IT()` розроблені так, що вони не чекають завершення передачі даних, а завантажують новий байт в регістр DR, або завантажують його вміст у пам'ять програми. Це виконується за допомогою процедури ISR, коли генерується переривання RXNEIE / TXEIE.

Щоб передати послідовність байтів у режимі переривання, HAL визначає функцію:

```
HAL_StatusTypeDef HAL_UART_Transmit_IT (UART_HandleTypeDef *huart, uint8_t *pData, uint16_t Size);
```

де:

- `huart`: це вказівник на екземпляр структури `UART_HandleTypeDef`, яка ідентифікує та налаштовує периферійний пристрій UART;
- `pData`: це вказівник на масив, довжина якого дорівнює параметру Size, що містить послідовність байтів, які необхідно передати; функція не

блокуватиме очікування на передачу даних, і вона передає керування в основний потік, як тільки завершиться налаштування UART.

Для отримання послідовності байтів через USART в режимі переривання HAL надає функцію:

```
HAL_StatusTypeDef HAL_UART_Receive_IT(UART_HandleTypeDef *huart, uint8_t *pData, uint16_t Size);
```

де:

- `huart`: це вказівник на екземпляр структури UART_HandleTypeDef, яку бачили раніше, яка ідентифікує та налаштовує периферійний пристрій UART;
- `pData`: це вказівник на масив, довжина якого принаймні дорівнює параметру Size, що містить послідовність байтів, які ми збираємося отримати. Функція не блокуватиме очікування отримання даних і передає управління основному потоку, щойно завершиться налаштування UART.

6.5 Програмування UART в STM32CubeMX та IAR EW

У режимі опитування дані передаються і приймаються з блокуванням, тобто процесор блокуватиме всі інші операції, поки передача або прийом даних не буде завершена. Цей метод добре використовувати, якщо використовується лише UART і нічого більше, інакше виконання всіх інших операцій буде затримуватись.

Вікно налаштувань інтерфейсу UART в STM32CubeMX наведено на рисунку 6.4. На закладці *Pinout & Configuration* на панелі *Categories* в категорії

System Core зі списку потрібно вибрати потрібний порт (для прикладу USART1). На панелі *USART1 Mode & Configuration* у вікні *Mode* потрібно вибрати режим *Asynchronous*.

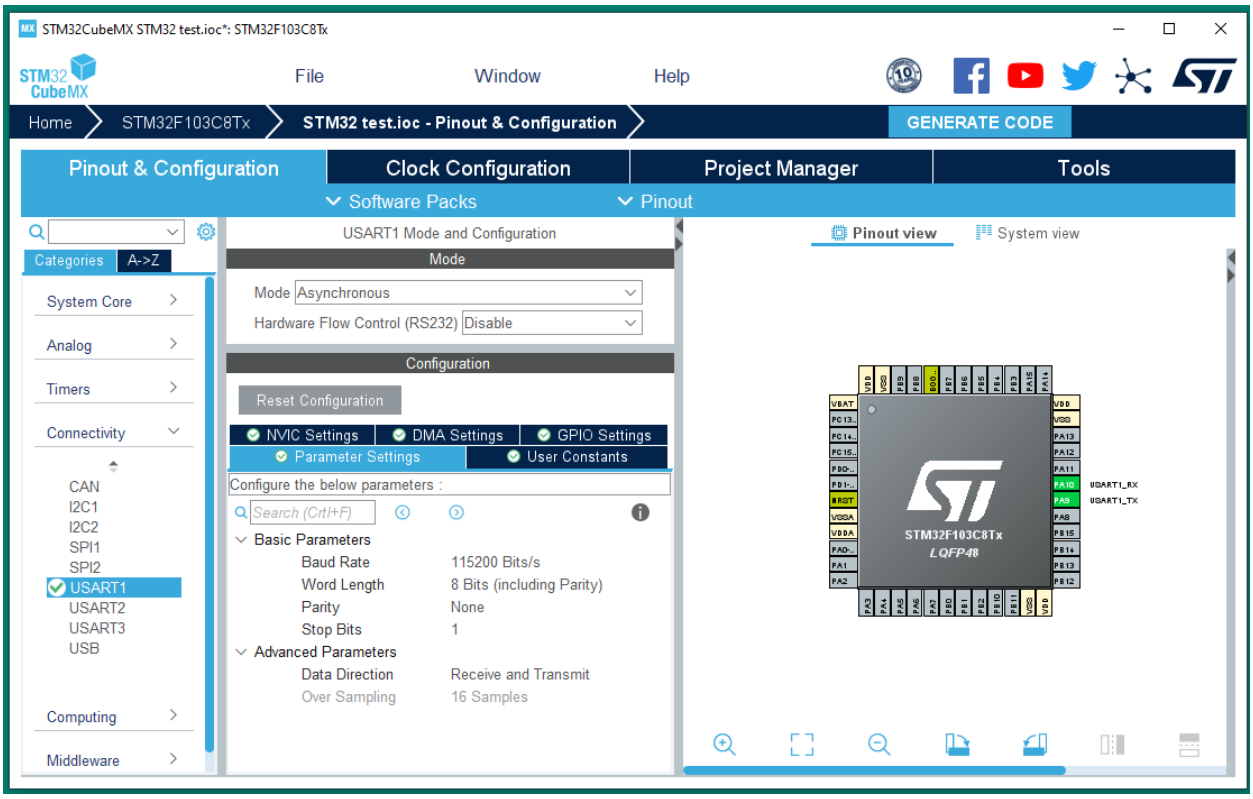

Рисунок 6.4 Налаштування параметрів USART

Після генерації коду в файлі `main.c`, в функції `main(void)` з'являється функція ініціалізації USART1 `MX_GPIO_Init()`, яка містить задані для нього в STM32CubeMX налаштування. Тепер в циклі можна розмістити простий код прийому та передачі даних в режимі опитування. Час очікування завершення операції становить 10 мілісекунд:

```
uint8_t Tx_data[] = "HELLO WORLD \r\n";
uint8_t Rx_data[10];  //  буфер з 10 байтів
int main(void)
{
  /* Reset of all peripherals, Initializes the Systick. */
  HAL_Init();
```

```
  /* Configure the system clock */
  SystemClock_Config();
  /* Initialize all configured peripherals */
  MX_USART1_UART_Init();
  /* USER CODE BEGIN WHILE */
  while (1)
  {
     // передача рядка
     HAL_UART_Transmit (&huart1, Tx_data, sizeof (Tx_data), 10);
     // прийом 4-х байтів даних
     HAL_UART_Receive (&huart1, Rx_data, 4, 10);
     HAL_Delay (250);  // 250 ms delay
    /* USER CODE END WHILE */
  }
}
```

У режимі переривання передача відбувається в режимі без блокування або у фоновому режимі. Тож решта процесів працює без затримок. Коли передача даних завершена, викликається функція завершення передачі *Tx Complete Callback,* яка містить код того, що потрібно робити після завершення передачі. Прийом у режимі переривання відбувається також у фоновому режимі. Таким чином, прийом даних практично не впливає на інші процеси. Коли прийом даних завершено, викликається функція завершення прийому *Rx Complete Callback*, де можна написати код того, що потрібно робити після завершення прийому.

Вікно налаштувань інтерфейсу UART в STM32CubeMX з вибором режиму переривання наведено на рисунку 6.5. На панелі *USART1 Mode & Configuration* у вікні *Configuration* потрібно вибрати закладку *NVIC Settings* і поставити чек *Enabled* для глобального переривання UART1.

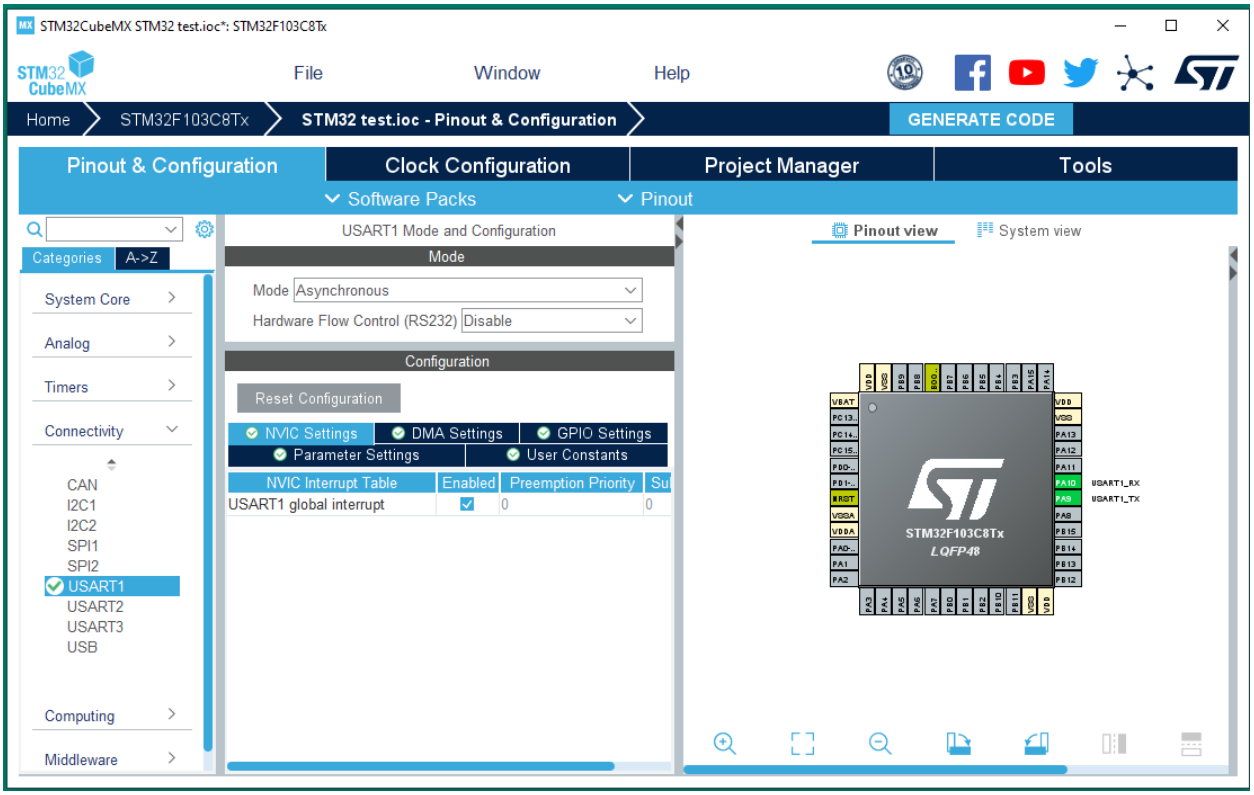

Рисунок 6.5 Включення режиму переривання для USART1

Після генерації коду STM32CubeMX додає в файл переривань `stm32f1xx_it.c` функцію переривань `USART1_IRQHandler()` і надає можливість викликати `Callback` функції, які розташовані в файлі `stm32f1xx_hal_uart.c`.

Скорочений варіант файлу `main.c` наведено нижче. Тепер основний цикл програми пустий і вся комунікація відбувається у перериваннях. Після закінчення прийому визивається функція `HAL_UART_RxCpltCallback` яка, як приклад, змінює стан світлодіода. Після закінчення передачі визивається функція `HAL_UART_TxCpltCallback` яка також змінює стан світлодіода.

```
uint8_t Rx_data[10];  //  creating a buffer of 10 bytes
uint8_t Tx_data[] = "HELLO WORLD \r\n";


void HAL_UART_RxCpltCallback(UART_HandleTypeDef *huart)
{
  HAL_GPIO_TogglePin(GPIOC, GPIO_PIN_13);
```

```
}
void HAL_UART_TxCpltCallback(UART_HandleTypeDef *huart)
{
   HAL_GPIO_TogglePin(GPIOC, GPIO_PIN_13);
}
main ()
{
  MX_USART1_UART_Init();
  HAL_UART_Receive_IT (&huart1, Rx_data, 4);
  HAL_UART_Transmit_IT(&huart1, Tx_data, sizeof (Tx_data));
  while (1)
  {
  }
}
```

Контрольні запитання

1. Що таке UART/USART? Для чого використовуються?
2. Чим відрізняються способи обміну даними між пристроями у режимі опитування та у режимі переривання використовуючи UART?
3. Яким чином можна визначити яке саме переривання мало місце при роботі UART у режимі переривання?

Лекція 7. Інтерфейс I²C

Сучасні друковані плати містить крім мікроконтролера дві або більше цифрових інтегральних схем (IC), призначених для виконання конкретних завдань. АЦП і ЦАП, пам'ять EEPROM, датчики, логічні порти вводу/виводу, тактовий генератор RTC, радіочастотні схеми та PKI-контролери — це лише невеликий список можливих мікросхем, які спеціалізуються на виконанні лише одного завдання. Сучасний дизайн цифрової електроніки – це правильний вибір (і програмування) потужних, специфічних і, в більшості випадків, дешевих мікросхем для розміщення на друкованій платі.

Залежно від характеристик цих мікросхем, вони часто призначені для обміну повідомленнями та даними з програмованим пристроєм (який зазвичай є, але не обмежуючись, мікроконтролером) відповідно до чітко визначеного протоколу зв’язку. Два з найбільш використовуваних протоколів для зв’язку в межах плати – I²C і SPI, обидва датуються початком 80-х, але все ще широко поширені в електронній промисловості, особливо коли швидкість зв’язку не є суворою вимогою і обмежена межами плати.

Майже всі мікроконтролери STM32 забезпечують спеціалізовані апаратні периферійні пристрої, здатні спілкуватися за допомогою протоколів I²C і SPI. Нижче буде розглянуто протокол I²C і пов’язані API CubeHAL для програмування відповідних периферійних пристроїв.

7.1 Загальні відомості про інтерфейс I²C

Inter-Integrated Circuit (він же I²C - вимовляється I-квадрат-C або дуже рідко I-two-C) - це специфікація обладнання та протоколу, розробленого підрозділом напівпровідників Philips (тепер NXP Semiconductors) у 1982 році. Це напівдуплексна, одностороння 8-розрядна орієнтована послідовна шина,

яка використовує лише два дроти для з'єднання великої кількості підпорядкованих пристроїв із ведучим.

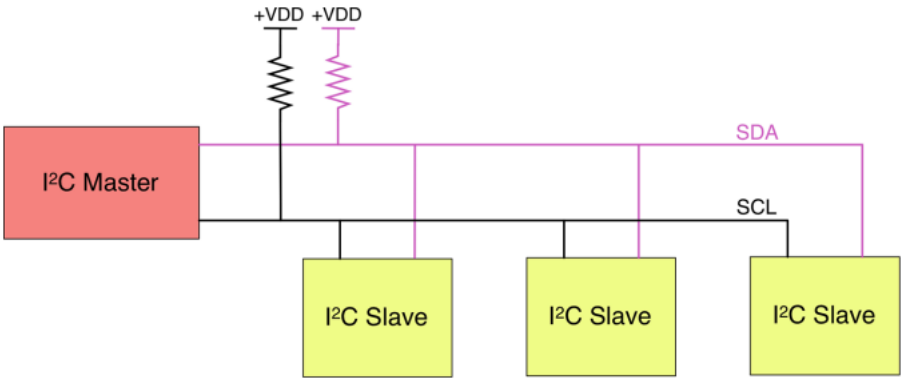

Рисунок 7.1 Графічне зображення шини I²C

Два проводи, що утворюють шину I²C, є двонаправленими лініями з відкритим стоком, які називаються послідовними лініями передачі даних *Serial Data Line* (SDA) і послідовними тактовими лініями *Serial Clock Line* (SCL) відповідно (див. Рисунок 7.1). Протокол I²C визначає, що ці дві лінії потрібно підтягнути до напруги живлення за допомогою резисторів. Номінал цих резисторів безпосередньо пов'язаний з ємністю шини та швидкістю передачі. Однак найбільш поширеним є використання резисторів зі значенням 4,7 кОм.

Оскільки протокол заснований лише на двох проводах, то має бути спосіб адресації окремого підпорядкованого пристрою на одній шині. З цієї причини I²C визначає, що кожен підпорядкований пристрій надає унікальну адресу підлеглого для даної шини[7]. Адреса може мати ширину 7 або 10 біт (цей останній варіант досить рідкісний).

Швидкість шини I²C чітко визначена специфікацією протоколу, хоча мікросхеми, здатні працювати з іншою швидкістю. Звичайними швидкостями шини I²C є 100 кГц, також відомий як стандартний режим, і 400 кГц, відомий

як швидкий режим. Останні редакції стандарту можуть працювати на більших швидкостях (1 МГц, відомий як швидкий режим плюс, і 3,4 МГц, відомий як високошвидкісний режим, і 5 МГц, відомий як надшвидкий режим).

Протокол I²C є досить простим протоколом, тому MCU може «симулювати» виділений периферійний пристрій I²C, якщо він його не має: ця техніка називається біт-бангом, і вона зазвичай використовується в дійсно недорогих 8-розрядних архітектурах, які не забезпечують виділений інтерфейс I²C для зменшення кількості контактів та/або вартості IC.

7.2 Особливості реалізації протоколу I²C

У протоколі I²C всі транзакції завжди ініціюються та завершуються ведучим. Це одне з небагатьох правил цього протоколу зв'язку, яке слід пам'ятати під час програмування (і, особливо, налагодження) пристроїв I²C. Усі повідомлення, якими обмінюються через шину I²C, розбиваються на два типи кадрів: кадр адреси, де головне вказує, якому підпорядкованому пристрою надсилається повідомлення, і один або кілька кадрів даних, які є 8-бітовими повідомленнями даних, що передаються від головного до підпорядкованого або навпаки. Дані розміщуються в рядку SDA після того, як SCL стає низьким, і вибірка здійснюється після того, як лінія SCL стає високою. Час між фронтами тактового сигналу та зчитуванням/записом даних визначається пристроями на шині і варіюється від мікросхеми до мікросхеми.

Як було сказано раніше, і SDA, і SCL є двонаправленими лініями, підключеними до позитивної напруги живлення через джерело струму або підтягуючий резистор (див. Рисунок 7.1). Коли шина вільна, обидві лінії знаходяться у високому стані. Вихідні каскади пристроїв, підключених до шини, повинні мати відкритий стік або відкритий колектор для виконання

функції провідного I. Ємність шини обмежує кількість приладів, підключених до неї. Для одномастерного варіанту вихід SCL може бути двотактним, якщо на шині немає пристроїв.

Нижче проаналізовано основні етапи комунікації по I²C.

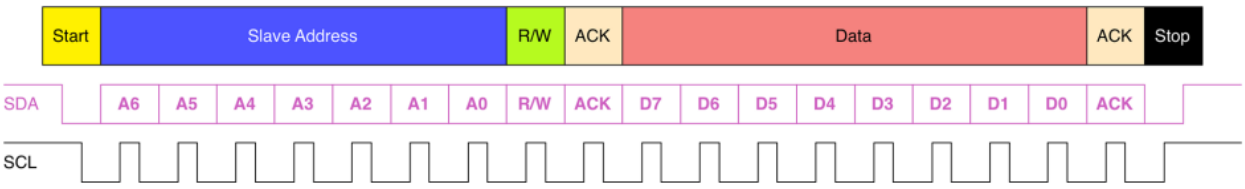

Рисунок 7.2 Структура базового повідомлення I²C

Усі транзакції починаються зі START і завершуються зупинкою (див. Рисунок 7.2). Перехід HIGH до LOW на лінії SDA, коли SCL є HIGH, визначає стан START. Перехід від LOW до HIGH на лінії SDA, коли SCL є HIGH, визначає стан STOP.

Стани START та STOP завжди генеруються ведучим. Шина вважається зайнятою після стану START. Через певний час після стану STOP шина вважається знову вільною. Шина залишається зайнятою, якщо генерується повторний START (також званий умовою RESTART) замість стану STOP. У цьому випадку стани START і RESTART функціонально ідентичні.

Кожне слово, передане по лінії SDA, повинно мати вісім біт, і це також включає кадр адреси. Кількість байтів, які можуть бути передані за передачу, необмежена. За кожним байтом має слідувати біт підтвердження (ACK). Дані передаються спочатку з найбільш значущим бітом (MSB) (див. Рисунок 7.2). Якщо підлеглий не може отримати або передати ще один повний байт даних, поки він не виконає іншу функцію, наприклад, обслуговує внутрішнє переривання, він може утримувати тактовий рядок SCL LOW, щоб примусово

перевести ведучого у стан очікування. Потім передача даних продовжується, коли підпорядкований пристрій готовий отримати інший байт даних і звільняє тактовий рядок SCL.

Адресний кадр завжди є першим у будь-якій новій послідовності зв'язку. Для 7-розрядної адреси спочатку тактується старший біт (MSB), а потім біт R/W, який вказує, чи це операція читання (1) чи запису (0) (див. Рисунок 7.2).

Рисунок 7.3 Структура повідомлення у випадку, якщо використовується 10-бітова адресація

У 10-розрядній системі адресації (див. рисунок 7.3) для передачі веденої адреси потрібні два кадри. Перший кадр складатиметься з коду 1111 0XXD 2, де XX — це два біти MSB 10-бітової адреси підпорядкованого пристрою, а D — біт R/W, як описано вище. Перший біт ACK кадру буде підтверджено всіма підлеглими, які відповідають першим двом бітам адреси. Як і у випадку звичайної 7-бітової передачі, одразу починається інша передача, і ця передача містить біти [7:0] адреси. У цей момент адресований ведений повинен відповісти бітом ACK. Якщо це не так, режим відмови такий же, як і 7-розрядній системі.

Необхідно зауважити, що 10-розрядні адресні пристрої можуть співіснувати з 7-розрядними адресними пристроями, оскільки провідна частина адреси 11110 не є частиною жодної дійсної 7-розрядної адреси.

Підтвердження ACK відбувається після кожного байту. Біт ACK дозволяє одержувачу сигналізувати передавачеві про те, що байт було успішно отримано, і може бути надісланий інший байт. Майстер генерує всі тактові імпульси по лінії SCL, включаючи дев’ятий тактовий імпульс ACK.

Сигнал ACK визначається наступним чином: передавач звільняє лінію SDA під час тактового імпульсу підтвердження, щоб приймач міг зробити рівень лінії SDA LOW, і він залишається стабільним LOW протягом періоду HIGH цього тактового імпульсу. Коли SDA залишається ВИСОКИМ протягом цього дев’ятого тактового імпульсу, це визначається як сигнал не підтвердження (NACK). Майстер може потім сформувати умову STOP, щоб перервати передачу, або умову RESTART, щоб почати нову передачу. Існує п'ять умов, що призводять до створення NACK:

1. На шині з переданою адресою немає приймача, тому немає пристрою, який би відповів підтвердженням.

2. Приймач не може приймати або передавати, оскільки виконує певну функцію в режимі реального часу і не готовий розпочати зв'язок з ведучим.

3. Під час передачі одержувач отримує дані або команди, які він не розуміє.

4. Під час передачі одержувач не може отримати більше байтів даних.

5. Ведучий-приймач повинен повідомити про закінчення передачі веденому передавачу.

Після відправлення кадру адреси можна почати передачу даних. Майстер просто продовжуватиме генерувати тактові імпульси на SCL через регулярні інтервали, а дані будуть розміщені на SDA або ведучим, або підпорядкованим, залежно від того, чи вказує біт R/W операцію читання чи

запису. Зазвичай перший або перші два байти містять адресу підпорядкованого регістру для запису/читання. Наприклад, для I²C EEPROM перші два байти після кадру адреси представляють адресу місця в пам'яті, залученого до транзакції.

Залежно від біта R/W наступні байти заповнюються ведучим (якщо біт R/W встановлено на 1) або підпорядкованим (якщо біт R/W дорівнює 0). Кількість кадрів даних є довільною, і більшість підпорядкованих пристроїв автоматично збільшують внутрішній регістр, що означає, що наступне читання або запис буде здійснюватися з наступного регістра в рядку. Цей режим також називають послідовним або пакетним режимом (див. Рисунок 7.4), який дозволяє прискорити швидкість передачі.

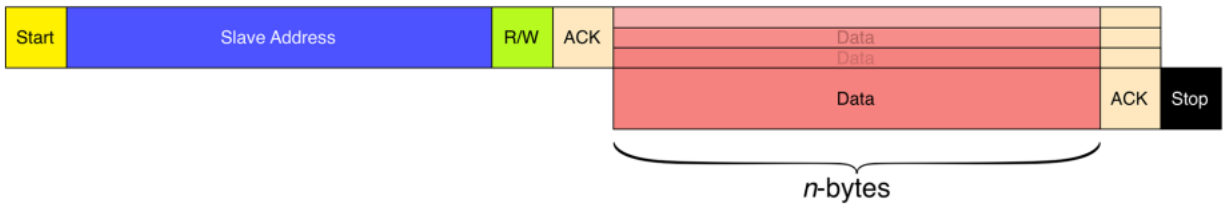

Рисунок 7.4 Передача в пакетному режимі, коли в одній транзакції обмінюються кількома байтами

Протокол I²C, по суті, має просту схему зв'язку:

- провідний передає на шину адресу підпорядкованого пристрою, який бере участь у транзакції;
- біт R/W, який є розрядом LSB в байті адреси підпорядкованого пристрою, встановлює напрямок потоку даних (від провідного до підпорядкованого - W - або від підпорядкованого до провідного - R)

- певна кількість байтів надсилається, кожен з яких перемежовується бітом ACK, одним із двох партнерів відповідно до напряму передачі, доки не виникне умова STOP.

Цей алгоритм зв’язку має особливість: якщо потрібно прочитати довільну адресу пам’яті для підпорядкованого пристрою, то потрібно використовувати дві окремі транзакції. Наприклад, є I²C EEPROM. Щоб отримати вміст якоїсь комірки пам’яті, майстер повинен виконати такі дії:

- розпочати транзакцію в режимі запису (останній біт адреси підпорядкованого пристрою встановлено на 0), відправивши адресу підпорядкованого пристрою на шину I²C, щоб EEPROM почала вибірку повідомлень по шині;
- відправити два байти, що представляють адресу комірки пам'яті, яку потрібно прочитати;
- завершити транзакцію, надіславши умову STOP;
- розпочати нову транзакцію в режимі читання (останній біт адреси підпорядкованого пристрою встановлено на 1), надіславши адресу веденого на шину I²C;
- прочитати n-байтів (зазвичай один, якщо зчитувати пам'ять у випадковому режимі, більше одного, якщо читати її в послідовному режимі), надіслані підпорядкованим пристроєм, а потім завершити транзакцію з умовою STOP.

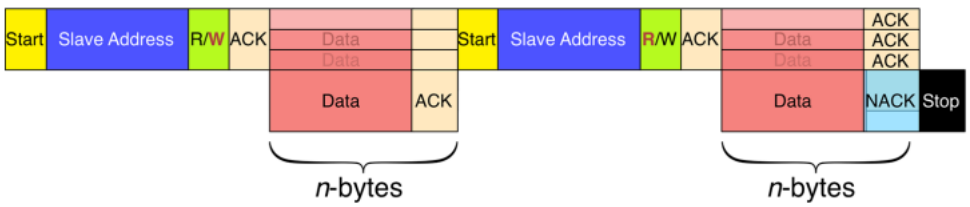

Рисунок 7.5 Структура комбінованої транзакції

Для підтримки цієї загальної схеми зв’язку протокол I²C визначає об’єднані транзакції, де напрямок потоку даних інвертується (зазвичай від підпорядкованого до головного або навпаки) після передачі певної кількості байтів. На Рисунку 7.5 схематизується цей спосіб зв'язку з підлеглими пристроями. Ведучий починає надсилати адресу підпорядкованого пристрою в режимі запису, а потім надсилає адреси регістрів, які ми хочемо прочитати. Потім надсилається нова умова START, не припиняючи транзакцію: ця додаткова умова START також називається повторною умовою START (або RESTART). Ведучий знову надсилає адресу підпорядкованого пристрою, але цього разу транзакція починається в режимі читання. Тепер підлеглий передає вміст бажаних регістрів, а ведучий підтверджує кожен надісланий байт. Майстер завершує транзакцію, видаляючи NACK (це дійсно важливо, як ми побачимо далі) та умову STOP.

Залежно від типу сімейства та використовуваного пакета, мікроконтролери STM32 можуть забезпечувати до чотирьох незалежних периферійних пристроїв I²C. Далі розглянуто програмну реалізацію інтерфейсу I²C.

7.3 Модуль HAL_I2C

Для програмування периферійного пристрою I²C CubeHAL визначає структуру C I2C_HandleTypeDef, яка визначається таким чином:

```
typedef struct {
I2C_TypeDef *Instance; /* I²C registers base address */
I2C_InitTypeDef Init; /* I²C communication parameters */
uint8_t *pBuffPtr; /* Pointer to I²C transfer buffer */
uint16_t XferSize; /* I²C transfer size */
```

```
__IO uint16_t XferCount; /* I²C transfer counter */
DMA_HandleTypeDef *hdmatx; /* I²C Tx DMA handle parameters */
DMA_HandleTypeDef *hdmarx; /* I²C Rx DMA handle parameters */
HAL_LockTypeDef Lock; /* I²C locking object */
__IO HAL_I2C_StateTypeDef State; /* I²C communication state */
__IO HAL_I2C_ModeTypeDef Mode; /* I²C communication mode */
__IO uint32_t ErrorCode; /* I²C Error code */
} I2C_HandleTypeDef;
```

Призначення основних полів наведеної С-структури.

- `Instance`: це вказівник на дескриптор I²C, який буде використано. Наприклад, I2C1 є дескриптором першого периферійного пристрою I²C.
- `Init`: є екземпляром структури С `I2C_InitTypeDef`, яка використовується для початкового налаштування периферійного пристрою.
- `pBuffPtr`: вказівник на внутрішній буфер, який використовується для тимчасового зберігання даних, переданих до та з периферійного пристрою I²C. Використовується, коли I²C працює в режимі переривання, і його не слід змінювати з коду користувача.
- `hdmatx`, `hdmarx`: вказівник на екземпляри структури `DMA_HandleTypeDef`, що використовується, коли периферійний пристрій I²C працює в режимі DMA.

Налаштування периферійного пристрою I²C виконується за допомогою екземпляра структури С I2C_InitTypeDef, яка визначається таким чином:

```
typedef struct {
```

```
uint32_t ClockSpeed; /* Specifies the clock frequency */

uint32_t DutyCycle; /* Specifies the I²C fast mode duty cycle. */

uint32_t OwnAddress1; /* Specifies the first device own address.*/

uint32_t OwnAddress2; /* Specifies the second device own address
                         if dual addressing mode is selected */

uint32_t AddressingMode; /* Specifies if 7-bit or 10-bit
                            addressing mode is selected. */

uint32_t DualAddressMode; /* Specifies if dual addressing mode is
                             selected. */

uint32_t GeneralCallMode; /* Specifies if general call mode is
                             selected. */

uint32_t NoStretchMode; /* Specifies if nostretch mode is
                           selected. */

} I2C_InitTypeDef;
```

Опис полів структури `I2C_InitTypeDef`.

- `ClockSpeed`: це поле визначає швидкість інтерфейсу I²C і має відповідати швидкості шини, визначеній у специфікаціях I²C (стандартний режим, швидкий режим тощо). Максимальне значення для цього поля для більшості мікроконтролерів STM32 становить 400 000 (400 кГц), що означає, що мікроконтролери STM32 можуть підтримувати швидкий режим. MCU STM32F0/F3/F7/L0/L4 підтримують також швидкий режим плюс (1 МГц).
- `DutyCycle`: це поле може приймати значення I2C_DUTYCYCYCLE_2 та I2C_DUTYCYCYCLE_16_9, щоб вказати робочий цикл, що дорівнює 2:1 і 16:9. Вибравши заданий тактовий режим, можна попередньо

масштабувати периферійні тактові імпульси для досягнення бажаної тактової частоти I²C.

- `OwnAddress1`, `OwnAddress2`: периферійний пристрій I²C у мікроконтролерах STM32 можна використовувати для розробки як ведучих, так і підпорядкованих пристроїв I²C. При розробці підпорядкованих пристроїв I²C поле `OwnAddress1` дозволяє вказати адресу підпорядкованого I²C: периферійний пристрій автоматично визначає дану адресу на шині I²C і автоматично запускає всі пов'язані події (наприклад, він може генерувати відповідне переривання, щоб почати нову транзакцію на шині). Периферійний пристрій I²C підтримує 7- або 10-бітну адресацію, а також 7-розрядний режим подвійної адресації: у цьому випадку можна вказати дві різні 7-бітові адреси підпорядкованого пристрою, щоб пристрій міг відповідати на запити, надіслані на обидві адреси.
- `AddressingMode`: це поле може приймати значення I2C_ADDRESSINGMODE_7BIT або I2C_ADDRESS-INGMODE_10BIT для визначення 7- або 10-бітового режиму адресації відповідно.
- `DualAddressMode`: це поле може приймати значення I2C_DUALADDRESS_ENABLE або I2C_DUALAD-DRESS_DISABLE, щоб увімкнути/вимкнути 7-розрядний режим подвійної адресації.
- `GeneralCallMode`: загальний виклик – це свого роду широкомовна адресація в протоколі I²C. Спеціальна адреса підпорядкованого пристрою I²C 0x0000 000 використовується для надсилання повідомлення всім пристроям на одній шині. Загальний виклик є додатковою функцією, і, встановивши для цього поля значення I2C_GENERALCALL_ENABLE, периферійний пристрій I²C генеруватиме події, коли буде збігатися загальна адреса виклику.

- `NoStretchMode`: це поле, яке може приймати значення I2C_NOSTRETCH_ENABLE або I2C_-NOSTRETCH_DISABLE, використовується для вимкнення/включення додаткового режиму розтягування тактових інтервалів. Даний режим тут не розглядається.

Як зазвичай, для налаштування периферійного пристрою I²C використовують функцію:

```
HAL_StatusTypeDef HAL_I2C_Init(I2C_HandleTypeDef *hi2c);
```

яка приймає вказівник на екземпляр I2C`_HandleTypeDef`, розглянутий раніше.

7.4 Використання пристрою I²C в режимі Master

Щоб виконати транзакцію через шину I²C в режимі запису, CubeHAL надає функцію:

```
HAL_StatusTypeDef HAL_I2C_Master_Transmit(I2C_HandleTypeDef *hi2c, uint16_t DevAddress, uint8_t *pData, uint16_t Size, uint32_t Timeout);
```

де:

- `hi2c`: це вказівник на екземпляр структури `I2C_HandleTypeDef`, яка ідентифікує периферійний пристрій I²C;
- `DevAddress`: це адреса підпорядкованого пристрою, яка може мати 7 або 10 біт в залежності від конкретної мікросхеми;
- `pData`: це вказівник на масив, довжина якого дорівнює параметру `Size`, що містить послідовність байтів, які необхідно передати;
- `Timeout`: максимальний час, виражений в мілісекундах, очікування завершення передачі. Якщо передача не завершується за вказаний час очікування, функція переривається і повертає значення HAL_TIMEOUT;

інакше він повертає значення HAL_OK, якщо інших помилок не виникає. Крім того, можна передати тайм-аут, рівний HAL_MAX_DELAY (0xFFFFFFFF), щоб необмежено чекати завершення передачі.

Щоб виконати транзакцію в режимі читання, можна використовувати замість цього таку функцію:

```
HAL_StatusTypeDef HAL_I2C_Master_Receive(I2C_HandleTypeDef *hi2c, uint16_t DevAddress, uint8_t *pData, uint16_t Size, uint32_t Timeout);
```

Обидві попередні функції виконують транзакцію в режимі опитування. Для транзакцій на основі переривань можна використовувати функції:

```
HAL_StatusTypeDef HAL_I2C_Master_Transmit_IT(I2C_HandleTypeDef *hi2c, uint16_t DevAddress, uint8_t *pData, uint16_t Size);
```

```
HAL_StatusTypeDef HAL_I2C_Master_Receive_IT(I2C_HandleTypeDef *hi2c, uint16_t DevAddress, uint8_t *pData, uint16_t Size);
```

Ці функції працюють так само, як і функції передачі UART в режимі переривання. Щоб їх правильно використовувати, потрібно ввімкнути відповідний ISR і викликати підпрограму `HAL_I2C_EV_IRQHandler()`, яка, у свою чергу, викликає `HAL_I2C_MasterTxCpltCallback(I2C_HandleTypeDef *hi2c)` щоб повідомити про завершення передачі в режимі запису та `AL_I2C_MasterTxCpltCallback(I2C_HandleTypeDef *hi2c)` щоб повідомити про закінчення передачі в режимі читання. За винятком сімейств STM32F0 і TM32L0, периферійний пристрій I²C у всіх MCU STM32 використовує окреме переривання для сигналізації про умови помилки. З цієї причини у відповідному ISR потрібно викликати `HAL_I2C_ER_IRQHandler()`,

який, у свою чергу, викликає `HAL_I2C_ErrorCallback(I2C_HandleTypeDef *hi2c)` у разі помилки. `I2Cx_EV_IRQHandler()` є ISR, який викликає зворотний виклик. Існує десять різних зворотних викликів, які викликає CubeHAL, нижче наведено деякі з них:

- `HAL_I2C_MasterTxCpltCallback()` – сигналізує про те, що передача від головного до підпорядкованого завершена;
- `HAL_I2C_MasterRxCpltCallback()` – сигналізує про те, що передача від підпорядкованого до ведучого завершена;
- `HAL_I2C_MemTxCpltCallback()` – сигналізує про завершення передачі від головного пристрою в зовнішню пам'ять;
- `HAL_I2C_MemRxCpltCallback()` – сигналізує про завершення передачі від зовнішньої пам’яті до головного пристрою;
- `HAL_I2C_ErrorCallback()` – сигналізує про те, що сталася помилка

Нарешті, функції:

```
HAL_StatusTypeDef   HAL_I2C_Master_Transmit_DMA(I2C_HandleTypeDef *hi2c,uint16_t DevAddress, uint8_t *pData, uint16_t Size);
```

```
HAL_StatusTypeDef    HAL_I2C_Master_Receive_DMA(I2C_HandleTypeDef *hi2c, uint16_t DevAddress, uint8_t *pData, uint16_t Size);
```

дозволяють виконувати транзакції I²C за допомогою DMA.

7.5 Програмування I²C в STM32CubeMX та IAR EW

Вікно налаштувань інтерфейсу I²C в STM32CubeMX наведено на рисунку 7.6. На закладці *Pinout & Configuration* на панелі *Categories* в категорії System Core зі списку потрібно вибрати потрібний інтерфейсу I²C (для

прикладу I2C1). На панелі I2C1 *Mode & Configuration* у вікні *Mode* потрібно вибрати I2C. Також для майстра можна вибрати режим швидкості та значення швидкості.

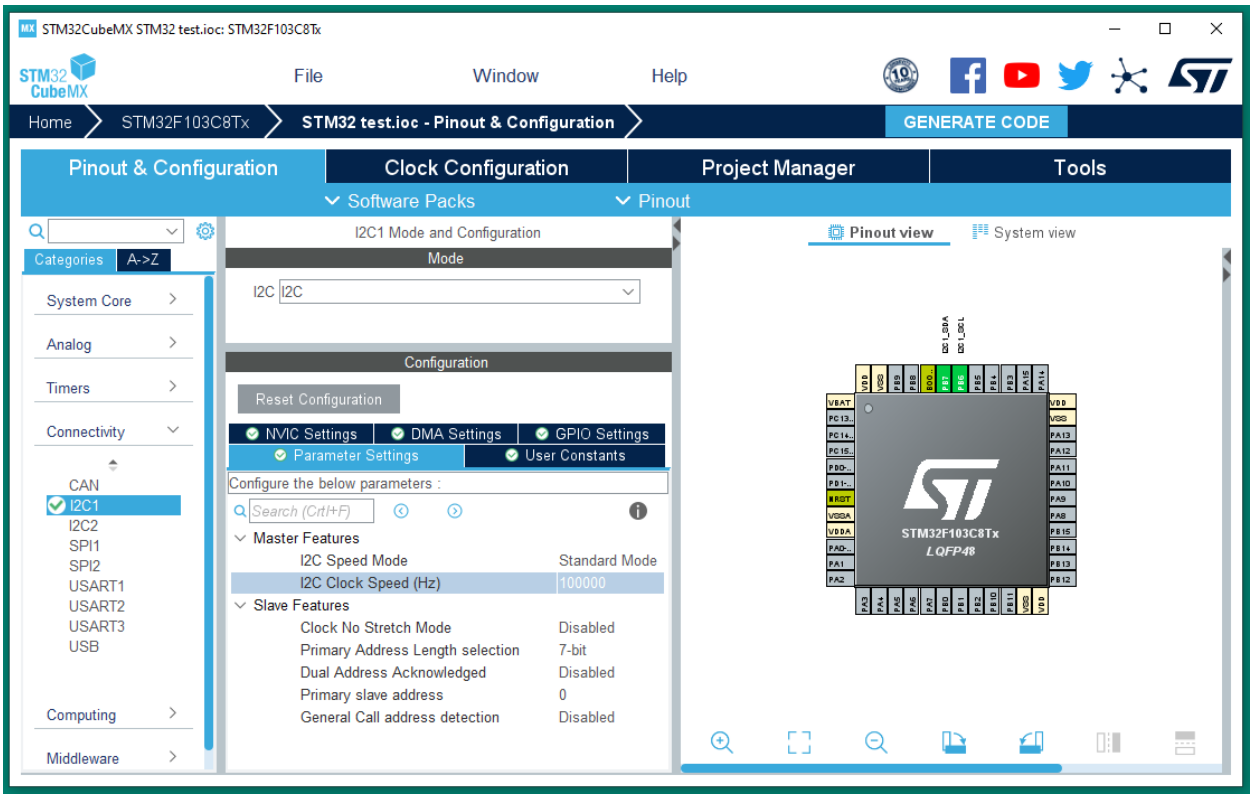

Рисунок 7.6. Налаштування інтерфейсу I²C в STM32CubeMX

Перший приклад це використання універсальних функцій передачі та прийому даних інтерфейсу I²C. Такі функції можна застосовувати для роботи з усіма пристроями, де є I²C. У даному прикладі зчитується і передається в послідовний порт температура з цифрового датчика TMP102 від Texas Instruments. Початкові налаштування згенеровано в STM32CubeMX.

```
static const uint8_t TMP102_ADDR = 0x48 << 1; // Use 8-bit address
static const uint8_t REG_TEMP = 0x00;
int main(void) {
HAL_StatusTypeDef ret;
```

```
uint8_t buf[12];
int16_t val;
float temp_c;

HAL_Init();
SystemClock_Config();
MX_USART2_UART_Init();
MX_I2C1_Init();

  while (1)
  {
    // Передача адреси регістра температури датчика TMP102
    buf[0] = REG_TEMP;
    ret = HAL_I2C_Master_Transmit(&hi2c1, TMP102_ADDR, buf, 1,
HAL_MAX_DELAY);
    if ( ret != HAL_OK ) {
      strcpy((char*)buf, "Error Tx\r\n");
    } else {

      // Read 2 bytes from the temperature register
      ret = HAL_I2C_Master_Receive(&hi2c1, TMP102_ADDR, buf, 2,
HAL_MAX_DELAY);
      if ( ret != HAL_OK ) {
        strcpy((char*)buf, "Error Rx\r\n");
      } else {
        // Формування даних для виводу
        val = ((int16_t)buf[0] << 4) | (buf[1] >> 4);
        // Врахування мінусової температури
        if ( val > 0x7FF ) {
          val |= 0xF000;
        }
```

```
        // Перетворення в float значення температури по Цельсію
        temp_c = val * 0.0625;
        // Перетворення температури в десятковий формат
        temp_c *= 100;
        sprintf((char*)buf, "%u.%u C\r\n",
              ((unsigned int)temp_c / 100),
              ((unsigned int)temp_c % 100));
      }
    }
    // Надіслати буфер (температура або повідомлення про помилку)
    HAL_UART_Transmit(&huart2,      buf,      strlen((char*)buf),
HAL_MAX_DELAY);
    // Wait
    HAL_Delay(500);
  }
}
```

Плата STM 32 Smart V2 містить мікросхему пам’яті AT24C04 з інтерфейсом I²C. Тому у другому прикладі використані спеціальні функції `HAL_I2C_Mem_Read()` та `HAL_I2C_Mem_Write()` для зчитування та запису даних в мікросхеми пам’яті, які мають інтерфейс I²C. Ці функції є в бібліотеці `stm32f1xx_hal_i2c.c`. У даному прикладі в пам’ять записується тестовий рядок, а потім зчитується. Якщо записані і зчитані дані співпадають, то на платі буде мигати світлодіод.

```
int main(void) {
char wmsg[] ="Test string";
char rmsg[20];

HAL_Init();
```

```
MX_I2C1_Init();

HAL_I2C_Mem_Write(&hi2c1,  0xA0,  0x1AAA,  I2C_MEMADD_SIZE_16BIT,
(uint8_t*)wmsg, 24 strlen(wmsg)+1, HAL_MAX_DELAY);

while(HAL_I2C_IsDeviceReady(&hi2c1,  0xA0,  1,  HAL_MAX_DELAY)  !=
HAL_OK);


HAL_I2C_Mem_Read(&hi2c1,  0xA0,  0x1AAA,  I2C_MEMADD_SIZE_16BIT,
(uint8_t*)rmsg,28 strlen(wmsg)+1, HAL_MAX_DELAY);

if(strcmp(wmsg, rmsg) == 0) {

while(1) {

HAL_GPIO_TogglePin(GPIOC, GPIO_PIN_13);

HAL_Delay(100);

}

}

while(1);

}
```

Контрольні запитання

1. Призначення та характеристики інтерфейсу I²C?
2. Які умови призводять до створення сигналу NACK?
3. Опишіть алгоритм зв’язку по інтерфейсу I²C.
4. Як налаштувати I²C в STM32CubeMX?

Лекція 8. Інтерфейс SPI

Усі мікроконтролери STM32 мають принаймні один інтерфейс SPI, що дозволяє розробляти як провідні, так і підпорядковані програми. CubeHAL реалізує всі необхідні елементи для легкого програмування таких периферійних пристроїв. У цьому розділі дається короткий огляд модуля HAL_SPI після, як зазвичай, короткого введення в специфікацію SPI.

8.1 Опис специфікації SPI

Послідовний периферійний інтерфейс (SPI) — це специфікація послідовного, синхронного та дуплексного зв'язку між головним контролером і кількома підлеглими пристроями. Інтерфейс SPI дозволяє здійснювати як повний, так і напівдуплексний зв’язок по одній шині. Типова шина SPI формується чотирма сигналами, як показано на Рисунку 8.1.

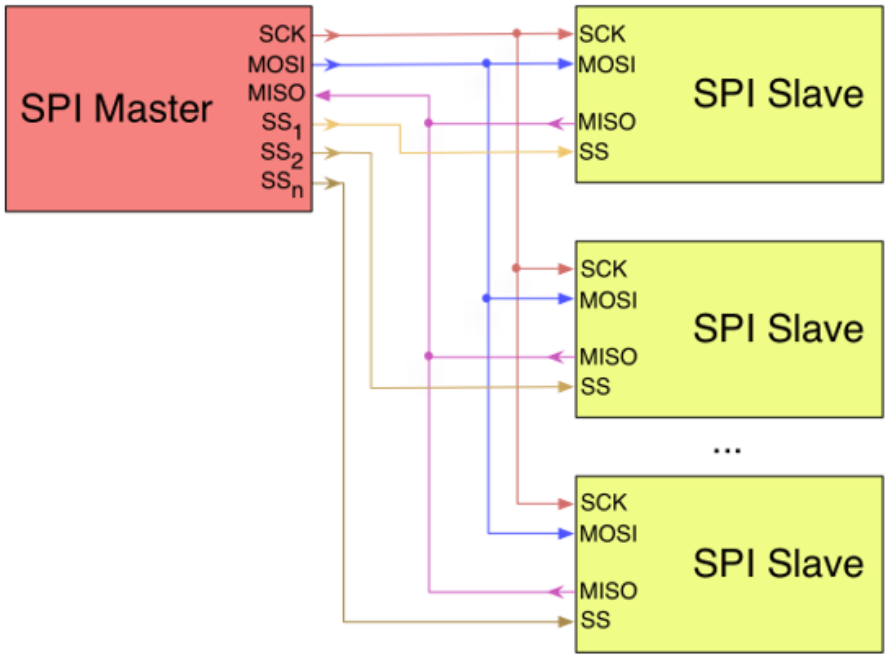

Рисунок 8.1 Структура типової шини SPI

Специфікація SPI є стандартом де-факто, вона була визначена компанією Motorola наприкінці 1970-х років, і все ще використовується як протокол зв'язку для багатьох цифрових мікросхем. На відміну від протоколу I²C, специфікація SPI обмежується сигналами шини, і надає підлеглим пристроям свободу щодо структури повідомлень. Основні сигнали шини SPI:

- SCK: цей сигнал введення-виведення використовується для генерації годинника для синхронізації передачі даних по шині SPI. Він генерується головним пристроєм, а це означає, що в шині SPI кожна передача завжди починається ведучим. На відміну від специфікації I²C, SPI по суті швидше, а тактова частота SPI зазвичай становить кілька МГц. Сьогодні досить часто зустрічаються пристрої SPI, здатні обмінюватися даними зі швидкістю до 100 МГц. Крім того, протокол SPI дозволяє пристроям з різною швидкістю зв'язку співіснувати на одній шині.
- MOSI: назва цього сигнального I/O означає *Master Output Slave Input*, і він використовується для передачі даних від головного пристрою до підпорядкованого.
- MISO: це означає *Master Input Slave Output* і відповідає лінії вводу/виводу, яка використовується для надсилання даних від підпорядкованого пристрою до ведучого.
- SSn: це означає *Slave Select*, і в типовій шині SPI існують «n» відокремлених ліній, які використовуються для адресації конкретних пристроїв SPI, які беруть участь у транзакції. На відміну від протоколу I²C, SPI не використовує адреси підпорядкованих пристроїв для вибору пристроїв, але він вимагає цієї операції на фізичній лінії, для якої визначено LOW, щоб виконати вибір. У типовій шині SPI тільки один

підпорядкований пристрій може бути активним одночасно, маючи низький рівень його лінії SS. Це причина, чому пристрої з різною швидкістю зв'язку можуть співіснувати на одній шині.

Маючи дві відокремлені лінії передачі даних, MOSI і MISO, SPI по суті дозволяє повнодуплексний зв'язок, оскільки підпорядкований пристрій може надсилати дані до ведучого, тоді як він отримує від нього нові. У шині SPI «один до одного» (тільки один провідний і один підпорядкований) сигнал SS можна не використовувати.

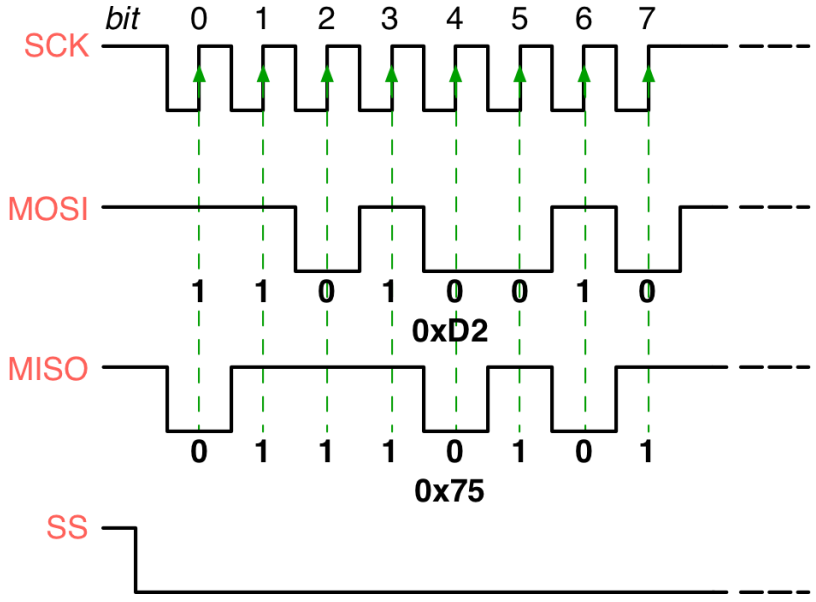

Рисунок 8.2 Обмін даними через шину SPI при дуплексній передачі

Кожна транзакція по шині починається з переводу лінії SS в стан LOW та роботи лінії SCK. При передачі зазвичай включаються два регістри заданого розміру слова, один у ведучого та один у веденого. Дані зазвичай зміщуються першим із найбільш значущим бітом, а новий біт зсувається в той же регістр. У той же час дані з підпорядкованого пристрою дані зміщуються в регістр з найменшого значущого біта. Після того, як біти регістра були зсунуті та введені, ведучий і підпорядкований обмінюються даними. Якщо потрібно

обміняти більше даних, регістри зсуву перезавантажуються, і процес повторюється. Передача може тривати протягом будь-якої кількості тактів. Після завершення ведучий припиняє перемикання тактового сигналу і, як правило, знімає вибір підлеглого.

На Рисунку 8.2 показано, як дані передаються при повнодуплексній передачі, а на Рисунку 8.3 показано, як вони зазвичай передаються в напівдуплексному з'єднанні.

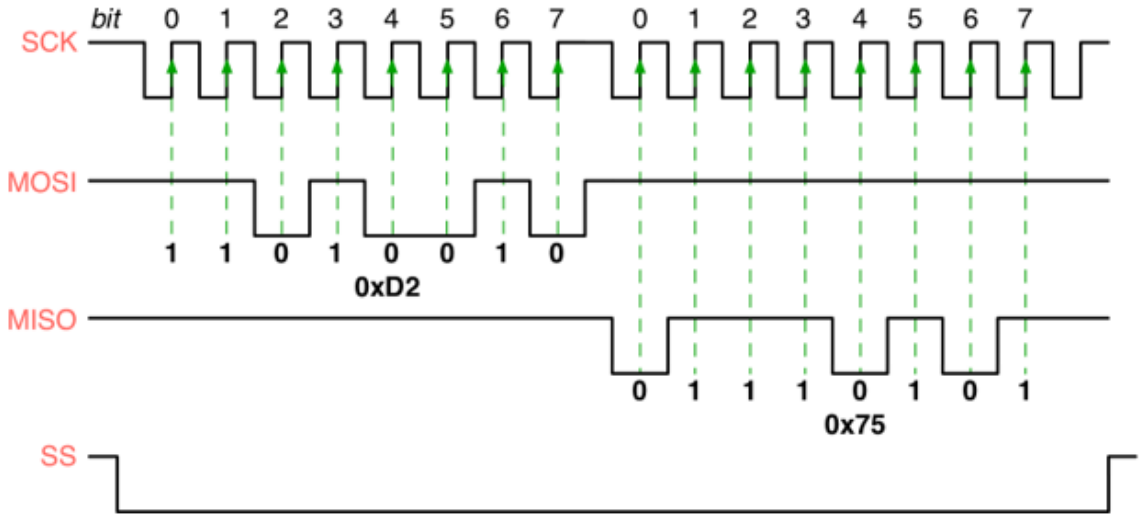

Рисунок 8.3 Обмін даними по шині SPI при напівдуплексній передачі

8.2 Налаштування сигналів інтерфейсу SPI

На додаток до встановлення тактової частоти шини, ведучий і підлеглий також повинні узгодити полярність і фазу тактування щодо даних, якими обмінюються по лініях MOSI і MISO. Специфікація SPI від Motorola[4] називає ці два параметри CPOL і CPHA відповідно, і більшість постачальників кремнію прийняли цю конвенцію.

Комбінації полярності та фази часто називають режимами шини SPI, які зазвичай нумеруються відповідно до таблиці 1. Найпоширенішими режимами

є режим 0 і режим 3, але більшість ведених пристроїв підтримують принаймні кілька режимів шини.

Таблиця 8.1 Режими шини SPI відповідно до конфігурації CPOL і CPHA

Mode	CPOL	CPHA
0	0	0
1	0	1
2	1	0
3	1	1

Часова діаграма відповідно до налаштувань CPOL і CPHA показана на Рисунку 8.4, далі вона описана нижче:

- При CPOL=0 базове значення годинника дорівнює нулю, тобто активний стан дорівнює 1, а стан очікування дорівнює 0.
  - Для CPHA=0 дані фіксуються на передньому фронті SCK (перехід LOW → HIGH), а дані виводяться за спадним фронтом (перехід HIGH → LOW).
  - Для CPHA=1 дані фіксуються за спадним фронтом SCK, а дані виводяться за наростаючим фронтом.
- При CPOL=1 базове значення годинника дорівнює одиниці (інверсія CPOL=0), тобто активний стан дорівнює 0, а стан очікування дорівнює 1.
  - Для CPHA=0 дані фіксуються за спадним фронтом SCK, а дані виводяться за наростаючим фронтом.
  - Для CPHA=1 дані фіксуються за переднім фронтом SCK, а дані виводяться за спадним фронтом.

Тобто CPHA=0 означає вибірку на першому фронті тактової частоти, тоді як CPHA=1 означає вибірку на другому фронті тактової частоти, незалежно від того, зростає цей фронт чи спадає. Зауважте, що при CPHA=0 дані повинні бути стабільними протягом півперіоду перед першим тактовим циклом.

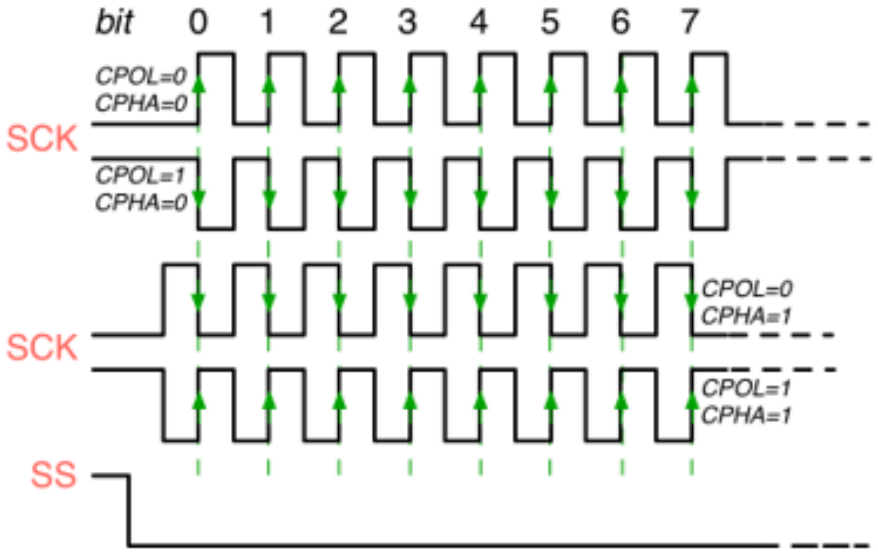

Рисунок 8.4 Часова діаграма SPI відповідно до налаштувань CPOL і CPHA

Підпорядковані пристрої SPI не мають адреси, яка ідентифікує їх на шині, але вони ведуть обмін даними з ведучим до тих пір, поки сигнал вибору підпорядкованого пристрою (SS) низький. Мікроконтролери STM32 забезпечують два різних режими для обробки сигналу SS, який в документації ST називається NSS:

- Програмний режим NSS: сигнал SS управляється програмним забезпеченням, і будь-який вільний GPIO може використовуватися для керування IC, коли MCU працює в режимі ведучого, або для виявлення,

коли інший головний пристрій починає передачу, якщо MCU працює у веденому режимі.

- Апаратний режим NSS: для керування сигналом SS використовується певний вивід MCU, який внутрішньо керується периферійним пристроєм SPI. Залежно від конфігурації виходу NSS можливі дві конфігурації:
  - Вихід NSS увімкнено: ця конфігурація використовується лише тоді, коли пристрій працює в головному режимі. Сигнал NSS передається на LOW рівень, коли головне розпочинає зв'язок, і зберігається LOW, доки SPI не буде вимкнено. Важливо зауважити, що цей режим підходить, коли на шині є лише один підпорядкований пристрій SPI і його вхід SS підключений до сигналу NSS. Ця конфігурація не дозволяє режим з кількома майстрами.
  - Вихід NSS вимкнено: ця конфігурація дозволяє використовувати декілька головних пристроїв для пристроїв, що працюють у режимі головного. Для пристроїв, встановлених як підпорядковані, контакт NSS діє як класичний вхід NSS: підпорядкований вибирається, коли NSS має значення LOW, і знімається, коли NSS HIGH.

Периферійні пристрої SPI в мікроконтролерах STM32 підтримують режим TI під час роботи в головному режимі і коли сигнал NSS налаштований на роботу в апаратній частині. У режимі TI полярність і фаза тактового сигналу мають відповідати вимогам протоколу Texas Instruments, незалежно від встановлених значень. Керування NSS також є специфічним для протоколу TI, що робить конфігурацію керування NSS прозорою для користувача. У режимі

TI, фактично, сигнал NSS змінюється в кінці кожного переданого байту (він переходить від LOW до HIGH від початку біта LSB і переходить від HIGH до LOW на початку біта MSB, формуючи наступний переданий байт).

8.3 Програмний модуль HAL_SPI

Для програмування периферійного пристрою SPI HAL визначає структуру `SPI_HandleTypeDef`, яка визначається таким чином:

```
typedef struct___SPI_HandleTypeDef {
 SPI_TypeDef        *Instance;
 SPI_InitTypeDef    Init;
 uint8_t            *pTxBuffPtr;
 uint16_t           TxXferSize;
 ____IO uint16_t    TxXferCount;
 uint8_t            *pRxBuffPtr;
 uint16_t           RxXferSize;
 ____IO uint16_t    RxXferCount;
 DMA_HandleTypeDef  *hdmatx;
 DMA_HandleTypeDef  *hdmarx;
 HAL_LockTypeDef    Lock;
 ____IO HAL_SPI_StateTypeDef  State;
 ____IO uint32_t    ErrorCode;
} SPI_HandleTypeDef;
```

Найважливіші поля цієї структури:

- `Instance`: це вказівник на дескриптор SPI, який буде використано. Наприклад, SPI1 є дескриптором першого периферійного пристрою SPI.
- `Init`: є екземпляром структури `SPI_InitTypeDef`, яка використовується для налаштування периферійного пристрою.

- pTxBuffPtr, pRxBuffPtr: вказівники на внутрішні буфери, які використовуються для тимчасового зберігання даних, переданих до та з периферійного пристрою SPI.
- hdmatx, hdmarx: показники на екземпляри структури DMA_HandleTypeDef, що використовуються, коли периферійний пристрій SPI працює в режимі DMA.

Налаштування периферійного пристрою SPI виконується за допомогою екземпляра структури SPI_InitTypeDef, яка визначається таким чином:

```
typedef struct {
 uint32_t Mode;
 uint32_t Direction;
 uint32_t DataSize;
 uint32_t CLKPolarity;
 uint32_t CLKPhase;
 uint32_t NSS;
 uint32_t BaudRatePrescaler;
 uint32_t FirstBit;
 uint32_t TIMode;
 uint32_t CRCCalculation;
 uint32_t CRCPolynomial;
} SPI_InitTypeDef;
```

- Mode (Режим): цей параметр встановлює SPI в режимі Master або Slave. Він може приймати значення SPI_MODE_- MASTER і SPI_MODE_SLAVE.
- Direction: вказує, що підпорядкована периферійна система працює в 4-провідному (дві відокремлені лінії для введення/виводу) або 3-

провідному (лише одна лінія для введення/виводу). Він може приймати значення SPI_DIRECTION_2LINES, щоб налаштувати повнодуплексний 4-провідний режим; значення SPI_DIRECTION_2LINES_RXONLY для налаштування напівдуплексного 4-провідного режиму; значення SPI_DIRECTION_1LINE, щоб налаштувати напівдуплексний 3-провідний режим.

- `DataSize`: налаштовує розмір даних, що передаються по шині SPI, і може приймати значення SPI_DATASIZE_8BIT і SPI_DATASIZE_16BIT.
- `CLKPolarity`: налаштовує параметр SCK CPOL і може приймати значення SPI_PO-LARITY_LOW (що відповідає CPOL=0) і SPI_POLARITY_HIGH (що відповідає CPOL=1).
- `CLKPhase` це поле встановлює фазу тактування, і воно може приймати значення SPI_PHASE_1EDGE (що відповідає CPHA=0) і SPI_PHASE_2EDGE (що відповідає CPHA=1).
- `NSS`: це поле визначає поведінку NSS I/O. Він може приймати значення SPI_NSS_SOFT для налаштування сигналу NSS в програмному режимі; значення SPI_NSS_HARD_INPUT і SPI_NSS_HARD_- OUTPUT для налаштування сигналу NSS в апаратному режимі входу та виходу відповідно.
- `BaudRatePrescaler`: він встановлює попередній масштаб тактової частоти APB і встановлює максимальну тактову швидкість SCK. Він може приймати значення SPI_BAUDRATEPRESCALER_2, SPI_BAUDRATEPRESCALER_4,
- …, SPI_BAUDRATEPRESCALER_256 (усі два ступеня від 2^1 до 2^8).

- `FirstBit`: визначає порядок передачі даних і може приймати значення SPI_FIRST-BIT_MSB і SPI_FIRSTBIT_LSB.
- `TIMode`: використовується для ввімкнення/вимкнення режиму TI і може приймати значення SPI_TIMODE_- DISABLE та SPI_TIMODE_ENABLE.
- `CRCCalculation` та `CRCPolynomial`: периферійний пристрій SPI у всіх мікроконтролерах STM32 підтримує генерацію CRC в апаратному забезпеченні. Значення CRC може передаватися як останній байт у режимі Tx, або автоматична перевірка помилок CRC може бути виконана для останнього отриманого байта. Значення CRC обчислюється за допомогою непарного програмованого полінома для кожного біта. Розрахунок обробляється на фронті тактового сигналу вибірки, визначеному конфігураціями CPHA та CPOL. Розраховане значення CRC перевіряється автоматично в кінці блоку даних, а також для передачі, керованої ЦП або DMA. Коли виявляється невідповідність між CRC, обчисленим внутрішньо на отриманих даних, і CRC, надісланим передавачем, встановлюється умова помилки. Функція CRC недоступна, коли SPI керується в цикловому режимі DMA.

Для ініціалізації периферійного пристрою SPI використовується функція:

```
HAL_StatusTypeDef HAL_SPI_Init(SPI_HandleTypeDef *hspi);
```

яка приймає вказівник на екземпляр структури `SPI_HandleTypeDef`, описану раніше.

8.4 Обмін повідомленнями між пристроями SPI

Після налаштування периферійного пристрою SPI можна почати обмін даними з веденими пристроями. Оскільки специфікація SPI не вимагає використання певного протоколу зв'язку, немає різниці між підпрограмами CubeHAL під час використання периферійного пристрою SPI в режимі підпорядкованого або головного. Єдина відмінність полягає в конфігурації периферійних пристроїв, відповідному встановленні параметра Mode структури SPI_InitTypeDef.

CubeHAL забезпечує три способи зв'язку через шину SPI: опитування, переривання та режим DMA.

Щоб надіслати кількість байтів на підпорядкований пристрій у режимі опитування, використовується функція:

```
HAL_StatusTypeDef HAL_SPI_Transmit(SPI_HandleTypeDef *hspi, uint8_t *pData, uint16_t Size, uint32_t Timeout);
```

Структура функції майже ідентична іншим процедурам зв'язку, (наприклад, тим, які використовуються для роботи з UART). Цю функцію можна використовувати, якщо периферійний пристрій SPI налаштовано на роботу в режимах SPI_DIRECTION_1LINE або SPI_- DIRECTION_2LINES. Щоб отримати байти у режимі опитування, використовується функція:

```
HAL_StatusTypeDef HAL_SPI_Receive(SPI_HandleTypeDef *hspi, uint8_t *pData, uint16_t Size, uint32_t Timeout);
```

Цю функцію можна використовувати у всіх трьох режимах.

Якщо підпорядкований пристрій підтримує повнодуплексний режим, використовується функція:

```
HAL_StatusTypeDef HAL_SPI_TransmitReceive(SPI_HandleTypeDef *hspi, uint8_t *pTxData, uint8_t *pRxData, uint16_t Size, uint32_t Timeout);
```

Функція дозволяє передавати задану кількість байтів, одночасно одержуючи ту саму кількість.

Для обміну даними через SPI в режимі переривань CubeHAL надає такі функції:

```
HAL_StatusTypeDef HAL_SPI_Transmit_IT(SPI_HandleTypeDef *hspi, uint8_t *pData, uint16_t Size);
```

```
HAL_StatusTypeDef HAL_SPI_Receive_IT(SPI_HandleTypeDef *hspi, uint8_t *pData, uint16_t Size);
```

```
HAL_StatusTypeDef HAL_SPI_TransmitReceive_IT(SPI_HandleTypeDef *hspi, uint8_t *pTxData, uint8_t *pRxData, uint16_t Size);
```

Процедура CubeHAL для обміну даними через SPI в режимі DMA ідентична трьом попереднім, за винятком того факту, що вони закінчуються на _DMA. Використовуючи підпрограми на основі переривань і DMA, необхідно очікувати повідомлення про завершення передачі, оскільки вони виконуються асинхронно. Це означає, що має бути включене відповідне переривання на рівні NVIC і задіяна функція `HAL_SPI_IRQHandler()` з ISR. Існує шість різних зворотних викликів, які можна реалізувати, як зазначено в таблиці 8.3.

Таблиця 8.3. Доступні зворотні виклики CubeHAL, коли периферійний пристрій SPI працює в режимі переривань або DMA

Callback	**Опис**
`HAL_SPI_TxCpltCallback()`	Задану кількість байтів передано
`HAL_SPI_RxCpltCallback()`	Задану кількість байтів прийнято
`HAL_SPI_TxRxCpltCallback()`	Задану кількість байтів передано і прийнято
`HAL_SPI_TxHalfCpltCallback()`	Сигналізує про те, що процес передачі половини вмісту буфера DMA SPI завершено
`HAL_SPI_RxHalfCpltCallback()`	Сигналізує про те, що процес прийому половини вмісту буфера DMA SPI завершено
`HAL_SPI_TxRxHalfCpltCallback()`	Сигналізує про те, що процес передачі та прийомуполовини вмісту буфера DMA SPI завершено

8.5 Програмування SPI в STM32CubeMX та IAR EW

Мікроконтролери серії STM32F103 мають два інтерфейси SPI. Вікно налаштувань інтерфейсу SPI в STM32CubeMX наведено на рисунку 8.5. На закладці *Pinout & Configuration* на панелі *Categories* в категорії System Core зі списку потрібно вибрати потрібний інтерфейсу SPI (для прикладу SPI 1). На панелі *SPI Mode & Configuration* у вікні *Mode* можна вибрати потрібний режим зв'язку, наприклад, *Full-Duplex Master,* та поведінку сигналу NSS. Після встановлення цих двох параметрів можна продовжити налаштування інших параметрів SPI у перегляді конфігурації CubeMX.

Нижче наведено приклад коду для роботи з акселерометром ADXL345. Це 3-х вісний цифровий датчик, що вимірює проекції прискорення на три просторові осі (x, y, z). Знаючи ці виміри і з огляду на величину вільного

падіння, можна визначити орієнтацію самого акселерометра у просторі. Цифрові результати вимірювання подаються у вигляді 16-розрядних чисел у додатковому коді. В даному прикладі наведено функції запису та зчитування даних. Сигнал NSS генерується програмно.

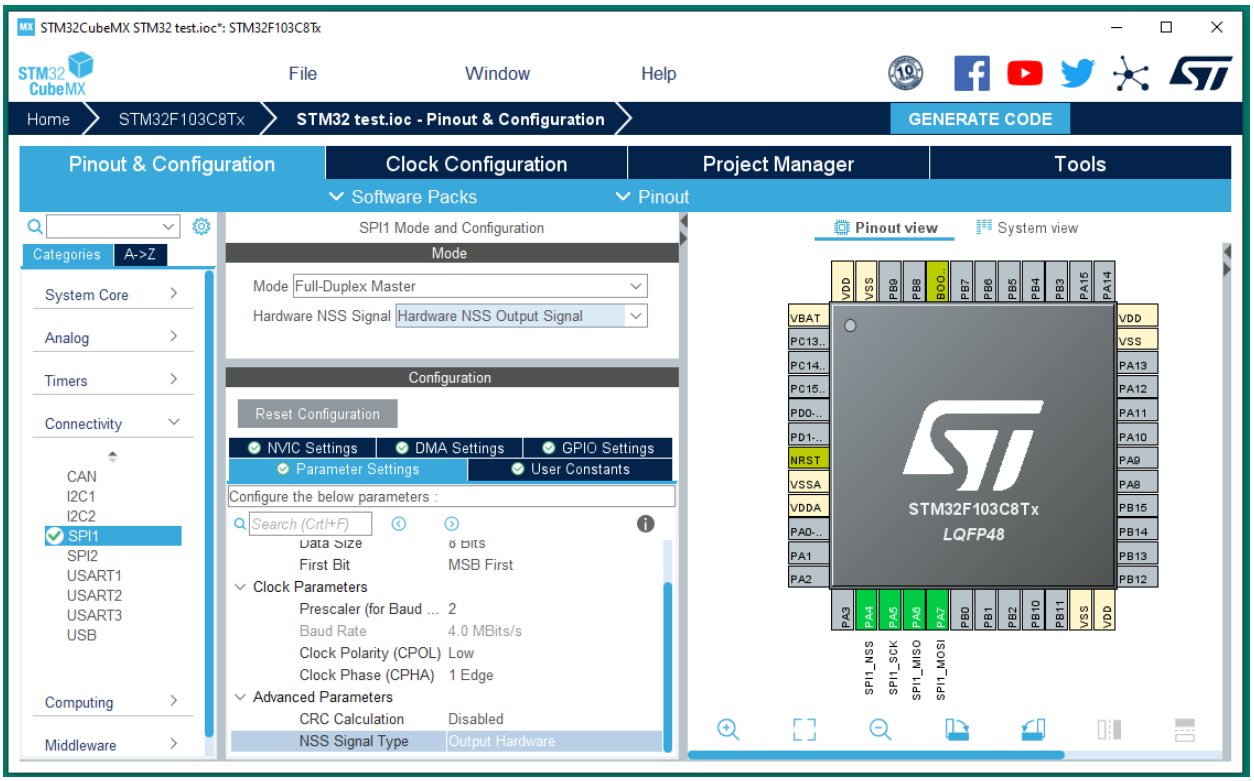

Рисунок 8.5. Налаштування інтерфейсу I²C в STM32CubeMX

Функція запису має наступний вигляд:

```
void adxl_write (uint8_t address, uint8_t value)
{
	uint8_t data[2];
	data[0] = address|0x40;  // multibyte write
	data[1] = value;

	// pull the cs pin low
	HAL_GPIO_WritePin (GPIOB, GPIO_PIN_6, GPIO_PIN_RESET);
```

```
    // write data to register
    HAL_SPI_Transmit (&hspi1, data, 2, 100);
    // pull the cs pin high
    HAL_GPIO_WritePin (GPIOB, GPIO_PIN_6, GPIO_PIN_SET);
}
```

Функція зчитування даних наступна:

```
void adxl_read (uint8_t address)
{
    address |= 0x80;  // read operation
    address |= 0x40;  // multibyte read
    uint8_t rec;

    // pull the pin low
    HAL_GPIO_WritePin (GPIOB, GPIO_PIN_6, GPIO_PIN_RESET);
    // send address
    HAL_SPI_Transmit (&hspi1, &address, 1, 100);
    // receive 6 bytes data
    HAL_SPI_Receive (&hspi1, data_rec, 6, 100);
    // pull the pin high
    HAL_GPIO_WritePin (GPIOB, GPIO_PIN_6, GPIO_PIN_SET);
}
```

Більш докладну інформацію про роботу з датчиком ADXL345 можна прочитати в [8].

Контрольні запитання

1. Призначення та характеристики інтерфейсу SPI.
2. Призначення та порядок роботи сигналів шини SPI.
3. Опишіть часову діаграму відповідно до налаштувань CPOL і CPHA.
4. Які способи зв’язку через шину SPI забезпечує CubeHAL?

СПИСОК ЛІТЕРАТУРИ

1. Carmine Noviello Mastering STM32: eBook. Leanpub, 2018. 852 pages.
2. STM32F103C8 - Mainstream Performance line, Arm Cortex-M3 MCU with 64 Kbytes of Flash memory, 72 MHz CPU, motor control, USB and CAN. st.com: веб-сайт. URL: https://www.st.com/en/microcontrollers-microprocessors/stm32f103c8.html (дата звернення: 10.05.2022).
3. STM32CubeMX - STM32Cube initialization code generator st.com: веб-сайт. URL: https://www.st.com/en/development-tools/stm32cubemx.html (дата звернення: 08.06.2021).
4. IAR Embedded Workbench for Arm IAR Systems: веб-сайт. URL: https://www.iar.com/products/architectures/arm/iar-embedded-workbench-for-arm/ (дата звернення: 10.05.2022).
5. Getting started with STM32 st.com: веб-сайт. URL: https://wiki.st.com/stm32mcu/wiki/STM32StepByStep:STM32_step_by_step_overview (дата звернення: 10.05.2022).
6. STM32-base project: веб-сайт. URL: https://stm32-base.org/guides/getting-started.html (дата звернення: 10.05.2022).
7. STM32 microcontroller GPIO configuration for hardware. st.com: веб-сайт. URL: an4899-stm32-microcontroller-gpio-configuration-for-hardware-settings-and-lowpower-consumption.pdf (дата звернення: 10.05.2022).
8. ADXL345 Digital Accelerometer. analog.com: веб-сайт. URL: https://www.analog.com/media/en/technical-documentation/data-sheets/ADXL345.pdf (дата звернення: 10.05.2022).